National Research University of Information Technologies,
Mechahics and Optics

Computer Technologies Department

Michael Lukin, Sergey Velder,
Anatoly Shalyto, Bulat Yaminov

Verification of automata-based programs

St.Petersburg

2011

Contents

Preface

Validation concept

In everyday life information technology are increasingly used both directly (by using computers and internet) and indirectly (by using TV, microwaves, mobile phones, cars, public transport etc). In 1995 it was estimated that a person interacts daily with 25 devices processing information. We also know that 20% of the cost of developing vehicles, trains and planes account for computer components. Due to the high integration of information technologies to all applications we have to increasingly rely on the reliability of software and hardware. It is natural to assume that there must not be situations when the phone is faulty or when the VCR is unpredictable and incorrectly responds to commands sent from the control panel. Nevertheless, these errors are in a certain sense insignificant. However, errors in systems that are critical for safety, such as, for example, nuclear power or flight control systems are not acceptable. The main problem for such systems is that their complexity is rapidly increasing and, consequently, the number of possible errors.

Even if we ignore the aspects of security, errors can still be very expensive, especially if they occur after the product is released to the market. There are several striking examples of such adverse effects. For example, the error in the team division of floating point numbers in the Intel *Pentium* caused damage of about $ 500 million. Major damage was caused by the crash of *Ariane-5* rocket, which probably occurred due to an error in the flight control program.

Therefore, validation of the systems (the process of validation of specifications, design and product) is an activity increasingly important. Validation is a way to maintain the quality control systems.

Of particular importance is ensuring the quality of software. Modern programming practice shows that the systems are checked by people (expert analysis) and dynamic testing. Support of these processes, even by relatively simple tools, not to mention the methods and tools with the serious mathematical base, is currently imperfect.

In view of the increasing size and complexity of systems, it is important to ensure the validation process systems using techniques and tools that facilitate the automatic analysis of correctness, as, for example, manual verification can be as inaccurate as the program itself.

Methods of validation systems

The most important methods of validation are expert analysis, testing, simulation, formal verification and validation of models. Will focus on two of them.

Testing is an effective way to verify that a given implementation of a system fit to the abstract specification. Testing can only be used after the implementation of the prototype system.

Formal verification, as opposed to testing is based on a mathematical proof of the correctness of programs.

Both of these methods can be supported and partially supported by the tools. For example, in testing there is a growing interest in developing algorithms and software for automatic generation and selection of tests on a formal specification of the system. The basis for formal verification is a program for automated theorem proof and test evidence, but even their use usually requires highly skilled users.

Model checking

The book covers a different validation method: model checking. This is an automated method which for a given model of system behavior with a finite number of states and logical properties, recorded in a suitable logical formalism (usually in the temporal logic), checks the validity of this property in this model. Model checking can be used to verify both hardware and software. The success of a number of projects using this method of verification increases the interest in it. For example, Intel has launched several research laboratories verification of new chips. An interesting aspect of model checking is that it supports partial verification: the system can be tested by a partial specification when considering only a subset of all claims.

The subject of the book review

The book is dedicated to the concepts, algorithms and tools for model checking software.

Ideas of testing models have such mathematical foundation as logic, automata theory, data structures, algorithms on graphs.

The book is structured as follows. The first chapter is about general issues of validation of software systems. We consider the formulation of the problem of validation and define the role of validation in software. Also different types of validation are distinguished, the basic concepts and definitions for each of them are given.

The second chapter outlines the theoretical issues of model checking, we introduce the different formalisms of temporal logics and model checking algorithms are described for the specifications expressed in these temporal logics. There are two types of models: *Kripke* models and *Petri* nets. Also some mathematical results concerning the properties of these models are presented.

Three types of test models are described: for linear and branching temporal logics and temporal logic for real-time. The first two capture the functional and qualitative aspects of the system, and the third is also to conduct a quantitative analysis.

The third chapter demonstrates the model-checking algorithms on examples of specific tools. A short description of the syntax of these tools and examples of the implementation of communication algorithms are given. Also there are results of testing models of these algorithms.

The presentation of general issues validation, the mathematical formalism and examples of verification models in first three chapters are according to a course of lectures of P. Cato [1].

The fourth chapter is devoted to the verification of program models that are based on automata approach [2]. The main feature of these programs is that the model behavior of the program, presented as a system of control automata state for which code is generated, is built by the developer already at the design stage, rather than on the final program, as proposed to do with the traditional method of using model checking. These programs come in line with other engineering developments (such as airplanes and cars), for which the models are first created, and then based on them goods are manufactured, and not vice versa.

This edition is a translation of the main sections, made by M. Lukin, a monograph by the same authors with the same name, which is published by the publishing house “Nauka”, Saint-Petersburg in 2011.

Chapter 1. Validation of systems

1.1. Validation of systems tasks

Introduction and motivation

The systems which are connected with the information processing more and more often provide critical services. They are used in a some conditions when failure could lead to the fatal consequences. Information processing widely used in such systems as nuclear power plant or chemical reactors. In these systems the errors are very dangerous. Another example of widely-distributed systems which are critical to safety is irradiative medical gadget.

So we could say that *robustness and safety of information processing is a key factor in these systems*.

How can we ensure safety of program systems? Usually program development begins from analisys of the acceptance criteria. After passing of several development phases the prototype is come out. Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. In a diagram form the strategy of validation strategy is represented in fig. 1.1 [1].

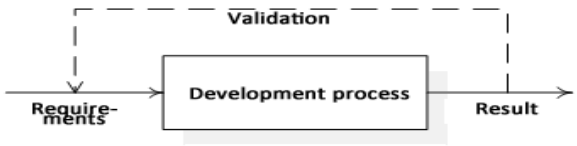


Fig. 1.1. Schematic view of posterior validation

What is the general practice solution to the question when to validate? Clearly that the error checking only in the end of the development is not acceptable: if an error is found, required a lot of effort to fix it, because the whole development process must be passed again, to see where and how the error could occur. Such operations are usually expensive. Therefore, it is advisable to validate “on the fly” – in the process of developing system, as it significantly reduces the cost of development. Schematically, this process is shown in Fig. 1.2.

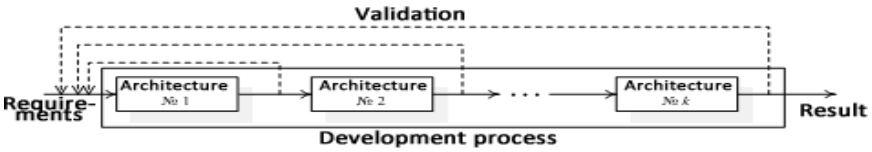


Fig. 1.2. Schematic view of “on-the-fly” validation

In practice, in order to ensure that the final result actually does what was prescribed there are two methods: expert analysis and testing. Researching on development programs show that 80% of all projects use expert analysis. Expert analysis is a fully manual activity in which a prototype or its part is researched by a team of experts who were not involved in the development of the system components. Testing is an important method of validation to validate the received implementations. However, tests are usually generated manually, and there is little attention given to instrumental support.

Validation of system is an important step in their development: most of the projects spent more time and effort on validation than on development. Mistakes can be very expensive. For example, the error in the Denver baggage handling system delayed the opening of a new airport to nine months (expense – $ 1.1 million per day).

Validation of the system as a part of development process

Interviews with specialists of a number of companies engaged in software development, has once again shown that the validation of systems must be integrated into the development process at an early stage. This is true, in particular, from an economic point of view, but now most of the errors are found during testing, where software modules are already composed and the whole system is studied.

Formal methods

For quality assurance of complex software systems it is required the use of modern methods and tools that support the validation process. The currently applied methods of validation are highly specialized and based on specifications formulated in natural language. Therefore, this book examines approaches to validation based on formal methods. When using such techniques design systems should be defined in terms exact and unambiguous specifications that provide the basis for systematic analysis. List the basic approaches to validation:

1. *Simulation.*
2. *Testing*.

3. *Formal verification*.
4. *Model checking (verification on model).*

Later we will briefly discuss the first three approaches, but the main theme, as noted above, is model checking.

Reactive Systems

This book focuses on the validation of reactive systems. *Reactive systems* are characterized by continuous interaction with the environment. They take the inputs from its environment and usually with little delay, react to them. Typical examples of this class of systems are operating systems, aircraft control systems, vehicles and processes, communication protocols, etc. For example, the control of chemical processes on a regular basis takes the control signals such as temperature and pressure at different points of the process. Based on this information, the program may decide to include a heating element, turn off the pump, etc. If you have a dangerous situation, for example, the pressure in the tank outside certain limits, the control program must perform certain actions. Typically, such reactive systems are quite complex.

As noted above, the correctness of reactive systems is crucial. To build a working rocket system correctly requires a clear methodology, which distinguishes the following phases.

1. On the basis of an exhaustive analysis of the requirements its specification should be formed.

2. Conceptual design gives an abstract specification of the project. This specification can be checked for consistency and compliance. This validation process can be supported by formal verification, simulation and model testing – such processes, in which the model of an abstract design specification (for example, a system of state machines) can be exhaustively tested.

3. When a credible specification is obtained, move to a system that implements an abstract specification. In this testing is a useful method of checking the implementation for compliance original requirements.

1.2. Simulation

Simulation is based on a model that describes the possible behavior of the system. This model is feasible in a certain sense, while the software tool (called a *simulator*) can determine the system behavior in relation to some scenarios. In this way the user gets a certain understanding of how the system responds to stimulus. Scripts can be supplied by the user or a tool such as, for example, a generator, which constructs random scenarios. Simulation is mainly useful for rapid, initial assessment of project quality. It is not well suited to find subtle bugs as simulation of all possible scenarios is impractical (and frequently impossible).

1.3. Testing

As noted above, a widely used traditional way of validation of project correctness is testing [6]. In testing system used in the form in which it is implemented (program, device, or combination of them). Certain values of the input data, called tests are served to its entrance, and the response of the system is studied. After it we check whether the response of the system corresponds to the required output. Principles of testing are almost the same as in the simulation. Their major difference lies in the fact that testing is done on the current system implementation, and simulation – on a model of system.

Testing is a validation method that is widely used in practice, but almost always performed on the basis of informal and heuristic methods. As testing is based on the consideration of only a small subset of possible examples of system behavior, it can never be complete. E. Dijkstra pointed out that testing can only show the presence of bugs but not their absence.

Testing, however, can complement the formal verification and validation of models that run on a mathematical model of the system rather than actual system. As the testing is used to realization it is useful mainly in the following cases:

- when the correct model of system is difficult to build;
- when parts of the system can not be formally modeled (for example, physical devices);
- when the model is proprietary.

Generally, testing is the dominant method of validation systems. It is applied to a number of tests that are normally received heuristically. In recent years,

however, an interest in the application of formal methods in testing is increasing. For example, in the field of communication protocols, this type of research has led to a draft international standard, "Formal methods in testing fitness". The testing process is divided into several phases:

1. Test generation. Abstract descriptions of the tests are generated systematically on the basis of exact and unequivocal set of properties required in the specification.
2. The choice of tests. The set of samples of abstract descriptions of the tests are chosen.
3. The implementation of tests. Abstract descriptions of the tests are transformed into executable tests.
4. Execution of tests. Executable tests are used to test the implementation by running them on a testing system. Considered and recorded the results of execution.
5. Test analysis. Logged results are analyzed to determine if they satisfy the expected results.
6. The different phases of testing can intersect and often intersect in practice, especially the last.

Testing is a method that can be used as prototypes in the form of systematic simulation, as well as to the final products. There are two known basic approaches: the white-box testing, when the internal structure of the implementation can be observed and sometimes partially controlled (stimulated), and black box testing. In the second case only communication between the test system and the environment can be verified, and the internal structure of the "box" is completely hidden from the tester. In practical circumstances, testing is somewhere between these two extremes, and sometimes is called as gray box testing.
List the main types of tests.

1. Unit tests. Test is written for a class, a separate method of this class, etc. Usually one script of using of class, method, or a module is checked.
2. Functional tests. Not an element of the code is testing but functionality. These tests make possible identifying structural errors.
3. Acceptance tests. Verified that the program does exactly what the customer wanted.
4. Stress Test. Operability testing program under heavy load.

5. Monkey Test. Program is served by the random input data, and its performance on such data and fault tolerance are tested. Vulnerabilities are also scanned for. Such tests are particularly important for server applications.
6. Parallel tests. Verified that the new version works just like the old one.
7. Regression tests. Written after the error message. The test repeats scenario in which the error occurred.

Test-driven development

One of the technologies of software development is test-driven development (TDD). Note that TDD is not a test technology. This process consists of short iterations of development of programs, each of which consists of the following steps:

1. Writing test.
2. Compilation of the test. The test should not be compiled. If the test is compiled, it means that you have created an entity that already exists.
3. Removing compilation errors.
4. Run the test. The test should fail.
5. Writing the code. Written as a simple code as only possible.
6. Run the test. This time the test should be passed.
7. Refactoring (if required).

The second and third steps are performed only after creating new modules or classes.

When the program uses the equipment which access is difficult to, mock objects are used or mocks. Mocks are automatically generated stubs. Mocks also can be used to top-down programming. To create a mock object frameworks, there are special (*RhinoMock*, *NMock*, *JMock*, etc.). Properties of development through testing:

- approach makes to develop the program from a user perspective;
- it makes writing classes, independent of each other;
- tests are also documentation to the code;
- it takes time to learn this approach in practice.

1.4. Formal verification

Complementary method to the simulation and testing is a rigorous proof that the system is working correctly. Such mathematical demonstration of the correctness of the system is called *formal verification* [11]. The basic idea is to construct a formal (mathematical) model of the system, which reflects (specifies) the possible behavior of the system. The requirements of correctness are written as a formal requirement specification, which reflects the desired behavior of the system. Based on these two specifications using formal proof we can verify whether the possible behavior is consistent with the desired. As there is a verification of a mathematical form, the notion of consistency can be precise and verification consists in proving the correctness towards this formal representation.

For formal verification are required:

- System model, typically containing a set of states, which keep information about the values of variables, software counters, etc., and the ratio of transitions, which describes how the system moves from one state to another.
- Specification method for expressing requirements in a formal way.
- The set of rules of evidence, allowing determination of whether the model meet the formulated requirements.

In order to more precisely understand what is meant, consider the method by which sequential programs can be formally verified.

Verification of sequential algorithms

This approach can be used to prove the correctness of sequential algorithms [1, 12], such as quick sort or calculation of the greatest common divisor of two integers. It begins with the formalization of the desired behavior using pre-and postconditions based on predicate logic formulas. The syntax of these formulas can be determined, for example, as follows:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi).$$

Here p is basic proposal (for example, “x equals 2”), the symbol “¬” is denial, and the symbol “∨” is disjunction. Other logical connectives can be defined as follows: $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, true = $\varphi \vee \neg\varphi$, false = ¬true and

$\varphi \rightarrow \psi = \neg\varphi \vee \psi$. For simplicity, we omit the existential and universal quantifiers.

A precondition describes the set of starting conditions studied (valid values of inputs), and post-condition – the set of desired final states (the required values of the outputs). When pre-and postconditions are formalized, the algorithm is coded in an abstract pseudo-code, and step by step, we prove that it meets specifications.

To build the proof we use the *formal system* which is a set of *inference rules*. These rules are usually associated with program building (algorithm). They are written as follows:

$$\{\varphi\}\ S\ \{\psi\}.$$

Here, φ is a precondition, S is a software operator and ψ is postcondition.

Triple {φ}, S, {ψ} is known as the triple Hoare and named in honor of one of the pioneers in the field of formal verification of computer programs. There are two interpretations of Hoare triples, depending on whether partial or total correctness is considered.

- The formula {φ} S {ψ} is called partially correct if each stopping calculation of S, starting in a state wherein φ is running, finish in the state in which ψ is running.
- Formula {φ} S {ψ} is called totally correct if every calculation of S, starting in a state in which φ is running, stop and finish in the state in which ψ is running.

Thus, in the case of partial correctness we make no assumptions about the calculation of S, which did not stop and hang up. In further explanation the partial correctness is considered, if not stated otherwise.

The main idea of the approach of Hoare is to prove the correctness of programs at the syntactic level, using only the triples of determined above forms. Deterministic sequential programs are designed according to the following grammar:

$$S ::= \textbf{skip} \mid x := E \mid S;\ S \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi} \mid \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}.$$

Here, skip is no operation, x: = *E* is assignment expression *E* to variable *x* (we assume that *x* and *E* have the same type), *S*; *S* is a composition of operators. The last two are alternative and iteration, respectively (*B* is Boolean expression).

The rules of inference should be read as follows: if all conditions located above the line are true, then the corollary under the dash is also true. For rules with a condition it is necessary to write only the result. These inference rules are called axioms. A formal system for sequential deterministic programs is shown in table 1.1.

Table 1.1. Formal system for partial correctness of sequential programs

Axiom for skip	$\{\varphi\}\,\mathbf{skip}\,\{\varphi\}$
Axiom for assignment	$\{\varphi[x:=k]\}\,x:=k\,\{\varphi\}$
Sequental composition	$\dfrac{\{\varphi\}\,S_1\,\{\chi\},\ \{\chi\}\,S_2\,\{\psi\}}{\{\varphi\}\,S_1;S_2\,\{\psi\}}$
Altenative	$\dfrac{\{\varphi\wedge B\}\,S_1\,\{\psi\},\ \{\varphi\wedge\neg B\}\,S_2\,\{\psi\}}{\{\varphi\}\,\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\,\{\psi\}}$
Iteration	$\dfrac{\{\varphi\wedge B\}\,S\,\{\varphi\}}{\{\varphi\}\,\mathbf{while}\ B\ \mathbf{do}\ S\,\{\varphi\wedge\neg B\}}$
Consequence	$\dfrac{\varphi\to\varphi_1,\ \{\varphi_1\}\,S\,\{\psi_1\},\ \psi_1\to\psi}{\{\varphi\}\,S\,\{\psi\}}$

Inference rule for the operator skip, which does nothing, is quite expected: in all conditions, if φ is true before the operator, it is also true after it. According to the axiom for the assignment, start with the postcondition φ and define the precondition by substituting φ [x: = k]. This substitution means formula φ, in which all occurrences x are replaced by k.

For example:

{k2 is even and k = y} x: = k {x2 is even and x = y}.

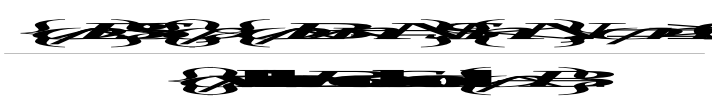
If the proof process begins with an analysis of post-conditions, it is usually used in series to parts of the program so that in the end we can prove the precondition for the program.

The rule of the sequential composition uses intermediate predicate χ, which characterizes the final state of the S_1 and the initial condition S_2. The rule of alternative uses a boolean expression B, whose value determines what exactly is executed: S_1 or S_2.

The rule for iteration requires explanation. It determines that predicate φ is performed after the end of "while B do S od", if the validity of φ can be maintained during the each execution of the loop body S. This explains why φ is *invariant*. One major difficulty in proving the correctness of programs using this approach consists in finding suitable invariants. In particular, this complicates the automation of such proofs.

All the rules are focused on such syntax that to each syntactic construction corresponds rule of inference. This is different from the rules of the investigation, which establishes a link between verification of programs and logic.

The rule of investigation allows to increase precondition and weaken postcondition. At the same time it facilitates the application of other rules. In particular, this rule allows you to replace the pre-and post-conditions to equivalent ones. However, it should be noted that the proof of the implication of the form $\varphi \Rightarrow \varphi'$ in general is undecidable. Now let us discuss the total correctness. Proof system in table 1.1. is not enough to prove that the sequential program stops. The only syntax, which can lead to crashes (not stopping) calculations is iteration. To prove the presence of the halt the inference rule for the iteration can be improved as follows:



Here, an auxiliary variable N is not included in φ, B, n or S. The meaning of this rule lies in the fact that N is the starting value of n, and at each iteration, the n value decreases, but remains non-negative. This construction eliminates the endless calculations because n can not decrease infinitely often without violating the condition $n \geq 0$. The variable n is called a variant.

Formal verification of parallel systems

Suppose that for the operators S_1 and S_2 structure $S_1 \| S_2$ means parallel composition of these operators. The main application for formal verification of parallel programs is the following inference rule:

This rule allows to verify the parallel systems in the same way as a consistent, considering the parts of a program separately. Due to the interaction between S_1 and S_2, at least through access to shared variables or message exchange, this rule is, unfortunately, is false in general. Many efforts have been made to obtain inference rules of a described form. There are several reasons why this is not easy to achieve.

The insertion of parallelism leads to the insertion of nondeterminism. This means that for parallel programs that communicate using shared variables, the behavior of the input-output depends on the order in which these shared variables are accessed.

For example, if S_1 is $x := x + 2$, S_2 is $x := x + 1; x := x + 1$ and S_3 is $x := 0$, value of x after $S_1 \| S_3$ can be 0 or 2, and value of x after $S_2 \| S_3$ can be 0, 1 or 2. Different values of x depend on the order executing statements in the S_1 and S_3 or S_3 and S_3. Moreover, despite the fact that the input-output behaviorof S_1 and S_2 is identical (increase x in 2), there is no guarantee that this will be true in a parallel context.

Parallel processes can potentially interact at any point of their performance, not just at the beginning of the calculation. If you want to make a conclusion about how parallel programs interact, it is not enough to know the properties of their starting and final states. It is also necessary to be able to form judgments about what happens during the computation. Thus, the properties must refer not only to the starting and final states, but also on the calculations themselves.

The main problem of the classical approach to the verification of concurrent and reactive systems is that, as explained above, that verification is fully focused on the idea of how the program computes the function from inputs to outputs. At the same time some valid inputs are given and desired outputs

are produced. For parallel systems calculation is usually not complete, and correctness depends on the behavior of the system over time, and not only from the final calculation result (if it ever ends). Global properties of parallel programs often can not be formulated in terms of relations between inputs and outputs.

Various attempts have been made to generalize the classical formal verification of concurrent programs. Due to the interaction between the components the rules of inference are usually quite complex, and full development of a formal system for parallel systems that can communicate using shared variables or by (synchronous or asynchronous) messaging becomes difficult. Therefore, for real systems, proving in this style is usually very large and complex. It is required to consider user interaction and control from it (in particular, in the form of finding suitable invariants). As a result, such proof is very cumbersome, resistant to bugs, and arrange them in an understandable manner is difficult.

Temporal logic

As noted above, the correctness of reactive systems is considered in relation to the behavior of systems during the time, not only to the relationship between inputs and outputs (pre-and post conditions) calculations, since the calculation of reactive systems is usually not complete. Consider, for example, the communication protocol between two agents (a sender and a receiver), which are connected by two-way communication channel. In this case, the property "If the process *P* sends a message he would not send the next message until confirmation will be received" can not be formulated in terms of pre-and postconditions.

In order to facilitate the formal specification of such properties, propositional logic should be extended by operators that refer to the behavior of the system over time. U (until) and G (globally) – are examples of statements that refer to a sequence of states (such as, for example, calculations). In this case φ U ψ means that the property φ is performed in all states until you reach a state in which the ψ is performed, and G φ means that always, in all future states φ is performed. Using these operators, we can formalize the above-described property of protocol, for example, as follows:

$$\mathbf{G}\,[snd_P(m) \rightarrow \neg snd_P(nxt(m))\ \mathbf{U}\ rcv_P(ack)].$$

In other words, if a message m is sent by process P, then this process does not transmit the following message nxt (m), until it receives confirmation.

Logics, advanced by operators, which allow to express properties of computation (in particular, that can express properties of mutual order between events), are called temporal logic. These logic in computer science were introduced by Pnueli [15, 16]. Temporal logic is a widely used method of specification for expressing properties of computing of reactive systems at a rather high level of abstraction.

In the same way as in the verification of sequential programs, we can construct inference rules for temporal logic (for reactive systems), and prove the correctness of these systems by the same approach as was shown for sequential programs using predicates. Disadvantages of the method of verification of proof, which requires a great deal of human labor, the same as checking for parallel systems: proof is bulky, cumbersome and requires a high level of user control.

This book show a different type of formal verification method, which is based on temporal logic, but, in general, requires less user involvement in the verification process. It is called *model checking*.

1.5. Model checking

The main idea of the method, called model checking [1, 17, 18], is to run the algorithms executed by the computer to check the correctness of systems. Using this approach, the user enters a description of the model system (possible behavior) and a description of the specification requirements (desired behavior), and machine holds the verification. If an error is found, the tool (verifier) provides a counterexample showing that under what circumstances the error can be detected. A counterexample is a scenario in which the model behaves in an undesirable way. He indicates that the model is not valid and should be corrected. This allows the user to detect the error and correct model specification before continuing verification. If no errors are found, the verification can be completed or the user can specify the model (taking into account more design decisions, so that the model becomes more concrete and realistic) and repeat the verification process. The scheme of model checking is shown in Fig. 1.3.

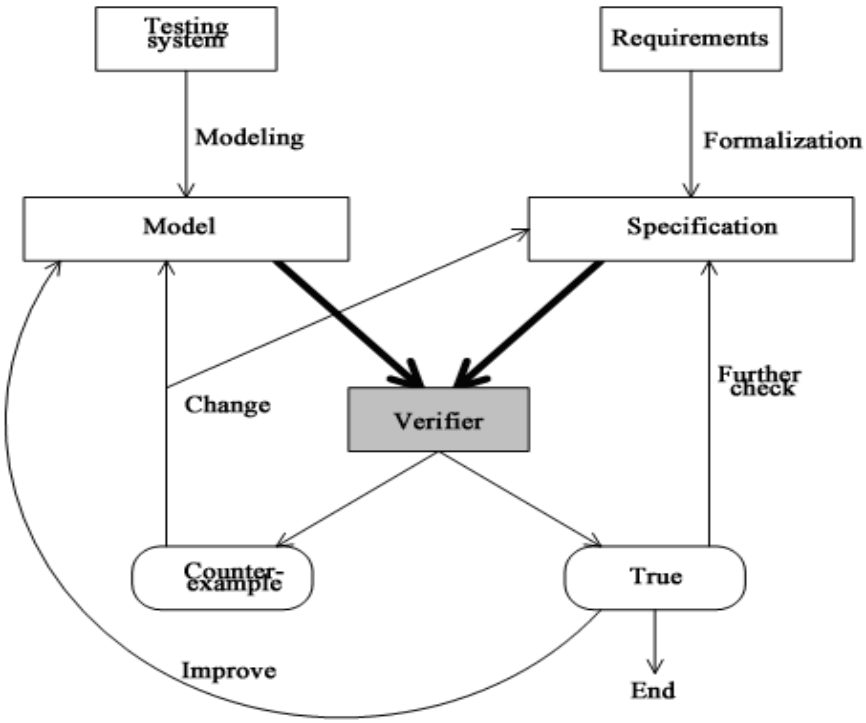


Fig. 1.3. Model checking

Algorithms for testing models are usually based on an exhaustive review of the set of all states of the model system: for each state of the system is checked, "Does it behave correctly" – satisfies a required property. In the simplest form, this method is known as attainability analysis. For example, suppose it is required to find out whether the system can reach a state in which the computation can not continue (so-called blocking). In this case, is sufficient to determine all the attainable states and determine whether there is an attainable state in which the computation is blocked. Attainability analysis is applicable only to prove the absence of locks and invariant properties that are performed during the whole computation. This is not enough, for example, for communication protocols, for which one of the important properties is the following: if the message is sent, it must ever be received. These types of properties are not covered by standard test of feasibility.

Protocols are modeled by sets of finite state machines that communicate by asynchronous message exchange [19]. Starting from the initial state of the system, which is expressed in terms of states of interacting automata and message buffers, all the states of the system are determined, which can be achieved by exchanging messages. Model checking can be considered as the

successor to these early methods of review of all states for protocols. It allows you to check a wider class of properties and manages a set of states much more effective than earlier methods.

Methods of model checking

There are two known approaches of model checking. They differ in how to describe the desired behavior which is a specification of requirements.

Logical or mixed approach. In this case, the desired system behavior is described by the language of numerous properties in a suitable logic (temporal or modal). The system is usually modeled as a finite state machine, in which states reflect variables and control positions, and transitions indicate how system could change one state to another. The system is correct in relation to requirements if the given set of initial states fulfills these requirements.

Behavioral or uniform approach. In this case both desirable and possible behaviors are defined in the same notation (machine), and as a criterion for the correctness equivalence relations are used (or pre-orders). Equivalence relations usually fix representation of the form "behaves like", while the preorder relations indicate the submission of the form "behaves at least like". As there are different views about what it means for two processes "behave the same way" (or "to at least like"), it defines various notions of equivalence and preorder. One of the most well-known concepts of equivalence is a double modeling, in which the two machines model each other, if one machine can simulate every step of the other machine and vice versa. Frequent presentation of pre-order is the inclusion of languages. The automata A is included in the automata B, if all the words allowable by A, shall also be permitted by B. The system is considered correct if desired and possible behavior are equivalent (or ordered) in relation to studied equivalence (or preorder).

Despite the fact that both of these approaches are conceptually different, connection between them can be set as follows. Logic induces an equivalence relation on systems like this: two systems are equivalent if and only if they satisfy the same formulas. Using this concept, we can establish the relationship between logic and equivalence relations. For example, we know that two machines model each other if they satisfy the same formulas of the logic *CTL*, widely used in the process of model checking. The

relationship between the two approaches is clear now: if two models satisfy the same properties (this is checked using a logical approach), they are behaviorally equivalent (this can be checked using a behavioral approach). The backward path is more interesting, as is generally impractical to inspect all properties in a certain logic, but verification of equivalences, such as dual modeling can be done efficiently.

This book uses logical approach [20]. Since there is essentially verified that the description of the system is a model of temporal logic formulas, this logical approach is originally called model checking. An exhaustive review of the set of states guaranteed end because of the finiteness of the model.

The advantages of model-checking

- This is a common approach with applications to verification of hardware, software, communication protocols, multi-agent systems, embedded systems, etc.
- Approach supports partial verification: the project can be verified by a partial specification when considering only a subset of all claims. This approach provides a high efficiency because you can limit the validation test of the most important properties while ignoring the less important test, but computationally more expensive claims.
- Inlining test patterns into the design process does not require more time than the simulation and testing. In some cases the use of model-checking leads to a reduction of development time. In addition, the use of appropriate methods, model checking programs can work with rather large state spaces.
- Program to test the models could potentially be used regularly by specialists in systems development with the same ease with which used, for example, compilers, as model checking does not require a high degree of user interaction.
- Reliable mathematical basis: modeling, semantics, logic and automata theory, data structures, algorithms on graphs.

Limitations of model checking

The major limitations of model checking:

- It applies mainly to the management applications, in which the components interact with each other. It is less suited to data applications as well as in such applications infinite state spaces are usually introduced.
- When using the model checking only the model of the system is verified, not the system itself. The fact that the model has certain properties, does not guarantee that the final implementation will have the same properties (to verify the final implementation additional methods such as systematic testing are needed). Simply put, any validation using model checking is as good as the model system.
- Finding a suitable abstraction (such as a model system, and suitable properties in temporal logic) requires appropriate qualification (but less than evidence-based verification).
- Like any tool, software for model checking can be unreliable. However, as the basis for model checking are standard and well-known algorithms, the reliability of such programs is usually no big problem. In some cases, the correctness of the most difficult parts of the software model checking has been proved with the use of automatic theorem proving programs.
- Model checking does not allow to verify the generalizations. If, for example, the protocol is verified for one, two and three processes using model checking, it does not give any result for a different number of processes. Model checking is practical only for special cases. Model checking, however, can help to formulate the theorem with arbitrary parameters, which can later be proven using formal verification.

It is impossible to reach absolutely guaranteed correctness of systems of real-size. Despite these limitations, we can say that model checking greatly increases the level of confidence in systems.

As in model checking the basic idea is to describe the behavior of the system by finite automata, under certain conditions, the number of states can go beyond the size of available memory. This, in particular, may be for parallel and distributed systems in which there are many system states. The size of the set of states of such systems in the worst case is proportional to the product of the sizes of the sets of states of individual components. The problem of excessive increase in the number of states is called the combinatorial explosion problem [17]. As shown below, the use of

automata-based programming [2] allows you to look at this problem from another point of view.

1.6. Automatic theorem proving

Automatic theorem proving can be used effectively in areas where mathematical abstractions of tasks are available. For the case of validation systems specification and realization of system are regarded as formulas, for example, φ and ψ, written in a certain logic. The checking that the implementation satisfies the specification reduces to checking formulas $\psi \to \varphi$. This means that behavior of implementation that satisfies ψ, is a possible behavior of the system specification, and therefore satisfies φ. Note that the specification of the system can allow other behavior that is not realized. To prove $\psi \to \varphi$ programs of automatic theorem proving are used.

Verification of the proof is an area closely related to the proof of theorems. User can send proof of theorem to the program to check the evidence. Software responds whether that proof is true. Programs of verification of evidence do a simpler problem than programs of automatic theorem proof. Therefore, they can work with more complicated proofs.
In order to reduce the amount of search in proof theorems, it makes sense to interact with the user, who may be knowledgeable about the best strategy for constructing a proof. Usually such interactive systems help in searching for evidence by maintaining a list of actions that need to be done, and provide clues how yet not proved theorems can be proved. Moreover, every step of the proof is verified by the system. As a rule, for evidence it must be made a lot of small steps, and high level of user interaction is required. Usually people overlook small parts of the evidence ("trivial", "same"), while the program requires an explicit presence of these parts. The verification process using the automatic theorem proof software is slow and laborious. In addition, the using instruments typically require quite high qualification of users.

The logic used by programs of proof theorems and programs of test evidence, usually is a variant of first-order predicate logic. In this logic there is an unlimited set of variables, the set of functional and predicate symbols of specified ary. Ary indicates the number of arguments of the functional or predicate symbol. Term is a variable or a string of the form $f(t_1, ..., t_n)$,

where *f* is a function symbol of ary, n and t_i are terms. Constants can be considered as a function of ary 0. Predicate has the form *P* (*t1, ..., tn*), where P is a predicate symbol of ary n, and t_i – terms.

Propositions of first-order logic are predicates, logical combinations of proposal or proposals, provided with quantification of the existence or universality. In a typed logic, there are also many types, and each variable has a type (as a program variable x has type int). Each function symbol has a lot of argument types and result type, and each predicate symbol has many types of arguments, but not the type of result. For this reason, in this logic, quantifications are also typed.

Algorithmic components of programs of proof theorems are the methods of application inference rules and the consequences of receiving. Important approaches used by programs for this are a natural deduction (e.g., if φ1 and φ2 are true we can conclude that $\varphi_1 \wedge \varphi_2$ are also true), resolution and unification (a procedure that is used for comparing two terms with each other by providing all permutations of the variables in which terms are the same). In contrast to traditional model checking, theorem proof can work directly with infinite sets of states and check the validity of properties with arbitrary values of the parameters.

These methods are not sufficient to find the evidence of given theorem, if the evidence exists. The tool should have a strategy that says how to find evidence. This strategy may offer the using of inference rules from the end, starting with the proposal, which is required to prove. The strategies that people use to find evidence are not formalized. Strategies used by programs proofs of theorems are based on the traversal algorithm in width and in depth.

Finally, programs of proving theorems are not very useful in practice: the problem of proving of theorems is exponentially complex. Sentence of length *n* can have a proof of exponential size of *n*. Searching for this proof requires exponential time on its length. Therefore, in general, theorem proof is double exponentially in relation to the length of proof proposition. For interactive programs of theorem proving, this complexity is greatly reduced.

List the differences between theorem proving and model checking:

- Model checking is fully automatic and fast.

- Model checking can be applied to partial implementations. Therefore, it can provide useful information about correctness of the system even if the system is not fully defined.
- Model checking programs have user-friendly interface and are easy to use, while the use of software test evidence requires a rather high qualification of the users in order to guide and accompany the process of verification. In particular, it is difficult to introduce someone to the logical language of program proving of theorems, which is usually the expressive logic of higher order.
- Model checking is useful to management applications, such as reactive systems. Proving of the theorems is applicable to work with infinite sets of states, and therefore it can be used for data processing applications.
- In case of success proving of theorems gives (almost) the highest level of accuracy and reliability of the evidence.
- Model checking can generate counterexamples that can be used to assist in debugging.
- When using the model checking, the project is checked for a fixed (and finite) set of parameters. The using of programs of proving theorems is possible for arbitrary values of the parameters.

Model checking is not considered as a "best" approach compared with the proving of theorems. These methods complement each other because each of them has certain advantages. Attempts were made to integrate these methods in order to get the effect of combining the advantages of both approaches.

Section 2. The mathematical apparatus of model

2.1. Modeling

Model checking method does not work with program directly. It works with model of program. In this section is described how to create model.

At first the specification for the program is created. So long as model checking method works with model, the model must have all the properties which are described in the specification. However inessential details are not needed. For instance if in the specification of lift control system described

that the lift doors must be closed in time of moving, then speed of lift is inessential.

In this book the most interesting systems are reactive systems [17]. These systems can not be modeled in terms of input-output because they can work indefinitely long. So for these systems their time behavior is verified. For this the notion *state of the system* is introduced. State of the system is its instant description with fixed values of all variables in the system. Work of reactive system can be imagined as transitions between states.

Formally we can represent work of this system as *Kripke model* [17].

Firstly introduce the notion of *set of atomic propositions*. *Atomic proposition* is a proposition such that inner structure can not change. Atomic propositions are basic propositions. Set of atomic propositions is denoted by *AP*.

Examples of atomic propositions:

- "x > 0"
- "x = 5"

Kripke model over the set of atomic propositions *AP* is a triple (*S*, *R*, *Label*) such that:

- *S* is a nonvoid set of states;
- $R \subseteq S \times S$ is a *total transitions relation* on *S*, which associates an element $s \in S$ the set of its possible successors;
- *Label*: $S \to 2^{AP}$ associates each state $s \in S$ atomic propositions *Label*(*s*), that are true in *s*.

Relation $R \subseteq S \times S$ is *total*, if it associates each state $s \in S$ at least one successor ($\forall s \in S$: $\exists s' \in S$: $(s, s') \in R$).

2.2. Model checking for linear temporal logic

2.2.1. *LTL* Syntax

Following definition defines set of basic formulae which can be expressed in linear temporal logic (*LTL*).

Let *AP* is set of atomic propositions. Then:

1. *p* is formula for each $p \in AP$.

2. If φ is formula, then $\neg\varphi$ is formula.
3. If φ and ψ are formulae, then $\varphi \vee \psi$ is formula.
4. If φ is formula, then $\mathbf{X}\ \varphi$ is formula.
5. If φ and ψ are formulae, then $\varphi\ \mathbf{U}\ \psi$ is formula.

Set of formulae constructed using these rules is called *LTL formulae*.

Note: set of formulae that constructed using the three first rules defines set of all propositional logic formulae. Therefore propositional logic is a subset of *LTL*.

The *LTL* syntax can be defined in Backus-Naur form:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{X}\ \varphi \mid (\varphi\ \mathbf{U}\ \varphi).$$

Logical operators satisfy the following:

$$\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi),$$

$$\varphi \rightarrow \psi = \neg\varphi \vee \psi,$$

$$\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi).$$

true equals $\varphi \vee \neg\varphi$, *false* equals $\neg$*true*. Temporal operators **G** (Globally, "always") and **F** (Future, "some time in future") by definition put:

$$\mathbf{F}\ \varphi = \textit{true}\ \mathbf{U}\ \varphi,$$

$$\mathbf{G}\ \varphi = \neg\mathbf{F}\ \neg\varphi.$$

All the states satisfy *true* therefore **F** φ means that φ come true some time in future. **G** φ means that "it is wrong that some time in future $\neg\varphi$ come true". Therefore it means that φ always is true.

2.2.2. *LTL* Semantics

Shown above definition gives a method to construct *LTL* formulae but does not give an interpretation. Formally **X** φ means that φ is *true* in the next state, **F** φ means that φ will be *true* (now or in some moment in future). But what do the words "state", "next state" and "some moment in future" means? The formal interpretation (it usually called semantics) helps us

unambiguously determine these terms. The formal meaning of properties in temporal logic is defined in terms of *model*.

Definition. *LTL-model* is a triple $M = (S, R, Label)$ such that:

1. S is a nonvoid set of states;
2. $R: S \to S$ maps the state $s \in S$ the single next state $R(s)$ (next function).
3. $Label: S \to 2^{AP}$ maps each state $s \in S$ atomic propositions $Label(s)$, which are true in s.

Example. Let $AP = \{x = 0, x = 1, x \neq 0\}$ is a set of atomic propositions, $S = \{s_0, \ldots, s_3\}$ is a set of states, $R(s_i) = s_{i+1}$ for $0 \leq i < 3$ и $R(s_3) = s_3$ is a next function and $Label(s_0) = \{x \neq 0\}$, $Label(s_1) = Label(s_2) = \{x = 0\}$, $Label(s_3) = \{x = 1, x \neq 0\}$ is a label function. In the model $M = (S, R, Label)$ the atomic proposition "$x = 0$" is true only in states s_1 and s_2, the atomic proposition "$x \neq 0$" is true only in states s_0 and s_3, the atomic proposition "$x = 1$" is true only in state s_3.

The meaning of formulae in logic is defined in terms of *formula satisfiability relation*. Let $\models$ be formula satisfiability relation.

Formula satisfiability relation means that $M, s \models \varphi$ if and only if когда φ is true in the state s of the model M. When the model M is obvious from the context, we will write $s \models \varphi$ instead of $M, s \models \varphi$.

Define LTL semantics. Let $R^0(s) = s$, $R^{n+1}(s) = R(R^n(s))$ for each $n \geq 0$. Let $p \in AP$ is atomic proposition, $M = (S, R, Label)$ is *LTL* model, $s \in S$ and φ, ψ are *LTL* formulae. By definition, put

$$
\begin{aligned}
s &\models p && \Leftrightarrow p \in Label(s);\\
s &\models \neg\varphi && \Leftrightarrow \neg(s \models \varphi);\\
s &\models (\varphi \vee \psi) && \Leftrightarrow (s \models \varphi) \vee (s \models \psi);\\
s &\models \mathbf{X}\,\varphi && \Leftrightarrow R(s) \models \varphi;\\
s &\models (\varphi\ \mathbf{U}\ \psi) && \Leftrightarrow \exists\, j \geq 0\colon R^j(s) \models \psi \wedge (\forall 0 \leq k < j\colon R^k(s) \models \varphi).
\end{aligned}
$$

For instance, consider semantics of $\mathbf{F}\,\varphi$.

$s \models \mathbf{F}\,\varphi$

$\Leftrightarrow$ {by definition of **F**}

$s \models \text{true } \mathbf{U}\ \varphi$

$\Leftrightarrow$ {semantics **U**}

$\exists\, j \geq 0\colon R^j(s) \models \varphi \wedge (\forall\ 0 \leq k < j\colon R^k(s) \models \text{true})$

$\Leftrightarrow$ {simplify}

$\exists\, j \geq 0\colon R^j(s) \models \varphi.$

Therefore **F** φ is true in the state s if and only if there is exists one of next states such that φ is true in that state or φ is true in the state s.

Model checking and satisfiability

In the first section the informal definition of model checking problem was given. Now we can give the formal definition:

Suppose the finite model M, the state s and the formula φ. Is $M, s \models \varphi$ true?

The task of model checking should not be confused with the more traditional problem of the satisfiability in logic. *Satisfiability problem* can be formulated as follows:

Suppose the property φ. Do exist the model M and the state s such that $M, s \models \varphi$?

The satisfiability problem for *LTL* is solvable. Therefore the model checking problem is solvable too.

2.2.3. Verification of *LTL* using *Buchi* automata

There are several methods to represent *LTL* formula as transitional graph for model checking. One of this methods uses *Buchi automata*.

Let *AP* is a set of atomic propositions. The *Buchi automaton over the alphabet* 2^{AP} is the quadruple $A = (Q, q_0, \delta, F)$ such that

- Q is a finite set of states;
- q_0 is the start state;
- $\delta \subseteq Q \times 2^{AP} \times Q$ is a total transition relation;
- $F \subseteq Q$ is a set of accept states.

We describe the algorithm due to Gerth, Peled, Vardi and Wolper [16, 17] for constructing a *Buchi* automaton from *LTL* formula.

Let **R** (Release) be a temporal operator such that

$$\varphi \mathbf{R} \psi = \neg(\neg\varphi \mathbf{U} \neg\psi).$$

It satisfies the following identity:

$$\varphi \mathbf{R} \psi \equiv \psi \wedge (\varphi \vee \mathbf{X}(\varphi \mathbf{R} \psi)).$$

This algorithm requires *LTL* formula to be in negative normal form.

Conversion to negative normal form:

1. Change all sub formulae like **F** φ to true **U** φ.
2. Change all sub formulae like **G** φ to false **R** φ.
3. Using Boolean identities remove all logical operators except $\neg$, $\vee$, $\wedge$.
4. Sink all negations inside temporal operators using following identities:
    1. $\neg(\varphi \mathbf{U} \psi) \equiv \neg\varphi \mathbf{R} \neg\psi$,
    2. $\neg(\varphi \mathbf{R} \psi) \equiv \neg\varphi \mathbf{U} \neg\psi$,
    3. $\neg\mathbf{X} \varphi \equiv \mathbf{X} \neg\varphi$.

For algorithm we need following data structures:

- *UID* is unique identifier;
- *Formula* is LTL formula;
- *Node* is a vertex of transition graph of Buchi automaton.

The format of *UID* and *Formula* is not significant so describe a *Node* structure (listing 2.1).

Listing 2.1. Structure *Node*

```
struct Node
{
  UID id;
  list<NodeID> incoming;
  list<Formula> old;
  list<Formula> new;
  list<Formula> next;
};
```

Incoming is a list of predecessor vertices (edges lead from this nodes to considered node). In the fields: *old*, *new* and *next* there are lists of sub formulae of source formula.

The function *CreateAutomaton* (listing 2.2) creates the transition graph of Buchi automaton from formula *f*.

Listing 2.2. Function *CreateAutomaton*

```
list<Node> CreateAutomaton (Formula f)
{
  Node n;
  n.incoming = {init};
  n.old = ∅;
  n.new = {f};
  n.next = ∅;
  return expand(n, ∅);
}
```

Function Expand is shown in listing 2.3.

Listing 2.3. Function *Expand*

```
list<Node> Expand (Node currentNode, list<Node> nodes)
{
  if (currentNode.new == ∅)
  {
    if (∃ Node r ∈ nodes: r.old == currentNode.old
                      && r.next == currentNode.next)
    {
      r.incoming = r.incoming ∪ currentNode.incoming;
      return nodes;
    }
    else
    {
      Node newNode;
      newNode.incoming = {currentNode};
      newNode.old = newNode.next = ∅;
      newNode.new = currentNode.next;
      Expand(newNode, nodes ∪ {currentNode});
    }
  }
  else   // currentNode.new is not void.
  {
    Choose Formula n из currentNode.new;
    currentNode.new = currentNode.new \ {n};
    if (n ∈ currentNode.old) Expand(currentNode, nodes);
    else
    {
      if (n == false or !n ∈ currentNode.old) return
              nodes;
      if (n ∈ AP or !n ∈ AP or n == true)
      // node replacement.
      {
```

```
  node newNode;
  newNode.incoming = currentNode.incoming;
  newNode.old = currentNode.old ∪ {n};
  newNode.new = currentNode.new;
  newNode.next = currentNode.next;
  Expand(newNode, nodes);
}
if (n has the form f ∨ g)
// Replacement of node currentNode by node newNode.
{
  node newNode1, newNode2;
  newNode1.incoming = currentNode.incoming;
  newNode1.old = currentNode.old ∪ {n};
  newNode1.new = currentNode.new ∪ {f};
  newNode1.next = currentNode.next;
  newNode2.incoming = currentNode.incoming;
  newNode2.old = currentNode.old ∪ {n};
  newNode2.new = currentNode.new ∪ {g};
  newNode2.next = currentNode.next;
  Expand(newNode2, Expand(newNode1, nodes));
}

if (n has the form f U g)
// f U g ⇔ g ∨ (f ∧ X (f U g)).
// Splitting.
{
  node newNode1, newNode2;
  newNode1.incoming = currentNode.incoming;
  newNode1.old = currentNode.old ∪ {n};
  newNode1.new = currentNode.new ∪ {f};
  newNode1.next = currentNode.next ∪ {f U g};
  newNode2.incoming = currentNode.incoming;
  newNode2.old = currentNode.old ∪ {n};
  newNode2.new = currentNode.new ∪ {g};
  newNode2.next = currentNode.next;
  Expand(newNode2, expand(newNode1, nodes));
}
if (n has the form f R g)
// f R g ⇔ g ∧ (f ∨ X (f R g)).
// Splitting.
{
  node newNode1, newNode2;
  newNode1.incoming = currentNode.incoming;
  newNode1.old = currentNode.old ∪ {n};
  newNode1.new = currentNode.new ∪ {f};
  newNode1.next = currentNode.next;
  newNode2.incoming = currentNode.incoming;
```

```
            newNode2.old = currentNode.old ∪ {n};
            newNode2.new = currentNode.new ∪ {f, g};
            newNode2.next = currentNode.next ∪ {f R g};
            Expand(newNode2, expand(newNode1, nodes));
          }
          if (n has the form f ∧ g)
          // Replacement of node currentNode by node newNode.
          {
            node newNode;
            newNode.incoming = currentNode.incoming;
            newNode.old = currentNode.old ∪ {n};
            newNode.new = currentNode.new ∪ {f, g};
            newNode.next = currentNode.next;
            Expand(newNode, nodes);
          }
          if (n has form X f)
          // Replacement of node currentNode by node newNode.
          {
            node newNode;
            newNode.incoming = currentNode.incoming;
            newNode.old = currentNode.old ∪ {n};
            newNode.new = currentNode.new;
            newNode.next = currentNode.next ∪ {f};
            Expand(newNode, nodes);
          }
        }
      }
    }
```

Model checking using *Buchi* automaton

Let there be given *Kripke* model and *LTL* formula whose execution on the model is needed to be checked. The general idea of the algorithm is as follows:

- From the denial of *LTL* formula equivalent *Buchi* automaton is constructed.
- Kripke model is also converted into a *Buchi* automaton.
- Third *Buchi* automaton is constructed as the intersection of the first two. This state machine will allow the paths of the original model which does not satisfy the *LTL* formula.
- If a language allowed by the constructed automaton-intersection is empty then verification is successful. Otherwise the path allowed by the automaton-intersection is a counterexample.

First we construct a *Buchi* automaton corresponding to the verifying *LTL*n formula. Recall what it represents. *Buchi* automaton is a finite automaton over infinite words. *Buchi* automaton transitions are marked by predicates of the original formula *LTL*. Automaton works as follows. At each step it takes a regular set of values of the predicate of the sequence, reflecting the history of the program. Using these values the automaton computes marks on the transitions from the current state. If there are more than one active transition, then automaton nondeterministically chooses one of them. The example of Buchi automaton constructed from LTL formula is shown in fig. 2.1. Formula "**GF** *p*" means that "*p* will be true indefinitely many times".

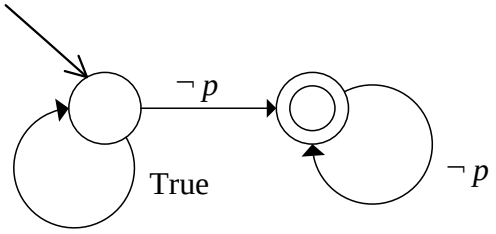


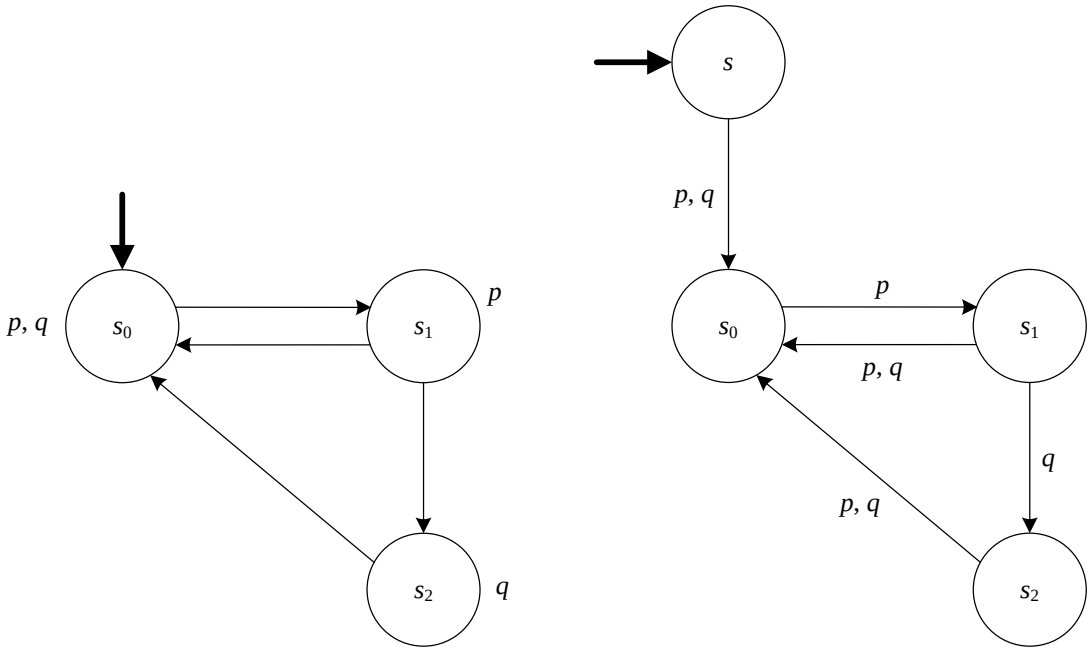
Fig. 2.1. The *Buchi* automaton for formula "**GF** *p*"

The next stage of verification is conversion *Kripke* model to *Buchi* automaton [17]. The example of *Kripke* model and corresponding *Buchi* automaton is shown in fig. 2.2.

After that the intersection of two these *Buchi* automata is constructed. Therefore it accepts only sequences of predicates that are accepted by the *Kripke* model and *Buchi* automaton. This intersection is a *Buchi* automaton. If the acceptance path is exists then:

- it is a possible scenario of the model;
- it violates verifying *LTL* formula.

Therefore it is a counter-example.



a b

Рис. 2.15. Example of *Kripke* model (a) and corresponding *Buchi* automaton (б)

2.3. Model checking for branching temporal logic

A. Pnueli introduced the temporal logic into computer science for specification and verification of reactive systems [15]. Above *LTL* was considered - an important representative of the temporal logic. This logic is called linear, because when it is used a qualitative notion of time is linear: at any given time, there is only one child state, and therefore only one future. Formally speaking, this follows from the fact that the interpretation of temporal logic formulas, using the satisfiability relation ⊨, defined in terms of a model in which the state s has exactly one child R (s). Thus, for each state s model generates unique infinite sequence of states s, R (s), R (R (s)), ... The sequence of states is the calculation. As the semantics of linear temporal logic is based on such "generating sequence" models, the temporal operators **X**, **U**, **F** and **G** in fact describe the sequence of events along the one time way (one computing system).

In the early 80s for specification and verification was proposed another type of temporal logic, which is based not on the linear but on the branching notion of time. This logic is formally based on models, in which at each

moment there may be several different possible futures. Given this notion of branching time temporal logic, this class is called branching temporal logic. Consequently, R (s) is a (nonempty) set of states, not one state, as in *LTL*.

Presentation of the semantics of branching temporal logic, therefore, is based on a tree of states instead of a sequence. Each path in this tree should present one possible computation. The tree itself represents all possible computations. More precisely, a tree, suspended in a state s, represents all possible infinite computations which start in state s.

Temporal operators in the branching temporal logic allow expressing properties (all or some) of the calculations in the system. For example, the property of **EF** φ means that there is a computation along which runs **F** φ. The essence of this property is that there is at least one possible computation, in which, ultimately status, performing φ is achieved. However, this does not exclude the fact that there may be computations, for which this property does not hold - calculations in which φ is never hold. The property **AF** φ for example, differs from this existential property of calculations the fact that it requires that all computations satisfy the property of **F** φ.

The existence of two types of temporal logic (linear and branching) led to the development of two "schools" of model checking. Despite the advantages and disadvantages of each of them, there are two reasons that justify consideration in this book both a model checking for linear and branching temporal logic:

- The expressive power of many linear and branching temporal logics is uncompared. This means that some of the properties expressible in linear temporal logic can not be expressed in some branching temporal logic and vice versa.
- Traditional methods used for efficient model checking for linear temporal logic are very different from the methods used for branching temporal logic. This leads, in particular, to quite different estimates of complexity.

This section focuses on model checking for branching temporal logic *CTL* (Computational Tree Logic). Importantly, it can be considered as an analogue of the branching *LTL*, for which it is possible an effective model checking.

Section outlines based on rate [1], and provides examples of [4, 26].

2.3.1. *CTL* syntax

Define syntax of *Computational tree logic* (*CTL*). Let $p \in AP$. By definition put Backus-Naur form

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \vee \varphi) \mid \mathbf{EX}\ \varphi \mid \mathbf{E}[\varphi\ \mathbf{U}\ \varphi] \mid \mathbf{A}[\varphi\ \mathbf{U}\ \varphi].$$

The following atomic operators are used:

- EX (in the next state for some path);
- E (for certain path);
- A (for all paths);
- U (until).

X and **U** are linear temporal operators. They express properties on a fixed path. The quantifier **E** expresses a property on a certain path. The quantifier **A** expresses a property on all paths. The quantifiers **E** and **A** can be used only in combination with **X** or **U**. The operator **AX** is not atomic and is defined below. Boolean operators true, false, $\wedge$, $\rightarrow$ and $\leftrightarrow$ have standard definitions.

From **F** φ = true **U** φ follows:

EF φ = **E**[true **U** φ];

AF φ = **A**[true **U** φ].

EF φ means "φ comes true potentially". **AF** φ means "φ is inevitable".

2.3.2. *CTL* semantics

As marked above, the interpretation of LTL is defined in terms of sequence of states. *CTL* refers to many computational paths. So to adequately represent branching the concept of sequence was changed to concept of *tree*. *CTL* model is a model that generates tree. Like model *CTL model* is a triple $M = (S, R, Label)$. The only difference is that R is a *total relation* instead of total function.

Example. Let $AP = \{x = 0, x = 1, x \neq 0\}$ is a set of atomic propositions, $S = \{s_0, \ldots, s_3\}$ is a set of states with following marks:

$Label(s_0) = \{x \neq 0\}$,

$Label(s_1) = Label(s_2) = \{x = 0\}$,

$Label(s_3) = \{x = 1, x \neq 0\}$,

transition relation R is following:

$R = \{(s_0, s_1), (s_1, s_2), (s_1, s_3), (s_3, s_3), (s_2, s_3), (s_3, s_2)\}$.

Consider CTL model $M = (S, R, Label)$. It is shown in fig. 2.16 (a).

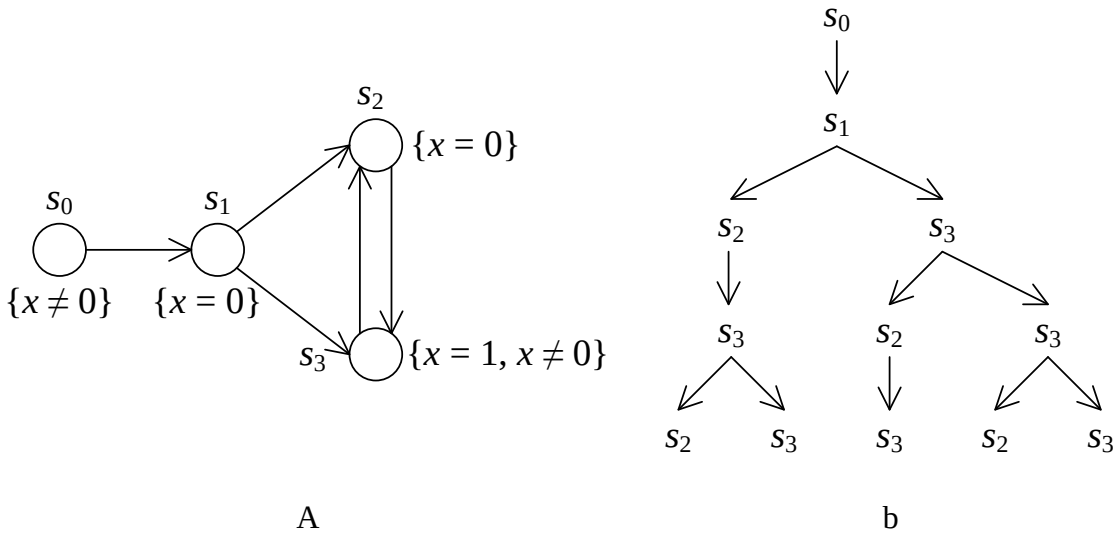


Рис. 2.17. Example of *CTL*-model (a)
and a prefix of one of it's computational trees (b)

Example. Consider *CTL* model at fig. 2.16 (a). Finite prefix of infinite tree is shown in fig. 2.16 (b). Examples of paths are: $s_0 s_1 s_2 s_3^{\omega}$, $s_0 s_1 (s_2 s_3)^{\omega}$ and $s_0 s_1 (s_3 s_2)^{*} s_3^{\omega}$.

Semantics of *CTL* is also defined in terms of formula satisfiability relation ($\models$).

Let $p \in AP$ is atomic proposition, $M = (S, R, Label)$ is *CTL* model, $s \in S$ and φ, ψ – *CTL* formulae. By definition, put:

$s \models p \quad \Leftrightarrow p \in Label(s)$;

$s \models \neg\varphi \quad \Leftrightarrow \neg(s \models \varphi)$;

$s \models (\varphi \vee \psi) \quad \Leftrightarrow (s \models \varphi) \vee (s \models \psi)$;

$s \models \mathbf{EX}\ \varphi \quad \Leftrightarrow \exists\sigma \in P_M(s)\colon \sigma[1] \models \varphi$;

$s \models \mathbf{E}[\varphi\ \mathbf{U}\ \psi] \quad \Leftrightarrow \exists\sigma \in P_M(s)\colon (\exists j \geq 0\colon \sigma[j] \models \psi \wedge$

$$\wedge\ (\forall 0 \le k < j\colon \sigma[k] \models \varphi));$$

$$s \models \mathbf{A}[\varphi\ \mathbf{U}\ \psi] \quad \Leftrightarrow \forall \sigma \in P_M(s)\colon (\exists j \ge 0\colon \sigma[j] \models \psi \wedge$$

$$\wedge\ (\forall 0 \le k < j\colon \sigma[k] \models \varphi)).$$

The Interpretation of temporal operators **AX** φ, **EF** φ, **EG** φ, **AF** φ and **AG** φ can be produced from the definition.

Section 3. Verificators overview

3.1. *SPIN*

SPIN [46] supports model checking and design of asynchronous process systems. There are four kinds of process interaction:

- *rendezvous* channels [46];
- buffered channels;
- shared variables;
- combined method.

SPIN supports an interactive and random simulation. In the interactive simulation non-deterministic choices are made by user. In the random simulation non-deterministic choices are made randomly.

SPIN accepts design specifications written in the verification language *PROMELA* (a *PROcess Meta Language*). The correctness claims must be specified in the synax of standard *Linear Temporal Logic* (*LTL*).

The basic structure of the *SPIN* model checker isillustrated in Fig. 3.1. The verification algorithms are based on convertation from *LTL* formula to the *Buchi automaton*.

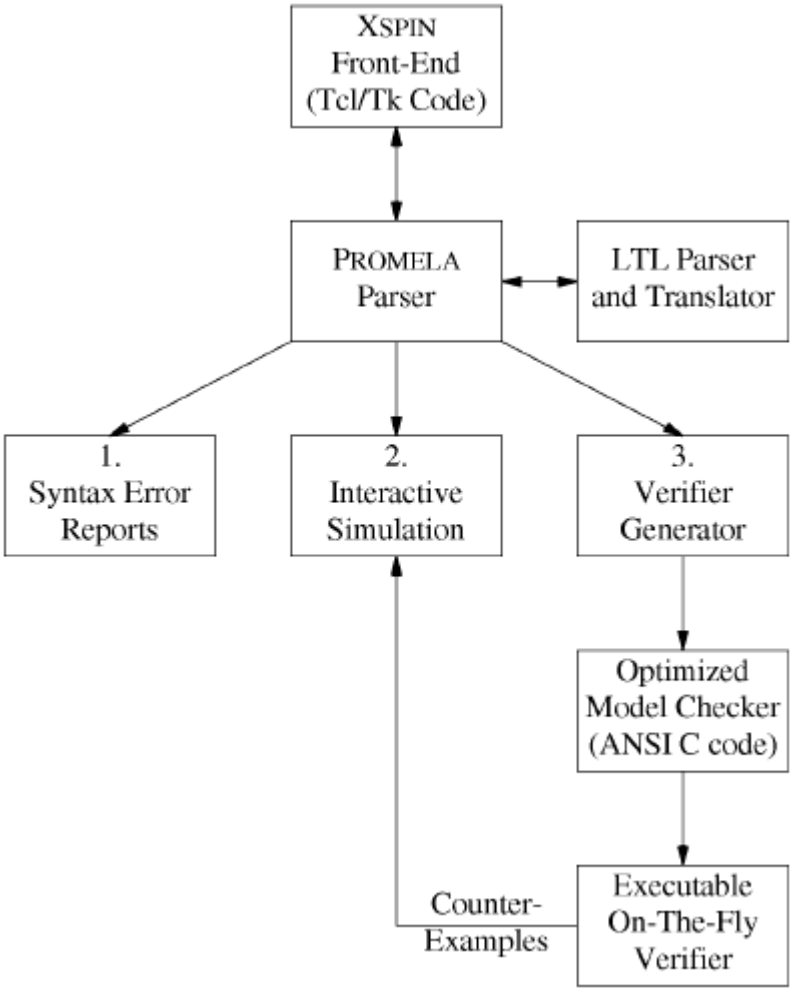


Fig. 3.1. The structure of *SPIN* simulation and verification [46]

The syntax of *PROMELA* like the syntax of *C* language. The *PROMELA* model consists of the following components:

- type declaration (user-defined data-types);
- channel declaration;
- variable declaration;
- process declaration;
- the `init` process.

User-defined data types can be introduced by `typedef` declaration. Typedef declarations are like structures in *C*:

Listing 3.1. Typedef

```
typedef myType
{
  int i;
  byte arr[5]
};
```

User-defined data types can be used as elements of other user-defined data types:

Listing 3.2. Typedef

```
typedef myOtherType
{
  myType t;
  short s
};
```

PROMELA has five built-in variable types:

bit [0…1];

bool [0…1];

byte [0…255];

short [-2^{16}–1…2^{16}–1];

int [-2^{32}–1…2^{32}–1].

All variables should be declared. The default initial value of basic variables is 0. There are three ways which variable can be given a value:

assignment;

argument passing;

message passing;

Arrays are supported by *PROMELA.* Start index of arrays is 0.

The process can be considered as a procedure which runs in a separate thread. It has a local state which consists of *process counter* and local variables' values. The process declaration:

Listing 3.3. Declaration and definition of the process

```
proctype proc(int a; int b)
{
  byte b; /* local variable */
  /* process body */
}
```

The body of a process consists of a sequence of statements. A statement can be in one of two states: *executable* and *blocked*. The *executable* statement can be executed immediately. The *blocked* statement cannot be executed. An assignment is always executable. All expressions are the statements. Expression is executable if it evaluates to non-zero:

4 <= 8 is always executable.

a > 14 is executable if a greater than 14. Otherwise it's blocked.

x – 1 is executable if $x \neq 1$.

A *printf* statement is always executable. It is equivalent to *printf* function in *C*. It is not evaluated during verification.

The *skip* statement is always executable. It does nothing. The only result is increment of *process counter*.

The processes can have parameters and local variables. The `active` modificator means that the process is run in the initial system state. Processes are created using `run` statement:

Listing 3.4. Process running

```
init
{
  run proc(1, 5);
}
```

The number of processes is bounded. This is because *PROMELA* defines finite state systems. The limit of active precesses in *SPIN* is 255. So *run* statement is only executable if it can create the process.

The control-flow constructions:

Listing 3.5. If construction

```
if
  :: guard1 -> S1
  :: guard2 -> S2
```

```
  ...
  :: else -> Sk
fi
```

The *do* construction is a loop.

Listing 3.6. Do construction

```
do
  :: guard1 -> S1
  :: guard2 -> S2
  ...
  :: else -> Sk
od
```

These constructions are based on Dijkstra's guarded commands [47]. Each construction must have at least one option. Each option consists of double colon, *guard* statement and other statements. An option can be selected only if the guard statement is executable. If there are several executable guard statements, one of them will be selected non-deterministically. If there are no executable guard statement these construction are blocked.

Listing 3.7. If example

```
if
  :: a < 0 -> x = x + 1;
  :: a == 0 -> x = x + 2;
  :: a >= 0 => x = x + 3;
fi;
```

In this example when *a* is equal to 0 the second (a = = 0) and the third (a >= 0) guard statements are both executable. Therefore one of these statements will be selected non-deterministically.

Listing 3.8. Do example. Exponentiation by squaring: a^e mod m

```
x = a;
int p = 1;
int q = a;
int n = e;
do
       ::n > 0 ->
```

```
                if
                  :: (n % 2) == 1 -> p = (p *
            q) % m; n = n - 1;
                  :: else -> skip;
                fi;
                q = (q * q) % m;
                n = n/2;
            :: else -> break;
    od;
```

PROMELA allows to use message passing *channels*. Channels are declared using keyword *chan* just like data variables in *PROMELA*. The instruction

```
chan a, b[4];
```

declares aninitialized channel *a* and an array of uninitialized channels *b*.

Listing 3.9. Declaraions with initializations

```
chan a = [10] of {int, byte};
chan b[4] = [0] of {short};
```

We declared the channel *a* which can store up to 10 messages. Each message is defined as a pair of *int* and *byte*. The second line of listing 3.9 declares an array of *rendezvouz channels*.

3.2. *SMV*

The tool *SMV* (*Symbolic Model Verifier*) [49] supports the verification of co-operating processes which interact via shared variables. Firstly this tool was developed for automatic verification of synchronous hardware circuits. It was also very useful for verification of communicational protocols. It was also used for big program systems, for example, in aviation [50]. In *SMV* processes may be run in synchronous or asynchronous modes. This verification tool supports model checking for *CTL* formulae. Let's show on the examples how systems may be specified and verified using *SMV*. Description of *SMV* and examples are developed in accordance with the course [1].

The specification in *SMV* input language consists of descriptions of processes, descriptions (local and global) of variables, descriptions of

formulae and formulae' specifications, which should be verified. The main module is named *main* like in *C* language. The global structure of *SMV* specification is given in listing 3.10.

Listing 3.10. Structure of *SMV* specification

```
MODULE main
VAR variables are defined here
ASSIGN global variables are assigned here

/* optional */
DEFINITION the definition of virified properties

SPEC verified CTL-specification

MODULE /* submodule 1 */

MODULE /* submodule 2 */

.............
```

The basic components of systems specification in *SMV* are:

Data types. The only types of data provided by *SMV* are bounded integer scopes and enumerated types.

Descriptions and initialization of processes. The process named *P* is defined this way:

```
MODULE P (formal parameters)
VAR local variables
ASSIGN starting assignations to variables
ASSIGN assignation to variables on transitions
```

This construction describes the module also named *P*, for which there are two methods of instance definition:

1. The instanse of *Pasync* process will be executed in asynchronous mode:

   ```
   VAR Pasync: process P(parameters)
   ```

2. The instance of *Psync* process will be executed in synchronous mode:

   ```
   VAR Psync: P(параметры)
   ```

The difference between synchronous and asynchronous modes is discussed below.

Assignation to variables. In *SMV* the process is considered as a finite state machine and is defined by enumeration for each (global and local) variable the starting value (values in the start state) and the value which will be assigned to variable in the next state. The latest value is usually depending on current values of variables. For example, `next(x):= x+y+2` assignes to *x* variable *x+y+2* value in the next state. For *x* variable the assignation `init(x):= 3` means that *x* primarily posseses the value equal to 3. The assignation `next(x):= 0` assigns 0 value to *x* variable in the next state. Assignations can be indeterministic. For example, `next(x):= {0, 1}` means that the next *x* value equal to 0 or 1. Assignasion also can be conditional. For example, the assignation

```
next(x) := case b = 0: 2;
            b = 1: {7, 12}
           esac;
```

assigns to *x* variable value 2 if *b* equal to 2 and (indeterministic) value 7 or 12 if *b* equal to 1. If *x* is a variable in the instance of *Q* process it is written as *Q.x*.

The assignation to global variables are permitted only if these variables are parameters in the process instance.

Synchronous and asyncronous modes. In acynchronous mode all the assignments to variables are executed in one atomic step. Intuitively this means that the one global clock is existing and every their tick every module makes a step. In every specified moment of time the process is either executed or not executed. Assignment to a variable *next* value is made only in case of process is executed. If the process is not executed than the value of a variable in the next step stays unchangeable.
In asynchronous mode the new value is assigned only to variables of an "active" process. Therefore, pending every tick of the global clock one process is chosen indeterministically for executing, and one step of this process is executed (while other unchosen processes maintain their state).

CTL-formulae specification. For *CTL*-formulae specification *SMV* uses symbols `&` for conjunction, `|` for disjunction, `->` for implication and `!` for negation. The checking program of *SMV* models is checking if all the possible start states fullfill the specification.

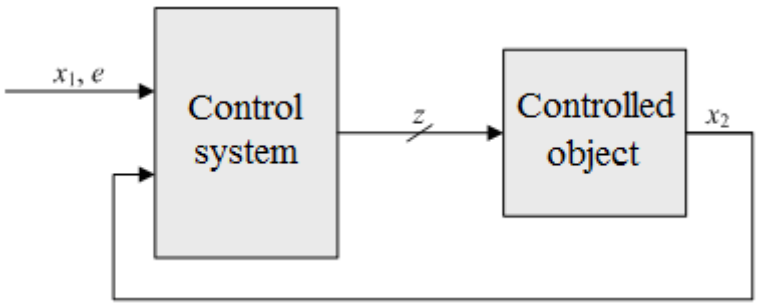
Section 4. Verification of automata-based programs

4.1. Automata-based programs

In this section the automata-based programming technology is described. The automata-based programs can be verified more easy and effective than other programs [2, 51].

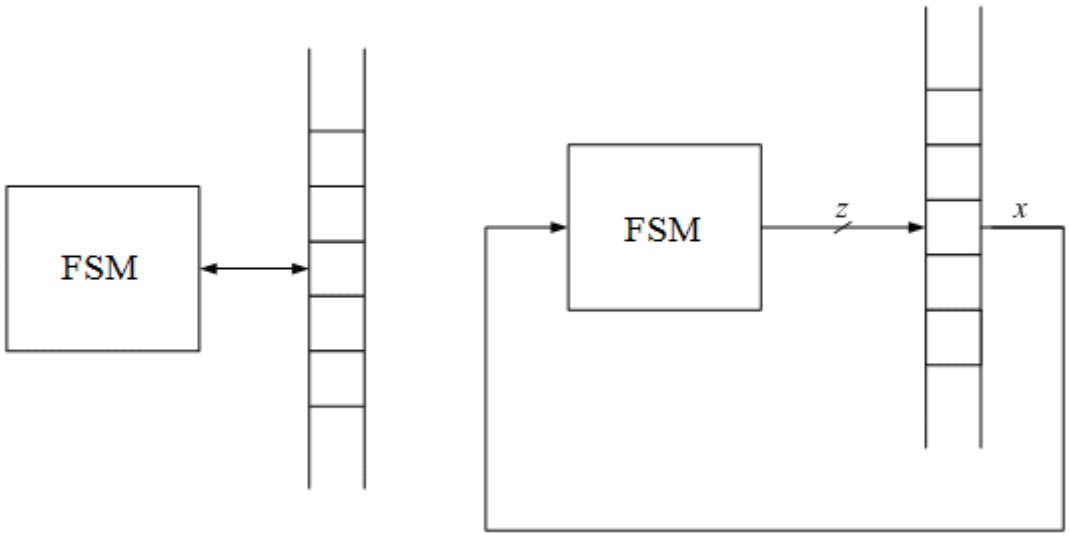
In this approach we sort out the input effects providers and automatized controlled objects. Each controlled object contains control system (system of state machines). Controlled object implements the output effects which are initiated by the control system and forms input effects. These input effects implement feedback from controlled object to control system. The input effects are also formed by external environment (by user or by adjacent system). The input effects can be two types:

- short-term events;
- input variables.



Fi g. 4.1. Scheme of automatized controlled object

The typical computability model is Turing machine (Fig 4.2). Every alghorithm can be implemented on it. Despite of this, in practice it is very laborious process. Automata-based programming uses the aspects of this mathematical model which are useful for practical programming.



a b

Fig. 4.2. Turing Machine: classic (a) and modified (b) representation

State is a base conception of automata-based programming. All states must be singled out explicitly.

Input effects are considered as tools for changing states. Output effects can be formed by state machines either in states or between states (in transitions).

There are two kinds of states: *control states* and *computational states.* In the Turing machine the finite state machine with a few states can control infinite states on the tape. In the sequel we will use the notion “state” as “control state”.

Example of automata-based program is shown in fig. 4.3.

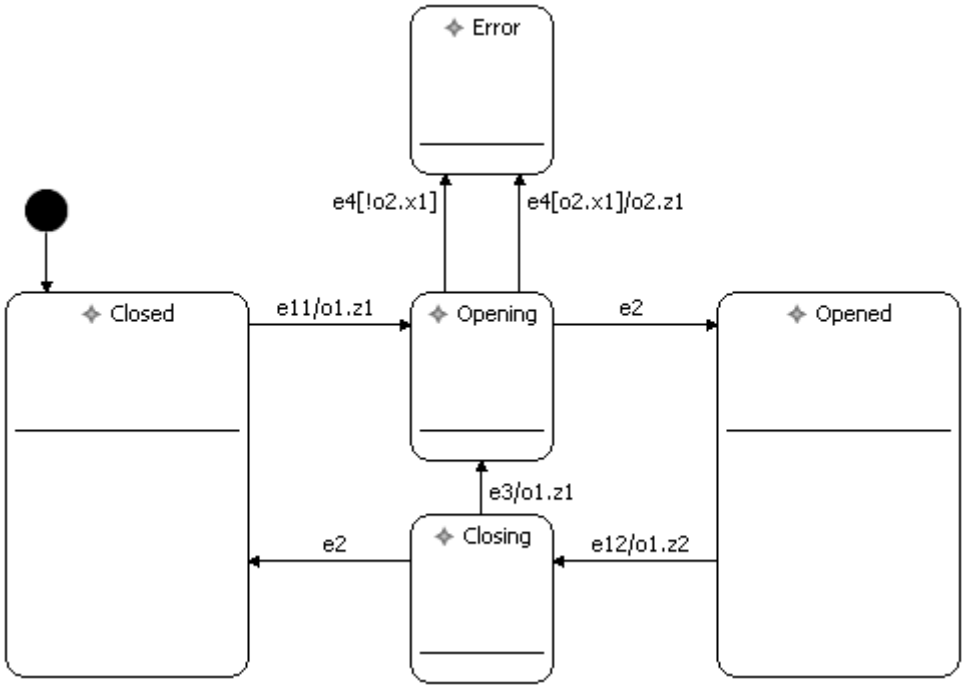


Fig. 4.3. Transition graph of finite state machine, which control lift doors

This state machine controls lift doors. Work is started when doors are in *Closed* state. When user pushes the *Open* button(`e11` event) door release is started (*`o1.z1`*). When event of opening (*`e2`*) is received state machine does to *Opened* state. The process of door closing goes in much the same way. `e12` is button *Close* event, `o1.z2` is starting of door closing mechanism. If an obstacle prevents doors from closing `e3` event is occurred. So state machine goes to *Opening* state and the doors are being opened. Also an error can be occurred (*`e4`* event) which makes state machine to go to *Error* state. If alert signal is enabled (`o2.x1`) then call emergency service is made. *Error* state is created to demonstrate facilities of verification so state machine can go to this state only from *Opening* state.

There are three kinds of state machine interaction.

1. Nesting: one state machine is nested into the state of another.
2. Calling: one state machine call another state machine.
3. Interaction by states: one state machine checks in what state *FSM* is.

In object-oriented automata-based programs interaction by states is not used.

4.2. Existing products abstract

In the beginning of research in state contract for "Verification of automata-based programs" theme [53] patent search was conducted [54].

Subjects of patent search are:

- model checking;
- temporal logic;
- preventing errors by testing and debugging.

By the result of analisys abstract of products was created. The main products are enumerated in table 4.1.

Table 4.1. Abstract of main products

Title	Assignment, description
Bogor	Perform a model for language *BIR*. Supports most basic ways of working with multithreading. *BIR* language is easy to use of object-oriented programming paradigm.It has a *GUI* based on *Eclipse*™.
Cadena	*Cadena* is an *Eclipse*™-based extensible integrated modeling and development framework for component-based systems. *Cadena's* meta-modeling capabilities can be used to formally capture the definition of widely used component models such as the *CORBA* Component Model (CCM), Enterprise Java Beans (*EJB*), *nesC* (a component model for sensor networks built on *TinyOS*). Meta-models can include attributes that represent settings and parameters for underlying middleware frameworks on which systems will be deployed.
CADP (*Construction and Analysis of Distributed Processes*)	*CADP* is a toolbox for the design of communication protocols and distributed systems. *CADP* offers a wide set of functionalities, ranging from step-by-step simulation to massively parallel model-checking. It is the only toolbox to offer: Compilers for several input formalisms, e.g.: High-level protocol descriptions written in the ISO language *LOTOS* [International Standard 8807]. The toolbox contains two compilers (*CAESAR* and

	CAESAR.ADT) that translate *LOTOS* descriptions into C code to be used for simulation, verification, and testing purposes; Low-level protocol descriptions specified as finite state machines; Networks of communicating automata, i.e., finite state machines running in parallel and synchronized together (either using process algebra operators or synchronization vectors). Several equivalence checking tools (minimization and comparisons modulo bisimulation relations), such as BCG_MIN and BISIMULATOR. Several model-checkers for various temporal logic and mu-calculus, such as EVALUATOR and XTL. Several verification algorithms combined together: enumerative verification, on-the-fly verification, symbolic verification using binary decision diagrams, compositional minimization, partial orders, distributed model checking, etc. Plus a bunch of other tools with advanced functionalities such as visual checking, performance evaluation, etc. *CADP* is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces (such as the BCG and OPEN/CAESAR software environments), which allow the *CADP* tools to be combined with other tools and adapted to various specification languages.
CBMC (*A Bounded Model Checker for C/C++ programs*)	The product checks the model for languages *ANSI C* и *C++*. It allows you verify the output of the border arrays, safety pointers, exceptions, and user assertions. Supports the most important elements of *ANSI C*: multidimensional arrays, pointer arithmetic, fractional fixed-point arithmetic, etc.
GEAR (*A game based model checking tool capable of CTL, modal &-calculus and*	Graphical environment, which allows you to build models and perform their verification using logic *CTL* and μ-calculation. There is a lot of visual material for studying. Collaborates with project Specification patterns, which is an organization of repository of the various requirements of correctness.

specification patterns)	
Java Pathfinder	Verifies the executable Java byte code. Can find execution paths that lead to an unhandled exception, locks, etc. Check the program up to 10 000 lines of code. It is used in *NASA* Ames Research Center.
LTSA (*Labelled Transition System Analyser*)	The product is intended for the verification of distributed and parallel systems (concurrent systems). They are defined by *LTS* (Labelled Transition System) and *FSP* (Finite State Processes). Correctness requirements are specified as finite state machines (in the earlier version supported by the formula *LTL*). There are plugins, for example, for verification of web services and a graphical interface.
MOPS (*Model Checking Programs for Security Properties*)	The verifier models extracted from the program code written in C. Correctness requirements are defined in a special form and correspond to the assertions of so-called "defensive programming". Contains a ready base of such statements, it is planned to write a graphical user interface.
NuSMV (*A New Symbolic Model Checker*)	This is an updated version of the verifier *SMV* – symbolic verifier models (Symbolic Model Checker). It performs the verification by combining *BDD* (Binary Decision Diagrams) and model checking based on *SAT* (*SAT*-based model checking). Supported methods of specifying the requirements of correctness: *CTL*, *LTL*.
ORIS (*Uses a CTL-like temporal logic with real-time bounds, action and state based*)	Performs symbolic verification of systems given in the form of hierarchical automata for which the requirements of a well.
SMV (*Symbolic Model Checker*)	Performs symbolic verification of systems given in the form of hierarchical automata for which the requirements of correctness are defined in temporal logic CTL. Argued that the system can be defined as detailed and abstract. Uses the transformation of machines into the Kripke model.

SPIN	Performs verification of the model for language *PROMELA*. This language supports only discrete data types and functions for working with multithreading. Uses the logic *LTL*.
UPPAAL (*Uppaal Model Checker*)	Environment for modeling and verification of real-time systems. Uses the network time machine, advanced data types (bounded integers, arrays, etc.). Modeling languages - Timed Automata. Correctness requirements are specified in the sublanguage *TCTL*.
VIS (*Verification Interacting with Synthesis*)	Performs verification of the model using the hierarchical finite state machines and language *Verilog*. Verifiable statements are defined by logic *CTL*.
dSPIN	Expanding the tool *SPIN*. Allows verifying distributed and parallel systems more effectively. This tool extends the capabilities of the tool *SPIN*. Implements some new features in addition to the study algorithms of state and space reduction in the number of states used in *SPIN*: pointers, dynamic allocation/deallocation, recursive functions, function pointers, garbage collection, symmetrical reduction. Sufficiently powerful input language (an extension of language *PROMELA*).
DBRover	Monitor execution time (runtime monitor) for temporal rules, written in languages *LTL* and *MTL*. This is an automatic, remote and the graphical version of tools *TemporalRover*. Contains an editor of temporal formulas, generator and code compiler, simulator. Modeling languages: Ada, *C, C + +*, *Java*, *VHDL* and *Verilog*. Supported languages of requirements of correctness: *LTL*, *MTL* (*Metric Temporal Logic*), etc.
Reactis Tester	Models and testing rocket systems. Modeling languages: *Simulink* / *Stateflow*.
Temporal Rover	Perform requirements formulated in the language of *LTL* with constraints of real time (with real time constraints). Generates executable code for specifications written in the form of comments. Supports the validation of the properties of the form “after the occurrence of an event e value of x in 5 percent of cases does not change, and its

	average value is greater than 100 for one hour or until such time when the event e1 happens twice". Input languages: *Ada*, *C*, *C + +*, *Java*, *VHDL*, Verilog. Supported methods of specifying the requirements of correctness: *LTL*, *MTL* (*Metric Temporal Logic*), etc. There is also a project *StateRover*, which is based on this tool, but is newer and powerful. *StateRover* has more opportunities than *Temporal Rover* and operates on a subset of *UML*. It can generate code in languages *Java*, *C*, *C + +*. It has a powerful GUI, implemented as a plugin for *Eclipse* ™.

4.3. Tools and subjects of verification

4.3.1. Automatic teller machine model

Automatic teller machine (*ATM*) is device for automatization operations of issuance and transfer money. Client is identified by his banking card and corresponding PIN. ATM performs the following operations:

- identifies client;
- allows to inspect available funds;
- allows to draw out;
- connects with bank.

ATM model consists of two state machines. State machine *AClient* controls user interface. State machine *AServer* performs account operations and connects with bank. In addition there are following event providers and controlled objects.

Event providers:

- *HardwareEventProvider* – generated by hardware system events;
- *HumanEventProvider* – events which are initiated by user;
- *ServerEventProvider* – answers from server;
- *ClientEventProvider* – requests to server.

Controlled objects:

- *FormPainter* – provides form visualisation;
- *ServerQuery* – sends requests to server;

- *ServerReply* – sends answers from server.

ATM model was developed using *UniMod* tool which is created in *SPbIFMO SU* [58, 75, 76]. Connection scheme is represented on fig. 4.4.

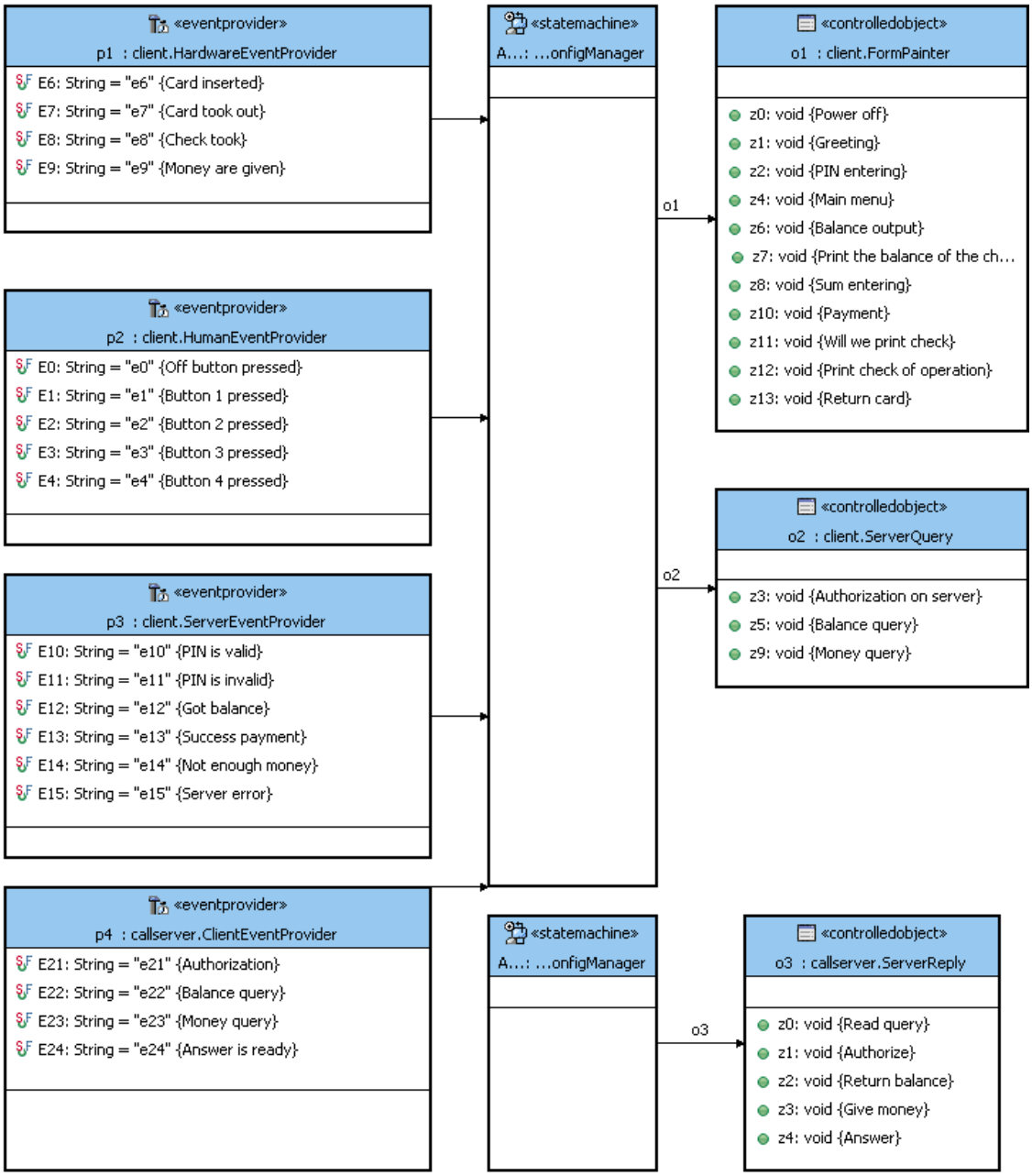


Рис. 4.4. Connection scheme of *ATM* model

Transition graph of *AClient* state machine is represented in fig. 4.5. Transition graph of *Aserver* state machine is represented in fig. 4.6.

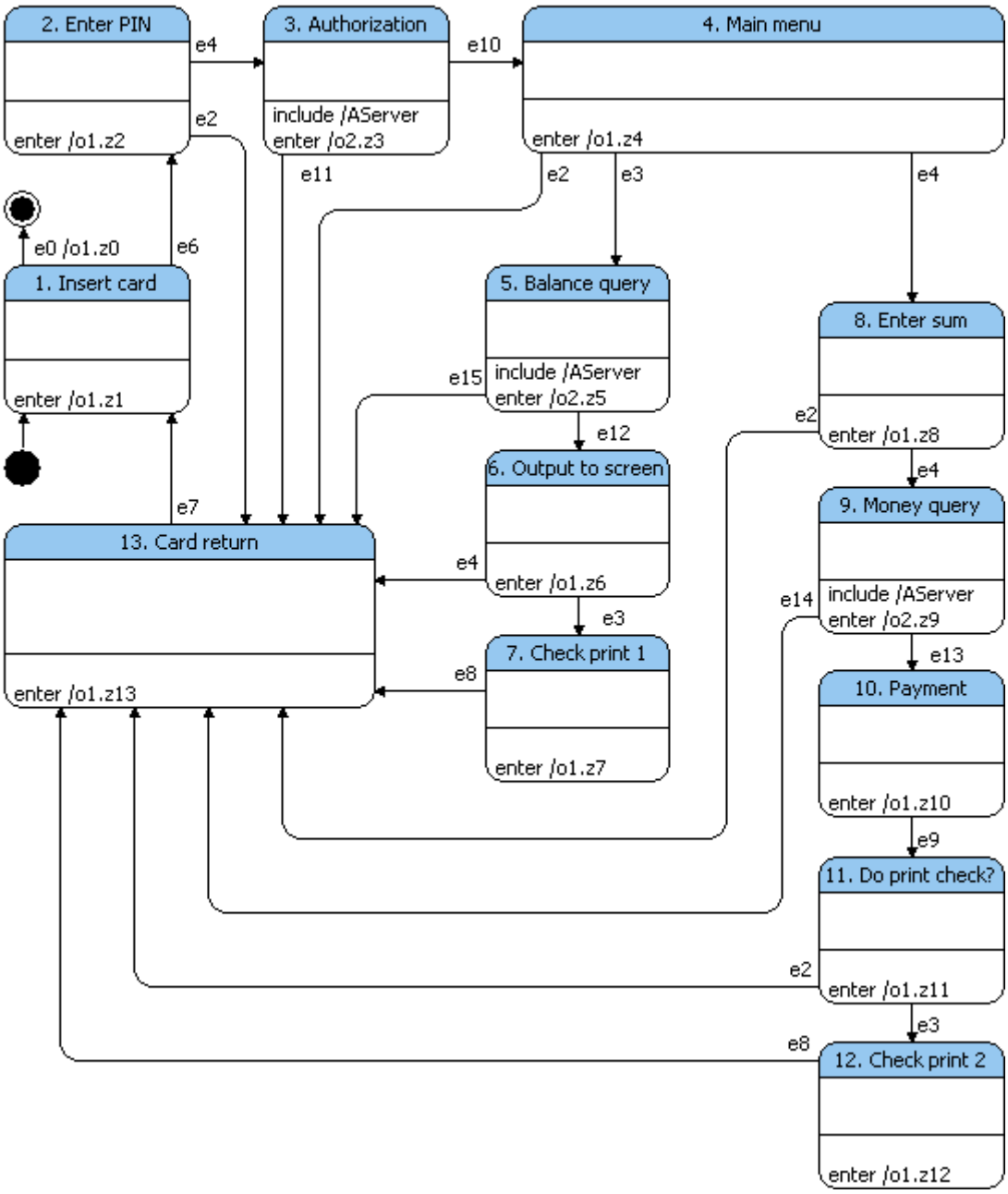


Fig. 4.5. *AClient* state machine

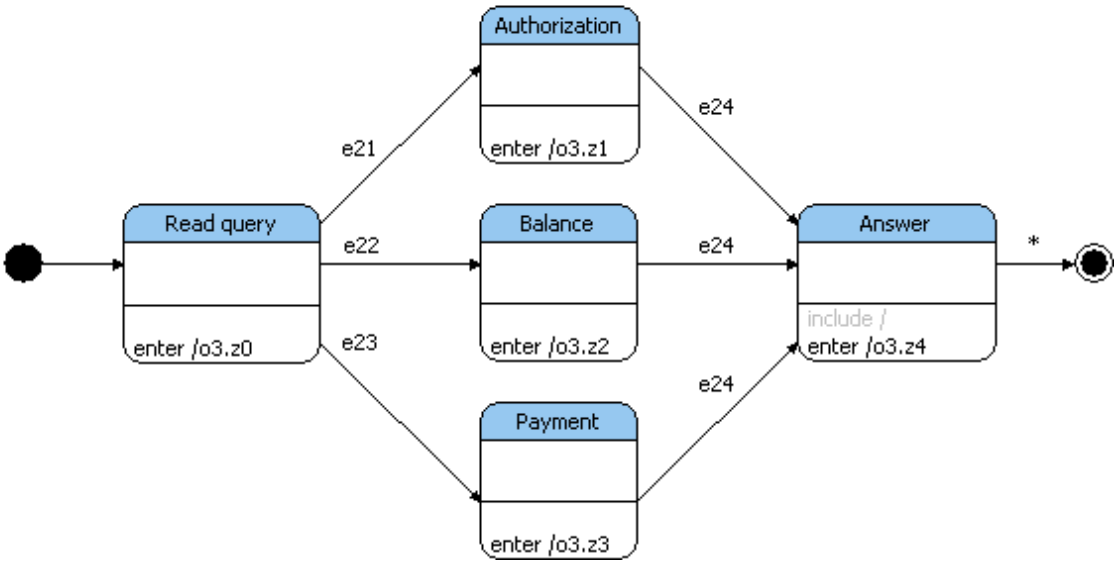


Fig. 4.6. *AServer* state machine

4.3.2. Properties of *ATM* to verify

Each verification tool will check following two properties.

The first property (call it Σ) is "User can not receive money if he did not enter correct PIN". This assertion must be true in ATM so the verification result must be successful. Let us reformulate this assertion: "It can not be that user did not enter correct PIN before he receive money". Verbal formula is translated to LTL-formula directly:

¬(¬[*enter correct PIN*] **U** [*payment*]).

As represented in fig. 4.5 payment occurs in `o1.z10` action. So long as one of verification tools can not check execution of output actions we will use that fact that this action is executed only in "*10. Payment*" state. When user enters PIN correctly `e10` event is occurred. Therefore formula takes the form:

¬(¬*e*10 **U** (*AClient* in "*10. Payment*"))

In CTL this assertion is formulated in much the same way:

¬**E**[¬*e*10 **U** (*AClient* in "*10. Payment*")].

The second assertion (Ω) is "User is sure to receive money". This assertion must not to be true in ATM model. So verification tools must represent

counter-examples for this assertion. Verbal formula is translated to LTL-formula directly:

F[*payment*].

As considered before payment occurs in only one state so formula takes the form:

F(*AClient* in "*10. Payment*").

In *CTL*:

AF(*AClient* in "*10. Payment*").

4.4. Tools utilizing existing verifiers

4.4.1. *Converter*

Common description

Converter [77–79] utilizes may be the most known verifier: *SPIN* [46, 80]. This tool uses *LTL* for verification. Model for checking is described in *PROMELA* language.

The *Converter* tool standard tasks for this case:

- Conversion of automata-based system into *PROMELA* language;
- definition of predicates for assertions formulation;
- conversion of counter-example into terms of source system.

Atomic states extraction

The described in *PROMELA* language model contains following variables in which model state is stored:

`int lastEvent;` This variable stores a number of last processed event;

`int stateAi;` This variable stores a number of current state of state machine `Ai`.

In this approach the atomic state of automata-based system is set of current states of each state machine and last processed event.

Checking assertions

This verification tool checks formulae which are written in LTL. There are two kinds of predicates:

`lastEvent = e1` is true when last processed event is `e1`;

`stateAi = 1` is true when the number of current state in `Ai` state machine in described in *PROMELA* model equals `1`. Note: this number is not a name in source system, so user need to find it in described in *PROMELA* model manually.

Counter-example conversion

In case of finding error *SPIN* can "replay" the counter-example in the model. It means that *SPIN* will execute *PROMELA* model as a real program.

Converter uses this feature. Information about current model state is printed very transition. So when *SPIN* replies counter-example this information is printed to standard output. Example of this output is represented in listing 4.1.

Listing 4.1. Output of counter-example by *Converter*

```
State Test 1 : init
Going to state Test 2 : s0
Event = e2
State Test 2 : s0
Going to state Test 33 : s1-1
Event = e3
State Test 33 : s1-1
```

Detailed tool description

Converter works with automata-based programs which are developed using *UniMod* tool. This tool, as mentioned before, provides creating automata-based programs visually and execution of them. Automata-based program is represented as *UML*-diagram. Controlled objects and event providers are developed in *Java*.

UniMod allows to save automata-based program as *XML* file. This file is used as input for *Converter*.

Distribution kit is provided with all necessary libraries and *SPIN* tool. The only required thing is to install *gcc* or compatible compiler *C* and include path to the *gcc* to the PATH environment variable.

This command runs *Converter* tool:

```
run.cmd <XML file of system> <report file>
        <LTL-formula>
```

Where

`XML file of system` – automata-based system to verify (developed in *UniMod* and saved as *XML* file);

`Report file` – path to file where will report about verification will be put;

`LTL-formula` – formula with assertion to verify, for example, `"!(<>{lastEvent == e1})"`.

For LTL-formulae following notation is used:

`[]` – *Globally* (always);

<> – *Future* (some time in future);

`U` – *Until* ;

`V` – *Reverse until* `p V q` is equivalent to `!(!p U !q)`;

`!` – negation;

`&&` – logical *AND*;

`||` – logical *OR*;

`->` – implication;

`<->` – equivalency.

ATM-model verification

Converter works with *UniMod* models. Therefore no additional conversions needed. We need only to generate *XML* file with model description from *UniMod* graphical user interface.

Verification of the first property. Firstly we need to generate a *XML* file with model description. Let it be `Bankomat.xml`. *Converter* creates an identifier for every state of state machines in system so these identifiers are

needed in *LTL* formulae. To learn what identifier will be assigned to "*10. Payment*" state we need to run *Converter* with no formula:

```
run.cmd Bankomat.xml report.txt ""
```

As a result `model.ltl` file will be generated. At the beginning of this file correspondence between state identifiers and names (listing 4.2).

Листинг 4.2. Result of verification with no formula

```
...
#define STATE_10  10 /*8. Ввод суммы*/
#define STATE_11  11 /*10. Выдача денег*/
#define STATE_12  12 /*9. Запрос денег*/
...
```

So we learnt that to "*10. Payment*" state has "11" identifier. Now we can write formula for *Converter*:

```
!(!{lastEvent == e10} U {stateAClient == 11}).
```

So the input parameters for *Converter* will be following:

```
run.cmd Bankomat.xml report.txt "(!{lastEvent ==
e10} U {stateAClient == 11})"
```

The result of verification is stored in `report.txt` file (listing 4.3).

Listing 4.3. Result of verification Σ assertion

```
Converter v. 0.50
warning: for p.o. reduction to be valid the never claim
      must be stutter-invariant
(never claims generated from LTL formulae are stutter-
      invariant)
(Spin Version 4.2.8 – 6 January 2007)
      + Partial Order Reduction


Full statespace search for:
      never claim          +
      assertion violations  + (if within scope of claim)
      acceptance   cycles   + (fairness disabled)
      invalid end states  - (disabled by never claim)


State-vector 32 byte, depth reached 139, errors: 0
```

```
      129 states, stored
        8 states, matched
      137 transitions (= stored+matched)
        0 atomic steps
hash conflicts: 0 (resolved)
```

Important information consists in "`errors: 0`" line. It means that there are no errors found in model. The result of verification is correct.

Verification of the second property (Ω):

```
run.cmd Bankomat.xml report.txt
        "(!<>{stateAClient == 11})"
```

As a result there is following information in `report.txt` file (only significant lines):

Listing 4.4. Результат верификации свойства Ω банкомата

```
...
State-vector 28 byte, depth reached 33, errors: 1
...
Never claim moves to line 267 [(!((stateAClient==11)))]
          State AClient 1 : s1
          Going to state AClient 13 : 1. Insert card
          State AClient 13 : 1. Insert card
          Going to state AClient 6 : s2
          Event = e0
          State AClient 6 : s2
spin: trail ends after 34 steps
...
```

The report contains the counter-example for verifying assertion. It is easy to read this file and complementary conversion is not needed. The ATM starts working in the state *s1*. It goes to state "*1. Insert card*". After that user pushes the power off button and event *e0* is arisen. So the ATM goes to the terminal state and powers off. Therefore user never gets the money. So this counter-example is correct.

4.4.2. *Unimod.Verifier*

General description

UniMod.Verifier [77, 81, 82] uses verifier *Bogor* [83]. The input language of verifier *Bogor* is named *BIR* and can be extended by new types of data. These data types are classes with internal states. User can call actions which can change the internal state of the class and request values of class variables.

Possibility of input language extension allows abstracting away from unnecessary details of verifying model. In theory it reduces amount of model states and therefore speeds up verification process.

In *UniMod.Verifier* a new class was implemented. This class performs only one operation: *step* (process next event). This operation nondeterministically chooses an event and sends it to automata-based program.

The main advantage of this approach is that the description of the program in the *Bogor* input language is became trivial and the same for any automata-based program. The logic of the model and the states storage are implemented in software and once for all automata-based systems. No need to generate a new intermediate input data for each verifying system.

In addition, in *UniMod.Verifier* is used another original solution. In other verifiers (for example, *Converter*) there are different implementations of state machines in the original program and model. Therefore there is no guarantee that original program and model will work equally. In *UniMod.Verifier* we can say that *Bogor* verifies state machine directly in the *UniMod*. The interaction between *UniMod* and *Bogor* in *UniMod.Verifier* in diagram form is shown in fig. 4.7.

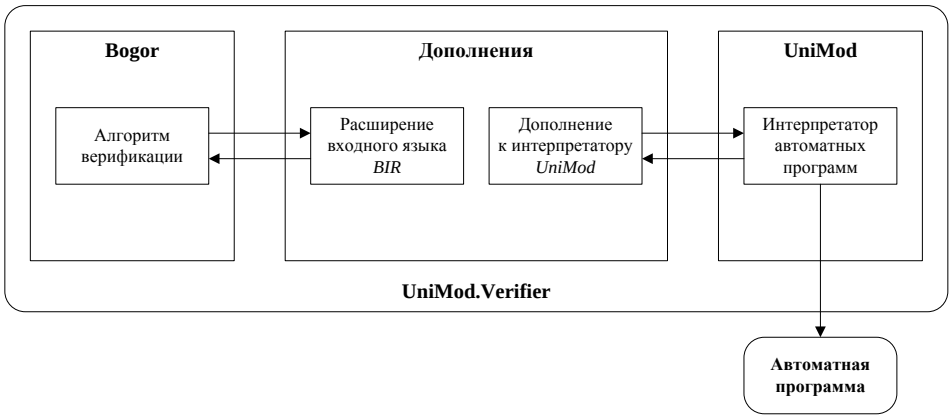


Fig. 4.7. Interaction between *UniMod.Verifier* components

Atomic states extraction

Every class which extends *BIR* must return to verifier a number or a set of numbers which encode its current state. Automata extension defined in the verifier *UniMod.Verifier*, is programmed to return the current set of states of automata as a condition of the entire machine-gun program. Thus, in this verifier machine transitions are not split to the elementary states.

Verified properties

UniMod.Verifier works with *LTL*. As a predicate automaton *BIR* language extension provides the following features:

- *wasEvent(e)* returns *True*, if in the last step event *e* was chosen to process and *False* otherwise;
- *wasInState(sm, s)* returns *True*, if state machine *sm* before the last step was in *s* state;
- *isInState(sm, s)* returns *True*, if after the last step automata *sm* is in state *s*;
- *cameToState(sm, s)* returns *True*, if after the last step state machine *sm* changed state to *s*. Equals *(isInState(sm, s) && !wasInState(sm, s))*;
- *cameToFinalState()* returns *True*, after the last step the main state machine came into final state. This means that the program finished its work;
- *wasAction(z)* returns *True*, if action *z* was invoked;
- *wasFirstAction(z)* returns *True*, if the first invoked action was *z*;
- *wasLastAction(z)* – returns *True*, if the last invoked action was *z*;
- *getActionIndex(z)* return number of action in the list of invoked actions during the last step. This predicate is used to formulate statements that specify how to call action control objects in the automata-based program;
- *wasTrue(g)* returns *True*, if during the last step one of transitions was marked by condition *g* and *g* was *True*. Example of condition: *g = !o1.x1 && o1.x2*;
- *wasFalse(g)* returns *True*, if during the last step one of transitions was marked by condition *g* and *g* was *False*.

Counter-example conversion

UniMod.Verifier works directly with automata-based program so conversion is not needed.

Tool description

Let us describe how to work with *UniMod.Verifier*.

Firstly we need to formulate property. Formula is written to `unimod.bir` file. This file contains model information besides formula.

- *LTL.always* – (**G**) globally, always;
- *LTL.eventually* – (**F**) sometimes in future;
- *LTL.next* – (**X**) in the next step;
- *LTL.until* – (**U**) until;
- *LTL.weakUntil* – (**W**) until or always;
- *LTL.release* – (**R**) release: $p \mathbf{R} q = \neg(\neg p \mathbf{U} \neg q)$;
- *LTL.negation* – negation;
- *LTL.equivalence* – equivalence;
- *LTL.implication* – implication;
- *LTL.conjunction – AND*;
- *LTL.disjunction – OR*.

Formula is written as a function (listing 4.5).

Listing 4.5. Example of function with LTL-formula

```
fun NoPinNoMoney() returns boolean =
   LTL.temporalProperty(
     property.createObservableDictionary(
        property.createObservableKey("correct_pin",
              AutomataModel.wasEvent(model, "e10")),
        property.createObservableKey("give_money",
              AutomataModel.wasAction(model, "o1.z10"))
      ),

      LTL.weakUntil (
```

```
            LTL.negation(LTL.prop("give_money")),
            LTL.prop("correct_pin")
        )
    );
```

This function expresses *!o1.z10 W e10* formula.

After that we call verifier:

```
verifier.cmd Bankomat.xml NoPinNoMoney
```

`Bankomat.xml` is a file created by *UniMod*. It contains automata-based program which we want to verify. `NoPinNoMoney` is name of function of *LTL*-formula (listing 4.5).

Result of verification will be printed to standard output. At the end of verification *UniMod.Verifier* prints a message about successful verification or counter-example.

ATM-model verification

UniMod.Verifier works with *XML*-description of automata-based program generated by *UniMod* (just like as *Converter*).

Let us verify the first property (“User can not receive money if he did not enter correct PIN”).

Listing 4.6. Property Σ for *UniMod.Verifier*

```
fun NoPinNoMoney() returns boolean =
   LTL.temporalProperty(
     property.createObservableDictionary(
        property.createObservableKey("correct_pin",
            AutomataModel.wasEvent(model, "e10")),
        property.createObservableKey("give_money",
            AutomataModel.isInState(model, "/AClient",
            "10. Выдача денег"))
     ),

     LTL.negation (
        LTL.until (
           LTL.negation(LTL.prop("correct_pin")),
```

```
                LTL.prop("give_money")
            )
        )
    );
```

In this function we define predicates *correct_pin* (event *e10* was occured) and *give_money* (*AClient* state machine is in state "*10. Payment*"). After that we write *LTL* formula *!(!<correct_pin> U <give_money>)*.

Command-line instruction for verification of this property:

```
verifier.cmd Bankomat.connectivity NoPinNoMoney
```

As a result the following information will be printed:

Listing 4.7. Result of verification of property Σ

```
(W) Unknown option
  edu.ksu.cis.projects.bogor.module.Isearcher.maxErrors


Transitions: 1, States: 1, Matched States: 0, Max
  Depth: 1, Errors found: 0, Used Memory: 2MB
Transitions: 63, States: 41, Matched States: 22, Max
  Depth: 14, Errors found: 0, Used Memory: 1MB
Total memory before search: 708a688 bytes (0,68 Mb)
Total memory after search: 1a134a712 bytes (1,08 Mb)
Total search time: 688 ms (0:0:0)
States count: 41
Matched states count: 22
Max depth: 14
Done!
Verification successful!
```

As expected, in the last line was said that there are no errors.

Let us verify the second property ("User is sure to receive money"). Its *LTL* formula is shown in listing 4.8.

Listing 4.8. LTL formula for Ω property for *UniMod.Verifier*

```
    fun AlwaysMoney() returns boolean =
```

```
LTL.temporalProperty (
  property.createObservableDictionary (
     property.createObservableKey("give_money",
        AutomataModel.isInState(model, "/AClient",
        "10. Выдача денег"))
   ),

   LTL.eventually (LTL.prop ("give_money"))
);
```

Start verification process (save result to `verifier.out` file):

```
verifier.cmd Bankomat.connectivity AlwaysMoney
             > verifier.out
```

After verification we will get following result (there are only significant lines in listing 4.9):

Listing 4.9. The result of verification of property Ω

```
Generating error trace 0...
Done!
1 traces were found.
Replaying the trace with least states (#0).
Replaying trace by key: 0
Stack of transitions leading to the error:
Model [ step [0] event [null] guards [null]
transitions [null] actions [null] states [null] ]
fsaState [bad$accept_init]
Model [ step [0] event [] guards [] transitions []
actions [] states [(/AClient:9. Money
query/AServer) - (Top); (/AClient:5. Balance query
/AServer) - (Top); (/AClient:3.
Авторизация/AServer) - (Top); (/AClient) - (Top)] ]
fsaState [bad$accept_init]
Model [ step [1] event [*] guards [] transitions
[s1#1. Insert card#*#true] actions [o1.z1] states
[(/AClient:9. Money query/AServer) - (Top);
(/AClient:5. Balance query/AServer) - (Top);
(/AClient:3. Authorization/AServer) - (Top);
(/AClient) - (1. Insert card)] ] fsaState
[bad$accept_init]
```

```
Model [ step [2] event [e6] guards [true->true]
transitions [1. Insert card#2. Enter PIN#e6#true]
actions [o1.z2] states [(/AClient:9. Money
query/AServer) – (Top); (/AClient:5. Balance
query/AServer) – (Top); (/AClient:3.
Authorization/AServer) – (Top); (/AClient) – (2.
Enter PIN)] ] fsaState [bad$accept_init]
Model [ step [3] event [e2] guards [true->true]
transitions [2. Enter PIN#13. Card return#e2#true]
actions [o1.z13] states [(/AClient:9. Money
query/AServer) – (Top); (/AClient:5. Balance
query/AServer) – (Top); (/AClient:3.
Authorization/AServer) – (Top); (/AClient) – (13.
Card return)] ] fsaState [bad$accept_init]
Model [ step [4] event [e7] guards [true->true]
transitions [13. Card return#1. Insert
card#e7#true] actions [o1.z1] states [(/AClient:9.
Money query/AServer) – (Top); (/AClient:5. Baance
query/AServer) – (Top); (/AClient:3.
Authorization/AServer) – (Top); (/AClient) – (1.
Insert card)] ] fsaState [bad$accept_init]
Done!
```

As expected `verifier.out` contains a counter-example. This counter-example corresponds to following situation:

6. ATM starts working. *AClient* state machine goes to "*1. Insert card"* state.
7. User inserts a card (*e6* event).
8. *AClient* state machine goes to "*2. Enter PIN"* state.
9. User pushes "Cancel" button.
10. *AClient* state machine goes to "*2. Card return"* state.
11. User removes the card (*e7* event).
12. *AClient* state machine goes to "*1. Insert card"* state.

So verifier found a cycle. If this cycle is repeated indefinitely, user will not receive money. Therefore this result is counter-example.

4.4.3. *FSM Verifier*

General description

FSM Verifier [65, 77] is based on *NuSMV* verifier which uses *SMV* as input language. Operation with it consists of a few standards for this case steps:

1. Convert automata-based program to *SMV* language.
2. Write specification for program in temporal logic language.
3. Run *NuSMV*.
4. Convert counter-example to terms of automata-based program.

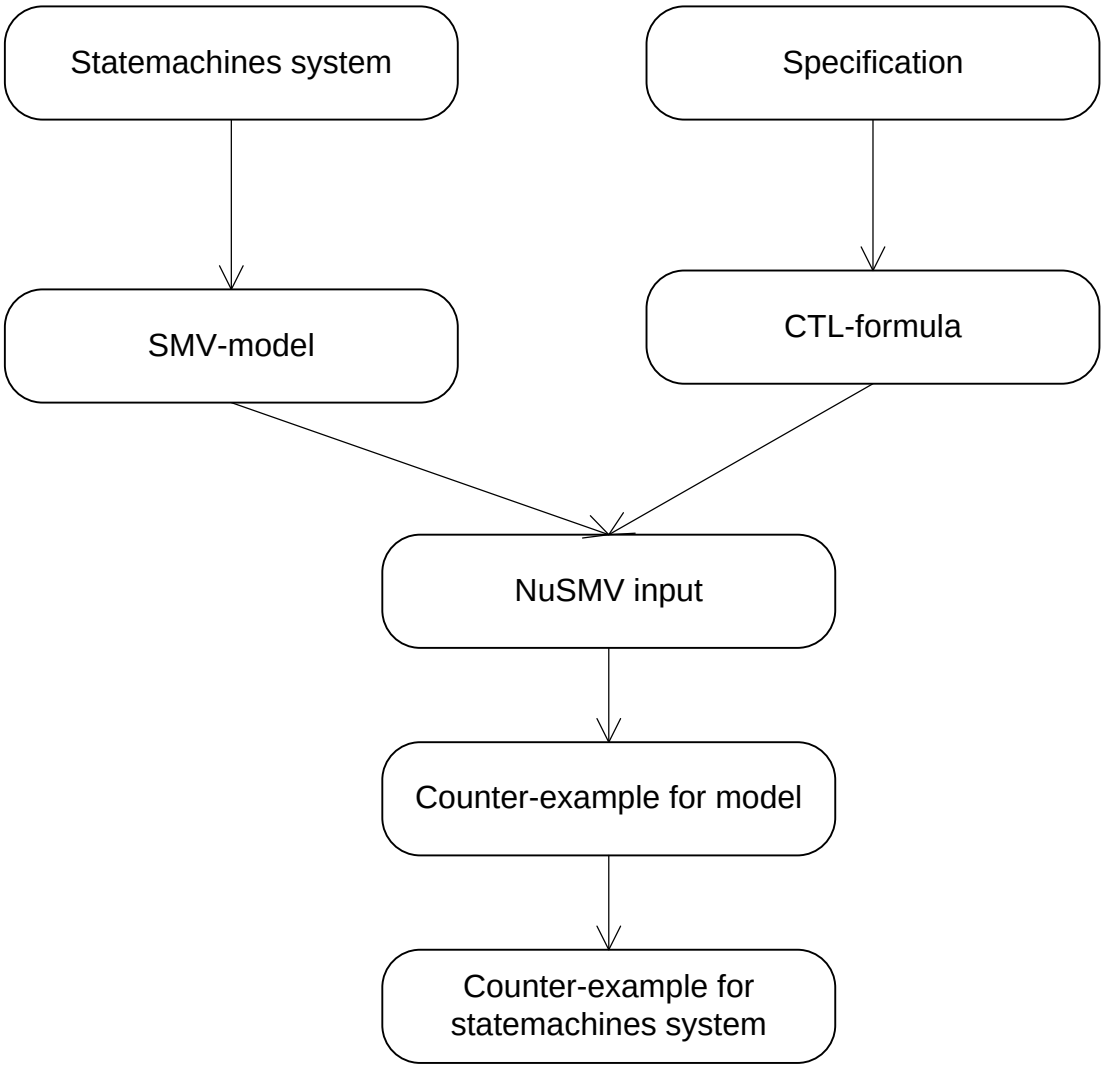


Рис. 4.8. Scheme of *FSM Verifier* work

The state machines system which *FSM Verifier* verifies is written in its own format. The system consists of set of state machines with output effects on their transitions and states. For each state machine a set of events to process is specified. Each transition can contain an event and a condition. Condition is a logical formula. Its' predicates are input variables ($x_1, x_2, \ldots$) and expressions like A_i in s_j (is true when A_i is in state s_j). The input variables can be Boolean only. Transition can also contain the sequence of output effects like $o.z_i()$ and a command like $A_i.e_j()$ for transfer of control to other state machines.

Atomic states extraction

FSM Verifier separates an intermediate state each time when state machine performs one of following actions:

- invoke an output effect;
- Invoke other state machine.

The example of intermediate states extraction is shown in fig. 4.9.

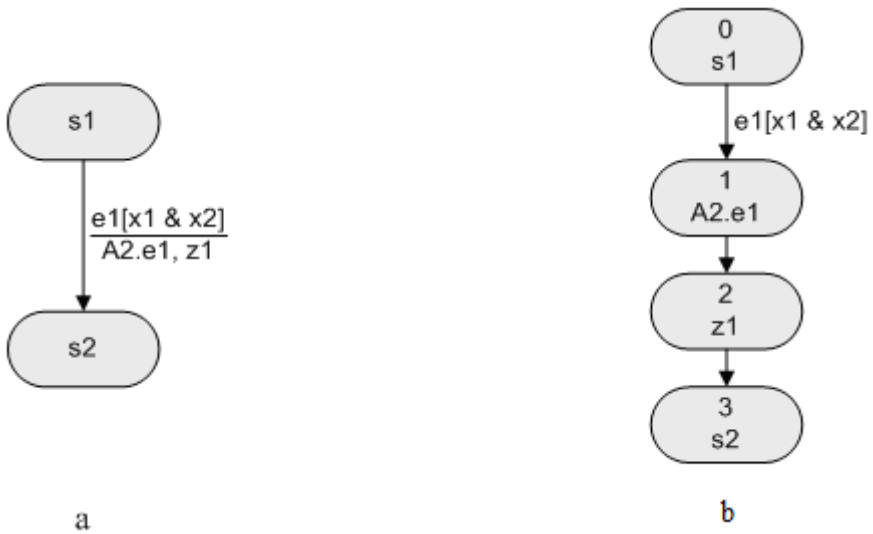


Рис. 4.9. Intermediate states extraction.
Source transition (a), transformed transition (b)

Verified properties

FSM Verifier allows verifying temporal formulae using following predicates:

- A_k is in state s_j;
- z_i output effect was invoked;
- event e_i was occurred.

NuSMV verifier and therefore *FSM Verifier* allow verifying properties formulated in CTL temporal logic. **AF** *f*, **AG** *f*, **A** [*f* **U** *g*] temporal operators are allowed in formulae. In addition standard logical operators are allowed.

Counter-example conversion

The report about error in verification which is returned by *NuSMV* verifier contains a counter-example in terms of created model of automata-based program. The counter-example contains sequence of intermediate states leading to error. Each intermediate state unambiguously defines state or transition of source system. So it is easy to convert counter-example to terms of source system. Detailed implementation of this method is described in [65].

Tool description

The distribution kit of this tool contains two files:

- *verifier.jar*. Program for conversion state machine system to model in *SMV* language;
- *counterexample.jar*. Program for counter-example conversion.

In addition following programs are needed for tool working:

- *NuSMV* verifier;
- *Java Runtime Environment*.

All of these programs can work both of *Windows* and *Linux*.

The input of *FSM Verifier* is state machines system written in *XML* format. Structure of input file was developed for *FSM Verifier* and is not supported by the other verification tools. This structure can be found in [77].

Formula with specification is written in the *XML* file with model as plain text (listing 4.10).

Листинг 4.10. Запись формулы в верификаторе *FSM Verifier*

```
<specification>
```

```
    <string>AG (A0.s1 -&gt; AF A1.s1)</string>
  </specification>
```

For verification we need to run following instructions:

- For creating model from automata-based system and specification (both are written in *inputfile.fsm*). The model is written to *input.smv* file:

```
java –jar fsmverifier.jar inputfile.fsm > input.smv
```

- For invoking *NuSMV* verifier:

```
NuSMV input.smv > verifier.out
```

- For counter-example conversion:

```
java –jar counterexample.jar verifier.out inputfile.fsm
```

The result is printed into standard output as a table in *HTML* format. Each row contains the following information:

- step number;
- name of active state machine;
- processing event;
- name of current state of each state machine;
- performing action;
- values of output effects.

ATM-model verification

The format of *XML* file of automata-based program developed in *UniMod* differs from input format for *FSM Verifier*. To convert it to the format for *FSM Verifier* the special algorithm is used.

There is a possibility to assign any name to states in the *UniMod*. But *FSM Verifier* can not work with some symbols in the names so *FSM Verifier* gives new names to states. In addition names of some special events.

As a result of conversion of state machine *AClient* "*10. Payment*" got the identifier *s11*, so the predicate *AClient in "10. Payment"* takes the form *AClient.s11*. Also in the *FSM Verifier* tool in writing the predicate of this event is required to write which state machine handles it. In Fig. 4.5, 4.6, is shown that the event *e10* is processed by *AClient* state machine only. Formula for property Σ is represented in listing 4.11.

Listing 4.11. Formula for property Σ

```
<specification>
  <string>!E[!AClient.e10 U AClient.s11]</string>
</specification>
```

Convert the ATM model from *UniMod* format to *Bankomat.fsm* file (in *FSM Verifier* format) and write the specification here. Run the following command:

```
java –jar verifyer.jar Bankomat.fsm
```

The result of this command is *out.smv* file. Run *NuSMV* verifier for this file:

```
NuSMV out.smv
```

The result of this command is shown in listing 4.12.

Listing 4.12. Result of property Σ verification

```
C:\Verifiers\FSM Verifier>NuSMV out.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54
UTC 2007)
*** For more information on NuSMV see
<http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.


*** This version of NuSMV is linked to the MiniSat SAT
solver.
*** See
http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright I 2003-2005, Niklas Een, Niklas Sorensson


-- specification !E [ !AClient.e10 U AClient.s11 ]   is
true
```

As expected in the last line said that property Σ is true.

Verification of Ω property. CTL formula for this property is shown in listing 4.13.

Listing 4.13. Formula for Ω property

```
<specification>
  <string>AF AClient.s11</string>
```

```
  </specification>
```

Input for *NuSMV* generation:

```
java –jar verifyer.jar Bankomat.fsm
```

running of *NuSMV*:

```
NuSMV out.smv
```

The significant part of the result is shown in listing 4.14.

Listing 4.14. The significant part of Ω property verification

```
C:\Verifiers\FSM Verifier>NuSMV out.smv
*** This is NuSMV 2.4.3 (compiled on Tue May 22
14:08:54 UTC 2007)
*** For more information on NuSMV see
<http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT
solver.
*** See
http://www.cs.chalmers.se/Cs/Research/FormalMethods/Min
iSat
*** Copyright I 2003-2005, Niklas Een, Niklas Sorensson

-- specification AF AClient.s11  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  AClient.State = 0
  AServer.State = 0
  Active = 0
  Event = 0
AClient.s9 = 0
```

. . .

NuSMV found a counter-example. Let us save the *NuSMV* ‘s output to file:

```
NuSMV out.smv > verifier.out
```

After that convert counter-example to terms of automata-based program:

```
java –jar counterexample.jar verifier.out Bankomat.fsm
```

The result of conversion is *out.html* file. This file contains table shown in table 4.2.

Table 4.2. Counter-example for Ω property of ATM model

Step	Active	Event	AClient	AServer	Action
1	AClient	–	s1	s1	–
2	AClient	eAlways	s1	s1	o1.z1
3	AClient	–	s13	s1	–
4	AClient	e6	s13	s1	o1.z2
5	AClient	–	s9	s1	–
6	AClient	e2	s9	s1	o1.z13
7	AClient	–	s7	s1	–
8	AClient	e7	s7	s1	o1.z1
9	AClient	–	s13	s1	–

To understand this counter-example we need to learn the correspondence between names in source program and given by *FSM Verifier*. The correspondence is following:

- *s1* is start state;
- *s13* is “*1. Insert card*”;
- *s9* is “*2. Enter PIN*”;
- *s7* is “*13. Card return*”.

This counter-example corresponds the following situation:

1. *ATM* starts working. *AClient* state machine goes to state “*1. Insert card”*.

2. User inserts a card (event *e6*).
3. *AClient* state machine goes to state "*2. Enter PIN"*.
4. User pushes the button "Cancel".
5. *AClient* state machine goes to state "*2. Card return"*.
6. User removes the card (event *e7*).
7. *AClient* state machine goes to state "*1. Insert card"*.

So this counter-example is equivalent to counter-example generated by *UniMod.Verifier*.

4.5. Autonomous verifiers

4.5.1. *CTL Verifier*

General description

CTL Verifier tool [63, 77, 84] does not use any other verifiers. In this tool the algorithm of *CTL* formulae verification was implemented. There is no need to convert the automata-based program to the input language of any verifier. However it is required to convert the program into the *Kripke* model which is verified by *CTL Verifier*. The result of verification will be also expressed in terms of *Kripke* model. Therefore there is a counter-example conversion in this method.

Atomic states extraction

Consider *Label* marking set ($Label \subseteq S \times AP$) for *Kripke* model $M = (S, R, Label)$. Instead of $R(s, t)$ we will write $s \to t$.

Consider program, which model is set by state machines system which interact by nesting. We need to convert this model to a single *Kripke* model. This model must describe the whole behavior of source system.

Firstly for set of state machines topological sorting with respect nesting is performed. This relationship must not have cycles. Otherwise the model will have unlimited size. The *Kripke* model is generated inductively for each state machine in system. The state machines are processed in order formed by topological sorting. This order provides that in time of processing of each state machine all of nested machines will be processed.

Method of full transition graph

Method described in this section is the most expressive in the sense that it provides verification of properties which include all of the aspects of automata-based programs behavior. It allows working with system of parallel state machines. We will consider this case.

Let $\{Y_1, Y_2, \ldots\}$ be the set of names for all states, $\{e_1, e_2, \ldots\}$ be the set for events, $\{x_1, x_2, \ldots\}$ be the set of input effects, $\{z_1, z_2, \ldots\}$ be the set of output effects, *Names* be the set of state machines' names, *InState*, *InEvent* and *InAction* be control atomic propositions for easy recognition vertices which are created from states, events and output effects.

Then by definition put the set of atomic propositions

$$AP = \{Y_1, Y_2, \ldots\} \cup \{e_1, e_2, \ldots\} \cup \{x_1, x_2, \ldots\} \cup \{z_1, z_2, \ldots\} \cup$$

$$\cup \{InState, InEvent, InAction\} \cup Names.$$

Now construct *Kripke* model in parts. Firstly construct parts such that they correspond to states of state machine. We must process the output effects and the nested into these states state machines. After that adding of information about transitions into the model. At first step put set S be equal to a set of the source state machine's states and for each s state add two marks: (s, s) и $(s, InState)$ into the *Label* relationship.

After that for each s state perform following operation. Let s contains the $z_{s[1]}, \ldots, z_{s[u]}$ output effects such that $z_{s[1]}, \ldots, z_{s[u]}$ are true under the entrance of state s. Construct $\{r_1, \ldots, r_u\}$ states and transitions: $r_1 \to r_2, \ldots, r_{u-1} \to r_u$, $r_u \to s$. Add $(r_k, z_{s[k]})$, $(r_k, InAction)$ marks into *Label* relationship for each $k = 1, \ldots, u$. In the sequel each transition going to the state s we redirect to state r_1.

Example is shown in fig. 4.10.

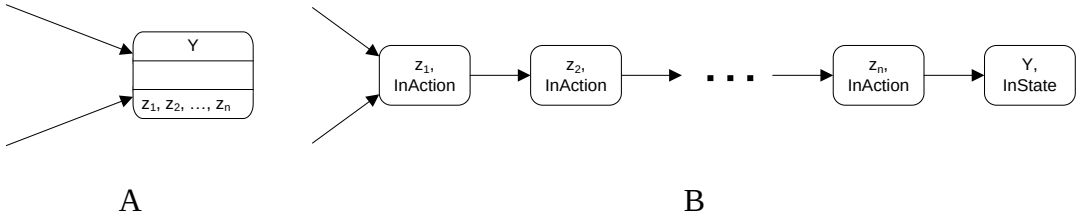


Fig. 4.10. A state with the output effects
before conversion (a) and after conversion (b)

We shall say that this operation is called *output variables and states division*.

Let state machines $A_{Y,1}$, $A_{Y,2}$, …, $A_{Y,v}$ are nested into the state Y of the external state machine (for these state machines the *Kripke* model has already constructed by the inductive hypothesis). Let us construct the *Kripke* model for the state Y. Add the *Kripke* model to model for the state Y of all of the state machines $A_{Y,1}$, $A_{Y,2}$, …, $A_{Y,v}$ (copy all of the states, transitions and marks of nested state machines' models to the external state machine's *Kripke* model).

Add a transition from terminal state of the state machine $A_{Y,i}$ to the $A_{Y,i}$ for $i = 1, \ldots, v - 1$.Add a transition from terminal state of the state machine $A_{Y,v}$ to the state Y. Example is shown in fig. 4.11.

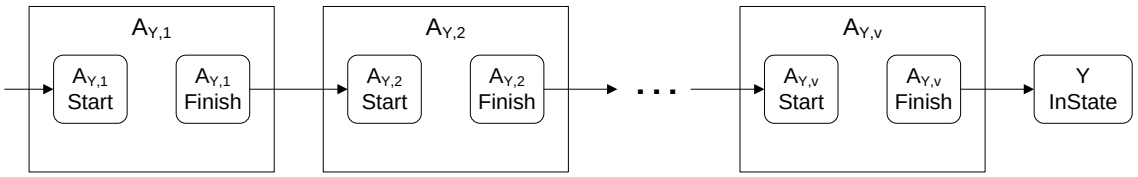


Fig. 4.11. Nested state machines processing

Note: There are might be several copies of one state machine nested into another state machine. In this case for each copy a separate *Kripke* model is created.

This algorithm will be illustrated by the example of ATM model. This model consists of two state machines: *AClient* and *AServer*, *AServer* is nested into *AClient*. Conection scheme is represented on fig. 4.4. Transition graph of *AClient* state machine is represented in fig. 4.5. Transition graph of *Aserver* state machine is represented in fig. 4.6. For this system the Kripke model is too large. So we represent it if fig. 4.12 in a simplified form.

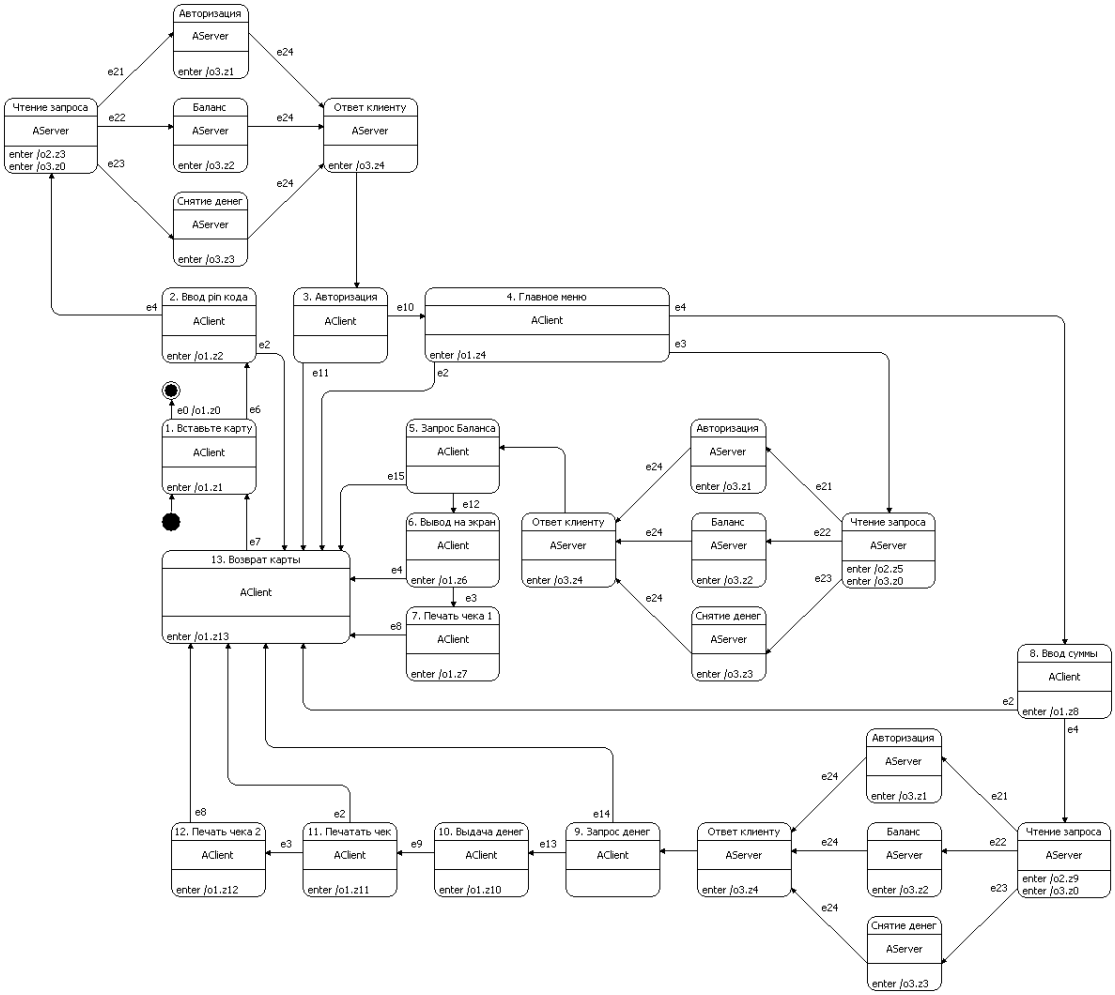


Рис. 4.12. Transition graph of *ATM* model

After states we need to process transitions. Consider set V of all Boolean variables in the source state machine. It consists of input effects and output effects. For each state p and binary sequence α such that can be assigned to elements of V. Define the scenario such that it must be happen in case that value of V is α. This scenario can be described in natural language the following manner. In the state p happened following events: $e_{i[1]}, \ldots, e_{i[s]}$. And input variables $x_{j[1]}, \ldots, x_{j[t]}$ (and only they) were true. After that $z_{k[1]}, \ldots, z_{k[u]}$ output effects were invoked and the state machine went to the state q. For each scenario like this (call it r) create addition states: $\{r_e, r_1, \ldots, r_u\}$, additional transitions: $p \to re$, $re \to r1$, $r1 \to r2$, ..., $ru–1 \to ru$, $ru \to q$ and add marks $(r_e, e_{i[i^*]})$, $(r_e, x_{j[j^*]})$ $(r_e, InEvent)$, $(r_{k^*}, z_{k[k^*]})$, $(r_{k^*}, InAction)$ for each $i^* = 1, \ldots, s$, $j^* = 1, \ldots, k$ and $k^* = 1, \ldots, u$. into *Label*.

Conversion of state machine into *Kripke* model is illustrated by the example of *ATrig* state machine. *Atrig* emulates operation of *RS*-trigger [85] (fig. 4.13, 4.14).

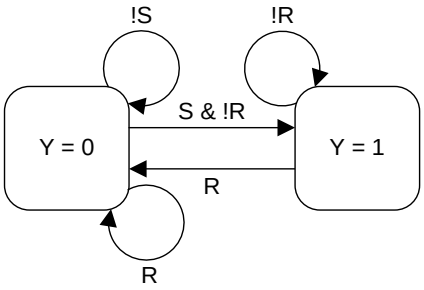


Рис. 4.13. Transition graph of *ATrig*

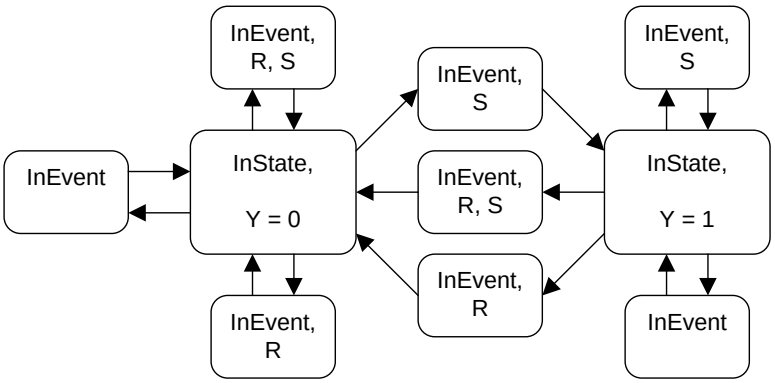


Рис. 4.14. Kripke model for *ATrig* state machine

Method of reduced transition graph

In this method by definition put

$$AP = \{Y_1, Y_2, \ldots\} \cup \{e_1, e_2, \ldots\} \cup \{x_1, x_2, \ldots\} \cup \{!x_1, !x_2, \ldots\} \cup$$

$$\cup \{z_1, z_2, \ldots\} \cup \{InState, InEvent, InAction\} \cup Names.$$

Start like method of full transition graph. Let S is a set of states of source state machine. For each $s \in S$ add two marks: (s, s) and (s, *InState*) into the *Label* relation. As described in method of full transition graph perform division of output variables and states. Separate states from nested state machines.

Processing of transitions. Consider the following set:

$$\{x_1, !x_1;\ x_2, !x_2;\ x_3, !x_3; \ldots\}.$$

We can say that it is a set of all the literals of input variables. We should distinguish between symbols "¬" and "!". First of them means performing of logical negation. Second means symbol (part of "$!x_i$" string). Put $h_{j[j^*]} = x_{j[j^*]}$ or $!x_{j[j^*]}$. Then for each transition r of source state machine such that it leads for state p to q state with following mark:

$$e_i \,\&\, h_{j[1]} \,\&\, h_{j[2]} \,\&\, h_{j[3]} \,\&\, \ldots \,\&\, h_{j[m]} / z_{i[1]}, \ldots, z_{i[n]}$$

Add $\{r_e, r_1, \ldots, r_n\}$ states and

$$p \to r_e,\ r_e \to r_1,\ r_1 \to r_2,\ \ldots,\ r_{n-1} \to r_n,\ r_n \to q$$

transitions into the model.

Add following marks into the *Label* relation: (r_e, e_i), $(r_e, \mathit{InEvent})$, $(r_k, z_{i[k]})$, $(r_k, \mathit{InAction})$ for each $k = 1, \ldots, n$ and $(r_e, h_{i[1]})$, $(r_e, h_{i[2]})$, …, $(r_e, h_{i[m]})$.

Example of this conversion is shown in fig. 4.15.

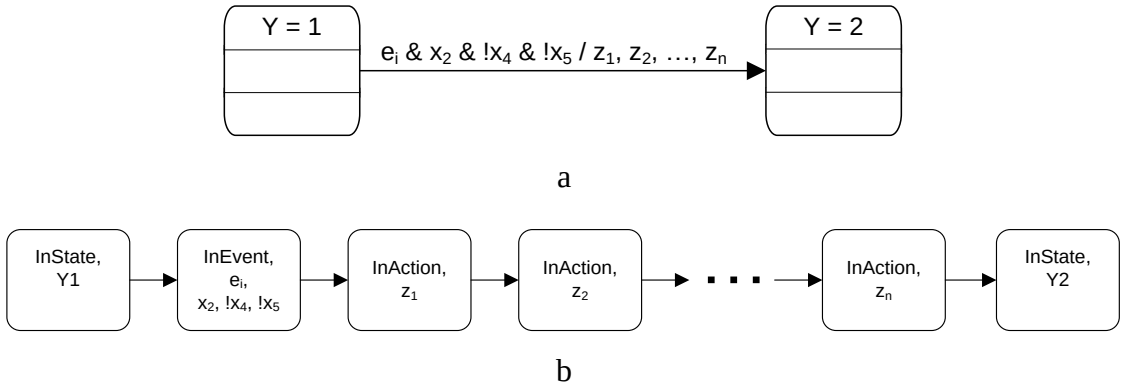


Fig. 4.15. Transition before conversion (a) and after (b)

At the end of the processing add mark with atomic proposition corresponding state machine's name into *Label* relation, just like previous method.

This method is demonstrated by the example of the state machine of lift doors control (fig. 4.3). Construct the *Kripke* model for this state machine. In fig. 4.16 the *Kripke* model is shown.

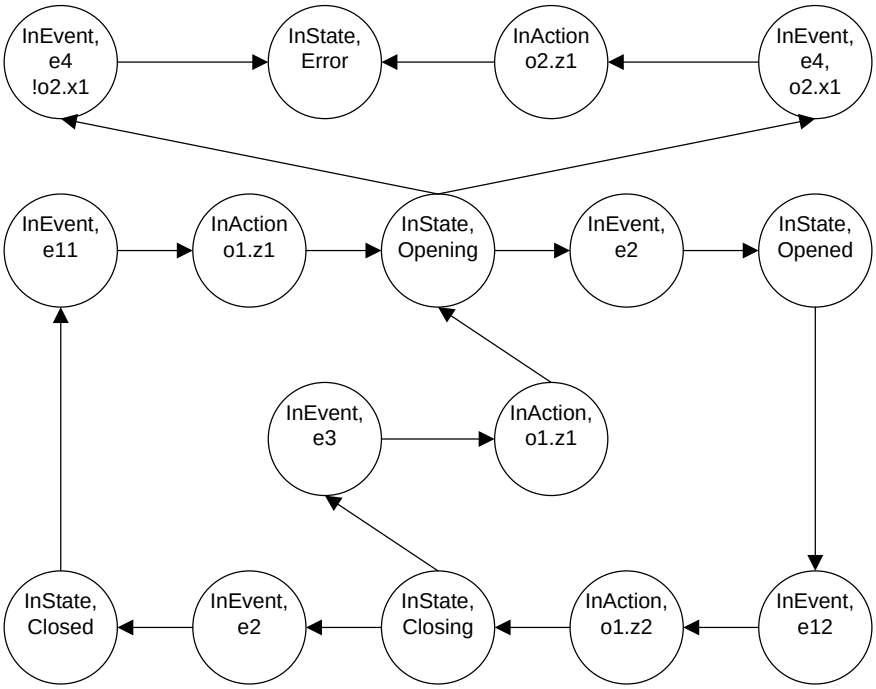


Fig. 4.16. *Kripke* model

Example of *Kripke* model for system of interacting state machines is shown in fig. 4.17, 4.18.

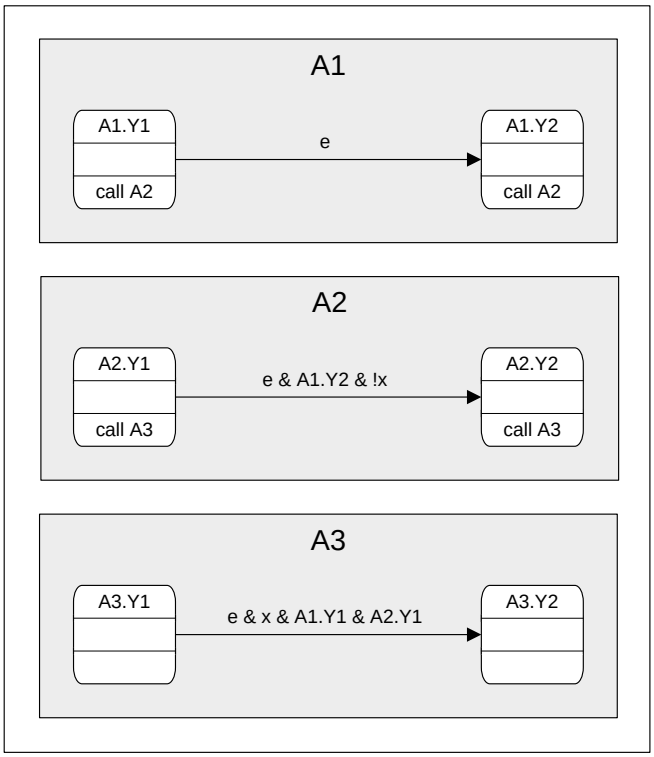


Рис. 4.17. System of interacting by nesting state machines with conditions of external state machine's states

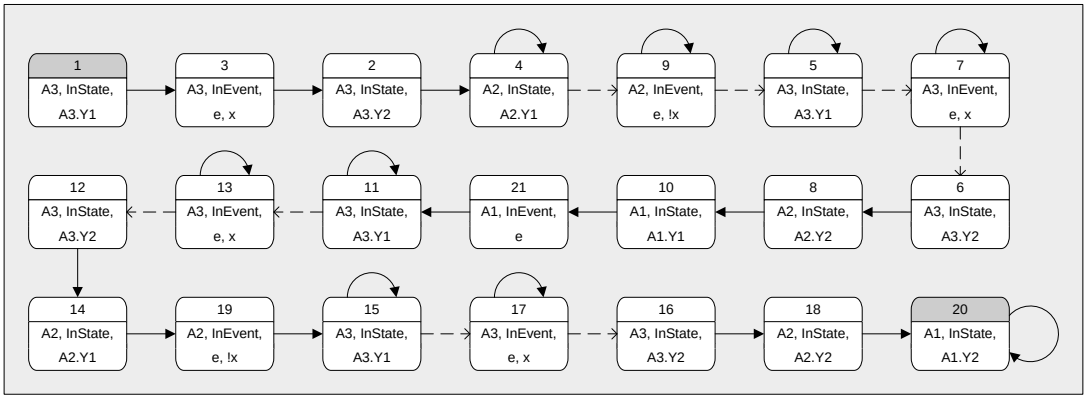


Рис. 4.18. *Kripke* model for this system

Checking of *CTL* formula

Semantics of *CTL* in this method slightly differs from standard: Before verification it should be cast to the canonical form. Firstly all the double negations should be deleted (by the changing expressions $\neg\neg f$ to f). After this write two negations before all the input variables in formula: $h_i \rightarrow \neg!h_i$. After these modifications we can verify *CTL* formula using methods for *CTL* language. The reason of this modification is following: we must ensure that every reference to unchecked in this transition input variables.

Verified properties

As predicates in *CTL Verifier* can be used following conditions:

- State machines *A* is in state *s*;
- event *e* is processing;
- input variable *x* is true or false;
- output effect *z* is invoked;

CTL Verifier uses only following temporal operators: *EX*, *EG*, *EU*. Other temporal operators can be expressed by these ones using the following terms:

- *AX g = !EX !g*;
- *EF g = 1 EU g*;
- *AF g = !EG !g*;

- *AG g = !EF !g = !(1 EU g)*;
- *f AU g = !((!g EU !(f || g)) || EG !g)*.

Counter-example conversion

After work of verifier we should determine satisfiability of CTL formulae. Scenario for every sub formula is a path in the *Kripke* model. This path illustrates validity or invalidity for this sub formula. The problem is to represent the scenario in the *Kripke* model in the source automata-based system.

For described above method this operation is performed unambiguously. Proof. Consider *Kripke* model states such that they contain atomic proposition *Y* = or auxiliary atomic proposition *InState*. These states are converted to corresponding source system's states unambiguously. Path between these states is a "line" of the event and output effects. Any of these intermediate states unambiguously determines the main state that starts this line. The events are restored unambiguously from this "line". The output effects are also restored unambiguously from this "line".

Tool description

The input for *CTL Verifier* is a state machines system, which described in a text format. This format is developed specially for this tool and is not used by other programs.

The input file also contains CTL formula with specification. CTL formula must be written inductively by construction. For example, "*EG* $!e_1$" formula is written in listing 4.

Listing 4.15. *EG* $!e_1$

```
[Properties]
f1 = e1
f2 = !f1
f3 = $EG f2
```

Verification is performed by the following command:

```
CTLVerif.exe <input file>
            [ <output file> [<output folder>] ]
```

CTL Verifier works only under *Windows*.

The result is following information:

- List of predicates that used in the constructed *Kripke* model;
- The *Kripke* model. All the atomic states are enumerated.
- For each formula the list of states such that formula is met in these states. If the formula contains the temporal operator and is proved by a cycle of states, the list of states is printed. For example, 1 34 35 (3 38 39 5 109 110 8 91 92 15 85 86 20 82 83);
- If the output folder is specified, then the counter-example in terms of source system is printed. One file with counter-example for each formula.

ATM-model verification

We need to convert the *UniMod*-program to format of *CTL Verifier*. The converting algorithm was developed for *CTL Verifier*. This algorithm makes changes to system such that state names changing, union of all the termination states to one, et cetera.

Result of conversion of the ATM model is *Bankomat.dat* file. The state "*10. Payment*" state of *AClient* state machine got *s10* name. Section with property Σ of this file is represented in listing 4.16.

Listing 4.16. The property Σ

```
[Properties]
; !(!e10 EU s10)
f1 = e10
f2 = !f1
f3 = s10
f4 = f2 $EU f3
f5 = !f4
```

Start the verifier:

```
CTLVerif.exe Bankomat.dat out.txt out
```

Information of Kripke model is stored in *out.txt* file. In the *out* folder *f1*, *f2*, *f3*, *f4* and *f5* files were created. File *f5*contains the result of verification for property Σ. The first four lines of this file are shown in listing 4.17.

Listing 4.17. Result of verification of property Σ

```
$   1: AClient InState s0
   28: AClient InAction o1.z1
   29: AClient InState s12
   30: AClient InAction o1.z7
```

The first line reports that formula *f5* is true in the start state (*s0*) of the main state machine (*AClient*). state *s0* corresponds state *s1* in source program. Therefore the source program satisfies the property Σ.

Verify the Ω property. The CTL formula is *AF s10*. Convert it: *AF s10 = !EG !s10*. Write it to the *Bankomat.dat* file (listing 4.18).

Листинг 4.18. The property Ω

```
[Properties]
; AF s10 = ! EG !s10
f1 = s10
f2 = !f1
f3 = $EG f2
f4 = !f3
```

Run the verifier:

```
CTLVerif.exe Bankomat.dat out.txt out
```

There are only three states that satisfy formula *f4*. There is no start state among them.

Listing 4.19. The result of property Ω verification

```
    4: AClient InAction o1.z10
    5: AClient InState s10
  106: AClient e13 InEvent
```

If the start state does not satisfy the formula *f4*, then it satisfies its negation. The negation of formula *f4* is formula *f3*. So prove for the formula *f3* is a counter-example for the formula *f4*. This counter-example is shown in listing 4.20.

Listing 4.20. Counter-example for the property Ω

```
[1]
$   1: AClient InState s0
   89: * AClient InEvent
```

```
    28: AClient InAction o1.z1
    29: AClient InState s12
   108: AClient e0 InEvent
   109: AClient InAction o1.z0
Cycle:
%  80: AClient InState s5
```

This counter-example in terms of source program is shown in listing 4.21.

Листинг 4.21. Counter-example of the property Ω in terms of source program

```
$   1: AClient InState s1
    89: * AClient InEvent
    28: AClient InAction o1.z1
    29: AClient InState "1. Insert card"
   108: AClient e0 InEvent
   109: AClient InAction o1.z0
Cycle:
%  80: AClient InState s2
```

This counter-example corresponds the following scenario:

1. ATM starts working. The sstate machine *AClient* goes to "*1. Insert card"* state.
2. User turns it off (event *e0*).
3. The state machine *AClient* goes to the terminal state and stays in this state forever.

So user never gets the money.

4.5.2. *Automata Verificator*

General description

In this verifier [64] the algorithm of double depth-first search is implemented. This algorithm allows verifying LTL formulae described using *Buchi* automaton. The feature of this tool is multithreading implementation of double depth-first search algorithm.

Automata Verificator works with programs that developed using *UniMod*.

Atomic states extraction

In considered verifier the atomic state is defined as a set of current states of state machines in the system.

Verified properties

The predicates list is similar to the predicates list of *UniMod.Verifier*:

- *wasEvent*;
- *wasInState*;
- *isInState*;
- *cameToFinalState*;
- *wasAction*;
- *wasFirstAction*;

The meaning of these predicates is similar to *UniMod.Verifier*. User can use names of states, events and actions of source program even if they contain space symbols and other special symbols.

In addition user can create his own predicates. For this effect user should develop a class in *Java* that contains a method with annotation "*@Predicate*".

Counter-example conversion

There is no need to convert counter-example.

Tool description

Automata Verificator was developed as *Java*-classes and is not represented as a program that can be executed from command line. So authors developed a class that allows to run this program from command line and built with all the classes of *Automata Verificator*.

This tool works with *XML*-files created using *UniMod*. Let *A.xml* is a file with automata-based program, *A1* is a main state-machine, and *F(wasEvent(p.e1))* is a formula. The following command runs the verification:

```
java –jar verifier.jar A.xml A1 "F(wasEvent(p.e1))"
```

The result of work is Buchi automaton that generated from LTL formula and the result of verification. If program satisfies the LTL formula then message "Verification successful" is printed. Otherwise the counter-example is printed.

Listing 4.22. Counter-example that generated by *Automata Verificator*

```
LTL: F(isInState(AClient, AClient["10. Payment"]))
initial 0
BuchiNode 0
→[!isInState(AClient, 10. Payment)] 0
Accept set 0 [0]

DFS 2 stack:
→["<"13. Card return", "s1">", 0, 0]→["<"1. Insert
card", "s1">", 0, 0]
DFS 1 stack:
→["<"s1", "s1">", 0, 0]→["<"1. Insert card", "s1">",
0, 0]
→["<"2. Enter PIN", "s1">", 0, 0]→["<"3.
Authorization", "s1">", 0, 0]
→["<"4. Main menu", "s1">", 0, 0]→["<"13. Card
return", "s1">", 0, 0]
```

The counter-example is concatenation of DFS 1 stack and DFS 2 stack. Before concatenation we should exclude the fist state of DFS 2 stack because it doubles last state of DFS 1 stack.

Each system's state is written in square brackets. It contains the following information:

1. Set of current system states.
2. Number of current Buchi automaton state.
3. Set of accept states of Buchi automaton.

ATM-model verification

Verify property Σ. Write assertion that event *e10* was happened:

```
wasEvent(p3.e10)
```

p3 is an event provider that generated event *e10* (fig. 4.5). Write assertion that state machine *AClient* is in state "*10. Payment*":

```
isInState(AClient, AClient[\"10. Payment\"])
```

Run verification :

```
java –jar verifier.jar Bankomat.xml AClient
"!U(!wasEvent(p3.e10), isInState(AClient, AClient[\"10.
Payment\"]))"
```

The result of verification is shown in listing 4.23.

Listing 4.23. The result of verification of property Σ

```
LTL: !U(!wasEvent(p3.e10), isInState(AClient,
AClient["10. Выдача денег"]))
initial 1
BuchiNode 0
 →[true] 0
BuchiNode 1
 →[!wasEvent(e10)] 1
 →[isInState(AClient, 10. Выдача денег)] 0
Accept set 0 [0]

Verification successful
```

As expected, there is no errors.

Verify property Ω:

```
java –jar verifier.jar Bankomat.xml AClient
"F(isInState(AClient, AClient[\"10. Выдача денег\"]))"
```

The result is shown in listing 4.22. This counter-example corresponds the following scenario:

1. ATM starts working. *AClient* goes to "*1. Insert card"* state.
2. User inserts a card (*e6* event).
3. *AClient* goes to "*2. Enter PIN"* state.
4. *AClient* goes to "*3. Authorization"* state.
5. User enter PIN.
6. *AClient* goes to "*4. Main menu"* state.
7. User pushes the cancel button.
8. *AClient* goes to "*13. Card return"* state.
9. User takes his card back.
10. *AClient* goes to "*1. Insert card"* state.

This counter-example is correct but it differs from found by the other verifiers' counter-examples.

Conclusion

The main problem in the development of software systems is that validation implemented by a software system is extremely complex. While designing such systems the most of the time is taken on analysis and debugging rather than on writing code. There are many methods validation programs, and they correspond to different classes of them. In this book we have considered a class of systems that are critical for safety, in validating the application is inadmissible incomplete induction. Even if you check the correctness of the basic scenarios of the system, it would still be insufficient for the recognition system robust. The most profound and difficult to detect errors can be tolerated in the design phase. Therefore the book is devoted to methods of testing, which guarantee the correctness of any behavior of the system (or its model).

However, in order to get one hundred percent proof of the correctness of the program, you must expend a tremendous effort. The method of formal verification does not apply to large and complex programs. Specialists in software development and verification say that they have finished working with the 15-page report while the program takes a half page. Therefore it is necessary to observe a certain balance between the expected reliability of the evidence and by how convenient and efficient it can be obtained.

Most effectively, this balance is observed in the method of testing models, which exists about 30 years. Its popularity is due, on the one hand, highly automated proof of correctness, and little involvement of the developer in the verification process and on the other - the ability to clearly point out the error in the case when the developer has to deal with the wrong prototype. The relevance of research in the field of model validation is also confirmed by the award in 2007 founders of this approach, Turing Award, Edmund Clarke, Allen Emerson and Joseph Sifakisu. Model checking has amazing achievement, especially in the verification of hardware. For example, it is known that model checking would have found a bug in the Pentium I processors and would prove faithful to fix it by Intel Corporation. Since then, Intel is one of the most systematically applying this technology corporations.

The main problem that arises when using model checking is a combinatorial explosion in the state space model. The first model-checking algorithms could work with a number of states of about 40 000. Over time, the size of transition systems that allow effective verification, have increased significantly. For the implicit representation of the problem was initially established technology symbolic model checking, based on ordered binary allow diagrams, and then - testing models of limited depth (bounded model checking), which increased numerical limits of testing models by several orders of magnitude. Significant gains were also achieved through the development of methods for the reduction of partial orders, methods of use of symmetry in parallel systems and methods of abstraction to scan models. In the present model-checking technology can verify the program, approximately of 10 000 lines of code. The most impressive example demonstrating the ability of symbolic model checking is a verification protocol cache coherency bus standard IEEE Futurebus + (IEEE 896.1-1991 standard.) Although the development of this protocol was initiated in 1988, all previous attempts to justify its validity were based solely on informal reasoning. In the summer of 1992 research team from Carnegie Mellon University built an exact model of the protocol in the language of SMV, and then, applying the verification system SMV, showed that the system satisfies a formal specification of the transitions. They were also able to detect a number of previously unnoticed errors and localize potentially faulty parts in the draft protocol. Verification is important for software systems with complex behavior. For this class of systems, technology development and verification should be selected already at the design stage.

A good choice for such problems is the automata-based programming technology. First of all, automata-based programming makes it possible to successfully decompose the control logic and computation in the establishment phase of a software system. It allows the use of a visual graphical notation to describe the complex behavior, thus making the system less error prone. On the other hand, when applying model checking to an automaton program construction of the model program for its specification is greatly simplified compared to traditional approaches to programming. When using an automaton approach the model of a program, suitable for verification is already based on the stage of design. Finally, the automaton programming can decompose the verification program for verifying its behavior (management) model, which is performed by model checking, and independent verification of atomic computing effects. If the system is successfully designed based on automata approach, and the input and output

effects are small, relatively simple lines of code, they can be checked by the formal verification of operating a dependency output of the algorithm from its input. Output actions in automata programs can also be verified on the basis of the approach outlined by Gnesi and Mazzanti [59].

Examples of verification tools in more detail are in the books [93, 94]. Questions of general type of program verification are discussed in the books [17, 18].

The first work on the verification automata programs, appeared in 2006 first [87], and then [70]. Research on automata verification of programs is going on at St. Petersburg State University of Information Technologies, Mechanics and Optics [53, 63, 77] and in Yaroslavl Demidov State University. [95, 96]. On the subject begin to defend his [97, 98] PhD theses have been already defended on this theme.

The authors suppose that this book will draw attention to automata programming, as approach oriented to verification.

Bibliography

1. *Katoen J.-P.* Concepts, Algorithms, and Tools for Model Checking. Lehrstuhl für Informatik VII, Friedrich-Alexander Universität Erlangen-Nürnberg. Lecture Notes of the Course "Mechanised Validation of Parallel Systems" (course number 10359). 1998/1999. http://fmt.isti.cnr.it/~gnesi/matdid/katoen.pdf
2. *Shalyto A. A.* Switch-tehnologiya. Algoritmizatsiya i programmirovanie zadach logicheskigo upravleniya. SPb.: Nauka, 1998. http://is.ifmo.ru/books/switch/1
3. *Liggesmeyer P., Rothfelder M., Rettelbach M., Ackermann T.* Qualitдtssicherung Software-basierter technischer Systeme – Problembereiche und Lцsungsansдtze // Informatik Spektrum. 21: 249–258, 1998.
4. *Baier C., Katoen J.-P.* Principles of Model Checking. The MIT Press, 2008. http://is.ifmo.ru/books/_principles_of_model_checking.pdf
5. *Harel D., Pnueli A.* On the Development of Reactive Systems // Logics and Models of Concurrent Systems. V. F-13 of NATO ASI Series. NY, Springer-Verlag, 1985. http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/ReactiveSystems.pdf
6. *Sinitsyn S. V., Nalyutin N. Yu.* Verifikatsiya programmnogo obespecheniya. M.: BINOM, 2008.

7. *ISO/ITU-T.* Formal Methods in Conformance Testing. Draft International Standard, 1996.

8. *Beizer B.* Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley, 1995.

9. *Beck K.* Test-driven Development by Example. Addison-Wesley Professional, 2002.

10. *Freeman S., Pryce N., Mackinnon T., Walnes J.* Mock Roles, not Objects. http://www.jmock.org/oopsla2004.pdf

11. *Umrigar Z., Pitchumani V.* Formal verification of a real-time hardware design / Proceedings of the 20th Design Automation Conference, 1983. http://portal.acm.org/ft_gateway.cfm?id=800667&type=pdf&CFID=112534228&CFTOKEN=12780503

12. *Guts A. K.* Matematicheskaya logika i teoriya algoritmov. Omsk: Naslediye, Dialog-Sibir, 2003.

13. *Hoare C. A. R.* An axiomatic basis for computer programming // Communications of the ACM. 1969/12, pp. 576–583. http://se.ethz.ch/teaching/ss2005/0250/readings/Axiomatic_Basis.pdf

14. *Owicki S., Gries D.* An axiomatic proof technique for parallel programs // Acta Informatica. 1976/6, pp. 319–340. http://www.springerlink.com/content/x12541v1q15570n2/

15. *Pnueli A.* The temporal logic of programs / 18th IEEE Symposium on Foundations of Computer Science. 1977, pp. 46–57. http://www.inf.ethz.ch/personal/kroening/classes/fv/f2007/readings/focs77.pdf

16. *Thayse A.* Approche logique de l'intelligense artificielle. Donod, 1988.

17. *Clarke E., Grumberg O., Peled D.* Model Checking. The MIT Press, 1999.

18. *Karpov Yu. G.* Model Checking: verifikatsiya parallelnyh i raspreselennyh programmnyh system. SPb.: BHV-Peterburg, 2010.

19. *West C. H.* Applications and limitations of automated protocol validation / 2nd Symposium on Protocol Specification, Testing and Verification. 1982, pp. 361–371.

20. *Clarke E. M., Emerson E. A.* Synthesis of synchronization skeletons for branching time logic // Logic of Programs. LNCS 131. 1981, pp. 52–71. http://www.springerlink.com/content/w1778u28166t2677/

21. *Apt K. R., Kozen D. C.* Limits for the automatic verification of finite-state concurrent systems // Information Processing Letters. 1986/22, pp. 307–309.

22. *Konev B. Yu.* Vvedenie d modelirovanie i verifikatsiyu apparatnyh sistem.
http://logic.pdmi.ras.ru/~kulikov/verification/10.pdf

23. *Lichtenstein O., Pnueli A., Zuck L.* The glory of the past // Logics of Programs. LNCS 193. 1985, pp. 196–218.
http://www.springerlink.com/content/7681m36026888082/

24. *Smelyanskiy R. L.* Primenenie temporalnoy logiki dlya spetsifikatsyy poveleniya programmnyh sistem // Programmirovanie. 1993. № 1, p. 3–28.

25. *Sistla A. P., Clarke E. M.* The complexity of propositional linear temporal logics // Journal of the ACM. 32(3). 1985, pp. 733–749.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.178&rep=rep1&type=pdf

26. *Clarke E. M., Draghicescu I. A.* Expressibility results for linear time and branching time logics // Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency. LNCS 354. 1988, pp. 428–437.
http://www.springerlink.com/content/5n2702u432119wx8/

27. *Kripke S. A.* Semantical considerations on modal logic // Acta Philosophica Fennica 16: 83–94, 1963.
http://condor.wesleyan.edu/courses/2007s/phil390/01/e-texts/Kripke/Kripke,%20Semantical%20Considerations%20on%20Modal%20Logic.pdf

28. *Emerson E. A., Halpern J. Y.* “Sometimes” and “not never” revisited: on branching versus linear time temporal logic // Journal of the ACM. 33(1). 1986, pp. 151–178.
http://www.cs.cmu.edu/~emc/15-820A/reading/p127-emerson.pdf

29. *Bryant R.* Graph-based algorithms for boolean function manipulation // IEEE Transactions on Computers. C-35. 1986/8, pp. 677–691.
http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf

30. *Mironov A. M.* Verifikatsiya program metodom Model Checking.
http://intsys.msu.ru/staff/mironov/modelchk.pdf

31. *Mironov A. M.., Zhukov D. Yu..* Matematicheskaya model I metody verifikatsii programmnyh sistem// Intellektualnye sistemy. T. 9. 2005. Vyp. 1–4, p. 209–252.
http://www.intsys.msu.ru/magazine/archive/v9(1-4)/mironov-209-252.pdf

32. *Wegener I.* Branching Programs and Binary Decision Diagrams. SIAM monographs on discrete mathematics and applications, 2000.

33. *Clarke E. M., Grumberg O., Long D.* Verification tools for finite-state concurrent systems // A Decade of Concurrency–Reflections and Perspectives. LNCS 803. 1993, pp. 124–175.
http://www-2.cs.cmu.edu/~modelcheck/ed-papers/VTfFSCS.pdf

34. *McMillan K. L.* Symbolic Model Checking. Kluwer Academic Publishers, 1993.
http://cadence.com/cadence/cadence_labs/Documents/mcmillan_CMU_1992_Symbolic.pdf

35. *Clarke E. M., Emerson E. A., Sistla A. P.* Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Transactions on Programming Languages and Systems. 8(2). 1986, pp. 244–263.
http://www.cs.cmu.edu/~modelcheck/ed-papers/AVoFSCSU.pdf

36. *Kropf T.* Hardware Verifikation. Habilitation thesis. University of Karlsruhe, 1997.

37. *Tarjan R.* Depth-first search and linear graph algorithms // SIAM Journal on Computing. Vol. 1 (1972). No. 2, pp. 146–160.
http://rjlipton.files.wordpress.com/2009/10/dfs1971.pdf

38. *Eppstein D.* Design and Analysis of Algorithms. Lecture notes for 1996.
http://www.ics.uci.edu/~eppstein/161/960220.html

39. *Alur R., Courcoubetis C., Dill D.* Model-checking in dense real-time // Information and Computation. 104: 2–34, 1993.
http://www.cis.upenn.edu/~alur/Lics90D.ps

40. *Alur R., Henzinger T. A.* Real-time logics: Complexity and expressiveness // Information and Computation. 104: 35–77, 1993.
http://www.cis.upenn.edu/~alur/Lics90H.ps

41. *Alur R., Henzinger T. A.* Back to the future: towards a theory of timed regular languages / IEEE Symp. on Foundations of Computer Science. 1992, pp. 177–186.
http://www.cis.upenn.edu/~alur/Focs92.ps

42. *Yovine S.* Model checking timed automata // Embedded Systems. LNCS 1494, 1998.
http://www-verimag.imag.fr/~yovine/articles/embedded98.ps.gz

43. *Petri C. A.* Kommunikation mit Automaten. Ph. D. Thesis. University of Bonn, 1962.

44. *Esparza J., Nielsen M.* Decidability issues for Petri nets – a survey // Bulletin of the EATCS, 1994.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.3965

45. *Boucheneb H., Hadjidj R.* Model checking of time Petri nets. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6973&rep=rep1&type=pdf

46. *Holzmann G. J.* The Model Checker SPIN // IEEE Transactions on software engineering. 1997, V. 23, I. 5. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.7596&rep=rep1&type=pdf

47. *Dijkstra E. W.* Guarded commands, non-determinacy and formal derivation of programs // CACM. 18(8), 1975. http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD418.PDF

48. *Dolev D., Klawe M., Rodeh M.* An O(*n* log *n*) unidirectional distributed algorithm for extrema finding in a circle // Journal of Algorithms. 1982/3, pp. 245–260. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.129.7495&rep=rep1&type=pdf

49. *McMillan K. L.* The SMV System. Technical Report CS-92-131. Carnegie-Mellon University, 1992. http://www.comp.nus.edu.sg/~cs3234/smvmanual.pdf

50. *Chan W., Anderson R. J., Beame P., Burns S., Modugno F., Notkin D., Reese J. D.* Model checking large software specifications // IEEE Transactions on Software Engineering. 24(7). 1998, pp. 498–519. http://www.cs.washington.edu/homes/beame/papers/fse.pdf

51. *Polikarpova N. I., Shalyto A. A.* Avtomatnoe programmirovanie. Spb.: Piter, 2010. http://is.ifmo.ru/books/_book.pdf

52. *Hopkroft J., Motwani R., Ullman J.* Introduction to Automata Theory, Languages and Computation. Addison Wesley, 2006.

53. Otchet po kontraktu o verifikatsii avtomatnyh program. Etapy 1, 2. 2007.

http://is.ifmo.ru/verification/_2007_01_report-verification.pdf
http://is.ifmo.ru/verification/_2007_02_report-verification.pdf

54. Otchet o patentnyh issledovaniyah po 1-mu etapu kontrakta o verifikatsii avtomatnyh programm. http://is.ifmo.ru/verification/_2007_01_patent-verification.pdf

55. *Gnesi S., Mazzanti F.* A model checking verification environment for UML statecharts / Proceedings of XLIII Congresso Annuale AICA, 2005. http://fmt.isti.cnr.it/~gnesi/matdid/aica.pdf

56. *Volobuev V. N., Kalachinskiy A. V.* Opyt ispolzovaniya avtomatnogo podhoda pri razrabotke pogrammnogo obespecheniya system boevogo upravleniya // Sistemy upravleniya I obrabotki informatsii. Vyp. 18. 2009, p. 88–92.
http://is.ifmo.ru/works/_volobuev.pdf

57. *Remizov A. O., Shalyto A. A.* Verifikatsiya avtomatnyh program / Sbornik dokladov nauchno-tehnicheskoy konferentsii "Sostoyanie, problemy I perspektivy sozdaniyz korabelnyh informatsionno-upravlyaushih kompleksov. OAO "Kontsern Morinfosistema Agat". M.: 2010, c. 90–98.
http://is.ifmo.ru/works/_2010_05_25_verific.pdf

58. *Gurov V. S., Mazin M. A., Narvskiy A. S., Shalyto A. A.* Instrumentalnoye sredstvo dlya podderzhki avtomatnogo programmirovaniya // Programmirovanie. 2007. № 6, c. 65–80.
http://is.ifmo.ru/works/_2008_01_27_gurov.pdf

59. *Zakonov A., Stepanov O., Shalyto A.* GA-Based and Design by Contract Approach to Test Generation for EFSMs / Proceedings of IEEE East-West Design & Test Symposium (EWDTS`10). St. Petersburg. 2010, pp. 152–155.
http://is.ifmo.ru/works/_ewdts_2010_zakonov.pdf

60. *Klebanov A. A., Stepanov O. G., Shalyto A. A.* Primenenie shablonov trebovaniy k formalnoy spetsifikatsii I verifikatsii avtomatnyh programm / Trudy seminara "Semantika, specifikatsiya I verifikatsiya programm: teoriya i prilozheniya". Kazan, 2010, p. 124–130.
http://is.ifmo.ru/works/_2010-10-01_klebanov.pdf

61. *Barr M.* Real men program in C.
http://www.eetimes.com/General/DisplayPrintViewContent?contentItemId=4027479

62. *Kolskiy N. I.* Yazyk programmirovaniya vstroennyh system: svoboda vybora ili zhestkiy determinism? // Mir kompyuternoy avtomatizatsii: vstraivaemye komp'yuternye sistemy. 2010. № 4, p. 54–60.

63. *Vel'der S. E., Shalyto A. A.* Verifikatsiya avtomatnyh modeley metodom redutsirovannogo grafa perehodov // Nauchno-tehnicheskiy vestnik SPbGU ITMO. 2009. Vyp. 6(64), p. 66–77.
http://is.ifmo.ru/works/_2010_01_29_velder.pdf

64. *Egorov K. V., Shalyto A. A.* Metodika verifikatsii avtomatnyh programm // Informatsionno-upravlyayuschie sistemy. 2008. № 5, p. 15–21.
http://is.ifmo.ru/works/_egorov.pdf

65. *Kurbatskiy E. A.* Verifikatsiya programm, postroennyh na osnove avtomatnogo podhoda s ispol'zovaniem programmnogo sredstva SMV // Nauchno-tehnicheskiy vestnik SPbGU ITMO. Vyp. 53. Avtomatnoe programmirovanie. 2008, p. 137–144.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

66. *Lukin M. A., Shalyto A. A.* Verifikatsiya avtomatnyh programm s ispol'zovaniem verifikatora SPIN // Nauchno-tehnicheskiy vestnik SPbGU ITMO. Vyp. 53. Avtomatnoe programmirovanie. 2008, p. 145–162.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

67. *Gurov V. S., Yaminov B. R.* Verifikatsiya avtomatnyh programm pri pomoschi verifikatora UNIMOD.VERIFIER // Nauchno-tehnicheskiy vestnik SPbGU ITMO. Vyp. 53. Avtomatnoe programmirovanie. 2008, p. 162–176.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

68. *Roux C., Encrenaz E.* CTL May Be Ambiguous when Model Checking Moore Machines. UPMC LIP6 ASIM, CHARME, 2003.
http://sed.free.fr/cr/charme2003.ps

69. *Kuz'min E. V.* Ierarhicheskaya model' avtomatnyh programm // Modelirovanie i analiz informatsionnyh sistem. 2006. № 1, p. 27–34.
http://is.ifmo.ru/verification/_hamp.pdf

70. *Vinogradov R. A., Kuz'min E. V., Sokolov V. A.* Verifikatsiya avtomatnyh programm sredstvami CPN/Tools // Modelirovanie i analiz informatsionnyh sistem. 2006. № 2, p. 4–15.
http://is.ifmo.ru/verification/_cpnverif.pdf

71. *Vasil'eva K. A., Kuz'min E. V.* Verifikatsiya avtomatnyh programm s ispol'zovaniem LTL // Modelirovanie i analiz informatsionnyh sistem. 2007. № 1, p. 3–14.
http://is.ifmo.ru/verification/_LTL_for_Spin.pdf

72. *Vasil'eva K. A., Kuz'min E. V., Sokolov V. A.* Verifikatsiya avtomatnyh programm s ispol'zovaniem LTL.
http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf

73. *Kuz'min E. V., Sokolov V. A.* Modelirovanie, spetsifikatsiya i verifikatsiya avtomatnyh programm // Programmirovanie. 2008. № 1, p. 38–60.
http://is.ifmo.ru/download/2008-03-12_verification.pdf

74. *Kozlov V. A., Komaleva O. A.* Modelirovanie raboty bankomata. SPbGU ITMO, 2006.
http://is.ifmo.ru/unimod-projects/bankomat

75. *eVelopers Corporation homepage.* http://www.evelopers.com

76. *UniMod homepage*. http://unimod.sf.net

77. Otchet po kontraktu o verifikatsii avtomatnyh programm. Etapy 3, 4. 2008.
http://is.ifmo.ru/verification/_2007_03_report-verification.pdf
http://is.ifmo.ru/verification/_2007_04_report-verification.pdf

78. *Lukin M. A.* Verifikatsiya avtomatnyh programm. Bakalavrskaya rabota. SPbGU ITMO, 2007.
http://is.ifmo.ru/papers/_lukin_bachelor.pdf

79. *Lukin M. A.* Verifikatsiya vizual'nyh avtomatnyh programm s ispol'zovaniem instrumental'nogo sredstva SPIN. Magisterskaya rabota. SPbGU ITMO, 2009.
http://is.ifmo.ru/papers/_lukin_master.pdf

80. *Spin home page*. http://spinroot.com

81. *Yaminov B. R.* Avtomatizatsiya verifikatsii avtomatnyh UniMod-modeley na osnove instrumental'nogo sredstva Bogor. Bakalavrskaya rabota. SPbGU ITMO, 2007.
http://is.ifmo.ru/papers/_jaminov_bachelor.pdf

82. *Yaminov B. R.* Sravnenie metodov verifikatsii UniMod-modeley. Magisterskaya rabota. SPbGU ITMO, 2009.
http://is.ifmo.ru/papers/_jaminov_master.pdf

83. *Bogor home page*. http://bogor.projects.cis.ksu.edu

84. *Vel'der S. E., Shalyto A. A.* Metody verifikatsii modeley avtomatnyh programm // Nauchno-tehnicheskiy vestnik SPbGU ITMO. Vyp. 53. Avtomatnoe programmirovanie. 2008, p. 123–136. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

85. *Shalyto A. A.* Logicheskoe upravlenie. Metody apparatnoy i programmnoy realizatsii algoritmov. SPb.: Nauka, 2000.
http://is.ifmo.ru/books/log_upr/1

86. *Vel'der S. E., Bednyj Yu. D.* Universal'nyj infrakrasnyj pul't dlya bytovoy tehniki. SPbGU ITMO, *200*5.
http://is.ifmo.ru/projects/irrc/

87. *Vel'der S. E.* Vvedenie v verifikatsiyu avtomatnyh programm na osnove metoda model checking. Bakalavrskaya rabota. SPbGU ITMO, 2006
http://is.ifmo.ru/papers/_velder_bachelor.pdf

88. *Egorov K. V., Shalyto A. A.* Razrabotka verifikatora avtomatnyh programm. // Nauchno-tehnicheskiy vestnik SPbGU ITMO. Vyp. 53. Avtomatnoe programmirovanie. *2008*, c. 177–188.
http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

89. *Hoffman L.* Talking Model-Checking Technology // Communications of the ACM. 2008. Vol. 51. № 07/08, pp. 110–112. http://is.ifmo.ru/verification/_model_checking.pdf

90. *Hoffman L.* In Search of Dependable Design // Communications of the ACM. 2008. Vol. 51. № 07/08, pp. 14–16. http://is.ifmo.ru/verification/_v_poiskax_nadejnogo_koda.pdf

91. *Biere A., Heule M., Maaren H. van, Walsh T. (eds.)* Handbook of Satisfiability. IOS Press, 2009.

92. *Long O. E.* Model Checking, Abstraction and Compositional Reasoning. PhD thesis. Carnegie Mellon University, 1993.

93. *Schnoebelen P., Bérard B., Bidoit M., Laroussinie F., Petit A.* Vérification de logiciels: techniques et outils du model-checking. Vuibert, 1999.

94. *Bérard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L., Schnoebelen P.* Systems and Software Verification. Model-Checking Techniques and Tools. Springer, 2001.

95. *Kubasov S. V., Sokolov V. A.* Sinhronnaya model' avtomatnoy programmy // Modelirovanie i analiz informatsionnyh sistem. 2007. № 1, c. 11–18. http://mais.uniyar.ac.ru/ru/article/61

96. *Vel'der S. E.* Primenenie metodov snizheniya razmernosti k zadacham verifikatsii TCTL i optimal'noy ukladki grafov. Magisterskaya dissertatsiya. SPbGU ITMO, 2008. http://is.ifmo.ru/papers/_velder_master.pdf

97. *Kubasov S. V.* Verifikatsiya avtomatnyh programm v kontekste sinhronnogo programmirovaniya. Dissertatsiya na soiskanie uchenoy stepeni kand. tehn. nauk. Yaroslavl'. YaGU im. P. G. Demidova, 2008.http://is.ifmo.ru/disser/kubasov_disser.pdf

98. *Vel'der S. E.* Verifikatsiya modeley avtomatnyh programm. Dissertatsiya na soiskanie uchenoy stepeni kand. tehn. nauk. SPbGU ITMO, 2011.