FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Efficient Algorithms and Tools for Practical Combinatorial Testing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Kristoffer Kleine, BSc.
Matrikelnummer 01226240

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Lektor Dr. Dimitris E. Simos
Mitwirkung: Privatdoz. Dipl.-Ing. Mag. Dr. Edgar Weippl

Wien, 23. Mai 2018

Kristoffer Kleine   Dimitris E. Simos

Efficient Algorithms and Tools for Practical Combinatorial Testing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Kristoffer Kleine, BSc.
Registration Number 01226240

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Lektor Dr. Dimitris E. Simos
Assistance: Privatdoz. Dipl.-Ing. Mag. Dr. Edgar Weippl

Vienna, 23rd May, 2018

Kristoffer Kleine   Dimitris E. Simos

Erklärung zur Verfassung der Arbeit

Kristoffer Kleine, BSc.
Lindenstraße 3/5, 3071 Böheimkirchen

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Mai 2018

Kristoffer Kleine

Danksagung

Mein größter Dank geht an meine Eltern, Karen-Bodil und Michael, für deren immer präsente Unterstützing. Ohne sie wäre nichts von dem möglich gewesen. Ich war die ganze Studienzeit in einer sehr priviligierten Position, welche es mir erlaubte sorgenfrei und konzentriert zu studieren. Dafür werde ich auf immer dankbar sein.

Dimitris Simos verdanke ich meine ganze bisherige wissenschaftliche Karriere. Sein exzellentes Mentoring und seine Supervision während meiner Transition zu meinem derzeitigen Forschungsfeld hat ein neues Interesse entfacht mich mit neunen, coolen und interessanten "Sachen" zu beschäftigen.

Weiters möchte ich meinen großartigen Kollegen Bernhard Garn, Ludwig Kampel und Manuel Leithner für die vielen aufschlussreichen Diskussionen, immer präsente Hilfsbereitschaft und Freundlichket bedanken.

Schlussendlich möchte ich mich auch bei allen ehemaligen und derzeitigen Kollegen bei SBA Research bedanken neben denen ich die Freude hatte zu arbeiten. Insbesondere bei Manuel Wiesinger und Sebastian Neuner für die vielen ausgedehnten Mittage und interessanten (mehr oder weniger arbeitsrelevanten) Diskussionen.

Acknowledgements

My deepest gratitude goes towards my parents, Karen-Bodil and Michael, for their life-long support and presence. Without them, none of what I have achieved would have been possible. Due to them I was in a very privileged position which allowed me to study care-free and to that I am eternally thankful.

To Dimtris Simos I owe my entire scientific career so far. The excellent mentoring and supervision of the transition to my current research topics have been crucial in sparking fresh curiosity in pursuing research into all kinds of new, cool and yet unknown “stuff”.

Furthermore, I want to thank my great colleagues Bernhard Garn, Ludwig Kampel and Manuel Leithner for many insightful discussions and always being helpful and also nice.

Finally, I would like to thank former and present colleagues at SBA Research with whom I have had the pleasure of working beside, most notably Manuel Wiesinger and Sebastian Neuner for many extended lunches and interesting (more or less on-topic) discussions.

Kurzfassung

Kombinatorisches Testen ist eine effektive Teststrategie, welche zunehmend im Softwaretesting Bereich Einzug findet. Als eine Black-Box-Test Technik geht es darum, Testfälle zu erzeugen die in der Lage sind, eine mathematische Garantie der Abdeckung des Eingaberaumes bereitzustellen während gleichzeitig die Anzahl der Testfälle niedrig bleibt.

Es wird angenommen, dass die Erzeugung optimaler kombinatorischer Testsätze ein hartes Optimierungsproblem darstellt. Das hat die Entwicklung von mehreren "Greedy" Algorithmen angespornt, die Optimalität für Geschwindigkeit eintauschen. Einer der beliebtesten Vertreter davon ist die In-Parameter-Order Familie von Algorithmen. Diese Klasse von Algorithmen hat eine breite Akzeptanz im Bereich von Software-, Hardware- und Sicherheitstesting gefunden, da sie Testsätze für die meisten Instanzen erstellen können, die in der Praxis auftreten.

In dieser Arbeit wird ein effizientes Design der In-Parameter-Order Algorithmen vorgestellt, welches die Zeit und den erforderlichen Speicher drastisch reduziert, um kombinatorische Testsätze zu berechnen Das verbesserte theoretische Design wird in einem Prototypen implementiert und einem Benchmark unterzogen. Außerdem werden die Algorithmen in einem universellen Test Tool integriert, das für diese Arbeit entwickelt wird und offen und frei für jedermann zugänglich ist. Die Arbeit untersucht weiters auch die Auswirkungen von Tie-Breaking, Parameterreihenfolge und Tuplenumerationsreihenfolge auf die resultierenden Testsätze. Eine umfangreiche experimentelle Studie zeigt, dass verschiedene Tie-Breaker im Durchschnitt keine signifikanten Auswirkungen haben und die Ordnung von Parametern in absteigender Domänengröße zu drastischen Verbesserungen führen können.

Abstract

Combinatorial Testing (CT) is an effective testing strategy which has seen increased adoption in the field of software testing. As a black-box testing technique it concerns itself with generating input test cases able to provide mathematical input-space coverage guarantees while keeping the number of test cases low.

Generating optimal combinatorial test sets is believed to be a hard optimization problem which has spurred the development of several greedy algorithms which trade-off optimality for speed. One of the most popular representatives is the In-Parameter-Order family of algorithms. This class of algorithms has seen a wide adoption in including software, hardware and security testing as they are able to construct test sets for most instances occurring in practice.

In this thesis, an efficient design of the In-Parameter-Order algorithms is presented which drastically reduces the time and required memory to compute combinatorial test sets. The improved theoretical design is implemented in a prototype and benchmarked against the state of the art implementation. Furthermore, the algorithms are made available in a general purpose test generation tool developed for this thesis which is open and free to use by anyone. The thesis also studies the impact of tie-breaking, parameter ordering and tuple enumeration order on the resulting test sets. An extensive experimental study shows that different tie-breakers have no significant impact on average and that ordering parameters in order of descending domain size can lead to drastically smaller test sets.

Publications arisen from this Thesis

During the work conducted as part of this Master Thesis, the following scientific publications have been published in the field of Algorithms for Combinatorial Testing:

1. Kristoffer Kleine and Dimitris E. Simos, "*An efficient Design and Implementation of the In-Parameter-Order Algorithm*", in **Mathematics in Computer Science**, 12(1), pp. 51-67, 2018.
2. Kristoffer Kleine, Ilias Kotsireas, and Dimitris E. Simos, "*Evaluation of Tie-breaking and Parameter ordering for the IPO Family of Algorithms used in Covering Array Generation*", in **IWOCA 2018: 29th International Workshop on Combinational Algorithms**, to appear.

Contents

CHAPTER 1

Introduction

As technology continues to advance and the number of connected devices keeps growing at a rapid pace [Gar17], so too does the potential for errors in these systems and their software. A study conducted about the state of the US software industry [Tas02] from 2002 estimates that faults in software incur a cost of tens of billions of dollars (about one percent of the country's GDP). Since then, as software has become even more ubiquitous and its complexity has further increased, this cost has grown as well: in a recent study by Tricentis, the estimated sum of damages caused by high-impact software faults is reported to be at least 1.7 trillion USD [Tri18]. This enormous economic incentive has driven research to provide better means and methods to adequately test these systems.

Many different software testing strategies have been devised through the years [ND12] and they can roughly be categorized into white-box and black-box testing techniques. The former, sometimes also called structural testing, bases its testing on the available source code and aims to cover data and control flows to ensure a sufficient level of confidence in the quality of the tested software. Black-box testing on the other hand considers the test subject to have no internal structure and just exercises inputs to the system and observes its output to determine whether it behaves correctly.

Black-box testing is thus mostly concerned with modelling the input space of the program and deciding if a test case is considered passing or failing based on the systems behavior or output. The basis for all black-box testing approaches is a suitable input parameter model (IPM) [GO07]. Such a model discretizes the input space of the system into separate parameters and associated values. A test case then constitutes a selection of one value for each parameter from that parameter's domain.

One of the main challenges of black-box testing is to derive an effective test set or test suite. Effective in this instances means to generate test cases which exercise a wide variety of system states and provide a satisfying coverage of all possible behaviors. The simplest test generation method is to generate an exhaustive test set, i.e., test all possible combinations of parameters and their values. However, the number of test cases grows exponentially with the number of parameters and this method quickly becomes infeasible in practice.

A simple, yet effective test generation scheme is known as random testing [DN84]. Values for each parameter in a test case are randomly drawn from the parameters domain. The domain can either be modelled by the software tester to include domain knowledge (e.g., values from real-world data or exemplary invalid values) or be only constrained by the parameters type (e.g., a signed integer parameter can take values between -2^{31} and $2^{31}-1$). Adaptive random testing (ART) [CKMT10] extends random testing by exploring a more diverse set of test cases by constructing new tests based on their distance (as defined by some metric) to previously executed test cases. Anti-random testing [Mal95] constructs test sets such that each new test has the maximum distance to all previous tests. Each-choice testing [GOA05] ensures that each value of each parameter is used in at least one test. Base-choice testing [AO94] first defines a *base test* which contains default values. These default values are selected by the tester and can be interpreted as values which result in a normal (expected) system behavior. Further test cases then vary the value of one parameter while keeping the values of the base test for the other parameters. Failures in a test case can then most likely be linked to the parameter which diverges from the base test.

In recent years, combinatorial testing (CT) [KKL13] has emerged as a black-box testing strategy which can provide mathematical input-space coverage guarantees while keeping the number of tests low. At its base lies the observation that only a limited number of parameters of a given system (e.g., a software system) are responsible for triggering a fault in that system [KWG04]. This means, that if only the interaction of t out of k ($t << k$) parameters cause a failure, then a test set which covers all t-way interactions has the same fault detection capabilities as an exhaustive test set. Furthermore, such test sets, called combinatorial test sets or t-way test sets, increase in size only logarithmically in relation to the number of parameters.

Consider the artificial example given in Listing 1.1. In this case, a fault is only triggered by the interaction of the two parameters *month* and *is_leap_year*. Other parameters of the system are not relevant for this failure and exercising those will not provide any testing benefit.

```
if month == "feb" && is_leap_year {
        // fault
}
```

Listing 1.1: Fault caused by the interaction of two parameters

Faults caused by the interaction of t parameters are called t-way interaction faults. In the case of $t = 2$, this testing strategy is often called “pair-wise”, “all-pairs” or “2-way interaction” testing.

Browser	Adblocker	Transport-Security
Firefox	true	None
Firefox	false	TLS 1.2
Firefox	true	TLS 1.3
Chrome	false	None
Chrome	true	TLS 1.2
Chrome	false	TLS 1.3
Edge	true	None
Edge	false	TLS 1.2
Edge	true	TLS 1.3
Safari	true	None
Safari	false	TLS 1.2
Safari	false	TLS 1.3

Table 1.1: Pair-wise (2-way) test set for an end-to-end test of a website

Consider now an end-to-end test of a website. This test requires the tester to manually access the site via a browser and perform some pre-defined test and verify that the website works as expected. This kind of testing is often necessary to make sure that the whole system acts as intended and is hard to automatize since many non-functional observations (performance, usability, responsiveness, etc.) are needed to judge the functionality of the tested system. This makes this kind of test very expensive to perform. In addition, it is desirable to test as many system configurations as possible, at the very least test some configurations which are used by a significant fraction of the actual user base. In this fictitious example, such a configuration might include the browser (Firefox, Chrome, Edge or Safari), whether an ad-blocker is being used and if the website is accessed over a secure transport link. From this model, a combinatorial test set can be constructed which consists of 12 tests. See Table 1.1 for a listing of such a test set. This is a significant reduction from the exhaustive test set (i.e., testing all combinations of parameters) which would contain 24 tests. In practice, the reduction in the number of tests can be in the orders of magnitude [KSTJV15].

Motivation

Combinatorial test sets can lead to a significant test set size reduction which decreases the time needed to test a system since fewer tests need to be executed. Research into combinatorial test set generation methods (see Section 2.2) has mainly focused on optimizing the size of produced test sets. Having fewer tests to execute will reduce the duration of the overall testing cycle. However, more often than not, test execution is fast enough to not impact overall testing time very much. In these cases, reducing the test set by a couple of tests will barely be noticeable during testing. Instead, optimizing test sets can be very costly as test generation time can become the dominating factor [KSTJV15], [KS17].

The In-Parameter-Order (IPO) family of algorithms, often referred to as IPO strategy, is a class of algorithms in the field of combinatorial testing which yield combinatorial test sets whose sizes are competitive with most other approaches. They furthermore exhibit comparable or lower test set generation times. Research on algorithms in the IPO family has mainly focused on minimizing the generated test set sizes, but there has been a distinct lack of work towards on improving test generation efficiency itself.

In this thesis, we describe existing efforts to implement the IPO strategy and propose significant improvements to the state of the art. These improvements do not influence the resulting combinatorial test sets while yielding major performance improvements. These improvements can enable testers to lower total time of testing without sacrificing benefits that combinatorial testing provides.

Structure of the Work

This thesis is structured as follows: Chapter 2 will give preliminary definitions and mathematical notation used in the remainder of the thesis. Furthermore, an in-depth overview and survey of related literature in the field of combinatorial test generation algorithms as well as a detailed introduction to the In-Parameter-Order family of algorithms will be given. Chapter 3 describes the implementation aspects of IPO in general and will propose novel optimizations to the algorithm. Their effectiveness will be evaluated in an experimental setting. In Chapter 4, design choices of IPO will be discussed and their influence on the resulting test sets will be evaluated. Chapter 5 presents a new tool for combinatorial test case generation based on the optimized algorithms described in this thesis. Its architecture and design will evaluated in Chapter 6. Finally, the thesis is concluded in Chapter 7.

Part I

The In-Parameter-Order Algorithm

CHAPTER 2

Background

The main property of combinatorial test sets, i.e., providing at least one test for each t-way interaction of parameters, can be translated to the world of mathematics via a structure called *covering array*. Each combinatorial test set represents the translation of a covering array to a useful software testing artifact. One and the same covering array can be used in multiple testing scenarios, if the systems have the same number of parameters and values. Thus, it makes sense to abstract this translation away for the purpose of generating the arrays. In the following chapters, we will mostly concern ourselves with covering arrays directly with the knowledge that a combinatorial test set can easily be derived via a simple translation.

2.1 Preliminaries

A covering array (CA) is a mathematical object defined by four positive integers and denoted as $CA(N; t, k, v)$. It is an $N \times k$ matrix where N is the number of rows, k the number of columns (often referred to as parameters), t the size of interactions that are covered and v is the size of the alphabet. The entries of the array are integers from the set $V = \{0, \ldots, v-1\}$. A covering array is defined by its *t-covering property*:

Definition (t-covering property)**.** *For any t-selection of columns, all v^t t-tuples between the selected columns occur at least once in the array.*

The parameter t is also called the strength of the CA. Table 2.1 shows an example of a $CA(4; 2, 3, 2)$ on the left. On the right it can be seen that all 2-tuples are present in the array.

A mixed-level covering array (MCA) is a generalization of a CA where each column i has its own alphabet size v_i. An MCA is denoted as $MCA(N; t, k, (v_0, \ldots, v_{k-1}))$. The alphabet for column i is defined as $V_i = \{0, \ldots, v_i - 1\}$. The tuple $(t, k, (v_0, \ldots, v_{k-1}))$

a	b	c	(a, b)	(b, c)	(a, c)
0	0	0	(0, 0)	(0, 0)	(0, 0)
0	1	1	(0, 1)	(1, 1)	(0, 1)
1	0	1	(1, 0)	(0, 1)	(1, 1)
1	1	0	(1, 1)	(1, 0)	(1, 0)

Table 2.1: $CA(4; 2, 3, 2)$ (left) covering all pairs of columns (right)

is referred to as the configuration of an MCA. Note that $t \leqslant k$ and $v_i > 0,\ 0 \leqslant i < k$. From this point forward, we will use the term CA and MCA interchangeably, if the distinction is clear from the context.

For a given configuration $(t, k, (v_0, \ldots, v_{k-1}))$ there always exists a covering array: the cross-product between the domains of all k columns is trivially a covering array. Furthermore, there exists an infinite number of covering arrays for any given configuration as any MCA can be extended with additional rows without destroying its t-covering property.

Definition (Optimal Mixed-level Covering Array)**.** *For a configuration $(t, k, (v_0, \ldots, v_{k-1}))$, an optimal MCA is an $MCA(N; t, k, (v_0, \ldots, v_{k-1}))$ which has the minimal number of rows N. The minimum N is denoted as $MCAN(t, k, (v_0, \ldots, v_{k-1}))$.*

It has been shown that the size of an optimal covering array has an upper bound which grows logarithmically in the number of parameters k [Col04]. This is one of the main benefits that combinatorial testing offers over exhaustive testing where test set sizes grow exponentially in k.

2.2 Covering Array Generation Methods

The general problem of constructing optimal covering arrays is believed to be a hard combinatorial optimization problem [Har05],[HR04] as it is tightly coupled with NP-hard problems [Che07]. As a result, there has been a lot of effort on developing and improving algorithmic approaches for covering array generation. This section will give an overview of these methods. The categorization is based on a survey [TJIM13] which studies different algorithms for CA construction.

Exact Methods

Algorithms knows as exact methods are able to find optimal solutions for a given covering array configuration. They usually involve a specification of the search space together with an algorithm to fully explore it. This naturally results in the discovery of the optimal solution. Since the search space can be very large, these methods are only suitable for small instances.

In [HPS$^+$04], the authors present a constraint programming (CP) encoding which models the values of each entry in the array (for a fixed N). If a CA for the given N exists, the solver will return a solution. Solving a series of such problems with decreasing N can determine the $MCAN$ for that instance. This work introduces several modelling techniques able to handle MCAs, partial coverage and user-specified constraints. Symmetry-breaking constraints as well as SAT-encodings of the CP model are presented and evaluated. [YZ06] base their model on this SAT-encoding and apply a tool to further break symmetries in the SAT clauses. Additionally, they present an exhaustive backtracking search technique. They also model the array as an $N \times k$ matrix using constraints and apply several techniques such as symmetry-breaking, tree-search pruning and constraint propagation to improve the performance. A branch-and-bound technique for binary ($v = 2$) covering arrays is presented in [BRTJRT09]. It also restricts the search to non-isomorphic (i.e., asymmetric) covering arrays to significantly reduce the search space. This algorithm is constructive and builds the CA column-by-column.

Further SAT encoding strategies are presented in [LETJRTRV08] and [BMTI10].

Greedy Methods

Greedy methods for constructing covering arrays are among the most popular algorithms in practice. Due to their heuristic nature, they provide no guarantee for finding optimal CAs, but their predominant use in practice suggests that the produced quality of test sets is sufficient in most cases. The greedy methods can roughly be divided into *one-test-at-a-time* and *one-column-at-a-time* algorithms.

One of the first *one-test-at-a-time* algorithms is the Automatic Efficient Test Generator (`AETG`) [CDKP94],[CDFP97]. An important result of this work is a proof that there exists a greedy algorithm which can construct covering arrays with asymptotic upper bounds on the number of rows N. In particular, it was shown that N grows at most logarithmically in k (number of columns) and quadratically in v. The results were shown for strength $t = 2$, however they have been generalized to arbitrary strengths in [BC09]. In [KS] this proof is generalized to mixed-level covering arrays. This algorithm constructs the covering array one row at a time. In each step, the new row is chosen such that it covers the most new and so far uncovered tuples. However, the problem of finding such an optimal row has been shown to be NP-complete [Col04]. In practice, this search quickly becomes infeasible, since the number of possible rows is equal to v^k. Instead, a candidate set of M rows is generated and best one (i.e., one which covers the most new tuples) is chosen. The candidate rows are generated by a randomized greedy algorithm which selects entries for each position in the row such that the greatest number of new tuples is covered.

One drawback of `AETG` is that it does not provide a logarithmic upper bound on the test set size (for increasing k) since the algorithm does not make optimal decisions in each extension step. Furthermore, the algorithm is randomized which results in a non-reproducible test set generation.

The deterministic density algorithm (`DDA`) [BC07],[BC09] is a deterministic *one-test-at-a-time* algorithm which is able to provide a logarithmic worst-case guarantee on the number of rows generated. The main result of these works is that it is not necessary to find a new row which covers the most new tuples. Instead, it suffices to find a new row which covers the average number of uncovered tuples. Finding such a test can be implemented by computing local density values for each column which are then used to select a value for each entry in the row. This process is completely deterministic as it requires no randomization.

The TCG (Test Case Generator) [TA00] algorithm works similar to `AETG` but uses a deterministic approach to generate candidate tests.

In [BCC05], the authors propose a general framework for *one-test-at-a-time* greedy construction algorithms. Concrete ways to instantiate the framework to obtain generalized versions of greedy *one-test-at-a-time* algorithms are discussed and the influence of the framework's parameters on the resulting performance is analyzed.

The In-Parameter-Order (IPO) strategy is an *one-column-at-a-time* greedy construction algorithm and was first proposed in [LT98]. An initial array of the first t columns is created and subsequently extended by adding the remaining columns. Each extension is divided into a horizontal and an optional vertical extension. We discuss the algorithm in greater detail in Section 2.3.

While the original algorithm was limited to arrays of strength 2 (pair-wise), subsequent works have generalized the algorithm to allow the generation of higher strength arrays [LKK$^+$07] (`IPOG`) as well as integrated constraint handling in [YLN$^+$13] and [YDL$^+$15]. In [FLL$^+$08], the authors propose variants of the IPO strategy, namely `IPOG-F`, `IPOG-F2`, by extending the search space in the horizontal extension. In [LKK$^+$08], the `IPOG-D` variant is presented which includes a recursive construction method aimed at reducing the number of combinations to be enumerated.

In [CG09], the IPO strategy is enhanced with a *coverage inheritance* step which copies an existing column into the place of a newly added column. This ensures coverage with all preceding but one column. The missing tuples are then added to complete the covering array.

Many works have been dedicated to improving parts of the IPO algorithms in order to minimize covering arrays sizes. In [DLY$^+$15] a graph-coloring scheme integrated into the vertical extension is proposed to reduce the resulting array sizes. [YZ11] modify `IPOG` with additional optimizations aimed at reducing *don't-care* values in order to minimize the number of rows. [GLD$^+$14] and [GLD$^+$15] discuss and evaluate the impact of tie-breaking on the generated arrays and propose a new tie-breaker which reduces the generated array sizes.

TConfig [Wil00] constructs larger covering arrays from small seed covering arrays by concatenation.

Metaheuristic Methods

Metaheuristic algorithms for covering array construction do, similarly to greedy methods, not provide a guarantee to find an optimal solution, but they often are able to find approximate solutions close to the optimum.

In his Master's Thesis [Sta01], Stardom presents Simulated Annealing (SA), Tabu Search (TS) and Genetic Algorithm (GA) methods for constructing covering arrays. Fundamental to all of them is a built-in binary search for the minimum N of rows for which a feasible solution can be found. Initially, the algorithms start with a given upper bound (available through theory or some other means) and the known lower bound (in this case v^2) and choose N to be the midway point between these two. The binary search executes the Metaheuristic and adapts the bounds based on the result. The procedure terminates as soon as both bounds converge.

In [TJRT12], the authors improve the simulated annealing approach by using an efficient method to construct initial solution whose columns contain a balanced distribution of values as well as a composed neighborhood function. Three different parallelization techniques for SA are explored by [AGTJH12b]. First, an independent search runs the SA algorithm in parallel starting from the same initial solution. At the end, the best among all local optima is selected. Second, a semi-independent approach exchanges local information across parallel runs, such that all runs collectively converge towards a better solution. Lastly, a cooperative search uses asynchronous communication to access the global search state which can reduce idle times between runs. These parallelization techniques are further studied by [AGTJH12a]. Several improvements to covering array construction using SA are presented in [GCD09] and [GCD11].

In [Nur04], the author presents a tabu search approach which starts with a random array and uses the number of uncovered t-tuples as the cost function. Moves are defined as a single change to the array which covers an uncovered tuple. The procedure terminates when a zero-cost solution is found. In that case, the search may be started again, starting from a smaller array in order to achieve an optimization of the number of rows. [GHRVTJ12] report different ways to adapt the tabu search procedure such as initial solution selection, tabu list length and different neighborhood functions and evaluate them in a case study.

Algebraic Methods

Algebraic methods use mathematical properties of certain sub-classes of covering arrays or other mathematical structures such as groups or finite fields to construct covering arrays. Usually these are direct constructions which have the drawback to only exist for a certain limited number of configurations. Some constructions can also lead to suboptimal array sizes.

A few examples of such constructions include Roux-type constructions [Slo93],[CK02] which use small covering arrays as seeds to construct larger covering arrays and Bush's construction [Bus52] for configurations where v is a prime power and $k = v+1$. In [GS17], the authors present an algebraic modelling technique for binary covering arrays which relies on linear and commutative algebra.

2.3 IPO Family of Algorithms

The In-Parameter-Order (IPO) strategy is a family of algorithms for generating covering arrays. It was first introduced by [LT98] for generating covering arrays of strength 2 (pair-wise). The basic strategy is depicted in Algorithm 1.

The only input for the algorithm is the configuration of the desired array, i.e., a triple $(t, k, (v_0, \ldots, v_{k-1}))$, where t denotes the strength, k the number of columns and $(v_0, \ldots, v_{k-1})$ represents the alphabet size of each column. Let the alphabet for each column be defined as $V_i = \{0, \ldots, v_i - 1\}$. In the case of CAs all columns share the same alphabet in which case the alphabet size is referred to as just v.

The algorithm incrementally constructs a covering array by extending an intermediate covering array for a subset of columns (which was obtained in a previous iteration of the algorithm) with a new column. This process is repeated until every column has been added. The initial covering array is obtained by computing the cross-product of the first t columns which is a trivial covering array.

Extending an array by a new column i works by enumerating all t-tuples where one entry is for the new column i. At first, all tuples are marked as uncovered. In two separate extension steps, these tuples are added to the array. First, an horizontal extension assigns values for the new column and second, an optional vertical extensions adds new rows if the horizontal extension was not able to cover all tuples.

Algorithm 1 IPO Strategy

```
procedure IPO(t, k, $(v_0, \ldots, v_{k-1})$)
    $Array \leftarrow$ cross-product of first $t$ columns
    for $i \leftarrow t, \ldots, k$ do
        HorizontalExtension($i$)
        if there are uncovered tuples then
            VerticalExtension($i$)
```

While the originally proposed IPO strategy was limited to pair-wise covering arrays, further work has produced variations of the algorithm which are able to generate covering arrays for arbitrary strengths. The three most prominent representatives of the IPO family are `IPOG`, `IPOG-F` and `IPOG-F2`. They all employ the same two-dimensional growth strategy and vertical extension mechanism. The only difference lies in the horizontal extension step. These differences are further discussed in Section 2.3.1, Section 2.3.2 and Section 2.3.3.

Algorithm 2 Vertical Extension

```
procedure VerticalExtension($i$)
    for all uncovered tuples $tuple$ do
        if $\exists row$ such that its entries match $tuple$ then
            add $tuple$ to $Array[row]$
            mark new tuples in row as covered
        else
            add new row to $Array$ containing only don't-care values
            add $tuple$ to new row
```

In all three algorithms, after the horizontal extension step, there might still be tuples left uncovered. In the vertical extension (see Algorithm 2) all remaining uncovered tuples are added to the array to ensure that the first i columns form a covering array. Tuples can either be added by appending a new row to the array that contains the tuple or by finding an already existing row which can fit the tuple. The latter case is possible since *don't-care* values can occur in the array and may be overwritten by a tuple without destroying the t-covering property. In this case, additional tuples might be covered in the row to which the tuple is added.

Remark (IPO Extension Mode)**.** *Due to its design, IPO is optimally suited to extend existing covering arrays with additional columns which is not as easily possible with one-test-at-a-time type algorithms.*

2.3.1 IPOG

The IPO strategy has become popular with its instantiation in the IPOG (In-Parameter-Order-General) [LKK+07] algorithm which is able to generate covering arrays for arbitrary strengths. It features a top-to-bottom horizontal extension, i.e., the new column is extended row-by-row from top to bottom. The procedure is depicted in Algorithm 3.

Algorithm 3 Horizontal Extension (IPOG)

```
procedure HorizontalExtension-IPOG($i$)
    for row $\leftarrow 0, \ldots,$Array.rows do
        $v \leftarrow$ select value with highest coverage gain
        if multiple candidate values then
            break tie
        Array[row][$i$] $\leftarrow v$
        Mark new tuples as covered
        if all tuples are covered then return
```

For each already existing row in the array, a new value is assigned to column i. Values are chosen from the columns domain V_i and in each step the coverage gain for all values

c_0	c_1	c_2	c_3
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	

(a) During Horizontal Extension

c_0	c_1	c_2	c_3
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1
$*$	0	$*$	1
$*$	1	$*$	0

(b) After Vertical Extension

Figure 2.1: Extending a $CA(4; 2, 3, 2)$ to a $CA(6; 2, 4, 2)$ with new column c_3

is computed. Coverage gain refers to the number of tuples that are marked as uncovered that would be covered should v be chosen for this particular row. The new value is then chosen such that it has maximal coverage gain. The extension ends when either all rows have been assigned a value or all tuples have been covered. In the latter case the extension terminates without having assigned a value to all rows which are left as *don't-care* values as their specific value does not change the t-covering property of the array. We will denote the presence of *don't-care* values with the symbol $*$.

In cases where multiple values provide equal, maximum coverage gain, a strategy must be chosen to break ties. This is studied in detail in chapter 4.

Example

To illustrate `IPOG` on an example, consider the generation of a covering array of strength 2 with 4 binary columns shown in Figure 2.1. Assume the algorithm has already generated a covering array for the first 3 columns and, in the process of extending the array by c_3, has already assigned a value to the first three rows during the horizontal extension. Next, a value is selected for the fourth row. The coverage gains are computed for each possible binary value: 0 has a coverage gain of 1 new tuple $(c_1 = 1, c_3 = 0)$ while 1 covers two tuples, $(c_0 = 1, c_3 = 1)$ and $(c_2 = 0, c_3 = 1)$ and is chosen for the fourth row. Now, as each existing row has been assigned a value and there are still tuples left which are not covered by the array, the vertical extension adds the remaining tuples by appending two new rows containing those tuples. In this case, *don't-care* values arise in the columns c_0 and c_2 since the newly added tuples are only between c_1 and c_3.

2.3.2 IPOG-F

In contrast to `IPOG`, `IPOG-F` [FLL+08] does not assign values to rows from top to bottom. Instead, it adds an additional dimension to the search space and searches for the optimal row as well as the optimal value for that row to assign such that the achieved coverage gain is maximal. This variant has the potential to find smaller covering arrays, since it has more degrees of freedom when constructing the new column. However, due to the extended search space and the fact that the array can have many rows, it also incurs additional complexity. The algorithm can however be implemented more efficiently using dynamic programming techniques as shown next. The algorithm is depicted in Algorithm 4. A naive approach would calculate the newly gained coverage for each row and value pair. This requires to examine all tuples in each row and checking their coverage status. This requires $\mathcal{O}(N \times \binom{i-1}{t-1} \times v_i)$ time. These checks can be completely eliminated by the following observation: Each row covers exactly $\binom{i-1}{t-1}$ tuples after extension step i. During the extension when a row is assigned a value in the new column i, then the coverage gain is $\binom{i-1}{t-1} - t_c$ where t_c denotes the number of tuples present in the row which have already been covered previously. Since $\binom{i-1}{t-1}$ is a constant, it suffices to maintain the number of covered tuples t_c for each row and value combination and we can compute the best $(row, value)$ pair in $\mathcal{O}(N \times v_i)$, where N is the number of rows in the array. t_c only needs to be updated when a $(row, value)$ pair has been selected. This is done by comparing common tuples between the selected row and all other unassigned rows and increasing the amount of covered tuples by the amount of shared and newly covered tuples.

Algorithm 4 Horizontal Extension (IPOG-F)

```
procedure HorizontalExtension-IPOG-F($i$)
    $T_c[row, v] \leftarrow 0, \forall row, v$
    while $\exists$ unassigned row do
        $row, v \leftarrow$ unassigned-row, value, s.t. $t_n = \binom{i-1}{t-1} - T_c[row, v]$ is a maximum
        if $t_n = 0$ then return
        Array$[row][i] \leftarrow v$
        for all unassigned rows $r$ do
            $S \leftarrow$ set of columns where the values for row and r match
            for all $t-1$ combinations $\Lambda$ between columns in $S$ do
                if $\Lambda$ uncovered then
                    $T_c[r, v] \leftarrow T_c[r, v] + 1$
        Mark new tuples covered in $row$ as covered
```

2.3.3 IPOG-F2

While `IPOG-F` considers the actual coverage to decide on the best value, `IPOG-F2` [FLL$^+$08] tries only to estimate the amount of covered tuples. This is an optimization which allows for a faster implementation compared to `IPOG-F`, but it sacrifices accuracy. The estimation is achieved by keeping an estimated value of covered tuples in an array for each row and value. When a value v is chosen for a row, the estimator is incremented by the number of shared tuples between that row and all other unassigned rows s. This is a simplification which assumes that all tuples shared between two rows have previously not been covered. The pseudo-code for the horizontal extension of `IPOG-F2` is shown in Algorithm 5.

Algorithm 5 Horizontal Extension (IPOG-F2)

```
procedure HorizontalExtension-IPOG-F2($i$)
    $T_c[row, v] \leftarrow 0, \forall row, v$
    while $\exists$ unassigned row do
        $row, v \leftarrow$ unassigned-row, value, s.t. $t_n = \binom{k-1}{t-1} - T_c[row, v]$ is a maximum
        if $t_n = 0$ then return
        Array[$row$][$i$] $\leftarrow v$
        for all unassigned rows $r$ do
            $S \leftarrow$ set of columns where the values for row and r match
            $T_c[r, v] \leftarrow T_c[r, v] + \binom{|S|}{t-1}$
        Mark new tuples covered in $row$ as covered
```

CHAPTER 3

Algorithmic Engineering

This chapter will expand on the high-level algorithm descriptions given in Section 2.3 and present algorithms and data structures necessary to implement the IPO strategy. First, techniques will be described as they are presented in the literature and afterwards optimizations to them will be proposed. As all three main representatives of the IPO strategy (`IPOG`, `IPOG-F` and `IPOG-F2`) share most of their implementation characteristics, this chapter will focus on `IPOG`, but all presented techniques are also applicable to the other two variants.

3.1 Algorithmic Design

This section describes important aspects and sub-procedures used in `IPOG` that were only described from a high-level point of view previously. These techniques have been described by [LKK+07] and [FLL+08], however, we will give some additional insights into the subject as a deeper understanding will help motivate the optimizations presented in Section 3.2.

3.1.1 Tuple Enumeration

Consider a CA with $i-1$ columns already constructed. In extension step i, all t-tuples between $t-1$ column selections for the first $i-1$ columns together with the to be constructed column i are considered. As the first $i-1$ columns already form a covering array it is not necessary to consider them in this step. Thus, there are $\binom{i-1}{t-1}$ t-column selections which include column i where each selection has as tuples the cross-product of the individual domains to be covered.

To keep track of all tuples and their meta data, a two-level data structure was proposed by [LKK+07]. A specific example is depicted in Figure 3.1. For the remaining chapters, we will refer to the data structure as *coverage-map*. Note, that the layout of the depicted data structure differs from the design of [LKK+07] as it relates to how it is stored in memory, but this difference does not impact the algorithm in any way.

This data structure stores all $\binom{i-1}{t-1}$ column selections (combinations) in its first level and each combination points into the second level where all its associated tuples are stored. The column selections are explicitly enumerated in the beginning of each column extension.

Tuples are only stored implicitly by means of associating each t-tuple with one bit of information indicating its coverage status. The tuple itself is implicitly encoded in the index of the coverage bit. We define this concept mentioned informally in literature [LKK+07] as a bijective function *pack* which maps tuples to an index. The inverse function can be used to instantiate a tuple from a given index. Let the set of possible tuples for a given column selection $\{i_1, \ldots, i_t\}$ be given by $\Gamma = \bigtimes_{1 \leqslant j \leqslant t} V_{i_j}$. *pack* : $\Gamma \to |\Gamma|$ is then defined as

$$pack((x_{i_1}, \ldots, x_{i_t})) = \sum_{1 \leqslant j \leqslant t} (x_{i_j} \cdot \prod_{1 \leqslant l < t-j} v_{i_l})$$

where v_{i_l} refers to the domain size of column i_l. An example of the mapping between indices and tuples can be found in Figure 3.1: Each entry in the coverage bit-vector is labelled by an index which is (implicitly) reset for each new column selection. The corresponding t-tuple is shown below the index as a column vector of length t.

Not having to store the actual tuples has the main advantage of storing only a constant amount of memory per tuple (i.e., one bit), regardless of the tuple size t.

3.1.2 Heuristic Value Selection

In the horizontal extension, values for the new column i are selected row-by-row from top to bottom. `IPOG` maximizes the *coverage gain* in each step and selects a value $v \in V_i$ which will cover the most not-yet covered tuples. The algorithm is depicted in Algorithm 6. For each value v, all possible $\binom{i-1}{t-1}$ tuples are enumerated and their index is computed. This index is then checked in the *coverage-map* and if that tuple is not marked as covered, the coverage gain is incremented by one. The value with the highest gain will then be selected for the row. Enumerating all tuples works by iterating over each column selection in the first level of the *coverage-map*. The values for the tuple are then looked up in the corresponding column in the array and are immediately used to compute the index (Line 6).

After the value with the highest gain is selected, it is assigned to the current row in column i and the covered tuples are marked in the *coverage-map*. See Figure 3.2 for an illustration.

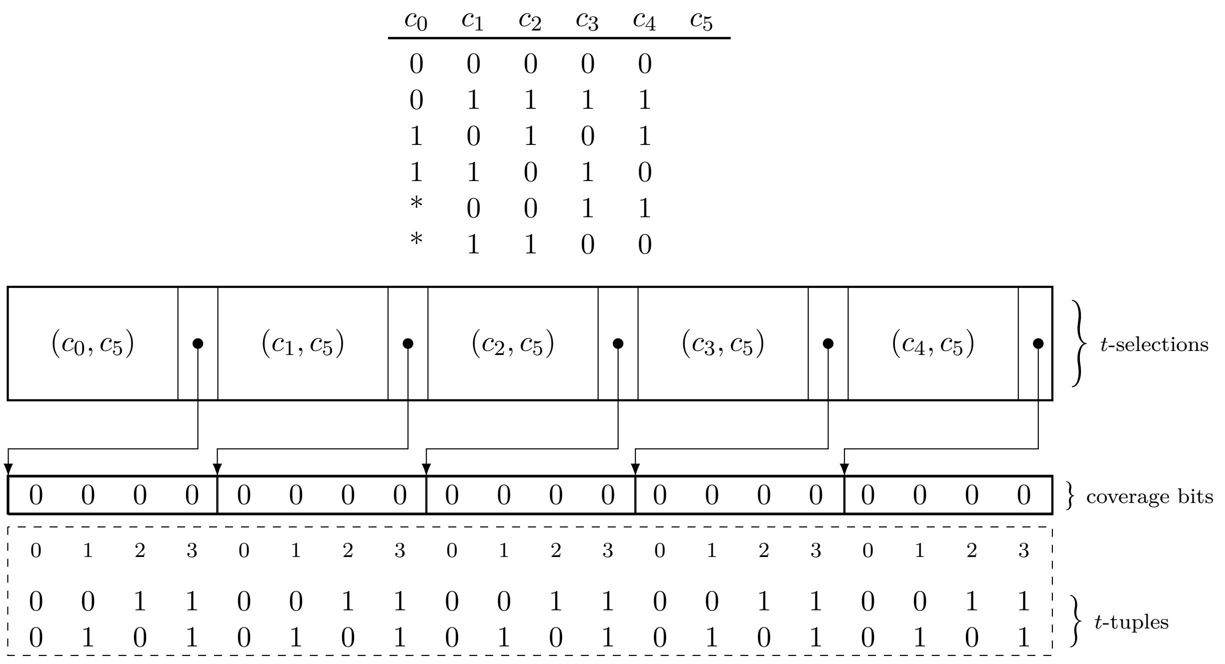


Figure 3.1: Initial *coverage-map* for a $CA(N; 2, 5, 2)$ being extended by a sixth binary column. All coverage bits are zero as no tuples have been covered yet. Items in the dashed box are only stored implicitly and do not take up any space.

Algorithm 6 Select-Best-Value

```
1: procedure SELECT-BEST($row$)
2:     $best \leftarrow undefined$
3:     for $v \leftarrow V_i$ do
4:         $gain \leftarrow 0$
5:         for all column selections $\{j_0, \ldots, j_{t-1}\}$ of the first $i-1$ columns do
6:             $index \leftarrow pack((Array[row][j_0], \ldots, Array[row][j_{t-1}], v))$
7:             if tuple at $index$ is not marked as covered in coverage-map then
8:                 $gain \leftarrow gain + 1$
9:         if gain higher than previous best or best is undefined then
10:            $best \leftarrow v$
    return best
```

c_0	c_1	c_2	c_3	c_4	c_5
0	0	0	0	0	0
0	1	1	1	1	1
1	0	1	0	1	0
1	1	0	1	0	1
*	0	0	1	1	
*	1	1	0	0	

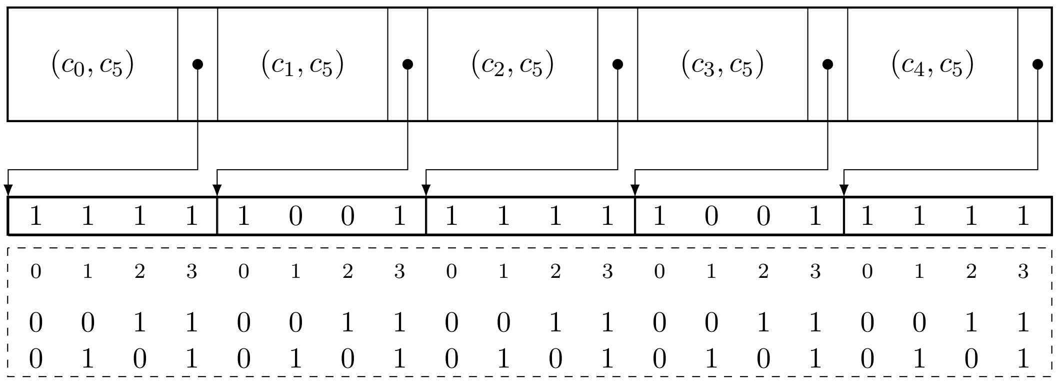


Figure 3.2: State of the *coverage-map* after the first four rows have been assigned a value for column c_5.

3.1.3 Suitable Row Selection

After the horizontal extension, it might be the case that there are still uncovered tuples left which trigger a vertical extension. This step covers the remaining tuples by appending new rows to the array. It is important to add as few new rows as possible in order to obtain small arrays. To facilitate this, `IPOG` tries to find existing rows in the array which are able to include uncovered tuples. We introduce the notion of a *suitable row* to be a row which can include a tuple without overwriting any entry with a different value, i.e., each entry in the tuple is only allowed to overwrite a value in the row if the value at that position is equal or a *don't-care* value.

For each new tuple to be added in the vertical extension, the algorithm will try to find an existing, suitable row. A naive solution will just check each row in the array and verify if the row is suitable for the given tuple by comparing each of its entries with the corresponding entry in the row.

However, we can limit the search to only those rows which contain at least one *don't-care* value since a tuple, which is added in the vertical extension step, does not, by construction of the algorithm, occur anywhere else in the array. Thus, it can only be added to an existing row if it overrides at least one *don't-care* value. Before the vertical extension, the array is scanned for rows which contain *don't-care* values and these rows will then be used to check for suitability.

c_0	c_1	c_2
0	1	0
0	*	1
*	1	1
*	0	0

(a) Suitable row candidates

c_0	c_1	c_2
0	1	0
0	*	1
*	1	1
1	0	0

(b) Tuple $(c_0 = 1, c_2 = 0)$ covered in existing row

Figure 3.3: Excerpt from a partial CA during construction.

Please refer to Figure 3.3 for an illustration of these concepts: Figure 3.3a highlights the candidate rows containing *don't-care* values which are able to include uncovered tuples. Figure 3.3b shows the result after the previously uncovered tuple $(c_0 = 1, c_2 = 0)$ has been added to the array.

3.2 Optimizations for the Algorithmic Design of IPOG

In this section we will present various optimizations to the `IPOG` algorithm. The common trait among these optimizations is that they reduce the amount of unnecessary computations by taking advantage of different properties inherent in the algorithm or the data structures it employs. As such, the application of these optimizations does not change the observable behaviour in any way, i.e., the generated (M)CAs are identical to the ones produced by the original, non-optimized version of the algorithm.

3.2.1 Simultaneous Coverage Gain Computation for Same-Prefix Tuples

Recall the implementation of selecting the next best value to cover a given row during the horizontal extension which was detailed in Algorithm 6. It will compute the coverage gain for value $v \in V_i$ separately and choose the one which maximizes that gain. However, this algorithm does not take advantage of the fact that the first $i - 1$ columns of the array are constant throughout the horizontal extension. Consider a tuple $(x_{i_1}, \dots, x_{i_{t-1}}, x_i)$ for a t-selection of columns $\{i_1, \dots, i_{t-1}, i\}$. Note that each column selection considered during the horizontal extension contains the currently being extended column i. As the prefix $(x_{i_1}, \dots, x_{i_{t-1}})$ is constant for a given row, computing the indices $\mathit{pack}((x_{i_1}, \dots, x_{i_{t-1}}, v))$ for all possible tuples $(x_{i_1}, \dots, x_{i_{t-1}}, v)$ where $v \in V_i$ will result in multiple identical computations for the first $t - 1$ elements.

A more efficient implementation is detailed in Algorithm 7. Here, we take advantage of the fact that tuples are stored in lexicographically increasing order in the *coverage-map*. This implies that tuples with the same $t - 1$-prefix, of which there are v_i, are always located next to each other.

It then suffices to calculate a base index $b = \mathit{pack}((x_{i_1}, \dots, x_{i_{t-1}}))$ of the $t - 1$-prefix and then check the coverage bits $b, \dots, b + v_i - 1$ and increase the coverage gain for those values where the bit is not set.

Complexity Analysis

In [LKK+07], the authors asses the complexity of determining the value with maximum coverage gain (Algorithm 6) as $O(\binom{k}{t-1} \times v)$. However, this ignores the complexity of *packing* tuples to their index. As this is feasible in linear time with respect to the tuple size t, $O(\binom{k}{t-1} \times v \times t)$ would constitute a more accurate bound. This algorithm, however, will perform many redundant index computations for the $t-1$-prefix of each column selection it considers. Since the first $i-1$ columns are constant throughout the horizontal extension, the proposed improvement in Algorithm 7 can avoid recomputations and minimize the time spent on calculating indices by reusing the base index for the $t-1$-prefix. The complexity is thus lowered to $O(\binom{k}{t-1} \times (v+t))$.

Algorithm 7 Improved-Select-Best-Value

```
procedure Improved-Select-Best($row$)
    for $v \in V_i$ do
        $gain_v \leftarrow 0$
    for all column selections $\{j_0, \dots, j_{t-1}\}$ of the first $i-1$ columns do
        $base\_index \leftarrow pack((A_{row,j_0}, \dots, A_{row,j_{t-1}}))$
        for $v \in V_i$ do
            $index \leftarrow base\_index + v$
            if tuple at $index$ is not marked as covered in coverage-map then
                $gain_v \leftarrow gain_v + 1$
    return $v$ such that $gain_v$ is maximal
```

3.2.2 t-column selection level search space pruning

We now turn to another optimization in the horizontal extension which enables the algorithm to skip fully covered portions of the *coverage-map*. In Algorithm 7, each column selection is considered once in order to compute coverage gains. However, if such a column selection is already fully covered, i.e., all of its associated tuples already have been added to the array, then this column selection will not contribute to the coverage gain. All such column selections can safely be skipped without changing the outcome of the horizontal extension.

In order to be able to skip fully covered column selections, we need to keep fine-grained information about each selection in the coverage map: For each selection in the first level of the *coverage-map* we store an integer which holds the number of already covered tuples for this column selection. Initially this count is zero. Each time we cover a new tuple for this column selection, the count is increased by one. To check whether a column selection is fully covered and can be skipped, it suffices to check that the count is equal to the total number of tuples belonging to this selection.

An example for the state of the enhanced *coverage-map* can be found in Figure 3.4.

c_0	c_1	c_2	c_3	c_4	c_5
0	0	0	0	0	0
0	1	1	1	1	1
1	0	1	0	1	0
1	1	0	1	0	1
*	0	0	1	1	
*	1	1	0	0	

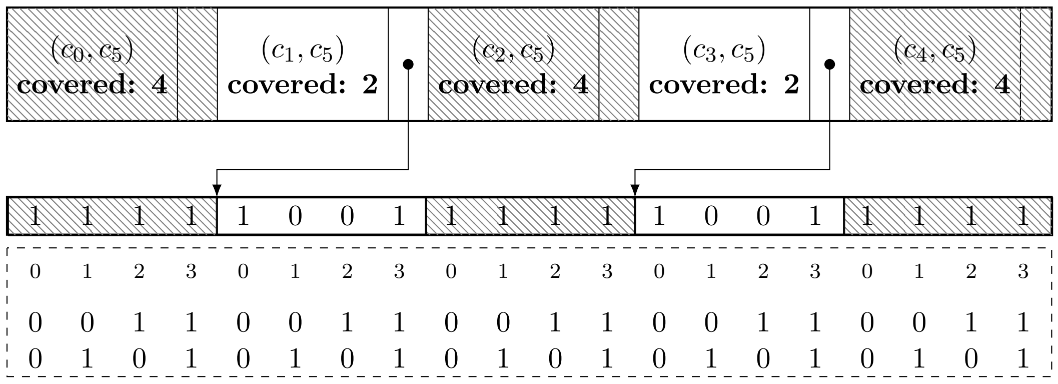


Figure 3.4: *Coverage-map* with fine-grained column-selection coverage information during the extension of column c_5. Patterned parts of the map are not consider in future steps of the algorithm.

3.2.3 Partitioning of Suitability Checks

This section presents an optimization in the search for suitable rows for which a simple approach was previously described in Section 3.1.3. A key insight into the problem here is that each tuple considered in extension step i contains exactly one entry for the new column i, i.e., the entry for column i is defined in every tuple. Checking rows which differ in position i from the tuple to be added will always result in a non-match.

Leveraging this property, a partition of all candidate rows (i.e. rows with at least one *don't-care* value) can be computed such that each set in the partition contains those rows which have the same value in column i. By testing only those rows for suitability which have the same matching value for column i as the to be added tuple, a large amount of trivial non-matches can be avoided and the total number of suitability checks can be reduced drastically. This partition can be computed once at the beginning of the vertical extension. See Figure 3.5 for an example of this partitioning.

The effect of this optimization will become more prominent for instances with growing v_i $(1 \leqslant i \leqslant k)$ as there are always v_i partitions.

c_0	c_1	c_2
0	1	0
0	*	1
*	1	1
*	0	0

(a) Candidate rows with $c_2 = 0$

c_0	c_1	c_2
0	1	0
0	*	1
*	1	1
*	0	0

(b) Candidate rows with $c_2 = 1$

Figure 3.5: Suitable row candidates, partitioned by the value of c_2

3.3 Implementation-level Optimizations

This section describes design considerations aimed at improving the efficiency of any implementation of `IPOG`. These optimizations do not change the algorithm, but allow for better performance by minimizing utilized memory and allocations of the resulting covering array. Also, a technique is discussed which promotes the parameter t of a covering array configuration to a compile-time constant.

3.3.1 Memory Optimizations

Optimizing the memory usage is of much importance when dealing with combinatorial problems [KS98]. Covering array generation is no exception and `IPOG`, as well as many other combinatorial algorithms, suffers from increasing memory demands as instance sizes increase. It is therefore important to manage the available space efficiently.

Memory usage in `IPOG` is dominated by two data structures, namely the *coverage-map* and the actual covering array. Considerations regarding the latter will be discussed in Section 3.3.2. Some memory saving techniques related to the *coverage-map* were discussed in previous sections, most importantly the implicit storage of t-tuples and their coverage representation by just one bit per tuple. Another possible optimization is to omit the explicit storage of the currently extended column i in the enumerated column selections. As column i is part of every column selection in extension step i, the first level of the *coverage-map* does not need to store it explicitly in each selection.

From a practical point of view, one key observation is that the number of columns and their associated domain sizes are usually small and seldom exceed $2^8 = 256$ [KBD+15]. This allows an implementation to use a single unsigned byte value to store the information. A fallback to a two-byte representation can easily be provided, should an instance require more columns or larger domain sizes.

Lastly, an important aspect in the design of data structures is their layout in memory and how they are accessed. The presented *coverage-map* uses two contiguous arrays, i.e., all column selections and tuples are stored adjacent in memory. As both arrays are accessed sequentially, this yields good spatial and temporal memory locality.

3.3.2 Array Representation

Although `IPOG` incrementally adds one column at a time it does not make sense to allocate new space for a column in each extension as it comes as an additional run-time cost. It is known from the beginning that the resulting array will consist of rows each having k entries. Therefore, the entire array can efficiently be stored as an $N \times k$ matrix where tests are stored row-wise. Row j then spans from the index $j \times k$ to $(j + 1) \times k$.

As k is fixed new rows can easily be added at the end of the array by allocating space for k new values. Initially, all values will be set to *don't-care* values. As `IPOG` considers adding one column at a time, columns beyond the i-th column (i.e. the one currently being added) will just be ignored by the algorithm at that stage.

3.3.3 Compile-time Specialization

A useful observation about the generation of covering arrays is that in practice the desired strength is a small integer [KWG04] as the size of the array grows exponentially with t. We can leverage this fact by parametrizing the implementation by a compile-time integer representing the strength. The compiler will then generate a specialized version of the algorithm for all strengths that are chosen to be supported. At run-time, the concrete specialized implementation for the selected strength is chosen. This optimization allows the compiler to treat the strength as a constant and can allow additional optimizations. As an added benefit, the amount of space required to store a column selection can be determined at compile-time enabling the implementation to use fixed-sized arrays instead of dynamically sized lists with the benefit of the compiler knowing the exact size to reserve for each column selection.

3.4 Evaluation

In order to compare and evaluate the proposed optimizations we conducted a set of experiments aimed at assessing the performance gain each optimization is able to achieve as well as the impact when all are combined together. Furthermore, we compare the results against one of the most widely used `IPOG` implementations, ACTS [YLKK13].

We implemented our algorithmic design of `IPOG` in Rust, a fast and safe systems-level programming language. We will refer to our implementation of `IPOG` as `FIPOG` (Fast In-Parameter-Order) for the remainder of this work. The Rust implementation will further be described in Chapter 5.

We created a baseline implementation, *FIPOG-Baseline*, which implements `IPOG` without any of the optimizations proposed in Section 3.2. It does however apply the implementation-level optimizations discussed in Section 3.3. The baseline and the `IPOG` implementation of ACTS are roughly comparable, but note that since ACTS is implemented in Java it becomes hard to isolate the cause of performance differences as they can either stem from the choice of language, the compiler and runtime, or the implementation.

	FIPOG				
Optimization	**Base**	**Simult.**	**Skip**	**Partitioned**	**All**
Simultaneous Coverage Gain Computation		✓			✓
t-column selection level search space pruning			✓		✓
Partitioning of Suitability Checks				✓	✓

Table 3.1: Test Subject Configurations.

From the baseline implementation, we constructed four additional versions of `FIPOG`. *FIPOG-Simultaneous*, *FIPOG-Skip* and *FIPOG-Partitioned* each including one of the proposed optimizations but not the other two while *FIPOG-All* includes all three optimizations. An overview of the configuration of each `FIPOG` version and which optimization is applied to which can be found in Table 3.1.

Each of the benchmark subjects as well as ACTS produce exactly the same covering arrays and thus there is no trade-off between performance and resulting arrays.

3.4.1 Experimental Setup

We selected a diverse set of test instances to assess the performance of each implementation. These test instances are categorized into two separate sets and are summarized in Table 3.2. The last column N contains the size of the produced arrays. The first set, *CA*, is a set of covering array instances where two of the three parameters of the configuration are fixed and either v or k is varied. The selection of these instances is aimed at showing the performance behaviour of the implementations as one parameter grows larger. Selecting from a wide range of possible values for v and k was done in order to ensure that the performance for large and small instances of covering arrays can be determined. The second set, *MCA*, is a set of mixed-level covering array instances. These present configurations for MCA instances which have been applied in real-world testing scenarios to test X.509 certificates [KS17] and cross-site scripting vulnerabilities [BGSW15]. Here, we test each instance for increasing strength t.

The experiments were performed on a machine with an Intel Core i7-4770 CPU clocked at 3.40GHz with 24GB of RAM. Our implementations were compiled with Rust version `1.17` and `ACTS` was executed using Java version `1.8.0_121`. We measured the elapsed time and maximum resident set size (RSS) using the Unix tool `time`.

3.4.2 Results and Evaluation

In Table 3.3, the results for the instance $CA(N; 3, 6, v)$ with $v \in \{20, 30, 40, 50, 60\}$ are shown. The first section of the table shows the measured time in seconds it took each implementation to generate the requested covering array. The second half of the table shows the speed-up factor of each implementation relative to the *FIPOG-Baseline*. A larger factor indicates faster generation time. The speed-up is furthermore visualized in

Group	Name	Configuration $(M)CA(N;t,k,v)$	N
CA	$CA(N;3,6,v)$	$CA(N;3,6,20)$	12,602
		$CA(N;3,6,30)$	41,323
		$CA(N;3,6,40)$	96,121
		$CA(N;3,6,50)$	184,904
		$CA(N;3,6,60)$	315,516
	$CA(N;4,k,8)$	$CA(N;4,10,8)$	12,224
		$CA(N;4,20,8)$	20,014
		$CA(N;4,30,8)$	24,805
		$CA(N;4,40,8)$	28,301
	$CA(N;5,k,2)$	$CA(N;5,20,2)$	160
		$CA(N;5,40,2)$	234
		$CA(N;5,60,2)$	279
		$CA(N;5,80,2)$	310
MCA	Coveringcerts	$MCA(N;4,33,(2^{23}3^{8}4^{1}5^{1}))$	456
		$MCA(N;5,33,(2^{23}3^{8}4^{1}5^{1}))$	1,636
		$MCA(N;6,33,(2^{23}3^{8}4^{1}5^{1}))$	5,385
	XSS	$MCA(N;3,11,(3^{6}9^{1}11^{1}14^{1}15^{1}23^{1}))$	5,626
		$MCA(N;4,11,(3^{6}9^{1}11^{1}14^{1}15^{1}23^{1}))$	60,651
		$MCA(N;5,11,(3^{6}9^{1}11^{1}14^{1}15^{1}23^{1}))$	510,156
		$MCA(N;6,11,(3^{6}9^{1}11^{1}14^{1}15^{1}23^{1}))$	1,850,808

Table 3.2: Benchmark instances

Figure 3.6. The results show that the suitability partition optimization speeds up the generation the most compared to the other two optimizations and gives a performance improvement by up to a factor of 6. Keeping track of column selection coverage and skipping fully covered ones seems to incur a small overhead and yields slightly worse generation times compared to the baseline. When all three optimizations are combined, however, the performance gain increases considerably, by up to a factor of 9.6. As these instances have a large value for v (compared to k and t) the results are explained by the partition of the search space for suitability checks. The larger v grows the more partitions exist which cut down the amount of time spent searching for suitable rows. Furthermore, the speed-up of the baseline implementation over ACTS is almost constant across different values of v, suggesting that an implementation in Rust instead of Java will give a performance benefit limited by a constant factor only.

Table 3.4 and Figure 3.7 display the results for the experiments performed for instances of $CA(N;4,k,8)$ with $k \in \{10,20,30,40\}$. The *FIPOG-Simultaneous* implementation improves performance the most by up to a factor of 1.58. The other two optimizations are either neutral or deteriorate the generation time by almost 20%. Taken in combination, this leads to an overall slightly worse performance gain than when just run with *FIPOG-*

			FIPOG				
Metric	v	**IPOG-ACTS**	**Baseline**	**Simultaneous**	**Skip**	**Partitioned**	**All**
Time (s)	20	0.68	0.05	0.04	0.05	0.03	0.02
	30	2.90	0.35	0.33	0.36	0.14	0.09
	40	10.72	1.44	1.41	1.48	0.42	0.27
	50	35.99	4.82	4.71	4.94	1.05	0.65
	60	98.60	12.77	12.43	12.99	2.13	1.33
Speedup	20	0.07	1.00	1.25	1.00	1.67	2.50
	30	0.12	1.00	1.06	0.97	2.50	3.89
	40	0.13	1.00	1.02	0.97	3.43	5.33
	50	0.13	1.00	1.02	0.98	4.59	7.42
	60	0.13	1.00	1.03	0.98	6.00	9.60

Table 3.3: Results for $CA(N; 3, 6, v)$

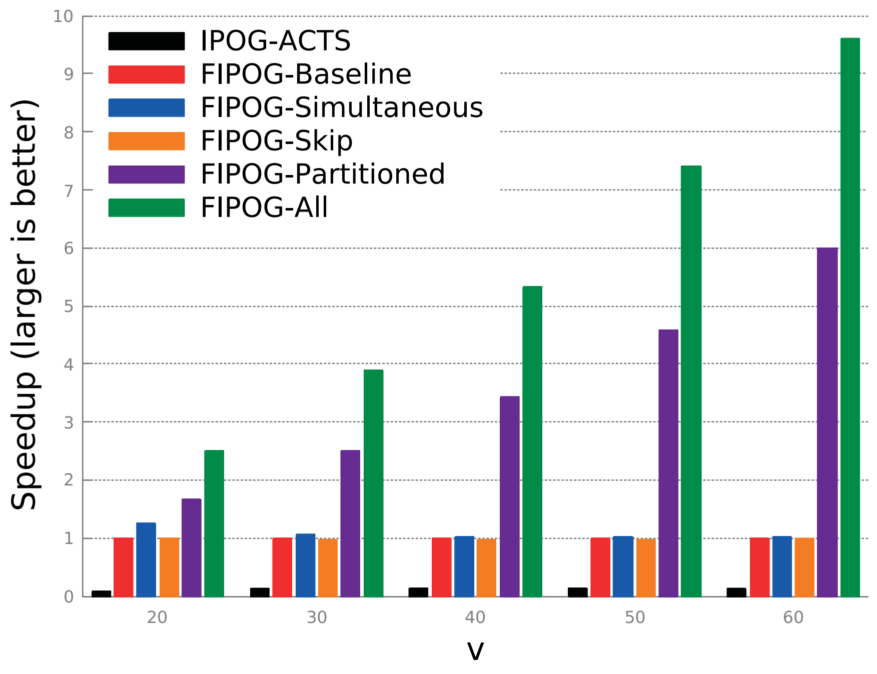


Figure 3.6: $CA(N; 3, 6, v)$: speedups relative to baseline

Simultaneous. This is due to the negative performance impact of the *FIPOG-Skip* optimization that performs extra work to keep track of covered column selections which does not translate into an improvement of generation time in this particular instance.

The benchmark results for the instances $CA(N; 5, k, 2)$ with $k \in \{20, 40, 60, 80\}$ are summarized in Table 3.5 and depicted in Figure 3.8. The test generation times for these instances improve with both the *FIPOG-Simultaneous* and *FIPOG-Skip* implementations. The latter improves the time by up to 25% while the former achieves up to 18% decreased

			FIPOG				
Metric	k	**IPOG-ACTS**	**Baseline**	**Simultaneous**	**Skip**	**Partitioned**	**All**
Time (s)	10	1.20	0.19	0.16	0.20	0.12	0.09
	20	19.99	4.31	2.98	4.99	4.14	2.98
	30	148.56	31.52	21.10	38.17	31.08	22.05
	40	544.78	125.18	79.02	153.17	124.67	83.16
Speedup	10	0.16	1.00	1.19	0.95	1.58	2.11
	20	0.22	1.00	1.45	0.86	1.04	1.45
	30	0.21	1.00	1.49	0.83	1.01	1.43
	40	0.23	1.00	1.58	0.82	1.00	1.51

Table 3.4: Results for $CA(N; 4, k, 8)$

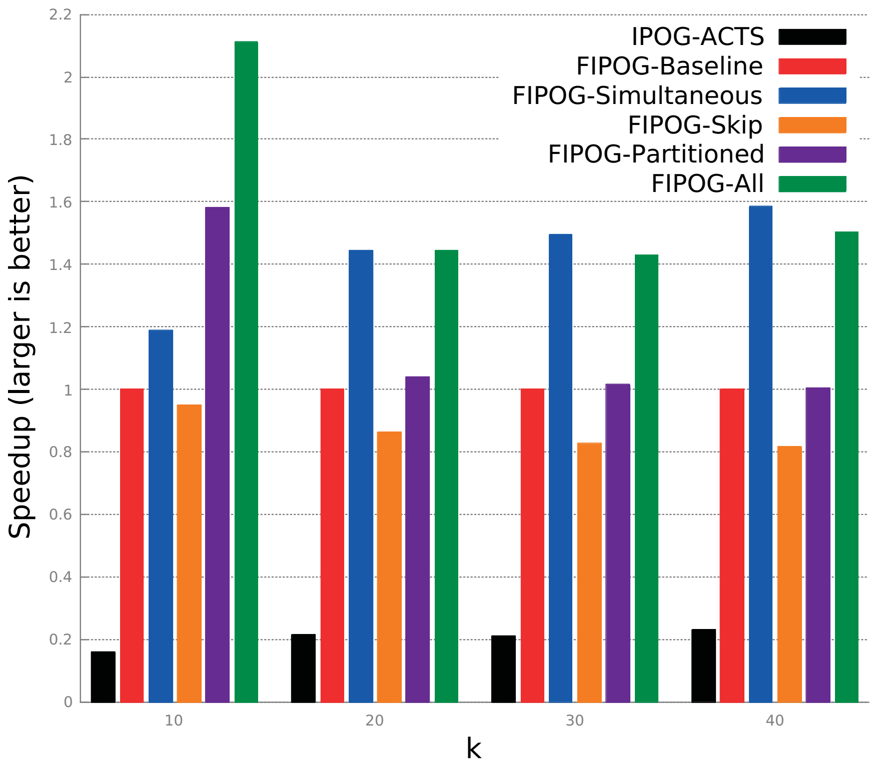


Figure 3.7: $CA(N; 4, k, 8)$: speedups relative to baseline

generation times. Furthermore, combined the results show an even larger performance gain by a factor of at most 1.63. As the strength t and the number of columns k are quite high in this benchmark (compared to the others), there are up to $\binom{80}{5} \approx 2.4 \cdot 10^7$ column selections to consider. The optimizations which benefit the performance in this case are both aimed at improving the performance of the horizontal extension which is dominated by the amount of column selections. Thus, the improved complexity as

			FIPOG					
Metric	k	**IPOG-ACTS**	**Baseline**	**Simultaneous**	**Skip**	**Partitioned**	**All**
Time (s)	20	0.25	0.03	0.03	0.03	0.04	0.03
	40	14.20	2.12	1.82	1.81	2.13	1.41
	60	119.82	20.37	17.61	16.41	20.42	12.65
	80	636.25	100.90	85.79	81.39	100.57	61.93
Speedup	20	0.12	1.00	1.00	1.00	0.75	1.00
	40	0.15	1.00	1.16	1.17	1.00	1.50
	60	0.17	1.00	1.16	1.24	1.00	1.61
	80	0.16	1.00	1.18	1.24	1.00	1.63

Table 3.5: Results for $CA(N; 5, k, 2)$

described in Section 3.2.1 and the skipping of already covered column selections yield better results as the number of column selections grow.

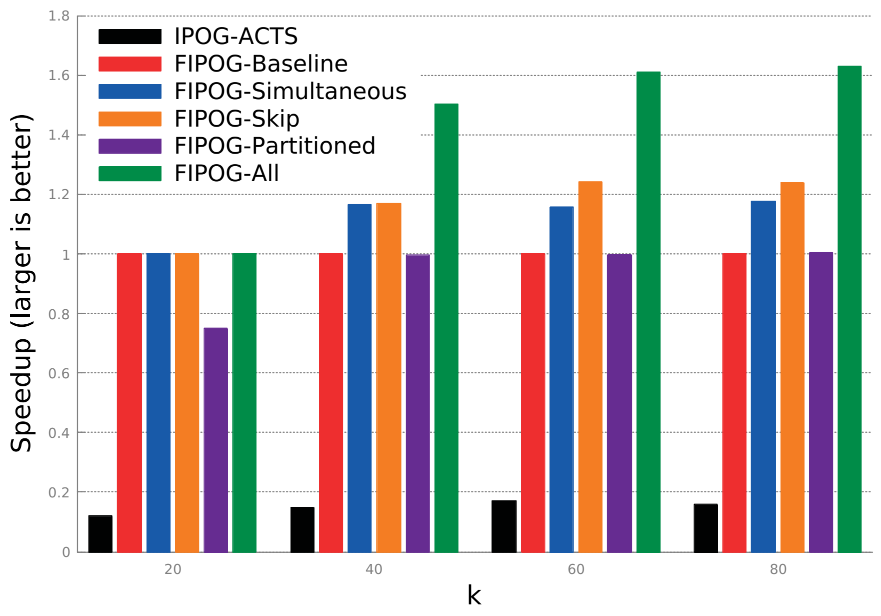


Figure 3.8: $CA(N; 5, k, 2)$: speedups relative to baseline

The result for the *Coveringcert* and *XSS* instances can be found in Table 3.6, Figure 3.9a, Table 3.7 and Figure 3.9b respectively. In the case of *XSS* there is no result for ACTS at strength 6 since the benchmark was not finished after being allowed to run for over 24 hours. As these instances represent combinatorial test models used to test real-world software, it is of particular interest to compare the applicability in the testing cycle. In many reported combinatorial testing scenarios, ACTS has been used to generate the test sets (i.e., mixed-level covering arrays). However, as can be seen in the benchmark results, test generation time can quickly become very large and inhibit fast iteration

			FIPOG				
Metric	t	**IPOG-ACTS**	**Baseline**	**Simultaneous**	**Skip**	**Partitioned**	**All**
Time (s)	4	0.96	0.14	0.11	0.17	0.14	0.13
	5	48.61	3.14	2.37	3.49	3.13	2.32
	6	1953.59	85.31	53.84	63.51	85.24	38.42
Speedup	4	0.15	1.00	1.27	0.82	1.00	1.08
	5	0.06	1.00	1.32	0.90	1.00	1.35
	6	0.04	1.00	1.58	1.34	1.00	2.22

Table 3.6: Results for Coveringcerts

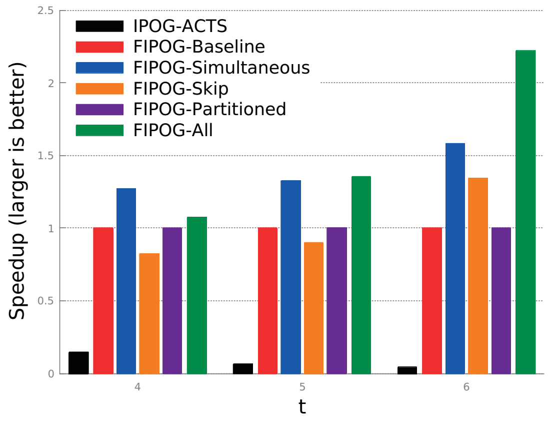
			FIPOG				
Metric	t	**IPOG-ACTS**	**Baseline**	**Simultaneous**	**Skip**	**Partitioned**	**All**
Time (s)	3	0.90	0.06	0.07	0.07	0.02	0.02
	4	61.30	8.35	8.23	7.81	1.44	1.33
	5	5927.00	539.30	540.36	541.09	41.81	40.36
	6	N/A	24956.00	24518.00	25705.00	3970.00	2446.66
Speedup	3	0.07	1.00	0.86	0.86	3.00	3.00
	4	0.14	1.00	1.01	1.07	5.80	6.28
	5	0.09	1.00	1.00	1.00	12.90	13.36
	6	N/A	1.00	1.02	0.97	6.29	10.20

Table 3.7: Results for XSS

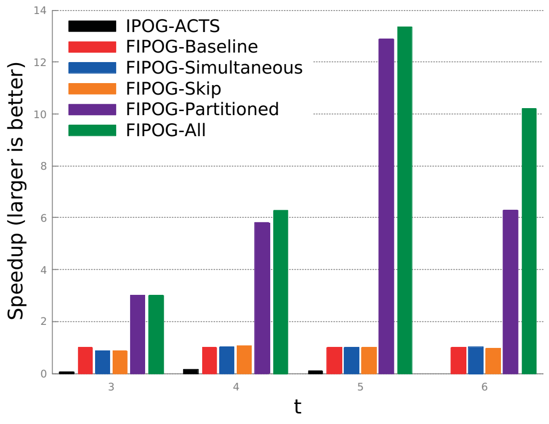
times, especially when t increases. All versions of *FIPOG* improve the time to generate a certain covering array by a very large factor and can even generate a covering array of strength $t + 1$ faster than ACTS manages to do for an array with just strength t.

We furthermore measured the peak memory utilization for each implementation and benchmark and the results are summarized in Table 3.8. The *FIPOG* implementations have a similar memory usage profile in most benchmarks which is to be expected as they differ only in small parts with regards to the data structures they employ. There are however two cases where a significant difference can be observed. First, the benchmark $CA(N; 5, 80, 2)$ shows that the additional coverage tracking counters present in the *FIPOG-Skip* and *FIPOG-All* implementations can contribute to the overall memory usage. Second, the *FIPOG-Partitioned* and *FIPOG-All* implementations exhibit a larger memory footprint in the $MCA(N; 6, 11, (3^6 9^1 11^1 14^1 15^1 23^1))$ (XSS) benchmark, compared to the rest. As this instance produces especially large arrays, it is explained by the fact, that storing the partition of suitable rows requires more memory than the naive approach, as some rows may occur in all partitions if they contain a *don't-care* value in the extended column i.

All *FIPOG* implementations use significantly less memory compared to ACTS, reducing the amount of allocated memory by up to a factor of 250.



(a) Coveringcerts



(b) XSS

Figure 3.9: Speedups relative to baseline

		FIPOG				
Benchmark	**IPOG-ACTS**	**Baseline**	**Simult.**	**Skip**	**Partitioned**	**All**
$CA(N; 3, 6, 60)$	2006.72	8.23	8.17	8.22	10.25	10.05
$CA(N; 4, 40, 8)$	2109.15	12.21	12.27	12.08	12.27	12.25
$CA(N; 5, 80, 2)$	2633.47	29.79	28.65	38.77	28.83	38.74
Coveringcerts, $t = 6$	1636.17	18.69	18.71	18.34	18.74	18.52
XSS, $t = 6$	N/A	56.12	56.17	56.16	83.18	79.82

Table 3.8: Peak memory utilization in megabytes for selected benchmarks.

In summary, `FIPOG` outperforms the `IPOG` implementation of ACTS in all of our benchmarks and improves test generation times by up to a *factor* of 146. On average, the optimizations improve test generation times by a factor of 3.5 compared to the `FIPOG` baseline or 17.5 times compared to `ACTS`. The average speed-ups are depicted in Figure 3.10. Additionally, `FIPOG` saves large amounts of memory compared to ACTS and lowers the peak memory usage by more than two orders of magnitude on our benchmarks. It should furthermore be highlighted that all tested implementations produce *exactly* the same covering arrays, so there is no trade-off between quality of generated arrays and performance.

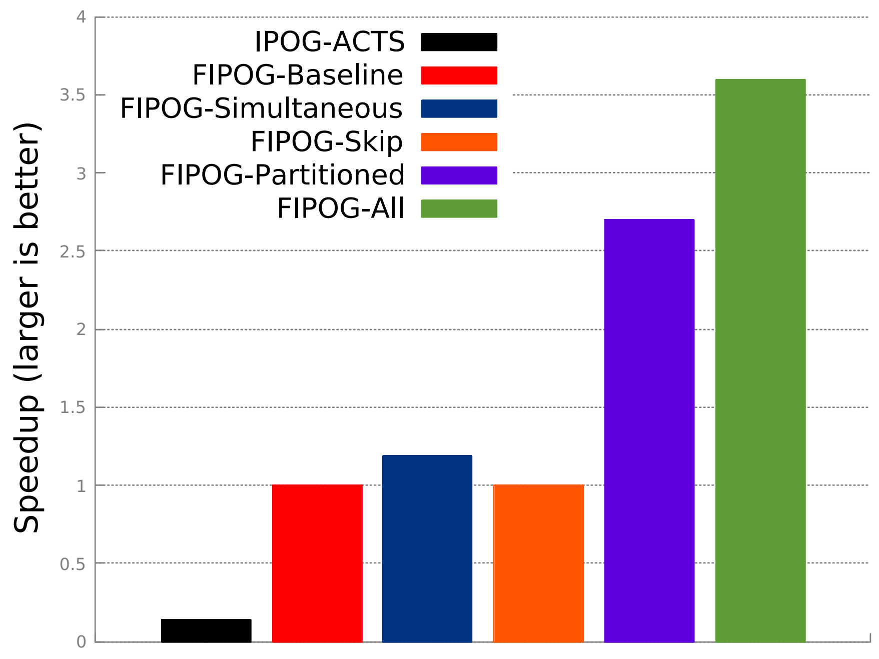


Figure 3.10: Average speed-up for different optimizations

CHAPTER 4

Algorithm Variations

This chapter will expand on so-far ignored design choices of the IPO strategy which can influence the generated covering arrays. Since the algorithms in the IPO family are greedy, they provide no guarantee that the generated test sets are optimal. Nonetheless, it is of interest to study and understand the behavior of IPO and optimize parameters to the algorithm such that better test sets are produced by the algorithms.

The IPO strategy has no explicitly tunable parameters, but the algorithm design itself can be instantiated in a few different ways which have an impact on the resulting test sets. In the next sections, we will explore different means of tie-breaking, and study the impact of the order in which tuples are enumerated as well as the order of column extensions.

4.1 Tie-Breaking

During the horizontal extension (see Algorithm 3), tie-breaking may be necessary in the case that two or more values for a row in the new column provide the same, maximum coverage gain, i.e., cover the most new t-tuples. We will refer to these equally-well suited values as candidates. Figure 4.1 illustrates the general problem: the heuristic has only limited local information, i.e., how many new t-tuples each possible value covers in this row. If the primary heuristic selection criterion is unique (i.e., only one value provides the best coverage gain) then the algorithm can proceed. Otherwise, a choice must be made about which of the possible candidates to chose. This selection might have an influence on future decisions where inferior tie-breaking decisions from previous rows reduce the potential to cover new tuples at a later stage of the algorithm.

As it would be too costly to try all possible tie-breaking choices (i.e., a tree search where nodes are tie-breaking decision points), tie-breaking can also only be performed heuristically. In the following, we will give an overview into possible tie-breaking strategies.

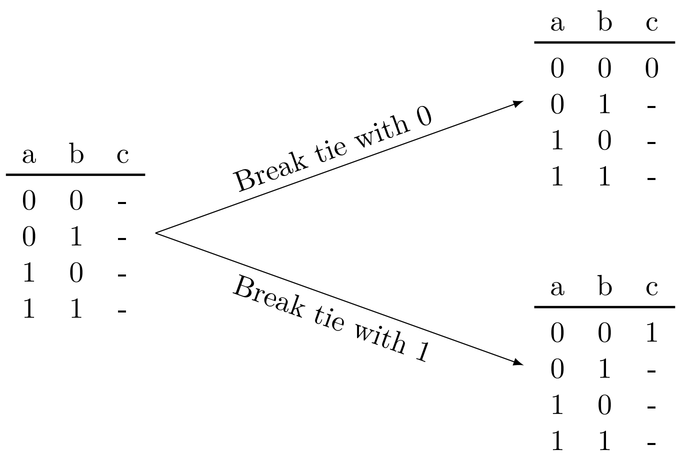


Figure 4.1: Tie-breaking possibilities for a binary array. Both values provide equal coverage gain.

Random Tie-Breaker

The simplest approach is to choose one value out of all candidates at random. This can be implemented efficiently but it will introduce non-determinism to algorithm and the generated covering arrays will possibly differ on subsequent runs of the algorithm. This tie-breaker is oblivious to the previous history of the extension.

Deterministically-seeded Random Tie-Breaker

This is a variant of the random tie-breaker. Here, ties are still broken randomly with the help of a pseudo-random generator, but the generator is seeded with a constant at the beginning which results in a deterministic behaviour of the algorithm.

Lexicographic Tie-Breaker

This tie-breaker will always prefer the (lexicographically) smallest candidate if multiple are available. This can of course introduce a bias towards smaller values in the new column.

Cyclic Tie-Breaker

This tie-breaker builds upon the lexicographic tie-breaker, but maintains the last chosen value and starts the search from this one instead of the first. The aim is to remove bias towards smaller values, however the last chosen value is more likely to be picked again in the next iteration.

Cyclic-next Tie-Breaker

This tie-breaker works exactly as the cyclic tie-breaker, but will start from the next value following the last chosen value. This tie-breaker was first proposed in [GLD$^+$14] and [GLD$^+$15].

Value-balanced Tie-Breaker

This tie-breaker keeps track of how many times a value has been used so far in the extended column. In an optimal situation, each value for the new column occurs exactly the same amount of times and the aim of this tie-breaker is to mimic this behaviour by balancing the occurrences of these values. Values are preferred when they so far have occurred less frequently than other candidate values.

α-balanced Tie-Breaker

This tie-breaker builds upon the value-balanced tie-breaker by not only considering the balance of values in the new column, but the balance of lower-strength tuples involving the new parameter. This is based on the notion of α-balance which was introduced by [KS16] and functions as a tie-breaker in the following way: first, the number of would-be-covered $t-1$ tuples are compared for each candidate. If there still is a tie, the next lower strength is tried and so on. If at $t=1$ there still exists a tie, then the smallest value will be preferred.

4.2 Orderings of Parameter and Tuple Enumeration

4.2.1 Tuple Enumeration Order

In the vertical extension, uncovered tuples are added one-by-one to the array. So far it has not been studied, if different enumeration orders of these tuples have any impact on the resulting arrays. We propose, besides the common lexicographically-ascending (tuples of small lexicographical order first) order, the reverse, i.e., from lexicographically largest to smallest. Furthermore, switching between the orderings every other vertical extension could prove beneficial to achieve smaller arrays.

4.2.2 Parameter Ordering

One simple option to influence the covering array generation is the order in which columns are extended. Since the covering property is not affected by column permutations one can permute the configuration before starting the generation and apply the reverse permutation afterwards. Note that this is only useful for mixed-level covering arrays. Informal consensus is that the IPO strategy generates smaller arrays when columns are sorted by decreasing alphabet size, but, to the best of our knowledge, this has so far not been subject to an experimental evaluation.

While the number of column permutations in general is too large in practice, we propose to investigate the following:

Ascending Sort columns with increasing alphabet size from smallest to largest

Descending Sort columns with decreasing alphabet size from largest to smallest

Alternating Intersperse large and small columns and switch between large and small columns from one extension to the next. We propose two variants. The first starts with the smallest, followed by the largest and thirdly the second-smallest, etc. The second starts with the largest, followed by the smallest and so on.

4.3 Evaluation

4.3.1 Setup

To evaluate the different algorithm configurations we chose a set of (M)CA instances based upon the benchmarks used in [BCC05] to study the behaviour of greedy, one-test-at-a-time MCA generation algorithms. The instances are summarized in Figure 4.2a.

Instances
10^4
3^{40}
3^4
6^4
$3^4, 4^5$
$6^6, 5^5, 3^4$
$7^8, 2^{20}$
$5^1, 3^8, 2^2$
$5^{10}, 2^{10}$
$8^2, 7^2, 6^2, 5^2$
$10^1, 9^1, 8^1, 7^1, 6^1, 5^1, 4^1, 3^1, 2^1, 1^1$

(a) Set of benchmark (M)CA instances

Tie Breakers	Tuple Orders	Parameter Orders
Alpha-Balanced	Alternating	Alternating-large
Cyclic	Ascending	Alternating-small
Cyclic-next	Descending	Ascending
Deterministic		Descending
Lexicographic		
Random		
Value-Balanced		

(b) Configuration options for `IPO`

Figure 4.2: Benchmark Setup

We implemented all IPO variants in Rust which is further described in Chapter 5. The particular algorithm as well as the tie-breaker, tuple order and parameter ordering are selectable via a configuration option at runtime. This results in 63 distinct algorithm configurations for CA generation and 252 distinct configurations for MCA generation. The configuration options are summarized in Figure 4.2b.

Each algorithm configuration was used to generate (M)CAs for the selected benchmark instances for strengths between 2 and 4. The experiments were conducted on the Graham

	IPOG	IPOG-F	IPOG-F2
Tie Breaker			
Alpha-balanced	1.0128 ±0.0759	0.9632 ±0.0619	1.0572 ±0.0662
Cyclic	1.0175 ±0.1394	0.9461 ±0.0678	1.0379 ±0.0867
Cyclic-next	0.9721 ±0.0858	0.9403 ±0.0799	1.0296 ±0.1012
Deterministic	0.9920 ±0.0429	0.9560 ±0.0539	1.0500 ±0.0664
Lexicographic	1.0140 ±0.0764	0.9651 ±0.0687	1.0580 ±0.0673
Random	0.9933 ±0.0465	0.9548 ±0.0545	1.0497 ±0.0673
Value-balanced	0.9951 ±0.0630	0.9590 ±0.0563	1.0549 ±0.0645
Tuple Order			
Alternating	0.9950 ±0.0656	0.9533 ±0.0580	1.0454 ±0.0681
Ascending	0.9987 ±0.0663	0.9582 ±0.0626	1.0584 ±0.0794
Descending	0.9944 ±0.0630	0.9530 ±0.0561	1.0432 ±0.0644
Parameter Order			
Alternating-large	0.9962 ±0.0309	0.9529 ±0.0268	1.0503 ±0.0510
Alternating-small	1.0082 ±0.0374	0.9667 ±0.0336	1.0763 ±0.0525
Ascending	1.0285 ±0.0443	1.0006 ±0.0440	1.0870 ±0.0519
Descending	0.9463 ±0.0791	0.8910 ±0.0442	0.9952 ±0.0793

Table 4.1: Relative improvement for different configurations compared to the mean

cluster of the Shared Hierarchical Academic Research Computing Network (SHARCNET). In the following, we discuss selected and aggregated results, but we provide the full data set as well as visualizations on a dedicated website[1] for the interested reader.

4.3.2 Results

In order to meaningfully compare different configurations options across instances we first normalized the computed covering array sizes to a relative measure representing the deviation of the mean. We computed the mean for each instance and based on the result computed the relative improvement or degradation for each individual run. This value shows how much better or worse one configuration performs in comparison to the other ones. The results are summarized in Table 4.1 and are visualized in Figures 4.3a and 4.3b.

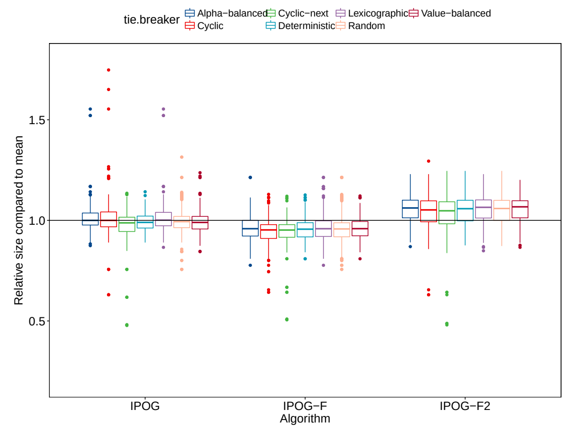
In general, `IPOG-F` produces the smallest arrays, followed by `IPOG` and `IPOG-F2`. Comparing the results for the different tie-breakers, no one choice seems to impact array sizes significantly, however, the `Cyclic-next` tie-breaker overall yields the best results. It, together with the `Cyclic` tie-breaker is able to generate some arrays with up to 50% less rows. However, the `Cyclic` tie-breaker exhibits extreme results in the other direction

[1] `https://matris.sba-research.org/data/iwoca2018`

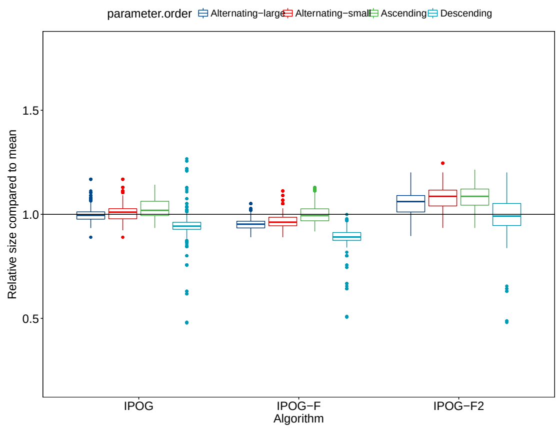
and in corner-cases with array sizes exceeding 50% larger than the mean are produced. This is also the case for the `Alpha-balanced` and `Lexicographic` tie-breaker.

Judging from the results in Table 4.1, the order in which tuples are enumerated does not seem to affect the resulting covering array size in any significant way.

The largest impact can be attributed to the sorting order of columns. Sorting in descending order of alphabet size leads to significantly smaller covering arrays, especially in the case of `IPOG-F`. Alternating between large and small columns has some impact and is better than sorting columns in ascending order.



(a) Tie-breakers



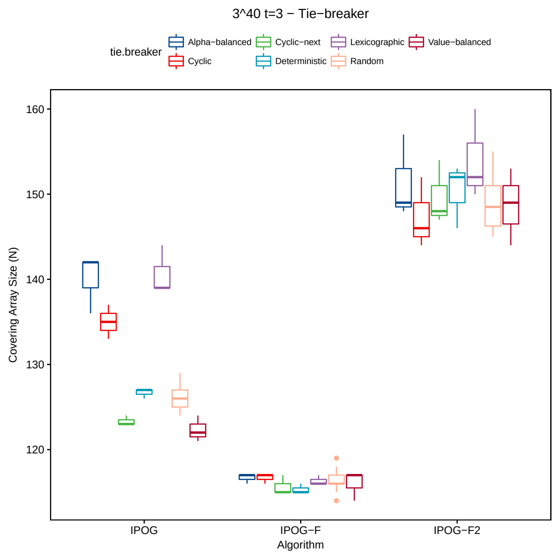
(b) Parameter orders

Figure 4.3: Relative improvement compared to the mean

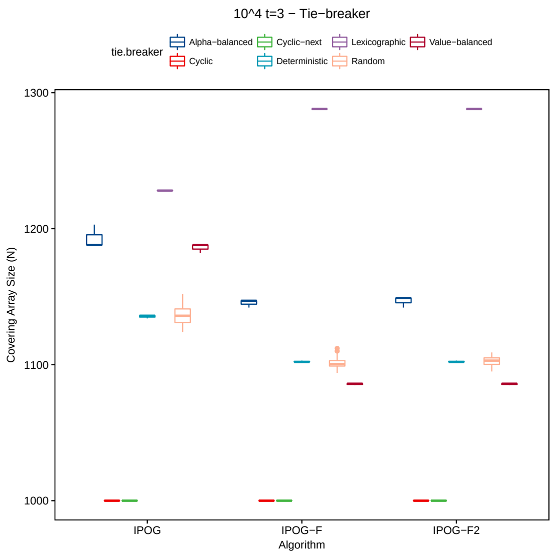
Selected benchmark results. Aside from the general performance, for specific instances the various configuration options can have differing impact. In the following, we discuss some results for selected instances. In order to meaningfully analyze the results we have grouped the results by both the algorithm (i.e., `IPOG`, `IPOG-F` or `IPOG-F2`) and one of either tie-breaker, tuple-order or parameter-order. Inside each group we have computed the mean and the standard deviation. The results show absolute values instead of relative difference.

3^{40} $(t = 3)$

The results of this experiment are summarized in Table 4.2 and the generated covering array sizes per tie-breaker are visualized in Figure 4.4a. `IPOG-F` produces the smallest arrays and shows very low variance when comparing different tie-breakers. In contrast, the results for `IPOG` are much more dependent on the tie-breaker. Here, the best results are obtained with the `Value-balanced` tie-breaker which produces arrays 16% smaller than when using the `Alpha-balanced` tie-breaker. `IPOG-F2` shows no significantly differing behavior with different tie-breakers. Furthermore, the order in which tuples are enumerated have no major impact.



(a) 3^{40} $(t = 3)$



(b) 10^4 $(t = 3)$

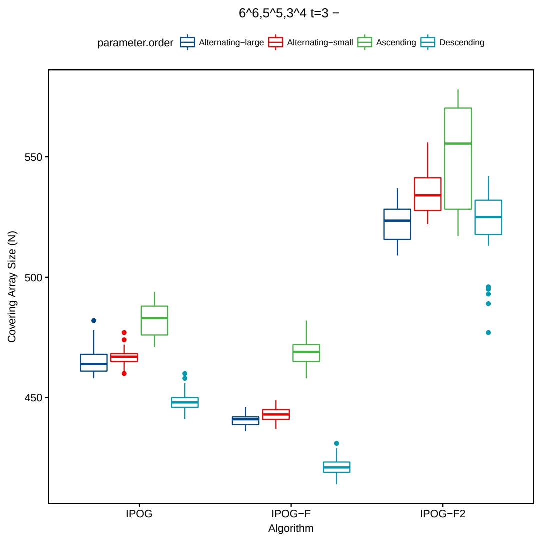
Figure 4.4: Results for different tie-breakers

	3^{40} $t = 3$			10^4 $t = 3$		
	IPOG	IPOG-F	IPOG-F2	IPOG	IPOG-F	IPOG-F2
Tie Breaker						
Alpha-balanced	140.0 ±3.5	116.7 ±0.6	151.3 ±4.9	1193.0 ±8.7	1145.3 ±2.9	1146.7 ±4.0
Cyclic	135.0 ±2.0	116.7 ±0.6	147.3 ±4.2	1000.0 ±0.0	1000.0 ±0.0	1000.0 ±0.0
Cyclic-next	123.3 ±0.6	115.7 ±1.2	149.7 ±3.8	1000.0 ±0.0	1000.0 ±0.0	1000.0 ±0.0
Deterministic	126.7 ±0.6	115.3 ±0.6	150.3 ±3.8	1135.3 ±1.2	1102.3 ±0.6	1102.3 ±0.6
Lexicographic	140.7 ±2.9	116.3 ±0.6	154.0 ±5.3	1228.0 ±0.0	1288.0 ±0.0	1288.0 ±0.0
Random	125.9 ±1.3	116.4 ±1.0	148.9 ±2.7	1136.2 ±7.0	1101.4 ±4.3	1102.2 ±3.7
Value-balanced	122.3 ±1.5	116.0 ±1.7	148.7 ±4.5	1186.0 ±3.5	1085.7 ±0.6	1085.7 ±0.6
Tuple Order						
Alternating	127.6 ±5.7	116.2 ±0.9	147.4 ±2.3	1130.0 ±58.2	1102.4 ±62.1	1103.7 ±62.1
Ascending	128.7 ±6.4	116.5 ±1.0	152.6 ±3.0	1131.9 ±59.2	1102.1 ±61.9	1102.1 ±61.9
Descending	127.6 ±4.8	116.1 ±1.0	148.3 ±2.5	1132.6 ±58.2	1102.1 ±62.0	1102.6 ±62.2

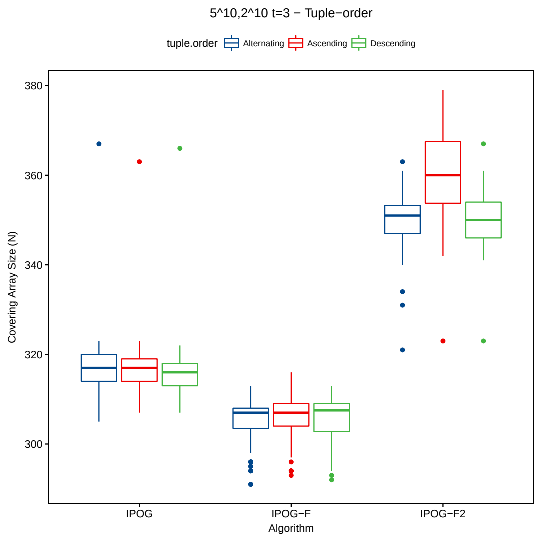
Table 4.2: Results for CA experiments

10^4 $(t = 3)$

The results for this experiment can be found in Table 4.2 and a comparison of the tie breakers can be found in Figure 4.4b. Here, the configurations which use either the `Cyclic` or `Resuming` tie-breakers manage to generate an orthogonal array (since the size is equal to v^t) for the three algorithms. Interestingly, the `Lexicographic` tie-breaker, although similar to the other two, performs the worst in all cases with almost 30% larger array sizes. As before, in this case the tuple order has no real impact.



(a) $6^6 5^5 3^4$ $(t = 3)$



(b) $5^{10} 2^{10}$ $(t = 3)$

Figure 4.5: Results for different parameter orders (left) and tuple orders (right)

$6^6 5^5 3^4$ $(t = 3)$

For this instance (see Table 4.3), there is no large variance when comparing different tie-breakers. `IPOG-F` produces the smallest arrays, while `IPOG-F2` produces the largest. Here, the parameter order has a measurable impact and ordering the parameters by descending size can improve array sizes by up to 5% in this case. These results are visualized in Figure 4.5a.

$5^{10} 2^{10}$ $(t = 3)$

The results for this instance are described in Table 4.3 and a comparison of different tuple orders is visualized in Figure 4.5b. The tuple order seems to only make a difference for `IPOG-F2`, where both the `Alternating` and `Descending` order outperform the `Ascending` order. This is also the case in instance $6^6 5^5 3^4$ $(t = 3)$.

	$6^6 5^5 3^4 t = 3$			$5^{10} 2^{10} t = 3$		
	IPOG	IPOG-F	IPOG-F2	IPOG	IPOG-F	IPOG-F2
Tie Breaker						
Alpha-balanced	470.5 ±13.6	441.9 ±17.1	533.8 ±18.2	316.0 ±5.2	305.5 ±5.4	353.5 ±7.8
Cyclic	465.8 ±14.1	443.9 ±16.3	532.3 ±16.4	331.1 ±20.7	308.2 ±1.6	344.2 ±14.0
Cyclic-next	464.7 ±12.3	444.0 ±18.1	529.0 ±23.8	316.3 ±4.4	307.6 ±3.9	352.0 ±11.8
Deterministic	465.2 ±13.9	442.8 ±20.0	532.1 ±15.8	316.2 ±3.5	304.1 ±6.0	355.0 ±7.7
Lexicographic	471.6 ±9.8	442.3 ±17.2	525.5 ±26.1	316.9 ±4.0	306.2 ±7.0	355.0 ±8.3
Random	465.2 ±13.2	443.5 ±17.3	533.8 ±16.9	315.4 ±4.1	304.6 ±5.5	353.8 ±8.5
Value-balanced	462.9 ±14.2	443.7 ±19.3	530.8 ±16.1	314.6 ±4.3	304.8 ±5.7	352.5 ±6.6
Tuple Order						
Alternating	466.2 ±14.0	442.5 ±17.9	527.6 ±18.7	316.6 ±7.9	304.7 ±5.6	349.8 ±6.6
Ascending	466.3 ±14.8	444.6 ±19.1	542.5 ±20.2	317.0 ±7.0	305.4 ±5.5	359.9 ±10.2
Descending	464.9 ±10.3	443.0 ±14.9	527.7 ±8.0	316.0 ±7.4	305.3 ±5.3	349.7 ±6.1
Parameter Order						
Alternating-large	465.1 ±5.1	440.4 ±2.6	522.5 ±8.0	317.8 ±2.4	307.2 ±2.4	348.3 ±3.7
Alternating-small	467.0 ±3.4	442.9 ±3.0	534.7 ±7.9	317.7 ±2.2	308.1 ±2.2	353.2 ±5.1
Ascending	482.8 ±6.3	468.8 ±5.4	550.8 ±20.7	317.6 ±2.6	307.5 ±2.5	360.4 ±9.3
Descending	448.4 ±4.1	421.4 ±3.6	522.4 ±13.9	313.2 ±13.8	297.7 ±5.4	350.7 ±11.4

Table 4.3: Results for MCA experiments

Part II

A New Tool for Combinatorial Test Set Generation

CHAPTER 5

Architecture and Design

Having fast and efficient algorithms available for combinatorial test set generation is only one precursor for an adoption by the wider software testing community. It is almost equally important to provide a user-friendly and accessible interface to the underlying algorithms in order to make them easy to use and lower the entry barrier.

This chapter introduces CAgen, a new tool for combinatorial test set generation, built on top of the Fast-In-Parameter-Order algorithm (FIPO) implementations presented in the previous chapters. We will give an overview into the design decisions and present the technical architecture of the tool.

Several considerations relating to the architecture are based on shortcomings of existing combinatorial testing tools. In the following, the key problems and potential solutions will be discussed. In Chapter 6 these design choices will be compared to existing tools in this area.

The first major decision is that of software distribution: at the end of the software development cycle a method of how to deliver the software artifacts to the end-user must be chosen. Three of the most common approaches are to directly provide the source code, pre-compiled binaries or offer a service over the internet. The first option has the benefit of being the simplest for the developers since the source code is already available as part of the project and must just be made accessible to the user. The drawback to this is that the user needs to have enough technical expertise to download and build the runnable software artifacts and this is not useful for a wide range of users with insufficient technical background. The second option is to provide binary or otherwise directly executable artifacts of the software which can easily be installed by any user for the targeted platforms. Popular examples are downloads via app-stores or package repositories used in many Linux distributions. This method requires more work from the developer as each targeted platform may need different preparations to package the software artifact. For example, supporting multiple Linux distributions requires the developer to prepare the software in different package formats depending on the distribution. For instance, a Debian package is not directly suitable to run on a Redhat-based distribution. The last option, to offer a service reachable over the internet, has seen wide-spread adoption in the last decade. Here, a server provides an accessible interface which can be used by clients (often website-based, accessible via the browser). This has several advantages: most importantly it becomes easier to perform updates, as the server is usually under the control of the vendor and the source code or binary executable is not exposed to the user which might be preferable due to copyright considerations. One disadvantage is that server resources need to be provided which incur operation costs such as the cost for server operation and maintenance.

Another consideration is that of user privacy. As the targeted audience are software testers who might be testing internal services and software, care must be taken that intellectual property rights are not violated by testers using CAgen. Test models used to derive combinatorial test sets might contain sensitive information and it should not leave the testers device.

Lastly, for any tool to be useful, it must be user-friendly and allow the user to do its intended task without unnecessary hurdles.

5.1 Core Algorithms

This section will outline some concrete design decisions of the core `FIPO` algorithms.

The choice of the programming language for implementing the algorithms was guided by the need for a high-performance solution which keeps the memory footprint low. For this purpose, Rust[1] was chosen since it provides several key properties.

The main advantage is that Rust's focus on systems-level programming gives the developer fine-grained control over data structure layout in memory. This control is safe-guarded by the compiler which enforces memory safety and thus prevents errors often occurring in languages such as C and C++. Compared to other memory-safe languages, the enforcement happens entirely at compile-time such that it incurs no additional run-time overhead. The control over the memory layout allows for a faithful implementation of the *coverage-map* data structure as described in Figure 3.4.

The Rust project is structured into three main parts. Figure 5.1 shows the relationships of the different modules. The top-level crate (i.e., a Rust package) is a library named `fipo-core` and contains the implementation of the `FIPO` algorithms and the necessary data structures. The crate is further structured into modules. The base is the `model` module, which contains common data types used by all other modules such as representations for the `Parameter` and `Value` types. The `parser` module provides parsers for common input formats used by other combinatorial testing tools. The `coverage-map` module provides implementations for the *coverage-map* data structure which in turn is

[1]`https://www.rust-lang.org/en-US/`

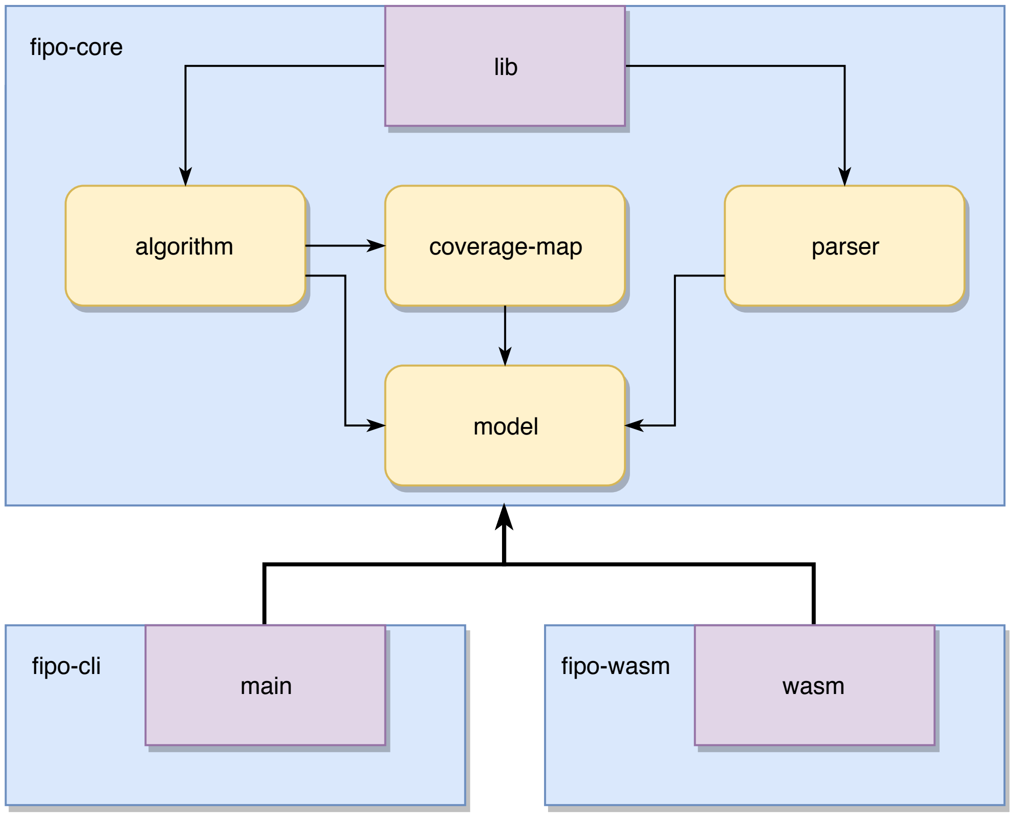


Figure 5.1: Rust project structure of CAgen tool

used by the `algorithm` module to implement the actual algorithm. The `lib` module is the public interface of the crate.

Listing 5.1 depicts the interface of the core `FIPO` algorithm. It provides two methods for the horizontal and vertical extension which each are implemented by specific implementations (`FIPOG`, `FIPOG-F`, etc.). Note that both are parameterized by `T` which is a type-level representation of the strength. Since test set sizes grow exponentially as the strength increases, the parameter t is usually a small integer. We can utilize this and specialize the algorithm for a subset of integers such that the compiler can treat the strength as a constant. This allows for additional optimizations which can be beneficial to the algorithms performance. Some optimizations enabled by this representation include:

1. **Compile-time sized t-selections**: selections take up a constant amount of space in memory and can be tightly packed in memory without necessary meta-data to store the size
2. **Constant iteration bounds**: The loop computing indices for tuples (see definition of *pack* in Section 3.1.1) can be unrolled, eliminating the inner-most loop of the algorithm altogether

The compiler will provide a monomorph instantiation of each method tailored specifically to the choice of strength t. Initially the non-generic method `generate_array` is called, which takes the strength as a normal parameter. Then, the specialized implementation is dynamically selected based on t which now becomes part of the type.

```rust
trait CoreIpoAlgorithm {
    fn extend_horizontal<T>(&mut self, column: usize)
        where T: Unsigned;

    fn extend_vertical<T>(&mut self, column: usize)
        where T: Unsigned;

    // non-generic
    fn generate_array(&mut self, strength: usize) {
        // select implementation with given strength
        // -> strength becomes part of the type
        match strength {
            1 => self.generate_with_strength::<U1>(),
            2 => self.generate_with_strength::<U2>(),
            [..]
            _ => panic!(
                format!("t={}␣not␣implemented", strength)
            ),
        };
    }

    fn generate_with_strength<T>(&mut self)
        where T: Unsigned
    {
        // from here on, the strength is part of the type
    }
}
```

Listing 5.1: Core IPO trait (interface) specification

Since Rust (at the time of writing) does not support compile-time integers, we must fall back to type-level encodings. The `typenum`[2] crate is one popular implementation of this scheme. The `Unsigned` type is the main type defined by this crate for unsigned type-level integers and `U1` and `U2` are types representing 1 and 2 respectively.

[2] https://crates.io/crates/typenum

```
Usage: fipo-cli

Options:
    -h, --help          print help
    -t, --strength      Strength
    -i, --instance      Instance to generate an array for. Can be a
                        path to a ACTS configuration file or a text
                        representation in either linear form
                        (e.g., 2,2,3,2,4,5) or exponential notation
                        (e.g., 2^3,3,4,5)
    -a, --algorithm     Values: ipog, ipog-f, ipog-f2
    -p, --print         Print CA
    -q, --quiet         Print nothing except size
    -r, --header        Output a header row
```

Figure 5.2: Command-line arguments for fipo-cli

5.2 Command Line Interface

A separate crate (`fipo-cli`) is built on top of `fipo-core`. It provides a simple command-line interface to the `FIPO` test generation algorithms. As shown in Figure 5.1, `fipo-cli` only consists of a main module. This module is responsible for parsing command line arguments, reading input files and invoking the correct test generation functions. After test generation, the test set is printed to "stdout" according to the formatting options provided by the users. Figure 5.2 shows the available command line arguments of `fipo-cli`.

In Figure 5.3 an example run of the tool is shown.

The main use-case for the command-line interface is to use it as a test generation service which can be embedded into a larger testing work-flow. The interface is kept simple and only allows for the generation of mixed-level covering arrays (MCAs). As such, no explicit test translation (i.e., using model values from an IPM) is provided by `fipo-cli`, but this is trivial to implement by any simple substitution. As test translation often requires additional steps to convert abstract (CA-level) test sets into concrete test sets it makes sense to separate these two parts.

5.3 Web Frontend

This section introduces CAGEN, a new tool for combinatorial test set generation. It is a web frontend for the core `FIPO` algorithms described previously. The general architecture can be seen in Figure 5.4. The main distinguishing architectural feature is the complete lack of a backend service. The application runs entirely on the client side (i.e., in the user's browser) and only needs a web-server capable of serving static files. The advantages

```
$ fipo-cli --instance 2^4,3,4 --strength 2 --print --header
p0,p1,p2,p3,p4,p5
0,0,0,0,0,0
1,1,1,1,1,0
1,0,1,0,2,0
0,1,0,1,0,1
0,0,1,0,1,1
1,1,0,1,2,1
1,1,1,0,0,2
0,0,0,1,1,2
0,*,*,*,2,2
1,1,1,0,0,3
0,0,0,1,1,3
*,*,*,*,2,3
```

Figure 5.3: Example output of fipo-cli for a $CA(12; 2, 6, (2^4, 3^1, 4^1))$

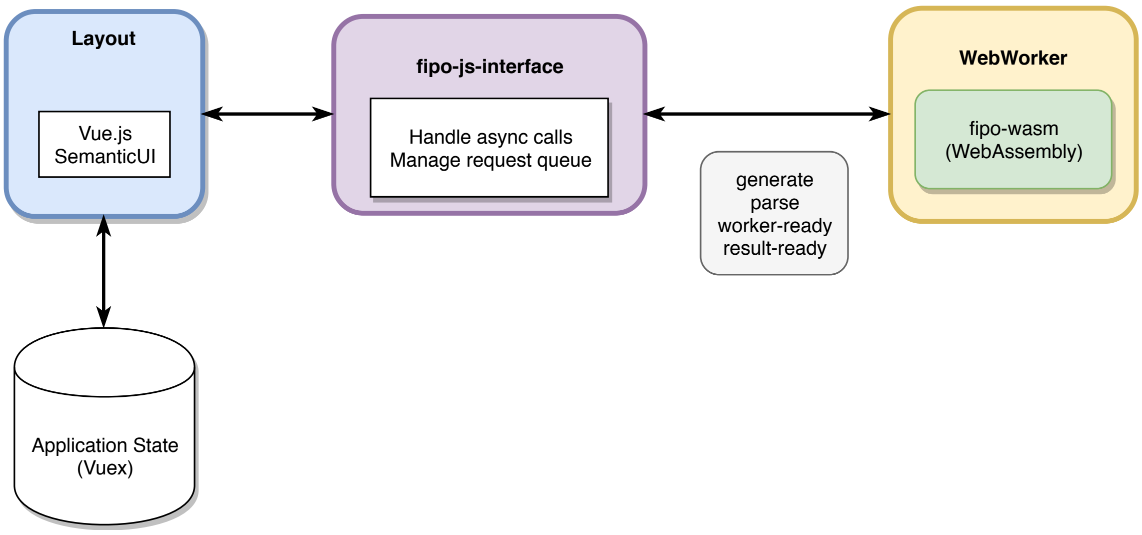


Figure 5.4: Architectural overview of CAgen

of this design will be discussed further in the next chapter. The tool is available for free for anyone to use at `https://matris.sba-research.org/tools/cagen`.

The web frontend is designed as a single-page application utilizing the Vue.js[3] Javascript framework. The layout of the application is written in SemanticUI[4] which provides a wide range of layout options and user interface elements. The application uses Vuex[5] to manage the central application state. The general work-flow is based around a global

[3]`https://vuejs.org/`
[4]`https://semantic-ui.com/`
[5]`https://vuex.vuejs.org/en/intro.html`

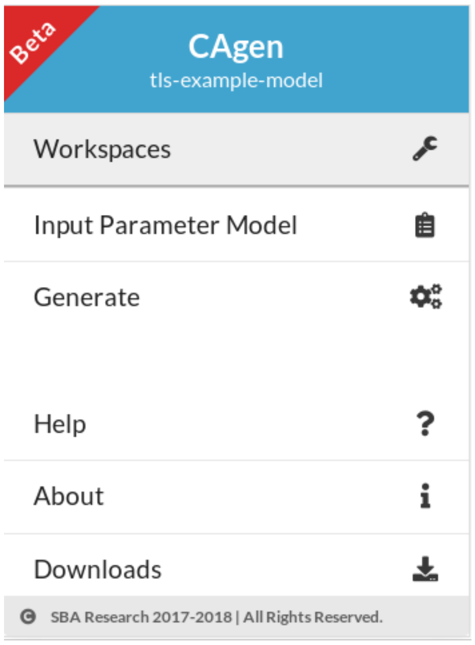


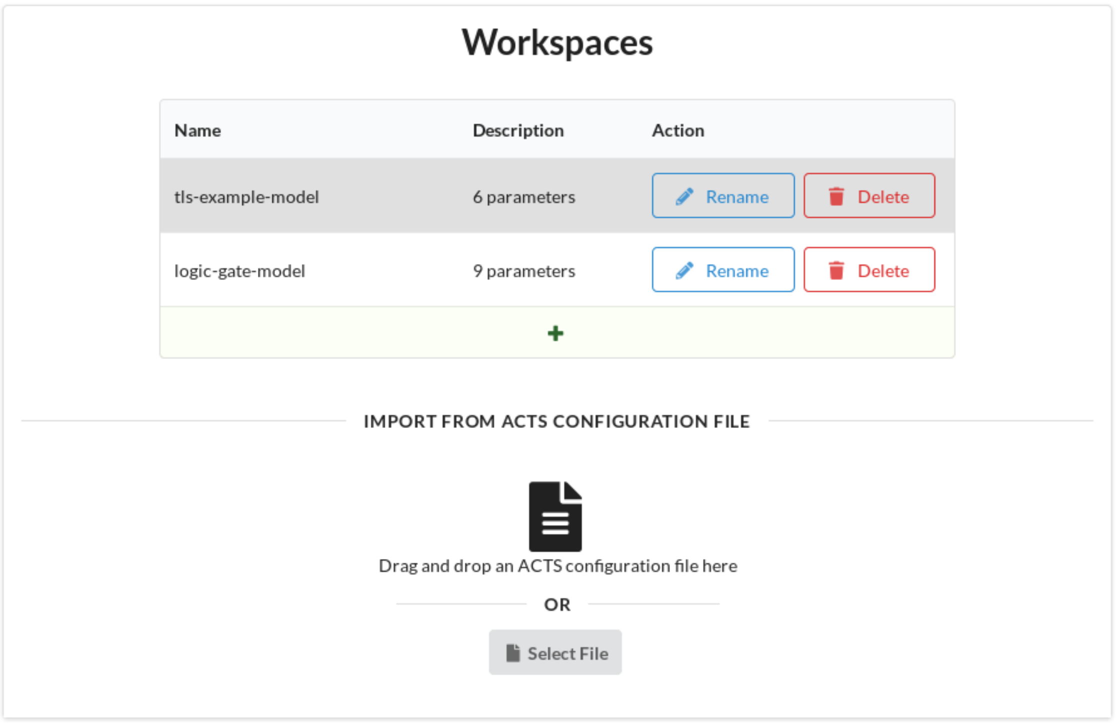
Figure 5.5: Navigation sidebar of CAgen

state object which is mutated via pre-defined actions (mutations). These state changes then trigger an update to the current view. This way, presentation and data manipulation can be cleanly separated. In CAgen, the central state revolves around the management of "workspaces" which are encapsulations of user-defined input parameter models (IPMs).

Users can define new, edit existing and upload IPMs from existing files using a Drag&Drop mechanism. Figures 5.6 and 5.7 show screenshots of these parts of the tool. Parameters can easily be added to the current IPM and different frequently used types are available such as boolean, enumerative and ranged parameters. Additionally, multiple parameters can be added at once by just supplying the number and their domain sizes in exponential notation which is often used for specifying the configuration of an MCA. For example, "$2^4, 3, 6^2$" represents four binary parameter, one ternary parameter and two parameters with six possible values.

Calling Rust via WebAssembly

Rust has support for WebAssembly compilation in its nightly compiler via the "wasm32-unknown-unknown" compilation target. WebAssembly is a standardized binary format for executable code supported by all major browsers. It is intended to be a common compilation target for a variety of source languages such that programs written in them can then directly be executed in the browser. This of course limits the programs being compiled to does which do not rely on platform specific APIs (e.g., access to the file system). In the case of `FIPO`, this is no limitation and as illustrated in Figure 5.1, we

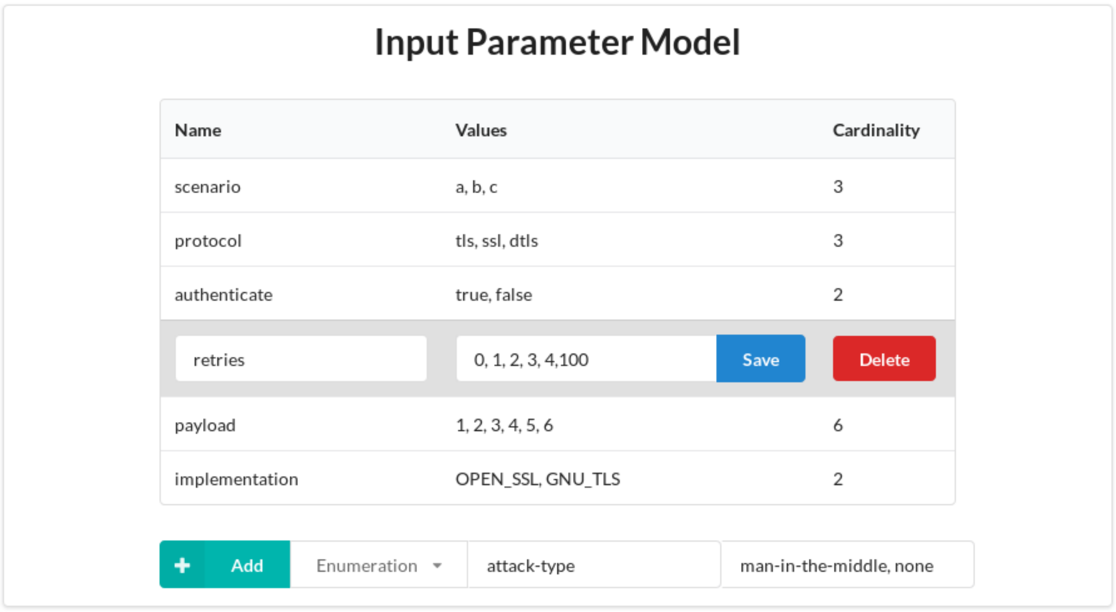


Workspaces

Name	Description	Action
tls-example-model	6 parameters	Rename Delete
logic-gate-model	9 parameters	Rename Delete

+
IMPORT FROM ACTS CONFIGURATION FILE
Drag and drop an ACTS configuration file here
OR
Select File

Figure 5.6: Workspaces encapsulate different input parameter models



Input Parameter Model

Name	Values	Cardinality
scenario	a, b, c	3
protocol	tls, ssl, dtls	3
authenticate	true, false	2
retries	0, 1, 2, 3, 4,100 Save	Delete
payload	1, 2, 3, 4, 5, 6	6
implementation	OPEN_SSL, GNU_TLS	2

Figure 5.7: Parameters can be added, edited, and deleted

simply provide a dedicated crate which exposes an interface to the `fipo-core` library for the web-application. The `fipo-wasm` crate is then compiled to WebAssembly. The crate uses the std-web[6] crate which handles serialization of inputs and de-serialization of outputs to and from the `fipo-core` crate. This results in a binary “fipo-wasm.wasm” file and a correspondingly generated Javascript runtime which takes care of parsing of arguments and results as well as some general purpose initializations.

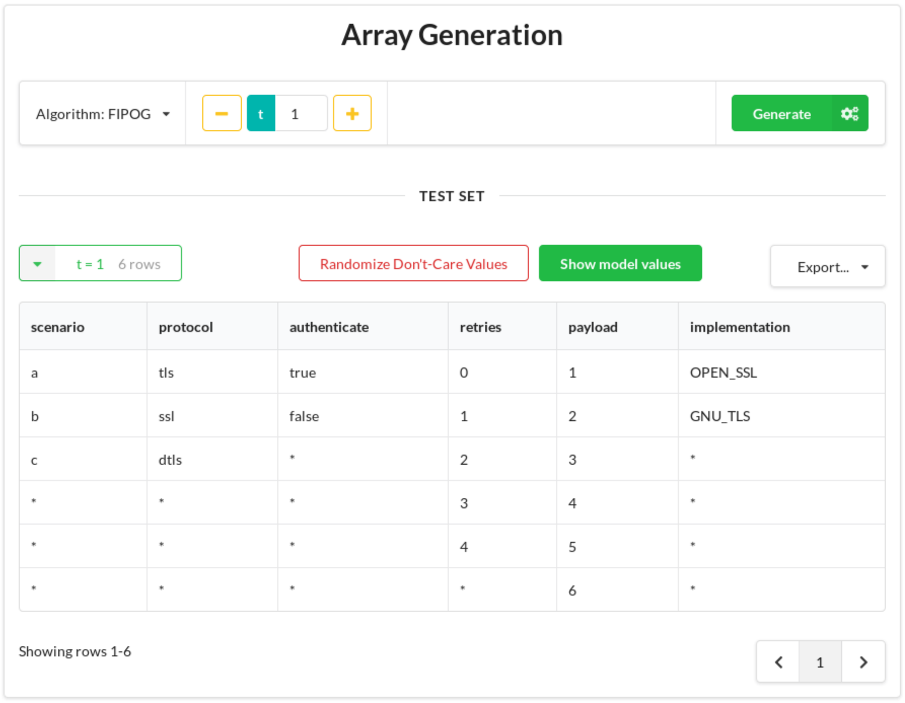
In the web-application, the exposed wasm functions are wrapped in a WebWorker which allows us to execute the test generation algorithms in the background. This is necessary, since test generation can take up a lot of time and the user interface should not be blocked during the computation. A Javascript interface abstracts over the communication with the WebWorker (which is based on message-passing) in order to provide a usable API to the web-application. The interface handles test generation request asynchronously and queues additional request.

Test Generation

The test generation feature is CAgen is shown in Figure 5.8, 5.9, 5.10 and 5.11.

The user can request the generation of arrays of any strength ($t \leqslant k$) and choose the desired algorithm. Once a test set is generated it is shown in a table. The view of the test set can be customized depending on the use-case. For example, one can toggle between viewing the underlying MCA or choose to translate the entries with the values from the IPM. *Don't-care* values can optionally be randomized in order to select a concrete value instead of a wildcard. Test sets can also be exported, currently there is support for CSV and Matlab since they are two of the most common used means for further processing. The test sets can be downloaded as a file or be copied into the clipboard.

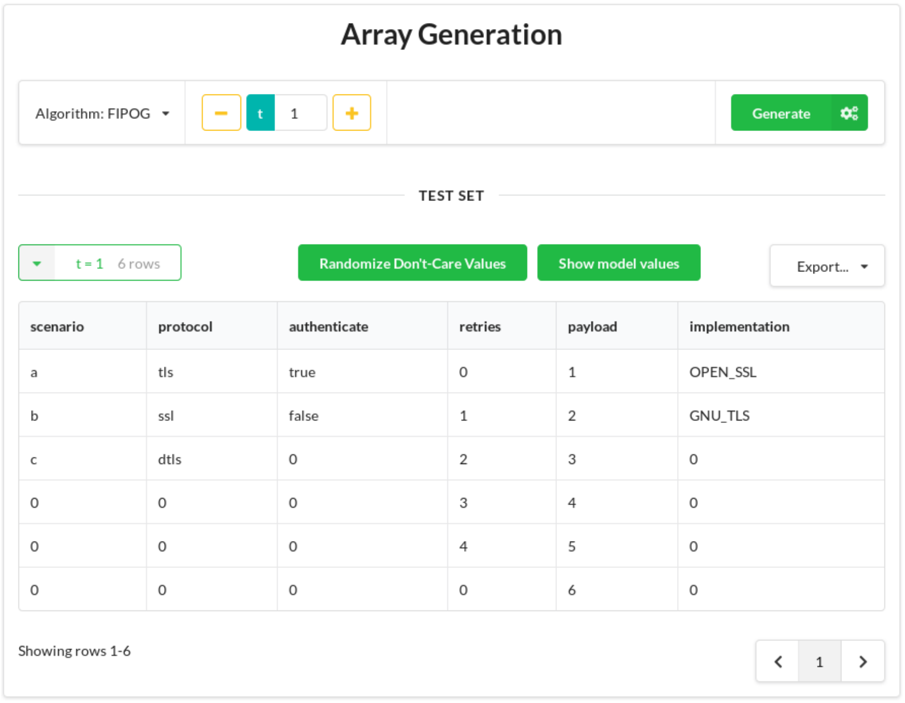
[6] `https://docs.rs/stdweb/*/stdweb/`



scenario	protocol	authenticate	retries	payload	implementation
a	tls	true	0	1	OPEN_SSL
b	ssl	false	1	2	GNU_TLS
c	dtls	*	2	3	*
*	*	*	3	4	*
*	*	*	4	5	*
*	*	*	*	6	*

Showing rows 1-6

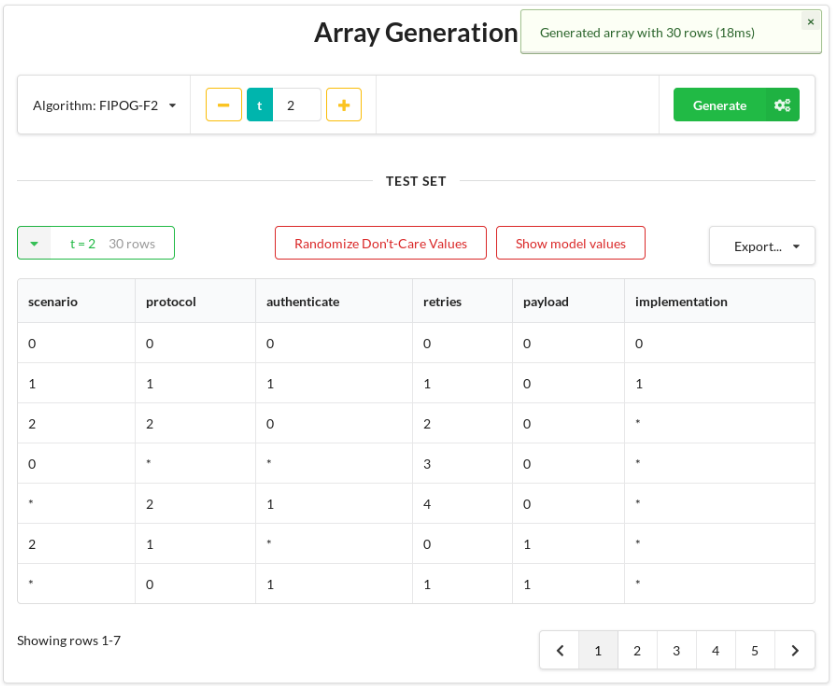
Figure 5.8: Arrays can generated with configurable strength and algorithm



scenario	protocol	authenticate	retries	payload	implementation
a	tls	true	0	1	OPEN_SSL
b	ssl	false	1	2	GNU_TLS
c	dtls	0	2	3	0
0	0	0	3	4	0
0	0	0	4	5	0
0	0	0	0	6	0

Showing rows 1-6

Figure 5.9: Array is manipulatable after generation


Array Generation

Generated array with 30 rows (18ms)

Algorithm: FIPOG-F2 — t 2 — Generate

TEST SET

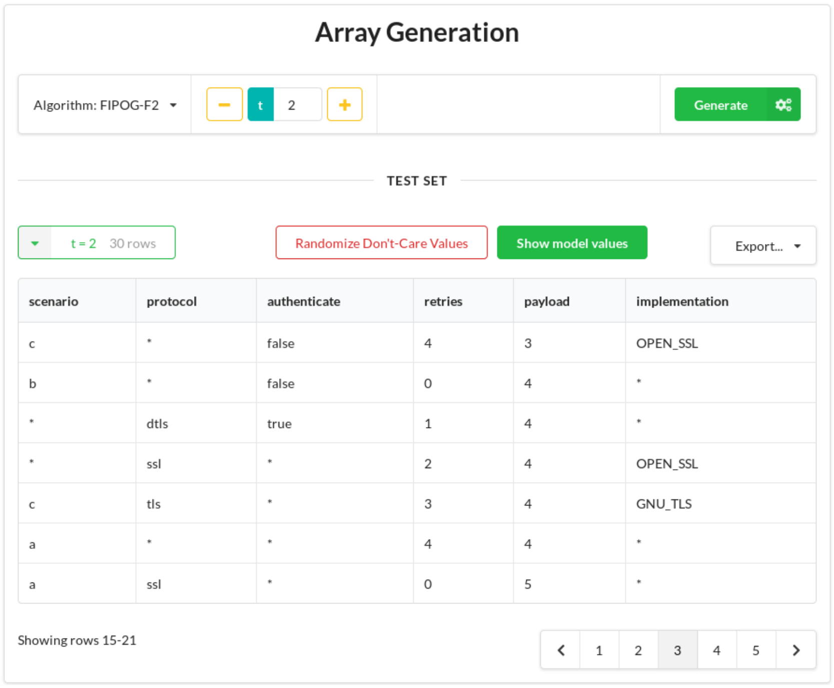
t = 2 30 rows — Randomize Don't-Care Values — Show model values — Export...

scenario	protocol	authenticate	retries	payload	implementation
0	0	0	0	0	0
1	1	1	1	0	1
2	2	0	2	0	*
0	*	*	3	0	*
*	2	1	4	0	*
2	1	*	0	1	*
*	0	1	1	1	*

Showing rows 1-7

< 1 2 3 4 5 >

Figure 5.10: Generated MCA (with no translated model values)


Array Generation

Algorithm: FIPOG-F2 — t 2 — Generate

TEST SET

t = 2 30 rows — Randomize Don't-Care Values — Show model values — Export...

scenario	protocol	authenticate	retries	payload	implementation
c	*	false	4	3	OPEN_SSL
b	*	false	0	4	*
*	dtls	true	1	4	*
*	ssl	*	2	4	OPEN_SSL
c	tls	*	3	4	GNU_TLS
a	*	*	4	4	*
a	ssl	*	0	5	*

Showing rows 15-21

< 1 2 3 4 5 >

Figure 5.11: Same array, but with model values

CHAPTER 6

Evaluation

This chapter will evaluate and discuss the design of CAgen presented in the last chapter. A comparison with other testing tools in this area provides context and justifications for the architecture.

6.1 Related Combinatorial Testing Tools

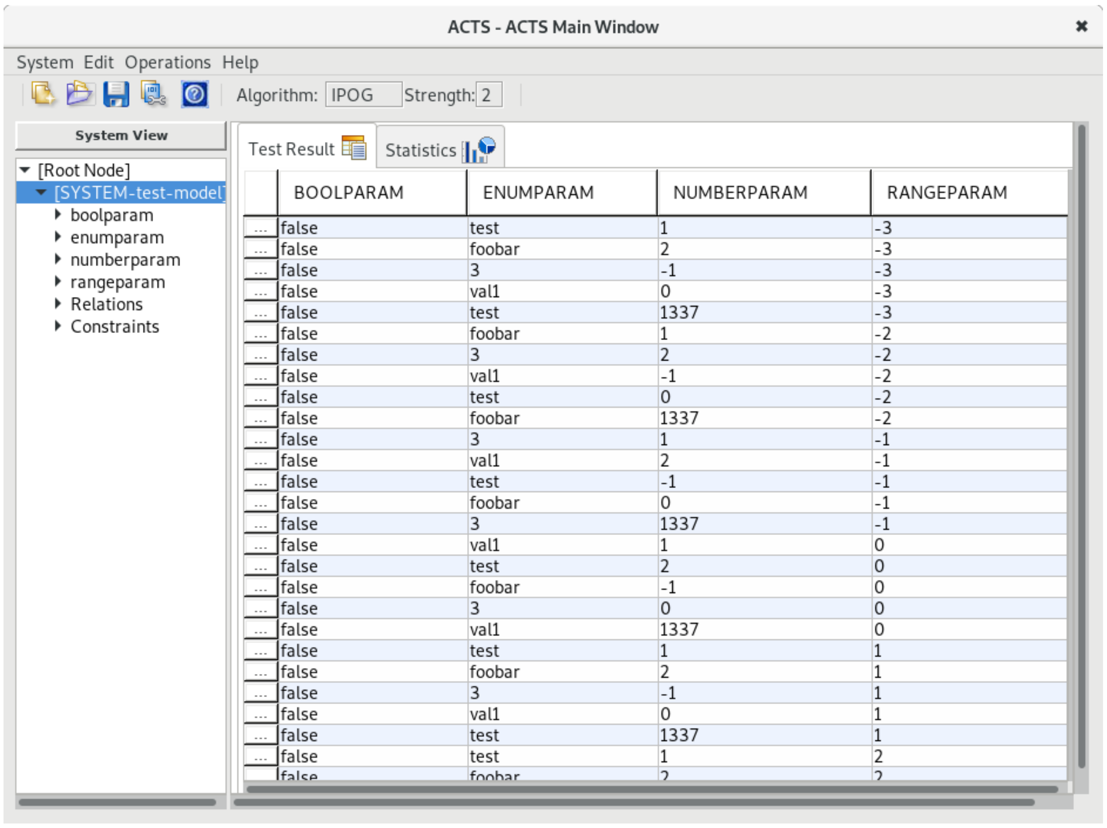
There exist a wide variety of combinatorial testing tools[1], however most of them are not suitable for wide-spread use as they are either only available commercially or are limited in their feature-set. Many tools only support pair-wise test set generation which is often not enough for thorough combinatorial testing.

In this evaluation, we will focus on a free tool called ACTS which is the most well-known implementation of the IPO family of algorithms and a recent effort of bringing combinatorial testing to the web named CTWedge.

One of the most widely used combinatorial testing tools[2] is ACTS (Automated Combinatorial Testing for Software) [YLKK13] with more than 3000 users as of 2018. ACTS is written in Java and provides a GUI as well as a command line interface. It is freely available upon request from the authors. It was first released in 2006 and supports many test generation algorithms. Most importantly, ACTS implements `IPOG`, `IPOG-F` and `IPOG-F2`. Additionally, implementations of `IPOG-D`, base-choice testing and PaintBall, a random test generation algorithm, are provided. ACTS furthermore supports user-specified constraints and generation of mixed-strength arrays. These arrays can have varying interaction coverage between explicitly defined relations of parameters. This can be used to designate more "important" sets of parameters which can be tested at a

[1]`http://www.pairwise.org/tools.asp`

[2]`https://csrc.nist.gov/Projects/Automated-Combinatorial-Testing-for-Software`



Algorithm: IPOG Strength: 2

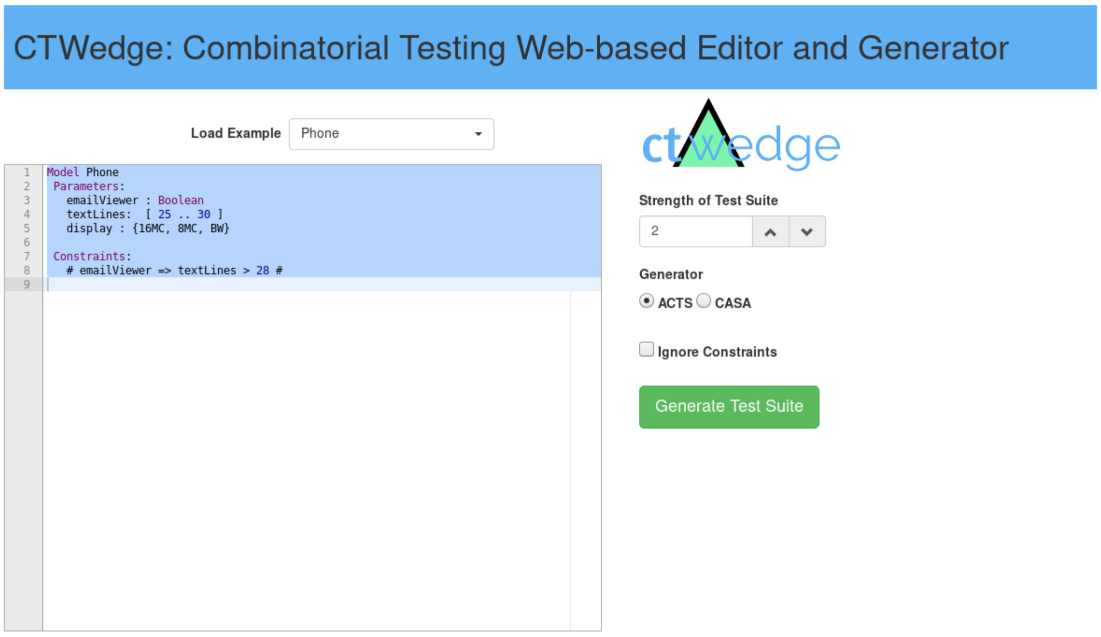
	BOOLPARAM	ENUMPARAM	NUMBERPARAM	RANGEPARAM
...	false	test	1	-3
...	false	foobar	2	-3
...	false	3	-1	-3
...	false	val1	0	-3
...	false	test	1337	-3
...	false	foobar	1	-2
...	false	3	2	-2
...	false	val1	-1	-2
...	false	test	0	-2
...	false	foobar	1337	-2
...	false	3	1	-1
...	false	val1	2	-1
...	false	test	-1	-1
...	false	foobar	0	-1
...	false	3	1337	-1
...	false	val1	1	0
...	false	test	2	0
...	false	foobar	-1	0
...	false	3	0	0
...	false	val1	1337	0
...	false	test	1	1
...	false	foobar	2	1
...	false	3	-1	1
...	false	val1	0	1
...	false	test	1337	1
...	false	test	1	2
	false	foobar	2	2

Figure 6.1: Screenshot of ACTS (GUI)

higher strength than the rest of the system. Figure 6.1 shows the main test generation window of ACTS.

Recently, the authors in [GR18] surveyed existing combinatorial test generation tools available as web applications. Their analysis shows that available solutions leave a lot of room for improvement. Specifically, three out of the surveyed five tools only offer pair-wise test generation, one supports of to 3-way and one up to 6-way coverage. Furthermore, all but two of the tools are freely available since the others are commercialized and require a subscription or license. To address these shortcomings, the authors present CTWedge[3] (Combinatorial Testing Web EDiting and GEneration), a tool built as a System as a Service (SaaS) solution. It provides a web frontend for input parameter model editing using Xtext and Ace (a Javascript-based editor). Figure 6.2 shows a screenshot of CTWedge. The editor provides syntax highlighting as well as auto completion and points out errors in the model. Syntax-level errors are detected and additional semantic validation routines check for errors such as invalid range specifications or duplicate parameter values. The frontend is supported by a REST service which handles auto completion and test set generation. The test generation is provided by external tools. In

[3]`http://foselab.unibg.it/ctwedge/`



CTWedge: Combinatorial Testing Web-based Editor and Generator

Load Example: Phone

```
Model Phone
 Parameters:
   emailViewer : Boolean
   textLines:  [ 25 .. 30 ]
   display : {16MC, 8MC, BW}

 Constraints:
   # emailViewer => textLines > 28 #
```

Strength of Test Suite: 2

Generator: [✓ ACTS] CASA

Ignore Constraints: ☐

Figure 6.2: Screenshot of CTWedge

its current version CTWEDGE supports test generation using the ACTS and CASA tools. The REST service handles requests to generate a test set by synchronously invoking the corresponding tool and returning the resulting test set. Currently, the algorithm used by ACTS is fixed and can not be configured. `IPOG` is used when ACTS is selected as test generation backend.

6.2 Comparison

All three tools presented in the previous sections (ACTS, CTWEDGE and CAGEN) support the same basic use-case. Given an input parameter model and strength t, all generate a combinatorial test set. There are however differences which will be discussed in the following paragraphs.

In Table 6.1, the main features provided by all tools are listed together with an overview of which tool supports which subset. ACTS supports the generation of test sets only up to strength 6. This is mainly a limitation in the GUI as the command-line version supports arbitrary strengths. CTWEDGE and CAGEN don't impose any limits on the strength, however, CTWEDGE allows to choose a strength larger than the number of parameters k which does not make sense since $t = k$ results in the exhaustive test set.

ACTS and CAGEN both provide an IPM editor which is based on GUI elements such as tables and buttons while CTWEDGE uses an interactive text editor. All tools support the test set export as comma-separated values (CSV) files, while ACTS also allows to

Feature	ACTS	CTWedge	CAgen
Max t	6	unlimited	unlimited
IPM editor	GUI	Text Editor	GUI
Constraints	✓	✓	✗
Mixed-strength	✓	✗	✗
Show MCA	✗	✗	✓
Export format	CSV, NIST, Excel	CSV	CSV, Matlab
Technology	Java	Java & JS	Rust, WebAssembly & JS
Distribution	Binary	SaaS	Client-side
Privacy	✓	✗	✓

Table 6.1: Feature support

export in the so-called NIST format and as an Excel file. CAgen additionally allows direct export to Matlab compatible matrices.

CAgen is missing some features compared to ACTS such as support for constraints. This is due to the tool being in an early stage of development and are set to be implemented at a later time.

Performance

The main difference between the three tools is the test generation time. Since CAgen uses the FIPO algorithms described in Chapter 3 and 5, the performance comparison to ACTS of Section 3.4 also applies here: Test generation times are drastically lower (sometimes by orders of magnitude) and also the memory footprint is kept small which is an important consideration when running in the browser.

CTWedge uses ACTS as a backend, so it automatically inherits the same performance characteristics in that case.

Distribution

A big difference lies in the manner in which the tools are distributed. ACTS is only available as a packaged binary (.jar file) and requires a locally installed Java run-time environment. CTWedge is available through any web-browser, but requires the vendor to provide sufficient server resources to handle test generation requests. Since generation times can be in the order of hours or even days (e.g., when using ACTS as a backend for large models), it is infeasible to host this service for an extended user base. CAgen does not suffer from this problem since the application is entirely client-side. The user itself is providing the computing capabilities to generate the test sets and can influence the performance by using a more powerful machine. Such scaling is not possible when using a SaaS-backed solution.

Privacy

Privacy in this context of software testing refers to the way in which details about the testing process are exposed to third-parties. If the system under test is modelled such that internal details are inferable from it, the tester might prefer to use tools which do not require handing over of these details. Some companies might even require that only local tools are used which do not communicate with external services due to legal issues or copyright considerations.

ACTS and CAgen both run locally and the processed input models are not send to any external service. Although CAgen can be hosted as an internet-accessible website it is strictly client-side and could just as well be hosted locally on the testers machine.

Due to its design, CTWedge however requires to send the IPM to the server such that a test set can be generated.

6.3 Conclusion

As shown in the preceding comparison, the client-side computation model supported by WebAssembly has major benefits compared to existing approaches. Since any user with a web-browser can access the tool, accessibility is drastically improved. By not requiring additional computing resources compared to SaaS-based solutions, the costs for the vendor can essentially be eliminated and it furthermore offers favorable properties regarding user privacy.

Since high-performance implementations written in Rust can be reused without rewriting the core library in JavaScript (formerly the only supported language in all browsers), this architecture also imposes minimal overhead in terms of development. Since the result is a static website, the cost of server monitoring and maintenance is eliminated completely.

CHAPTER 7

Conclusion of the Thesis

This thesis has explored the field of combinatorial testing, specifically the challenges of combinatorial test generation using the In-Parameter-Order family of algorithms. The motivation for combinatorial testing methods was grounded in the need for effective and efficient test sets. Having such test sets available which offer input-space coverage guarantees can help to provide better software quality by reducing the risk that remains after testing resulting from untested parts of the input-space. One particular family of algorithms for generating these test sets, namely the In-Parameter-Order family, was examined from both a high-level (to aid understanding of its general workings) as well as from a low-level view. The latter analyzed the necessary data structures and sub-procedures needed to implement the algorithms.

The main part examined in detail how to design an efficient implementation of the In-Parameter-Order family. To this end, a central data structure named *coverage-map* was introduced which allows for memory-efficient tuple enumeration and coverage-tracking. Then, several novel optimizations were presented which exploit structural properties of covering arrays to prune the search space and avoid unnecessary work.

The presented optimizations to the algorithms in general improve the performance greatly with up to 13 times faster generation times compared to the baseline. Furthermore, the developed `FIPOG` implementation represents a significant improvement in terms of performance over ACTS, one of the most widely used IPO implementations, with test generation time reductions of up to a factor of 145. As part of the optimization work, the complexity of selecting the best value to cover as part of the horizontal extension was improved upon compared to the previously reported result. The presented data structures succeeds at keeping memory demands low, even for sizeable instances where many tuples need to be enumerated and the resulting arrays become large.

These results are promising also for practical applications in the field of combinatorial testing. Reduced test generation time can lead to much reduced testing cycles allowing faster and better quality assurance as both iteration time during development and response time of testing a production system can be lowered significantly. These improvements are all achieved with no sacrifice of the quality of produced covering arrays as the results remain unchanged by the optimizations.

Additionally, we have studied the impact of tie-breaking, parameter ordering and tuple enumeration order in the IPO family of algorithms. We have compared their effectiveness in terms of their ability to reduce covering array sizes in a large case study. In summary, `IPOG-F` overall manages to produce the smallest arrays compared to `IPOG` and `IPOG-F2`. The most surprising result is that the choice of tie-breaker seems to not matter a great deal when averaging over all instances. While some tie-breakers perform better than others on specific instances, there is none which is strictly better to a significant degree. This is contrary to previous work in the field which concluded the opposite albeit judging from a limited set of benchmark instances. In the case of MCA generation, we measured the largest reduction in array size when ordering columns by decreasing alphabet size, with up to 12% reduction in size compared to the mean.

Lastly, we presented a new tool for combinatorial test set generation based upon the efficiently implemented algorithms. The tool is available for free for anyone to use as it can run standalone as a client-side web application. The novelty lies in its ease of use, accessibility as well as its speed. This tool can assist software testers to quickly iterate on their models to improve overall testing quality.

Future Work

As future work we envision three main areas. The first is to extend the core algorithms with additional features such as support for constraints. Constraints in the IPM are very common when applied in real-world testing and it will be important to implement this feature with degrading performance more than absolutely necessary. The second area is about improving the CAgen tool even further with features that help software testers successfully apply combinatorial testing. The entry barrier should be as low as possible in order to further establish CT as an accepted testing technique. Additional features include the possibility to extend existing test sets to combinatorial test sets as well as a simple constraint editor which visualizes the model such that interactions between multiple constraints are clearly communicated to the user. As the third and last area we want to explore the applicability of our improvements to the IPO family to other covering array generation algorithms such as AETG.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AGTJH12a] Himer Avila-George, Jose Torres-Jimenez, and Vicente Hernández. New bounds for ternary covering arrays using a parallel simulated annealing. *Mathematical Problems in Engineering*, 2012, 2012.

[AGTJH12b] Himer Avila-George, Jose Torres-Jimenez, and Vicente Hernández. Parallel simulated annealing for the covering arrays construction problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1–7, 2012.

[AO94] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 69–79, 1994.

[BC07] Renée C Bryce and Charles J Colbourn. The density algorithm for pairwise interaction testing. *Software Testing Verification and Reliability*, 17(3):159–182, 2007.

[BC09] Renée C Bryce and Charles J Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability*, 19(1):37–53, 2009.

[BCC05] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 146–155. ACM, 2005.

[BGSW15] Josip Bozic, Bernhard Garn, Dimitris E Simos, and Franz Wotawa. Evaluation of the ipo-family algorithms for test case generation in web security testing. In *2015 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10. IEEE, 2015.

[BMTI10] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126. Springer Berlin Heidelberg, 2010.

[BRTJRT09] Josue Bracho-Rios, Jose Torres-Jimenez, and Eduardo Rodriguez-Tello. A new backtracking algorithm for constructing binary covering arrays of variable strength. In *MICAI 2009: Advances in Artificial Intelligence*, pages 397–407. Springer Berlin Heidelberg, 2009.

[Bus52] Kenneth A Bush. Orthogonal arrays of index unity. *The Annals of Mathematical Statistics*, pages 426–434, 1952.

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The automatic efficient test generator (aetg) system. In *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pages 303–309, 1994.

[CG09] A. Calvagna and A. Gargantini. Ipo-s: Incremental generation of combinatorial interaction test data based on symmetries of covering arrays. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 10–18, 2009.

[Che07] Christine T. Cheng. The test suite generation problem: Optimal instances and their implications. *Discrete Applied Mathematics*, 155(15):1943 – 1957, 2007.

[CK02] M Chateauneuf and Donald L Kreher. On the state of strength-three covering arrays. *Journal of Combinatorial Designs*, 10(4):217–238, 2002.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[Col04] Charles J. Colbourn. Combinatorial aspects of covering arrays. *Le Mathematiche*, LIX(I-II):125–172, 2004.

[DLY$^+$15] Feng Duan, Yu Lei, Linbin Yu, Raghu N Kacker, and D Richard Kuhn. Improving ipog's vertical growth based on a graph coloring scheme. In *2015 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–8. IEEE, 2015.

[DN84] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, 1984.

[FLL$^+$08] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287, 2008.

[Gar17] Inc. Garnter. Leading the IoT. `https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf`, 2017. Accessed: 2018-04-22.

[GCD09] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 13–22. IEEE, 2009.

[GCD11] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[GHRVTJ12] Loreto Gonzalez-Hernandez, Nelson Rangel-Valdez, and Jose Torres-Jimenez. Construction of mixed covering arrays of strengths 2 through 6 using a tabu search approach. *Discrete Mathematics, Algorithms and Applications*, 4(03):1250033, 2012.

[GLD$^+$14] Shiwei Gao, Jianghua Lv, Binglei Du, Yaruo Jiang, and Shilong Ma. General optimization strategies for refining the in-parameter-order algorithm. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 21–26. IEEE, 2014.

[GLD$^+$15] Shi-Wei Gao, Jiang-Hua Lv, Bing-Lei Du, Charles J Colbourn, and Shi-Long Ma. Balancing frequencies and fault detection in the in-parameter-order algorithm. *Journal of Computer Science and Technology*, 30(5):957–968, 2015.

[GO07] Mats Grindal and Jeff Offutt. Input parameter modeling for combination strategies. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 255–260. ACTA Press, 2007.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[GR18] Angelo Gargantini and Marco Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018. to appear.

[GS17] Bernhard Garn and Dimitris E. Simos. Algebraic modelling of covering arrays. In *Applications of Computer Algebra*, pages 149–170. Springer International Publishing, 2017.

[Har05] Alan Hartman. Software and hardware testing using combinatorial covering suites. In MartinCharles Golumbic and IrithBen-Arroyo Hartman, editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US, 2005.

[HPS$^+$04] Brahim Hnich, Steven Prestwich, Evgeny Selensky, et al. Modeling the covering test problem. In *CSCLP 2004: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, 2004.

[HR04] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1):149–156, 2004.

[KBD$^+$15] D Richard Kuhn, Renee Bryce, Feng Duan, Laleh Sh Ghandehari, Yu Lei, and Raghu N Kacker. Combinatorial testing: Theory and practice. *Advances in Computers*, 99:1–66, 2015.

[KKL13] D Richard Kuhn, Raghu N Kacker, and Yu Lei. *Introduction to combinatorial testing.* CRC press, 2013.

[KS] Ludwig Kampel and Dimitris E. Simos. A survey on the state of the art of complexity problems for covering arrays. under review.

[KS98] Donald L Kreher and Douglas R Stinson. *Combinatorial algorithms: generation, enumeration, and search*, volume 7. CRC press, 1998.

[KS16] Ludwig Kampel and Dimitris E. Simos. Set-based algorithms for combinatorial test set generation. In Franz Wotawa, Mihai Nica, and Natalia Kushik, editors, *Testing Software and Systems*, pages 231–240. Springer International Publishing, 2016.

[KS17] Kristoffer Kleine and Dimitris E Simos. Coveringcerts: Combinatorial methods for x. 509 certificate testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 69–79. IEEE, 2017.

[KSTJV15] Paris Kitsos, Dimitris E Simos, Jose Torres-Jimenez, and Artemios G Voyiatzis. Exciting fpga cryptographic trojans using combinatorial testing. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 69–76. IEEE, 2015.

[KWG04] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[LETJRTRV08] Daniel Lopez-Escogido, Jose Torres-Jimenez, Eduardo Rodriguez-Tello, and Nelson Rangel-Valdez. Strength two covering arrays construction using a sat representation. In *Mexican International Conference on Artificial Intelligence*, pages 44–53. Springer, 2008.

[LKK$^+$07] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog: A general strategy for t-way software testing. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, pages 549–556. IEEE, 2007.

[LKK$^+$08] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[LT98] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 254–261. IEEE, 1998.

[Mal95] Y.K. Malaiya. Antirandom testing: getting the most out of black-box testing. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 86–95, Oct 1995.

[ND12] Srinivas Nidhra and Jagruthi Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[Nur04] Kari J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138(1):143 – 152, 2004.

[Slo93] Neil JA Sloane. Covering arrays and intersecting codes. *Journal of combinatorial designs*, 1(1):51–63, 1993.

[Sta01] John Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, 2001.

[TA00] Yu-Wen Tung and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484)*, volume 1, pages 431–437, 2000.

[Tas02] G. Tassey. *The economic impacts of inadequate infrastructure for software testing.* National Institute of Standards and Technology, 2002.

[TJIM13] Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez. Survey of covering arrays. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 20–27. IEEE, 2013.

[TJRT12] Jose Torres-Jimenez and Eduardo Rodriguez-Tello. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1):137 – 152, 2012.

[Tri18] Tricentis. Software Fail Watch 5th Edition. `https://www.tricentis.com/software-fail-watch/`, 2018. Accessed: 2018-04-23.

[Wil00] Alan W. Williams. *Determination of Test Configurations for Pair-Wise Interaction Coverage*, pages 59–74. Springer US, Boston, MA, 2000.

[YDL$^+$15] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–9, 2015.

[YLKK13] Linbin Yu, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375. IEEE, 2013.

[YLN$^+$13] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 242–251, 2013.

[YZ06] J. Yan and J. Zhang. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 385–394, 2006.

[YZ11] Mohammed I Younis and Kamal Z Zamli. Mipog-an efficient t-way minimization strategy for combinatorial testing. *International Journal of Computer Theory and Engineering*, 3(3):388, 2011.