Optimal Scheduling for Combinatorial Software Testing and Design of Experiments

Robert V. Binder
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
rvbinder@sei.cmu.edu

***Abstract*—Test case ordering can have significant effects on the cost, duration, or safety of a test suite. As the total number of possible orderings is *n!* for *n* test cases, finding a cost-optimal ordering can be a non-trivial problem. Combinatorial algorithms that generate *t*-wise test suites either explicitly randomize sequence or order them as a side effect of the algorithm. Design of experiments uses similar strategies to select test configurations and requires sequence randomization for statistical validity. Both approaches produce test sequences that are very likely sub-optimal with respect to cost. This paper presents an integer programming model that minimizes the total cost of a test sequence and notes, for experimental design, how statistical validity may be preserved for a non-random order.**

Keywords—combinatorial software testing, cost minimization, cost reduction, design of experiments, integer programming, operational testing, software testing, split plot design, traveling salesman problem

I. MOTIVATION AND RELATED WORK

A. Testing and Switching Costs

Switching costs are the direct and variable costs that arise when one test case or configuration is followed by another. When they are non-trivial and varying, several key test planning questions arise: How to sequence configurations? What is the least cost sequence? What is the shortest sequence? Would an alternate sequence significantly reduce overall time or cost? For *n* test cases, there are *n!* possible test case sequences. Test plans that rely on a randomized sequence, practical considerations, or a technical prioritization scheme are almost certainly non-optimal with respect to switching costs.

In combinatorial software testing, *factors* are the input or state variables to be controlled for a test. *Levels* refer to the set of values to be assigned to a factor. In this paper, a *test case* is one combination of level values for all factors. A *test suite* is a collection of test cases.

Design of experiments and combinatorial software testing have a common goal of selecting a minimal yet adequate number of test cases. For example, in experimental design, a *two factor interaction* corresponds to a combinatorial *pair-wise* software test suite. Both are said to have an *interaction strength* of 2. Either criterion produces a test suite such that every possible pair of level values occurs at least once in the test suite, referred to as a 2-*way covering array* [1]. Test suites that achieve higher interaction ($t \geq 3$) usually have more test cases.

The switching cost in a typical application of combinatorial test design for software testing is often small enough to be ignored. However, significant switching costs can arise in software systems. A recent attempt at exhaustive configuration coverage of a web application code generator suggests the resources needed for a brute-force approach: "We built a testing scaffold for the 26,000+ configurations of JHipster using a cluster of 80 machines [that ran] for a total of 4376 hour-machine [sic]." [2] More complex systems would require proportionally more resources. For example, suppose a test suite uses on-demand cloud resources and requires creating or restoring a large database prior to each run. Some factor levels could call for a completely empty database and others for one pre-populated with many terabytes of structured instances. Even with present-day scalable cloud resources, the cost of (re)establishing such a database would be significant, especially if the test suite is repeated frequently to support continuous integration [3]. When design of experiments is applied to physical systems, factor level sequences can have significant cost differences. For example, a test is to evaluate the reaction of different alloys in a turbine drive shaft to several fuels. Each shaft change requires thousands of labor hours, so the least cost sequence would minimize the number of shaft changes.

B. Sequence Selection in Software Testing

Test case prioritization strategies seek to reduce test suite run time, often to facilitate regression testing, by "…ordering test cases for early maximization of some desirable properties, such as the rate of fault detection. It seeks to find the optimal permutation of the sequence of test cases. It does not involve selection of test cases, and assumes that all the test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process." [4] Many kinds of test prioritization algorithms have been described and studied [4] [5] [6] [7]. Although test case prioritization strategies typically determine a sequence of test cases, test case to test case switching cost is not among criteria used for their sequencing. Several use integer programming to optimize variables of interest [8] [9] [10] [11] [12].

Combinatorial selection algorithms that produce covering arrays treat the resulting order of test cases as a benign side effect. There is no necessary relationship between these orderings and resultant switching costs. For example, in AETG, the output sequence will be partly determined by the relative frequency of each level value and partly by randomization of candidates for selection at each iteration [13]. The IPOG algorithm first processes factors with the greatest number of levels, and then factors with fewer levels, until a cover is produced [1]. The output sequence reflects the number of levels in each factor. The Testcover.com web service first generates a superset of covering test cases conforming to model constraints. Their sequence is then determined by a greedy search among the test cases [14].

Combinatorial design has been used to generate sequences of actions as test cases. Each test case contains a different action sequence. However, the sequence of these test cases is also a side effect of the generation algorithm. For example, Kuhn et al describe covering-array algorithms for event-driven systems where each factor represents one step in a sequence; their levels are all the same and represent the events to be arranged in a *t*-way sequence, in each test case [15]. Although the coverage algorithm selects different level value combinations representing different test sequences in each generated test case, the test suite sequence is a result of randomization and a greedy algorithm similar to the IPOG strategy. Sherwood shows how elements of extended state machine models may be usefully represented as factors and levels to generate test cases subsets whose order of execution is not constrained [1].

Given a *t*-way software test suite, Kruse represents risk of failure and likelihood of error as unit weights for selected levels, which are used to score each test case or configuration. The test suite may be ordered or prioritized accordingly [16]. Other researchers have developed strategies that use cost-related criteria to order a suite of *t*-way test cases. Kimoto et al present two algorithms that achieve pair-wise cover for copier configuration testing and seek to minimize the total cost of configuration changes [17]. They note that an "…earlier test case has more significant impact on interaction coverage," and present two greedy algorithms that place higher interaction test cases earlier in the test suite. Srikanth et al present an algorithm that orders test cases according to "…configuration switching cost—the lower the cost, the higher the priority." [18] Bryce et al generate a two way covering array of test cases, and then order them from least to highest cost. Test length is used as a proxy for actual cost [19].

C. Sequence Selection in Design of Experiments

Design of experiments is used to design statistically adequate test suites for a wide range of applications including safety-critical electro-mechanical systems [20] [21]. As in combinatorial software testing, controlled variables of an experimental design problem are called factors; the values, states, events, conditions etc. of a factor to be tested are called levels. For example, factors might be terrain, weather, mode of operation, and resource status; the levels for terrain could be desert, at sea, alpine, urban, etc.

An experimental design typically defines *response variables* to be measured as each test case is applied. Conducting an experiment produces a *response data set* analyzed to evaluate its statistical significance, i.e., test adequacy. The test inputs and observed responses may be used to produce a regression model that predicts response variables for all level values. In contrast, combinatorial test design does not use statistical analysis and leaves the overall question of test adequacy to other testing criteria. However, studies have shown that software test suites minimally covering all 3-way interactions reveal nearly all interaction-related bugs [22].

The statistical validity of an experimental design is predicated on a completely randomized sequence of test cases: all factors must be independently reset for each test case as well as randomizing their sequence [23]. However, some test sequences may be more expensive than others, impractical, or dangerous. For example, if test equipment and staff must be transported to a testing facility to achieve a certain level, it may be less expensive to start testing in the closest locale rather than a faraway locale, or vice versa. Although imposing such a sequence will de-randomize the test sequence, *split-plot design techniques* can mitigate confounding effects [24]. Split-plot sequencing techniques use *blocks* (heuristic groupings of levels) such as "easy to change", "hard to change", and "very hard to change" instead of explicit cost values [25].

II. MODEL FORMULATION

A. Overview and Example

None of the noted approaches for test prioritization and combinatorial testing of software determine a cost-optimal test case sequence, although they address related issues. The split-plot technique uses pragmatic qualitative groupings to minimize the confounding effects of de-randomization, but does not attempt to find a cost-optimal sequence. The integer programming model presented here may be applied to test suites produced by either of these combinatorial test design strategies to minimize switching cost.

The test configuration switching cost model will be illustrated with a notional two factor design which may be covered with four configurations, for which there are 4! = 24 possible configurations sequences. The example is a simplified design for testing of a mobile or cellular radio in a physical environment with low or high radio frequency interference resulting from terrain features (urban and desert) and high or low interference resulting from background energy or high energy signals intended to impair endpoint transmissions. All example values are notional.

B. Identification of Switching Costs

Switching costs are the direct and variable costs that arise when a test case or configuration *p* is followed by another, *q*. For any *p* and *q* where *q* immediately follows *p*, their switching cost is denoted *Cost(p,q)*.

For simplicity, quantification of a switch effect is its "cost," but this does not limit effects to monetary units. Some testing projects are more sensitive to elapsed time than resource costs. Timing effects for configuration switching may be modeled by simply using time units instead of monetary units. Costs may be positive, zero, or negative, and should be expressed in the same units: dollars, euros, days, etc.

The present example uses typical testing activities of setup, run, analyze and teardown, but they have no intrinsic significance. Other categories may be used. In any case, these activities should be defined so that they are mutually exclusive and collectively exhaustive.

Cost(p,q) is the sum of the costs to perform test *q*, given that test *p* is its immediate predecessor, i.e., *Cost(p,q) = Setup(p,q) + Run(p,q) + Analyze(p,q) + Teardown(p,q). Setup(p,q)* is the total cost to establish the preconditions and/or configuration for test case *q* after completing the teardown of *p*. This includes the costs necessary to complete the setup of *q*, given antecedent *p*. For example, the cost of transporting staff and equipment from *p* to *q*, the non-variable cost to allocate cloud services necessary for *q* given antecedent *p*, etc. *Run(p,q)* is the direct cost to completely run test case *q* at the levels required given antecedent *p*. *Analyze(p,q)* is the direct cost of evaluating test run results for test case *q* given antecedent *p*. *Teardown(p,q)* is the direct cost to terminate, dismantle, cleanup, etc. test *q* given antecedent *p*. Teardown costs for test *p* are not included *Cost(p,q).* Table 3 provides notional values for our example.

It is likely that some activity costs will be the same. If all activity costs are the same for all *p,q* the cost of all sequences will be same, so no single optimum exists.

Accurate identification and estimation of costs and their dependencies is typically a straightforward task that is often complicated by practical considerations. The example costs could arise from transporting persons and equipment from locale *x* to locale *y*. *Run* and *Analyze* costs for certain levels may be more or less expensive depending on *x* or *y*. For example, testing at night would require overtime pay. Certain level/locale combinations could result in higher setup or teardown costs. For example, batteries must be replaced after five hours, and cost substantially more in locale *x* than locale *y*.

Once activity switching costs are established, they are summarized in the switching cost table. A null starting and ending configuration αΩ is added so that any test case/configuration is a candidate for first or last.

C. *Integer Programming Formulation*

Integer Programming (IP) is a deterministic mathematical optimization technique in which some or all model variables are limited to integer values [26]. If they are further constrained to 0 or 1, the model can represent binary, yes/no conditions. For a scheduling model, a binary integer variable can represent the assignment of an action to a step. When set to 1, it indicates that the action is scheduled in that step. Typically, an IP scheduling model is constructed so that only one action assignment variable for a step is set to 1. All other actions for the step are 0, indicating that they are not scheduled for a step. Costs are represented as objective function coefficients for step/node variables. Given a feasible formulation, IP solvers can determine an exact global optimum for the objective function.

The well-known travelling salesman problem (TSP) was one of the first notable applications of integer programming [27]. In a TSP, a salesman must visit each city in a set of cities once, starting at and returning to the same city. If travel costs among cities are different, some routes will be more or less expensive than others. The problem is to find the least cost route. The IP model of a TSP uses an integer variable for each city and step and a function to return the travel cost between each city. In the solution, variables set to 1 indicate the sequence of cities (nodes) with the least cost route. Although the TSP is NP-complete in the general case, models with many thousands of nodes are routinely solved [28] [29].

The test case sequence scheduling problem is structurally similar to TSP, with the exception that the first and last configuration are not fixed and must not be the same. TSP requires that the route starts at and returns to a “home” node. In the test sequence cost model, a dummy node, αΩ, serves as the home node. The optimal sequence will then begin and end at the αΩ node. This allows the first and last test configurations (nodes) to be determined according to their switching cost.

The formulation begins with an objective function that defines the cost of all *p, q* switches and requires that this function is minimized. Equations representing steps and configurations are added and constrained so that only one configuration may be selected for any step.

Fig. 1 presents the switching cost model expressed in the What'sBest solver [30] which corresponds to general form IP equations [26]. Configurations are numbered from 1 to *n* and tagged in this example to indicate the pair of configurations with which they are associated. The problem is to assign each configuration to a step such that the value of the objective function is minimized over all possible node/step assignments. The model has three parts.

- *Objective Function and Switching Cost Matrix.* Each cell in the transition cost matrix is the total cost of switching from one configuration to another. This is the total of teardown, setup, run, and analyze costs for each factor in the configuration, as shown in Table 4. In WhatsBest, the objective function is the Excel product sum of the cost value matrix elements and each corresponding element of the Sequence Selection matrix. As all elements of the selection matrix are constrained to be either 0 or 1, the objective function always provides the cost of the sequence represented in the Sequence Selection Matrix.
- *Sequence Selection Matrix.* Selection of switch *p,q* is indicated with a 1 and indicates that the test configuration of row *p* is followed by the test configuration of column *q*. The row sums and column sums are constrained to equal 1. This constraint forces the matrix to assign a configuration to exactly one step. The switching cost matrix and the sequence selection matrix must have the same number of rows and columns. The rows and columns must represent the same *p* and *q* in both matrices.
- *Tour Constraints.* As in a canonical TSP IP, the Miller–Tucker–Zemlin (MTZ) constraints are added to prevent the selection of subtours, i.e., sequences that do not include the start/end node [31].

The sequence selection model must be initialized with exactly one 1 in each row and each column. When the IP optimization algorithm completes, this matrix will indicate the optimal solution. The solution for the example problem is shown in Fig 1.

Table 1 Example Factors and Levels

Factor	Level
Terrain	Desert
	Urban
Electro Magnetic Interference (EMI)	Nominal
	Jamming

Table 2 Example Pair-wise Test Suite

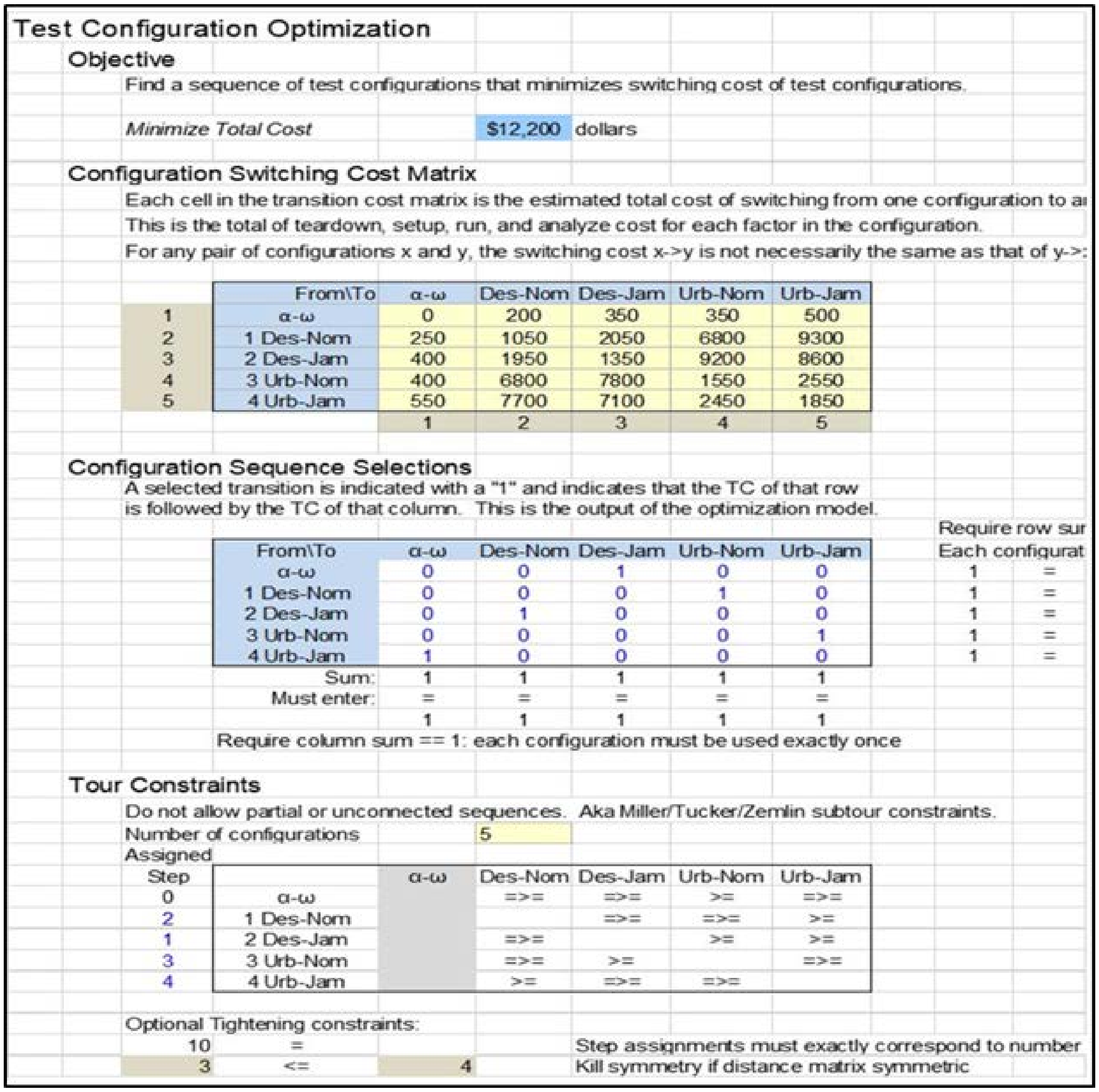
Test Configuration	Terrain	EMI
1	Desert	Nominal
2	Desert	Jamming
3	Urban	Jamming
4	Urban	Nominal

Table 3 Example Switching Cost Tabulations

	Factor 1: Terrain			Factor 2: EMI		
Setup	From/To	Desert	Urban	From/To	Nominal	Jamming
	Desert	100	4500	Nominal	100	500
	Urban	3500	250	Jamming	500	250
Run	From/To	Desert	Urban	From/To	Nominal	Jamming
	Desert	100	100	Nominal	100	300
	Urban	100	300	Jamming	200	100
Analyze	From/To	Desert	Urban	From/To	Nominal	Jamming
	Desert	200	200	Nominal	200	200
	Urban	200	200	Jamming	200	200
Teardown	From/To	Desert	Urban	From/To	Nominal	Jamming
	Desert	150	3000	Nominal	100	500
	Urban	2500	300	Jamming	500	250
Total	From/To	Desert	Urban	From/To	Nominal	Jamming
	Desert	550	7800	Nominal	500	1500
	Urban	6300	1050	Jamming	1400	8 00

Table 4 Summary Switching Cost Table

From/To	Initial	C1: Desert, Nominal	C2: Desert, Jamming	C3: Urban, Nominal	C4: Urban, Jamming	Final
Initial α	NA	200	350	350	500	NA
C1: Desert, Nominal	NA	1050	2050	6800	9300	250
C2: Desert, Jamming	NA	1950	1350	9200	8600	400
C3: Urban, Nominal	NA	6800	7800	1550	2550	400
C4: Urban, Jamming	NA	7700	7100	2450	1850	550
Final Ω	NA	NA	NA	NA	NA	NA



Test Configuration Optimization

Objective

Find a sequence of test configurations that minimizes switching cost of test configurations.

Minimize Total Cost $12,200 dollars

Configuration Switching Cost Matrix

Each cell in the transition cost matrix is the estimated total cost of switching from one configuration to a

This is the total of teardown, setup, run, and analyze cost for each factor in the configuration.

For any pair of configurations x and y, the switching cost x->y is not necessarily the same as that of y->:

	From\To	α-ω	Des-Nom	Des-Jam	Urb-Nom	Urb-Jam
1	α-ω	0	200	350	350	500
2	1 Des-Nom	250	1050	2050	6800	9300
3	2 Des-Jam	400	1950	1350	9200	8600
4	3 Urb-Nom	400	6800	7800	1550	2550
5	4 Urb-Jam	550	7700	7100	2450	1850
		1	2	3	4	5

Configuration Sequence Selections

A selected transition is indicated with a "1" and indicates that the TC of that row
is followed by the TC of that column. This is the output of the optimization model.

From\To	α-ω	Des-Nom	Des-Jam	Urb-Nom	Urb-Jam	Require row sur Each configurat	
α-ω	0	0	1	0	0	1	=
1 Des-Nom	0	0	0	1	0	1	=
2 Des-Jam	0	1	0	0	0	1	=
3 Urb-Nom	0	0	0	0	1	1	=
4 Urb-Jam	1	0	0	0	0	1	=
Sum:	1	1	1	1	1		
Must enter:	=	=	=	=	=		
	1	1	1	1	1		

Require column sum == 1: each configuration must be used exactly once

Tour Constraints

Do not allow partial or unconnected sequences. Aka Miller/Tucker/Zemlin subtour constraints.

Number of configurations 5

Assigned Step		α-ω	Des-Nom	Des-Jam	Urb-Nom	Urb-Jam
0	α-ω		=>=	=>=	>=	=>=
2	1 Des-Nom			=>=	=>=	>=
1	2 Des-Jam		=>=		>=	>=
3	3 Urb-Nom		=>=	>=		=>=
4	4 Urb-Jam		>=	=>=	=>=	

Optional Tightening constraints:

10	=		Step assignments must exactly correspond to number
3	<=	4	Kill symmetry if distance matrix symmetric

Fig. 1 Example IP Switching Cost Minimization Model in WhatsBest

III. CONCLUSIONS AND FUTURE WORK

This research began with an intuition that the total cost of a randomized test sequence typically used in combinational test design and design of experiments could probably be minimized with mathematical optimization.

The design of experiments and combinatorial algorithms that produce *t*-way test suites reviewed for this paper do not represent or use information about inter-test costs or constraints to determine the sequence of tests in a test suite. For any *t*-way test suite, this paper has presented an IP formulation that will produce a sequence of test cases that achieves the minimum total switching cost.

Although switching cost reduction models have been established for both combinational test design and design of experiments, they rely on either a rough cost approximation or heuristic greedy algorithms. Neither can guarantee global optimization. In contrast, IP produces global optimizations with well-understood properties. In addition, a solved IP supports sensitivity analysis, which may be used to easily and reliably investigate the effects of changes to cost factors and related constraints [26] [32].

Although the example is a toy problem, the model scales symmetrically to practical sizes without any structural changes. That is, a 50 test case or configuration model will require a 50x50 cost table and 50 sequence selection equations. The structure of small and large models will be the same, regardless of the testing or experimental design application. Tabulation of costs is a straightforward, if tedious, task.

On present-day computers, 100 node TSP problems are routinely solved in a few minutes. A comprehensive TSP performance benchmark study conducted in 2002 reports that 3000 node problems may be solved in about 20 hours on a

500 MHz Alpha processor [28]. Given the substantial improvement in compute power since then, it is clear that models with several thousand nodes can be solved in a time span that will accommodate a wide range of practically large applications.

The basic model in the paper suggests many interesting questions.

- Sub-plot design and analysis techniques may be used to mitigate the confounding effects of a non-random sequence on statistical validity of an experimental design. This trades a small loss in predictive precision for a potentially large reduction in testing cost. As this technique is well-established and there is no apparent difference in the statistical effect of blocking derived by IP cost optimization or grouping by heuristics, it seems that experimental designs can be both cost optimal and statistically valid. Development of a sufficient example is a future task.

- Practical testing considerations often require that some sequences are strictly required or prohibited. Safety is such a constraint. For example, as trace quantities of fuel *X* react explosively with fuel *Y*, model constraints should prevent fueling with *X* before *Y*. In the software space, using a production database with a certain access profile would expose personally identifiable information in subsequent tests, so this access profile must not follow a test case where the production database is instantiated.

- Do IP models that represent both switching time and switching cost provide any utility?

- To what extent may IP sensitivity analysis be used to evaluate the tradeoff among increasing *t*, reliability risk reduction, and total cost?

- Can the selection of a covering array be usefully modeled in IP along with the cost of transitions?

ACKNOWLEDGEMENT

This paper is dedicated to the memory of Alex Orden.

I thank Dave Zubrow, Julie Cohen, Bill Novak, Mike Phillips, and John Robert of the SEI for their support and insights.

The author presented the underlying concepts of this paper at the Software Solutions Symposium 2017, March 23, 2017, Arlington, Virginia.

This material is based upon work funded and supported by the U.S. Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. DM18-0067.

REFERENCES

[1] D. R. Kuhn, R. N. Kacker and Y. Lei, Introduction to Combinatorial Testing, Boca Raton: CRC Press, 2013.

[2] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin and B. Baudry, "Test them all: is it worth it? A ground truth comparison of configuration sampling strategies," *Empirical Software Engineering*, in press.

[3] S. Elbaum, G. Rothermel and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Hong Kong, 2014.

[4] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67-120, 2012.

[5] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal*, vol. 21, pp. 445-478, 2013.

[6] Y. Singh, "Systematic literature review on regression test prioritization techniques," *Informatica*, vol. 36, pp. 379-408, 2012.

[7] H. d. S. C. Junior, M. A. P. Araujo, J. M. N. David, R. Braga, F. Campos and V. Stroele, "Test case prioritization: a systematic review and mapping of the literature," in *SBES'17*, Fortaleza, 2017.

[8] J. Black, E. Melachrinoudis and D. Kaeli, "Bi-criteria models for all-uses test suite reduction," in *26th International Conference on Software Engineering*, Edinburgh, 2004.

[9] S. Alspaugh, K. R. Walcott, M. Belanich, G. M. Kapfhammer and M. L. Soffa, "Efficient time-aware prioritization with knapsack solvers," in *WEASELTech '07, 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, 2007.

[10] H.-Y. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *31st International Conference on Software Engineering*, Vancouver, 2009.

[11] L. Zhang, S.-S. Hou, C. Guo, T. Xie and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *International Symposium on Software Testing and Analysis*, Chicago, 2009.

[12] J. W. Lin, R. Jabbarvand, J. Garcia and S. Malek, "Nemo: multi-criteria test-suite minimization with integer nonlinear programming," in *40th International Conference on Software Engineering*, Gothenburg, 2018.

[13] D. M. Cohen, R. S. Dalal, M. L. Fredman and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions On Software Engineering*, vol. 23, no. 7, pp. 437-444, 1997.

[14] G. Sherwood, Email, 2018.

[15] D. R. Kuhn, J. M. Higdon, F. J. Lawrence, N. R. Kacker and Y. Lei, "Combinatorial methods for event sequence testing," in *IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montréal, 2012.

[16] P. M. Kruse and I. Schieferdecker, "Comparison of approaches to prioritized test generation for combinatorial interaction testing," in *Federated Conference on Computer Science and Information Systems*, Warsaw, 2012.

[17] S. Kimoto, T. Tsuchiya and T. Kikuno, "Pairwise testing in the presence of configuration change cost," in *Second International Conference on Secure System Integration and Reliability Improvement*, Yokohama, 2008.

[18] H. Srikanth, M. B. Cohen and Q. Xiao, "Reducing field failures in system configurable software: cost-based prioritization," in *International Symposium on Software Reliability Engineering*, Mysuru, 2009.

[19] R. C. Bryce, S. Sampath, J. B. Pedersen and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, June 2011.

[20] D. C. Montgomery, Design and Analysis of Experiments, 7th ed., New York: John Wiley & Sons, 2009.

[21] G. Hutto and J. Higdon, "Survey of design of experiments (DOE) projects in developmental test CY07-08," in *U.S. Air Force T&E Days 2009*, Albuquerque, 2009.

[22] D. R. Kuhn, D. R. Wallace and A. M. Gallo, "Software fault interactions and implications for sofftware testing," *IEEE Transactions on Software Engineering,* vol. 30, no. 6, pp. 418-421, 2004.

[23] J. Ganju and J. M. Lucas, "Randomized and random run order experiments," *Journal of Statistical Planning and Inference,* vol. 133, no. 1, pp. 119-210, 2005.

[24] B. Jones and C. J. Nachtsheim, "Split Plot Designs: What, Why, and How," *Journal of Quality Technology,* vol. 41, no. 4, pp. 340-346, October 2009.

[25] F. T. Anibari and J. M. Lucas, "Designing and Running Super-Efficient Experiments: Optimum Blocking with One Hard-to-Change Factor," *Journal of Quality Technology,* vol. 40, no. 1, pp. 31-45, 2008.

[26] L. A. Wolsey, Integer Programming, New York: Wiley, 1998.

[27] G. B. Dantzig, D. R. Fulkerson and S. M. Johnson, "Solution of a large scale traveling salesman problem," Technical Report P-510, RAND Corporation, Santa Monica, California, 1954.

[28] D. S. Johnson and L. A. McGeoch, "Experimental Analysis of Heuristics for the STSP," in *The Traveling Salesman Problem and Its Variations*, New York, Springer Science+Business Media, LLC, 2007, pp. 369-443.

[29] E. Klarreich, "Computer Scientists Find New Shortcuts For Infamous Traveling Salesman Problem," January 2013. [Online]. Available: https://www.wired.com/2013/01/traveling-salesman-problem/. [Accessed 18 January 2018].

[30] L. Schage, What'sBest! User's Manual, Version 15.0, Chicago, Illinois: LINDO Systems, Inc., 2017.

[31] M. Desrochers and G. Laporte, "Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints," *Operations Research Letters,* vol. 10, no. 1, pp. 27-36, 1991.

[32] M. Salimian, "LINDO Tutorial - 2: Sensitivity Analysis and Integer Programming," [Online]. Available: https://www.youtube.com/watch?v=muLLnPrRdf8. [Accessed 18 January 2018].