ISSN 0361-7688, Programming and Computer Software, 2010, Vol. 36, No. 5, pp. 289–305. © Pleiades Publishing, Ltd., 2010.
Original Russian Text © V.V. Kuliamin, 2010, published in Programmirovanie, 2010, Vol. 36, No. 5.

Component Architecture of Model-Based Testing Environment

V. V. Kuliamin

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia

e-mail: kuliamin@ispras.ru

Received January 21, 2010

Abstract—In the paper, an approach to constructing architecture of tools for model-based testing that rely on modern component-based technologies is presented. One of the basic ideas underlying this approach consists in application of noninvasive composition techniques, which make it possible to integrate a set of independently developed components into a complex system and reconfigure it without modifying codes of the components. The approach suggested in the paper is one of the first applications of the component-based technologies to designing test systems. A prototype implementation of the suggested approach based on available libraries is described, and an example of its use for test construction is presented.

DOI: 10.1134/S036176881005004X

1. INTRODUCTION

The growth of the number and variety of problems currently solved by computer systems leads to permanent complication of both the systems and all activities associated with their construction and maintenance. The problems related to complexity growth are dealt with *component-based technologies* of computering systems construction of [1, 2], which have been actively developed during last decades.

In the framework of these technologies, systems are built from independent *components*, each of which solves a specific set of tasks and interacts with the environment via a specified interface, i.e., is a *module*. Besides, component-based software technologies facilitate integration of components, automatically linking and running components (quite often, on-the-fly, without interruption of system operation) placed in binary form in the context of certain technological infrastructure (*component environment*). As a result, components can be created and supported by independent developers, which reduces cost of development and maintenance of complex systems.

Component-based technologies made it possible to considerably increase complexity of systems being developed; however, the development of these technologies was not accompanied by the corresponding progress in the quality control of software. As a result, ensuring quality of component systems (including check of correctness of their operation) becomes more complicated as complexity and importance of problems being solved grows.

The source of these difficulties is extremely high effort required for checking correctness of the system, given that the number of possible scenarios of component interaction and the number of implicit dependences between them nonlinearly depend on the number of the components. In turn, the causes of these difficulties are as follows.

- **Present practice of incomplete and inaccurate description of interfaces between the components** gives rise to appearance of many implicit dependences between them. According to D. Parnas [3], one of the founders of the modular approach, complete interface of a module should specify not only its syntactic structure (names of operations, types of their parameters and results) but also its semantics (rules of operation use and expected module behavior). In practice, only syntax is usually fixed; semantics is described partially and ambiguously and only for system interfaces the incorrect use of which is associated with high risks. However, the number of "less risky" interfaces, the semantics of which is not described at all, is much greater, and the total cost of errors in their operation is quite high [4].

- **Difficulties associated with integration of verification tools into software development processes.** Such tools are mainly "all-in-one" tools or integrated solutions with a great number of functions. The set of activities aimed at controlling software quality, relationships between them, and supported verification techniques are predetermined by the tool designers and cannot be considerably modified or extended. However, processes of software development in different organizations differ considerably; therefore, there is a need in modular development tools that can easily be integrated with other tools and be supplemented with new modules implementing modern techniques of software development and analysis. Complex sys-

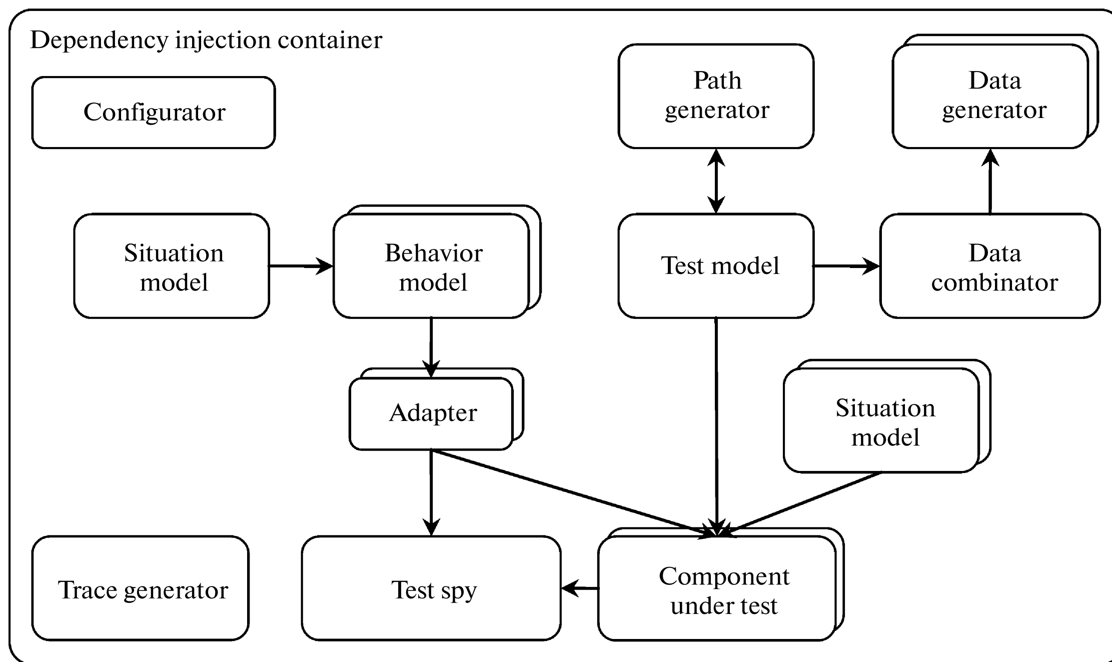

Fig. 1. Test system construction scheme.

tem development tools should also be component ones and should require minimum expenditures in order to be included in various technological processes. Among already existing verification tools, only *unit testing* tools [8] (the most well-known example is JUnit [6]) possess these properties. No modular support for more complicated kinds of verification or conformance testing is currently available.

• **Weak support of reuse of quality control artifacts** (tests, verified statements, modules, etc.). This is especially important for component-based technologies, where the cost of verification of one component is usually quite acceptable and, in the course of the verification, many auxiliary artifacts are created, However, there is no way to use these artifacts in the verification of interacting components or for much more expensive quality control of systems that include this component under check. The reuse of such artifacts could greatly reduce cost of verification of component-based software.

The motivation for this work was our assurance that systematic use of component-based technologies for constructing verification systems will help us to overcome difficulties associated with the gap between complexity of modern systems and possibilities for controlling their quality. A way out of the situation is to create component-based technologies for the verification of software and hardware systems, which ensure independent quality control for separate components in accordance with the requirements imposed on them, and, then, use the artifacts obtained in the course of the verification for more accurate control of their integration and quality of the whole system, gradually increasing the number of the components and subsystems checked, as well as their complexity. These technologies should extend the existing component-based development technologies and make use of elements the developers are familiar with.

Flexibility of a technology and possibility of its use for solving various problems depend on the architecture underlying it. For the component-based technologies, architecture is even more important, since it directly determines rules of creating, developing, and adding new components, their kinds, patterns of their interactions, possible configurations, and so on.

One of the promising approaches to systematic check of complex system correctness is model based testing, which combines the rigor of formal methods and accurate error detection with flexibility and practicality of ordinary testing.

In this work, some elements of a Java-based component verification technology that uses model based testing are presented. On the whole, the presented technology is still under development. The main part of the paper is devoted to its architecture framework, which includes instrumental libraries and a collection of basic component types and rules of their integration. Such component architecture for a model based testing is used for the first time, although a number of its elements were borrowed from most advanced unit testing tools.

In the second section, a model based testing approach, requirements it imposes on the architecture of the supporting tools, and available implementations of the approach are discussed. Next, principles of the

```
public class AccountContract
{
  int balance;
  int maxCredit;

  Account checkedObject;

  public void setCheckedObject(Account checke-
  dObject)
  {
    this.checkedObject = checkedObject;
    this.balance       = checkedObject.getBalance();
    this.maxCredit     = checkedObject.getMaxCredit();
  }

  public boolean possibleTransfer(int sum)
  {
    if (balance + sum > maxCredit) return true;
    else                           return false;
  }

  public boolean transferPostcondition(int sum)
  {
    boolean permission =
      checkedObject.getValidator().validateTransfer(checkedObject, sum);

    if (Contract.oldBooleanValue(possibleTransfer(sum)) && permission)
      return
         Contract.assertEqualsInt(Contract.intResult(), sum
                  , "Result should be equal to the argument")
      && Contract.assertEqualsInt(balance, Contract.oldIntValue(balance)+sum
                  , "Balance should be increased on the argument")
      && Contract.assertEqualsInt(maxCredit, Contract.oldIntValue(maxCredit)
                  , "Max credit should not change");
    else
      return
         Contract.assertEqualsInt(Contract.intResult(), 0
                  , "Result should be 0")
      && Contract.assertEqualsInt(balance, Contract.oldIntValue(balance)
                  , "Balance should not change")
      && Contract.assertEqualsInt(maxCredit, Contract.oldIntValue(maxCredit)
                  , "Max credit should not change");
  }

  public void transferUpdate(int sum)
  {
    if(   possibleTransfer(sum)
       && checkedObject.getValidator().validateTransfer(checkedObject, sum))
      balance += sum;
  }
}
```

Fig. 2. Account basic functionality model.

```
public class AccountLogSpy
{
  int balance;
  int maxCredit;

  Account checkedObject;
  AuditLog logSpy;

  public void setCheckedObject(Account checkedObject)
  {
    this.checkedObject = checkedObject;
    this.balance       = checkedObject.getBalance();
    this.maxCredit     = checkedObject.getMaxCredit();
    logSpy = Mockito.spy(checkedObject.getLog());
    checkedObject.setLog(logSpy);
  }

  int oldBalance;
  boolean wasPossible;

  public boolean possibleTransfer(int sum)
  {
    if (balance + sum > maxCredit) return true;
    else                           return false;
  }

  public void initSpy(int sum)
  {
    Mockito.reset(logSpy);
    oldBalance = balance;
  }

  public void transferLogSpy(int sum)
  {
    boolean permission =
      checkedObject.getValidator().validateTransfer(checkedObject, sum);

    if (wasPossible && permission)
    {
      Mockito.verify(logSpy).logKind("SUCCESS");
      Mockito.verify(logSpy).logNewBalance(balance);
    }
    else if (!permission)
      Mockito.verify(logSpy).logKind("BANNED");
    else
      Mockito.verify(logSpy).logKind("IMPROPER");

    Mockito.verify(logSpy).logOldBalance(oldBalance);
    Mockito.verify(logSpy).logSum(sum);
  }

  public void transferUpdate(int sum)
  {
    if(   possibleTransfer(sum)
      && checkedObject.getValidator().validateTransfer(checkedObject, sum))
    {
      wasPossible = true;
      balance += sum;
    }
    else
      wasPossible = false;
  }
}
```

Fig. 3. Model of the work with an audit log.

component-based design are formulated, and basic elements of the proposed architecture framework are presented. Then, an example of test creation with its help is given. The concluding section summarizes the discussion and outlines directions of future development of the ideas discussed.

2. MODEL BASED TESTING AND TESTING TOOLS

Model based testing [7, 8] is an approach to testing in the framework of which tests are constructed manually, in an automated way, or completely automatically on the basis of a behavior model of the system under test and a model of situations associated with its operation.

The *behavior model* formalizes requirements to the system under test, i.e., describes what external actions and in what situations are admissible and how the system should react on these actions.

The behavior model is a basis for a test oracle [9–11], a component that estimates system behavior in the course of testing.

Most often, various automaton models—(extended) finite-state machines, transition systems [7], Statecharts [12, 13], timed automata [14, 15], etc.—are used for testing. It is often possible to use models of other kinds: contract specifications in the form of pre- and postconditions of operations, algebraic specifications in the form of equivalence rules for various chains of operation calls, or trace models describing possible sequences of actions and system reactions.

The *situation model* formalizes structure of possible test situations components of which are external actions and states of the system and its environment. It defines various classes of situations and their importance from the quality control standpoint. Usually, such a model specifies a finite set of situation equivalence classes, assuming that, upon testing, it is sufficient to check system operation in at least one situation from each class. Sometimes, the situation model describes a finite set of elementary events, with each class being associated with some set of such events. An example of such an event is execution of a given instruction in the code of the system being tested. In more complicated cases, the classes defined may intersect; they also may have weights showing importance of checking a situation from the class.

Situation models are used for solving two closely related tasks: selection of test adequacy or test completeness criterion [16] and selection of numeric metrics of test completeness. If determination of test completeness is based on coverage of classes of equivalent situations or elementary events, we speak of a *test coverage criterion* and *test coverage metrics*, and the percentage of classes covered achieved in the course of testing is called *test coverage*.

Model based test construction consists in the following: a behavior model of the system under test is created (extracted from project documents or taken from somewhere else in a ready-to-use form), a situation model reflecting main project priorities and making use of structural elements of the behavior model is developed, and a test suite is constructed (generated automatically, constructed manually or with the help of tools). These tests check correspondence between the actual behavior of the system under test and its behavior model. In so doing, the test set is created in such a way that it satisfies the test completeness criterion specified by the situation model.

Methods used for the test construction can be classified into three groups.

- *Probabilistic methods* use pseudorandom generation of data of elementary types, pseudorandom aggregation of these data into more complicated structures, and pseudorandom generation of sequences of test actions if it is required to check system behavior in various states. Note that test completeness is ensured owing to the large number of tests, since construction of a separate test is not a labor-consuming task. The situation model here is the probability density of various events.
- *Targeted methods* construct test data and sequences purposefully such that they satisfy certain features (most frequently, implement situations from classes specified by the situation model). When using methods of this kind, the almost minimal number of tests is required to meet the test completeness criterion. However, construction of each test usually requires much effort of the operator and/or great computational resources.
- *Combinatorial methods* construct test data and sequences by combining various elements of these sequences following certain schemes. The creation of one test in this case requires much less expenditures compared to the targeted construction and a little bit more effort than that in the case of the probabilistic construction. The test set is bigger than that in the targeted testing but much less than that in the probabilistic testing (and has greater variety than the latter). In addition, the set obtained has no identical tests, which often appear in randomly generated sets. The combinatorial methods efficiently filtering out tests that do not contribute into achieving the test completeness criterion are comparable in terms of efficiency with the targeted methods. An example of such techniques is test construction based on *automaton models*, which creates tests in the form of a set of paths in the graph of the automaton transitions by minimiz-

```
public class AccountCoverage extends AccountContract
{
  public void transferCoverage(int sum)
  {
    boolean permission =
      checkedObject.getValidator().validateTransfer(checkedObject, sum);

    if (possibleTransfer(sum)) Coverage.addDescriptor("Possible transfer");
    else                       Coverage.addDescriptor("Too big sum");

    if (permission)            Coverage.addDescriptor("Permitted");
    else                       Coverage.addDescriptor("Not permitted");

    if(balance == 0)           Coverage.addDescriptor("Zero balance");
    else if(balance > 0)       Coverage.addDescriptor("Positive balance");
    else                       Coverage.addDescriptor("Negative balance");

    if(sum == 0)               Coverage.addDescriptor("Zero sum");
    else if(sum > 0)           Coverage.addDescriptor("Positive sum");
    else                       Coverage.addDescriptor("Negative sum");
  }
}
```

Fig. 4. Situation model.

ing it from the point of view of covering all possible states and transitions.

2.1. Requirements to Model Representation and Testing Tools

Model based testing tools work with some representations of behavior and situation models. In order to be successfully used in the framework of this approach, these representations should possess the following properties.

- Behavior models.

  ◦ The behavior models should have means for describing requirements in different styles: as pure declarative constraints (constraints on input data of operations and their results), as contract specifications (pre- and postconditions of operations) that use model state, and executable models that determine the way the system under test operate on a higher abstraction level. This requirement is explained by the necessity to express requirements formulated in a natural language as clear as possible, permitting reformulation in another style in the course of model construction, since this makes it possible to reveal inaccuracies, inconsistency, and incompleteness of original formulations and make them more understandable [17].

  ◦ The behavior models should have means to explicitly indicate links to documents and standards containing the requirements. This makes it possible to track relationships of the original requirements with models and automatically generated tests.

  ◦ The behavior models should have possibility to be automatically transformed into test oracles.

  ◦ The behavior models should be capable of using model structures for specifying situation models.

  ◦ There should be possible to use these models in the frameworks of other verification techniques (for example, model checking or static analysis).

  ◦ It is useful to have a possibility of transforming a behavior model into unambiguous and complete documentation of the system being modeled. However, the ways this can be achieved are beyond the scope of this paper.

- Situation models.

  ◦ The situation models should have means for describing situations related to various aspects of behavior and structure of the system under test. This includes structure of input data and results, system states, executable code instructions and functions called, interaction patterns inside the system, possible system errors, etc.

  ◦ The situation models should have means for explicit indication of relationships of defined classes of situations with the original requirements to the system.

  ◦ The situation models should have possibility of automatic transformation into components determin-

ing the class of a current situation and measuring the coverage achieved.

◦ The situation models should have possibility of automatic extraction from the code of the system under test or its models for targeted construction of the corresponding tests.

The requirements to model representations are classified into three types: expressiveness sufficient for practical purposes, possibility of requirements tracking, and possibility to automate or facilitate solution of various testing tasks. These tasks are presented in the following list.

• Automatic test execution. Without automation of test execution, it is impossible to organize systematic testing of complex systems.

• Verification of testing results. When a great number of tests are performed automatically, it is impossible to check their results manually. Therefore, special test components (oracles) are required to automatically estimate whether the system under test behaves correctly in each particular case.

• Creation of test data and test sequences. Both are important if the behavior of the complex system under test depends on its state. Both test data and test sequences should be formed such that test completeness according to the situation model used increases. Automated test construction suggests that necessary sequences of calls and data of individual calls are created automatically.

• Test sets configuring. A test suite for a complex system consists of a great number of tests aimed at checking different aspects of its behavior and different structural elements. In order that such a suite could efficiently be used in the course of system development, flexible configuring tools meeting the constraints on the interaction between different tests are required, which allow one to easily determine in each particular case what tests need to be performed and what tests are not required.

2.2. Unit Testing Tools

Let us consider existing testing tools from the point of view of their use in a component-based technology.

The most appropriate among them in this regard are *unit testing* tools [5]. The most well-known tool of this kind is JUnit [6], which is written in Java and designed for testing codes in this language (historically, the first tool was SUnit [18, 19], which was designed for programs written in Smalltalk).

These tools are characterized by high flexibility, possibility of adding absolutely independent units, and possibility of using them in more complex test systems. One of the unit testing tools possessing the richest functionality is TestNG [20, 21]. Its basic characteristics are as follows.

◦ Basic test elements are test classes and test methods described as Java classes and methods with annotation *Test*.

◦ A test set has a hierarchical structure: it is composed of test suites consisting of tests. Further in this list, these terms are used only in the sense specific to TestNG. Tests and tests suites are determined by *test configuration*, which is described in some XML-based format. A test is composed of methods selected either based on their names or membership in *groups* specified in annotations of the methods.

◦ Set-up and tear-down methods used for initializing data before test execution and releasing resources afterwards can be defined for all—suites, tests, classes, and methods—hierarchy elements.

◦ It is possible to specify dependences between test methods and groups, which control the order of their execution and cancel test method execution if one of the methods on which it depends was executed with an error.

• Test data and objects.

◦ Test methods in TestNG can be parameterized. The set of parameter values used in the framework of a test is indicated by means of an additional annotation or in a configuration file and should be represented either as a collection of objects or as a sequence of results returned by some method upon test execution.

◦ It is possible to create factories of objects that construct various objects from the test classes. All methods included in a test are executed for each object of this kind.

• Verification of testing results.

◦ Like in other unit testing tools, verification is performed by means of calls of special assertion methods. Each method of this kind (their names begin with word assert) checks an elementary property of its arguments (equality, inequality null, membership of an object in a collection, etc.) and, upon its violation, outputs a message in the trace, which is also presented as an argument.

◦ Additionally, TestNG supports indication of possible exceptions and constraints in the method annotation during execution of a test method.

Unit testing tools widely use independent units for solving specific problems: for example, dbUnit [22] for organization of work with databases and httpUnit [23] for processing HTTP queries. For a clearer notation of checks performed in tests (similar to formulation in a natural language), one can use libraries included in behavior driven development tools, e.g., JBehave [24] or NSpecify [25], for organization of test stubs, such tools as Mockito [26], EasyMock [27], and the like.

```
@Test public class AccountTest
{
  Account account;
  boolean permission = true;

  @Mock Validator validatorStub;

  public AccountTest()
  {
    MockitoAnnotations.initMocks(this);
    Mockito.when(validatorStub.validateTransfer(Mockito.<Account>any()
                                , Mockito.anyInt())).thenReturn(true);
  }

  public void setAccount(Account account)
  {
    this.account = account;
    account.setValidator(validatorStub);
  }

  public Validator getPermitterStub() { return validatorStub; }

  @State public int getBalance() { return account.getBalance(); }

  @State public boolean getPermission() { return permission; }

  @Test
  @DataProvider(name = "sumArray")
  @Guard(name = "bound")
  public void testDeposit(int x)
  {
    account.transfer(x);
  }
```

Fig. 5. Test model.

2.3. Model Based Testing Tools

Model based testing tools can be divided into three groups from the point of view of how well they suit the modular architecture.

• Traditional "monolith" tools using specific languages for model formulation. The addition of new components into them can be done by only their developers, and the use of these tools in the framework of a wider tool is possible only if their designers took care of appropriate interfaces.

Among the tools of this kind, almost all research model based testing tools and a number of more robust tools used in various projects—TorX [28, 29], TGV [30, 31], BZ-TT [32], and Gotcha-TCBeans [33, 34]—are classified. All of them make use of various automaton models. The same principles underlie commercial tools Conformic Qtronic [35] and Smartesting Test Designer (earlier version had name Leirios) [36].

• "Monolith" tools based on extensions of widely used programming languages. Examples of this kind are tools CTESK and JavaTESK supporting the UniTESK technology [37–39] and SpecExplorer [40, 41] developed by Microsoft Research. In both cases, to model behavior, combinations of automaton models and contract specifications are used.

• Tools that use ordinary programming languages for model formation and possess some modularity characteristics. In particular, they are easily integrated with components produced by independent designers and can be used in the framework of wider tools.

Tools of this kind appeared only recently, about four or five years ago. The two most well-known examples are ModelJUnit [8, 42] and NModel [43, 44]. A similar tool mbt.tigris.org [45] uses graphic notation for model description, and, therefore, is less suitable for using in the framework of other tools. A different example is the CodeContracts library [46, 47] developed by Microsoft Research for description and check

```
  @Test
  @DataProvider(name = "sumIterator")
  public void testWithdraw(int x)
  {
    account.transfer(-x);
  }

  @Test
  @Guard(name = "bound")
  public void testIncrement()
  {
    account.transfer(1);
  }

  @Test
  public void switchPermission()
  {
    permission = !permission;
    Mockito.when(validatorStub.validateTransfer(Mockito.<Account>any()
                              , Mockito.anyInt())).thenReturn(permission);
  }

  public boolean bound()
  {
    return getBalance() < 5 || !permission;
  }

  public int[] sumArray = new int[]{1, 2};

  public Iterator<Integer> sumIterator()
  {
    return (Utils.ArrayToTypedList(sumArray)).iterator();
  }
}
```

Fig. 5. (Contd.)

of declarative constraints on behavior of .NET components.

The capabilities of the tools of the last kind, which have sound modular structure, are worthy a more detailed discussion.

ModelJUnit and NModel are constructed as extensions of simple unit testing tools. Situation models in these tools are not specified; completeness criteria are implicitly built in the test construction algorithms used. The behavior model is transformed into a dual *test model*, which determines the way the behavior is studied and checked rather than the behavior itself.

• The test model is defined as an extended finite-state machine represented in a programming language (Java or C#) as a test class, like in the unit testing tools. Such a class is either marked by an annotation (attribute in C#) or implements some interface. Elements of the automaton model—states, transitions, and transition guard conditions—are defined by means of methods and fields of this class.

◦ A state of a test model in NModel is given by a set of values of fields of the test class (it is possible to define fields not included in the state) and by the result of method getState in ModelJUnit.

◦ *Actions* the execution of which corresponds to transitions in the test model are represented by methods marked by a certain annotation (attribute). In NModel, actions can be parameterized, with the set of parameter values used in the framework of the test being indicated by means of an additional annotation in the form of a collection of objects of an appropriate type.

◦ Actions may have *guard conditions*, which must be satisfied in order that the corresponding action could be executed. These conditions are represented as methods with the names constructed from the name of the corresponding method–action and some postfix.

• NModel has the following additional capabilities.

◦ Composition of several models in which actions with one name are executed simultaneously. The mod-

els used in the composition are indicated by listing names of the corresponding classes.

◦ Test model checking. Safety properties correspond to invariants represented as methods with a specific attribute. Liveness properties can be checked by analyzing reachability of states in which specially marked characteristic methods return true.

The CodeContracts library provides means for describing purely declarative constraints on properties of input parameters and results of operations. Modeling of states is not supported. The following capabilities are available.

• Constraints are written in C# as Boolean expressions passed as arguments of library methods.

• Constraints of the following types can be described:

◦ preconditions of operations (Contract.Requires method);

◦ postconditions of operations upon normal operation (Contract.Ensures);

◦ postconditions of operations upon exceptions (Contract.EnsuresOnThrow);

◦ assertions on fulfillment of constraints at some point of the code inside a method (Contract.Assert);

◦ class invariants (methods invoking Contract.Invariant marked by a specific attribute).

• In addition to standard expressions in C#, the following ones can be used:

◦ quantifier expressions in the form of calls to Contract.Exists and Contract.ForAll;

◦ access to expression values before execution of the method under test in the postconditions in the form of call of the Contract.OldValue method;

◦ access to the value of the result in the postconditions by means of Contract.Result.

• The CodeContracts library is supplemented with two tools: one for static check of formulated constraints based on theorem proving and another for dynamical check in the course of execution of methods the constraints on which are described.

2.4. Summary of the Survey of the Existing Tools

It can be seen from the survey presented that the model based unit testing tools are actively developing; however, their capabilities are worse than those of the unit testing tools. The existing developments should be supplemented by the following features.

• **Explicit definition of the behavior model of the component being checked independent of the test model.** This separation is necessary to support modeling accuracy and completeness. It also provides an opportunity to construct various tests aimed at achieving different goals based on one behavior model both for this component and for subsystems containing it. Another useful feature is the possibility of using such models in other verification techniques. The CodeContracts library is an example of the behavior model separation.

• **Extended expressive capabilities for describing behavior models.** In particular, it is required to ensure possibility of using different modeling styles and different test construction techniques. This requirement results from complexity and diversity of requirements to modern software [48, 49]. The CodeContracts library allows one to specify constraints only on input data and results, which considerably reduces its use in practically significant systems the behavior of which depends on the state.

• **Explicit definition of the situation model for test construction.** Implicit specification of the situation models in the existing tools restricts possibilities of using various test completeness criteria and makes their selection more difficult for the test designers. The explicit selection of these models will make it possible to combine various completeness criteria with regard to coverage of both the code being checked and the requirements to it.

• **Means for explicit tracing of models to requirements in a natural language.** Capabilities of this kind, which are necessary for the requirement tracking, are available in the CTESK and SpecExplorer tools. In the unit testing means, one can use text messages in the assertion verification methods for this purpose; however, NModel and ModelJUnit lack this possibility.

• **Joint use of models of behavior, situations, and tests related by various aspects of functionality of one component based on noninvasive composition.** This greatly simplifies reuse of models in tests and other verification techniques. NModel partially solves this problem for the test models owing to the possibility of extending definitions of classes by new fields and methods in C#.

3. ARCHITECTURE FRAMEWORK FOR MODEL BASED TESTING

In this section, the proposed architecture of the model based testing tools satisfying the above-formulated requirements is described. First, we make some comments related to the selection of means for solving the posed problems.

• As has already been noted, it is useful to employ different techniques for describing behavior of the components under test. However, it seems impossible to support all kinds of models in one technology. Therefore, it is useful to identify at once practically important approaches that can simultaneously be supported in the framework of one technology.

In modeling program interfaces with known requirements, it is convenient to use *contract specifications* in the form of pre- and postconditions relying on the model states of the components. The approaches based on such specifications demonstrate good scalability and efficiency in terms of working hours

required for description of some set of interface elements [38, 50, 51].

On the other hand, to model float-point calculations, complex protocols, and some other software, it is sometimes more convenient to use *executable models*, which are simpler implementations of the same functionality. The most convenient models of this kind in practice are extended finite state machines and transition systems, as well as their compositions.

In modeling some reactive systems processing large streams of events or services regularly processing data from large databases, *data-flow contracts*, which describe constraints on the processing of one element in the input data stream rather than on the final result, turn out useful.

• The behavior and situation models, as well as the tests themselves and elements of the development tools, should have form of components or sets of components (subsystems) in the framework of the selected component-based technology with minimal addition of new constructs requiring additional language and instrumental support.

Such an approach allows one to apply all tools, means, and techniques provided by the base component technology to work with these models and their integration with the components under test. This greatly reduces expenditures on the development of tools supporting such a technology and makes it possible to use the same tools and created component models in the construction of tests for large-scale systems. This lays foundation for the fulfillment of the requirements to the component-based verification technologies.

• The application of widely used component-based technologies and programming languages makes it easier to learn how to use the tools and allows one to take advantage of using numerous auxiliary libraries designed for solving specific problems of unit testing.

• It often happens that verification systems (which include great variety of components) are more complicated than those for verification of which they have been created. Therefore, it is important to reduce their complexity and effort required for their design. To facilitate integration and reconfiguration of systems consisting of many components, it is recommended to use whenever possible noninvasive component integration techniques, which avoid introduction of modifications in the codes of the components. This can be achieved through wide use of the dependency injection pattern [52] and libraries–containers supporting it.

3.1. Types of Components and Integration of Them

We propose to develop a model based testing tool on the basis of the *dependency injection container*, which makes it possible to specify the list of components of the test system, initialize them, and externally define relationships between them, without intervention into codes of these components.

Verification systems are constructed from components of different types.

• Components to be verified.

These components should support possibility of external initialization with the help of the dependency injection container. If this requirement is not fulfilled, it is usually not difficult to write a component–wrapper that meets this requirement and provides access to the operations of the original component to be verified.

The components under check do not depend on the test system, except for using stubs, which replace components needed for their operation.

• Behavior models (generalized contracts).

These components are constructed in the basic programming language as classes with several methods performing certain roles. For example, in a purely declarative specification, pre- and postconditions should be defined (note that any method without side effects that returns a Boolean value may play these roles). For a specification that uses component model state, a state synchronizer is necessary to synchronize the model state with the actual state of the component under check. Executable specifications must define the preconditions and changes of the states.

The behavior models depend on the components under test or, if interfaces of the model and component differ, on the adapters eliminating such differences.

• Interaction models.

When describing many-component systems, in addition to the models of separate components, it is required sometimes to explicitly introduce models of their interaction, which allow one to estimate correctness of complex sets of actions and reactions in which several components take part, with each of them being familiar with only a part of the events. For example, the so-called *interleaving semantics* is an interaction model for asynchronous interactions of components working in parallel. In the framework of this model, any set of events is correct if it can be linearly ordered such that each separate event under this ordering is correct with respect to models of the components that create or process it [50].

Interaction models are constructed in the form of template library modules attached in the configuration file to the corresponding groups of components. For each particular interaction, an instance of such a pattern is generated, which depends on the behavior models of the components taking part in this interaction.

• Situation models.

The situation models are constructed in the basic programming language as methods that monitor occurrences of certain situations after checking constraints describing them. For some operations, situation models may include both pre-situations, which are determined by arguments of the operation and

states of the components before the operation was called, and post-situations, which correspond to certain properties of the results and states after the operation was performed. Like contract specifications, models of post-situations may have model states and methods—synchronizers.

The situation models depend on the components being verified, behavior models, or test models, depending on what terms are used to describe the situation.

The situation models can automatically be extracted from the behavior models, interfaces and codes of the components under test, since criteria of test completeness based on code structure or functionality are often used in test construction. These generated components are further referred to as *secondary* components.

• Tests.

Like the situation models, the tests can be created by the designers or generated from the behavior models and interfaces of the components under test. Each test should define a sequence of calls to operations of the component under test (which may consist of only one call) and values of parameters of these calls and test data.

◦ The first task can be solved with the help of two approaches.

■ In practice, a combination of an automaton model of the test and a generator of paths in the automaton transition graph is most frequently used. This technique underlies ModelJUnit and NModel.

In this case, the automaton model of the test must determine methods that play roles of actions and guard conditions, as well as return current state of the automaton. The test model may depend on the component under test or on its behavior model.

■ In a number of cases, it is more efficient to use a monolith generator of sequences based on the information on the interface under test.

◦ Generation of test data, especially data of complex structure, can use many components playing different roles.

■ Primary generators, which construct objects of a certain type. Such a generator may be organized as an iterator of some collection.

■ Filters discarding data that do not satisfy certain constraints.

■ Constraint solvers that directly construct data satisfying certain constraints.

■ Combinators constructing complex data from simple objects.

■ Transformers generating data of some type from a simpler coded representation of these data.

• Adapters.

The adapters alleviate possible differences between interfaces of models and the components they model.

The adapters depend on the components under test. If they are responsible for synchronization of model states, they depend also on the corresponding behavior model.

It should be noted that, in many cases, small differences between the model interface and verified interface do not require writing an adapter and can be eliminated by indicating a library transformation procedure. This refers to the cases where the differences reduce to the lack of a number of parameters, permutation of parameters, or simple type transformations, e.g., numbers to strings and vice versa. In all these cases, the adapter is constructed automatically, by the indication of the corresponding transformation in the configuration file of the test system, rather than manually.

• Stubs (test doubles).

Stubs replace components that affect those under test. They are classified into two types.

◦ *Mock passes* some data in the form of results returned by its methods to the component under test and serves as an additional source of actions on it.

◦ *Test spy* records calls of its operations and their arguments for checking correctness of calls made by the component under test.

Theoretically, one stub can simultaneously play both roles; however, this is seldom required in practice (this may be an evidence of too complicated test organization, which, possibly, needs modifications).

The component under test depends on the stubs used. The test and behavior model depend on the test spy they use. Vice versa, the mock itself depends on the test, since the test determines results of the current call of its operations.

• Auxiliary components. The auxiliary components include those that solve numerous tasks of organization of the verification system work, integration of its components, and collection of information about the events.

◦ Tracers of various events. Basically, a separate event tracer is attached to each component. All tracing messages are collected by one or several trace generators.

◦ Planners of tests that include asynchronous actions responsible for the creation of separate processes and threads inside the test system and distribution of actions on them.

◦ Dispatchers and synchronizers of separate operations in asynchronous and parallel tests.

◦ Configurators determining links inside a group of components.

Figure 1 demonstrates one possible configuration of a test system based on the proposed architecture. Links between the components depicted in the figure are dependences typical of components of the given type (not all possible dependences are shown). Links of the trace generator and configurator are not shown, since all, or almost all, components are related to them.

A particular set of components and links between them are described in the configuration file in the XML format that comes to the input of the dependency injection container, which initializes all components and links between them as required. Such a method of links specification allows us to construct various configurations of the test system without modifying codes of its components and even having no access to them. On the other hand, it is possible to define fixed links in the code itself, as well as to apply flexible linking by means of annotations and to create special components–configurators that contain explicit initialization of the components and links between them in the basic programming language.

3.2. Implementation of the Proposed Approach

For the basic programming language to implement the proposed architecture, Java was selected. It possesses features necessary for description of various roles of classes and methods, as well as links between the components, such as means for description of declarative information about elements of the code in the form of annotations and support in obtaining dynamical information on the structure of the components and signatures of their operations (introspection or reflection). Some other languages, say C#, also possess these features.

For the dependency injection container, the open library Spring [45, 46] was selected, which supports many functions of systems of this type.

To describe behavior models, a small library was developed. On the one hand, it is similar to the assertion verification libraries used in the unit testing tools (various assert() methods are used); on the other hand, it reminds Microsoft CodeContracts (to access the operation result and expression pre-values, methods result() and oldValue() are used). In contrast to CodeContracts, creation of contract specifications that use the model state is supported. In the created library, quantifier expressions (which are available in CodeContracts) are not used, and static analysis of constraints is not supported.

A small library was also created for describing the situation models to ensure tracking of information about coverage of specified situations.

Tests are represented in the style that is similar to that in ModelJUnit and NModel but has some extensions borrowed from TestNG.

- An extended hierarchy of test elements—test suites, tests, test classes, test methods—is supported. A test suite consists of tests; one test may include several test classes.
- A test class describes an extended finite state machine.
- A state of this finite state machine is a compound result returned by all methods marked by annotation State. This makes it possible to add new elements in a state without modifying the earlier written code. The test state is a list of states of classes containing in it.
- Test methods define actions (possibly, parameterized) of the finite state machine. Parameter values are extracted from the data provider associated with the test method. The provider may be a generator of sets of values defined, for example, as elements of some collection or may be constructed dynamically from data generators for different parameters with certain strategy of their combining (select all combinations, every value of each parameter at least once, all possible pairs of values, and the like). The data providers and the method of their combining are given by means of annotations of the method and its separate parameters.
- Actions may have guard conditions in the form of methods returning a Boolean result and depending on the state of an object from the test class. A guard condition is attached to the test method by means of annotations without using agreements on method naming. Hence, one and the same condition can be used for different methods, and one method may have several guard conditions. Besides, the parameter of a guard condition may be any set that is the beginning of the parameter set of the corresponding method, including the empty set (in this case, the guard condition depends on only the current state).
- Like in TestNG, any test element—suite, test, class, or method—may have set-up and tear-down methods. In addition, it is possible to define configuration methods called upon attending a current state.

To construct stubs, the open library Mockito [26] is used. It has sufficiently rich capabilities of defining mocks and test spies and uses intuitively clear syntax for their description. This example shows that, upon presence of a Java library with the required functionality, it can easily be used in the framework of the proposed architecture.

4. EXAMPLE OF TEST CONSTRUCTION

Below is an example of use of the proposed approach for constructing tests for a simple implementation of functionally of a bank account. The interface of the component under test is presented below.

```
public interface Account
{
  int getBalance();
  int getMaxCredit();

  Validator getValidator();
  void setValidator(Validator p);

  AuditLog getLog();
  void setLog(AuditLog log);

  int transfer(int sum);
}
```

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                  http://www.springframework.org/schema/aop
                  http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <bean id="accountImpl" class="mbtest.tests.AccountImpl"></bean>

    <bean id="accountTest" class="mbtest.tests.AccountTest">
      <property name="account" ref="accountImpl"/>
    </bean>

    <bean id="accountContract" class="mbtest.tests.AccountContract">
      <property name="checkedObject" ref="accountImpl"/>
    </bean>

    <bean id="accountCoverage" class="mbtest.tests.AccountCoverage">
      <property name="checkedObject" ref="accountImpl"/>
    </bean>

    <bean id="accountLogSpy" class="mbtest.tests.AccountLogSpy">
      <property name="checkedObject" ref="accountImpl"/>
    </bean>

    <bean id="accountContractExecutor" class="mbtest.contracts.ContractExecutor">
      <property name="postcondition"
                value="mbtest.tests.AccountContract.transferPostcondition"/>
      <property name="updater" value="mbtest.tests.AccountContract.transferUpdate"/>
      <property name="object" ref="accountContract"/>
    </bean>

    <bean id="accountCoverageExecutor" class="mbtest.coverage.CoverageExecutor">
      <property name="coverage"
                value="mbtest.tests.AccountCoverage.transferCoverage"/>
      <property name="updater" value="mbtest.tests.AccountCoverage.transferUpdate"/>
      <property name="object" ref="accountCoverage"/>
    </bean>
```

Fig. 6. Test system configuration.

Methods `getBalance()` and `getMaxCredit()` are used to obtain the current balance and maximum possible credit. If the balance is negative, it cannot exceed the maximum credit in absolute value.

Method `int transfer()` transfers money from/to the account, depending on the sign of its argument. If the argument is positive, the corresponding sum is added to the account, increasing the balance. If it is negative, this sum is withdrawn, assuming that the balance does not go beyond the limit. The result of this method is the sum transferred or 0 if the transfer was not performed.

The current account makes it possible to use a special transaction validator `Validator`, which is questioned upon any transfer by means of the method `boolean validateTransfer(Account a, int sum)` it provides and may permit or ban the transfer.

Another account function is writing information about attempts of money transfer in a log for the subsequent audit. In this case, interface methods `AuditLog`:

```
  <bean id="accountSpyExecutor" class="mbtest.contracts.SpyExecutor">
    <property name="initialization" value="mbtest.tests.AccountLogSpy.initSpy"/>
    <property name="postcondition"
              value="mbtest.tests.AccountLogSpy.transferLogSpy"/>
    <property name="updater" value="mbtest.tests.AccountLogSpy.transferUpdate"/>
    <property name="object" ref="accountLogSpy"/>
  </bean>

  <aop:config>
    <aop:aspect id="accountContractAspect" ref="accountContractExecutor">
      <aop:pointcut id="accoutTransfer"
                    expression="execution(* mbtest.tests.Account.transfer(..))"/>

      <aop:around pointcut-ref="accoutTransfer" method="execute"/>
    </aop:aspect>

    <aop:aspect id="accountCoverageAspect" ref="accountCoverageExecutor">
      <aop:pointcut id="accoutCTransfer"
                    expression="execution(* mbtest.tests.Account.transfer(..))"/>

      <aop:around pointcut-ref="accoutCTransfer" method="execute"/>
    </aop:aspect>

    <aop:aspect id="accountSpyAspect" ref="accountSpyExecutor">
      <aop:pointcut id="accoutSTransfer"
                    expression="execution(* mbtest.tests.Account.transfer(..))"/>
      <aop:around pointcut-ref="accoutSTransfer" method="execute"/>
    </aop:aspect>
  </aop:config>
</beans>
```

Fig. 6. (Contd.)

`logKind(String s)`, `logOldBalance(int b)`, `logSum(int sum)`, `logNewBalance(int b)` are invoked, which record the result of the transaction (SUCCESS, in the case of a successful transfer; BANNED, if the transfer was banned by the validator; and IMPROPER, if there was an attempt to withdraw too much money), the previous balance value, the sum transferred, and the new balance value.

The behavior model for the account is described as two independent components: the basic functionality model and the model of work with audit log. This allows one to modify and check these two groups of constraints independently. The description of the basic functionality—the postcondition of method `transfer()` and the corresponding synchronizer of the model state—is presented in Fig. 2.

The description of the requirements to the work with the log for audit is shown in Fig. 3. It uses the freely distributed library of stubs Mockito and installs a stub to observe calls between the account and the corresponding logger. The stub checks whether the logger methods are called in the required order and with the required arguments. Since the stub has a model state, it also contains a method–synchronizer of this state. The stub must be reinitialized after every call of `transfer()`. To this end, method `initSpy()` is defined in it.

The description of the situation model is presented in Fig. 4. Here, situations are classified in terms of the following four characteristics: transfer correctness, passing the validation, sign of the preceding balance value, and sign of the sum transferred. Since the determination of a situation depends on the model state of the account and needs state synchronizer, this model inherits the functionality models, reusing its code elements.

The test model for the account is presented in Fig. 5.

The test state consists of the following two elements: the current balance and the `permission` field value, which determines results of work of the validator mock. Withdrawal and deposit of money are tested by different methods in spite of the fact that the one and the same method of the object under test is invoked. There are four test methods corresponding to actions in the described finite state machine.

• Method `testDeposit()` checks money deposit on the account. The method is parameterized, and the values of the parameters are taken from the `sumArray` array. Besides, this method has a guard condition that permits its call only when the current balance does not exceed 5 and the validator–stub permits the operation.

• Method `testWithdraw()` checks money withdrawal from the account. The values of its parameter

are taken from the same array with the help of a method–iterator.

• Method `testIncrement()` checks addition of the sum equal to 1 to the account. It has the same guard condition as method `testDeposit()`.

• Method `switchPermission()` checks nothing and only switches the current value of field `permission` to test the work of the account with different balances and different results of transfer validation.

Finally, the configuration file for environment Spring determining links between all listed components is presented in Fig. 6.

In this configuration, it is indicated how to initialize objects of all above-listed types, and, additionally, the binding of postconditions and synchronizers of all models to method `transfer()` is defined with the help of the aspect binding technique supported by Spring.

The example presented demonstrates that the proposed method of constructing a test system from given components is not invasive: all these components know nothing about each other, except for types of objects they directly depend on. In this configuration, the model of basic functionality and the situation model are represented by different objects; however, since the second model inherits the first model, it is possible to implement them by means of one component that plays two different roles.

5. CONCLUSIONS

In the paper, component architecture of model based testing tools is presented. It is constructed on the basis of the component technologies and the principle of noninvasive composition. Although separate elements of the proposed architecture have already been used in some tools, like TestNG, ModelJUnit, and NModel, this is the first presentation of the architecture in its entirety. A Spring-based implementation of the proposed approach relying on the dependency injection technique is described. An example of application of the approach to constructing a test including several models of various aspects of behavior of the component under test is presented.

The existing implementation of the approach discussed has several drawbacks that need to be eliminated.

• First, it is required to modify standard dependency injection context of Spring to make it recognize components specific to the test systems (behavior model, stub, situation model, test model, etc.) and require less parameters for their initialization, as well as automatically construct their aspect binding to the components under test. This will considerably simplify creation and modification of the configuration files (in the example presented, will remove the text in the framework of element `<aop:config>` and the definitions of the last three components). The configuration files will considerably be simplified and may become useful tools for configuring and adjusting complex test suites without recompilation.

• Second, tools for generation of secondary components, situation models, and test models have not been implemented yet. It is assumed that they will be developed on the basis of one of the open libraries for transformations of Java byte-codes. Such an implementation will make it possible to generate secondary components without accessing source codes of their preimages. In this case, convenience of measuring coverage of the code or behavior models in the framework of the proposed architecture will be comparable with that achieved with the use of the leading specialized tools.

• Third, logically distinct elements of the framework for construction of test systems—generators of paths on the automaton model, library data generators, combinators, and the like—should also be separated as externally defined and configured components. This will improve flexibility of possible configurations of the test system and will make it possible to easily integrate new techniques of test construction in the approach discussed.

However, even now, the proposed architecture demonstrates its advantages over traditional "monolith" tools of test construction: high flexibility, possibility of joint use with various libraries and numerous tools designed for work with Java components (frameworks, code analyzers, debuggers, and so on), and possibility of integration in more powerful environments.

REFERENCES

1. Szyperski, C., *Component Software: Beyond Object-Oriented Programming,* Boston: Addison-Wesley, 2002, 2nd ed.
2. Heineman, G.T. and Councill, W.T., *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
3. Parnas, D., Information Distribution Aspects of Design Methodology, *Proc. of 1971 IFIP Congress*, North Holland, 1971.
4. *The Economic Impacts of Inadequate Infrastructure for Software Testing*, Tassey, G., Ed., NIST Report, 2002.
5. Hamil, P., *Unit Test Frameworks. Tools for High-Quality Software Development*, O'Reilly Media, 2004.
6. http://www.junit.org.
7. *Model-Based Testing of Reactive Systems. Advanced Lectures*, Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A., Eds., *Lecture Notes in Computer Science*, vol. 3472, Springer, 2005.
8. Utting, M. and Legeard, B., *Practical Model-Based Testing: A Tools Approach*, Morgan-Kaufmann, 2007.
9. Peters, D. and Parnas, D., Using Test Oracles Generated from Program Documentation, *IEEE Trans. Software Engineering*, 1998, vol. 24, no. 3, pp. 161–173.

10. Hoffman, D., Analysis of a Taxonomy for Test Oracles, *Quality Week*, 1998.

11. Baresi, L. and Young, M., Test Oracles, *Tech. Report CIS-TR-01-02*, http://www.cs.uoregon.edu/~michal/pubs/oracles.html.

12. Harel, D., Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Programming,* 1987, vol. 8, no. 3, pp. 231–274.

13. Drusinsky, D., *Modeling and Verification Using UML Statecharts*, Elsevier, 2006.

14. Alur, R., and Dill, D.L., A Theory of Timed Automata, *J. Theor. Comput. Sci.*, 1994, vol. 126, no. 2, pp. 183–235.

15. Springintveld, J., Vaandrager, F., and D'Argenio, P.R., Testing Timed Automata, *Theor. Comput. Sci.*, 2001, vol. 254, no. 1–2, pp. 225–257.

16. Zhu, H., Hall, P., and May, J., Software Unit Test Coverage and Adequacy, *ACM Computing Surveys*, 1997, vol. 29, no. 4, pp. 366–427.

17. Kuliamin, V.V., Pakulin, N.V., Petrenko, O.L., Sortov, A.A., Khoroshilov, A.V., Requirement Formalization in Practice, *Preprint of Inst. of System Programming, Russ. Acad. Sci.*, Moscow, 2006, no. 13.

18. Beck, K., *Kent Beck's Guide to Better Smalltalk: A Sorted Collection*, Cambridge Univ. Press, 1998.

19. http://sunit.sourceforge.net/.

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

21. http://testng.org/.

22. http://www.dbunit.org.

23. http://www.httpunit.org.

24. http://jbehave.org/.

25. http://nspecify.sourceforge.net/.

26. http://mockito.org/.

27. http://easymock.org/.

28. Tretmans, J. and Brinksma, E., TorX: Automated Model-Based Testing, *Proc. of 1st Eur. Conf. on Model-Driven Software Engineering*, Nuremberg, Germany, 2003, pp. 31–43.

29. http://fmt.cs.utwente.nl/tools/torx/introduction.html.

30. Fernandez, J.-C., Jard, C., Jéron, T., Nedelka, L., and Viho, C., Using On-the-Fly Verification Techniques for the Generation of Test Suites, *Lecture Notes in Computer Science* (Proc of 8th Int. Conf. on Computer-Aided Verification), Springer, 1996, vol. 1102, pp. 348–359.

31. http://www.inrialpes.fr/vasy/cadp/man/tgv.html.

32. Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N., and Utting, M., Z-TT: A Tool-set for Test Generation from Z and B Using Constraint Logic Programming, *Proc. of Formal Approaches to Testing of Software*, Brno, Czech Republic, 2002, pp. 105–119.

33. Hartman, A. and Nagin, K., TCBeans Software Test Toolkit, *Proc. of 12-th Int. Software Quality Week*, 1999.

34. Farchi, E., Hartman, A., and Pinter, S.S., Using a Model-based Test Generator to Test for Standard Conformance, *IBM Systems J.*, 2002, vol. 41, no. 1, pp. 89-110.

35. http://www.conformiq.com/qtronic.php.

36. http://www.smartesting.com/index.php/cms/en/explore/products.

37. Bourdonov, I., Kossatchev, A., Kuliamin, V., and Petrenko, A., UniTesK Test Suite Architecture, *Lecture Notes in Computer Science* (Proc. of FME 2002), Springer, 2002, vol. 2391, pp. 77–88.

38. Kuliamin, V.V., Petrenko, A.K., Kossatchev, A.S., and Bourdonov, 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. 310–322].

39. http://www.unitesk.ru.

40. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., and Veanes, M., Testing Concurrent Object-Oriented Systems with Spec Explorer, *Lecture Notes in Computer Science* (Proc. of Formal Methods Europe), Springer, 2005, vol. 582, pp. 542–547.

41. http://research.microsoft.com/en-us/projects/SpecExplrer/.

42. http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/.

43. Jacky, J., Veanes, M., Campbell, C., and Schulte, W., *Model-based Software Testing and Analysis with C#*, Cambridge Univ. Press, 2007.

44. http://nmodel.codeplex.com/.

45. http://mbt.tigris.org/.

46. Barnett, M., Fahndrich, M., de Halleux, P., Logozzo, F., and Tillmann, N., Exploiting the Synergy between Automated-Test-Generation and Programming-by-Contract, *Proc. of ICSE 2009*, Vancouver, Canada, 2009.

47. http://research.microsoft.com/en-us/projects/contracts/.

48. Kaner, C., Bach, J., and Pettichord, B., *Lessons Learned in Software Testing*, Wiley, 2002.

49. Kuliamin, V.V., Integration of Verification Methods for Program Systems, *Programmirovanie*, 2009, no. 4, pp. 41–55 [*Programming Comput. Software* (Engl. Transl.), 2009, vol. 35, no. 4, pp. 212–222].

50. Kuliamin, V., Petrenko, A., and Pakoulin, N., Practical Approach to Specification and Conformance Testing of Distributed Network Application, *Lecture Notes in Computer Science* (Proc. of ISAS'2005), Berlin, Springer, 2005, vol. 3694, pp. 68–83.

51. Grinevich, A., Khoroshilov, A., Kuliamin, V., Markovtsev, D., Petrenko, A., and Rubanov, V., Formal Methods in Industrial Software Standards Enforcement, *Proc. of PSI'2006*, Novosibirsk, Russia, 2006.

52. Fowler, M., Inversion of Control Containers and the Dependency Injection Pattern, 2004. http://www.martinfowler.com/articles/injection.html.

53. Johnson, R., Hoeller, J., Arendsen, A., Risberg, T., and Sampaleanu, C., *Professional Java Development with the Spring Framework*, Wrox, 2005.

54. http://www.springsource.org.