Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Survey on Test Data Generation Tools

An evaluation of white- and gray-box testing tools for C#, C++, Eiffel, and Java.

Stefan J. Galler and Bernhard K. Aichernig

Graz University of Technology, Inffeldgase 16b/II, 8010 Graz, Austria

received: date / revised version: date

Abstract. Automating the process of software testing is a very popular research topic and of real interest to industry. Test automation can take part on different levels, e.g., test execution, test case generation, test data generation. This survey gives an overview of state-of-the art test data generation tools, either academic or commercial. The survey focuses on white- and gray-box techniques. The list of existing tools was filtered with respect to their public *availability*, their *maturity*, and *activity*. The remaining seven tools, i.e., AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP, and PEX, are briefly introduced and their evaluation results are summarized. For the evaluation we defined 31 benchmark tests, which check the tools capabilities to generate test data that satisfies a given specification: 24 primitive type benchmarks and 7 non-primitive type and more complex with respect to the specification benchmarks. Most of the commercial tools implement a test data strategy that uses constant values found in the method under test or values that are slightly modified by means of mathematical operations. This strategy turns out to be very effective. In general, all tools that combine multiple techniques perform very well. For example PEX uses constraint solving techniques, but in cases were the constraint solver reaches its limitations it uses random based techniques to overcome those. Especially, the two commercial tools AgitarOne and PEX that combine multiple approaches to test data generation are able to pass all 31 tests. This survey reflects the status in 2011.

1 Introduction

Software is in every part of our life. It is software that wakes us up in the morning, makes us coffee, tells us the early morning news. It is software that drives us to our working place, that controls the traffic lights, that moves the elevator. It is software that flies planes and keeps nuclear reactors under control. And these software components are getting more and more complex, have to be maintained and updated. Therefore, testing is crucial. In academia and industry many people are working on new technologies that reduce both bugs in software and costs to find them.

Automating the test process still consists of multiple facets, ordered with respect to increasing complexity: *a*) executing tests, *b*) generating empty test classes and methods, *c*) generating test cases, and *d*) generating test data. Many tools exist that automate test execution, for example the JUnit framework. Most of the state-of-the-art integrated development environments (IDEs), such as Microsoft Visual Studio, and Eclipse, include tools that automatically generate empty test classes and methods. Current research efforts focus on automatically generating test cases and test data. The former automates the process of finding out which method sequence may reveal an error. The latter automates the generation of primitive values and especially non-primitive objects that can be used in test cases.

This survey attempts to give an overview of available commercial and academic tools with respect to their test data generation capabilities. Therefore, we compiled a list of test generation tools, filtered them with respect to their level of *availability*, *maturity*, and *activity*. The remaining seven tools, i.e., AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP, and PEX, are challenged with in total 31 benchmark tests: 24 benchmark tests show the tools capabilities to generate primitive values; seven benchmark tests show how well they perform on non-primitive types and complex specifications. The information collected in this survey reflects the status in 2011.

This survey continues as follows: Section 2 introduces the criteria for tool selection and evaluation. Thereafter, Section 3 presents the result of evaluating the tools. Each tool is shortly introduced and the evaluation result is discussed. The related work is mentioned in Section 4. The survey concludes in Section 5.

2 Evaluation Procedure

AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP and PEX are the seven tools that satisfy all criteria to be part of this survey. Section 2.1 *a*) shows a classification of all candidate tools, and *b*) introduces the selection criteria *availability*, *maturity* and *activity*. Furthermore, Section 2.2 *a*) introduces the evaluation criteria, and *b*) describes the evaluation procedure.

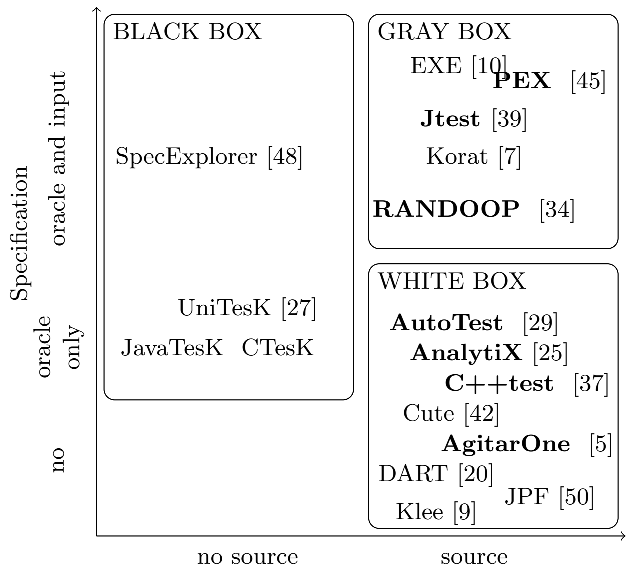

Fig. 1. Classification of Evaluated Tools

2.1 Candidate Tools

Figure 1 presents the map of all relevant tools on automatic test generation. The tools are categorized with respect to two dimensions:

1. source code required/present
2. specification usage

On the one hand we distinguish tools with respect to their access to source code. On the other hand we distinguish between tools that use no specification, use specification as test oracle only, and tools that use specification as test oracle as well as for steering the test input generation.

Figure 1 clusters the tools with respect to the well known terminology [1, p. 21] of black-box, white-box, and gray-box testing. Black-box tests are derived from external description of the software, e.g., specifications. White-box tests are derived from source code internals, e.g., branch conditions. The term gray-box testing is used for test generation approaches that use both, source code internals as well as external descriptions of the software.

This survey focuses on state-of-the art test data generating tools. To ensure the quality of this survey we have to further filter the candidate list. First, only white-box or gray-box testing tools are considered for this survey. Second, the remaining tools are rated with respect to *availability*, *maturity*, and *activity*.

availability Tools have to be publicly available. Either as free download or as commercial tool.

maturity Only tools that are already applied to industrial size applications are considered. We therefore rate all tools from 1 to 4:

1. commercial tool
2. applied to (at least one) industrial size case study
3. applied to (at least one) case study
4. no information about case studies available

activity Tools have to be maintained. In other words, only tools updated within the last three years (i.e., since 2009) are considered.

citation The amount of (scientific) publications that include references to the tool. Figures are extracted from Google scholar in December 2011. The delta value in brackets shows the amount of additional citations since October 2010.

Table 1 lists all white- and gray-box testing tools and summarizes the rating with respect to the introduced classification criteria. AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP, and PEX satisfy the criteria and are therefore part of the evaluation for this survey presented in Section 3. They are highlighted in the Table.

2.2 Evaluation Criteria

We aim for a uniform comparison and evaluation of the state-of-the art test data generation tools. Therefore, in Section 3 each tool is shortly introduced, its test data generation technique is explained in detail, and finally the evaluation result is presented and discussed.

The short introduction of the tool is summarized in a table that includes information on input and output of the tool, supported programming and specification languages, licensing issues and the user pace.

Additionally, we summarize in a table the particular test data generation techniques incorporated by the tool. This summary includes the following attributes:

approach primitive types What approaches are used to generate values for Boolean, byte, character and integer types (e.g., random, constraint solver)?

approach non-primitive types What approaches are used to generate instances for all object types?

Tool	Avail.	Mat.	Act.	Citations
AgitarOne [5]	✓	1	2010	64 (+14)
AnalytiX [25]	✓	1	2010	0 (+0)
AutoTest [29]	✓	1	2010	39 (+13)
Check'n'Crash [15]	✓	4	2005	121 (+22)
C++test [37]	✓	1	2011	0 (+0)
Cute [43]	✓	3	2006	519 (+124)
DART [20]	✗	3	2005	765 (+196)
Eclat [33]	✓	3	2005	144 (+26)
EXE [10]	✗	3	2008	389 (+124)
Klee [9]	✓	3	2009	271 (+94)
Jcrasher [14]	✓	n/a	2007	210 (+31)
JPF [50][1]	✓	3	2010	271 (+73)
Jtest [39]	✓	1	2011	0 (+0)
Korat [7]	✓	3	2007	419 (+37)
PEX [45]	✓	1	2010	199 (+88)
RANDOOP [36]	✓	1	2010	199 (+63)

Table 1. Candidate list. The highlighted tools are white- or gray-box testing tools that satisfy our criteria. These seven tools have been evaluated.

specification usage In what sense does the approach use given specification (e.g., as oracle only, to steer input data generation)?

specification dependencies Is the approach able to deal with specifications where one parameter value depends on another, e.g., $param1 > param2.size()$?

quantifiers Can the approach deal with quantifiers in the specification?

object pooling Does the tool store already instantiated objects for later reuse?

manual objects Is it possible to add manually constructed objects to the tool?

special values Which hard-coded values does the tool use (e.g., min and max value of a data type)?

The actual evaluation of the selected tools is based on analysing automatically generated tests for the given benchmarks. To find out the limitations of each tool we came up with a very structured set of benchmark tests. The benchmark tests are based on the different problem divisions [44] of the annual SMT solver competition (SMT-COMP): *a*) integer difference logic, *b*) real difference logic, *c*) linear integer arithmetic, *d*) linear real arithmetic, *e*) nonlinear integer arithmetic, and *f*) (quantified) arrays.

In addition we added nonlinear real arithmetic and similar expressions for Boolean, character and string types. Furthermore, we added the following tests that explicitly test specification dependencies with respect to parameters:

dependencies between parameters One parameter depends in any attribute from the value of another parameter.

requested null objects Explicitly requesting a *null* parameter.

object type parameters Explicitly requesting an object in a given state.

triangle example [32] The specification for a scalene triangle.

Tables 2 and 3 show the specifications used for each of the benchmark tests. The evaluation of the tools should find out, which tool is able to generate data that satisfies the given specification. Therefore, we implemented for each benchmark test a method that requires input values that satisfy the corresponding specification. We stated the requirement either as precondition or as assertion in the first line of the benchmark method. Tools that are able to execute the return statement of the benchmark method are able to call it with input data that satisfies the benchmark tests specification. An example benchmark method implementation is presented for each tool.

Note, throughout the paper we use logic notation for all given specifications, i.e., the single assignment character represents equality. Furthermore, we had to require non-null objects and non-empty string instances for object types and string types, respectively, to avoid *NullPointerAccess* exceptions while evaluating the Design by Contract™ specification or Java assertion statements.

3 Evaluation

AgitarOne, CodePro AnalytiX, AutoTest, C++test, Jtest, RANDOOP and PEX meet all filter criteria and were therefore evaluated. The presentation order of the evaluation results are determined by the time of publication of the original paper or launch of the tool.

3.1 Jtest

3.1.1 General Information

Jtest is a comprehensive testing product of Parasoft [38] for Java first introduced in 1997. It supports development teams in building new or improving quality of legacy Java applications likewise. Jtest facilitates static analysis, runtime analysis, code review process automation and unit testing. Static analysis includes coding standard checks, data flow analysis and common well-known coding style rules. Runtime analysis mainly provides different kind of coverage information, detects race conditions and security attack vulnerabilities. The code review process is supported through notification, documentation and tracking functionalities. In this evaluation we focus on the unit testing support of Jtest.

Jtest supports automatic generation of JUnit tests and automatic generation of regression tests. It can be used with and without Design by Contract™ specifications. In the former case, the methods postconditions

[1] First JPF test data generation publication

type	specification (arithmetic expression)			
	constant	simple linear arithmetic	simple non-linear arithmetic	inequality
Boolean	$a = true$	$a = b$	-	$a \mathrel{!=} false$
character	$a = 'b'$	-	-	$a > 'b'$
integer	$a = 3$	$a = 3 + 5$	$a = 3 * 5$	$a > 5$
float	$a = 3.2f$	$a = 3.2f + 5.1f$	$a = 3.2f * 5.1f$	$a > 5.1f$
double	$a = 3.2d$	$a = 3.2d + 5.1d$	$a = 3.2d * 5.1d$	$a > 3.2d$
string	$a = ''abcd''$	$a \mathrel{!=} null\ \&\&\ a.matches(''[a-z][0-9]+'')$	-	$a \mathrel{!=} null\ \&\&\ a \mathrel{!=} ''''\ \&\&\ a \mathrel{!=} ''abcd''$

Table 2. Type support tests. Each of the given specifications is used as precondition for a method. The evaluation result tables for each approach, show for what type of specification the approach was able to generate satisfying data.

test	parameters	specification
parameter dependencies	a: int, b: String	$b \mathrel{!=} null\ \&\&\ b.Length = a\ \&\&\ a > 32$
null object	a: Stack	$a = null$
object type	a: Stack	$a \mathrel{!=} null\ \&\&\ a.Count >= 2$
array type	a: int[]	$a \mathrel{!=} null\ \&\&\ a.Length > 2\ \&\&\ a[1] = 1$
forall quantifier	a: List< $String$ >	$a \mathrel{!=} null\ \&\&\ a.Count = 2\ \&\&\ \forall s \in a : s = ''abc''$
exists quantifier	a: List< $String$ >	$a \mathrel{!=} null\ \&\&\ a.Count = 2\ \&\&\ \exists s \in a : s = ''abc''$
scalene triangle example [32]	a,b,c: double	$a + b > c\ \&\&\ b + c > a\ \&\&\ a + c > b\ \&\&\ a \mathrel{!=} b\ \&\&\ a \mathrel{!=} c\ \&\&\ b \mathrel{!=} c$

Table 3. Structural tests. Each of the given specifications is used as precondition for a method. The evaluation result tables for each approach, show for what type of specification the approach was able to generate satisfying data.

are used as oracle. Furthermore, only tests that satisfy the precondition are exported to JUnit test files. In the latter case, Jtest uses thrown exceptions and assertion errors as oracle. For regression tests, the initial test execution run determines the expected return values, which are recorded and used in further execution runs. Jtest incorporates a powerful test data generation engine, which features are discussed in detail in Subsection 3.1.2. Furthermore, Jtest includes a tracing facility, which can be used to capture and replay interaction of Java applications with remote clients and servers.

Table 4 summarizes Jtest's general information. Jtest does not depend on the presence of Design by Contract^{TM} specification, but it improves test quality and reduce test effort if present. It is a commercial tool with a 14-days evaluation license and comes with extensive documentation.

institution	Parasoft
version (tool)	8.4
source code language	Java
specification language	Jcontract by Parasoft, very similiar but less expressive than JML
required input	Source code
optional input	Design by Contract^{TM} specification in Jcontract syntax
output	JUnit test classes
introduced in	1997
last updated in	2010
user pace	Parasoft has more than 10.000 worldwide customers, 58% of Fortune 500 companies
license/price	Commercial license, 14-days evaluation license
documentation	Comprehensive user manual (ca. 750 pages), well structured with examples and step-by-step tutorials
test classification	All tests that satisfy the precondition are exported.

Table 4. Jtest: General Information

3.1.2 Data Generation Approach

Jtest mainly operates on an object repository. This repository is pre-populated with test data for all primitive types: e.g., the minimum and maximum value, 0, -1, +1 are instances of the value pool for the integer type. The pool can be manually populated with values.

Jtest uses those values and tries possible combinations for a given method under test. In addition, Jtest includes some more sophisticated value generation strategies for primitive as well as for non-primitive data types. It is not documented which technologies are used for generating primitive values that satisfy a given precondition, but our evaluation showed that Jtest was able to generate values that were not initially in the pool.

Furthermore, Jtest automatically generates stub objects. A stub is an object that overrides the real objects implementation and returns only hard coded values [21].

approach primitive	Combinations of predefined values are candidates (integer: 0, -1, 1, 10, -10, ..., MAX int, MIN INT), but as can be seen in the test results, more sophisticated technologies are present (not documented which)
approach non-primitive	Stubs are created for all external resources, such as databases, third-party libraries, file system and network I/O. For EJBs Jtest even provides a dummy container.
approach uses specification	Yes, details are not published, but manual inspection of all generated tests let us assume that there exists some mechanism to use the specification for test data generation
parameter dependency	Yes
object pooling	Jtest integrates an object repository which is populated by Jtest itself
manual objects	Yes, the object repository may be populated with manually generated objects
special values	Jtest pre-populates the object pool with special values, such as maximum and minimum value of a data type
quantifiers	Not supported.

Table 5. Jtest: Test Data Generation Details

Jtest stubs are able to return different values each time a method is called.

In case the value combination satisfies the precondition of the method under test the test is executed, the result is recorded and a JUnit test method is exported. The postcondition is evaluated and violations are reported.

Table 5 summarizes Jtest's different strategies for test data generation.

3.1.3 Evaluation

Figure 2 shows an example implementation of a benchmark method. For the evaluation we executed the Jtest command *"Generate Unit Tests"* followed by *"Run Unit Tests (Report All Severities)"*. The result is a set of JUnit tests and a report containing coverage information.

Table 6 summarizes the result of the evaluation on the set of test data benchmark problems. Jtest managed to generate valid input data for all benchmarks. Therefore, we conclude that Jtest incorporates some more sophisticated technologies than mentioned in the official documentation.

Table 7 shows the results of the structural benchmark tests. Jtest is able to generate values even for specifications that include variables where one variable is constrained by the value of another variable. The approach does not work for more complex dependencies as imposed by the scalene triangle specification, or quantifiers.

```
/**
  * @pre (a==3.2f * 5.1f)
  * @post ($result == true)
  */
  public boolean floatNonLinear(float a)
  {
    return true;
  }
```

Fig. 2. Jtest: Example Evaluation Criteria Method. Jtest was only able to generate valid tests after adding the f to all float constants.

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	✓	-	✓

Table 6. Jtest: Results of Data Type Benchmark Tests

test	result
parameter dependencies	✓
null object	✓
object type	✓
array type	✓
forall quantifier	✗
exists quantifier	✗
scalene triangle example	✗

Table 7. Jtest: Results of Structural Tests

3.2 C++test

3.2.1 General Information

Parasoft's C++test [37] is a commercial software quality improvement tool for C/C++, introduced in 1999. It comes as stand alone IDE or Eclipse plugin. C++test supports coding standard checks, static analysis, runtime analysis, and automates code review and unit test generation. C++test incorporates best practice rules such as those proposed by Meyers [30]. C++test can be seen as the little brother of Jtest. Both have very similar feature lists, but due to advanced technologies provided by Java, e.g., Java reflection, Jtest implements more sophisticated data generation techniques than C++test.

Table 8 summarizes C++tests general information. C++test does not support any kind of specification. It

institution	Parasoft
version (tool)	7.3
source code language	C/C++
specification language	No specification
required input	Source and binary
optional input	
output	Unit tests
year	2010
user pace	Parasoft has more than 10.000 worldwide customers, 58% of Fortune 500 companies
license/price	Commercial license with 14-day trial
documentation	Well documented (tutorial, user manual, step-by-step tutorial, examples)
test classification	All tests are exported.

Table 8. C++test: General Information

approach primitive	C++test chooses a value from a pool of random values, pre-defined constants, constants extracted from source, and other values, such as the methods signature and the source files path.
approach non-primitive	A public constructor of the requested class is used for instantiation.
approach uses specification	No.
parameter dependency	C++test generates combinations of randomly chosen values. Not all combinations are exported.
object pooling	Yes, by means of manually written factory methods.
manual objects	Yes, the manually written factory methods for the pooling can instantiate any object instance.
special values	Pre-defined set of values, i.e., string that contains more than 256 characters, min/max values for the given data type
quantifiers	Not applicable, since no specification is supported.

Table 9. C++test: Test Data Generation Details

```
bool floatNonLinear(float a)
{
  assert(a == 3.2f * 5.1f);
  return true;
}
```

Fig. 3. C++test: Example Evaluation Criteria Method

is best used for generating regression tests, which detect changes in the systems under test behavior over time.

3.2.2 Test Data Generation

C++test does not support any form of specification but simple assertion statements. Therefore, all generated combinations of test input data are exported as unit tests. Other tools that support Design by Contract™ specifications can already classify tests called with values that are not supported by the method due to the precondition specification.

Based on the evaluation we can identify two different test data generation strategies: one for primitive and one for non-primitive types. For primitive types C++test selects randomly a value of

- a pre-defined pool of values, such as minimum and maximum value, -1, +1 and 0 for integer types, a string value that has more than 256 characters,
- the path of the file containing the method under test,
- the method under tests signature,
- constant values given within the method under tests body.

A non-primitive value is always constructed by means of a constructor call. In case a method requires multiple parameters, random combinations are generated. Manual inspection of the generated test shows that not all combinations are generated. It is not documented, which combinations are generated.

C++test does not explicitly use any form of an object pool. But it is able to use manually written factory methods. These methods allow to establish a repository of valid object instances that are used in automatically generated tests. Furthermore, C++test supports stubs: user-defined and automatically generated stubs. It only generates stubs if no user-defined version is available. If it is not able to generate a complete stub definition, it will generate a template which has to be customized manually later.

The summary of C++tests data generation techniques is given in Table 9.

3.2.3 Evaluation

Since C++test does not support any Design by Contract™ specification we implemented all benchmark problems by means of assertion statements in the first line as can be seen in Figure 3. Note that, we could not evaluate the string linear benchmark test due to the missing native regular expression support of C++.

For each of those benchmark methods we let C++test generate unit tests. Therefore, we followed Parasoft's recommendation for automatically generating and executing unit tests [37]:

1. generate test cases

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	-	-	✓

Table 10. C++test: Results of Data Type Benchmark Tests

test	result
parameter dependencies	✗
null object	✗
object type	✓
array type	✗
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

Table 11. C++test: Results of Structural Tests

2. generate stubs
3. build test executable
4. executes test cases

C++test generated more than 200 tests. Most of them fired the assertion statement and were therefore classified as *meaningless*. But some parameter combinations successfully passed the assertions. Tables 10 and 11 summarize the evaluation result.

For the character benchmark tests C++test used the integer number of the character (i.e., the character *b* is represented by the integer value 98 in the ASCII format) within the specified assertion and the off by one values (i.e., 97 and 99). Furthermore, it created two tests that passed the maximum and minimum character value, respectively, to the method under test. These two simple techniques for data generation enabled C++test to pass all character benchmarks.

C++test generated nine tests for the integer constant benchmark. Besides the special predefined values already mentioned in the general description (0, -/+1, max/min value) C++test uses the values within the assertion along with the off by one values. Furthermore, C++test uses the result of mathematical expressions present in the source code. For example, C++test generated tests that pass eight ($3 + 5$ taken from the linear integer specification) and 15 ($3 * 5$ taken from the non-linear integer specification) to the method under test.

The predefined values for *float* and *double* type values include in addition to the already mentioned values from the integer domain, the minimum negative and positive value. But the resulting test cases do not include any slightly modified values by means of mathematical operation, such as the off by one values for integer and character types.

For non-primitive types C++test always chooses a constructor. Therefore, C++test is not able to generate test input that is required to be *NULL*. Furthermore, C++test does not try to modify the object any further, i.e., no other methods are called after the constructor to change the object state. This conclusion is based on our empirical evaluation of C++test. The C++test User's Guide [37] does not say anything about the object creation strategy.

C++test failed also on the array and quantifier benchmark tests. For the array it passed *null*, which did not satisfy the specification. C++test has the technologies at hand to pass the quantifier benchmark tests, but unfortunately it did not use the required combination of parameter values.

3.3 AgitarOne

3.3.1 General Information

In 2004 Agitar Technologies released AgitarOne, a commercial tool based on academic research results. AgitarOne can be used as standalone IDE, as Eclipse add-on, or from the command-line. It includes automated JUnit test generation, code rule enforcement technologies and a management dashboard. Furthermore, it suggests assertions it revealed while dynamically analyzing the software under test.

In our evaluation we focus on the automated JUnit test generation feature. Since AgitarOne does not base its analysis on a specification language, it misses an oracle for all generated tests. Therefore, AgitarOnes is useful for automatically generating a regression test suite that captures the current behavior of the software under test.

During test generation AgitarOne collects observations of the code's behavior. Those observations are similar to invariants detected by Daikon [17]. In fact, Daikon and AgitarOne use very similar technologies, which were developed independently at about the same time. The user can then decide whether those observations should be *promoted* to assertions. Thus, AgitarOne helps the software engineer to write specification by means of Java assertions. Those assertions are used in later iterations of the test generation process.

AgitarOne includes a mocking library and is able to run automatically generated and hand-written JUnit tests side-by-side. It is a good example of transforming academic research into a usability friendly and scaling commercial product.

institution	Agitar Technologies
version (tool)	5.1.0.000021
source code language	Java
specification language	None. AgitarOne can handle normal Java assertion statements, and even suggests assertions based on dynamic source code analysis.
required input	Source code.
optional input	
output	A set of JUnit tests, including coverage information. Furthermore, AgitarOne provides a management dashboard to clearly structure and summarize all related information.
introduced in	2004
last update in	2010
user pace	Hundreds of organizations worldwide, from Global 100 to Silicon Valley startups.
license/price	Commercial tool with 30-day trial version
documentation	scientific publication, installation guideline
test classification	Al tests are exported.

Table 12. AgitarOne: General Information

approach primitive	In addition to constraint solver AgitarOne uses constants found in the source code, and values close to them by means of mathematical addition and subtraction operations.
approach non-primitive	Randomly generating objects by calling constructor and other methods of the requested type. Furthermore, AgitarOne includes a mocking library.
approach uses specification	AgitarOne has not specification language support, but is able to use Java assertions to steer the generation process.
parameter dependency	Values with dependencies between each other can be generated as long as they are specified by means of Java assertions.
object pooling	Yes, AgitarOne uses factories for each type, which behave similar to object pool.
manual objects	AgitarOne allows manual refinement of test input data, and provides factories for each data type, which can also be manually adapted to return manually specified objects. Furthermore, AgitarOne provides a pattern - strategy technology, which allows to specify which generation strategy should be used for different automatically detected patterns in the source code.
special values	AgitarOne uses pre-defined values, such as min/max value for each type.
quantifiers	AgitarOne does not have a special specification language, but understands Java assertions very well, even those in loops.

Table 13. AgitarOne: Test Data Generation Details

3.3.2 Data Generation Approach

Agitar Technologies coins the term *agitation* [5] for the process of test data generation. It includes: *a*) static and dynamic analysis of the software under test, *b*) automatic input generation, *c*) exercising the code based on the generated input, and *d*) collecting observations by means of mathematical relationships between variables.

Static and dynamic analysis focuses on collecting path constraints. A constraint solving system then provides required input data to generate tests that steer the execution along a specific path. In all those analysis steps, AgitarOne focuses on performance and scalability. Thus, AgitarOne prefers a fast "good guess" over a correctly calculated value which requires more time. The following paragraphs are based on 'From Daikon to Agitator' [5] and manually inspecting the generated JUnit tests for all benchmarks.

For all numerical types, i.e., *integer*, *float*, *double*, AgitarOne uses all constants found in the source code, the negation of them and the constants +/- a delta value from the original constant. For example, AgitarOne finds the integer constant five in the source code, then it uses the constants four, five and six as test input. For double values, it uses a delta of 0.001.

AgitarOne provides a string solver that can handle constraints imposed by the string API. This solver enables AgitarOne to generate string values that satisfy a regular expression specified through the *matches(...)* method of the *java.lang.String* class. Furthermore, AgitarOne uses *NULL*, the empty string, any string constants from the source code and random combinations of alphanumeric values.

Object types are generated by randomly calling a constructor and zero or more (state-changing) methods of the requested type. Required arguments are generated recursively. All generated objects are kept in a pool to be modified and reused at a later stage in the test generation process. Furthermore, AgitarOne includes a mocking library and enhanced technologies to specify the expected behavior of those mock objects. It records the interaction with the mock object and generates a unit test that expects the same behavior.

```java
public boolean floatNonLinear(float a)
{
  assert a == 3.2f * 5.1f;
  return true;
}
```

Fig. 4. AgitarOne: Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	✓	-	✓

Table 14. AgitarOne: Results of Data Type Benchmark Tests

```java
1  public boolean Exists(List<String> a)
2  {
3    assert a!=null && a.size() == 2;
4    boolean exist = false;
5    for(int i=0; i<a.size(); i++) {
6      if(a.get(i).equals("abc")) {
7        exist = true;
8      }
9    }
10   assert exist == true;
11   return true;
12 }
```

Fig. 5. AgitarOne: *Exists* benchmark

3.3.3 Evaluation

We implemented all benchmark tests by means of Java assertions and checked if AgitarOne is able to generate tests that cover the return statements after the assertion. If so AgitarOne generated a value that satisfies the specification. Figure 4 shows one of the implemented benchmark methods.

AgitarOne generated 112 JUnit tests. Tables 14 and 15 show that AgitarOne passed all benchmark tests. AgitarOnes capability to generate tests for the *forall* and *exists* benchmarks is very impressive. Due to the lack of a supported specification language we had to manually write the quantifiers by means of Java assertions (see Figure 5). The automatically generated JUnit test is presented in Figure 6. AgitarOne creates a new array and adds random values (Figure 6, Lines 5- 6). Finally, it determines that it has to set at least one element in the array to "abc" (Figure 6, Line 7) to satisfy the assertion statement in Line 10 of Figure 5.

```java
1  public void testExists()
2      throws Throwable {
3    List arrayList = new ArrayList(100);
4    boolean add =
5      arrayList.add((Object) null);
6    arrayList.add("testString");
7    arrayList.set(0, "abc");
8    boolean result =
9      new Benchmark().Exists(arrayList);
10   assertTrue("result", result);
11 }
```

Fig. 6. AgitarOne: Generated test for the *exists* benchmark

test	result
parameter dependencies	✓
null object	✓
object type	✓
array type	✓
forall quantifier	✓
exists quantifier	✓
scalene triangle	✓

Table 15. AgitarOne: Results of Structural Tests

The *forall* benchmark test looks very similar to the *exists* benchmark test, but it asserts in each iteration that the current value has to be equal to "abc". However, the generated JUnit test features AgitarOnes *Mockingbird* mock library. Figure 7 shows the generated test. Wherever complex objects have to be constructed, AgitarOne replaces the actual object with a mock object (Line 3). The Lines 5 to 12 define the behavior of the mock object. For each expected method call the return value is defined. Furthermore, the sequence of the method calls is defined and asserted.

In this case, the *ArrayList* has a *size* of two, and will return "abc" for both calls of *get* - once with argument 0, once with argument 1. This generated test satisfies the assertion.

Note, some times AgitarOne used the mock library for the *exists* test too. This let us conclude that some nondeterministic approaches are used.

3.4 AutoTest

3.4.1 General Information

AutoTest [12] started as research tool at ETH Zürich and is by now part of the commercially available Eiffel Studio. Eiffel Studio is the integrated development environment for Eiffel. Eiffel is until now the only programming language with built-in support for Design by Contract^{TM} specifications. AutoTest automatically generates tests for Eiffel programs. It uses different random

```
public void testForAll() throws Throwable {
  Benchmark benchmark = new Benchmark();
  ArrayList a = (ArrayList)Mockingbird.getProxyObject(ArrayList.class);
  Mockingbird.enterRecordingMode();
  Mockingbird.setReturnValue(false,a,"size","()int",new Object[] {},new Integer(2),1);
  Mockingbird.setReturnValue(false,a,"size","()int",new Object[] {},new Integer(2),1);
  Mockingbird.setReturnValue(false,a,"get","(int)java.lang.Object",
                                             new Object[] {new Integer(0)},"abc",1);
  Mockingbird.setReturnValue(false,a,"size","()int",new Object[] {},new Integer(2),1);
  Mockingbird.setReturnValue(false,a,"get","(int)java.lang.Object",
                                             new Object[] {new Integer(1)},"abc",1);
  Mockingbird.setReturnValue(false,a,"size","()int",new Object[] {},new Integer(2),1);
  Mockingbird.enterTestMode(Benchmark.class);
  boolean result = benchmark.forAll(a);
  assertTrue("result", result);
}
```

Fig. 7. AgitarOne: Generated test for *forall* benchmark

based approaches for generating test input data. AutoTest exports only tests that reveal an error. Furthermore, AutoTest implements two different minimization algorithms to reduce the amount of exported tests.

Table 16 summarizes the general information about AutoTest. It is limited to the Eiffel programming language with its built-in support for Design by Contract™ specifications. Eiffel has very prominent clients as listed, but the information which of those use AutoTest as well is not available.

institution	ETH Zürich, Eiffel Incorporation
version (tool)	6.6.8.3355 GPL
source code language	Eiffel
specification language	Eiffel
required input	Source code and Design by Contract™ specification. The specification is required because AutoTest exports only tests that cause a postcondition or invariant violation.
optional input	Test generation can be customized through a configuration file.
output	A set of Eiffel test classes (inheriting from EQA_TEST_SET).
introduced in	2005
last update in	2010
user pace	Group of researcher at ETH and worldwide customers such as AXA Rosenberg Investment Management, Boing, EMC^2.
license/price	Both, commercial and open source license.
documentation	online documentation, scientific publications
test classification	Through the configuration file different minimization algorithms can be activated to reduce the amount of exported tests.

Table 16. Eiffel: General Information. The years mentioned in the table refer to AutoTest, the test generation tool of Eiffel Studio, not Eiffel Studio itself.

3.4.2 Data Generation Approach

AutoTest implements a random based test data generation approach. It uses the Design by Contract™ specification as oracle only. In other words, AutoTest generates test input first and then checks whether it satisfies the precondition of the method under test or not.

AutoTest has two slightly different approaches for generating primitive and object type input data [29].

primitive types For the Eiffel primitive types *INTEGER*, *BOOLEAN*, *CHARACTER*, and *REAL* AutoTest maintains a list of preset values for each type. Candidate values for the *INTEGER* type are, e.g., minimum and maximum value as well as 0, -1, +1, -2, +2, -10, +10. On request it randomly chooses one of those values.

object types AutoTest maintains a pool of already created objects for each type. On request it randomly chooses one of the existing object instances from the pool. A predefined probability defines how often (in case of an empty pool always) a new instance for the requested type is generated and added to the pool. Furthermore, again with a preset frequency AutoTest chooses randomly an instance from the pool and calls modifier features (state changing methods) on it to diversify the pool.

Whenever a new instance has to be created AutoTest executes the following steps (taken from [29]):

1. choose one of the creation procedures (constructors) of the class

approach primitive	Randomly choosing one value from a list of predefined values.
approach non-primitive	Randomly generating instances through calls to the public interface of the type (enhanced random approaches such as ARTOO are supported as well).
approach uses specification	Specification is not used for test data generation, only as test oracle.
parameter dependency	Not applicable, since specification is not used at test data generation time.
object pooling	Yes, objects are stored for later reuse. Extensions exist that improve the pool by means of remembering which precondition predicates the object has already satisfied.
manual objects	Yes, one can add manually generated values to the pool.
special values	The pool is pre-filled with values, e.g., min/max value for each type.
quantifiers	Eiffel does not provide quantifier keywords.

Table 17. AutoTest: Test Data Generation Details

2. generate values for all arguments, recursively
3. call the creation procedure with those arguments

Table 17 summarizes all analyzed aspects of AutoTests test data generation technologies.

Since there is a very close connection between Eiffel Software Inc. and ETH Zürich, research initiatives eventually become part of Eiffel Studio.

Two AutoTest features recently developed at ETH Zürich are 'Adaptive Random Testing for Object-Oriented Software' [13] and 'Satisfying Test Preconditions through Guided Object Selection' [51].

The former enhances the random selection process of values from the pool. Instead of randomly selecting a value, it selects the one value with the highest distance to all already selected values in previous iterations. The distance of two integer values is their mathematical difference. The distance function of objects takes recursively the distance of all members and the distance in the inheritance hierarchy into account. Details are explained by Ciupa et al. [13].

The latter enhances the object pool by replacing it with a map from specification predicates to objects. For each object it is recorded which predicates it satisfies. Therefore, the pool can deliver objects that will likely satisfy the given precondition in case similar preconditions are given for multiple methods within the same system under test.

```
floatNonLinear(a :REAL) :BOOLEAN
  require
    a = 3.2 * 5.1
  do
    Result := true
  ensure
    Result = false
end
```

Fig. 8. AutoTest: Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✗	-	-	✓
integer	✓	✓	✗	✓
float	✗	✗	✗	✓
double	✗	✗	✗	✓
string	✗	-	-	✓

Table 18. AutoTest: Results of Data Type Benchmark Tests

3.4.3 Evaluation

Figure 8 shows the implementation syntax of one of the benchmark methods in Eiffel. *require* and *ensure* are the Eiffel keywords for specifying a methods pre- and postcondition, respectively. Note, AutoTest only exports test cases that violate the postcondition. Therefore, we implemented all methods such that they cause a postcondition exception, i.e., all methods return *true* and the postcondition requires *false*. All test cases that satisfy the precondition will fail on the postcondition and therefore get exported as unit tests.

AutoTest generated 117 tests. Tables 18 and 19 summarize the results.

AutoTest was able to generate valid test input for all inequality tests. Since each specification consists of only one inequality expression, the likelihood to select a value different than the one given in the specification is very high.

Furthermore, AutoTest was able to generate tests for the constant and linear integer benchmark. In both cases it generated exactly the required value which let us assume that AutoTest may include some more sophisticated approaches for integer values than random. For all other tests AutoTest failed, which is reasonable because it is very unlikely to generate the value *16.32* randomly, which for example is required to satisfy the non linear float benchmark.

test	result
parameter dependencies	✗
null object	✓
object type	✗
array type	✗
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

Table 19. AutoTest: Results of Structural Tests

3.5 CodePro AnalytiX

3.5.1 General Information

Google, Inc. bought CodePro AnalytiX from Instantiations, Inc. earlier in 2010. Along with the change in ownership, the previously commercial tool became publicly available under Apache License 2.0.

CodePro AnalytiX is a tool that helps to improve the quality of Java programs. It seamlessly integrates into Rational Developer, IBM WebSphere Studio or any Eclipse development environment [25]. CodePro AnalytiX includes - as all commercial tools - a rich set of metrics and a user-friendly reporting of them. Furthermore, CodePro AnalytiX is able to find similar code snippets in the system under test and can check the source code against security and style conventions. In the following we focus on CodePro AnalytiXs capabilities of automated JUnit test generation.

CodePro AnalytiX provides a rich set of configuration possibilities such as *a*) which parts of the project should be tested? *b*) how many tests should be generated? *c*) if tests that cause an exception should be exported? *d*) where the generated tests should be saved? For each method under test CodePro AnalytiX

– generates input values for all parameters,
– determines combinations,
– computes the result of executing the method under test,
– validates the result, and
– generates JUnit test files.

The process of test input data generation is described in Section 3.5.2. Typically, not all combinations of generated test input data can be tested, due to limited resources. Therefore, CodePro AnalytiX includes some rules to reduce the amount of combinations to a reasonable level. Afterwards, the result of executing the method under test with the determined set of combinations is calculated. CodePro AnalytiX records the result value of a non-void method and all thrown exceptions and determines how it can check these results in the JUnit test. Finally, exporting the result to JUnit test files is straight forward.

CodePro AnalytiX claims to support simple Design by Contract^{TM} specifications for class invariants, and

institution	Google Inc.
version (tool)	7.0.0
source code language	Java
specification language	Java assertions at the beginning of a method are interpreted as precondition. Furthermore, CodePro AnalytiX claims to support Design by Contract^{TM} specification in Java comments with a syntax similar to Jtest. Unfortunately, it did not work for us.
required input	Source code.
optional input	Java assertion statements or a tool specific Design by Contract^{TM} specification.
output	JUnit tests
introduced in	before 2007
last updated in	2010
user pace	Unknown
license/price	Apache License 2.0
documentation	It exists only a general overview in PDF and HTML format. In addition, a user forum is maintained.
test classification	All tests are exported.

Table 20. CodePro AnalytiX: General Information

method pre-/postconditions within JavaDoc comments. Unfortunately, we could not see any difference in terms of generated tests when adding Design by Contract^{TM} specification. Manually writing a test that definitely violated the contract of the tested did not result in any Design by Contract^{TM} specific violation message. Thus, we conclude that Design by Contract^{TM} support is not working in our setting.

3.5.2 Data Generation Approach

Only few details on the test data generation approach are available. CodePro AnalytiX analyzes the method under test to determine the usage of the parameters. Based on that analysis CodePro AnalytiX tries to generate values that help to explore the different behaviors of the method. For example, if an integer parameter is used in a switch statement, then it uses each of the values explicitly listed in non-empty case labels as well as some values that are not in any of the case labels [25].

In case CodePro AnalytiX does not find any values in this first phase it uses pre-defined default values for all well known types. Well known types are all primitive types and non-primitive types such as *java.lang.String*.

For all other cases, CodePro AnalytiX searches in the given order for zero-argument static accessor methods, constructors and multi-argument static accessors. It uses

approach primitive	CodePro AnalytiX analysis the usage of the parameter within the method under test and tries to generate values accordingly.In case this approach fails, pre-defined values are used for all known types (e.g., integer, string, ...).
approach non-primitive	For non-primitive types CodePro AnalytiX calls zero-argument static accessors, constructors and multi-argument static accessors.
approach uses specification	Design by Contract™ specification support did not work for the evaluation, but the approach filtered values that did not satisfy Java assertions.
parameter dependency	worked. Furthermore, CodePro AnalytiX includes heuristics to prune the set of all possible parameter value combinations.
object pooling	No
manual objects	Yes, through *Factories* the user can provide specific instances that should be used as test input data.
special values	Yes, CodePro AnalytiX uses pre-defined values.
quantifiers	No specification support for those quantifiers, but they can be written as Java assertions.

Table 21. CodePro AnalytiX: Test Data Generation Details

the first entry found to instantiate an object of that type. Values required as arguments are generated recursively.

CodePro AnalytiX features EasyMock [19]. EasyMock is a well-known mock library, which provides easy instantiation of mock objects and their configuration of the expected behavior. CodePro AnalytiX can be configured to use EasyMock objects for all interfaces by default. In addition, one can manually specify which classes should be mocked as well.

3.5.3 Evaluation Results

We started our evaluation with Design by Contract™ specifications as claimed in the documentation [25]. Unfortunately, we did not manage to get them working. Therefore, we added again assertion statements in the first line of each benchmark method. Figure 9 shows the *floatNonLinear* benchmark test method, including the Design by Contract™ specification that did not work, and the Java assertion statement in Line 7.

The generated test suite of in total 78 tests was able to satisfy most of the benchmark tests. Tables 22 and 23 summarize the evaluation result.

CodePro AnalytiX satisfies all primitive constant, linear and non-linear benchmark tests due to the fact, that the required input data is present as constants, or

```
1 /**
2  * @pre a==3.2f*5.1f
3  * @post $result==true
4  */
5 public boolean floatNonLinear(float a)
6 {
7   assert a == 3.2f*5.1f;
8   return true;
9 }
```

Fig. 9. CodePro AnalytiX: Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✗
double	✓	✓	✓	✗
string	✓	✓	-	✗

Table 22. CodePro AnalytiX: Results of Data Type Benchmark Tests

mathematical operations on constants in the source code of the method under test. For example, CodePro AnalytiX is able to generate the input value 16.32 for the *floatNonLinear()* benchmark given in Figure 9, since it finds the constant term $3.2 * 5.1$ and the result satisfies the assertion statement.

The inequality benchmarks are not satisfied due to the same reason. To satisfy those specifications, the result has to be modified slightly by means of a mathematical addition operation. But CodePro AnalytiX only uses exactly the constants present in the source.

For a similar reason CodePro AnalytiX does not perform very well on the structural benchmarks, which mostly deal with object type parameters. After calling the constructor of the object type CodePro AnalytiX does not call any further methods on it. Therefore, it does not change the initial state of the object, which in turn does then not satisfy the precondition of the method under test. The same reason prevents CodePro AnalytiX from generating tests for the *forall* and *exists* benchmark.

No constants are present in the scalene triangle benchmark test. Therefore, CodePro AnalytiX uses the set of pre-defined values only. They are not sufficient to find a combination to pass the scalene triangle benchmark test.

3.6 RANDOOP

3.6.1 General Information

Pacheco et al. introduced in 2007 RANDOOP [34,36] *(citation count: 17/136)*. In 2008 he ported the Java version

test	result
parameter dependencies	✓
null object	✓
object type	✗
array type	✓
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

Table 23. CodePro AnalytiX: Results of Structural Tests

to .NET and used it internally at Microsoft to test a very important component of the .NET framework [35].

RANDOOP is a tool implementation of *feedback–directed random testing*, which addresses random generation of unit tests for object-oriented programs. A non-primitive type is created by building a method sequence. Each generated method sequence is immediately executed to ensure that only non-redundant and legal objects are used. Two objects are redundant if their construction sequences are equivalent. In other words, if the generated code for two sequences modulo variable names is equal. An object is legal if it satisfies all *contracts* and *filters*. *Contracts* are methods that use the current state of the system and return either *violates* or *satisfies*. User can write *contracts* by implementing a class that inherits from *randoop.UnaryObjectChecker*. In addition, RANDOOP provides a default set of contracts, such as *NullPointer* occurences and *assertion* violations. Furthermore, for objects RANDOOP checks if *o.equals(o)* holds and methods such as *equals()*, *hashCode()*, and *toString()* do not throw any exception.

institution	MIT CSAIL
version (tool)	
source code language	Java, .NET
specification language	Contracts and filters
required input	Assembly.
optional input	List of user-defined contracts and filters, and a configuration file that specifies limits with respect to time, amount of tests generated, and length of tests generated.
output	Unit test suite of all passing and/or failing test cases.
introduced in	2007
last updated in	2010
number of researcher	
user pace	
license/price	MIT license
clients	Microsoft
documentation	Scientific publications of the technique (i.e., *feedback-directed random testing*), the Java and .NET tools including case study reports.
test classification	Each test case is executed and is classified as error-revealing, passing or illegal. Only error-revealing and passing tests can be exported (user defines, which of them should). Furthermore, equivalent test input data, with respect to the objects *equals(...)* method, is skipped.

Table 24. RANDOOP: General Information

3.6.2 Data Generation Approach

RANDOOP is a test generation tool for object-oriented programs. Therefore, it incorporates only weak data generation techniques for primitive types.

primitive types RANDOOP selects randomly an element from the pool. In the implementation the pool contains a small set of primitives:

– *Boolean*: *true*, *false*
– char: 'a', '4'
– byte, integer: -1, 0, 1, 3, 10, 100
– float: $0.0f$, $1.0f$, $10.0f$, $100.0f$
– double: $0.0d$, $1.0d$, $10.0d$, $100.0d$

object types For object types RANDOOP uses either *NULL*, or uses a sequence from the pool. New sequences are generated by combining two sequences from the pool with m calls to a randomly selected method. Candidate methods are public methods of the corresponding class. RANDOOP adds m calls to the existing sequence, since especially container classes often require more than one element in the container. Therefore, it makes sense to call for example *add(...)* multiple times in a row. A newly generated sequence is executed to determine that it is not redundant and constructs an object not violating any contracts.

3.6.3 Evaluation Results

We evaluated the Eclipse plugin of RANDOOP for Java. The .NET implementation is equivalent to the Java implementation. RANDOOP per default uses Java assertion statements to filter sequences that generate illegal object states. Therefore, we implemented our benchmark methods by means of Java assertions, as can be seen in Figure 10.

In addition to the class containing the benchmark methods, we told RANDOOP to use *java.util.ArrayList*, *java.util.LinkedList* and *java.util.Stack* and set the null object generation probability to 0.3. Otherwise, RANDOOP does not know them but they are required by some of the benchmark tests. The results did not improve when we increased the default timeout from 100 seconds to 300, or even 1000 seconds.

approach primitive	Selects a value from a fixed pool of values.
approach non-primitive	Either use null, or an existing sequence from the pool.
approach uses specification	It uses *contracts* and *filters* to check the constructed sequence before it gets executed on the method under test or exported to a unit test.
parameter dependency	Not applicable, since specification is given in terms of *contracts*. And RANDOOPs *contracts* are methods, that take the current state of the system and return *satisfied* or *violated*.
object pooling	Yes.
manual objects	Yes.
special values	Yes, the pool of primitive values is, e.g., populated with −1, 0, 1, 'a', *true*, and others.
quantifiers	Not applicable, since no formal specification used. (see parameter dependency)

Table 25. RANDOOP: Test Data Generation Details

```
public boolean floatNonLinear(float a)
{
  assert a == 3.2f * 5.1f;
  return true;
}
```

Fig. 10. Example Evaluation Criteria Method

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✗	-	-	✗
integer	✓	✗	✗	✓
float	✗	✗	✗	✓
double	✗	✗	✗	✓
string	✗	✗	-	✓

Table 26. RANDOOP: Results of Data Type Benchmark Tests

RANDOOP targets mainly the challenge of testing object-oriented programs. It is therefore obvious that it does not very well perform on any primitive type benchmarks. Table 26 summarizes the expected weak performance of RANDOOP on the primitive benchmark tests. Comparing the inequality benchmark specifications with the primitive values in the pool (see Section 3.6.2), shows that the pool contains at least one element for each type that satisfies the specification, but for the character type. For the character type the pool contains only an 'a', and the benchmark expression requires a character greater than 'b'.

test	result
parameter dependencies	✗
null object	✓
object type	✓
array type	✓
forall quantifier	✗
exists quantifier	✗
scalene triangle	✗

Table 27. RANDOOP: Results of Structural Tests

More interesting are the structural benchmark tests that include more object types. Table 27 summarizes RANDOOPs performance on this set of benchmark tests. Unfortunately, RANDOOP did not perform that well either. We expected that RANDOOP cannot satisfy the specifications for the *parameter dependencies* and the *scalene triangle* benchmarks. Those two test require sophisticated primitive value generation capabilities.

Manually inspecting why RANDOOP did not pass the two quantifier benchmarks, revealed that it was able to generate *java.util.List* objects with enough elements, but never with the expected values. This can be reduced to RANDOOPs weak primitive value generation capabilities.

3.7 PEX

3.7.1 General Information

Microsoft Research started a few years ago the development of PEX [45], a white-box test generation tool for .NET. Meanwhile it is not only a research tool but part of the Visual Studio 2010 Power Tools that help unit testing .NET applications. PEX started as a tool that creates a test suite, which achieves high branch coverage based on dynamic symbolic execution. Today, it perfectly incorporates other tools and research results. PEX uses the Z3 [31] SMT solver for solving the path constraints collected during dynamic symbolic execution; It uses REX [47] for generating string values specified by means of regular expressions; PEX supports Code Contracts [3], which is a Design by Contract^{TM} specification language for .NET; and features Moles [16], which is a light-weight mocking library from Microsoft Research. Furthermore, PEX fully integrates with Microsoft Visual Studio.

Table 28 summarizes the general information about PEX. It works on the intermediate language of .NET so it can be used for testing programs in any .NET language. Note, currently only unit tests for C# can be exported.

The core of PEX is a test data generator. It supports not only test data generation for Design by Contract^{TM} specification but features specifications as parameterized unit tests [46] as well. Parameterized unit tests are

institution	Microsoft Research
version (tool)	0.91 on Visual Studio 2010 Ultimate
source code language	Theoratically any .NET language, but test export is currently only available for C#.
specification language	PEX supports specification in terms of parameterized unit tests or Code Contracts.
required input	Source code.
optional input	A specification.
output	Unit tests/parameterized unit tests in one of the supported unit test framework formats (Visual Studio Unit Test, NUnit, Mb Unit, xUnit.net).
introduced in	2008
last updated in	2010
user pace	At least 10 research cooperations with world-wide research institutions, open source community.
license/price	Academic and commercial license (Microsoft Visual Studio 2010 Power Tools Software Terms).
documentation	Well documented at different technical levels including step-by-step tutorials and scientific publications.
test classification	Configurable what tests should be exported.

Table 28. PEX: General Information

approach primitive	Z3 SMT solver [31] and REX [47] for string values
approach non-primitive	Z3 and REX: objects are encoded as maps of their members as in ESC/Java [18]
approach uses specification	PEX does not only use Code Contracts specification but also collects all path constraints so that it is able to generate a test input data set that achieves high branch coverage.
parameter dependency	Yes, encoded in SMT constraint.
object pooling	No
manual objects	No
special values	No
quantifiers	Support of forall and exists.

Table 29. PEX: Test Data Generation Details

```
public bool floatNonLinear(float a)
{
    Contract.Requires(a == 3.2f * 5.1f);
    return true;
}
```

Fig. 11. Example Evaluation Criteria Method

unit test methods that have parameters. In other words, a parameterized unit test specifies the behavior of the method under test for all possible input values. One specific parameter combination is equivalent to a traditional unit test.

3.7.2 Data Generation Approach

PEX starts with simple random input for a given method under test. While executing the method PEX collects runtime information, e.g., symbolic values for all variables and path constraints. At each condition statement PEX collects information about the branching criteria. PEX re-executes the method with input values that satisfy all path conditions. This process is called dynamic symbolic execution [9,20]. It is also known as concrete symbolic (concolic) execution [43].

Therefore, it is able to explore all feasible paths of the method under test. The values are calculated by passing the path constraint to the Z3 SMT solver. Z3 is able to solve constraints on propositional logic, fixed-sized bit-vectors, tuples, arrays and quantifiers. Arithmetic constraints over floating point numbers are approximated by a translation to rational numbers. Heuristic search techniques are used outside of Z3 to find approximated solutions for floating point constraints [41]. Recently, PEX integrated REX, a technology for generating string values that are formalized by means of regular expressions [47].

Implementation details regarding the instrumentation process for symbolic execution, and the symbolic representation of values, pointers and objects can be found in a lot of technical reports and publications of Microsoft Research [45,41].

3.7.3 Evaluation Results

To evaluate PEX we implemented one method for each evaluation criterion. Its method body consists of a single *return true* statement. Figure 11 shows an example implementation.

We evaluated the data generation facility of PEX by letting it explore all paths. The result is a set of test data combinations such that all feasible paths are executed. The Code Contracts preconditions are recognized and PEX interprets them as different branching statement. In other words, it tries to generate test data such that each clause of the specification is once fulfilled and once not. The result is a set of test input data. Tests not satisfying the precondition are marked *meaningless*. Tables 30 and 31 show that PEX is able to pass all benchmark tests. This does not necessarily mean that PEX is able to test everything on the spot, but it is definitely the most advanced tool at the moment.

type	constraint			
	constant	linear	non-linear	inequality
Boolean	✓	✓	-	✓
character	✓	-	-	✓
integer	✓	✓	✓	✓
float	✓	✓	✓	✓
double	✓	✓	✓	✓
string	✓	✓	-	✓

Table 30. PEX: Results of Data Type Benchmark Tests

test	result
parameter dependencies	✓
null object	✓
object type	✓
array type	✓
forall quantifier	✓
exists quantifier	✓
scalene triangle	✓

Table 31. PEX: Results of Structural Tests

For all tests that included parameters of type *float* or *double* PEX issues a 'testability issue in floating point equality' warning. This warning tells the user that for floating point operations PEX only uses heuristics. Still PEX was able to generate correct input data for all those benchmarks.

4 Related Work

Throughout the paper we focused on tools for object-oriented languages. This holds for the related work as well. Furthermore, we do not include any UML based tools. Therefore, we do not consider tools such as QuviQ testing tools [2], CONFORMIQ [24], LEIRIOS [26], and the BZ testing tool [28].

This section is categorized in two parts:

- test data generation tools that were not considered to be part of the evaluation for this survey due to not fulfilling the required criteria listed in Section 2.1, and
- Black-Box testing tools.

The tools mentioned in the upcoming sections are ordered chronologically. We use the citation count to decide which tools are mentioned and which not. Section 4.1 includes only those tools that are cited at least 150 times. Section 4.2 requires at least 30 citations. The citation count was determined through Google Scholar on October 8th 2010.

4.1 Test Data Generation Tools

Korat [7] *(citation count: 384)* is a test case generation tool based on Design by Contract^{TM} specification. It uses a methods post condition as oracle, and uses the precondition to generate complex test input data. Korat uses a *repOK()* and a *finatization()* method for constructing all non-isomorphic test input data up to a given bound. The *finatization* method implements the search for new input. Korat observes access to precondition predicates and class fields to prune the search space. Furthermore, the bound is specified in the *finatization* method. Korat provides a preliminary implementation and the user is able to enhance it if necessary. The *repOK* method implements the precondition check. It returns true if the generated input satisfies the specification, and false otherwise.

Visser et al. introduced JPF [49] in 2003 as a tool for model checking Java programs. It is a very mature tool, which was already applied to real world case studies. Among them the real-time operating system DEOS from Honeywell [40] and prototype Mars Rover [8]. Based on the JPF framework Visser et al. introduced a test input data generation extension [50] *(citation count: 198)* years later. Similar to PEX, the test input data generation extension of JPF uses symbolic execution of a *repOK* method, the methods precondition, to generate all (non-isomorphic) input data. A manual bound for the input data size is given. Multiple extensions to JPF exist that even add Design by Contract^{TM} support, but unfortunately most of them are research prototypes or even not more than research ideas. JPFs test data generation capability is not included in this survey due to missing industrial size case studies.

In 2005 Sen was co-author of two very successful tools with respect to their publication count: DART and Cute. DART [20] *(citation count: 569)* is a test data generation tool that tries to cover all paths within the system under test, by combining concrete and symbolic execution. Cute [43] *(citation count: 395)* combines concrete and symbolic (concolic) execution, as well, but extends it to pointer structures. Note, concolic execution is equivalent to dynamic symbolic execution of PEX. DART combines three main techniques: *a*) automated interface extraction, *b*) automatic generation of test driver, and *c*) automatic generation of new test input data based on dynamic analysis of program behavior with respect to its input data. Program crashes or assertion violations build the test oracle. DART gathers path constraints while executing the program under test with initially random input values. New input values for the same test force the execution to take a new path (for all reachable paths). Due to concolic execution DART can replace a path constraint, which the corresponding constraint/SAT solver cannot solve, with a concrete value, e.g., *true* or *false*. This allows DART to be used for more complex case studies. Cute is a close work to DART, but improves some steps in the process of test data generation. Sen points out in the related work section [43, p. 271] that unlike DART, Cute can handle pointers and data structures as input parameters and it implements a new constraint solver that significantly speeds up the analysis.

EXE [10] *(citation count: 258)* is the "youngest" test data generation tool that already achieved enough citations to be mentioned in this survey. EXE uses symbolic execution to generate input data that forces the program under test to crash. The programmer can mark variables, i.e., memory locations, to be traced symbolically. The program is then instrumented to execute all feasible paths. In case a path terminates, e.g., program crashes, a call to *exit()*, or an assertion fails, a concrete value is generated which can reproduce the error/crash when executing the original program without instrumentation. EXE performed well on the BSD and Linux packet filter implementations, udhcpd DHCP server, the pcre regular expression library, and three Linux file systems [10].

4.2 Black-Box Testing Tools

SpecExplorer [11] *(citation count: 71)* [48] *(citation count: 69)* is a model-based testing tool from Microsoft for .NET programs. Initially, models had to be written in Abstract State Machine Language (AsmL) format. Later, the Spec# [4] language was developed to get the syntax of the specification language closer to the syntax of the programming language used for the implementation [22]. Spec# is a superset of the prominent C# programming language and basically adds Design by Contract™ keywords to it. Programs written in Spec# are thus model programs, that include a formal specification and can be executed. The latest release of SpecExplorer further reduces the gap, and the model can be written in C# syntax and fully integrates into the Visual Studio 2010 integrated development environment for .NET. Therefore, a SpecExplorer solution in Visual Studio 2010 consists of three projects: 1. the model in C#, 2. the implementation in any .NET language, e.g., C#, and 3. the test suite, in the Visual Studio Unit Test format. Being able to write the model in the same language as the implementation improves applicability of the approach and tool since developer can reuse their knowledge about the programming language. They can focus on what is the best model abstraction of the implementation, and not on how to write it [22]. The SpecExplorer Visual Studio tool is very mature and able to automatically generate test input data for complex models by means of a combination of generation techniques. For example it integrates combinatorial testing techniques, and SMT constraint solving [23]. Still the SpecExplorer approach is to separate the model from the implementation. SpecExplorer is designed to handle non-deterministic and multi-threaded software. Furthermore, SpecExplorer provides a facility to specify accepting states through a condition. This is important since multi-threaded and non-deterministic programs do not always terminate. A model program may correspond to an infinite large automaton. A *test purpose* can be used to slice a model to the parts a test is interested in. SpecExplorer supports both offline and online testing. Offline tests are generated from that model either to provide some kind of coverage or based on a random walk in the state space. Online tests are created on the fly as testing proceeds [48]. At each step one controllable action is selected, based on predefined or dynamically updated weights, to be executed.

UniTesK [6] *(citation count: 40)* is a general architecture for test generation with two specialized versions: JavaTesK for Java and CTesK for C and C++ programs. UniTesK test sequence generation tool family works on the specification only. It requires *Mediator* instances to link the specification with a given implementation and keep those two independent systems synchronized during test execution. UniTesK traverses all states of the specification which are limited by an arbitrary coverage criterion. This coverage criterion can be described by a set of predicates, which values are calculated based on the system's state, the current operation and all its parameters. As with all other approaches and tools UniTesK uses the given specification as test oracle. UniTesK and its related tools for Java and C are developed to be used in testing industrial software.

5 Conclusion

For this survey we evaluated only tools that are able to automatically generate tests including test input for a white- and gray-box scenario. Only tools that *a*) are publicly available, *b*) have already been applied on industrial size case studies, *c*) and received an update within the last two calendar years, are considered.

Table 32 summarizes the necessary input to the tool and the provided output. Most of the tools work on the actual source code, only C++test and RANDOOP require the compiled executable, either in addition or instead of the source code. All tools produce unit tests in the format that is most common for the corresponding programming language. In other words, JUnit tests for Java, Eiffel tests for Eiffel, CppTest for C++. Only PEX, which is the most advanced tool of all since it is not only a research tool but also part of the Visual studio distribution, interfaces to multiple unit test frameworks that are available for .NET applications.

In addition to the source code, some tools are able to understand contracts. Depending on the tool these contracts can be either assertions, as provided by the standard language definition of the corresponding programming language, or Design by Contract™ specifications integrated into or supported through add-ons to the programming language.

Eiffel is the leading programming language with respect to integration of Design by Contract™ specifications. From the very beginning of the language definition, Bertrand Mayer focused on creating a programming language that fully integrates mathematical correctness techniques and tools into the language. Based on the success of Eiffel, research groups began to develop Design by Contract^{TM} add-ons for other languages as well. The JMLSpec initiative became the more or less standard for Java applications. Unfortunately, Jtest took another approach and developed their own set of Design by Contract^{TM} specifications. The syntax is very similar to JMLSpec but not unfortunately not equal. It is again PEX, that provides the most flexible and advanced support for additional specifications. It understands assertions and seamlessly integrates with the Code Contracts (Microsoft) project. In addition, PEX is the only tool in that evaluation that collects path constraints based on standard programming language features, such as if-statements, for- or while-loops, and method calls. Therefore, PEX is able to produce not only one single input that satisfies a given specification as evaluated in this survey, but a set of input values that tries to achieve path coverage for the provided software.

Each of the tools were evaluated on a standard set of benchmarks that are presented in Section 2.2. This set of benchmarks was designed to find out the capability borders of each tool. The evaluation results are summarized and discussed in detail in each tool section. Furthermore, Tables 33 and 34 provide an overview over all tools, such that a tool comparison is achievable.

Jtest is a very mature tool that can deal with Java programs with and without Design by Contract^{TM} specifications. The supported Design by Contract^{TM} syntax is simple but misses important specification features, such as quantifiers. It satisfied most of the benchmark tests due to very useful practical technique: Jtest uses constants present in the method under test, and slightly modified values of those constants as test input. For example, Jtest adds and subtracts one from each integer value found. Such manipulation rules are available for all primitive data types.

C++test is the small brother of Jtest. It is Parasoft's test generation tool for C/C++ programs. C++test includes most of Jtest's features, but due to C/C++ limitations with respect to Java (e.g., Java reflection), it performs not as well as Jtest for object types. For primitive types it uses - similar to Jtest, CodePro AnalytiX, and AgitarOne- constants found in the source code and manipulates them. Therefore, the four mentioned tools together with PEX (that uses a completely different approach) are able to generate tests for all evaluated primitive data type benchmarks. For object type tests C++test uses public constructors of the requested type. Therefore, C++test was able to generate the "trivial" test that checked if the tool is able to generate an object in general. But it failed on all other tests that required some form of advanced generation technique.

AgitarOne is very similar to Jtest. It successfully made the transformation from a research project to a commercial application. It is one of two tools (the other is PEX) that passed all benchmark tests. Again, the key to success is the strategy to use slightly modified values found in the method under test by means of mathematical operations. In addition, AgitarOne generates mock object stubs for all object types that can be used to manually add objects of interest. It is the only tool that provides that kind of mechanism. Of course, others such as PEX use mock objects as well but they are able to fully generate them automatically. AgitarOne differentiates itself from Jtest for primitive data generation by incorporating constraint solving techniques.

AutoTest and RANDOOP are the only two pure random tools that qualified to be part of this survey. AutoTest performed a little bit better on the primitive type benchmark tests than RANDOOP. This can be explained by the focus of the tools. AutoTest randomly chooses values for primitive types, whereas RANDOOP selects a value from a predefined pool. On the structural benchmark tests RANDOOP performed better. Since both tools work on different programming languages they do not compete against each other. Therefore, Eiffel is still the only programming language that fully incorporates Design by Contract^{TM} specifications and has a precise mathematical semantics. RANDOOP on the other hand competes with Jtest, CodePro AnalytiX, and AgitarOne for Java programs, and with PEX for .NET programs. In both categories it is outperformed by all other mentioned and evaluated tools.

CodePro AnalytiX is another commercial tool for testing Java applications. It incorporates similar technologies as Jtest, but misses the functionality of modifying constants found in the method under test. Furthermore, it does not include any constraint solving techniques. These are the reasons why CodePro AnalytiX do not pass that much primitive type benchmark tests as Jtest does. CodePro AnalytiX claims to support Design by Contract^{TM} specification, which we could not relate to. It looks like Design by Contract^{TM} support is planned for the future but not part of the current release that was given to us for evaluation purposes.

PEX features the most sophisticated and recent data generation techniques. Research on string generation, mock object instantiation, parameterized unit tests, and constraint solving are perfectly incorporated in PEX. This allowed PEX to pass all benchmark tests. One has to note that PEX is the only evaluated tool that does not only try to generate unit tests, but a set of unit tests that achieves code coverage. It therefore collects path constraints with each generated test and tries to generate values that take another path in the next generation iteration.

Nevertheless, Tillmann et al. [45] list limitations of PEX that are mentioned here to point out common limitations to other tools as well:

	Required Input	Optional Input	Output	Section
AgitarOne	source		JUnit tests	3.3
AnalytiX	source	assertions	JUnit tests	3.5
AutoTest	source	Eiffel specifications	Eiffel tests	3.4
C++test	source + binary		Unit tests	3.2
Jtest	source	Jcontract specification	JUnit tests	3.1
RANDOOP	assembly	RANDOOP contracts/filters	JUnit tests	3.6
PEX	source	assertions, Code Contracts	Visual Studio Unit Tests, NUnit tests, Mb Unit tests, XUnit.net tests	3.7

Table 32. Tool Input and Output. Summary of required and optional input as well as the output of each evaluated tool

		AgitarOne Sec. 3.3	AnalytiX Sec. 3.5	AutoTest Sec. 3.4	C++test Sec. 3.2	Jtest Sec. 3.1	RANDOOP Sec. 3.6	PEX Sec. 3.7
Boolean	constant	✓	✓	✓	✓	✓	✓	✓
	linear	✓	✓	✓	✓	✓	✓	✓
	non-linear	-	-	-	-	-	-	-
	inequality	✓	✓	✓	✓	✓	✓	✓
character	constant	✓	✓	✗	✓	✓	✗	✓
	linear	-	-	-	-	-	-	-
	non-linear	-	-	-	-	-	-	-
	inequality	✓	✓	✓	✓	✓	✗	✓
integer	constant	✓	✓	✓	✓	✓	✓	✓
	linear	✓	✓	✓	✓	✓	✗	✓
	non-linear	✓	✓	✗	✓	✓	✗	✓
	inequality	✓	✓	✓	✓	✓	✓	✓
float	constant	✓	✓	✗	✓	✓	✗	✓
	linear	✓	✓	✗	✓	✓	✗	✓
	non-linear	✓	✓	✗	✓	✓	✗	✓
	inequality	✓	✓	✓	✓	✓	✓	✓
double	constant	✓	✓	✗	✓	✓	✗	✓
	linear	✓	✓	✗	✓	✓	✗	✓
	non-linear	✓	✓	✗	✓	✓	✗	✓
	inequality	✓	✓	✓	✓	✓	✓	✓
string	constant	✓	✓	✗	✓	✓	✗	✓
	linear	✓	✓	-	-	✓	✗	✓
	non-linear	-	-	-	-	-	-	-
	inequality	✓	✓	✓	✓	✓	✓	✓

Table 33. Primitive Data Type Benchmark Results Overview

	AgitarOne Sec. 3.3	AnalytiX Sec. 3.5	AutoTest Sec. 3.4	C++test Sec. 3.2	Jtest Sec. 3.1	RANDOOP Sec. 3.6	PEX Sec. 3.7
parameter dependencies	✓	✓	✗	✗	✓	✗	✓
null object	✓	✓	✓	✗	✓	✓	✓
object type	✓	✗	✗	✓	✓	✓	✓
array type	✓	✓	✗	✗	✓	✓	✓
forall quantifier	✓	✗	✗	✗	✗	✗	✓
exists quantifier	✓	✗	✗	✗	✗	✗	✓
scalene triangle example	✓	✗	✗	✗	✗	✗	✓

Table 34. Object Type Benchmark Generation Results Overview

concurrency PEX works for single threaded programs only.

native code Native code cannot be instrumented and therefore PEX cannot collect constraints on it. Nevertheless, PEX will try to generate input even without exact knowledge about the native code.

nondeterminism PEX assumes that the program under test is deterministic. In case PEX determines nondeterministic behavior through comparing actual with expected behavior (from previous runs) the search space is pruned and a warning is issued.

symbolic reasoning PEX uses Z3 for instantiating concrete values from (path) constraints, but SMT solvers do have limitations which therefore apply to PEX as well.

The first two issues hold for all tools. The third for all tools that try to achieve coverage in a systematic way. The remaining issues only for tools that use constraint solving.

The impressive evaluation results shown by AgitarOne and PEX are due to progress in solving technologies in recent years. Random testing did not perform very well in this evaluation. We therefore conclude that it does not perfectly fit for testing with respect to coverage. Nevertheless, random testing should be intensively used for robustness testing, which means testing for unexpected input values.

We do not want to classify the tools according to their generation technique, since the evaluation showed that those tools that incorporate different techniques work best. Instead Table 35 gives an overview what different techniques each of the evaluated tools incorporates, according to their publications and the manual inspection of the generated tests. The following paragraphs shortly explain the authors understanding of the row captions.

manual The tool provides a mechanism to add manual values or objects that are used in generated tests. Typically, tools allow to write some lines of code that add manual values or manually constructed objects to a pool of values/objects.

pre-defined values The tool randomly selects one value from a set of hard-coded values for each data type. For example, AutoTest fills its INTEGER pool from which it randomly selects one value with, minimum/-maximum value, 0 +1, -1, +2, -2, +10, and -10.

random The tool randomly generates a value on the fly.

constants extraction The tool extracts constants from the source file, or uses other constant values such as the name of the method, class, or path of the source file.

constants extraction + manipulation Those tools manipulate the extracted constants by a pre-defined set of rules. For example, AgitarOne generates integer type input values that differ by one from those extracted in the source (extracted value: 6; tested values: 5,6,7).

combinatorial testing In case more than one input value has to be generated, tools try to generate (a subset of) all possible combinations of values they have stored in their pool. This row is very unspecific, since all tools generate more than one test, and no tool generates all combinations. Therefore, all tools doe some form of combinatorial testing but none of them incorporates the strict definition.

constraint logic Those tools build up a constraint system that models the testing problem and uses tools such as constraint solvers, or SMT solvers to generate input values that satisfy the constraint system. Typically, those tools incorporate tricks and tweaks to improve generation results that are not well documented or easy to analyze. Therefore, we do not distinguish between different constraint logic approaches.

null objects Those tools use the null object as test input.

random constructor Those tools construct objects by randomly choosing one of the public constructors and executing it with random parameters.

random constructor + manipulation Those tools further manipulate the object state by calling randomly any other methods on the object after construction.

mock/stub generation Those tools generate stub or mock objects. A stub object is only a class or method definition without implementation. A human being has to look at all the generated stubs and write code that returns meaningful values/objects (similar to factory methods). Mock objects include already an implementation describing an object sequence that is for example extracted from a test execution.

The evaluation summarizes the capabilities of the tools to generate input values that satisfies either a given precondition or an assertion (in case the tool does not support Design by Contract^{TM} specifications). In our point of view it is valid to use this evaluation mechanism since the asserted expressions are not specific to Design by Contract^{TM} but can occur through out the source code as (branching-)conditions as well. The used expressions are synthetic, i.e., look very constructed. This is due to the fact that we wanted to clearly find the boundaries of a tool with respect to value generation capabilities. All used expressions are trivial and focus on one feature at a time. In real applications assertion statements or branching-conditions are a combination of those simple evaluation expressions. We argue that if a tool is not able to solve the trivial evaluation expression it will not be able to generate data that satisfies combinations of those expressions. Therefore, we conclude that the presented evaluation result is a good starting point to find out the boundaries of state-of-the art test data generation tools and techniques for object oriented languages.

		AgitarOne Sec. 3.3	AnalytiX Sec. 3.5	AutoTest Sec. 3.4	C++test Sec. 3.2	Jtest Sec. 3.1	RANDOOP Sec. 3.6	PEX Sec. 3.7
Primitive Types	manual	✓	✓	✓	✓			
	pre-defined values		✓	✓	✓	✓	✓	
	random			✓	✓			
	constants extraction	✓	✓		✓	✓		
	constants extraction + manip.	✓						
	combinatorial testing		✓		✓	✓		
	constraint logic	✓						✓
Object Types	null objects	✓	✓	✓		✓	✓	✓
	manual objects			✓	✓		✓	
	pre-defined objects							
	random constructor	✓	✓	✓	✓	✓	✓	
	random constructor + manip.	✓		✓			✓	
	mock/stub generation	✓	✓			✓		
	constraint logic							✓

Table 35. Tool Generation Techniques Overview. This table lists all incorporated generation techniques of the evaluated tools with respect to primitive and object data types.

Acknowledgment. The authors wish to thank the anonymous referees for their detailed and constructive feedback in order to improve the paper. The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

References

1. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
2. Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing Telecoms Software With Quviq QuickCheck. In *2006 ACM SIGPLAN workshop on Erlang*, Portland, Oregon, USA, 2006. ACM.
3. Mike Barnett, Manuel Fähndrich, Jonathan de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the Synergy Between Automated-Test-Generation and Programming-by-Contract. In *Proc. 31st International Conference on Software Engineering (ICSE'2009)*. IEEE, 2009.
4. Mike Barnett, KRM Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
5. Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In *2006 International Symposium on Software Testing and Analysis*, pages 169–180. ACM Press, 2006.
6. I.B. Bourdonov, A. Kossatchev, V.V. Kuliamin, and A. Petrenko. UniTesK Test Suite Architecture. In *FME 2002: Formal MethodsGetting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 121–152. Springer, 2002.
7. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *International Symposium on Software Testing and Analysis*, pages 123–133. ACM Press, 2002.
8. Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. *Formal Methods in System Design*, 25(2/3):167–198, September 2004.
9. Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX, 2008.
10. Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security*, 12(2):322–335, 2008.
11. Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical report, Microsoft Research, Redmond, 2005.
12. Ilinca Ciupa and Andreas Leitner. Automatic Testing Based on Design by Contract. *Proceedings of Net.ObjectDays 2005*, pages 545–557, 2005.
13. Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO: Adaptive Random Testing for Object-Oriented Software. In *30th International Conference on Software Engineering*, pages 71–80. ACM, 2008.
14. Christoph Csallner and Yannis Smaragdakis. JCrasher: an Automatic Robustness Tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.
15. Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In *27th*

ACM/IEEE International Conference on Software Engineering, pages 422–431. ACM, May 2005.

16. Jonathan de Halleux and Nikolai Tillmann. *Moles: Tool-Assisted Environment Isolation with Closures*, volume 6141 of *Lecture Notes in Computer Science*, pages 253–270. Springer, Berlin, Heidelberg, 2010.
17. Michael D. Ernst, Jake CockrelI, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *21st International Conference on Software Engineering*, pages 213–222, Los Alamitos, CA, USA, 1999. IEEE.
18. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *ACM SIGPLAN Notices, Conference on Programming Language Design and Implementation*, 37(5):234–245, 2002.
19. Tammo Freese. EasyMock: Dynamic Mock Objects for JUnit. In *3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2002)*, pages 2–5, 2002.
20. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.
21. Google Inc. ToT: Friends You Can Depend On, 2008. `http://googletesting.blogspot.com/2008/06/tott-friends-you-can-depend-on.html`.
22. W. Grieskamp. Multi-paradigmatic Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2006.
23. W. Grieskamp, X. Qu, X. Wei, N. Kicillof, and M. Cohen. Interaction Coverage Meets Path Coverage by SMT Constraint Solving. In *Testing of Software and Communication Systems*, volume 58262 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2009.
24. Antti Huima. Implementing Conformiq Qtronic. In *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12, Berlin, Heidelberg, 2007. Springer.
25. Google Inc. Codepro analytix user guide. `http://developers.google.com/java-dev-tools/codepro/doc/`.
26. Eddie Jaffuel and Bruno Legeard. LEIRIOS Test Generator: Automated Test Generation from B Models. In *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*, pages 277–280, Berlin, Heidelberg, 2006. Springer.
27. V.V. Kuliamin, A.K. Petrenko, A.S. Kossatchev, and I.B. Bourdonov. UniTesK: Model Based Testing in Industrial Practice. In *1st European Conference on Model Driven Software Engineering*, volume pages, pages 55–63, 2003.
28. Bruno Legeard, Fabien Peureux, and Mark Utting. Automated Boundary Testing from Z and B. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods*, volume 2391 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2002.
29. Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Liu. Automatic Testing of Object-Oriented Software. In *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007.
30. Scott Meyers. *Effecitve C++: 55 Specific Ways to Improve Your Programs and Design.* Addison-Wesley Professional, 3rd edition, 2005.
31. Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
32. Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing.* Wiley, 2nd edition, 2004.
33. Carlos Pacheco and Michael D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *ECOOP 2005: Object-Oriented Programming*, volume 3568 of *Lecture Notes in Computer Science*, pages 504–527. Springer, 2005.
34. Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-Directed Random Testing for Java. In *OOPSLA 2007: Conference on Object Oriented Programming Systems Languages and Applications*, pages 815–816. ACM, 2007.
35. Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding Errors in .NET with Feedback-Directed Random Testing. *ISSTA'08: International Symposium on Software Testing and Analysis*, pages 87–96, 2008.
36. Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, 2007. IEEE.
37. Paraosft. Parasoft C++test User's Guide, 2010.
38. Parasoft. http://www.parasoft.com. `http://www.parasoft.com`.
39. Parasoft. Using Design by Contract to Automate Java Software and Component Testing. `http://www.parasoft.com/jsp/products/article.jsp?articleId=579&product=Jcontract`.
40. John Penix, Willem Visser, SeungJoon Park, Corina Pasareanu, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verifying Time Partitioning in the DEOS Scheduling Kernel. *Formal Methods in System Design*, 26(2):103–135, March 2005.
41. Microsoft Research. Advanced Concepts: Parameterized Unit Testing with Microsoft Pex, 2010. `http://research.microsoft.com/en-us/projects/pex/pexconcepts.pdf`.
42. Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *18th International Conference on Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423, Seattle, Washington, USA, 2006. Springer.
43. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *10th European Software Engineering Conference*, volume 30 of *ACM SIGSOFT Software Engineering Notes*, pages 263–272, Lisbon, Portugal, 2005. ACM.
44. SMTCOMP. Call for Entrants, 2010. `http://www.smtcomp.org/2010/call10.txt`.
45. Nikolai Tillmann and J. De Halleux. Pex – White Box Test Generation for .NET. In *Proceedings of the 2nd international conference on tests and proofs (TAP 2008)*,

volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

46. Nikolai Tillmann, Wolfgang Grieskamp, and Wolfram Schulte. Parameterized Unit Tests. *SIGSOFT Software Engineering Notes*, 30(5):253–262, August 2005.
47. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 498–507. IEEE, 2010.
48. Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.
49. Willem Visser, Klaus Havelund, Guillaume Brat, S.J. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.
50. Willem Visser, Corina S. Psreanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.
51. Yi Wei, Serge Gebhardt, Manual Oriol, and Bertrand Meyer. Satisfying Test Preconditions through Guided Object Selection. In *3rd International Conference on Software Testing, Verification and Validation*, pages 1–10, Paris, France, 2010. IEEE.