Automated Combinatorial Test Methods – Beyond Pairwise Testing

D. Richard Kuhn and Dr. Raghu Kacker
National Institute of Standards and Technology

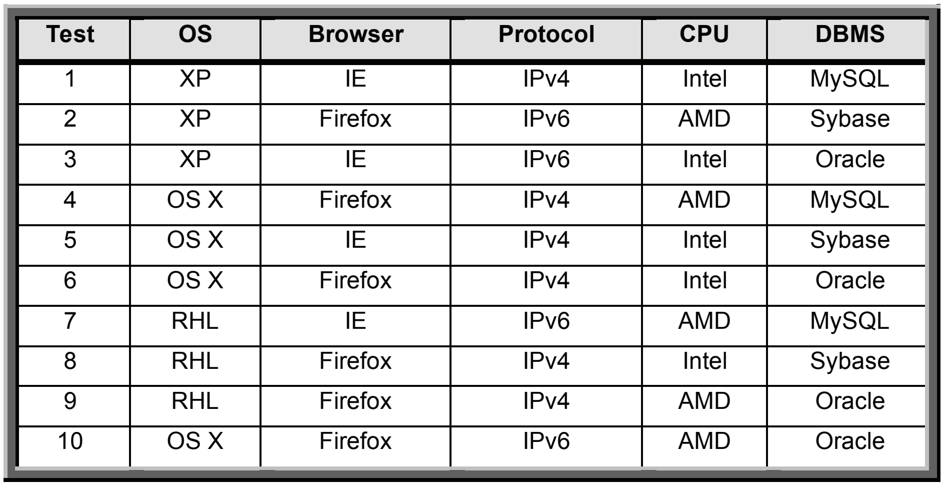
Dr. Yu Lei
University of Texas, Arlington

Pairwise testing has become a popular approach to software quality assurance because it often provides effective error detection at low cost. However, pairwise (2-way) coverage is not sufficient for assurance of mission-critical software. Combinatorial testing beyond pairwise is rarely used because good algorithms have not been available for complex combinations such as 3-way, 4-way, or more. In addition, significantly more tests are required for combinations beyond pairwise testing, and testers must determine expected results for each set of inputs. This article introduces new tools for automating the production of complete test cases covering up to 6-way combinations.

Many testers are familiar with the most basic form of combinatorial testing – *all pairs* or pairwise testing, in which all possible pairs of parameter values are covered by at least one test [1, 2]. Pairwise testing uses specially constructed test sets that guarantee testing every parameter value interacting with every other parameter value at least once. For example, suppose we had an application that is intended to run on a variety of platforms comprised of five components: an operating system (Windows XP, Apple OS X, Red Hat Linux), a browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), a processor (Intel, AMD), and a database (MySQL, Sybase, Oracle), a total of 3 x 2 x 2 x 2 x 2 = 48 possible platforms. With only 10 tests, as shown in Figure 1, it is possible to test every component interacting with every other component at least once, i.e., all possible pairs of platform components. The effectiveness of pairwise testing is based on the observation that software faults often involve interactions between parameters. While some bugs can be detected with a single parameter value, such as a divide-by-zero error, the toughest bugs often can only be detected when multiple conditions are true simultaneously. For example, a router may be observed to fail only for the User Datagram Protocol (UDP) when packet rate exceeds 1.3 million packets per second – a 2-way interaction between protocol type and packet rate. An even more difficult bug might be one which is detected only for UDP when packet volume exceeds 1.3 million packets per second and packet chaining is used – a 3-way interaction between protocol type, packet rate, and chaining option.

Unfortunately, only a handful of tools can generate more complex combinations, such as 3-way, 4-way, or more (we refer to the number of variables in combinations as the *combinatorial interaction strength*, or simply, interaction strength, e.g., a 4-way combination has 4 variables and thus its interaction strength is 4). The few tools that do generate tests with interaction strengths higher than 2-way may require several days to generate tests [3] because the generation process is mathematically complex. Pairwise testing, i.e. testing 2-way combinations, has come to be accepted as the standard approach to combinatorial testing because it is computationally tractable and can effectively detect many faults. For example, pairwise testing could detect 70 percent to more than 90 percent of software faults for the applications studied in [4].

Figure 1: *Pairwise Test Configurations*



Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

But if pairwise testing can detect 90 percent of bugs, what interaction strength is needed to detect 100 percent? Surprisingly, we found no evidence that this question had been studied when the National Institute of Standards and Technology (NIST) began investigating software faults in 1996. Results showed that across a variety of domains, all failures could be triggered by a maximum of 4-way to 6-way interactions [5]. As shown in Figure 2, the detection rate increases rapidly with interaction strength. With the NASA application, for example, 67 percent of the failures were triggered by only a single parameter value, 93 percent by 2-way combinations, and 98 percent by 3-way combinations. The detection rate curves for the other applications are similar, reaching 100 percent detection with 4-way to 6-way interactions. That is, six or fewer variables were involved in all failures for the applications studied, so 6-way testing could, in theory, detect all of the failures. While not conclusive, these results suggest that combinatorial testing that exercises high strength interaction combinations can be an effective approach to high-integrity software assurance.

Applying combinatorial testing to real-world software presents a number of challenges. For one of the best algorithms, the number of tests needed for combinatorial coverage of n parameters with v values each is proportional to $v^t \log n$, where t is the interaction strength [3]. Unit testing of a small module with 12 parameters required only a few dozen tests for 2-way combinations, but approximately 12,000 for 6-way combinations [6]. But a large number of test cases will not be a barrier if they can be produced with little human intervention, thus reducing cost. To apply combinatorial testing, it is necessary to find a set of test inputs that covers all t-way combinations of parameter values, and to match up each set of inputs with the expected output for these input values.

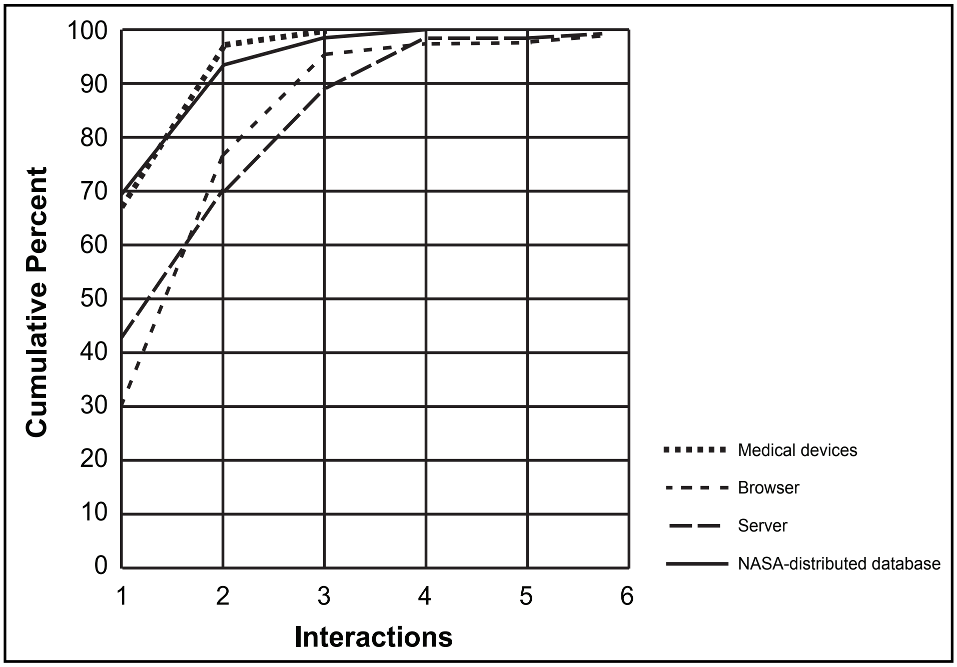


Figure 2: *Error Detection Rates for Interaction Strengths 1 to 6*

These are both difficult problems, but they can now be solved with new algorithms on currently available hardware. We explain these two steps followed by a small but complete illustrative example.

Computing T-Way Combinations of Input Values Using FireEye

The first step in combinatorial testing is to find a set of tests that will cover all *t*-way combinations of parameter values for the desired combinatorial interaction strength *t*. This collection of tests is known as a *covering array*. The covering array specifies test data where each row of the array can be regarded as a set of parameter values for an individual test. Collectively, the rows of the array cover all *t*-way combinations of parameter values. An example is given in Figure 3, which shows a 3-way covering array for 10 variables with two values each. The interesting property of this array is that any three columns contain all eight possible values for three binary variables. For example, taking columns F, G, and H, we can see that all eight possible 3-way combinations (000, 001, 010, 011, 100, 101, 110, 111) occur somewhere in the rows of the three columns. In fact, this is true for any three columns. Collectively, therefore, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage. Similar arrays can be generated to cover up to all 6-way combinations. A non-commercial research tool called FireEye [3], developed by NIST and the University of Texas at Arlington[1], makes this possible with much greater efficiency than previous tools. For example, a commercial tool required 5,400 seconds to produce a less-optimal test set than FireEye generated in 4.2 seconds.

Matching Combinatorial Inputs With Expected Outputs Using Nu Symbolic Model Verifier (SMV)

The second step in combinatorial test development is to determine what output should be produced by the system under test for each set of input parameter values, often referred to as the *oracle problem* in testing. The conventional approach to this problem is human intervention to design tests and assign expected results or, in some cases, to use a *reference implementation* that is known to be correct (for example, in checking conformance of various vendor products to a protocol standard). Because combinatorial testing can require a large number of tests, an automated method is needed for determining the expected results for each set of input data. To solve this problem, we use the open-source NuSMV model checker [7] (an enhanced version of the well-known SMV model checker [7]). Conceptually, the model checker can be viewed as exploring all states of a system model to determine if a property claimed in a specification statement is true. What makes a model checker particularly valuable is that if the claim is false, the model checker not only reports this, but also provides a *counterexample* showing how the claim can be shown false. As will be seen in the illustrative example, this gives us the ability to match every set of input test data with the result that the system should produce for that input data. Figure 4 outlines the process.

The model checker thus automates the work that normally must be done by a human tester – determining what the correct output should be for each set of input data. Other approaches to determining the correct output for each test can also be used. For example, in some cases we can run a model checker in simulation mode, producing expected results directly rather than through a counterexample, but the approach illustrated in this article is more general, and can be applied to non-deterministic systems or used with mutation-based methods in addition to combinatorial testing [8]. The method chosen for resolving the oracle problem depends on the problem at hand, but model checking can be effective in testing protocols, access control, or other applications where there is a state machine, unified modeling language state chart, or other formal model available.

Illustrative Example

Here we present a small example of an access control system. The rules of the system are a simplified multi-level security system, followed by a step-by-step construction of tests using an automated process. Each subject (user) has a clearance level *u_l*, and each file has a classification level *f_l*.

Figure 3: *3-way Covering Array for 10 Parameters With Two Values Each*

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

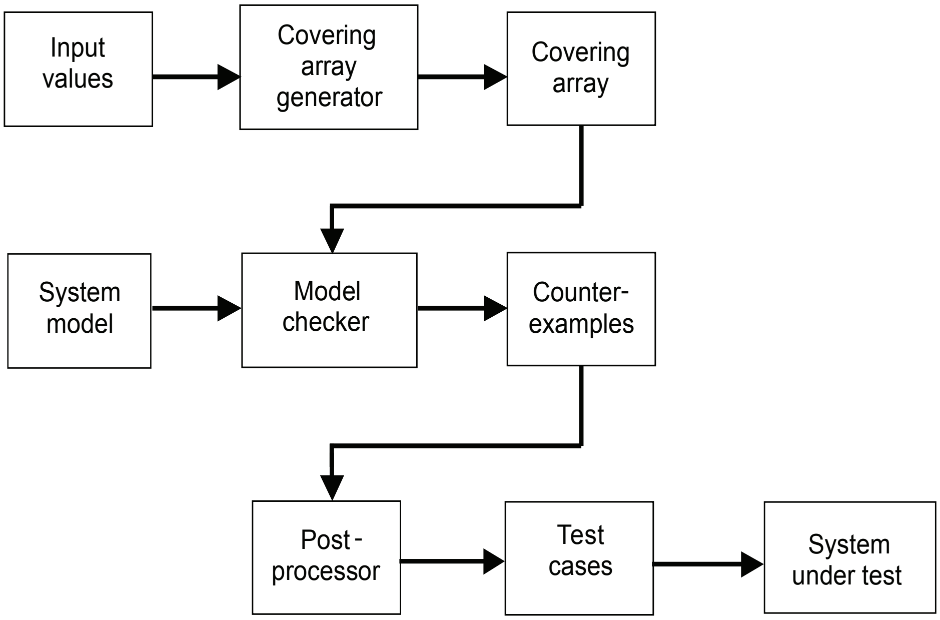


Figure 4: *Automated Combinatorial Test Construction*

Levels are given as 0, 1, or 2, which could represent levels such as Confidential, Secret, and Top Secret. A user *u* can read a file *f* if $u\_l \geq f\_l$ (the *no read up* rule), or write to a file if $f\_l \geq u\_l$ (the *no write down* rule).

Thus, a pseudo-code representation of the access control policy is:

```
if u_l >= f_l & act = rd then
   GRANT;
else if f_l >= u_l & act = wr
   then GRANT; else DENY;
```

Tests produced will check that these rules are correctly implemented in a system.

Figure 5: *SMV Model of Access Control Rules*

```
   1. MODULE main
   2. VAR
        --Input parameters
   3. u_l:   0..2;          -- user level
   4. f_l:   0..2;          -- file level
   5. act:  {rd, wr};       -- action

        --output parameter
   6. access: {START_, GRANT,DENY};

   7. ASSIGN
   8. init(access) := START_;
        --if access is allowed under rules, then next state is GRANT
        --else next state is DENY
   9. next(access) := case
   10.      u_l >= f_l & act = rd : GRANT;
   11.      f_l >= u_l & act = wr : GRANT;
   12.      1 : DENY;
   13.      esac;

   14.      next(u_l) := u_l;
   15.      next(f_l) := f_l;
   16.      next(act) := act;

-- reflection of the assigns for access
-- if user level is at or above file level then read is OK
SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));

-- if user level is at or below file level, then write is OK
SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));

-- if neither condition above is true, then DENY any action
SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ))
         -> AX (access = DENY));
```

System Model

This system is easily modeled in the language of the NuSMV model checker as a simple two-state finite state machine. Other tools could be used, but we illustrate the test production procedure using NuSMV because it is among the most widely used model checkers and is freely available. Our approach is to model the system as a simple state machine, then use NuSMV to evaluate the model and post-process the results into complete test cases.

Figure 5 shows the system model defined in SMV. The START state initializes the system (line 8), with the rule noted previously used to evaluate access as either GRANT or DENY (lines 9-13). For example, line 10 represents the first line of the pseudo-code example: in the current state, (always START for this simple model), if $u\_l \geq f\_l$ then the next state is GRANT. Each line of the case statement is examined sequentially, as in a conventional programming language. Line 12 implements the *else DENY* rule, since the predicate *1* is always true. SPEC clauses given at the end of the model define statements that are to be proven or disproven by the model checker. The SPEC statements in Figure 5 duplicate the access control rules as temporal logic statements and are, thus, provable. In the following sections, we illustrate how to combine them with input data values to generate complete tests with expected results.

In SMV, specifications of the form AG (predicate 1) -> AX (predicate 2) indicate essentially that for all paths (the A in AG) for all states globally (the G), if predicate 1 holds then (->) for all paths, in the next state (the X in AX) predicate *2* will hold. SMV checks the properties in the SPEC statements and shows that they match the access control rules as implemented in the finite state machine, as expected. Once the model is correct and SPEC claims have been shown valid for the model, counterexamples can be produced that will be turned into test cases.

Generating Covering Array

We will compute covering arrays that give all *t*-way combinations, with degree of interaction coverage two for this example. If we had a larger number of parameters, we would produce test configurations that cover all 3-way, 4-way, etc., combinations. (With only three parameters, 3-way interaction would be equivalent to exhaustive testing, so we use 2-way combinations for illustration purposes.) The first step is to define the parameters (using the graphical user interface if desired) and their values in a system definition file that will be used as input to the covering array generator FireEye with the following format: After the system definition file is saved, we run FireEye, in this case specifying 2-way interactions. FireEye produces the output shown in Figure 6.

Each test configuration defines a set of values for the input parameters `u_l`, `f_l`, and `act`. The complete test set ensures that all 2-way combinations of parameter values have been covered

Model Claims With Covering Array Values Inserted

The next step is to assign values from the covering array to parameters used in the model. For each test, we write a claim that the expected result will not occur. The model checker determines combinations that would disprove these claims, outputting these as counterexamples. Each counterexample can then be converted to a test with known expected result. For example, for Test 1 the parameter values are:

```
u_l = 0 & f_l = 0 & act = rd
```

For each of the nine configurations in the covering array (Figure 7), we create a `SPEC` claim of the form: `SPEC AG`*(covering array values)* -> `AX` *!(access = result).*

This process is repeated for each possible result, in this case either *`GRANT`* or *`DENY`*, so we have nine claims for each of the two results. The model checker is able to determine, using the model defined previously, which result is the correct one for each set of input values, producing a total of nine tests.

Excerpt:

```
SPEC AG((u_l = 0 & f_l = 0 & act
   = rd) -> AX !(access = GRANT));
SPEC AG((u_l = 0 & f_l = 1 & act
   = wr) -> AX !(access = GRANT));
SPEC AG((u_l = 0 & f_l = 2 & act
   = rd) -> AX !(access = GRANT));
etc.

SPEC AG((u_l = 0 & f_l = 0 & act
   = rd) -> AX !(access = DENY));
SPEC AG((u_l = 0 & f_l = 1 & act
   = wr) -> AX !(access = DENY));
SPEC AG((u_l = 0 & f_l = 2 & act
   = rd) -> AX !(access = DENY));
etc.
```

Generating Counterexamples With Model Checker

NuSMV produces counterexamples where the input values would disprove the claims specified in the previous section. Each of these counterexamples is, thus, a set of test data that would have the expected result of `GRANT` or `DENY.` For each SPEC claim, if this set of values cannot in fact lead to the particular result, the model checker indicates that this is true. For example, for the configuration below, the claim that access will not be granted is true, because the user's clearance level (`u_l = 0`) is below the file's level (`f_l = 2`):

```
-- specification AG (((u_l
= 0 & f_l = 2) & act = rd)
-> AX !(access = GRANT)) is
true
```

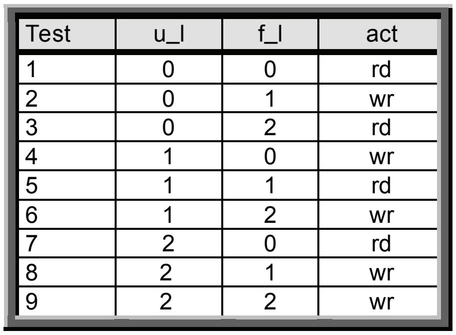
If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false. In effect, this is a complete test case, i.e., a set of parameter values and an expected result. It is then simple to map these values into complete test cases in the syntax needed for the system under test. An excerpt from NuSMV output is shown in Figure 8.

The model checker finds that six of the input parameter configurations produce a result of GRANT and three produce a DENY result, so at the completion of this step we have successfully matched up each input parameter configuration with the result that should be produced by the system under test.

At first, the method previously described may seem *backward*. Instead of negating each possible result, why not simply produce tests from model checker output such as `specification AG (((u_l = 0 & f_l = 2) & act = rd) -> AX (access = DENY))` *is true*? Such a procedure would work fine for this simple example, but more sophisticated testing may require more information. Note that if the claim is true, the model checker simply reports the fact while if it is false, a trace of inputs and internal states is produced to show how the claim fails. Some testing may require information on internal states or variable values, and the previous procedure provides this information.

```
u_l: 0,1,2
f_l: 0,1,2
act: rd, wr
```

Figure 6: *Model Parameters and Values*



Test	u_l	f_l	act
1	0	0	rd
2	0	1	wr
3	0	2	rd
4	1	0	wr
5	1	1	rd
6	1	2	wr
7	2	0	rd
8	2	1	wr
9	2	2	wr

Figure 7: *FireEye Output Test Values*

Shell Script Post-Processing to Produce Complete Tests

The last step is to use a post-processing tool that reads the output of the model checker and generates a set of test inputs with expected results. The post-processor strips out the parameter names and values, giving tests that can be applied to the system under test. Simple scripts are then used to convert the test cases into input for a suitable test harness. The tests produced are shown in Figure 9 (see next page).

Conclusion

While tests for this trivial example could easily have been constructed manually, the procedures introduced in this tutorial can – and have – been used to produce tens of thousands of complete test cases in a few minutes once the SMV model

Figure 8: *Counterexamples (excerpt)*

```
-- specification AG (((u_l = 0 & f_l = 0) & act = rd)
       -> AX !(access = GRANT))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  u_l = 0
  f_l = 0
  act = rd
  access = START_
-> Input: 1.2 <-`
-> State: 1.2 <-
  access = GRANT
...
etc.
```

```
u_l = 0 & f_l = 0 & act = rd -> access = GRANT
u_l = 0 & f_l = 1 & act = wr -> access = GRANT
u_l = 1 & f_l = 1 & act = rd -> access = GRANT
u_l = 1 & f_l = 2 & act = wr -> access = GRANT
u_l = 2 & f_l = 0 & act = rd -> access = GRANT
u_l = 2 & f_l = 2 & act = rd -> access = GRANT
u_l = 0 & f_l = 2 & act = rd -> access = DENY
u_l = 1 & f_l = 0 & act = wr -> access = DENY
u_l = 2 & f_l = 1 & act = wr -> access = DENY
```

Figure 9: *Test Cases*

has been defined for the system under test. The methods in this article still require human intervention and engineering judgment to define a formal model of the system under test and for determining appropriate abstractions and equivalence classes for input parameters. But by automating test generation we can provide much more thorough testing than is possible with most conventional methods. In addition, the testing has a sound empirical basis in the observation that software failures have been shown to be caused by the interaction of relatively few variables. By testing all variable interactions to an appropriate strength, we can provide stronger assurance for critical software.◆

References

1. Daich, G.T. "New Spreadsheet Tool Helps Determine Minimal Set of Test Parameter Combinations." CrossTalk Aug. 2003.
2. Phadke, M.S. "Planning Efficient Software Tests." CrossTalk Oct. 1997.
3. Lei, Y., R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing." Software Testing, Verification, and Reliability (to appear 2008).
4. Kuhn, D.R., D. Wallace, and A. Gallo. "Software Fault Interactions and Implications for Software Testing." IEEE Transactions on Software Engineering 30(6):418-421, 2004.
5. Wallace, D.R., and D.R. Kuhn. "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data." International Journal of Reliability, Quality and Safety Engineering 8(4):351-371, 2001.
6. Kuhn, D.R., and V. Okun. "Pseudo-Exhaustive Testing for Software." Proc. of 30th NASA/IEEE Software Engineering Workshop. Apr. 2006.
7. Cimatti, A., E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NuSMV 2: An OpenSource Tool for Symbolic Model Checking." Proc. of International Conference on Computer-Aided Verification. Copenhagen, Denmark.
8. Ammann, P., and P.E. Black. "Abstracting Formal Specifications to Generate Software Tests via Model Checking." Proc. of 18th Digital Avionics Systems Conference. St. Louis, MO. Oct. 1999.

Notes

1. Available on <http://csrc.nist.gov/acts>.
2. The tool can be downloaded at <http://nusmv.irst.itc.it/>. More information on SMV can be found at <www.cs.cmu.edu/~modelcheck/>.

About the Authors

D. Richard Kuhn is a computer scientist in the computer security division of the National Institute of Standards and Technology (NIST). His primary technical interests are in information security, software assurance, and empirical studies of software failure. He co-developed the role based access control model (RBAC) used throughout industry, and led the effort to establish RBAC as an American National Standards Institute standard. Kuhn has a masters degree in computer science from the University of Maryland, College Park, and a bachelors and master of business administration from William & Mary.

NIST
MS 8930
Gaithersburg, MD 20899-8930
Phone: (301) 975-3337
Fax: (301) 975-8387
E-mail: kuhn@nist.gov

Yu Lei, Ph.D., is an assistant professor of computer science at the University of Texas, Arlington. He was a member of the Fujitsu Network Communications, Inc., technical staff from 1998 to 2001. Lei's research is in the area of automated software analysis, testing, and verification. His current research is supported by NIST. Lei has a bachelor's degree from Wuhan University, a master's degree from Chinese Academy of Sciences, and a doctorate from North Carolina State University.

The University of Texas
at Arlington
Department of Computer
Science and Engineering
P.O. Box 19015
Arlington, TX 76019-0015
Phone: (817) 272-2341
Fax: (817) 272-3784
E-mail: ylei@cse.uta.edu

Raghu Kacker, Ph.D., is a mathematical statistician in the mathematical and computational sciences division of the NIST. His current interests include software testing, uncertainty in physical and virtual measurements, interlaboratory evaluations, and Bayesian uncertainty in measurement. Kacker received his doctorate in statistics from Iowa State University.

NIST
100 Bureau DR
MS 8910
Gaithersburg, MD 20899-8910
Phone: (301) 975-2109
Fax: (301) 975-3553
E-mail: raghu.kacker@nist.gov