Building T-wise Combinatorial Interaction Test Suites by means of Grid computing

Andrea Calvagna[1], Angelo Gargantini[2], Emiliano Tramontana[3]
[1]Dip. di Ingegneria Informatica e delle Telecomunicazioni, Università di Catania, Italy
[2]Dip. di ingegneria Informatica e Metodi Matematici, Università di Bergamo, Italy
[3]Dip. di Matematica e Informatica, Università di Catania, Italy
[3]Consorzio COMETA
andrea.calvagna@unict.it, angelo.gargantini@unibg.it, tramontana@dmi.unict.it

Abstract

Generally, systematic testing is the only way to assess the occurrence of failures in systems consisting in a set of components, however given the size of real-world systems, it would be very expensive to construct and test all the possible combinations of the states of components. Combinatorial interaction testing is an existing tecnique that appropriately reduces the number of test cases by choosing either pairs, triplets, etc., i.e. t-tuples, of input values. Of course, the effectiveness of a test suite is higher when choosing e.g. triplets of inputs rather than pairs. Since high values of t are preferable, a large number of test cases could still be generated.

This paper proposes a technique for building the smallest possible test suite of size t. This technique consists in reducing the number of test cases by carefully choosing non redundant t-tuples. The paper shows that for obtaining the smallest possible set of tests, it is best to generate a large 'flexible' set of t-tuples and then reduce such a set until the smallest one is obtained. Reduction is a computationally expensive operation and therefore it is worth performing it by parallelising its execution. This paper proposes a solution for executing the reduction algorithm over a set of Grid resources.

1. Introduction

Verification and validation of highly-configurable software systems, such as those supporting many optional or customizable features, is a challenging activity. In fact, due to its intrinsic complexity, formal specification of the whole system often require a great effort. Hence, modeling activities may become extremely expensive and time consuming, and the tester usually decide to model (at least initially) only the inputs and require they are sufficiently covered by tests. On the other hand, traditional approaches to systematic testing based on structural coverage criteria are unsuitable to detect incorrect behaviors or failures caused by unintended interaction between optional features [11], [17]. Since most of the faults in a software system are triggered by unintended interaction of a relatively low number of input parameters, typically 4 to 6 [11], testing all the combinations of input configurations can be very effective in revealing software defects [9].

Combinatorial Interaction Testing (CIT) systematically explores *t*-wise feature interactions inside a given system, by effectively combining all t-tuples of parameter assignments in the smallest possible number of test cases [11], [2]. In particular, *pairwise* testing ($t = 2$), consists in testing all *pairs* of input values at least once. It has been empirically confirmed that pairwise testing can detect a significantly large part (typically 50% to 75%) of faults in a system [13], [4], thus many CIT approaches (see [14] for an up-to-date listing) have been developed [3], [6], [11] and are currently applied in practice [1], [16], [10]. CIT can be applied to a wide variety of problems: highly-configurable software systems, software product lines, hardware systems, etc.

Table 1 reports the input domain model of a system with four distinct types of component, each of which can be set in one of three possible optional values. While testing all the possible configurations for this system would require $3^4 = 81$ tests, pairwise coverage can be obtained by the test suite reported in Table 2 that contains just 9 tests.

As shown by the previous example, significant time and cost savings can be achieved by using CIT techniques. In fact, a *t*-wise covering test of a system with n parameters, each ranging in r values, would require $r^t\binom{n}{t}$ t-tuples, i.e. a number of configurations that grows exponentially. However, while each configuration concerns only a subset of t ($t < n$) out of n parameters, a complete test case (a row assigning all the parameters) actually hosts $r^t\binom{n}{t}$ different configurations at the same time. As a consequence, by properly choosing the values of the test cases, the size of the resulting covering test suite can be lowered dramatically. E.g., for a system with a hundred boolean parameters (2^{100} exhaustive test cases), pairwise coverage would require 19800 configurations to be tested, however these can be smartly chosen in a test suite of only 10 test cases. Similarly, pairwise coverage of a system with twenty ten-valued options (10^{20} total test cases) requires coverage of 19000 pairs, for which a combinatorial test suite of only 200 tests cases, or less, can be computed.

Client	Web Server	Payment	Database
FireFox	WebSphere	MasterCard	DB/2
IE	Apache	VISA	Oracle
Opera	.Net	AmEx	Access

Table 1. A system having three options per feature

#	Client	Web Server	Payment	Database
1	FireFox	WebSphere	MasterCard	DB/2
2	FireFox	.Net	AmEx	Oracle
3	FireFox	Apache	VISA	Access
4	IE	WebSphere	AmEx	Access
5	IE	Apache	MasterCard	Oracle
6	IE	.Net	VISA	DB/2
7	Opera	WebSphere	VISA	Oracle
8	Opera	.Net	MasterCard	Access
9	Opera	Apache	AmEx	DB/2

Table 2. A pairwise covering test suite

Nevertheless, it should be noted that computing a CIT test suite is far from trivial, as the t-tuples for the test suite cannot be chosen independently of each other. In fact, it is very easy to have many redundant (repeated) t-tuples in the suite, originating from poorly combined t-tuples, hence the resulting test suite will then be bigger than necessary. Seroussi and Bshouty [15] showed that the problem of computing the least sized CIT test suite for t-wise coverage of a given system is NP-complete. Despite exact solutions to compute it by *algebraic* construction do exist [8], in fact these are not generally applicable to all systems. As a consequence, researchers have addressed the issue of designing generally applicable algorithms, based on *greedy* heuristics, although these may lead to sub-optimal results (typically, only an upper bound on the size of the constructed suite may be guaranteed).

Although many strategies for pairwise test suite construction have been already explored and proposed in the scientific literature, only a few of them support interaction degrees higher than pairwise [12]. In fact, without adequate computational resources the exponential increase in time and space complexity of the problem would make any implementation of impractical usage. On the other hand, higher interaction degrees of coverage can be remarkably more effective in revealing bugs than pairwise coverage, due to the fact that t-wise coverage is by construction a superset of all previous degrees of coverage, which makes this subject worth investigating. In this paper, a new approach to t-wise CIT test suite construction is presented, which is based on the use of Grid computing resources to tackle the exponential complexity of the problem.

The paper is structured as follows. Section 2 gives insights on the theoretical aspects of the considered problem and explains the idea behind our new construction technique. Section 3 describes the proposed approach for building and reducing a test suite. Section 4 gives the details of the algorithm for reducing the test suite. Section 5 provides the details for distributing the reduction algorithm on a Grid environment. Eventually, section 6 draws our conclusive statements.

2. Background

In this section, we will briefly introduce some basic notation for the combinatorial object named *Covering Array* (CA). Hartman et al. [7] have given the following definitions of covering array and mixed level covering array.

Covering array. A covering array, $CA(N;t;k,r)$, is an $N \times k$ array on r symbols such that every $N \times t$ sub-array contains all ordered subsets from r symbols of size t at least once. In such an array, t is called strength, k the degree and r the order. Note that all the parameters of the considered system have the same domain. When the parameters have different domains, *Mixed Covering Arrays* are used, which are a generalization of CAs.

Mixed covering array. A mixed covering array, $MCA(N;t;k;r_1,r_2,...r_k)$ is an $N \times k$ array on r symbols, where $r = \sum_{i=1}^{k} r_i$, with the following properties. (i) Each column i $(1 \leq i \leq k)$ contains only elements from a set P_i of size r_i. (ii) The rows of each $N \times t$ sub-array cover all t-tuples of values from the t columns at least once. Similarly, t is called strength, k the degree and r_i the order of the i^{th} column.

A covering array (or a mixed covering array) is optimal if it contains the minimum possible number of rows. A short-hand notation is used to describe covering arrays by combining equal entries in r_i. The example given in Section 1 is an instance of $CA(9;2;4,3)$ (or denoted as $CA(9;2;3^4)$). Similarly, the mixed covering array $MCA(6;2;5;3,2,2,2,2)$ can be denoted with the shorthand $MCA(6;2;3^1 2^4)$, that is a system with five components, one with three possible configurations and four with binary options. Often we refer to a CA or MCA of strength t as a t-wise covering array or t-covering array. When $t = 2$, it is also called a pairways covering array.

t-wise coupling. Let $P = \{P_1, P_2, ..P_n\}$ be a set whose elements are factors, each representing the parameters of the system under test. Let a factor P_i be a set of r_i distinct symbols (values) $P_i = \{v_1, v_2, .., v_{r_i}\}$, with r_i the range of factor P_i. Please note that as all the symbols are distinct by assumption, all P_i are disjunct and their elements may be conveniently represented by a unique integer enumeration[1]. The following notations can be used to conveniently

1. For enhanced clarity, we will also use the term *factor combination* to refer to a combination of factors from the set P (e.g., (P_1, P_2)), while the term *value combination* refers to a combination of symbols (values) assigned to a given factor combination (i.e., $(P1, P2) \rightarrow (3, 5)$).

represent set of combinations:

$$\begin{aligned} P_1 + P_2 + ... + P_n &: \text{ no combinations.} \\ P^t_{1..n} &: \text{ all t-strenght combinations.} \\ P_1 \cdot P_2 \cdot ... \cdot P_n &: \text{ all combinations.} \end{aligned}$$

Where the $\cdot$ (*dot*) product denotes the cartesian product of the elements of two given sets (*dot* can also be omitted), and the $+$ (*sum*) operator denotes the $\cup$ (set union) of two given sets.

The *t-wise* combinatorial product can then be defined as a generalization of cartesian product.

Combinatorial product. For a given set of constants $1 \leq i, j, .., k, t \leq n$, let $P^t_{i,j,..,k}$ denote an array containing the set of all possible **t-way** combinations of the elements in the set $\{P_i, P_j, .., P_k\} \subseteq P$. As an example, the strength-two expansion of a set of three factors $\{P_1, P_2, P_3\}$ would be $P^2_{1,2,3} = \{P_1P_2 + P_1P_3 + P_2P_3\}$, that is the sum of all their pair-way factor combinations. Note that $P^1_i \equiv P_i$, for all $1 \leq i \leq n$, and by convention $P^0_x \triangleq \emptyset$, for all (set of indexes) x. Also, it is easy to see that $P^1_{i,j..,k} = P_i + P_j + ... + P_k$, and that $P^n_{1..n} = P_1 \cdot P_2 \cdot ... \cdot P_n$.

3. Construction of t-wise covering test suites

The approach for the construction of a t-wise covering test suite chosen in this work is based on the assumption of the availability of large computing resources, such as those offered by a Grid computing environment. In this context, instead of building the test suite incrementally, with a greedy or algorithmic approach, the construction process is based on two complementary computational steps, which are namely the *expansion* and the *contraction* stages, briefly described in the following.

- **Expansion stage**: build up a t-wise covering test suite T by enumerating all combinatorial requirements, one per each row. As only t parameters are involved in each required combination, all other will be left unassigned.
- **Contraction stage**: search for an effective way to combine *compatible* rows together while preserving the coverage, in order to reduce the total number of rows, that is the total number of required tests.

Two rows are said *compatible* if each corresponding position in the rows is either assigned to the same value or it has not been assigned[2] in at least one of the two rows.

The peculiarity of this approach is that it builds up an intermediate test suite enumerating all the t-wise combinations for the parameters of the considered system under test. This would normally be impractical due to its expensiveness in terms of computing resources. On the other hand, if sufficiently large computing resources are available, it is then possible to derive a final test suite from the intermediate one

2. Unassigned positions are commonly marked as x, also called the *don't care* value.

$$\left[\begin{array}{ccc} 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ \hline 1 & 1 & 1 \end{array}\right] \qquad \left[\begin{array}{ccc} 0 & 0 & x \\ 0 & 1 & x \\ 1 & 0 & x \\ 1 & 1 & x \\ x & 0 & 0 \\ x & 0 & 1 \\ x & 1 & 0 \\ x & 1 & 1 \\ 0 & x & 0 \\ 0 & x & 1 \\ 1 & x & 0 \\ 1 & x & 1 \end{array}\right] \qquad \begin{array}{c} \Rightarrow \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & x & 0 \end{array}\right] \\ \\ \Rightarrow \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}\right] \end{array}$$

$$(a) \qquad\qquad (b) \qquad\qquad (c)$$

Figure 1. Pairwise test suites for $P^2_{1,2,3}$ built with greedy (a) and expansion/contraction (b)-(c) methods

by just finding the right way to effectively merge together as many *compatible* rows as possible in order to reduce the total number of rows appearing in the final test suite. Simply put, the final test suite is build by reducing the redundancy in the intermediate one. This is possible since each row in the intermediate test suite has $n - t$ unassigned positions, which can be used to host tuples of other *compatible* rows.

Of course, at the beginning the intermediate test suite does have many compatible rows that will have to be merged. By giving the intermediate test suite maximal redundancy by construction (one row per t-tuple) we enable all the conceivable ways of merging together the t-tuples. It is then possible to implement the contraction (reduction) stage as a search procedure, exploring ways to sequentially merge together the rows of the intermediate test suite, driven by the size of the resulting test suite as its optimization criteria. The search procedure will be ideally able to explore the whole space of all possible sequences of binary merges, and thus find the optimal result, although we might want to limit the depth of the searches or the number of searches to bound the computation time.

In contrast to this potentially exhaustive search-based approach, heuristic based construction techniques are by definition built around some optimization principle, used to make decisions on what might be the best value to assign to a position or the best test to add next. In these approaches, the test suite is computed incrementally, without intermediate stages, aiming at reducing the redundancy on the fly. As a consequence, they typically produce test suites whose rows are then more likely to be already incompatible, that is, where any left redundancy simply cannot be removed without compromising coverage.

Let us consider pairwise coupling of three boolean factors $P^2_{1,2,3}$. The matrix shown in Figure 3-(a) is a pairwise covering test suite built by a greedy algorithm that has enumerated all the pairs of the first two parameters, (P_1, P_2) and then coupled these pairs with each value of the third parameter P_3. This is sufficient to cover all required pairs between (P_1, P_3) and (P_2, P_3), so the greedy algorithm has completed its job. If we now try to reduce the redundancy in the resulting matrix we see that, apart from the first and last rows, all other rows cannot be merged nor deleted as each of them contains at least one non redundant pair that would be lost. Hence, the size of the resulting test suite is six rows.

Conversely, if we start from the expanded set of pairs, shown in Figure 3-(b), then it will be possible to merge compatible rows in many ways, and two possible results are the matrices shown in Figure 3-(c). The first matrix has been obtained by merging each row with the very next compatible one, and its size is five rows. By trying a slightly different sequence of merges the resulting matrix will be the smallest, four rows, and is shown below. Four rows is the optimum (minimal) size for the considered task. Note that this optimal result is unreachable when using the considered greedy construction method. Moreover, we need not have the ability to detect totally redundant rows. This ability is instead needed by the greedy method, in order to find and delete the first and last rows, and is computationally very expensive. With our technique, we simply need to compare two integer vectors.

4. Exploring the solutions space

The drawback of the contraction stage is that the number of possible merge sequences grows exponentially with the size of the (already big) intermediate matrix. Thus, even though computing one possible sequence is an easy task, computing all sequences can still be prohibitively expensive. This is why we assumed that it is possible to distribute this job on Grid resources.

Executing the contraction stage in a distributed computing environment, such as the Grid, allows several possible merge sequences to be computed in parallel on different computing nodes. Then the result is achieved by selecting the smallest matrix that has been produced by one of the sequences. This can improve the quality of the final result over a centralized implementation, as more sequences can be explored in still reasonable computing time, and because the number of sequences that can be checked grows with the number of available computing nodes. In fact, since the total number of possible merge sequences, M, can be much higher then the number of available computing nodes N, some sequential processing could still be needed in order to exhaust the search space.

Figure 4 shows method reduceThread(), a multithreaded algorithm that was derived by adding a few lines, marked with an asterisk, to the original single-threaded algorithm

```
1   reduceThread(TestSuite T){
2    T1 = local copy of T;
3     for each row r1 in T1 {
4       for each row r2 in T1-r1 {
5         if(compatible(r1, r2)) {
6*          wheel = random boolean;
7*          if(wheel) start new reduceThread(T1)
8           merge r2 in r1;
9           remove r2;
10      }
11     }
12    }
13*   if(T1.size()<current_min) {
14*         discard Tmin;
15*         Tmin=T1;
16*   }
17*   else discard T1;
18   }
```

Figure 2. Multithreaded reduction algorithm

that computed only one merge sequence. The parameter passed as an input value to reduceThread() is the previously computed, still redundant, test suite. Method reduceThread() searches for rows that can be merged and hence removed, and in order to ensure termination it proceeds merging compatible rows with a greedy approach. Of course, originally each row may be compatible with many rows in the test suite. Nevertheless, each performed merge will add to the test suite a completely new row, thus determining a potential change in all the compatibility relations. It is not possible to determine in advance if the choice of merging one row with another was the best possible, until reduceThread() has finished, and then the size of the resulting test suite is known.

In order to overcome this limitation, reduceThread() has been designed so as to allow exploring several ways to evolve the reduction process. In fact, before applying a change to the (local copy of the) test suite a new thread is started and it will be given the current version of the suite. The new thread will skip that change and search for an alternative chance of merging (which may be not possible anymore in the previous thread). Actually, exhaustive exploration of all possible sequences of merges would be performed, one per thread, unless some limitation is introduced. This has been done by tying the allocation of each new instance executing reduceThread() to the value of a stochastic variable. Different probability distribution functions for this variable can be used to tune the behaviour of the algorithm to available resources. In this work, we will rely on the resources of a Grid environment to distribute and execute in parallel all the instances running reduceThread(). At termination, each thread will compare its resulting test suite with the one currently minimal, stored in a shared location, and replace it in case of improvements. A running-threads counter is used to monitor the termination

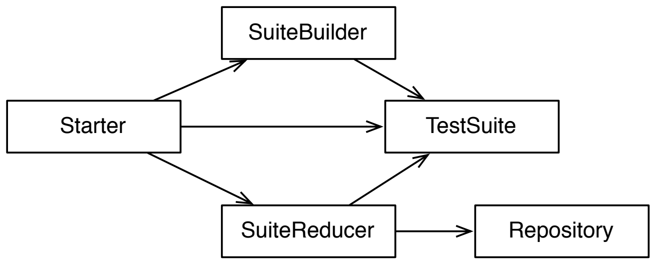

Figure 3. Class diagram for the tool producing a minimal test suite

of the whole process, though it can also be stopped in advance by the user if a satisfying result has already been achieved.

The speed-up of the multithreaded algorithm with respect to the single-thread execution is bound to the number N of available Grid nodes, that is, it is just a direct consequence of the ability to compute N merge sequences at the same time. Note that this does not take into account any delays due to the I/O overhead introduced by distributing the process over a network, but this is acceptable here since the data communication requirements of each thread are negligible with respect to the computing time required to process a complete merge sequence. In fact, apart from the input matrix `T` (the test suite) received at the start, and the output matrix produced at the end, there are no other synchronisation points during the thread executions, which helps keeping the networking overhead delay small. Although the number of overall occurrences of these data synchronizations will be two times the total number of thread instances running, it is still possible to control this latter factor through the *wheel* stochastic variable. This will also be useful to bound the substantially larger space requirements of the multithreaded algorithm, which amounts to an additional working copy of the current test suite for each run thread instance. This can also be easily alleviated by using a compact representation for the test suite data.

5. Distributing the tool producing test suites

Figure 3 shows the main classes of the tool that we have developed in order to generate a minimal test suite, according to the expansion and contraction stage described in Section 3. Class Starter contains the main program and will start the execution of operations for building the test suite and then for reducing it. Class SuiteBuilder generates the initial test suite, as an instance of class TestSuite that contains all the possible rows having each a t set of assigned values. Class SuiteReducer contains the algorithm able to reduce the test suite. Method reduceThread() (described in Section 4) has been implemented in class SuiteReducer. and will start execution when invoked by class Starter. Class Repository contains the version of the test suite that is considered minimal. Class SuiteReducer works on a local copy of the test suite, passed as an input parameter to its method reduceThread(), and after performing checks and merges transfers the resulting test suite to Repository. Checks on new test suites are performed on a separate thread, thus another instance of the same class SuiteReducer is created to work on an appropriate input.

In order to obtain real parallelism between threads, the instances of SuiteReducer should be executed each on a different host. For this, we have used the framework RexMidas [5] that gives support for automatically distributing an application. RexMidas takes as input a centralised Java application and produces a distributed version that can be executed over a set of hosts on a Grid environment. For the aim of distribution, the original application is automatically instrumented, i.e. appropriate jumps to a reflective *metalevel* handling the issues related with distribution are inserted into the bytecode. The metalevel takes care of: handing invocations to application objects that are located remotely, i.e. on a host different than the host of the caller object, sending input parameters and receiving results of invocations to remote objects, finding the host that is more appropriate for allocating new instances of application classes.

The programmer of the application is able to choose which instances of application classes are allowed to be send on a remote host and for these classes, which allocation strategy should be used. This choice should make the runtime execution over the distributed environment more effective for the application at hand. RexMidas provides components implementing allocation strategies that evaluate whether an host is appropriate for holding an instance of a given class, according to several parameters.

For distributing our tool, we have chosen the provided allocation strategy that assesses whether an host owns enough 'static' resources in terms of hardware and software libraries for the object to run smoothly. At run time, this allocation strategy compares the current load (and static resources) for each host and selects the least loaded one.

Figure 4 shows the deployment of the classes when using RexMidas. Class SuiteReducer is declared as distributable and associated with the allocation strategy that balances the load of each available host, e.g. by counting for each host how many instances of SuiteReducer are running and always chosing the host that is the least loaded. This would allow choosing an idle host, when available, for allocating a new instance of SuiteReducer, hence minimising execution time, since this class is CPU-intensive and communication performed is not so frequent.

6. Conclusions

This paper has proposed a technique for building a test suite of size t for a system under test. This technique

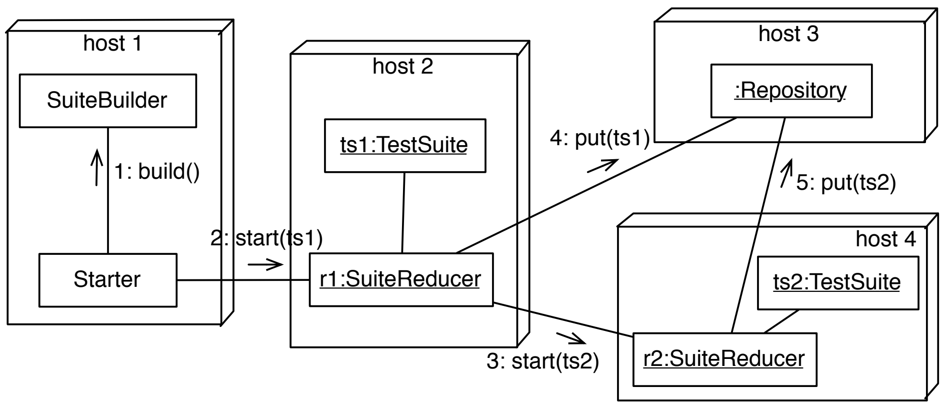

Figure 4. Deployment of the classes for the tool producing a minimal test suite

mainly focuses on the ability to reduce the number of test cases by merging all redundant t-tuples. An algorithm that implements such a technique and that is intrinsically parallel has been described. The benefits that it brings over traditional approaches are both methodological and practical: it is for sure able to arrive to the minimal test suite and can be easily be distributed to take advantage of real parallelism.

In order to be able to distribute over a a set of Grid resources the tool finding the minimal test suite, we have build a modular object-oriented version of the tool and then used the framework RexMidas that takes care of all the concerns related with distribution.

References

[1] R. Brownlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.

[2] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7):437–444, 1997.

[3] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott. Model-based testing of a highly programmable system. *issre*, 00:174, 1998.

[4] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering (ICSE)*, pages 285–295, New York, May 1999. ACM.

[5] P. Giarrusso, G. Pappalardo, L. Toscano, and E. Tramontana. REXMIDAS: A Reflective Middleware for Transparently and Effectively Distributing Objects on a Grid System. In *Proceedings of Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE, 2008.

[6] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.

[7] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.

[8] N. Kobayashi, T. Tsuchiya, and T. Kikuno. Non-specification-based approaches to logic testing for software. *Journal of Information and Software Technology*, 44(2):113–121, February 2002.

[9] D. R. Kuhn and V. Okum. Pseudo-exhaustive testing for software. In *Software Engineering Workshop (SEW)*, Los Alamitos, CA, USA, 2006. IEEE.

[10] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering workshop*, pages 91–95. IEEE, 2002.

[11] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.

[12] Y. Lei, 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*, 18(3):125–148, 2008.

[13] Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *International Symposium on High-Assurance Systems Engineering (HASE '98)*, pages 254–261. IEEE, 1998.

[14] Pairwise web site. http://www.pairwise.org/.

[15] G. Seroussi and N. H. Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.

[16] B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In C. Breckenridge, editor, *Proceedings of Artificial Intelligence Planning Systems (AIPS)*, 2000.

[17] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.