A Multi-Objective Genetic Algorithm for Location in Interaction Testing

Ryan E. Dougherty
ryan.dougherty@westpoint.edu
United States Military Academy
West Point, New York, USA

Dylan N. Green
dylan.green@westpoint.edu
United States Military Academy
West Point, New York, USA

Hyunmook Kang
hyunmook.kang@westpoint.edu
United States Military Academy
West Point, New York, USA

Grace M. Kim
grace.kim@westpoint.edu
United States Military Academy
West Point, New York, USA

Stephanie Forrest
steph@asu.edu
Arizona State University
Tempe, Arizona, USA

ABSTRACT

Software testing is a key component of the software engineering process, but modern software is highly complex. Software configurations involve many interacting components and interactions among them can strongly affect the software's behavior in hard-to-predict ways. Combinatorial interaction testing (CIT) concerns the creation of test suites that either detect or locate the most important interactions in a large scale software system. Locating Arrays (LAs) are a data structure that guarantees a unique location for every such set of interactions. In this paper we present LocAG, an algorithm that generates LAs. Our approach uses a simple but powerful "partitioning" method of interactions to greatly reduce the computational cost of verifying a candidate LA. Further, we use evolutionary computation to quickly determine any additional tests after the partitioning method is complete. We are able to generate LAs for larger systems faster, with any desired separation, and greater interaction size than any existing approach.

CCS CONCEPTS

• **Mathematics of computing → Combinatorial algorithms**; • **Software and its engineering → Search-based software engineering**.

KEYWORDS

locating arrays, combinatorial testing, genetic algorithm, evolutionary computation

ACM Reference Format:
Ryan E. Dougherty, Dylan N. Green, Hyunmook Kang, Grace M. Kim, and Stephanie Forrest. 2024. A Multi-Objective Genetic Algorithm for Location in Interaction Testing. In *Genetic and Evolutionary Computation Conference (GECCO '24 Companion), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3638530.3654260

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
GECCO '24 Companion, July 14–18, 2024, Melbourne, VIC, Australia
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0495-6/24/07.
https://doi.org/10.1145/3638530.3654260

1 INTRODUCTION

The analysis of large-scale software has always been challenging. It is imperative in software development to ensure correct operation, accomplished often with testing; with large systems, exhaustive testing is often infeasible. Combinatorial interaction testing (CIT) involves designing a test suite that covers many interactions of system components within the tests. Each test's execution produces a response from the system which can be checked against the desired output. CIT guarantees that if all specified tests pass, then any undesired behavior in the system is the result of more components than the size of the tested interactions. A limitation of this approach is that the tests do not guarantee *recovery* of which interactions caused a given response or which settings significantly impact the response.

Determining the fault(s) within a system is a well-studied problem. One strategy for CIT is to run a set of tests, note which tests passed and failed, and then test each possible interaction that appears within each failed test and estimate their likelihood of causing the fault(s). However, if two interactions appear in the same tests, then their calculated likelihood will be the same. If there is "noise" or nondeterminism within the testing environment, different runs of the same test may yield different responses. Locating arrays, introduced by Colbourn and McClary [2], are a CIT strategy that provides a guarantee that a set of interactions of at most a given size corresponds to a unique set of tests; this solves the first problem. Further, if redundancy is built into the locating array, the second problem can be avoided. Despite its general framework, the construction of locating arrays has only been well studied for recovering a single interaction, whether mathematically [2] or computationally [7, 8]. This restriction is due to the combinatorial cost involved of running the algorithms.

This paper presents an algorithm LocAG that uses an evolutionary algorithm to construct locating arrays much faster than previous methods. We can construct locating arrays far faster for larger and more complex systems than earlier approaches, including arrays that can locate two interactions for some systems. To the best of our knowledge, this work is the first to produce locating arrays that can locate two interactions for non-trivial systems.

2 PRELIMINARIES

We specify a formal testing model for CIT. Suppose a system has k components, and each has a set of categorical inputs. A *test* assigns

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{array} \qquad \begin{array}{cccc} 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{array}$$

Figure 1: (Left) A covering array. (Right) A $(\overline{1}, \overline{2}, 1)$-locating array.

to each component one of its inputs. A *t-way interaction* is a set of t pairs (C_i, x_i) where x_i is an input for component C_i. Let A be a collection of tests, T be a t-way interaction, and $\rho_{\mathsf{A}}(T)$ to be the set of tests in which T appears in A. There are two properties that A can have that we consider in this paper. For these properties, d and λ are given integers.

- (**Coverage Property**) For every t-way interaction T, if $\rho_{\mathsf{A}}(T)$ has size at least λ, then A has the *coverage property*, and is denoted a *covering array*.
- (**Locating Property**) Let $\mathcal{T}$ be a set of d t-way interactions, and $\rho_{\mathsf{A}}(\mathcal{T})$ the set of rows in A containing any interaction in $\mathcal{T}$. For every two $\mathcal{T}_1, \mathcal{T}_2$ collections of size at most d, containing interactions of size at most t, if the symmetric difference of $\rho_{\mathsf{A}}(\mathcal{T}_1)$ and $\rho_{\mathsf{A}}(\mathcal{T}_2)$ has size at least λ, then A has the *locating property*.

Informally, the coverage property holds if every interaction of t or less is tested by at least one row in the array, and the locating property holds if for any fault of t or fewer interactions, the array can unambiguously identify which components have failed.

If an array A satisfies the coverage property, it is a *covering array*; if in addition A satisfies the locating property, it is a $(\overline{d}, \overline{t}, \lambda)$-*locating array*. Figure 1 gives an example of a covering array with $N = 6$ tests, $k = 4$ components, 2 inputs per component, $t = 2$, and $\lambda = 1$ (left); and a locating array with $\overline{d} = 1, \overline{t} = 2, \lambda = 1$ (right). Here, λ represents how much *redundancy* we require. If there is experimental noise or measurement error in the system, then $\lambda = 1$ may not be sufficient to determine whether or not running the same test can yield different responses. Enforcing λ to be "large" decreases the likelihood of a response's being incorrect from the ground truth. For the locating property, when the difference-on-rows property holds for $\mathcal{T}_1$ and $\mathcal{T}_2$, we say that these two collections are a *locating pair*, and a *nonlocating pair* otherwise.

3 PROPOSED METHOD

LocAG has two stages, with the overall goal of creating a locating array. Recall that a locating array is required to have the coverage and locating properties. The first stage of LocAG generates an initial array and a list of nonlocating pairs. The second stage takes this list and uses a genetic algorithm to add rows that guarantee to locate these remaining pairs. Figure 2 shows the overall workflow of LocAG. Code is available at https://github.com/ryandougherty/LocatingArrayGenerator.

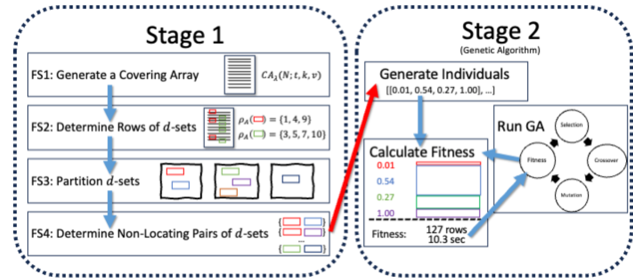

Figure 2: Overview of LocAG's workflow.

3.1 First Stage

Assuming that a locating array exists at all for the given system [2], the first stage has four sub-parts: (FS1) generates an initial covering array A; (FS2) determines the rows of interaction sets within A; (FS3) partitions these sets into appropriate groups; and (FS4) determines non-locating pairs of these sets based on this grouping.

In FS1, we create a covering array using the fast CAgen algorithm [16], which satisfies the covering property. In FS2, we determine the rows for all sets of interactions D of size at most d. For each interaction I in D, we determine $\rho_{\mathsf{A}}(I)$ and take the union all such sets to determine $\rho_{\mathsf{A}}(D)$.

In FS3, we calculate for each interaction set the number of rows in which it appears in A, and partition these sets based on their numbers. If the number of rows in any two sets in this partition differ by λ or more, then they automatically form a locating pair. The purpose of this partition is to avoid checking pairs of sets that are already locating, which greatly reduces computational cost.

FS4 determines all nonlocating pairs by comparing every two sets from FS3 that have their rows differing less than λ, and insert all nonlocating pairs found into a list. In practice, we can further improve performance by using sorted arrays instead of sets, because (1) computing set difference can be performed in linear time, and (2) one can terminate the comparison process if the second examined set appears in rows that are all strictly larger than those of the first.

3.2 Second Stage

We use a multi-objective, multi-stage evolutionary algorithm to locate the remaining pairs. The goal is to find a small number of tests that accomplish the goal with minimal computational cost.

Each individual's representation is a list of initially uniform-at-random real numbers between 0.0 and 1.0, with the last element of the list fixed to 1.0. For each number x within the list, we create a small number of tests that locates at least x percent of the remaining pairs. Once these tests are found, we update the remaining pairs.

Suppose that we need to locate x percent of remaining pairs. We use binary search to quickly find a number of tests that can locate an x percent of pairs. Suppose we want to determine if R tests are sufficient. We use a simulated annealing approach similar to Konishi et al. [8], but with random seed to ensure the fitness function is deterministic. If the process fails, R is increased and the EC tries again. Once a particular R succeeds, we iterate through *all* pairs to determine if they are locating in these R rows. The fitness function

outputs a pair (N, T) where N is the total number of rows generated and T is total wall clock time. The EC uses one-point crossover, non-dominated sorting [4], and simple mutation operators: uniformly-at-random choosing one of: (1) alter an existing number in the list, (2) add a new random number, or (3) delete a randomly chosen number.

4 EVALUATION OF LOCAG

Our experimental evaluation of LocAG centered on the following research questions:

- **RQ1**: How much faster does LocAG produce locating arrays than existing methods, if at all?
- **RQ2**: How much smaller are the locating arrays produced by LocAG compared to existing methods, if at all?

Our data set consists of the several real-world systems, described below: The numbers after each system describe how many components each system has (the exponents) and how many inputs each component has (the base). For example, the `Apache` system has 158 components with 2 inputs each, 8 components with 3 inputs each, etc.

- `Apache`: $2^{158}3^{8}4^{4}5^{1}6^{1}$
- `Bugzilla`: $2^{49}3^{1}4^{2}$
- `GCC`: $2^{189}3^{10}$
- `Mobile`: $2^{28}3^{9}4^{6}5^{4}6^{10}7^{5}8^{4}9^{1}10^{8}$
- `SPIN-S`: $2^{13}4^{5}$
- `SPIN-V`: $2^{42}3^{2}4^{11}$
- `Wireless`: $2^{3}3^{7}4^{5}5^{9}$

These examples were taken from [6, 8]. The initial covering array produced by CAgen often includes entries that are "don't cares," meaning that they can be set to any value and do not affect coverage. We ran LocAG by setting these entries uniformly-at-random five times and took the best results. For each of these, we ran LocAG on each systems with $\overline{d} = 1$, $\overline{t} = 2$, and $1 \leq \lambda \leq 4$ with a fixed choice of random selections to the "don't care" entries. The EC ran for 50 generations, had an individual list size between 10 and 30 initialized uniformly-at-random, 100 individuals in the population, used a crossover rate of 30% per pair, and mutation was set to 30% per individual.

We ran all experiments on a Windows laptop with an Intel Core i7 processor at 3.0GHz and 32GB of RAM using our `C++` implementation, and table 1 shows our results. It reports two pairs of numbers: the first–(LocAG, Min N)–contains the lowest number of rows found, within the Pareto frontier, together with the rutime time; the second column–(LocAG, Min Time)–reports results for the fastest run. Runtime is reported as total number of CPU seconds. The third column–(Best Known)–contains the best $(\overline{1}, \overline{2}, \lambda)$-locating array reported in the literature, with accompanying reference, and computational time if reported; the last column is the identical except for $t = 2$ instead of $\overline{t} = 2$. Although our own experiments only consider $\overline{t}$, which must involve more rows and computation time than for t, these results provide insight into LocAG's performance as there are few reported results for $\overline{t}$, and none with wall clock time for $\overline{t}$.

Although not shown in Table 1, LocAG can also generate locating arrays with $\overline{d} = 2$, $\overline{t} = 2$, and $k = 10$ components all under 2 hours of compute time. As far as we can tell this is the first algorithm that can generate small locating arrays able to locate two interactions in any "reasonable" amount of time.

RQ1: As is evident from Table 1, LocAG finds locating arrays far faster than any previously reported method, sometimes two orders of magnitude faster than even results that involve t (which is easier than $\overline{t}$). This effect is smaller for larger systems, but is still striking; most of computational time for these larger systems was spent determining nonlocating pairs in the first stage, since many pairs are already locating.

RQ2: Here, there are few previous results to which we can compare LocAGĊomparing to the previous results that do exist, LocAG improved three examples in terms of number of rows produced: `Mobile` and $\lambda = 2$, and `Wireless` for $\lambda = 1, 2$. Because LocAG relies on the CAgen algorithm to generate an initial covering array, and doesn't not modify existing entries from that array to ensure the coverage property, the larger number of rows observed here is not suprising. Despite this limitation, the number of rows at the end of LocAG is often competitive.

5 THREATS TO VALIDITY

These experimental datasets may not generalize to other systems. Further, the correctness of any implementation is always a concern. To determine the correctness of CAgen, we wrote an independent covering array verifier. Next, we ran the verifier on the initial covering arrays produced by CAgen, all of which passed verification. The algorithms for both algorithm stages were implemented in in both Python and C++. For the first stage and each system, we fixed a single covering array and verified that the resulting set of nonlocating pairs is the same for both implementations.

6 FUTURE WORK

Runtime for both stages varies widely depending on the system's profile, which was caused by systems with an unbalanced number of inputs leading to many "don't care" positions. In principle, one could add a third initial stage, where a greedy algorithm is used to find a better choice for the "don't care" entries than uniformly-at-random. Interestingly, if there is sufficient "imbalance" in the system's profile, then the initial covering array can be very close to being locating.

In the first stage of LocAG, the CAgen tool constructs covering arrays using the "in-parameter-order" algorithm [11]; our approach could adopt a similar algorithm for checking locating pairs. Additionally, one could add an intermediate "post-optimization" stage [13], which would take a locating array, rearrange rows, and randomize "don't care" values until a row is determined to be redundant.

7 RELATED WORK

We outline related works with regard to metaheuristic methods. For a general background of CIT, see [9, 14]. Nagamoto et al. [12] developed a two-stage method for locating array generation. The first stage of LocAG is a significant extension of their work based on the partitioning strategy. Lanus et al. [10] created a tree-based verification algorithm for locating arrays and was the first paper (as far as we are aware) to computationally create locating arrays with both $d > 1$; however, they only concerned *main effects* ($t = 1$) due to

Table 1: Results from LocAG for generating $(\overline{d}, \overline{t}, \lambda)$-locating arrays on all tested systems with $\overline{d} = 1$, $\overline{t} = 2$.

Name	Profile	λ	LocAG, Min N	LocAG, Min Time	Best Known $(\overline{1}, \overline{2}, \lambda)$	Best Known $(\overline{1}, 2, \lambda)$
Apache	$2^{158}3^8 4^4 5^1 6^1$	1	87, 12.2	91, 12.1		63, 2062.5 [8]
Bugzilla	$2^{49}3^1 4^2$	1	43, 0.6	44, 0.4		32, 82.4 [8]
GCC	$2^{189}3^{10}$	1	52, 11.8	56, 10.7		39, 1581.3 [8]
Mobile	$2^{28}3^9 4^6 5^4 6^{10} 7^5 8^4 9^1 10^8$	1	439, 18.5	467, 13.6	421 [1]	333, 3055.7 [8]
		2	**484**, 11.2	**499**, 7.5	654 [15]	
SPIN-S	$2^{13}4^5$	1	44, 0.2	55, 0.1		34, 24.6 [8]
SPIN-V	$2^{42}3^2 4^{11}$	1	66, 0.3	74, 0.2		50, 234.5 [8]
Wireless	$2^3 3^7 4^5 5^9$	1	**99**, 1.6	**105**, 0.9	109 [3]	74, 113.9 [8]
		2	**135**, 0.5	**141**, 0.3	144 [15]	
		3	180, 0.7	191, 0.5	169 [15]	
		4	221, 1.6	234, 0.6	194 [15]	

computational cost. There may potentially be some optimizations in our approach based on theirs. Dougherty [5] partitioned covering array generation into multiple stages (instead of just 1 or 2) using a genetic algorithm. Instead of using the algorithm to modify the arrays directly, the genetic algorithm helped determine the optimal number of stages in which to have every interaction covered λ times. In principle his methods can also be applied to locating array generation. As far as we are aware, only Konishi et al. [8] has used any metaheuristic for finding locating arrays, namely simulated annealing. LocAG is able to generate locating arrays on the same parameters far faster than their approach with slightly more rows.

8 CONCLUSION

This paper introduces LocAG, a two-stage algorithm for constructing locating arrays that uses a multi-objective evolutionary algorithm in the search. Our strategy relied on the empirical observation that in nearly every covering array with redundancy λ, there are many interactions that are covered more than λ times. If there is a sufficient difference between the number of times two sets of interactions are covered, then they automatically become locating pairs. Our partitioning step drastically reduces the number of pairs to be compared, and our results show that the number of pairs is much smaller than the total number of pairs, facilitating the use of an evolutionary algorithm to locate the remaining pairs. This approach allowed us to construct locating arrays for several previously studied systems much more quickly than any other approach.

ACKNOWLEDGMENTS

The opinions in the work are solely of the authors, and do not necessarily reflect those of the U.S. Army, U.S. Army Research Labs, the U.S. Military Academy, or the Department of Defense. SF gratefully acknowledges the partial support of NSF (CCF CCF2211750, CICI 2115075), DARPA (N6600120C4020, N6600122C4026), ARPA-H (SP4701-23-C-0074), and the Santa Fe Institute.

REFERENCES

[1] Abraham N. Aldaco, Charles J. Colbourn, and Violet R. Syrotiuk. 2015. Locating Arrays: A New Experimental Design for Screening Complex Engineered Systems. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 31–40. Place: New York, NY, USA Publisher: ACM.

[2] Charles J. Colbourn and Daniel W. McClary. 2008. Locating and detecting arrays for interaction faults. *Journal of Combinatorial Optimization* 15, 1 (Jan. 2008), 17–48. https://doi.org/10.1007/s10878-007-9082-4

[3] Randy Compton, Michael T. Mehari, Charles J. Colbourn, Eli De Poorter, and Violet R. Syrotiuk. 2016. Screening interacting factors in a wireless network testbed using locating arrays. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, San Francisco, CA, USA, 650–655. https://doi.org/10.1109/INFCOMW.2016.7562157

[4] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017

[5] Ryan E. Dougherty. 2020. Genetic algorithms for redundancy in interaction testing. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. Association for Computing Machinery, New York, NY, USA, 1241–1249. https://doi.org/10.1145/3377930.3390230

[6] Ryan E. Dougherty, Kristoffer Kleine, Michael Wagner, Charles J. Colbourn, and Dimitris E. Simos. 2022. Algorithmic methods for covering arrays of higher index. *Journal of Combinatorial Optimization* 45, 1 (Dec. 2022), 28. https://doi.org/10.1007/s10878-022-00947-x

[7] Tatsuya Konishi, Hideharu Kojima, Hiroyuki Nakagawa, and Tatsuhiro Tsuchiya. 2020. Finding Minimum Locating Arrays Using a CSP Solver. *Fundamenta Informaticae* 174, 1 (Jan. 2020), 27–42. https://doi.org/10.3233/FI-2020-1929 Publisher: IOS Press.

[8] Tatsuya Konishi, Hideharu Kojima, Hiroyuki Nakagawa, and Tatsuhiro Tsuchiya. 2020. Using simulated annealing for locating array construction. *Information and Software Technology* 126 (Oct. 2020), 106346. https://doi.org/10.1016/j.infsof.2020.106346

[9] D. RIchard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing*. Taylor & Francis, Boca Raton, FL.

[10] Erin Lanus, Charles J. Colbourn, and Douglas C. Montgomery. 2019. Partitioned Search with Column Resampling for Locating Array Construction. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Xi'an, China, 214–223. https://doi.org/10.1109/ICSTW.2019.00056

[11] Yu Lei and Kuo-Chung Tai. 1998. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium*. IEEE, Washington, DC, USA, 254–261. https://doi.org/10.5555/645432.652389

[12] Takahiro Nagamoto, Hideharu Kojima, Hiroyuki Nakagawa, and Tatsuhiro Tsuchiya. 2014. Locating a Faulty Interaction in Pair-wise Testing. In *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*. IEEE, Singapore, 155–156. https://doi.org/10.1109/PRDC.2014.26

[13] Peyman Nayeri, Charles J. Colbourn, and Goran Konjevod. 2013. Randomized post-optimization of covering arrays. *European Journal of Combinatorics* 34, 1 (Jan. 2013), 91–103. https://doi.org/10.1016/j.ejc.2012.07.017

[14] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *ACM Comput. Surv.* 43 (2011), 1–29. Publisher: ACM.

[15] Stephen A. Seidel, Kaushik Sarkar, Charles J. Colbourn, and Violet R. Syrotiuk. 2018. Separating Interaction Effects Using Locating and Detecting Arrays. In *Combinatorial Algorithms (Lecture Notes in Computer Science)*, Costas Iliopoulos, Hon Wai Leong, and Wing-Kin Sung (Eds.). Springer International Publishing, Cham, 349–360. https://doi.org/10.1007/978-3-319-94667-2_29

[16] Michael Wagner, Kristoffer Kleine, Dimitris E. Simos, Rick Kuhn, and Raghu Kacker. 2020. CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Porto, Portugal, 191–200. https://doi.org/10.1109/ICSTW50294.2020.00041