ISSN 0361-7688, Programming and Computer Software, 2009, Vol. 35, No. 4, pp. 212–222. © Pleiades Publishing, Ltd., 2009.
Original Russian Text © V.V. Kuliamin, 2009, published in Programmirovanie, 2009, Vol. 35, No. 4.

Integration of Verification Methods for Program Systems

V. V. Kuliamin

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: kuliamin@ispras.ru

Received December 24, 2008

Abstract—In the paper, an approach to constructing an extensible framework for the verification of program systems is suggested. In the author's opinion, it will facilitate application of modern rigorous verification methods to practically significant programs, the complexity of which permanently grows. This framework can also become a test harness for testing and adjustment of new techniques of formal verification and static analysis on various industrial software packages.

DOI: 10.1134/S0361768809040057

1. INTRODUCTION

During last two decades, we witnessed great progress in the software development technology. This progress, in particular, greatly improved productivity of the programmers in terms of the amount of code created in a time unit, which results in the increase of most complicated program systems up to tens of millions of code lines [1, 2]. However, the quality of programs almost has not been changed: as before, the average number of errors per thousand lines of code prior to testing varies from 10 through 50 [3]. Thus, the improvement of the software development methods, on the one hand, allows us to create more and more complicated systems necessary for modern science, economy, and state organizations and, on the other hand, increases the number of defects in these systems and risks associated with this.

Software defects and errors can be eliminated by means of *verification*, which checks mutual consistency of all design artifacts—project and user documentation, source code, deployment configurations, etc.—and their correspondence to the requirements to the given system and relevant standards. Methods of software verification are also rapidly developing, but not as fast as the development technologies. Therefore, the ultimate complexity of the software which can be made reliable and functioning correctly is considerably less than that demanded by a modern society.

Methods of software verification can be divided (this division is more historical than based on their essence) [4] into *formal methods*, which rely on rigorous analysis of mathematical models of the artifacts being checked and the desired properties; *static analysis methods*, which seek errors without running the software; *dynamic analysis methods*, which verify actual behavior of the system under study in some scenarios of its operation; and *review* (*inspection*),which is performed by experts based on their experience and knowledge.

All these methods have their advantages and disadvantages and different application domains, and their efficiency may differ significantly in different contexts. Valuable verification of large-scale complex systems is impossible without combined use of all these methods, because only their combination can overcome disadvantages of the individual methods. In so doing, on each level of system consideration and for each kind of components, it is desirable to select the most effective method that results in the most reliable contribution into the system quality estimate on the whole and requires minimum expenditures. Unfortunately, for the present there is no general approach to comparing efficiency of different verification methods and their combinations in different contexts as applied to real software systems.

To manage the permanently growing complexity of real systems, a great number of various verification methods and techniques, especially formal and based on static analysis, have been created during last 20–30 years [4]. However, these methods most often can effectively be used only by specialists in the corresponding fields. Many works of such kind were confined to formulation of ideas and algorithms. Occasionally, they resulted in prototype implementations, the goal of which was to demonstrate feasibility of the proposed technique on several examples. However, these prototypes cannot be used for industrial software development, where all tools should efficiently work in a very wide context, since the researchers usually have no resources and time to develop industrial applications.

In the rare cases when such a practical tool is created, it includes usually one or two dozens of different techniques and is capable of solving two or three verification problems. On the other hand, the process of industrial software development requires solving several dozens of such problems, whereas the majority of organizations manage to successfully introduce and start to actively use only two or three tools of this kind.

Another difficulty is the growing complexity of the creation and approbation of new verification techniques. All environment required for their operation—tools for the source code analysis, descriptions of formal models, libraries for work with internal code and model representations, tools implementing various code and model analyses, reporting tools—cannot be developed from scratch. To check feasibility of certain ideas, the researcher composes this environment from ready heterogeneous components and libraries. In the best case, one creates a prototype capable of coping with a couple of special examples. However, this way does not work when it is required to create an environment that could be used for analyzing feasibility and effectiveness of a new idea in a wide range of situations and on different kinds of applications and requirements to them. Therefore, the majority of new ideas are used in special circumstances only, and their effects in wider contexts remain unclear and unpredictable.

A way out of the difficulty is to develop a unified extensible framework for the verification of program systems that provides a common framework for solving verification problems and libraries of ready components implementing standard techniques. Such a framework could significantly simplify integration of modules implementing different verification techniques owing to unified extension interfaces.

By means of this framework, the researchers could greatly reduce expenses required for the testing of new methods and analysis of their applicability in various situations. The developers in industry could use it for integrating the desired set of techniques in the framework of a single tool and for efficient transfer of these techniques into practical use.

The existence of numerous synthetic verification methods evidences feasibility and efficiency of integration of different software verification methods in various situations.

2. SYNTHETIC SOFTWARE VERIFICATION METHODS

The synthetic verification methods use different techniques (according to the above classification) and combine ideas of different approaches to achieve better verification efficiency in terms of the required resources and confidence of the results obtained.

These methods are currently classified into the following groups:

• Static analysis is based on automated construction of certain models of the code of the system under check and verification of correctness of these models with respect to some set of rules (e.g., any variable must be initialized before every use), as well as searching certain errors by appropriate patterns (e.g., dereferencing of a pointer is a correct operation only after it has been checked that it is not null). For the models, labeled control flow and data flow graphs are usually used. Currently, specific kinds of static analysis are frequently used, where formal models and various tools for resolving constraints are applied for deeper analysis of the code features.

◦ *Extended static checking* [5–10] checks correspondence of the code to the software requirements, which are also written in the code, for example, as comments to its separate elements (procedures, data types, and class methods). Based on results of the code analysis, formal models of its behavior are automatically constructed. The conformance between requirements and these models is usually checked by applying theorem proving and solvers. In this approach, the static analysis is integrated with one of the formal methods, namely, with the theorem proving. Examples of such synthesis are the ESC/Java2 [6, 7], Boogie [8], Saturn [9], and Calysto [10] tools.

◦ *Static analysis based on automatic abstraction* [11, 14–19]. The use of such methods suggests that, on the basis of results of the static analysis of the code, more abstract (and, hence, simpler) models of the operation of the software under check are automatically constructed. Then, it is verified whether these models meet certain properties by using model checking or specialized solvers. In this case, the static analysis is integrated with the theorem proving or model checking. The difference of this approach from the previous one is that the properties being checked have the form of general rules of code correctness—only not-null pointers can be subjected to dereferencing, elements of an array with indices exceeding array dimensions cannot be accessed, resource capture on any path should be followed by its release, and the like—rather than are determined by requirements specific to the component under check. Accordingly, such rules are fixed for a given tool. Sometimes, partial verification of rules from some sufficiently large list is possible.

An example of the abstract models is *Boolean models* [11], i.e., sets of flags that represent information required for analysis of program properties (for example, on branching and loop conditions) in a concise form. Another example is *octagons* [12], i.e., sets of constraints of the form $x \pm y \in [a, b]$, where x and y are variables and a and b are constants. Such sets of constraints can be resolved more efficiently compared to arbitrary sets of linear inequalities. Some tools of this type use *counterexample-guided abstraction refinement* (CEGAR) [13]: if a requirement to a model is violated, an attempt is made to build the corresponding scenario of code operation; if this is impossible because of simplifications made in the model, elements of the code preventing execution of this scenario are determined, and refinements that describe operation of these elements more accurately are introduced into the model; after the refinement, it is checked again whether the model satisfies the given property. As a result, the tool either approves the conformance to the requirement, or finds a counterexample, or stops its operation upon expiration of some time or exhausting a resource without arriving to certain conclusions. Examples of tools based on the static analysis with automatic abstraction are PolySpace Verifier [14, 15] and ASTREE [16, 17]. Among the tools supporting the counterexample-guided abstraction refinement are the well-known SLAM [11] and BLAST [18] tools, as well as a lesser-known MAGIC [19].

• When using *synthetic structural testing* [20–29], after the first randomly selected test, all other tests are generated automatically to ensure coverage of the code elements that have not been covered earlier. Appropriate test data are selected with the help of solvers that take into account symbolic information on data constraints that prevent the tests passed from checking the code that has not been covered yet. To construct the desired sequences of actions, random generation guided both by the above-mentioned symbolic information and heuristic abstractions reducing the state space of the tested system is used. In this approach, static code analysis, structural testing, and theorem proving performed by solvers are integrated. Examples of such tools are CUTE and jCUTE [22], Crash'n'Check [23] and DSDCrasher [24], Rostra and Symstra [25], UnitMeister [26] and Pex [27], Exe [28] and RANDOOP [29].

• *Model based testing* [30–33] combines development of formal models of the requirements to the verified software and generation of tests on the basis of these models. In this case, the model structure serves as a basis for the test adequacy criteria, and model restrictions on the correct results of the software operation are used as test oracles evaluating correctness of the software behavior in the course of testing. Surveys of numerous model based testing tools can be found in [30, 31, 33, 34]. In the last two approaches (or independent of them), specific techniques of test generation based on various verification methods are used.

◦ *Test generation based on constraint solving techniques* [35–37]. In the development of tests on the basis of the test adequacy criteria, the so-called *test objectives* are often formulated. The latter are specific situations in which the behavior of the system under test should be verified to make sure that it works correctly. A test objective is formulated as a set of constraints imposed on the system states and the data of the actions performed in the course of testing. To generate a test that achieves this objective, special solvers can be used. Such a solver either automatically finds the desired data and a sequence of operation calls by solving the given system of constraints or advises that the system cannot be solved, i.e., the given test objective cannot be reached and there is no sense to generate tests aimed at it.

◦ *Generation of tests from counterexamples using model checking tools* [38–41]. Another way of test generation is to formulate negations of the constraints specifying the test objective as a property that can be either checked or disproved by using model checking tools. If this property is proved, then the test objective cannot be reached; if it is disproved, the tool constructs a counterexample that serves as a required test in the given case.

• *Runtime verification (passive testing) of formal properties* [42–45] also uses formal models of the requirements for evaluating correctness of the behavior of the tested system but does this only in the course of its ordinary operation, without using tests specially constructed for this purpose. Thus, in this approach, the model checking and runtime verification are integrated. Sometimes, the runtime verification is carried out in the framework of symbolic execution of the code rather than in the course of its ordinary operation. Examples of the runtime verification tools are Temporal Rover [46] and Java Path Finder [47, 48], with the latter performing symbolic execution of the programs under check.

As can be seen, all synthetic methods, in one way or another, try to combine advantages of different approaches to the verification, while avoiding their disadvantages. Currently, there exist many examples of successful development of such methods and their practical applications. Some of them are listed below.

• Numerous NASA projects on the development of software for control of space satellites, shuttles, and specialized research vehicles carried out with the help of model checking and test generation tools and runtime verification [49–51]. The tools used in these projects include model checking tool Spin [52, 53], test generator T-VEC [54, 55], and Java PathFinder [47, 48].

• Development and use of the Static Driver Verifier tool in Microsoft. This tool uses static analysis based on automatic abstraction for verification of correctness of Windows drivers operation [56, 57]. First, the model checking tool SLAM [11] was used in the project. Then, it was considerably revised to support analysis of arbitrary code in C and supplemented by a set of rules for correct use of functions of the Windows kernel in the drivers.

• Internal Microsoft project on formal specification and generation of test suites for various client–server protocols used in the products of this company [58]. In this project, the SpecExplorer tool [59, 60] developed by Microsoft Research was used, and the amount of work on the analysis and formalization of protocol-related documentation was estimated to be several dozens of person-years. A clear demonstration of interest of industry in the development of software verification methods is the work of the Research in Software Engineering (RiSE) group [61] in Microsoft Research: the most actively developing fields of its research are just various synthetic verification methods.

• Recent and current projects at the Institute for System Programming, Russian Academy of Sciences, on test generation on the basis of formal models of the operating system kernel, basic operating system libraries, telecommunication protocols of the IPv6 family, compiler optimizing units [62–65], which use mainly a family of model based testing tools UniTESK [33].

• Use of formal verification methods and extended static analysis tools in the development of the avionics systems in Airbus and Boeing [16, 17, 66]. In particular, in Airbus, a static analysis tool based on the ASTREE formal models was used [16].

• Use of formal models, model based testing, and runtime verification in the development of software for smart cards [67, 68].

All these examples approve efficiency of integration of different verification methods in practice. Nevertheless, in spite of the achievements, each of the synthetic approaches uses only a portion of the available potential and does not provide one with a unified integration framework for the entire variety of the software verification techniques. Moreover, certain heterogeneity of these approaches does not allow us to adequately compare their characteristics when they are applied to complex program systems.

3. APPROACH TO CONSTRUCTING EXTENSIBLE SOFTWARE VERIFICATION FRAMEWORK

Problems associated with the growing complexity in the development and testing of the software verification methods and the necessity in the creation of an extensible framework that would allow one to integrate different techniques and tools have been already discussed by the researchers (see, for example, [69]). However, no systematic approach to constructing such a framework has been presented in the literature yet.

To be implemented in practice, such an approach should suggest adequate solutions for the following methodological and organizational problems.

• Problems of interaction of various verification methods with the requirements analysis within the software development processes.

• Place and techniques of using the software reviews within the software verification approach.

• Use of models, languages, and notations of different kinds and methodological and technical difficulties of their integration in a unified framework.

• Selection of basic architecture of the integrated verification framework.

• Organization of work on the development of such a framework.

These problems are discussed in detail in what follows.

3.1. Analysis of Requirements

No verification is possible without preliminary clear formulation of the requirements being checked. In practice, verification almost always begins with an analysis of the requirements to the system and, as a rule, with their partial formalization.

However, there does not exist a unique approach to analyzing and representing the requirements, and, most likely, such an approach will not be created in the nearest future. Then, is it possible to construct a unified verification framework that integrates different approaches and uses different techniques for analyzing requirements?

To this end, we propose to leave the problems of the requirements analysis beyond the scope of the framework discussed and define a clear interface between it and the activity on requirements extraction. To justify adequacy of the verification performed for the software users and customers, it is required that each element of the models used and a section of the report on defects found could be associated with an element of the requirements formulated by the users and customers; i.e., the original requirements should be traceable. Therefore, leaving aside problems related to establishing various relationships between the requirements, ensuring their adequacy and completeness, and formalizing informal requirements, we may consider the requirements as a set of some objects with unique identifiers that make it possible to associate model elements, tests performed, and defects found with these objects. The nature of these objects—whether they are texts, formulas, images, schemes, etc.—is not important in this case.

Thus, in the verification framework, a mechanism for requirement tracing should be supported, which makes it possible to associate their unique identifiers with various verification artifacts and their separate elements.

3.2. Place of Reviews in the Integrated Approach to Software Verification

The form of using the reviews for software verification within the considered framework is an important issue. Review is applicable to any properties of the software and any artifacts, although, for different purposes, different kinds of reviews are used. Review makes it possible to reveal errors of all kinds on early stages, minimizing thus time of defect existence in the software lifecycle and resources required for eliminating it. Empirical studies show that the efficiency of code inspection (even that which does not involve other development artifacts), measured as the ratio of the number of defects found to the resources spent, is higher than that of other verification methods. According to various reports, from 50 to 90 per cent of errors during the software lifecycle can be detected by means of review [70–72]. High efficiency of the review can be explained by the ability of a human to find possible defects in uncertain situations without clear and complete understanding of the requirements to the software.

At the same time, the review cannot be automated and always require participation of people. Moreover, the review efficiency greatly depends on their experience and motivation, as well as on the organization of the development process and professional communications between the participants. This imposes serious restrictions on the distribution of resources in the projects and may lead to conflicts if the organizational aspects of the review are neglected.

In the considered framework, it is suggested to maximally use formalized representations for the majority of the development artifacts in order to be able to apply one or another automated analysis to them. The reviews in this case also give rise to good results; however, they are most advantageous when used in informal contexts, since, in such situations, a man is more effective than any tool in searching problems.

Therefore, review should be fully used in the course of the requirement analysis and formalization, which makes it possible to combine advantages of different verification methods in the best way. The reviews are most effective just on the stage of defining, refining, and analyzing the requirements, where all other methods do not work. On the other hand, automated techniques are more effective in the analysis of formalized artifacts.

3.3. Use of Different Models

One of the most important issues arising in the connection with the use of formal models for the verification is related to the types of models supported by the framework discussed. The answer to this question affects the following characteristics of the framework.

• Expressive capabilities of models, i.e., variety of the requirement types and the properties that can be described in these models; possibility of system description on different abstraction levels.

• Scalability of models and restrictions on system complexity that can be described by means of these models.

• Supported verification methods (some methods and techniques can be applied to models of only certain types).

• Convenience of work with models, the amount of man-hours required to study the framework and incorporate it into industrial process of software development.

Analysis of the experience in the verification of industrial software in various organizations and projects [32, 33, 49, 57, 58, 62, 64] allows us to conclude that contract and automaton models are most appropriate for the description of significant systems used in practice.

Contract models formulate requirements to a component as an abstract description of the structure of its internal state, a set of invariants determining correct states, and a set of pre- and postconditions for all operations of the component specifying, respectively, their domains and constraints on the correct results and modifications of states when calling these operations. Contract models make it possible to accurately define responsibilities of different sides: the client that access this component is responsible for the fulfillment of the preconditions of the operation, whereas the component itself is responsible for the fulfillment of the postconditions and invariants.

Automaton models describe behavior of the system (or its component) by specifying a set of its states, stimuli by means of which one can affect it, possible reactions, and a set of transitions between the states, each of which can be invoked by a certain stimulus, be related to some reaction, or be an internal transition (i.e., one that occurs without any visible external events). The assignment of stimuli and reactions to transitions may be different. In the simplest case, each transition corresponds to a pair stimulus–reaction. A transition may be internal or related only to either one stimulus or one reaction. In automaton models used in practice, stimuli and reactions may contain rather complex data structures.

The contract models make it possible to give more declarative and abstract descriptions of the system behavior. Moreover, unlike automaton models, they are more appropriate for describing non-determinism or behavior associated with transformation of complex data structures. However, they are not executable and not convenient for describing composition of the components. The automaton models are executable, and the composition of the components is defined for these models. In addition, they are more appropriate for using model checking methods and for generating test sequences.

Models of both types are extensively used for the description of program systems of high complexity (including the component-wise description) and are well-scaleable: contract models due to the possibility of working on different abstraction levels and automaton models owing to the use of composition for construction of system specification from specifications of its components.

3.4. Support of Different Languages and Notations

Many verification tools are designed for working with certain languages of model and requirement representation. When integrating different methods, we may pose the question of what languages to use at all and support of what languages should be implemented at first.

The experience acquired in the large number of projects on verification of industrial software [32, 33, 64] makes us conclude that it is preferable to use languages similar to widely used programming languages that are customary to the developers. Therefore, for representing models in the considered framework, one should use languages that are minimal extensions of programming languages. Later, one may add support for certain languages of formal specifications.

Support of different languages and notations should be implemented on the level of some common intermediate representation of language constructs that is used by all analysis tools, but not users themselves. The development of such intermediate representation that is applicable to many different languages is not a trivial task, especially if languages that differ significantly in terms of basic paradigms are used. As experience shows [73, 74], such a common representation of programs greatly depends on the problems solved on its basis, and the reuse of representations developed in other projects is hardly possible. Therefore, an interface for the intermediate representation used by the framework will, most likely, be developing gradually and will rely on the working experience acquired in the course of its development. The set of concepts itself on the basis of which such a representation can be created is not clear at the moment: it should be determined in the course of creation of the verification framework described in this paper.

In spite of the above difficulties, available standard or widely used high-level libraries should be used for work with the intermediate representation for a number of languages. For example, for the C and C++ languages, the intermediate representation used in the GCC compiler (the *trees level* [75]) becomes a de facto standard. The great advantage of using results of such projects is that their support and development in the near future are guaranteed.

3.5. Basic Architecture of the Verification Framework

The architecture of the verification framework discussed—a set of its basic components, their external interfaces and interaction rules, and rules for adding new components—determines one of the most important features of this framework, namely, its extensibility. On the other hand, solutions related to the basic principles of the framework construction may affect possibility of its integration with other software development tools.

First of all, to facilitate its use in the industrial software development, the verification framework should be built into one of the widely used development frameworks, such as Eclipse or Microsoft Visual Studio. Eclipse [76, 77] is the most appropriate integration framework for the first versions, since it possesses a huge set of extension modules, including verification tools and modules for support of various programming languages. In addition, the process of creating such modules is well documented.

In addition to the external environment, in the context of which the verification framework has to work, it is required to determine its frame, i.e., some base set of components implementing a basic set of functions and supporting main data flows inside the system. To this frame, other components supporting auxiliary and less significant functions will be added.

As the basis for constructing the verification framework, we suggest using an architecture frame of model-based testing tools. This is explained by the fact that such testing is one of the most complicated verification processes from the point of view of its organization: the nomenclature of the activity types in this process is the widest one. Usually, in the course of testing, it is required to do the following.

• To define a behavior model of the system under test, which formalizes requirements to the behavior.

• To analyze model structure for choosing coverage criteria and separate test objectives and defining these criteria and objectives.

• To construct a test execution environment (which includes monitoring and protocoling external actions, system reactions, and, possibly, internal events) and test oracles (program components determining whether the observed behavior of the system corresponds to that defined by the model). Usually, such an environment consists of a library supporting test execution, a set of test oracles for all components under check, and a set of adapters relating these components to their oracles. In the majority of cases, the oracles are generated automatically from a model constructed earlier.

• To construct (automatically or with the help of a man) a set of test scenarios determining sequences of calls of various operations of the system being tested, or sequences of messages or signals to be sent to it, or data transmitted as parameters of the operations and messages.

• To execute test scenarios protocoling all information related to the correspondence of the observed system behavior and its model, as well as the situations covered in the course of testing.

• To carry out analysis of the test results, in the course of which errors in the system or its model are revealed and analyzed (which manifest themselves as discrepancies between the expected and actual behaviors), the test coverage obtained is analyzed, and a decision is made on either generation of additional tests or finishing the activity.

In order to incorporate other synthetic verification methods in the model-based architecture frame, it is required to add modules for analyzing the source code of the components under test (all other required components are, in fact, already available in it [78]).

A preliminary variant of the architecture of a unified extensible software verification framework is

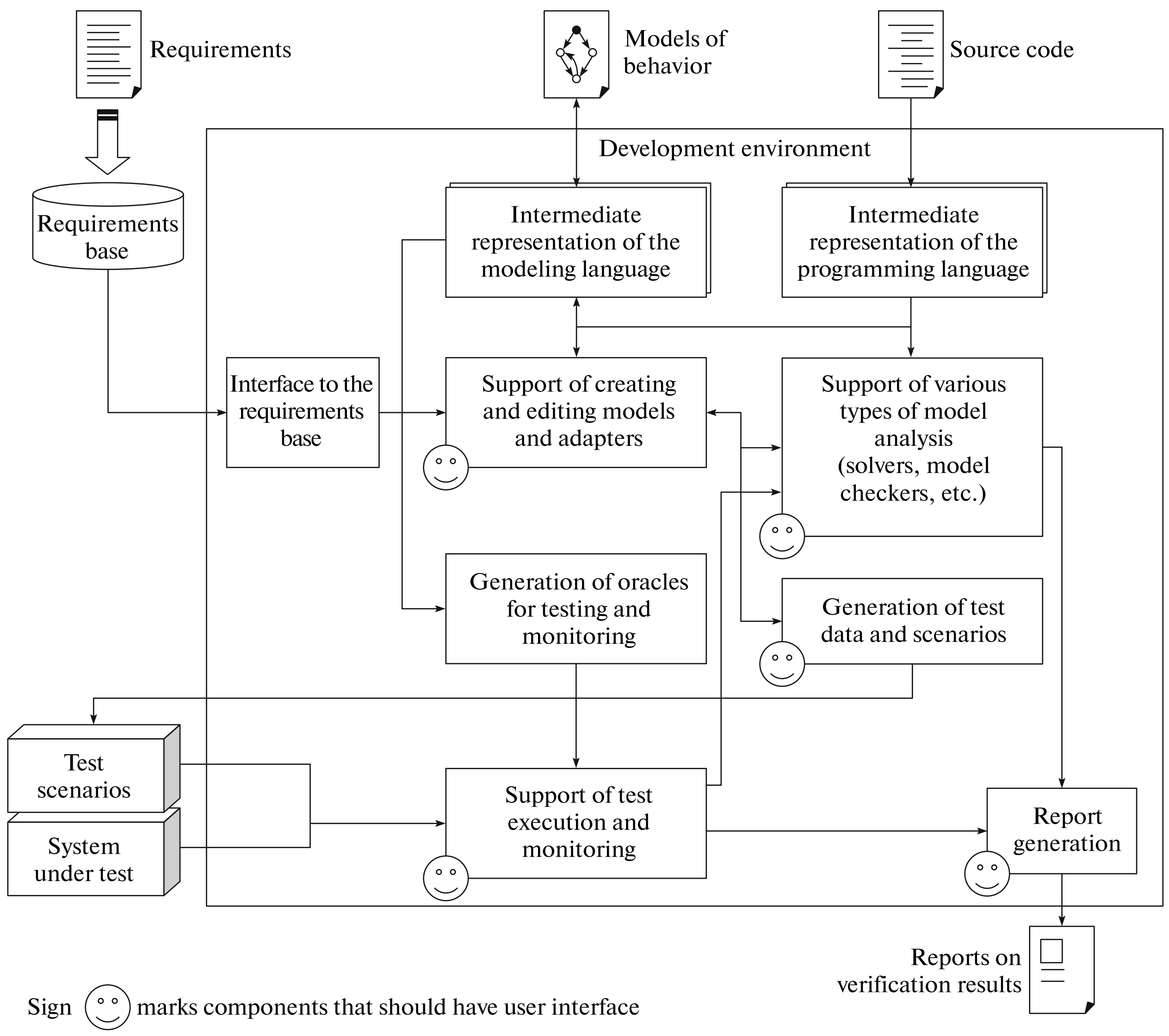
Requirements
Models of behavior
Source code
Development environment
Requirements base
Intermediate representation of the modeling language
Intermediate representation of the programming language
Interface to the requirements base
Support of creating and editing models and adapters
Support of various types of model analysis (solvers, model checkers, etc.)
Generation of oracles for testing and monitoring
Generation of test data and scenarios
Test scenarios
System under test
Support of test execution and monitoring
Report generation
Reports on verification results
Sign marks components that should have user interface

Fig. 1. Preliminary architecture of an extensible software verification framework.

shown in Fig. 1. Only large-scale components are presented in the figure. A more detailed architecture design may require division of these components into smaller parts and addition of other modules solving auxiliary tasks.

Among all synthetic verification methods, the extended static checking is the worst one with respect to scalability, since description of detailed constraints (including loop invariants) for a large code requires too much expenditure. However, all other methods can be supported on the basis of the suggested architecture.

In the framework of this architecture, model-based testing can be performed, since the majority of its components are taken from a typical architecture of tools for such testing. Monitoring of formal properties is organized on the basis of a subset of these components: for this subset, there is no need to generate and perform tests; it is sufficient to run the system under test in the framework of the testing and monitoring support environment with the use of test oracles.

Scenarios of execution of the static analysis based on automatic abstraction and synthetic structural testing are depicted in Fig. 2. For each method, the order of execution of separate actions—analysis of the source code, creation of a model on its basis, analysis of the model with the help of specialized techniques, test generation, etc.—is shown in the figure.

Steps 3 and 4 in the static analysis based on automatic abstraction are performed in a loop until the code correctness is proved, or a counterexample is found, or the resources assigned for the check are exhausted. Steps 4–7 in the synthetic structural testing are also performed in a loop until a set of tests is

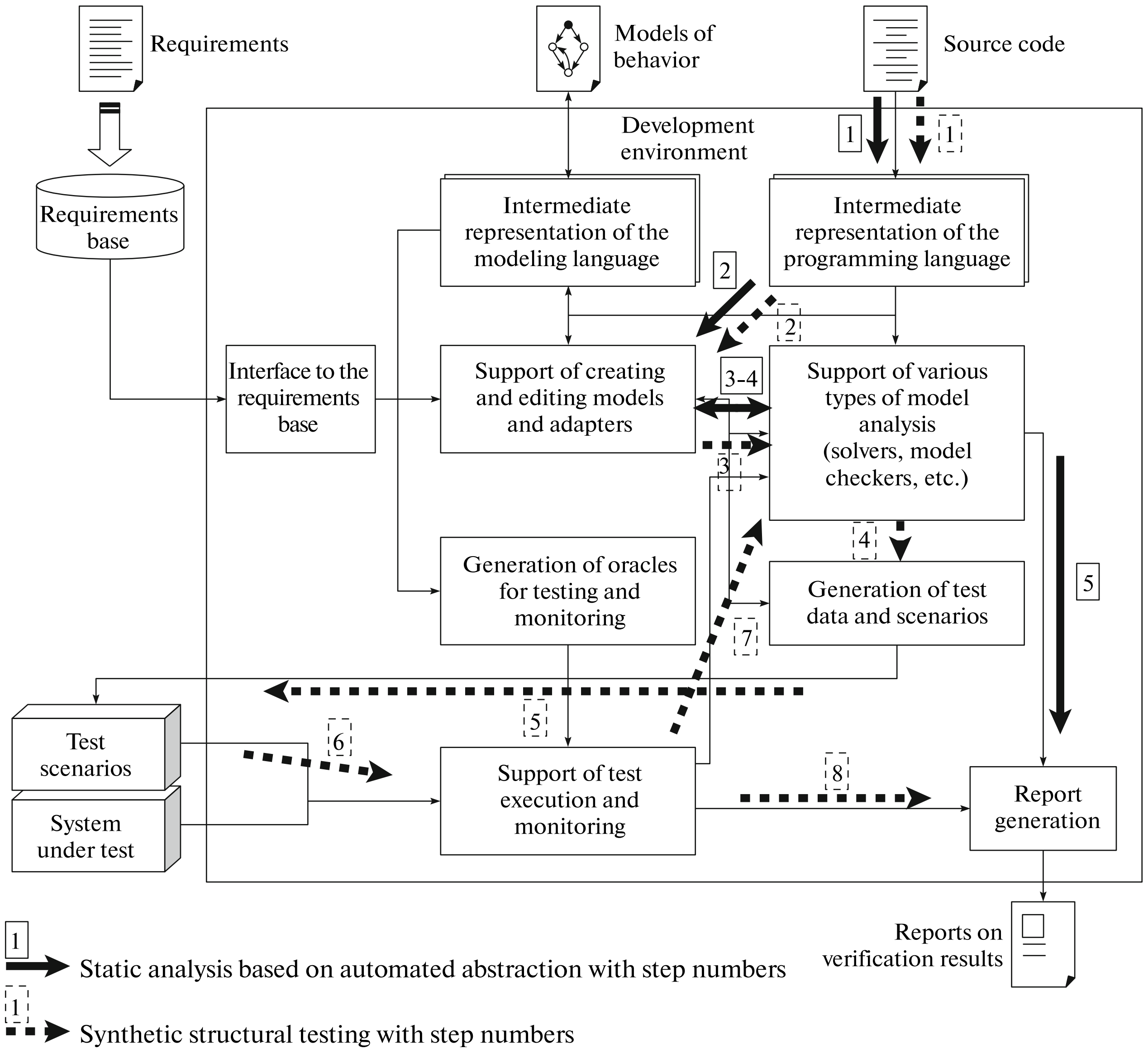
Requirements
Models of behavior
Source code
Development environment
Requirements base
Intermediate representation of the modeling language
Intermediate representation of the programming language
Interface to the requirements base
Support of creating and editing models and adapters
Support of various types of model analysis (solvers, model checkers, etc.)
Generation of oracles for testing and monitoring
Generation of test data and scenarios
Test scenarios
System under test
Support of test execution and monitoring
Report generation
Reports on verification results
1
2
3-4
3
4
5
6
7
8
Static analysis based on automated abstraction with step numbers
Synthetic structural testing with step numbers

Fig. 2. Support of various verification methods by the proposed architecture.

obtained that ensures the desired coverage or the resources assigned for the test generation are exhausted.

3.6. Organization of the Verification Framework Development

The development of the above-described verification framework will require considerable resources, even in the case of using available components implementing various types of analysis, test generation algorithms, or parsing code in certain programming languages. Because of the huge amount of the required labor, the development of the framework by a small group of people seems unrealistic.

Therefore, the development of such a framework can be organized as an open project in the Internet with the participation of any developer who agrees to follow suggested architectural solutions and other rules of the project.

To begin such a project, it is important to prepare a general architecture of the framework and an implementation of some significant part of its functionality. As the first variant, the unit testing system TestNG [79, 80] extended by model-based tests generation means can be considered. TestNG is a popular framework for development of unit and integration tests for Java applications, which allows one to create test sets for sufficiently complex systems and flexibly configure their execution. In addition, TestNG has an open code. Its extension by model-based testing capabilities and, at least, one or two types of model and code analyses (for example, those implementing synthetic structural testing in a number of situations) makes it possible to clearly demonstrate integration capabilities of the proposed approach.

4. CONCLUSIONS

In the paper, an approach to the integration of various software verification methods has been suggested. The goal is to considerably increase complexity of program systems, for which verification by rigorous methods based on formal models in an explicit or implicit form can yield valuable results under acceptable expenditures.

The proposed approach is based on combining several synthetic verification methods (extended static checking, synthetic structural testing, model-based testing, and runtime verification of formal properties) successfully used in practice in the unified extensible software verification framework. As a base architecture for such a framework, we suggest using the well-known model based testing tools architecture [48] extended by additional components for analysis of the source code of the components under test and for various types of model analyses, including various solvers. Model-based testing is chosen to be a foundation of the proposed architecture, since it is the most complicated verification method among those being combined.

A number of methodical and technical solutions, which, in the author's opinion, make the development of the discussed verification framework realizable and facilitate its application to solving practical problems of industrial software verification, are also presented.

Another possible application of such a framework is testing and adjustment of numerous new techniques of verification and analysis of software properties.

REFERENCES

1. Maraia, V., *The Build Master: Microsoft's Software Configuration Management Best Practices*, Addison-Wesley Professional, 2005.
2. Robles, G., Debian Counting. http://libresoft.dat.escet.urjc.es/debian-counting/.
3. McConnell, S., *Code Complete*, Microsoft Press, 2004.
4. Kulyamin, V.V., Software Verification Methods, *Vserossiiskii konkurs obzorno-analiticheskikh statei po prioritetnomu napravleniyu "Informatsionno-telekomunikatsionnye sistemy"* (All-Russian Competition of Analytic Survey Papers on Priority Direction "Information-Telecomunication System") 2008. http://window.edu.ru/window/library?p_rid=56168.
5. Detlefs, D.L., Leino, K.R.M., Nelson, G., and Saxe J.B., Extended Static Checking, *Tech. Report SRC-RR-159*, Digital Equipment Corporation, Systems Research Center, 1998.
6. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R., Extended Static Checking for Java, *Proc. of ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation*, 2002, pp. 234–245.
7. Cok, D.R. and Kiniry, J.R., ESC/Java2: Uniting ESC/Java and JML, *Lecture Notes in Computer Science* (Proc. of Int. Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)), Springer, 2005, vol. 3362, pp. 108–128.
8. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., and Leino, K.R.M., Boogie: A Modular Reusable Verifier for Object-Oriented Programs, *Lecture Notes in Computer Science* (Proc. of Formal Methods for Components and Objects), 2005, vol. 4111, Springer, 2006, pp. 364–387.
9. Xie, Y. and Aiken, A., Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability, *ACM Transactions Programming Languages Systems* (Proc. of Principles of Programming Languages (POPL 2005)), ACM, 2007, vol. 29, no. 3.
10. Babic, D. and Hu, A.J., Calysto: Scalable and Precise Extended Static Checking, *Proc. of the 30th Int. Conf. on Software Engineering*, 2008, pp. 211–220.
11. Ball, T. and Rajamani, S.K., Automatically Validating Temporal Safety Properties of Interfaces, *Lecture Notes in Computer Science* (Proc. of Model Checking of Software), Springer, 2001, vol. 2057, pp. 103–122.
12. Miné, A., The Octagon Abstract Domain, *Higher-Order Symbolic Computation*, 2006, vol. 19, no. 1, pp. 31–100.
13. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H., Counterexample-Guided Abstraction Refinement, *Lecture Notes in Computer Science* (Proc. of CAV 2000), Springer, 2000, vol. 1855, pp. 154–169.
14. Emanuelsson, P. and Nilsson, U., A Comparative Study of Industrial Static Analysis Tools, *Tech. Report 2008:3*, Linkoping University, 2008. http://www.ep.liu.se/ea/trcis/2008/003/trcis08003.pdf.
15. http://www.mathworks.com/products/polyspace/index.html.
16. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., and Rival, X., Design and Implementation of a Special-purpose Static Program Analyzer for Safety-critical Real-time Embedded Software *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, Schmidt, D.A. and Sudborough, I.H., Eds., Lecture Notes in Computer Science, Springer, 2002, vol. 2566, pp. 85–108.
17. Souyris, J. and Delmas, D., Experimental Assessment of ASTRÉE on Safety-Critical Avionics Software, *Lecture Notes in Computer Science* (Proc. of Int. Conf. on Computer Safety, Reliability, and Security, SAFECOMP 2007), Saglietti, F. and Oster, N., Eds., Nuremberg, Germany, Springer, 2007, vol. 4680, pp. 479–490.
18. Henzinger, T.A., Jhala, R., Majumdar, R., and Sutre, G., Software Verification with Blast, *Lecture Notes in Computer Science* (Proc. of the 10-th SPIN Workshop on Model Checking Software (SPIN 2003)), Springer, 2003, vol. 2648, pp. 235–239.

19. Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H., Modular Verification of Software Components in C, *IEEE Trans. Software Engineering*, 2004, vol. 30, no. 6, pp. 388–402.

20. Godefroid, P., Klarlund, N., and Sen, K., DART: Directed Automated Random Testing, *Proc. of 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM, 2005, pp. 213–223.

21. Godefroid, P., Compositional Dynamic Test Generation, *Proc. of the 34-th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (PLOP 2007)*, 2007, pp. 47–54.

22. Sen, K., Agha, G., CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools, *Proc. of Comput. Aided Verification*, 2006, pp. 419–423.

23. Smaragdakis, Y. and Csallner, C., Check 'n' Crash: Combining Static Checking and Testing, *Proc. of the 27-th ACM/IEEE Int. Conf. on Software Engineering (ICSE)*, 2005, pp. 422–431.

24. Smaragdakis, Y. and Csallner, C., Combining Static and Dynamic Reasoning for Bug Detection, *Lecture Notes in Computer Science* (Proc. of TAP 2007), Springer, 2007, vol. 4454, pp. 1–16.

25. Xie, T., Marinov, D., Schulte, W., and Notkin, D., Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution, *Proc. of the 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Edinburgh, 2005, pp. 365–381.

26. Tillmann, N. and Schulte, W., Parameterized Unit Tests with Unit Meister, *ACM SIGSOFT Software Engineering Notes*, 2005, vol. 30, no. 5, pp. 241–244.

27. Tillmann, N. and de Halleux, J., Pex—White Box Test Generation for .NET, *Lecture Notes in Computer Science* (Proc. of TAP 2008), Springer, 2008, vol. 4966, pp. 134–153.

28. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., and Engler, D.R., EXE: Automatically Generating Inputs of Death, *Proc. of the 13th ACM Conf. on Comput. and Communications Security*, Alexandria, Virginia, USA, 2006, pp. 322–335.

29. Pacheco, C., Lahiri, S.K., Ernst, M.D., and Ball, T., Feedback-Directed Random Test Generation, *Proc. of Int. Conf. on Software Engineering*, 2007, pp. 75–84.

30. Model Based Testing of Reactive Systems, *Lecture Notes in Computer Science*, Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A., Eds., Springer, 2005, vol. 3472.

31. Utting, M. and Legeard, B., *Practical Model-Based Testing: A Tools Approach*, Morgan-Kaufmann, 2007.

32. Jacky, J., Veanes, M., Campbell, C., and Schulte, W., *Model-Based Software Testing and Analysis with C#*, Cambridge University, 2007.

33. Kulyamin, V.V., Petrenko, A.K., Kosachev, A.S., and Burdonov, I.B., The UniTesk Approach to Designing Test Suites, *Programmirovanie*, 2003, no. 6, pp. 25–43 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 6, pp. 310–323].

34. Hartman, A., Model Based Test Generation Tools, *AGEDIS Project*, 2002. http://www.agedis.de/documents/ModelBasedTestGenerationTools.pdf.

35. Korel, B., Automated Test Data Generation, *IEEE Trans. Software Engineering*, 1990, vol. 16, no. 8, pp. 870–879.

36. DeMillo, R. and Offutt, A., Constraint-based Automatic Test Data Generation, *IEEE Trans. Software Engineering*, 1991, vol. 17, no. 9, pp. 900–910.

37. Gotlieb, A., Botella, B., and Rueher, M., Automatic Test Data Generation Using Constraint Solving Techniques, *ACM SIGSOFT Software Engineering Notes*, 1998, vol. 23, no. 2, pp. 53–62.

38. Gargantini, A. and Heitmeyer, C., Using Model Checking to Generate Tests from Requirements Specifications, *Proc. of the Joint 7th European Software Engineering Conf. and the 7-th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE99)*, ACM, 1999, pp. 146–162.

39. Hong, H.S., Lee, I., Sokolsky, O., and Cha, S.D., Automatic Test Generation from Statecharts Using Model Checking, *Tech. Report MS-CIS-01-07*, 2001.

40. Hamon, G., de Moura, L., and Rushby, J., Generating Efficient Test Sets with a Model Checker, *Proc. of the 2nd Software Engineering and Formal Methods Int. Conf.*, 2004, pp. 261–270.

41. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., and Majumdar, R., Generating Tests from Counterexamples, *Proc. of the 26th Int. Conf. on Software Engineering (ICSE)*, 2004, pp. 326–335.

42. Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M., Runtime Assurance Based On Formal Specifications, *Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications PDPTA'1999*, 1999, pp. 279–287.

43. Cheon, Y. and Leavens, G.T., A Runtime Assertion Checker for the Java Modeling Language (JML), *Proc. of Int. Conf. on Software Engineering Research and Practice (SERP'02)*, CSREA, 2002, pp. 322–328.

44. Cavalli, A., Gervy, C., and Prokopenko, S., New Approaches for Passive Testing Using an Extended Finite State Machine Specification, *Information Software Technology*, 2003, vol. 45, no. 12, pp. 837–852.

45. Drusinsky, D., *Modeling and Verification Using UML Statecharts*, Newnes, 2006.

46. Drusinsky, D., The Temporal Rover and the ATG Rover, *Lecture Notes in Computer Science* (Proc. of SPIN Workshop 2000), Springer, 2000, vol. 1885, pp. 323–329.

47. Brat, G., Visser, W., Havelund, K., and Park, S., Java PathFinder—Second Generation of a Java Model Checker, *Proc. of Workshop on Advances in Verification*, Chicago, 2000.

48. http://javapathfinder.sourceforge.net/.

49. Blackburn, M.R., Busser, R.D., and Nauman, A.M., Interface-Driven, Model-Based Test Automation, *CrossTalk, J. Defense Software Engineering,* 2003.

50. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., and Washington, R., Combining Test Case Generation and Runtime Verification, *Theoretical Comput. Sci.*, 2005, vol. 336, no. 2–3, pp. 209–234.

51. Brat, G., Havelund, K., Park, S., and Visser, W., Model Checking Programs, *Proc. of the 15th IEEE Int. Conf.*

on Automated Software Engineering, Grenoble, France, 2000, pp. 3–11.

52. Holzmann, G.J., *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, 2003.

53. http://spinroot.com/.

54. Blackburn, M., Busser, R.D., and Fontaine, J.S., Automatic Generation of Test Vectors for SCR-style Specifications, *Proc. of the 12th Annual Conf. on Comput. Assurance*, 1997, pp. 54–67.

55. http://www.t-vec.com/.

56. http://www.microsoft.com/whdc/devtools/tools/SDV.mspx.

57. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., and Ustuner, A., Thorough Static Analysis of Device Drivers, *ACM SIGOPS Operating Systems Review*, 2006, vol. 40, no. 4, pp. 73-85.

58. Grieskamp, W., Kicillof, N., MacDonald, D., Nandan, A., Stobie, K., and Wurden, F.L., Model-Based Quality Assurance of Windows Protocol Documentation, *Proc. of the 1st Int. Conf. on Software Testing, Verification, and Validation, ICST 2008*, Lillehammer, Norway, 2008, pp. 502–506.

59. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L., Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Formal Methods and Testing, *Lecture Notes in Computer Science,* Springer, 2008, vol. 4949, pp. 39–76.

60. http://research.microsoft.com/en-us/projects/specexplorer/.

61. http://research.microsoft.com/en-us/um/redmond/groups/rise/.

62. Bourdonov, I., Kossatchev, A., Petrenko, A., and Galter, D., KVEST: Automated Generation of Test Suites from Formal Specifications, *Lecture Notes in Computer Science* (Proc. of FM'99, Toulouse, France), Springer, 1999, vol. 1708, pp. 608–621.

63. Kuliamin, V., Petrenko, A., and Pakoulin, N., Practical Approach to Specification and Conformance Testing of Distributed Network Applications, *Service Availability*, Malek, M., Nett, E., Suri, N., Eds., Lecture Notes in Computer Science, Springer, 2005, vol. 3694, pp. 68–83.

64. Grinevich, A., Khoroshilov, A., Kuliamin, V., Markovtsev, D., Petrenko, A., and Rubanov, V., Formal Methods in Industrial Software Standards Enforcement, *Lecture Notes in Computer Science* (Proc. of PSI'2006, Novosibirsk, Russia, 2006), Springer, 2006, vol. 4378, pp. 459–469.

65. Zelenov, S.V., Zelenova, S.A., Kosachev, A.S., and Petrenko, A.K., Test Generation for Compilers and Other Formal Text Processors, *Programmirovanie,* 2003, no. 2, pp. 59–69 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 2, pp. 104–112].

66. Manolios, P., Subramanian, G., and Vroon, D., Automating Component-based System Assembly, *Proc. of ISSTA 2007*, London, UK, 2007, pp. 61–72.

67. Poll, E., van den Berg, J., and Jacobs, B., Specification of the JavaCard API in JML, *Proc. of CARDIS'00*, Kluwer Academic, 2000.

68. Bouquet, F. and Legeard, B., Reification of Executable Test Scripts in Formal Specification-based Test Generation: The Java Card Transaction Mechanism Case Study, *Proc. of the Int. Symp. of Formal Methods Europe*, Springer, 2003, pp. 778–795.

69. Bradley, A.R., Sipma, H.B., Solter, S., and Manna, Z., Integrating Tools for Practical Software Analysis, *Proc. of 2004 CUE Workshop*, Vienna, Austria, 2004.

70. Gilb, T. and Graham, D., *Software Inspection*, Addison-Wesley, 1993.

71. Porter, A., Siy, H., and Votta, L., A Review of Software Inspections, *Tech. Report CS-TR-3552,* University of Maryland at College Park, 1995.

72. Laitenberger, O., A Survey of Software Inspection Technologies, *Handbook on Software Engineering and Knowledge Engineering*, World Sci., 2002, vol. 2, pp. 517–555.

73. Demakov, A.V., Object-Oriented Description of Graph Data Structures, *Programmirovanie*, 2007, no. 5, pp. 261–271 [*Programming Comput. Software* (Engl. Transl.), 2007, vol. 33, no. 5, pp. 261–272].

74. Gomanyuk, S.V., An Approach to Creating Development Environments for a Wide Class of Programming Languages, *Programmirovanie*, 2008, no. 4, pp. 225–236 [*Programming Comput. Software* (Engl. Transl.), 2008, vol. 34, no. 4, pp. 225–236].

75. GNU Compiler Collection Internals, http://gcc.gnu.org/onlinedocs/gccint/index.html.

76. Daum, B., *Professional Eclipse 3 for Java Developers*, Wrox, 2004.

77. http://www.eclipse.org/.

78. Yorsh, G., Ball, T., and Sagiv, M., Testing, Abstraction, Theorem Proving: Better Together! *Proc. of ISSTA 2006*, Partland, Maine, USA, 2006, pp. 145–156.

79. Beust, C. and Suleiman, H., *Next Generation Java Testing: TestNG and Advanced Concepts,* Addison-Wesley Professional, 2007.

80. http://testng.org/doc/.