ISSN 0361-7688, Programming and Computer Software, 2008, Vol. 34, No. 6, pp. 351–363. © Pleiades Publishing, Ltd., 2008.
Original Russian Text © R.S. Zybin, V.V. Kuliamin, A.V. Ponomarenko, V.V. Rubanov, E.S. Chernov, C.C. Sidorov, 2008, published in Programmirovanie, 2008, Vol. 34, No. 6.

Automation of Broad Sanity Test Generation

R. S. Zybin, V. V. Kuliamin, A. V. Ponomarenko, V. V. Rubanov, and E. S. Chernov

Institute for System Programming, Russian Academy of Sciences,
Bol'shaya Kommunisticheskaya ul. 25, Moscow, 109004 Russia

e-mail: phoenix@ispras.ru, kuliamin@ispras.ru, susanin@ispras.ru, vrub@ispras.ru, ches@ispras.ru

Received April 28, 2008

Abstract—The technology for the broad generation of sanity tests for complex software developed in the Institute for System Programming (Russian Academy of Sciences) is presented. This technology is called Azov; it is based on using a database containing structured information about the interface operations of the system under test and on a procedure for enriching this information by refining constraints imposed on parameter types and results of operations. Results of a practical application of this technology prove its high efficiency in generating sanity tests for systems with a large number of functions.

DOI: 10.1134/S0361768808060066

1. INTRODUCTION

Presently, the number of tasks performed by software systems, their importance, and requirements for reliability are growing. For that reason, the complexity of software is ever increasing and the risks related to inevitable errors appearing while developing such systems are growing. The only means for minimizing these risks is to organize accurate and systematic verification of conformance to the requirements at all stages of the software development and maintenance. At the last stages of the software development, such verification is usually performed by testing; that is, by observing the system's behavior and analyzing its correctness in a series of specifically designed situations taking into account all the significant aspects of the target system's behavior.

The currently available testing techniques require considerable effort if one wants to guarantee the completeness of testing. As the complexity of software grows, these efforts increase nonlinearly. The automation of test generation usually cannot considerably reduce such efforts because it involves a formalization of the requirements for the software, which are originally informal, and a classification of test situations.

Sometimes, when testing results are not required to be very reliable and complete, these efforts are unjustified. For example, in the sanity testing, we only want to check that the system does not fail and returns results that pass the simplest correction tests (no complete testing is performed). Such kind of testing is performed to make sure that all the system's functions operate correctly in simple situations before the system is put to more thorough and systematic testing that requires much more effort but is senseless if the system cannot even cope with simple tasks. Such a testing saves effort for detecting and localizing gross bugs in complex software.

Usually, sanity tests are designed manually; however, sometimes a large number of interface operations in the system under test and the availability of complete and well-structured information about them allow automation of test generation. If operations are very numerous, the conventional techniques for test suites development becomes too labor consuming. At the same time, information about the syntax of those operations can be used to automatically generate prototypes of test suites.

Both factors—the large number of operations and the availability of a database containing information about the interface operations—are characteristic of the Linux Standard Base (LSB) [3], which includes several standards (POSIX [1], ISO C [4], Filesystem Hierarchy Standard [5], and others) and libraries (Xlib [6], OpenGL [7], GTK+ [8], and Qt [2]). On the whole, LSB version 3.1 includes over 30000 functions and methods. To maintain the text of the standard and the set of tools for performing various tests, the syntactic information about all the interfaces included in the standard is stored in a unified well-structured database. This makes it possible to use a novel approach to generating sanity tests for the LSB.

2. AUTOMATION TECHNOLOGY FOR SANITY TEST GENERATION

For the cases when the system's interface consists of thousands of operations and the information about the interface elements stored in a well-structured form suitable for automatic processing, an efficient automation technology for requirements-based sanity test generation can be proposed for generating large test suites. Such a technology called Azov was developed in the Institute for System Programming (Russian Academy of Sciences) in 2007.

This technology uses a database containing information about the syntax of operations of the system under test and assumes that this information is augmented; the augmentation is mainly reduced to refining (specializing) the types of operations' parameters and their results. The refinement is performed manually. After the refinement is performed, correct input data for each operation can be constructed and some properties of its result can be checked.

The technology includes the following elements.

• A procedure for refining data about interface operations.

• A database containing refined syntactic information about the operations under test.

• Tools used to add information to the database.

• Sanity test generator that can produce a test suite for a specified set of operations using the information stored in the database.

The main idea underlying the proposed technology is as follows. The information about the parameter types and results of the operations under test is refined so as to make it possible to generate parameter values for simple scenarios of the normal use of the corresponding operations and perform some (far from complete) correctness checks of their results. Since the same data types are used many times in large systems, the generation of large test suites can considerably reduce the effort per each generated test case.

2.1. Initial Data and Expected Results

The initial data for using the proposed technology are provided by the database containing well-structured information about the operations of the system under test (their signatures) and operations documentation. Currently, we assume that all the operations are functions in C or methods in C++. The change of the base programming language for another imperative language usually requires that only the output module of the test generator be modified. If the new language uses methods for passing parameters results that are considerably different from those used in C and C++, a modification of the database structure and test generation algorithms may be needed.

The good structure of the information about operations means that the types of the parameters and operation results are stored as individual entities referenced by the entities corresponding to operations. More detailed requirements for the initial information are presented in Section 2.5 called “Supporting Tools.”

The documentation for the system under test must contain sufficient information to enable a developer to reveal the main scenarios of using all the interface operations including data sources for operations' arguments and basic constrains on their results provided that they operate normally.

The result of working in accordance with the proposed technology is a sanity test suite for all the operations of the system. For each operation, the test suite includes a test calling this operation within a simple scenario of its normal operation that does not cause failures, exceptions or returning error codes if the system operates correctly. Also, the test must check the basic constraints on the operation result. All the arguments for the call must be formed correctly; if necessary, preliminary calls of other operations initializing the internal data must be performed; and all the allocated resources must be freed at the end of the test.

2.2. Organization of the Work in Accordance with the Azov Technology

The test suites generated using Azov consist of test cases in the form of programs that sequentially perform auxiliary operations to prepare the system for work, initialize parameter values for calling the operation under test, call this operation, and finalize the system (i.e., free the resources). In addition, the basic constraints on the results of all the operations performed in the course of the test execution are checked.

The development of test suites using Azov consists of the following stages.

• *Decomposition of the set of operations into functional groups.* The system’s interface is decomposed into groups of operations working on the same internal data and providing a complete set of actions on them. First of all, this is required to divide the development into independent parts that can be assigned to different developers.

• *Refining information about interface operations in the database.* Developers analyze the documentation concerning the operations in the groups assigned to them, determine the conditions of their normal operation, and constraints on the results. The constraints are written to the database in the form of specialized types of parameters and operation results. Each type may have a list of feasible values or actions needed to initialize or destroy data of this type. If, in addition to the operation’s arguments, global system data must be initialized for the normal work of an operation, the corresponding initialization and finalization procedures are specified. The refinement procedure is described below. To fill the database, auxiliary tools working through a Web interface are used that enable one to navigate through the database, find various additional information in it, and edit the data. Before the refinement procedure, the operations under test and the relevant data types are ordered so that the operations having more complex parameter types are placed after those that have simpler parameters and can be used for obtaining more complex data types. The refinement is performed beginning with simple operations and gradually proceeds to more complex ones. With such a procedure, there is no need to often switch to the analysis of other

operations, and the information revealed at earlier stages can be naturally and repeatedly used at later stages.

• *Verification of the quality of refinement.* The correctness of the information placed in the database is checked by reviewing and analyzing it by other developers; The verification is also performed by debugging the generated tests—they must be successfully compiled and assembled, and all the problems in their execution must be caused only by bugs in the system under test.

• *Test generation.* The final sanity test suite is generated using a test generator on the basis of the information stored in the augmented database.

• *Test execution.* The generated test suite can be produced as a single program or a set of programs in C or C++. In the latter case, the programs are executed in batch mode.

• *Analysis of testing results.* After the execution of each test, information about its correct execution, violation of one of the constraints, or about the destruction of the system under test is produced. The detected problem is analyzed by a developer and is either fixed as a bug or implies a modification of some data in the database; in the latter case, the test is executed once more.

2.3. Methodological Foundation of the Technology

The methodological foundation of Azov includes a technique for refining information about the interface operations and types of their parameters and results; it also includes a procedure for automatic test generation based on the refined information.

The information about the interface operations and types of their parameters is refined using the following procedures.

• Refinement (specialization) of types.

◦ If the normal call of an operation requires that an argument value be within a certain set, a specialized enumeration type for the corresponding parameter is defined that has elements of this set as its values.

◦ If the normal call of an operation requires that its argument value (or object if this operation is a method of a class) is a result of another operation, a specialized type is defined that is simultaneously specified as this argument's type and the type of the second operation result.

◦ If the normal call of an operation requires that its argument value (or object) must be used as an argument of other operations before calling the operation under examination, a specialized type is defined for the corresponding parameter and it is associated with an initialization procedure that calls those operations.

◦ If the use of an argument (or call object) requires that certain resources be freed after it is no longer needed, a finalization procedure is associated with the specialized type that frees those resources.

◦ To initialize or finalize certain objects of a given type, additional operations are sometimes required that must be called one time in each test case that uses this type. The procedure performing such additional operations is also considered as an additional attribute of the given type.

◦ When the types of parameters are refined, several parameters are sometimes joined into one abstract object such that its different elements are used as parameter values. In this case, a specialized type for such a composite object is defined.

For example, if a pointer to the beginning of a string of the type `char*` and the length of this string are used as parameters, then a specialized type called "string" may be defined. Then, the first argument is the pointer to the first element of the string and the second parameter is the result of applying the function `strlen()` to the first parameter.

When such composite types are defined, a code for getting the values of the individual parameters is associated with the composite object.

◦ If the normal call of an operation always returns a result satisfying certain conditions, for example, returns a nonempty list or an integer greater than zero, a specialized type for the such a result is defined.

◦ Links between the operation and the original type of its parameter or result are supplemented with similar links with the corresponding refined type.

◦ Each time a specialized type is to be introduced, the existing types are first analyzed to find out if one of them can be reused instead of defining a new type.

• *Defining initialization and finalization of operations.* If some preliminary actions for initializing internal data of the system must be performed before calling an operation and (or) some data must be finalized after the call, the corresponding initialization and finalization code is associated with the operation.

• *Defining values of parameter types.* From the set of possible types of parameters, simple and derived types are removed (pointers, references, and the like). We also remove the types whose values can be obtained only by calling special operations or constructors and such types that their arbitrary values can be used as a parameter value of the corresponding type when a normal call of an arbitrary operation with such a parameter is performed. For each of the remaining types, a value is defined that is used as a parameter value for the corresponding type when calling operations. A code for obtaining this value is stored in the database.

After the refinement is completed, a fairly simple strategy for test generation can be used. This strategy uses the refining information in the database to automatically generate sanity tests for all the interfaces. The main test generation procedure is as follows.

• The initialization code for the corresponding operation is placed at the beginning of the test.

• The values of all the operation's parameters are produced. For each argument type, the value of its source (not refined) type (or base type for pointers, references, and so on) is computed and is then transformed to the argument value.

◦ If a value for the type is defined in the database, it is used.

◦ If another operation or constructor must be invoked to obtain the value or if an initializing code must be called to this end, a call of this operation or the corresponding code is inserted. The values of the parameters for the operations invoked within this code are computed recursively using the same procedure for producing parameter values.

Definitions of the auxiliary operations needed to obtain the parameter values are placed at the beginning of the test.

◦ The values of other types are produced automatically. For simple types (numerical types, characters, strings), simple value generators are used. The values of derived types (pointers, references, and so on) are obtained from the values of their base types. For enumeration types, the first value in the enumeration is used. Structure objects are constructed field by field, and the field values are obtained recursively using the same procedure.

• The operation under test is called with the computed parameter values.

• If necessary, the finalization code for all the parameter values used in the test is included.

• At the end (after the call of the operation under test), the finalization code is included.

• For all the operation calls used in the code, constraints on their results associated with the corresponding specialized types are checked. In addition, for all the pointers appearing in the calls and used later, it is verified that they are not `NULL`.

When generating tests for protected methods of classes, which cannot be called from an arbitrary place, a slightly more complicated procedure is used. Namely, a class is generated inheriting the class in which the method under test is defined. In this class, a public method that is a wrapper over the inherited protected method is defined. Then, the generated test calls the public wrapper method for an instance of the inheriting class.

Some additional work is needed to construct a value of a type that is an abstract class. In this case, if no inherited classes are available, an inherited class is generated automatically. Here, all the abstract (pure virtual) methods are defined in the simplest fashion, and an object of this generated class is used as the value of the required type.

2.4. An Example of Test Generation Using Azov

In this section, we illustrate the test generation procedure according to the Azov technology using the screen saver functions from the Xlib library as an example (Xlib is a part of LSB).

This library includes five screen saver functions.

• `int XSetScreenSaver(Display*, int, int, int, int)` sets the mode of the screen saver operation for the specified display.

• `int XGetScreenSaver(Display*, int*, int*, int*, int*)` returns the current parameters of the screen saver for the specified display. The parameters are returned as pointers corresponding to the parameters of the preceding function.

• `int XForceScreenSaver(Display*, int)` activates or deactivates the screen saver for the specified display depending on the value of the second parameter.

• `int XActivateScreenSaver(Display*, int)` activates the screen saver for the specified display.

• `int XResetScreenSaver(Display*)` deactivates the screen saver for the specified display.

The Xlib documentation gives the following details for the interface.

• The second and the third parameters of `XSetScreenSaver()` are time intervals in seconds determining the screen saver operation. One may define the type `TimeIntervalInSeconds` for these parameters and define for it a correct value 1. The second and the third parameters of `XGetScreenSaver()` are pointers to values of the same type.

• The fourth and the fifth parameters of `XSetScreenSaver()`, as well as the second parameter of `XForceScreenSaver()` are enumerated types defining feasible modes of operation or activation (deactivation) of the screen saver. For them, we can define, respectively, the types `BlankingMode`, `ExposuresMode`, and `ForceMode`. The correct values for them are clearly indicated in the standard; they are, respectively, {`DontPreferBlanking`, `PreferBlanking`, `DefaultBlanking`}, {`DontAllowExposures`, `AllowExposures`, `DefaultExposures`}, and {`Active`, `Reset`}. The fourth and the fifth parameters of `XGetScreenSaver()` are pointers to values of the types `BlankingMode` and `ExposuresMode`.

• All the functions return a code, which may indicate problems occurred while the operation was executed. In this case, `BadValue` is returned. In the normal operation mode, the result type can be refined by naming it `XScreenSaveResult` and defining the distinction from `BadValue` as the base constraint for its values.

An analysis of the possible ways of obtaining a value of the type `Display*` yields the following results.

Use of specialized types

		Number	Original types
Maximal number of a specialized type uses		513	bool (specialized type for parameters taking the value true)
Number of specialized types used	400 or more times	3	bool, int
	200–399 times	5	bool, int, char*, QWidget*
	100–199 times	16	–
	10–99 times	225	–
	2–9 times	556	–
Total number of specialized types		1665	–
Number of uses of specialized types as types of parameters or object calls		11503	–
Number of uses of all types as types of parameters or object calls		22757	–

• There are 18 functions in LSB that return a value of the type `Display*` and two functions that return a reference to a value of this type; this reference can be used to construct a desired pointer.

Among these functions, six are within Xlib; these are `XDisplayOfIM()`, `XDisplayOfOM()`, `XDisplayOfScreen()`, `XOpenDisplay()`, `XcmsDisplayOfCCC()`, and `XkbOpenDisplay()`. The other functions are in other libraries—X Toolkit, GTK, OpenGL and Qt. For simple tests, it is preferable to use functions from one library; for that reason, we further analyze only the six functions indicated above.

• Four functions from the six ones listed above cannot be used because they indirectly require that a value of type `Display*` be already available. For example, `XDisplayOfIM()` has a parameter of the type `XIM`, which can only be obtained using one of the two functions `XOpenIM()` or `XIMOfIC()`. The first one requires `Display*` at its input, and the second function requires `XIC`, which, in turn, can be created only using the function `XCreateIC()`, which again requires a value of the type `XIM`. `XDisplayOfScreen()` requires a parameter of the type `Screen*`, which, in Xlib, can be obtained only via `XDefaultScreenOfDisplay()` and `XScreenOfDisplay()`; however, both of them require a parameter of the type `Display*`.

There is no description of `XkbOpenDisplay()` in Xlib documentation.

Thus, we have only the function `XOpenDisplay()` at our disposal because it requires only the parameter of the type const `char*`, which can be `NULL` in normal calls of this function.

Therefore, tests for the functions considered above are constructed as follows.

• The value of `Display*` is obtained by invoking the function `XOpenDisplay()` with the parameter `NULL`.

• Time intervals are always assigned the value 1.

• The specialized types `BlankingMode`, `ExposuresMode`, and `ForceMode` are assigned the values of `DontPreferBlanking`, `DontAllowExposures`, and `Active`, respectively.

• The results returned by all the functions are checked for equality to `BadValue`. If the equality is detected, a bug is reported.

• In addition, the validity of the results returned by reference from `XGetScreenSaver()` in its fourth and fifth parameters can be checked by comparing them to the feasible values of the specialized enumeration types `BlankingMode` and `ExposuresMode`.

Figure 2 shows the scheme of obtaining parameter values and constraints on the results for the functions in the example considered above (the specialized type `ExposuresMode` is omitted along with pointers to its values and links between both types).

This example also shows that the main source for improving the performance of test development is the wide reuse of specialized types. After refining the parameter type of the function `XOpenDisplay()`, we can obtain values of the type `Display*` required for many functions in Xlib. In this example, there are few functions, and the number of different parameters in them is large; therefore, no improvement in the test generation performance is achieved. However, if the number of functions using the same types of parameters is large, a significant improvement in performance can be obtained.

2.5. Supporting Tools

The work by the proposed technology is supported by the following tools:

• A database containing augmented information about the operations under test.

• A tool for editing information in the database (Web interface); the editor makes it possible to add information to the additional tables in the database, execute queries to find data, and navigate through the tables by

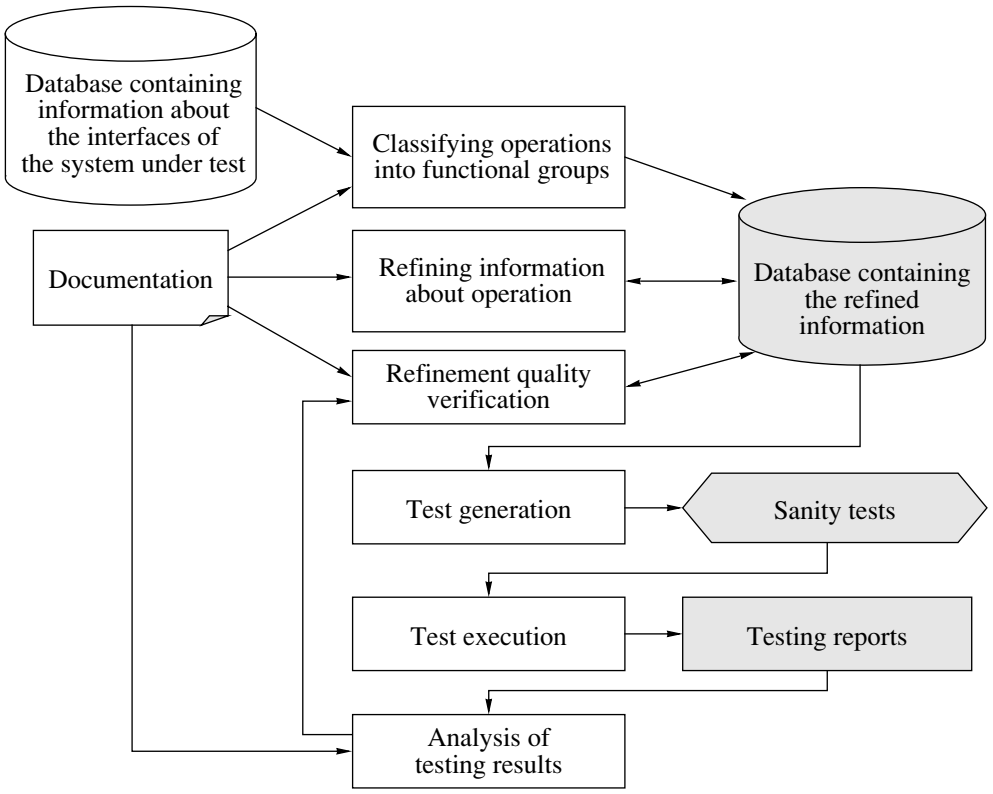

Fig. 1. Scheme of the works performed according to the Azov technology.

references. In particular, it enables one to find all the specialized types refining a given data type thus helping developers to reuse available specialized types.

• Test generator that produces tests on the basis of the information in the database using the procedure described in Section 2.3.

The schema of the main tables of the database used to store the original and additional information about the operations under test is shown in Fig. 3.

In this figure, the tables of the original database are shown by rectangles with thin borders, and the additional tables containing the refining information are shown by rectangles with thick borders. The names of the tables containing the refining information begin with the prefix TG (Test Generation).

We use the following information from the original database: information about the operations under test, the types of their parameters and results. The information about types and operations includes their names, descriptors (`public`, `protected`, or `static`), the sort of type (prime type, enumeration, class, structure, union, template, template instance, pointer, and so on), base type for pointers, references, and other derived types.

3. PRACTICAL USE OF THE PROPOSED TECHNOLOGY

The Azov technology was used to generate sanity tests for the Qt library (version 3) [9] included in LSB [3]; that library was designed for developing portable applications with a graphical user interface.

LSB includes 10873 `public` and `protected` methods (which are available for testing), constructors, and destructors in the Qt 3 library. The information about them (as well as the information about all the operations in this standard) is publicly available (see [10]).

However, some data required for generating correct tests (for example, signatures of pure virtual methods of classes) is not presented in the database. This information was added using the refining procedure.

The operations were classified into groups according to the classes in which they are defined. Since Qt contains many classes (about 400), they were also divided into several groups according to their main functions.

In the refining procedure, 1665 specialized types were defined, and initialization and finalization procedures were added for 36 operations. Some data concerning the use of specialized types are presented in the table. About a half of the specialized types were used two or more times, and some of them were used very

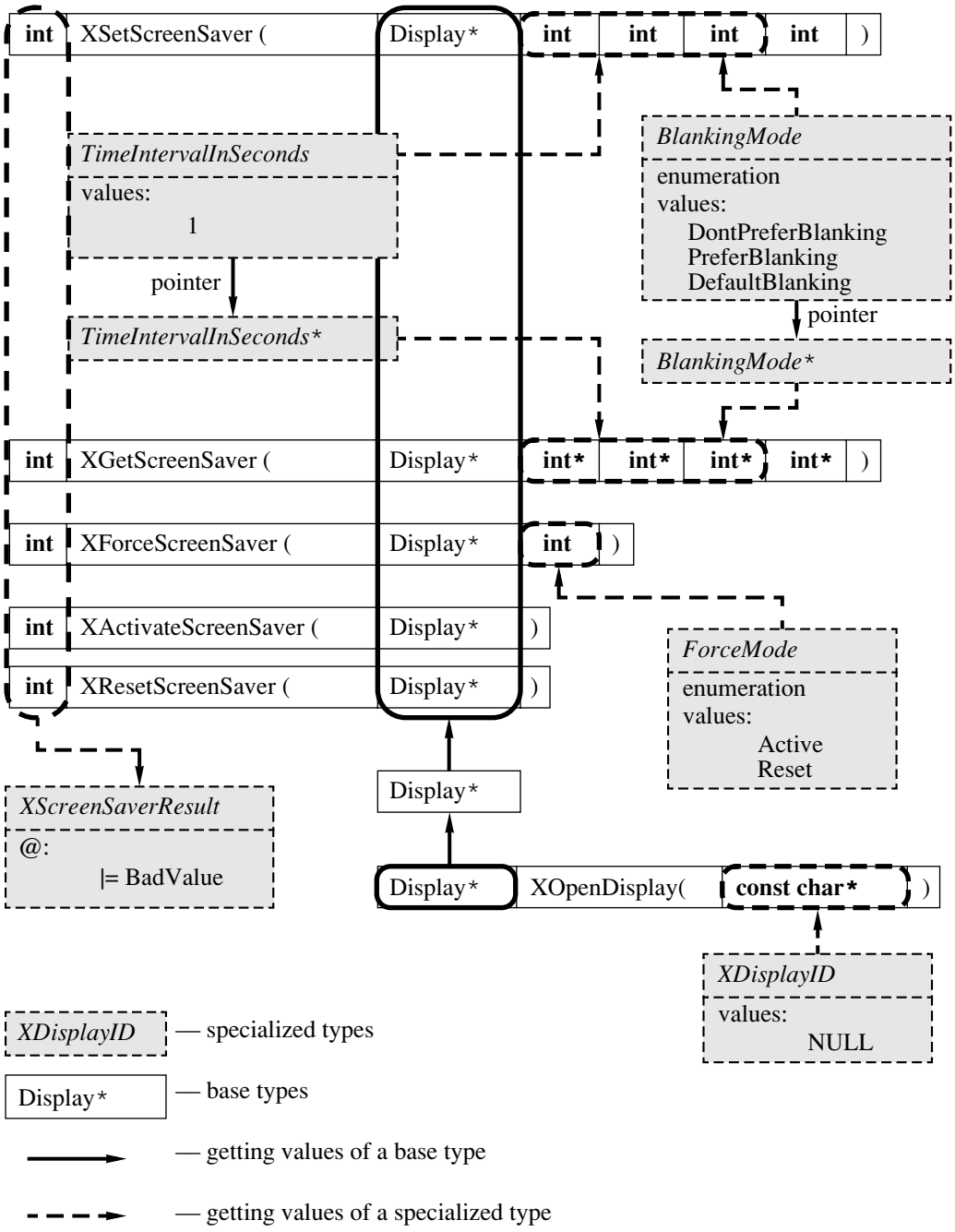

Fig. 2. Scheme for obtaining parameter values and constraints on the results for screen saver functions (the type ExposuresMode is not shown).

often. The table also demonstrates that, in about 50% of cases, the parameters and call objects are generated automatically without the use of explicitly specified values.

The test generation for Qt together with the development of the tools supporting the Azov technology took about four months and involved three developers. At the first stage of the project, considerable effort was devoted to the development and debugging of the tools. At the final stage, when the tools became mature, each developer produced tests for 80–100 functions each day taking into account the time needed to analyze the documentation, refine data, generate, compile, and debug the resulting tests. This is much more than 3–8 functions that can be processed in a day using the conventional manual test construction procedures. The reasons for such an improvement in performance are the reuse of specialized types and tool support.

As a result, test suites for 10803 functions and methods out of 10873 were generated. Only 70 methods

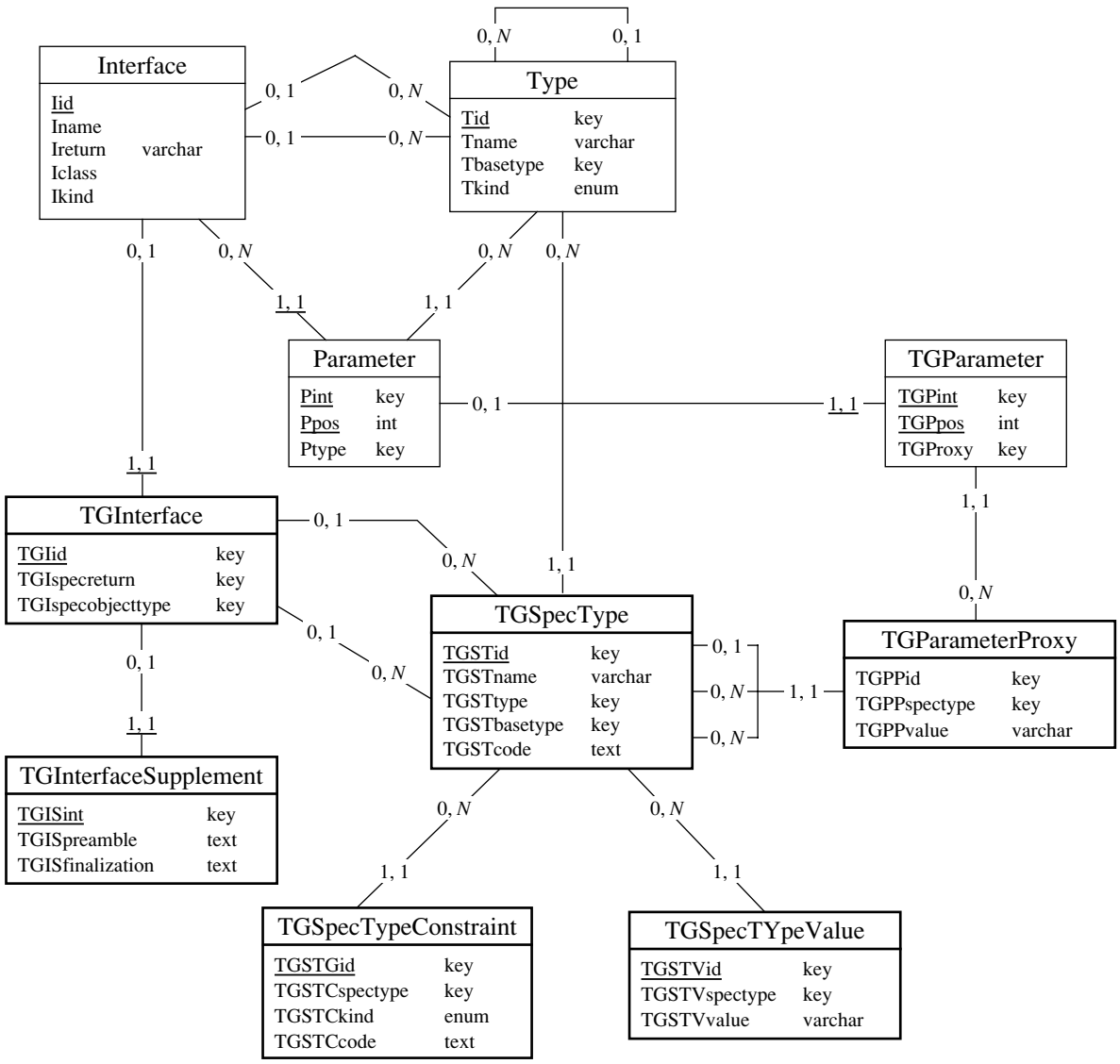

Fig. 3. Main tables of the database storing information about the operations under test.

(0.6%) were not covered for various reasons such as the absence of documentation, accidental membership of internal methods of the library in the standard, impossibility to call a method in C++, and so on.

The execution of the resulting tests for one of the implementations of Qt 3 helped detect about ten various bugs in the implementation although all the tests check only the simplest scenarios of using methods.

The successful application of the Azov technology in this project shows that it provides a proper tool for rapid sanity test generation for large industrial software.

Presently, we are working on generating tests for the Qt 4 library [2] and plan to use it for testing some other libraries in LSB that lack test suites.

4. COMPARISON WITH OTHER APPROACHES TO AUTOMATED TEST GENERATION

The main goal in the development of Azov was the generation of large test suites for large software systems, and the use of automated test generation in the framework of this technology is a necessary requirement. For that reason, in the review below we consider only the methods in which the test data, the sequences of test inputs, or both elements are generated automatically on the basis of certain information. Moreover, the technology proposed in this paper is designed for testing APIs, and it is reasonable to compare it with the approaches designed for the same purpose; indeed, test generation for other types of interfaces (GUI, message passing, event-driven interface, and so on) has specific features.

The methods for automated test generation described in the literature can be divided into the following classes.

• *Test generation methods based on covering arrays.* These methods are based on the combinatorial approach. Each test case is a combination of values of several parameters or factors, and each factor can take a finite number of values. The test suite is generated so as to use all possible combinations of pairs, triples, and so of factor values in a minimal number of test cases. This problem is equivalent to constructing a combinatorial scheme—a covering array with certain parameters. A fairly complete review of the techniques used to construct covering arrays and their use in testing can be found in [11, 12]. The majority of the available tools based on such techniques is presented at site [13]. Among them, AETG [14, 15], TestCover [16, 17], All-Pairs [18], Jenny [19], Intelligent Test Case Handler [20], and PICT [21] are most popular. The first two systems are more mature than the others: they have wide configuration capabilities and enable one to obtain slightly more compact test suites.

Such methods use for test generation only syntactic information about the interface and user-defined finite sets of parameter values. No verification of the results is performed; it is assumed that the results will be checked by people manually or using special tools.

Such methods do not suite well for sanity testing because only one test case is needed for each operation, and covering arrays are useless in the generation of such a test case.

• *Test data generation methods based on modeling data structures and gradual construction of complex objects from values of simple types.* These methods use a description of the data structure in a certain notation for generating test data having the specified structure. The data of simple types (numbers, characters, strings, dates, and so on) are either selected from certain preliminary specified sets or generated using simple (often random-based) algorithms. Next, various combining strategies are used to form more and more complex objects. Such methods can take into account simple constraints on the values of one or several elements, for example, that the values of certain fields must be identical. Partial or total filtering can be used to satisfy the constraints (data are generated and then the results that do not satisfy the constraints are eliminated). However, this approach becomes inefficient when the complexity of constraints grows. In this case, the methods belonging to the next group are used.

There are quite a few products that implement methods of this type. Most of them can produce results in a certain predefined format; they fill a relational database, XML documents, or generate texts in the language specified by a grammar. The vast majority of products that work with databases are commercial (see [22–26]). Also, there are a few research products that can take into account the structure of an application's queries along with the structure of the database. XML generators are mainly research projects or are distributed freely (see [30–34]). The first works in the field of complex data generation [35–37] dealt with the generation of data satisfying a certain grammar. Some more modern tools (see [38, 39]) continue this line of products. There are also some tools and frameworks for developing test data generators that can produce data in different formats. Examples of such tools are OTK [40] and Pinery [41] developed in the Institute for System Programming, Russian Academy of Sciences.

Such methods require that the data structure be described in a certain formal form (BNF, DDL, DTD, XML Schema, Relax NG, and the like) and that constraints on the size and structure of the generated objects be specified. The verification of results obtained by executing tests is usually performed manually or using other tools.

Such tools can be used to generate sanity tests if the interface operations of the system under test have fairly complex objects as their inputs.

In the Azov technology, we also use such a technique for generating objects of complex types; however, the preferred method for generating complex objects is the use of constructors or other operations of the system under test that return objects of the required type. In many cases, this allows one not to care about the correctness of the constructed object when its components must satisfy some specific conditions.

• *Methods for generating test data based on constraint solving techniques.* Under this approach, the constraints imposed on the elements of the test data are resolved without filtering. As a result, a correct array of data for one call of a function under test is produced.

Such test generation methods are used in structural and in functional testing. In both cases, constraints provide conditions of reaching a particular situation; in the structural case, it is described in terms of the structure of the code under test and, in the functional case, it is described in terms of the functionality to be checked.

The techniques used to solve constraints may be very different from each other. For example, they can directly solve systems of constraints, use logical programming [42], use symbolic program execution [43], search through randomly generated data to find data satisfying the constraints [44, 45], and so on.

There are several commercial products among the tools using this approach [46–48]. The most well-known products JTest, C++Test, and .TEST belong to the Parasoft company [46]. They use both the structure of data and the structure of the code of the methods under test to generate test cases. The verification of testing results is performed on the basis of the preconditions and postconditions of methods and invariants of data types that were specified by the user in the source

code in the form of comments. An exception while executing a test is classified as a bug. T-VEC [47] uses operation postconditions both for checks performed during testing and as a source for test data classification and further generation of class representatives. In SureSoft [48], test generation is performed on the basis of the code structure, and the correctness of the results is evaluated by detecting exceptions and gross failures of the program under test. The research projects TestEra [49] and Korat [50] use the user provided invariants and a constraint solving technique for generating objects of complex types in various states. In principle, such approaches can be used for sanity test generation; however, they are very inefficient in this case. Methods for structural test generation cannot be used for requirements-based sanity testing, and the formalization of requirements that is needed for the application of the available tools is too labor-consuming.

• *Dynamic generation methods for structural tests.* The term *dynamic test generation* is used to denote the methods in which objects and values created in the program in the course of testing are used for generating new tests [51]. The generation is aimed at covering certain situations and program states; for this purpose, search methods are used, such as genetic algorithms [52, 53] or hill climbing [54, 55].

The tests produced using such techniques are sequences of method calls rather than individual calls. This makes it possible to check the system operation in many states while using objects in various states as parameters.

The Azov technology makes intensive use of the dynamic test generation by constructing chains of calls that ensure the normal work of the operations under test. In the recent 5–7 years, quite a few research products of this type have appeared. Many of them can be applied to fairly large commercial systems. Since these tools often use a complex combination of various techniques for the analysis of the program under test and for test generation, they can be called *synthetic*.

◦ JCrasher [56], Check-n-Crash [57], and DSDCrasher [58, 59] were developed in the University of Oregon. JCrasher, which appeared first in this group, generates test suites for Java programs using random data of simple types, several heuristics for aiming at probable bugs, operations' syntax, and data structures. The verification is reduced to registering exceptions and gross failures. In Check-n-Crash, the first stage for the syntactic analysis of the code of operations under test was added, which is performed using ESC/Java [60]. At this stage, sets of constraints are obtained corresponding to various paths. These constraints are solved partially using direct methods and partially by slight modifications of random tests. DSDCrashher adds one more preliminary phase at which the program under test is executed on a set of random scenarios, and then the Daikon system [61] is used to reveal possible invariants of the program. These invariants are then used to remove incorrect testing scenarios that result in an error due to the incorrect use of the program rather than due to bugs in the program itself.

◦ AutoTest [62] uses the same techniques as JCrasher for generating tests for programs written in Eiffel. However, in addition to exceptions, user-defined invariants and postconditions are used to verify the results.

◦ Elcat [63] and Randoop [64] developed in MIT use random generation in combination with methods for reducing the set of states to be analyzed and some heuristics for getting into new situations (in Randoop, long sequences of calls of the same methods appearing with a certain probability are used). Also, Elcat can use Daikon to reject incorrect testing scenarios. Randoop was used to generate tests for large libraries (parts of the JDK and .NET with the size of about 700 thousands of code lines); it demonstrated the ability to produce test suites that can detect serious bugs.

◦ In Rostra [65] and Symstra [66], symbolic program execution is used to detect constraints on the input data that can lead to possibly incorrect behavior and to search through possible states. Another example of using symbolic execution is test generation in Java Path Finder [67], which can simulate the work of a Java machine. The parallel symbolic execution and the execution on concrete data corresponding to symbolic constraints are used in CUTE and jCUTE [68]. The random test generation directed by bug detecting heuristics and by reducing the space of states due to symbolic execution is used in DART [69] and Unit Meister [70].

Such approaches can be used for the sanity test generation; however, the resulting test can be used only for the code of one particular software product.

• *Test generation methods based on formal models.* These methods are based on the use of a formal model of the software behavior. Most often, such a model is a finite state machine (FSM) or a labeled transition system (LTS). However, there are also methods based on logical models of the software and on situations occurring in the deductive analysis of those models (theorem proving) [71, 72].

The test generation methods based on finite state machines are classified into two groups:

◦ Methods based on covering the structure FSM models.

◦ Methods using model checking for generating scenarios that bring the system under test into specific states [76–79].

Such methods are inefficient in sanity test generation because they require the construction of a formal model of the system's behavior.

5. CONCLUSIONS

Test generation automation is usually based on the formalization of a large number of rules and criteria that are informally used for manual test generation. Such automation is labor consuming, but it pays off due to the completeness and quality of the resulting test suites. However, one can automate the generation of much less accurate tests that check only the basic operability of a system at a much lower cost.

In this paper, the Azov technology for the broad generation of sanity tests developed in the Institute for System Programming (Russian Academy of Sciences) was described. It is based on refining information about the parameter and result types of the interface operations of the system under test and on using this information for test data generation. The technology is useful for systems that have a fairly rich interface (more than 500 operations), documentation describing the basic functionality of those operations, and a database containing well-structured information about the operations. Note that any compiler is able, in principle, to create such a database.

The proposed Azov technology was used to generate tests for the library Qt, which includes more than 10000 operations. It was demonstrated that this technology can be efficiently used for sanity testing of large systems. Although the tools supporting this technology were developed simultaneously with the test project execution, high test development performance was achieved.

REFERENCES

1. *IEEE 1003.1-2004. Information Technology—Portable Operating System Interface (POSIX)*, New York: IEEE, 2004.
2. http://doc.trolltech.com/4.2/index.html.
3. http://www.linuxbase.org.
4. *ISO/IEC 9899-1999. Programming Languages—C*, Geneva: ISO, 1999.
5. http://www.pathname.com/fhs/.
6. *XLib—C Language X Interface. X Consortium Standard*, http://refspecs.freestandards.org/X11/xlib.pdf.
7. http://www.opengl.org.
8. http://www.gtk.org.
9. http://doc.trolltech.com/3.3/index.html.
10. http://www.linux-foundation.org/navigator/commons/welcome.php.
11. Colbourn, C.J., Combinatorial Aspects of Covering Arrays, *Le Mathemetiche (Catania)*, 2004, vol. 58, pp. 121–167.
12. Hartman, A. and Raskin, L. Problems and Algorithms for Covering Arrays, *Discrete Math.,* 2004, vol. 284, nos. 1–3, pp. 149–156.
13. http://www.pairwise.org/tools.asp.
14. http://aetgweb.argreenhous.com.
15. Cohen, D.M., Dala, S.R., Kajla, A., and Patton, G.C., The Automatic Efficient Test Generator (AETG), *System. Proc. of the 5th Int. Symposium on Software Reliability Engineering (ISSRE)*, Monterey, Calif., 1994.
16. http://www.testcover.com.
17. Sherwood, G., Effective Testing of Factor Combinations, *Proc. of the 3rd Int. Conf. on Software Testing, Analysis, and Review*, Washington, 1994.
18. http://www.satisfice.com/tools.shtml.
19. http://burtleburtle.net/bob/math/jenny.html.
20. http://alphaworks.imb.com/tech/whitch.
21. http://download.microsoft.com/download/f/5/5/f5548df-8494-48fa-8dbd-8c6f76cc014b/pict33.msi.
22. http://www.turbodata.ca/.
23. http://www.sqlmanager.net/products.
24. http://www.sqledit.com/dg/.
25. http://www.dataect.com/.
26. http://www.forsql.com.
27. Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., and Weyuker, E.J., AGENDA: A Test Generator for Relational Database Applications, *Tech. Report of the Polytechnic University, Brooklin, New York* TR-CIS-2002-04, 2004.
28. Binning, C., Kossmann, D., and Lo, E., Testing Database Applications, *Proc. of the ACM SIGMOD Int. Conference*, Chicago, ACM, 2006, pp. 739–741.
29. Bruno, N., Chaudhuri, S., Flexible Database Generators, *Proc. of the 31st Int. Conf. on very Large Databases*, Trondheim, Norway, 2005, pp. 1097–1107.
30. Barbosa, D. and Mendelzon, A., Declarative Generation of Synthetic XML Data, *Software: Practice & Experience*, 2006, vol. 36, no. 10, pp. 1051–1079.
31. Lämmel, R. and Schulte, W., Controllable Combinatorial Coverage in Grammar-Based Testing, *Proc. of TESTCOM'2006*, *Lect. Notes Comput. Sci.*, 2006, vol. 3964, 2006, pp. 19–38.
32. http://www.alphaworks.imb.com/tech/xmplgenerator.
33. http://xml-xig.sourceforge.net.
34. http://iwm.uni-koblenz.de/datagen/.
35. Purdom, P., A Sentence Generator for Testing Parsers, *BIT*, 1972, vol. 12, no. 3, pp. 366–375.
36. Celentano, A., Crespi, Rghezzi, S., Della Vigna, P., Ghezzi, C., Granata, G., and Savoretti, F., Compiler Testing Using a Sentence Generator, *Software: Practice & Experience,* 1980, vol. 10, pp. 897–918.
37. Maurer, P., Generating Test Data with Enhanced Context-Free Grammars, *IEEE Software*, 1990, vol. 7, no. 4, pp. 50–56.
38. Zelenov, S.V. and Zelenova, S.A., Generation of Positive and Negative Tests for Parsers, *Programmirovanie*, 2005, vol. 31, no. 6, pp. 25–40 [*Programming Comput. Software* (Engl. Transl.), 2005, vol. 31, pp. 310–320].
39. http://www..mmsindia.com/JSynTest.html.
40. Zelenov, S.V., Zelenova, S.A., Kossatchev, A.S., and Petrenko, A.K., Test Generation for Compilers and Other Formal Text Processors, *Programmirovanie*, 2003, vol. 29, no. 2,.pp. 104–111 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 2, pp. 104–111].

41. Demakov, A.V., Zelenov, S.V., and Zelenova, S.A., Generation of Complex Structure Test Data with Account of Context Constraints, *Trudy ISP RAN*, 2006, vol. 9, pp. 83–96.
42. Gotlieb, A., Botella, B., and Rueher, M., Automatic Test Data Generation Using Constraint Solving Techniques, *ACM SIGSOFT Software Eng. Notes,* 1998, vol. 23, no. 2, pp. 53–62.
43. DeMillo, R.A. and Offut, A.J., Constraint-Based Automatic Test Data Generation, *IEEE Trans. Software Eng.*, 1991, vol. 17, no. 9, pp. 900–910.
44. Korel, B., Automated Software Test Data Generation, *IEEE Trans. Software Eng.*, 1990, vol. 16, no. 8, pp. 870–879.
45. Gupta, N., Mathur, A.P., and Soffa, M.L., Automated Test Data Generation Using an Iterative Relaxation Method, *ACM SIGSOFT Software Eng. Notes*, 1998, vol. 23, no. 6, pp. 231–244.
46. http://www.parasoft.com/jsp/products.jsp.
47. http://www.t-vec.com/solutions/tvec.php.
48. http://www.suresofttech.com/eng/main/product/api.asp.
49. Marinov, D. and Khurshid, S., TestEra: A Novel Framework for Automated Testing of Java Programs, *Proc. of the 16th IEEE Int. Conf. on Automated Software Engineering*, 2001, pp. 22–31.
50. Boyapati, C., Khurshid, S., and Marinov, D., Korat: Automated Testing Based on Java Predicates, *Proc. of Int. Symposium on Software Testing and Analysis,* Rome, 2002, pp. 123–133.
51. Korel, B., A Dynamic Approach of Automated Test Data Generation, *Proc. of Conf. on Software Maintenance*, San Diego, 1990, pp. 311–317.
52. Pargas, R.P., Harrold, M.J., and Peck, R., Test-Data Generation Using Genetic Algorithms, *Software Testing, Verification & Reliability*, 1999, vol. 9, no. 4, pp. 263–282.
53. Seesing, A. and Gross, H.-G. A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software, *Int. Trans. Syst. Sci. Appl.*, 2006, vol. 1, no. 2, pp. 127–134.
54. Korel, B., Automated Test Data Generation for Programs with Procedures, *Proc. of ISSSTA*, 1996, pp. 209–215.
55. Ferguson, R. and Korel, B., The Chaining Approach for Software Test Data Generation, *ACM Trans. Software Eng. Methodology*, 1996, vol. 5, no. 1, pp. 63–86.
56. Csallner, C. and Smaragdakis, Y., JCrasher and Automatic Robustness Tester for Java, *Software: Practice & Experience*, 2004, vol. 34, no. 11, pp. 1025–1050.
57. Csallner, C. and Smaragdakis, Y., Check 'n' Crash: Combining Static Checking and Testing, *Proc. of the 27th Int. Conf. on Software Engineering* (ICSE), ACM, 2005, pp. 422–431.
58. Csallner, C. and Smaragdakis, Y., DSD-Crasher: A Hybrid Analysis Tool fo Bug Finding, *Proc. of the ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2006, pp. 245–254.
59. Smaragdakis, Y. and Csallner, C., Combining Static and Dynamic Reasoning for Bug Detection, *Proc. of TAP2007*, *Lect. Notes. Comput. Sci.,* 2007, vol. 4454, pp. 1–16.
60. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R., Exrended Static Checking for Java, *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2002, pp. 193–205.
61. Ernst, M.D., Cockrell, J., Griswold, W.G., and Notkin, D., Dynamically Discovering Likely Program Invariants to Support Program Evolution, *IEEE Trans. Software Eng.*, 2001, vol. 27, no. 2, pp. 99–123.
62. Meyer, B., Ciupa, I., Leitner, A., and Liu, L., Automatic Testing of Object-Oriented Software, *Proc. of the 33rd. Conf. on Current Trends in the Theory and Practice of Computer Science (SOFSEM)*, Springer, 2007.
63. Pacheco, C. and Ernst, M.D., Eclat: Automatic Generation and Classification of Test Inputs, *Proc. of ECOOP*, 2005, pp. 504–527.
64. Pacheco, C., Lahiri, S.K., Ernst, M.D., and Ball, T., Feedback-Directed Random Test Generation, *Proc. of ICSE*, 2007, pp. 75–84.
65. Xie, T., Marinov, D., and Notkin, D., Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests, *Proc. of the 19th IEEE Int. Conf. on Automated Software Engineering (ASE 2004)*, Linz, Austria, 2004, pp. 196–205.
66. 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.
67. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Kurshid, S., Lowry, M., Pasareanu, C., Rosu, G., Sen, K., Visser, W., and Washington, R., Combining Test Case Generation and Runtime Verification, *Theor. Comput. Sci.*, 2005, vol. 366, nos. 2–3, pp. 209–234.
68. Sen, K. and Agha, G., CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools, *Proc. of Computer Aided Verification*, 2006, pp. 419–423.
69. Godefroid, P., Klarlund, N., and Sen, K., DART: Directed Automated Random Testing, *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Chicago, 2005, pp. 213–223.
70. Tillmann, N. and Schulte, W., Parameterized Unit Tests with Unit Meister, *ACM SIGSOFT Software Eng. Notes*, 2005, vol. 30, no. 5, pp. 241–244.
71. Yorsh, G., Ball, T., and Sagiv, M., Testing, Abstraction, Theorem Proving: Better Together, *Proc. of the Int. Symposium on Software Testing and Analysis*, Portland, Maine, 2006, ACM, 2006, pp. 145–156.
72. Brucker, A.D. and Wolf, B. Interactive Testing with HOL-TestGen, *Proc. of FATES*, 2006, *Lect. Notes Comput. Sci.*, 2006, vol. 3997, pp. 87–102.
73. Harman, A. Model-Based Test Generation Tools, 2002, http://www.agedis.de/documents/Model-BasedTestGenerationTools.pdf.
74. *Model-Based Testing of Reactive Systems: Advanced Lectures*, Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. (eds.), *Lect. Notes Comput. Sci.,* 2005, vol. 3472.

75. 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].

76. Ammann, P. and Black, P.E., Abstracting Formal Specifications to Generate Software Tests via Model Checking, NIST-IR 6405 (extended version), 1999.

77. Gargantini, A. and Heitmeyer, C., Using Model Checking to Generate Test Form Requirements Specifications, *Proc. of the Joint 7th European Software Engineering Conference and the 7th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (ESEC/FSE99)*, Toulouse, 1999.

78. Devaraj, G., Heimdahl, M.P.E., and Liang, D., Coverage-Directed Test Generation with Model Checkers: Challenges and Opportunities, *Proc. of the 29th Annual Int. Computer Software and Applications Conference (COMPSAC'05)*, 2005, Vol. 1, pp. 455–462.

79. Beyer, D., Henzinger, T.A., Jhala, R., and Majumdar, R., The Software Model Checker Blast: Applications to Software Engineering, *Int. J. Software Tools Technol. Transfer*, 2007, vol. 9, nos. 5–6, pp. 505–525.

SPELL: OK