2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) | 979-8-3315-3467-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSTW64639.2025.10962457

# A Search-Based Benchmark Generator for Constrained Combinatorial Testing Models

Paolo Arcaini
National Institute of Informatics
Tokyo, Japan
arcaini@nii.ac.jp

Andrea Bombarda
University of Bergamo
Bergamo, Italy
andrea.bombarda@unibg.it

Angelo Gargantini
University of Bergamo
Bergamo, Italy
angelo.gargantini@unibg.it

***Abstract*—In combinatorial testing, the generation of benchmarks that meet specific constraints and complexity requirements is essential for the rigorous assessment of testing tools. In previous work, we presented BENCIGEN, a benchmark generator for combinatorial testing, which had the limitation of often discarding input parameter models (IPMs) that fail to meet the targeted requirements in terms of ratio (i.e., the number of valid tests, or tuples, over the total number of possible tests or tuples) and solvability. This paper presents an extension to BENCIGEN, BENCIGEN$_S$, that integrates a search-based generation approach, aimed at reducing the number of discarded IPMs and enhancing the efficiency of benchmark generation. Instead of rejecting IPMs that do not fulfill the desired characteristics, BENCIGEN$_S$ employs search-based techniques that iteratively mutate IPMs, optimizing them accordingly to a fitness function that measures their distance from target requirements. Experimental results demonstrate that BENCIGEN$_S$ generates a significantly higher proportion of benchmarks adhering to specified characteristics, with sometimes a reduced computation time. This approach not only improves the generation of suitable benchmarks but also enhances the tool's overall effectiveness.**

***Index Terms*—Combinatorial testing, Benchmarks, Test generators, Solvable IPMs, Test Validity Ratio, Tuple Validity Ratio**

## I. INTRODUCTION

Combinatorial Interaction Testing (CIT) [28] has become a prominent research focus in recent years, demonstrating high effectiveness for testing complex systems with numerous inputs or configuration settings. CIT's primary goal is to assist testers in detecting defects arising from interactions between different inputs or parameters by systematically examining these interactions. The approach ensures that every possible combination of $t$ parameters (where each parameter takes on one of its possible values) is covered at least once [20], [27]. In practice, testers define an input parameter model (IPM) for the system under test (SUT), specifying possible values for each parameter along with any constraints between values of different parameters. Then, a test generator uses this model to produce a comprehensive test suite.

Over the years, several test generators have been proposed.[1] Though numerous algorithms and tools for combinatorial interaction testing (CIT) have been developed to advance software system testing, surprisingly little emphasis has been placed on rigorously testing and evaluating these CIT tools themselves in a systematic and fair manner. A key challenge lies in the lack of a comprehensive benchmark suite that could be used to verify the accuracy and measure the performance of these generators. Such benchmarks could be used to assess the performance of test generators (time required for test generation and test suite size), as well as to drive their improvement (e.g., including different parameter or constraint types). Often, CIT tools are evaluated on small examples or models that fail to capture the complexity of real-world scenarios, missing crucial characteristics of common testing challenges. For example, many of the tools evaluations were performed by not considering IPMs containing constraints, which represent the greatest challenge for test generators.

The relevance of this research area has been confirmed by the CT-Competition, a competition among combinatorial test generators [6], [22], held yearly during the International Workshop on Combinatorial Testing (IWCT). For the competition, benchmarks are generated by the BENCIGEN [9] tool, which is set to randomly generate sets of IPMs belonging to defined categories, i.e., the *tracks*, and having specific characteristics and complexities. In [9], it has been shown that BENCIGEN generates realistic benchmarks, meeting the same characteristics and presenting the same features of real-world or industrial models. However, BENCIGEN implements a highly inefficient benchmark generation when it comes to constraints definition and constrained instances. When the generated IPM contains too strict constraints, preventing any test from being generated from it, BENCIGEN discards it and starts generating a new IPM from scratch. Similarly, if the user asks the tool to generate IPMs with a precise ratio, i.e., the percentage of valid tests over the total possible tests or valid tuples over the total possible tuples (see Section II), and the generated models do not meet this requirement, they are discarded, and new ones are generated.

Thus, in this paper, we propose BENCIGEN$_S$, an extension of BENCIGEN which implements search-based strategies [23], [25] for repairing non-compliant IPMs. More specifically, the extension we propose in this paper for BENCIGEN involves a set of *mutations* that are applied to the generated non-compliant

The work of Andrea Bombarda is supported by PNRR - ANTHEM (AdvaNced Technologies for Human-centrEd Medicine) - Grant PNC0000003 – CUP: B53C22006700001 - Spoke 1 - Pilot 1.4. Paolo Arcaini is supported by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai.

[1]For a non-exhaustive list of tools see https://www.pairwise.org/tools.html.

IPM to generate new *individuals*. These mutations can evolve by removing a constraint, adding a parameter or a constraint, substituting a constraint with a new one, and so on. Afterward, BenCIGen evaluates the fitness *fit* of each individual by calculating the distance between the desired ratio and the one obtained by that individual, or the number of unsatisfied constraints. Since *fit* has to be minimized, individuals with lower fitness values are selected for the next generation. During the search process, $\text{BenCIGen}_S$ ensures that IPMs retain the desired characteristics, such as maintaining a specific number of parameters and constraints within the user-defined minimum and maximum bounds.

We evaluate the proposed enhancement of $\text{BenCIGen}_S$ with respect to the approach previously implemented by BenCIGen. Our experiments show that the implemented search-based strategy allows for improving tool's performance when generating combinatorial IPMs.

The paper is structured as follows. Section II reports the background on combinatorial testing and on the measures we use in our fitness evaluation, while Section III briefly describes the procedure used by BenCIGen in its regular implementation. The contribution of this paper, namely the search-based approach $\text{BenCIGen}_S$ for the generation of IPMs compliant with all requirements is presented in Section IV and evaluated in Section V. Section VI discusses potential threats to the validity of our findings. Finally, Section VII and Section VIII, respectively, report related work and conclude the paper.

## II. Background

Combinatorial test generators are tools designed to create test suites for systems modeled with an Input Parameter Model (IPM). An IPM defines the parameters of a system under test (SUT), the range of possible values for each parameter, and any constraints that exist between different parameter values. In this paper, we consider the following formal definition.

*Definition 1 (Input Parameter Model):* Let $S$ be the system under test, $P = \{p_1, \ldots, p_n\}$ be a set of $n$ parameters, where every parameter $p_i$ assumes values in the domain $D_i = \{v_1^i, \ldots, v_j^i\}$. Let $D$ be the set of all the $D_i$, i.e., $D = \{D_1, \ldots, D_k\}$ and $C = \{c_1, \ldots, c_m\}$ be the set of constraints over the parameters $p_i$ and their values $v_j^i$. We say that $M = (P, D, C)$ is an *Input Parameter Model* for the system $S$.

Based on an IPM, test generators create a *test suite* $TS$ consisting of multiple test cases $tc_i$, each assigning a specific value to every parameter $p \in P$. The primary goal of $TS$ is to ensure coverage of all feasible interactions among sets of $t$ parameters, where $t$ represents the *strength* of the test suite. An IPM can be written in one of the many syntaxes available in the CIT domain, such as CTWedge [14], ACTS [37], or PICT [26]. An example of IPM, in the CTWedge format, is given in Listing 1. The model includes two enumerative parameters (p1 and p3), one Boolean parameter (p2), and a parameter (p4) defined over an integer range. Additionally, it includes a set of three constraints applied to the parameters.

```
Model example1

Parameters:
p1 : {V1, V2}
p2 : Boolean
p3 : {V1, V2, V3}
p4 : [2 .. 5]

Constraints:
# p1 != p3 #
# (p3=V1 => p2=false) AND p1=V2 #
# (p4=3 <=> p2=true) OR p3=V3 #
```

Listing 1. Example of an IPM containing four params. and three constraints

In practice, an IPM may be syntactically *correct*, i.e., the set of parameters and the one of constraints are consistent and it has been written by using a correct grammar, but *unsolvable*, i.e., no valid test can be derived from that IPM. Formally, the following definition can be used.

*Definition 2 (Solvable IPM):* Let $M$ be the input parameter model as defined in Def. 1 and $TS$ a test suite containing only test cases complying with the constraints specified by $M$. We say that $M$ is *solvable* if $TS \neq \emptyset$.

The reason for which an IPM $M$ is unsolvable is that the set of constraints excludes all possible combinations of parameter values. However, in the context of benchmarking combinatorial test generators, having benchmarks guaranteeing their solvability is important. This is one of the rationales behind this paper.

When the model is solvable, a test suite $TS$ is generated by a combinatorial test generator. However, not all possible $t$-way interactions may be covered: Given a strength $t$, some of the $t$-uples may clash with one or a conjunction of constraints (i.e., the assignments contained in the $t$-uple violate at least a constraint or a combination of them). Thus, none of the tests generated by a test generator starting from an IPM will cover those $t$-uples and we say that they are *not feasible* or *invalid*. This will make the job of a test generator more difficult. To measure the effort required to a test generator to filter the not feasible $t$-uples out, we introduce the concept of *Tuple validity ratio* ($r_{tp}$), defined as follows.

*Definition 3 (Tuple validity ratio):* Let $M = (P, D, C)$ be the IPM for a system $S$ and $t$ be the required strength for test generation. We say that the *tuple validity ratio* $r_{tp}$ is the fraction of the number of valid $t$-uples over the total number of $t$-uples.

Similarly, because of the constraints, not all possible combinations of parameter values (i.e., the tests) may be valid: They may violate one or more constraints. Instead, tests complying with the constraints of the IPM are considered as *valid*. For this reason, in analogy with the *tuple validity ratio*, to estimate how difficult may be for a generator to generate valid test cases, we exploit the concept of *Test validity ratio* ($r_{ts}$).

*Definition 4 (Test validity ratio):* Let $M = (P, D, C)$ be the IPM for a system $S$, $TS_{nc}$ be the set of all possible test cases that can be generated when the constraints $C$ of $M$ are ignored. Let $TS \subseteq TS_{nc}$ be the set of valid test cases, i.e.,

**Algorithm 1** BenCIGen – Algorithm for the generation of benchmarks

```
Require: $nBench$, the number of IPMs to be generated
Require: $\langle kMin, kMax\rangle$, the min./max. number of params in each IPM
Require: $\langle lInt, uInt\rangle$, the lower/upper bounds for integer ranges
Require: $\langle vMin, vMax\rangle$, the min./max. cardinality
Require: $\langle cMin, cMax\rangle$, the min./max. number of constraints in each IPM
Require: $\langle dMin, dMax\rangle$, the min./max. constraint complexities
Require: $cnstrConfig$, the configuration for constraints
Require: $r_{tp}$, the desired tuple validity ratio
Require: $useTupleR$, whether to consider $r_{tp}$ during IPMs generation
Require: $r_{ts}$, the desired test validity ratio
Require: $useTestR$, whether to consider $r_{ts}$ during IPMs generation
Ensure: $modelsList$, the list of the generated benchmarks

  ▷ Initially, no models have been generated
 1: $modelsList \leftarrow \emptyset$; $nB \leftarrow 0$;
 2: while $nB < nBench$ do
 3:     for $nAttempts \leftarrow 1$ to $10$ do
            ▷ Randomly build the model
 4:         $m \leftarrow$ GenerateModel($kMin$, $kMax$, $lInt$, $uInt$, $vMin$,
                $vMax$, $cMin$, $cMax$, $dMin$, $dMax$)
 5:         if IsModelOK($m$,$r_{tp}$, $useTupleR$, $r_{ts}$, $useTestR$) then
 6:             $modelsList$.add($m$)
 7:             $nB \leftarrow nB + 1$; $nAttempts \leftarrow 0$
 8:             break
 9:         end if
10:     end for
11: end while

  ▷ Check if a model $m$ already satisfies the requirements
12: function IsModelOK(m, $r_{tp}$, $useTupleR$, $r_{ts}$, $useTestR$)
13:     return $m$.isSolvable() &
14:     $useTupleR \rightarrow m$.getTupleVRatio() $\approx r_{tp}$ &
15:     $useTestR \rightarrow m$.getTestVRatio() $\approx r_{ts}$
16: end function

17: function GenerateModel($kMin$, $kMax$, $lInt$, $uInt$, $vMin$, $vMax$,
    $cMin$, $cMax$, $dMin$, $dMax$)
        ▷ Randomly define parameters
18:     $nP \leftarrow$ randomBetween($kMin$,$kMax$)
19:     $pList \leftarrow$ defineParams($nP$, $lInt$, $uInt$, $vMin$, $vMax$)
        ▷ Randomly define constraints
20:     $nC \leftarrow$ randomBetween($cMin$,$cMax$)
21:     $cList \leftarrow$ defineCnstr($pList$, $nC$, $dMin$, $dMax$)
22:     return buildModel($pList$, $cList$)
23: end function
```

the set of those that do not violate any of the constraints in $C$. We say that the *test validity ratio* $r_{ts}$ is the fraction of the number of valid tests (i.e., the cardinality of $TS$) over the total number of possible tests $N$ (i.e., the cardinality of $TS_{nc}$).

For the work we present in this paper, we are interested in generating only *solvable* benchmarks, possibly meeting the desired *tuple* or *test validity ratio*.

## III. BenCIGen

In this section, we report the basic algorithm implemented by BenCIGen, which we have extended in this work. The tool, including the functionalities we present in this paper, is available at https://github.com/fmselab/CIT_Benchmark_Generator.

The procedure implemented by BenCIGen is reported in Algorithm 1. The algorithm's objective is to generate $nBench$ IPMs with the desired characteristics. Each benchmark is built with the function GenerateModel (line 4). For each IPM, the tool initially extracts a random number of parameters (line 18) within specified bounds, denoted as $kMin$ and $kMax$. Next, the function DefineParams is invoked to generate the set of parameters to be included in the IPM. This function randomly selects the parameter types and values for each parameter (line 19). Similarly, the constraints are defined (lines 20 and 21) in a number within specified bounds, denoted as $cMin$ and $cMax$, and with a complexity included in the range $dMin$ and $dMax$. In the case of parameters, their type and possible values are randomly defined while still keeping compliance with the IPM category requested by the user and with a cardinality included in the range $vMin$ and $vMax$. More specifically, models may contain only Boolean parameters, Boolean and Enumerative parameters, or Boolean, Enumerative, and Integer ranges (limited within $lInt$ and $uInt$). Note that when defining these categories, we based our choice on the models available in the literature. The type of each parameter is randomly chosen. Therefore, models containing, for example, only enumeratives are also possible and will fall into the same category as models with enumeratives and Boolean parameters. In the case of constraints, BenCIGen randomly sets their number, complexity, and form depending on the selected configuration $cnstrConfig$. Readers interested in better understanding the constraints and parameters generation process may refer to [9].

By following the outlined process, BenCIGen produces a single IPM (line 4). The IPM is checked to verify whether it is solvable (line 13) and, in the case in which the tuple validity ratio and/or the test validity ratio have to be met, has the required ratios (lines 14 and 15). If all requirements are met, the model is added to the $modelsList$, otherwise, a new model is generated. This process can take a long time, especially if checking some ratios is required. For this reason, we set a maximum number of $nAttempts$ of 10 trials for the single IPM. However, this process is highly inefficient, and even with this technique, no guarantee to generate the requested $nBench$ number of IPMs is given. This is the problem we try to address in this paper.

## IV. BenCIGen$_S$ – Search-based generation of compliant benchmarks

In this section, we present the search-based strategies we implemented in BenCIGen$_S$ to address the issue of discarding the generated benchmark IPMs when they fail to meet the predefined requirements, specifically regarding *ratios* or *solvability*. It exploits a custom evolutionary algorithm implemented with the Watchmaker Framework [10] and, by taking advantage of its functionalities, we have changed Algorithm 1 into Algorithm 2.

The process is very similar, in terms of the first model generation, to the one natively implemented by BenCIGen. The only difference is that, after having generated the model (line 3), the algorithm checks whether it is necessary to evolve the model because it is not solvable, the IPM does not meet the (possibly) required tuple validity ratio, or does not meet the required test validity (line 5). If so, BenCIGen tries to repair the model (line 9) as described in Section IV-A. At the end, if a compliant IPM has been generated (it could be either compliant without any evolution, or be repaired to meet the requirements

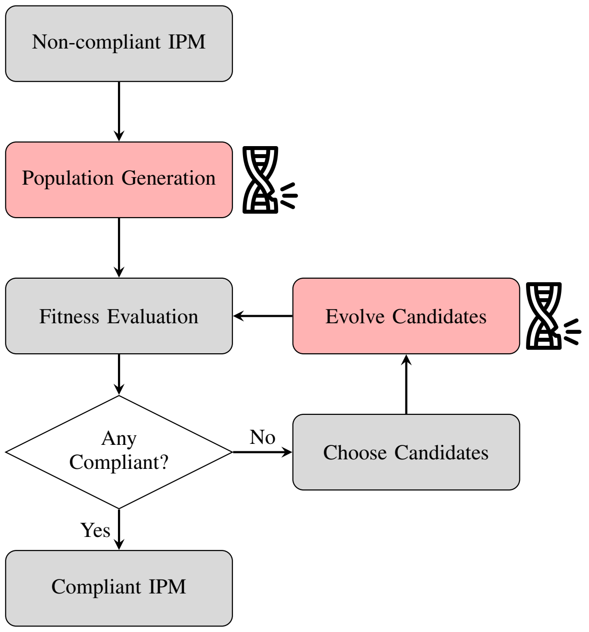

Fig. 1. Search-based process followed by BENCIGEN$_S$ for the generation of compliant IPMs

with the new search-based strategy) by BENCIGEN$_S$ (line 11), it is added to the list of benchmarks.

## A. Framework description

The solution implemented by BENCIGEN$_S$ for repairing non-compliant IPMs follows a typical search-based approach, as shown in Fig. 1. It implements the EvolveModel function (see line 9 in Algorithm 2).

The described process is only executed when a non-compliant model is generated, such as a non-solvable model or one that fails to meet the required ratio requirements (see Section II). In this case, a *population* of IPMs is generated starting from the one initially generated by BENCIGEN. These IPMs are generated by applying a set of mutations, with different probabilities, as discussed in more details in Section IV-B.

Each *individual* of the *population* is then evaluated based on its *fitness* to determine which IPMs are more likely to meet the user's requirements. Section IV-C will discuss more details on the fitness evaluation. After computing the fitness for every individual in the population, the process can have two possible outcomes: at least one individual meets the requirements, or no individual meets the requirements. In the former case, one of the IPMs compliant with the requirements, namely the one with lowest fitness, is given as output. In the latter case, a set of *candidates* is selected from those with the higher fitness value, and a new *population* is generated by evolving these candidates through a series of mutations (refer to Section IV-B for details).

This process is carried out by BENCIGEN$_S$ for each IPM to be adjusted for compliance with the user's requirements, continuing until either the time budget is consumed or a compliant IPM is identified.

**Algorithm 2** BENCIGEN$_S$ – Search-based algorithm for the generation of benchmarks

```
Require: $nBench$, the number of IPMs to be generated
Require: $\langle kMin, kMax\rangle$, the min./max. number of params in each IPM
Require: $\langle lInt, uInt\rangle$, the lower/upper bounds for integer ranges
Require: $\langle vMin, vMax\rangle$, the min./max. cardinality
Require: $\langle cMin, cMax\rangle$, the min./max. number of constraints in each IPM
Require: $\langle dMin, dMax\rangle$, the min./max. constraint complexities
Require: $cnstrConfig$, the configuration for constraints
Require: $r_{tp}$, the desired tuple validity ratio
Require: $useTupleR$, whether to consider $r_{tp}$ during IPMs generation
Require: $r_{ts}$, the desired test validity ratio
Require: $useTestR$, whether to consider $r_{ts}$ during IPMs generation
Require: $budget$, the time budget
Ensure: $modelsList$, the list of the generated benchmarks

   ▷ Initially, no models have been generated
 1: $modelsList \leftarrow \emptyset$; $nB \leftarrow 0$;
 2: while $nB < nBench$ do
        ▷ Randomly build the model
 3:     $m \leftarrow$ GENERATEMODEL($kMin$, $kMax$, $lInt$, $uInt$, $vMin$,
            $vMax$, $cMin$, $cMax$, $dMin$, $dMax$)
 4:     $evolve \leftarrow false$
        ▷ Check that the generated IPM meets the requirements
 5:     if not ISMODELOK($m$,$r_{tp}$, $useTupleR$, $r_{ts}$, $useTestR$) then
 6:         $evolve \leftarrow true$
 7:     end if
        ▷ Check whether it is necessary to repair the IPM
 8:     if $evolve$ then
 9:         $m \leftarrow$ EVOLVEMODEL($m$, $kMin$, $kMax$, $lInt$, $uInt$,
                $vMin$, $vMax$, $cMin$, $cMax$, $dMin$, $dMax$,
                $budget$)
10:     end if
        ▷ If a model is available, add it to the list
11:     if $m \neq null$ then
12:         $modelsList$.add($m$)
13:         $nB \leftarrow nB + 1$;
14:     end if
15: end while
```

## B. Mutations

As reported in Fig. 1, the *population generation* and *evolve candidates* phases require BENCIGEN$_S$ to apply mutations to IPMs to generate new individuals. The tool implements the mutations listed in Table I, which act both on constraints and parameters. From the implementation point of view, these mutations operate on the internal representation of mutated objects (i.e., IPMs, constraints, and parameters). Most mutations are the classical ones used for Boolean expressions. In addition, we added some mutations we specifically designed for combinatorial models.

For what concerns parameters, the ParAdd mutation aims at adding new parameters, while ParExt extends the domain of enumerative or integer parameters. The rationale behind these two mutations is that by extending the parameter size and number there will be more possible combinations, which may lead a not *solvable* model to become solvable. Generally, not all mutators can be applied to all inputs. A mutation is only applicable to IPMs of the corresponding type. While the first mutation can be applied to any tracks and constraint form, the second one requires the IPM to contain enumeratives or integers parameters. Therefore, it is only used when generating models for categories with these types of parameters (i.e., `MCAC` and `NUMC`).

TABLE I
LIST OF ALL MUTATIONS SUPPORTED BY OUR SEARCH-BASED FRAMEWORK FOR EVOLVING CIT MODELS

| Mutation | Description | Track | Constraint Form |
|---|---|---|---|
| ParAdd | Adds a new parameter | All | All |
| ParExt | Extends the domain of an enumerative or integer parameter | MCAC, NUMC | All |
| CnstrAdd | Adds a new constraint | BOOLC, MCAC, NUMC | All |
| CnstrDel | Removes a constraint | BOOLC, MCAC, NUMC | All |
| CnstrSubst | Substitutes a constraint with a new one | BOOLC, MCAC, NUMC | All |
| AndToOr | Changes an And into an Or in a constraint | BOOLC, MCAC, NUMC | All* |
| OrToAnd | Changes an Or into an And in a constraint | BOOLC, MCAC, NUMC | All* |
| ImplToDbl | Changes an Implies into a Double Implies in a constraint | BOOLC, MCAC, NUMC | General |
| DblToImpl | Changes a Double Implies into an Implies in a constraint | BOOLC, MCAC, NUMC | General |
| NotAdd | Adds a Not to a constraint | BOOLC, MCAC, NUMC | All* |
| NotDel | Removes a Not from a constraint | BOOLC, MCAC, NUMC | All* |

For what concerns constraints, the CnstrAdd, CnstrDel, and CnstrSubst act at model level. The first mutation adds new constraints, the second one removes a random constraint from the model, and the third substitutes a constraint with a new random one. In this way, we directly interfere with the solvability and the ratios: In general, removing (or relaxing) the constraints increases the probability of having higher ratios or *solvable* IPMs; Adding (or making stricter) constraints increases the probability of reducing ratios. Finally, the last group of mutations (AndToOr, OrToAnd, ImplToDbl, DblToImpl) is composed of operations that substitute a logical operator with a different one, while NotAdd negates a randomly selected constraint and NotDel randomly selects a constraint containing a not operator and removes it. All mutations that affect constraints require the track to be one of those containing constraints (for example, BOOLC, MCAC, or NUMC) to be applied. Finally, mutations that change implications can be applied only on models having constraints in general form, and not in CNF or forbidden tuples [36], as in these cases no implication exists in the constraints. In all other cases, mutations can be applied to any IPM with constraints. However, when mutations involve ANDs, ORs, or NOTs (as indicated by the stars in Table I),[2] an additional check is necessary to ensure that they do not alter the constraint form requested by the user. For instance, a NOT can only be added to a clause if constraints are in CNF format to maintain the constraint in CNF form.

Regardless of the type of mutation, each of them produces a benchmark IPM that is compliant with the user's settings. For example, considering the ParAdd mutation, new parameters will be added only until the number of parameters does not exceed the maximum number of parameters (see $kMax$ in Algorithm 1 and 2).

Note that, at the moment, BENCIGEN$_S$ implements only two mutations on parameters (i.e., ParAdd and ParExt) as removing parameters or changing their type deeply impacts constraints and requires strong refactoring. However, as future work, we are working on techniques for repairing constraints when parameters change. It will allow us to include additional mutations.

Each of the previously described mutations is implemented in Java in BENCIGEN$_S$, by implementing the interface provided by Watchmaker, i.e., EvolutionaryOperator. This interface defines an operation that takes a population of candidates as an argument and returns a new population that is the result of applying a transformation to the original population.

In addition, in the proposed implementation, we implemented a probabilistic criterion for applying mutations to introduce additional randomness in the IPM evolution: For each mutation, an *application probability* can be defined. We will detail the probabilities we used in our experiments in more detail in Section V-A.

## C. Fitness evaluation

For a search-based algorithm to effectively guide the evolution of the input, the *fitness* function is crucial to define [5]. It allows for evaluating the goodness of an *individual* (i.e., an IPM, in our case).

With BENCIGEN, our goal is to generate *solvable* models, and when requested by the user, we aim to meet a specific ratio (as outlined in Section II). Therefore, the fitness function must encompass both aspects, and we will provide a more detailed description of how this is achieved in the following. The fitness function is computed by the ModelSolvabilityEvaluator, ModelTestRatioEvaluator, and ModelTupleRatioEvaluator classes, which implement the FitnessEvaluator Java interface provided by Watchmaker. The Watchmaker framework aims to minimize the fitness function, which depends on the specific target. The approach we implemented in BENCIGEN$_S$ supports three different and mutually exclusive fitness functions.

If only the *solvability* is required we use the ModelSolvabilityEvaluator class and we define the fitness *fit* for an IPM $M$ as follows:

$$fit(M) = \#Unsat$$

In this formula, $\#Unsat$ represents the size of the minimum UNSAT core, i.e., the minimum number of clauses set by the

[2]The unconstrained tracks supported by BENCIGEN are: UNIFORM_BOOLEAN with only Boolean parameters, UNIFORM_ALL with all parameters with the same cardinality, and MCA with mixed cardinality parameters. The constrained tracks are: BOOLC with Boolean parameters, MCAC with mixed cardinality, and NUMC with mixed cardinality and constraints including mathematical operations.

IPMs that make it non-solvable.[3] The rationale behind the use of $\#Unsat$ is that we aim to reduce the size of this set and obtain an empty one. With this fitness function, we only accept models that are solvable without checking any ratios.

When a test validity ratio $r_{ts}$ has to be met, we use the ModelTestRatioEvaluator class and we evaluate the *fit* for an IPM $M$ as follows:

$$fit(M) = \begin{cases} \#Unsat & \text{if } M \text{ is not solvable} \\ |r_{ts} - \tilde{r}_{ts}| & \text{otherwise} \end{cases}$$

This function calculates the distance between the target ratio $r_{ts}$ and the one of the generated IPM, $\tilde{r}_{ts}$. Still, it also considers the model's solvability, which is a crucial factor. Indeed, a non solvable model will of course have a null $\tilde{r}_{ts}$.

Similarly, when a tuple validity ratio $r_{tp}$ has to be met, we use the ModelTupleRatioEvaluator class and we evaluate the *fit* for an IPM $M$ as follows:

$$fit(M) = \begin{cases} \#Unsat & \text{if } M \text{ is not solvable} \\ |r_{tp} - \tilde{r}_{tp}| & \text{otherwise} \end{cases}$$

This function evaluates the distance between the target ratio $r_{tp}$ and the one of the generated IPM, $\tilde{r}_{tp}$, but it also considers the model's solvability, which is a crucial factor.

Note that, in this work, we do not consider the scenario where both ratios must be met, as this would constitute a multi-objective optimization problem that requires a different approach for solution. However, in future work, we may explore this scenario to address this limitation. As reported by the mathematical description of the fitness functions, the IPM $M$ must be solvable also when the target is a specific ratio. Indeed, if it is not solvable, the fitness will always be the maximum one, i.e., $fit(M) \geq 1$. In all other cases, $fit(M)$ will be smaller than 1 and depend on the distance from the ratio achieved by the generated IPM and the target one.

In our optimization approach, we use the *roulette-wheel* selection to choose individuals for reproduction in each generation. This selection method is inspired by the spinning of a roulette wheel, where the probability of an individual being selected is proportional to its fitness relative to the entire population. It ensures that better-performing individuals are more likely to contribute to the next generation, driving the population toward optimal solutions while still maintaining diversity by allowing less-fit individuals a chance to participate.

## V. Evaluation

In this section, we evaluate the new search-based IPM repairing functionalities added to BenCIGen$_S$. In Section V-A we describe the experimental setup we used, while in Section V-B we discuss the results we obtained.

[3]The library we use to compute the UNSAT core does not require the constraints to be translated in CNF.

### A. Experimental setup

To assess the approach proposed in this paper, we employed the research methodology outlined below. The subjects of the evaluation are the two different approaches for IPM generation, i.e., the one originally implemented by BenCIGen and the one of BenCIGen$_S$. Considering that unconstrained benchmarks can not be unsolvable or meet a specific ratio (as all parameter value combinations will always be valid), in this evaluation we only focus on constrained IPMs.

The results we report in this evaluation section have been obtained by executing the experiments on a Server with 256GB RAM, an AMD Ryzen Threadripper PRO 7985WX 64-Cores CPU, and running Ubuntu Server 24.04.

In all cases requiring a statistical analysis, we used the Wilcoxon Signed-Rank test [34], a general test that compares the distributions in paired samples and that does not require data to be normally distributed. Given $x$ the measure to be compared between the two techniques, the test is performed using a significance level $\alpha = 0.05$ and the null hypothesis $H_0$ stating that the distributions of $x$ in the two techniques are equal. Besides the Wilcoxon Signed-Rank test, we have evaluated the effect size by computing the Cliff's delta $\delta_c$. We consider the effect of a technique to be small if $|\delta_c| < 0.147$, medium if $0.147 \leq |\delta_c| < 0.33$ or large if $|\delta_c| \geq 0.33$.

*1) Experimental data generation:* For each of the three constrained tracks supported by BenCIGen (namely, BOOLC, MCAC, and NUMC), we requested the tool to generate 150 IPMs, meeting the characteristics outlined by the CT-Competition [21]. For 50 IPMs, we only required the solvability of the IPM, while for the remaining 100 IPMs, we required half of the IPMs to meet a specific test validity ratio of $r_{ts} = 0.1$ and for half a tuple validity ratio of $r_{tp} = 0.1$. To account for the stochastic nature of the methods under analysis, we repeated each experiment 10 times under identical conditions. This repetition ensured that the influence of randomness was minimized, allowing for more reliable and statistically significant conclusions.

*2) Recorded metrics:* For each execution, we recorded the following metrics: a) The number of benchmarks successfully generated (i.e., solvable IPMs) by each approach; b) The relevant ratio measures of each benchmark. For what concerns the tuple validity ratio, we computed it precisely thanks to the *ModelAnalyzer* component of BenCIGen (see [9] for more details) which allows for extracting relevant information from combinatorial IPMs. Instead, for what concerns the test validity ratio, we computed it precisely when possible (i.e., for BOOLC and MCAC categories) and we use the approximate formula presented in [9] for NUMC benchmarks. By using the metrics we recorded, we could compute the distance of each generated IPM from the desired target (solvability, test validity ratio, or tuple validity ratio) and compare the two different approaches for benchmark generation.

*3) Tool configuration:* During experiments, we set BenCIGen and BenCIGen$_S$ with the same configurations for the CT-Competition and generated benchmarks for all the constrained categories supported by the tool. Regarding the

TABLE II
CONFIGURATIONS SET FOR EACH MUTATION IN OUR EXPERIMENTS

| Mutation | Probability | Solvability | $r_{ts}$ | $r_{tp}$ |
|---|---|---|---|---|
| ParAdd | 50 % | × | × | × |
| ParExt | 50 % | × | × | × |
| CnstrAdd | 50 % |  | × | × |
| CnstrDel | 50 % | × | × | × |
| CnstrSubst | 50 % | × | × | × |
| AndToOr | 50 % | × | × | × |
| OrToAnd | 50 % | × | × | × |
| ImplToDbl | 50 % | × | × | × |
| DblToImpl | 50 % | × | × | × |
| NotAdd | 50 % | × | × | × |
| NotDel | 50 % | × | × | × |

search-based generation, we configured BENCIGEN$_S$ with the probabilities reported in Table II and, we used all mutations when it was necessary to meet a defined ratio, while we did not use the CnstrAdd mutation when looking only for solvable models. Indeed, the impact of all mutations may vary, but for the CnstrAdd, its impact is certain to reduce the number of test cases because it makes the IPM more stringent. Additionally, we configured the evolution process with a population[4] of 10 individuals and an elitism percentage[5] of 50%. Regarding fitness, we used those described in Section IV-C, and we added a Stagnation component, blocking the evolution after 20 generations having the same fitness, and a time budget of 10 minutes for generating each IPM. Finally, for IPMs requiring the approximate computation for the ratio, we set $p = 90\%$, i.e., the probability of the estimation to be correct, and a tolerance in ratio $\varepsilon = 0.05$[6]. In the following, we will refer to solvable IPMs that have distance from the ratios, when required, lower than $\varepsilon$ as *compliant* IPMs.

*4) Research questions:* By applying the presented experimental methodology, we evaluate BENCIGEN and its search-based generation guided by the following research questions:

- **RQ1**: Does the use of BENCIGEN$_S$ result in a higher percentage of compliant benchmarks, regardless of the time required and with the same number of runs?
- **RQ2**: Is using BENCIGEN$_S$ advantageous in terms of the time required to generate benchmarks?

More specifically, RQ1 assesses the effectiveness of the mutations implemented by BENCIGEN$_S$, while RQ2 examines the efficiency of BENCIGEN$_S$.

## B. Experimental results

*1) RQ1 - Effectiveness of* BENCIGEN$_S$*:* In this research question, we compare the compliance of the IPMs generated by the traditional BENCIGEN approach and those with BENCIGEN$_S$ with respect to user requirements. More specifically, without any time constraint, we set the same initial conditions for the two approaches (i.e., 10 attempts for the

[4]The population size represents the number of individuals generated at each search iteration.

[5]The elitism percentage is the percentage of individuals selected from a population before its evolution.

[6]In this way, $r \approx r_t$ in Algorithm 1 and Algorithm 2 becomes $|r - r_t| \leq \varepsilon$

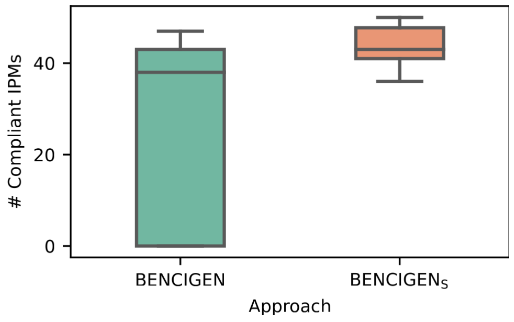

Fig. 2. RQ1 – Overall comparison among number of benchmarks with BENCIGEN and BENCIGEN$_S$

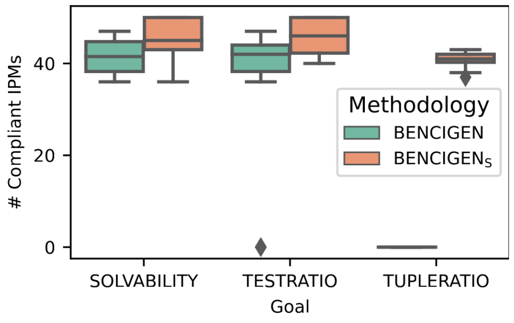

Fig. 3. RQ1 – Comparison among number of benchmarks with BENCIGEN and BENCIGEN$_S$ per goal

traditional BENCIGEN generation, and a population of 10 IPMs for the search-based strategy BENCIGEN$_S$), and we evaluate whether the defined mutations (see Section IV-B) are effective in increasing the number of compliant benchmarks.

Fig. 2 reports an overall view of the improvement introduced by using BENCIGEN$_S$ with respect to BENCIGEN. From the results we obtained, we can see that BENCIGEN$_S$ consistently performs better than the original approach implemented by BENCIGEN. Indeed, with BENCIGEN, on average, we were able to obtain 26.44 compliant IPMs, while with BENCIGEN$_S$ 44.17 over the 10 runs we performed with each approach. This conclusion is statistically relevant, as the Wilcoxon Signed-Rank Test resulted in a p-value of $3.79 \cdot 10^{-16}$ and with an effect size of 0.61, indicating a strong correlation between the improvement and the generation strategy.

From the results we obtained, we noticed that the impact of using search strategies is not the same with all goals. In Fig. 3 we report the comparison between BENCIGEN and BENCIGEN$_S$ considering the different goals (i.e., solvability, test validity ratio, and tuple validity ratio). We can notice that with all goals using BENCIGEN$_S$ allows for obtaining a higher number of compliant IPMs, but the goal that benefits the most from the new search-based strategy is that of meeting a desired tuple ratio. Indeed, while generating solvable IPMs, we obtained, on average, 41.60 IPMs with BENCIGEN and 45.60 with BENCIGEN$_S$ (p-value $4.96 \cdot 10^{-6}$ and effect size 0.59). When checking the test validity ratio, we obtained, on average, 37.73 IPMs with BENCIGEN and 45.87 with

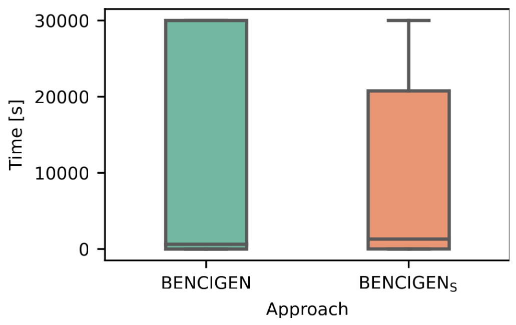

Fig. 4. RQ2 – Overall comparison among times with BENCIGEN and BENCIGEN$_S$

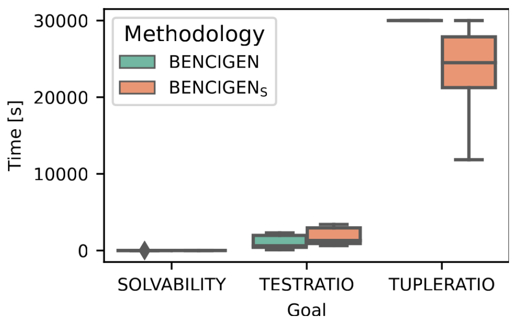

Fig. 5. RQ2 – Comparison among times with BENCIGEN and BENCIGEN$_S$ per goal

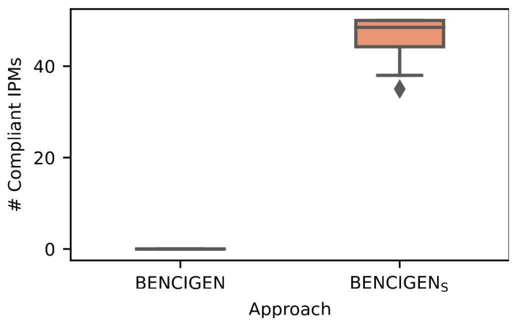

Fig. 6. RQ2 – Number of compliant IPMs generated by BENCIGEN and BENCIGEN$_S$ with goal TupleRatio

BENCIGEN$_S$ (p-value $2.26 \cdot 10^{-6}$ and effect size 0.61). Finally, checking the tuple validity ratio resulted in no model generated by BENCIGEN and 41.03 (p-value $1.43 \cdot 10^{-6}$ and effect size 0.62), on average, by BENCIGEN$_S$. As demonstrated by the p-values and effect sizes, all experiments are statistically significant and allowed us to state that the mutations and, in general, the process used by BENCIGEN$_S$ positively influence the probability of obtaining compliant benchmarks, given the considered tolerance $\varepsilon$.

*2) RQ2 - Generation time:* In this research question, we aim to investigate whether generating the same target number of IPMs, i.e., 50 benchmarks, takes longer with BENCIGEN$_S$ than with BENCIGEN, or not. In both approaches, we set the timeout for each IPM generation to 10 minutes, and we set the same initial conditions for the two approaches (i.e., 10 attempts for the traditional BENCIGEN generation, and a population of 10 IPMs for the search-based strategy).

Fig. 4 reports an overall comparison of the time required to generate IPMs by BENCIGEN and BENCIGEN$_S$. From the results we obtained, we can see that BENCIGEN$_S$, on average, performs comparably to BENCIGEN, with a slightly higher median value, but with a lower variability. With the regular approach, on average we were able to obtain the 50 compliant IPMs in $8.60 \cdot 10^3\ s$, while with the search-based approach in $10.34 \cdot 10^3\ s$. However, despite the difference in average value, it is not possible to claim that one method performs better than another since the Wilcoxon Signed-Rank Test returned a high p-value (0.92) and a very small effect size ($0.70 \cdot 10^{-2}$).

Upon a more thorough examination of the outcomes we achieved, we discovered that the approach employed by BENCIGEN$_S$ does not consistently provide the same level of advantage, in terms of time, in all situations. This is shown in Fig. 5, where we report the comparison between BENCIGEN and BENCIGEN$_S$ considering the different goals. The case in which we focused solely on the solvability of IPMs yielded the most comparable performance, even though the BENCIGEN approach is faster than BENCIGEN$_S$ (0.47 $s$ versus 6.74 $s$). Nevertheless, the difference is statistically significant (p-value $1.73 \cdot 10^{-6}$ and effect size 0.62), indicating that the conventional approach is faster, when a specific number of solvable benchmarks is required, compared to the BENCIGEN$_S$ approach.

Similarly, regarding experiments only targeting a specific test ratio $r_{ts}$, the difference is more evident, with the BENCIGEN approach requiring less time ($1.02 \cdot 10^3\ s$) than BENCIGEN$_S$ ($1.77 \cdot 10^3\ s$). The difference is, again, statistically relevant (p-value $2.35 \cdot 10^{-6}$) and with a large effect size 0.61. In future work, we might explore whether the IPMs obtained using BENCIGEN, while still compliant, are "less compliant" compared to those obtained using BENCIGEN$_S$, i.e., have a higher distance from the target ratios.

Finally, with the experiments targeting a specific tuple ratio $r_{tp}$, the solution implemented by BENCIGEN$_S$ is the only one that is able to effectively generate compliant IPMs (see Fig. 6). Therefore, it is evident that BENCIGEN$_S$ achieves the task of generating compliant benchmarks in a shorter time. Indeed, as reported in RQ1, BENCIGEN has not been able to generate any benchmark meeting $r_{tp} = 0.1$ within the defined timeout (i.e., 10 minutes for each benchmark), while BENCIGEN$_S$ succeeded in the generation process and generated an average number of 46.43 IPMs within the same timeframe.

## VI. THREATS TO VALIDITY

We here discuss potential threats to the validity [29] of our work and the actions we have taken to mitigate them.

*Internal validity* concerns the different outcomes observed with the techniques and tools under analysis. It refers to the fact that the results are genuinely due to the different approaches and the experimental methods used, rather than resulting from methodological errors. To minimize this risk, we thoroughly reviewed the code used in our experiments, looking for potential alternative factors that might have influenced the results, such as errors in the tools or in the experimental code itself. More specifically, two critical procedures are involved in the work presented in this paper: Computing the ratio and determining the IPM's solvability. Both measures rely on two tools: KALI [8] and MEDICI [15]. The former translates the IPM into its corresponding SMT representation and checks whether it is SAT [19]; the latter builds a Multi-Valued Decision Diagram for the IPM and counts the number of valid paths. Demonstrating the correctness of tools is not an easy task, but both tools were used in previous editions of the CT-Competition [21], and no errors were discovered.

One possible threat to *construct validity* stems from the assumption that the metrics we selected to evaluate the proposed approach — namely, solvability and ratios — are significant measures in the domain of combinatorial testing and, in particular, for evaluating combinatorial test generators. We rely on the literature for this, where similar concepts are exploited. In [30], the authors report that one of the desiderata for benchmarks is their solvability, as a task that is achievable, but not trivial, provides an opportunity for systems to show their capabilities and their shortcomings. Similarly, considering that the baseline for testing methods is random testing [4], a good measure for assessing how easy it is for a random test generator to generate valid tests is the ratio. Thus, we extend this consideration to combinatorial testing tools.

Finally, *external validity* is concerned with whether it is possible to generalize the results outside the scope of the presented outcomes. The algorithm's performance heavily depends on the randomness of the initial (unsolvable) IPM and the probabilities assigned to each mutation, as well as the population size and elitism percentage [3]. Moreover, different values for $p$ and $\varepsilon$ can favor BENCIGEN or BENCIGEN$_S$. Therefore, the outcome of the experiments may be different when considering different $p$ or $\varepsilon$. Consequently, further experiments are necessary to evaluate the statistical significance of each mutation on time consumption and the success rate, and to address potential scalability issues when complex IPMs are requested.In this paper, we have applied a search-based approach to fix combinatorial IPMs to be used as benchmarks. However, similar approaches may be applied to other kinds of benchmarks, when a set of constraints must be met.

## VII. RELATED WORK

Benchmark generation for testing tools [11], [18] has garnered substantial research attention because it plays a crucial role in evaluating the effectiveness and robustness of testing algorithms. For a combinatorial test generator, it is not enough to simply achieve the best performance, such as generating the smallest test suite, or in the shortest time possible. Additionally, it is essential to ensure that the generated test suites are valid and complete, meaning that the tests comply with the constraints and cover all possible $t$-way interactions.

In this context, several works have been presented in the past, trying to evaluate test generators and identifying those having the best performance. For example, in [6], the authors presented a benchmarking environment, exploiting CTWegde [14], in which several tools were compared and the idea of a competition between combinatorial test generator [21] born. More specifically, in that work the ACTS [37], MEDICI [15], PICT [26], CASA [16], and CAgen [32] generators were tested against a subset of 196 IPMs gathered from the literature. The sets of benchmarks and evaluated tools have been extended in [22], with the inclusion of 295 benchmarks generated by BENCIGEN [9] and the APPTS [33], IPOSolver [31], WCA [13], and pMEDICI [7], [8] tools.

In this paper, we introduced an extension of the BENCIGEN tool [9], BENCIGEN$_S$, that aims to eliminate the time-consuming process of generating non-compliant IPMs and discarding them. The one implemented by BENCIGEN is not the only attempt to generate combinatorial benchmarks. In [35], the authors proposed a method for generating benchmarks, with known solutions. This approach is different with respect to that BENCIGEN implements since our implementation does not require any solution to be known and, thus, it allows for better generalizing test models. In [2], instead, the authors propose a generator for benchmark IPMs which address only a limited set of features. For example, when considering constraints, only models containing Boolean parameters can be generated, while BENCIGEN supports also enumeratives and integer ranges. Despite these two tools effectively generate *solvable* models, to the best of our knowledge, no benchmark generator for combinatorial testing tool allows for setting the complexity of the generated IPMs through different ratios.

Search-based approaches, like the one we exploit in BENCIGEN$_S$, are commonly used in SW testing [12], [24], [25] and SW engineering [17]. Inspired by [1], where the authors adopted search-based strategies for test generation, in BENCIGEN$_S$ we added a search-based component for repairing benchmarks IPMs, which can be seen as test cases for test generators. To the best of our knowledge, this paper introduces the first approach that modifies combinatorial IPMs to repair them in accordance with specific targets (for instance, solvability and ratios).

## VIII. CONCLUSION

The availability of reliable and diverse benchmarks is critical for evaluating the performance and effectiveness of combinatorial test generators. Benchmarks enable practitioners to assess tool capabilities, identify limitations, and drive improvements in test generation. Recognizing the need for benchmarks that meet specific complexity requirements, in this paper we introduced an enhancement to BENCIGEN, a benchmark generator for combinatorial testing. Its original version faced limitations, particularly in discarding input parameter models (IPMs) that did not meet targeted requirements for ratio and solvability, leading to inefficiencies. To address these challenges, we proposed integrating a search-based generation approach: It employs iterative mutation of IPMs guided by a fitness function, allowing the tool to transform suboptimal IPMs into models that fulfill the desired characteristics instead of rejecting them outright.

Experimental results showed that BENCIGEN$_S$ generates a significantly larger proportion of benchmarks that meet specified criteria, sometimes accomplishing this within a reduced time compared to the original approach, especially when targeting a specific tuple validity ratio. Future work could involve refining the fitness function to incorporate additional model characteristics and expanding the mutation operators to better guide the evolution of models through their repair. For example, we may modify the fitness function to include the distance between individuals from their parent IPM and, in this way, avoid large shifts in the population.

## REFERENCES

[1] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, November 2010.

[2] Carlos Ansotegui and Eduard Torres. A benchmark generator for combinatorial testing, 2023.

[3] Andrea Arcuri and Gordon Fraser. *On Parameter Tuning in Search Based Software Engineering*, page 33–47. Springer Berlin Heidelberg, 2011.

[4] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, March 2012.

[5] Z. Bian, A. Blot, and J. Petke. Refining fitness functions for search-based program repair. In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, page 1–8. IEEE, June 2021.

[6] Andrea Bombarda, Edoardo Crippa, and Angelo Gargantini. An environment for benchmarking combinatorial test suite generators. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 48–56. IEEE, April 2021.

[7] Andrea Bombarda and Angelo Gargantini. Parallel test generation for combinatorial models based on multivalued decision diagrams. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 74–81. IEEE, April 2022.

[8] Andrea Bombarda and Angelo Gargantini. Incremental generation of combinatorial test suites starting from existing seed tests. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 197–205. IEEE, April 2023.

[9] Andrea Bombarda and Angelo Gargantini. Design, implementation, and validation of a benchmark generator for combinatorial interaction testing tools. *Journal of Systems and Software*, 209:111920, March 2024.

[10] Dan Dye. The Watchmaker Framework for Evolutionary Computation. https://watchmaker.uncommons.org.

[11] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, August 2006.

[12] Federico Formica, Tony Fan, and Claudio Menghi. Search-based software testing driven by automatically generated and manually defined fitness functions. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–37, December 2023.

[13] Yingjie Fu, Zhendong Lei, Shaowei Cai, Jinkun Lin, and Haoran Wang. Wca: A weighting local search for constrained combinatorial test optimization. *Information and Software Technology*, 122:106288, June 2020.

[14] 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)*, page 308–317. IEEE, April 2018.

[15] Angelo Gargantini and Paolo Vavassori. *Efficient Combinatorial Test Generation Based on Multivalued Decision Diagrams*, page 220–235. Springer International Publishing, 2014.

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

[17] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1), December 2012.

[18] Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Chen Fu, Qing Xie, Sangmin Park, Kunal Taneja, and B. M. Mainul Hossain. Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, ISSTA '12, page 1–6. ACM, July 2012.

[19] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. *JavaSMT: A Unified Interface for SMT Solvers in Java*, page 139–148. Springer International Publishing, 2016.

[20] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30(6):418–421, June 2004.

[21] Manuel Leithner, Andrea Bombarda, Michael Wagner, Angelo Gargantini, and Dimitris E. Simos. CT-Competition Web Page. https://fmselab.github.io/ct-competition.

[22] Manuel Leithner, Andrea Bombarda, Michael Wagner, Angelo Gargantini, and Dimitris E. Simos. State of the CArt: evaluating covering array generators at scale. *International Journal on Software Tools for Technology Transfer*, 26(3):301–326, May 2024.

[23] Nuno Macedo, Tiago Jorge, and Alcino Cunha. A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering*, 43(7):615–640, July 2017.

[24] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, May 2004.

[25] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, page 153–163. IEEE, March 2011.

[26] Microsoft Inc. PICT GitHub page. https://github.com/microsoft/pict.

[27] Xintao Niu, Changhai Nie, Yu Lei, and Alvin T.S. Chan. Identifying failure-inducing combinations using tuple relationship. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, page 271–280. IEEE, March 2013.

[28] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, 41(9):901–924, September 2015.

[29] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, dec 2008.

[30] S.E. Sim, S. Easterbrook, and R.C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *25th International Conference on Software Engineering, 2003. Proceedings.*, page 74–83. IEEE, 2003.

[31] Kenya Takemura. A C++ implementation of the IPO algorithm. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 72–73. IEEE, April 2022.

[32] Michael Wagner, Kristoffer Kleine, Dimitris E. Simos, Rick Kuhn, and Raghu Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 191–200. IEEE, October 2020.

[33] Yan Wang, Huayao Wu, Xintao Niu, Changhai Nie, and Jiaxi Xu. A constrained covering array generator using adaptive penalty based parallel tabu search. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 82–86. IEEE, April 2022.

[34] R. F. Woolson. Wilcoxon signed-rank test, February 2005.

[35] Abdunnaser Younes, Paul Calamai, and Otman Basir. Generalized benchmark generation for dynamic combinatorial problems. In *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, GECCO05, page 25–31. ACM, June 2005.

[36] Linbin Yu, Feng Duan, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 1–9. IEEE, April 2015.

[37] 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*, page 370–375. IEEE, March 2013.