An environment for benchmarking combinatorial test suite generators

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

Edoardo Crippa
Department of Engineering
University of Bergamo
Bergamo, Italy
e.crippa4@studenti.unibg.it

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

***Abstract*—New tools for combinatorial test generation are proposed every year. However, different generators may have different performances on different models, in terms of the number of tests produced and generation time, so the choice of which generator has to be used can be challenging. Classical comparison between CIT generators considers only the number of tests composing the test suite. Still, especially when the time dedicated to testing activity is limited, generation time can be determinant. Thus, we propose a benchmarking framework including 1) a set of generic benchmark models, 2) an interface to easily integrate new generators, 3) methods to benchmark each generator against the others and to check validity and completeness. We have tested the proposed environment using five different generators (ACTS, CAgen, CASA, Medici, and PICT), comparing the obtained results in terms of the number of test cases and generation times, errors, completeness, and validity. Finally, we propose a CIT competition, between combinatorial generators, based on our framework.**

***Index Terms*—Test Suite Generation; Combinatorial Testing; Benchmarking**

I. Introduction

Combinatorial interaction testing (CIT) has been an active area of research for many years and has proven to be very effective to test complex systems, especially for those with several input parameters. In [15] research groups that actively work on the CIT area have been listed, and many other recent groups and tools are not considered in that paper, while in [14] a lot of algorithms and tools available for CIT are analyzed. Tools for CIT include combinatorial test suite generators which exploit different algorithms in order to generate tests. The search for ever more powerful algorithms able to generate fewer tests for complex models has caused the community to introduce new tools for combinatorial test generation every year. However, benchmarking these new tools in a fair and effective way is a difficult task and there is not a well-established methodology nor an environment for generator comparison yet. Every research group has to come up with its own procedure. We have identified the following risks in the evaluation of performances of generators:

1) Only *subsets of interesting features* in combinatorial problems may be considered. For instance, a tool may support some domains but not others (like enumerative or integers) or it may allow the specification of constraints or not.

2) A tool may support a feature in a *limited way*, like the constraints given only in a certain form (like in a Normal Form). A tool may benefit from exploiting some assumptions over the combinatorial model.

3) The *benchmarks* may be selected (also unintentionally) in a way that penalizes or rewards a tool w.r.t. other chosen for comparison. For instance, a tool that works very well with boolean values may use benchmarks with only booleans.

4) The *tools that are picked for comparison*, may be not representative of the state of the art in test generation. A group may select a tool because of its availably or because it is easy to install or use, while other better-performing tools may be ignored because they require a bigger effort to make them work.

Similar problems about benchmarking algorithms and tools are faced in other fields in software engineering too. For instance, the SAT community has introduced a competition among SAT solvers that has helped the people working in that area to find more powerful tools widely adopted [13].

For these reasons, we have defined a benchmarking environment, integrating it with CTWedge [7], for evaluating combinatorial test suite generators, both in terms of the number of tests and generation time. Moreover, we have integrated methods to check the completeness and validity of the test suite (TS) and, in this way, to validate generators.

This is a different approach than the one used in classical generator evaluation since it is usually performed only taking into account the number of tests[1]. In some situations, the classical approach may be not optimal since it does not verify the completeness and the validity of TS and does not consider the generation time that, in some real scenarios, may be a significant value, when the time for testing activities in the SW development process is limited. We have applied the proposed framework available at

https://github.com/fmselab/ctwedge

to some generators already available in CTWedge (ACTS and CASA), we have integrated other new generators (CAgen,

[1] For instance, the results shown at http://pairwise.org/tools.asp provide only test suite sizes.

Medici, and PICT) and benchmarked all of them on a defined set of test models, assessing the best over generation time and tests number.

The paper is structured as follows. In Sect. II we provide some necessary background about the combinatorial testing, the model complexity, and the concepts of completeness and validity for a test suite. Sect. II-B describes the cost model we consider for evaluating combinatorial generators. In Sect. III the main features and the architecture of the proposed benchmarking framework are presented, while in Sect. IV we introduce the list of generators already available in our environment and the ones we have added. Sect. V presents the obtained results and Sect. VI reports some related works on benchmarking combinatorial test generators and proposes a CIT competition. Finally, Sect. VII concludes the paper.

II. Background

Combinatorial test generators are tools used to generate test suites suitable for testing a system that has been modeled using a constrained combinatorial model defined as follows.

Definition 1 (Constrained Combinatorial Model)**.** *Let $P = \{p_1, ..., p_n\}$ be a set of n parameters, where every parameter p_i assumes values in the domain $D_i = \{v^i_1, \dots, v^i_{o_i}\}$, let D be the set of all the D_i, i.e., $D = \{D_1, \dots, D_n\}$ and $C = \{c_1, ..., c_m\}$ be the set of constraints over the parameters p_i and their values v^i_j. We say that $M = (P, D, C)$ is a* Constrained combinatorial model.

Code 1 shows an example of constrained combinatorial model written in CTWedge.

```
Model example1

Parameters:
P1 : {V1, V2}
P2 : {V1, V2}
P3 : {V1, V2, V3}

Constraints:
# P1 != P2 #
# P3=V1 => P2=V2 #
```

Code 1. Example of a constrained combinatorial model

Because of the constraints, a test is *valid* if and only if it does not violate any constraint, otherwise is *invalid*. The same concept of validity can be extended to a tuple: a tuple t is valid, or *feasible* or *coverable* if there exists a valid test that covers t. Otherwise, a tuple is invalid or unfeasible.

When generating tests using combinatorial test generators, users must be sure about the validity of the test suite, defined as follows.

Definition 2 (Valid Test Suite)**.** *Let $M = (P, D, C)$ be a constrained combinatorial model. Given a combinatorial test suite TS, we say that TS is* valid *if all the tests $ts_i \in TS$ are valid, i.e. they do not violate any constraint in C.*

Example 1. Considering the model in Code 1, the test suite in Tab. I is *valid*, because it satisfies all the model constraints. On the other hand, the test suite in Tab. II is *not valid* because the constraint $P1! = P2$ is violated.

TABLE I
Example of a valid test suite

P1	**P2**	**P3**
V1	V2	V1
V1	V2	V2

TABLE II
Example of a not valid test suite

P1	**P2**	**P3**
V1	V2	V1
V1	V1	V2

A test suite must not only be valid, since some of the tuples may still be left uncovered, but it has to be *complete* as well, meaning that all the feasible tuples of values for parameters must be covered. Formally:

Definition 3 (Complete Test Suite)**.** *Let $M = (P, D, C)$ be a constrained combinatorial model. Given a combinatorial test suite TS and be t the strength for test generation, we say that TS is* complete *if any valid tuple tp of size t is covered by at least a test in TS.*

Example 2. Given the model in Code 1 and $t = 2$, the test suite in Tab. III is *complete*, because it covers all the tuples satisfying the model constraints. On the other hand, the test suite in Tab. IV is still valid but *not complete* because the pair $(P2 = V1, P3 = V3)$ is not covered.

A. How to compare Constrained Combinatorial Models

Each combinatorial model can have multiple parameters and constraints, leading to a different complexity that can be measured in different ways. We propose the following complexity measures:

- *Number of parameters*, since having more parameters means having more combinations to be checked.

TABLE III
Example of a valid and complete test suite

P1	**P2**	**P3**
V1	V2	V1
V1	V2	V2
V1	V2	V3
V2	V1	V2
V2	V1	V3

TABLE IV
Example of a valid but not complete test suite

P1	**P2**	**P3**
V1	V2	V1
V1	V2	V2
V1	V2	V3
V2	V1	V2

- *Size*, i.e. the total number of distinct tests (valid and invalid) that can be generated, corresponding to the product of the cardinalities of all the domains. Having more possible combinations means a higher complexity in terms of execution time.
- *Number of constraints*. More constrained models may need more constraint checks, which make more complex the test generation procedure.
- *Number of logical operators* in the constraints. CIT models may have a lot of simple constraints or a few constraints which are composed of a lot of logical operators. Thus, this definition of model complexity is not limited only to the number of constraints but considers their complexity as well.

Besides the measures given above, we introduce two further measures that refer only to the *semantics* of the models:

- *Tuple Validity Ratio*, given a strength t, the tuple validity ratio is the fraction of valid t-tuples over the total number of t-tuples. One way to compute the ratio is to enumerate all the tuples and check if they are valid or not. To check the validity of a tuple, any constraint or logical solver may be used. In our framework, we use Multivalued Decision Diagrams [8].
- *Test Validity Ratio*, intended as the fraction of valid tests over the total number of possible tests. The total number of feasible combinations may be computed by enumerating them and checking if they are valid or not. In the practice, this way requires and exponential time with the size of the problem and it is not doable for large models. However, there is a better way that does not require the enumeration of the tests and it is by using Multivalued Decision Diagrams [8] that can compute the number of valid models in a very efficient way.

B. How to compare generators

Having defined in Sec. II how we can measure the complexity of a constrained combinatorial model, we can now define the method we use to compare generators. In practice, it is not trivial to identify the best generator, since test generation for CIT is a multi-objective problem and must take into account both time and test suite size. Sometimes, generators can generate a lot of tests within less time, or generate a small and complete test suite in hours of computation.

In order to obtain a generation cost and a fair comparison between generators, the cost model proposed by [11] is based on three parameters

$$cost = time_{total} = time_{gen} + size \cdot time_{test}$$

where the $cost$ is equal to the total testing time, composed by the $time_{gen}$ used by generators to generate test suites and the product between the $size$ of the test suite and the average time required to perform the single test on the SUT. This cost model can be used as a method to evaluate a tool with respect to another one. For example, if a tool A requires a higher $time_{gen}$ and produces a test suite with a greater $size$ than a tool B, we can say that the second one is the best. Examples of plots describing this cost model are those in Fig. 1. Each line represented in Fig. 1 describes the $cost$ of generators, when the test execution time, i.e. the slope of the lines, is fixed. Thus, generators on the same line have the same $cost$, meaning that, for the example shown in Fig. 1, the generator $G5$ is the best one, while $G1$, $G2$ and $G3$ have the same cost, and $G4$ has the highest cost.

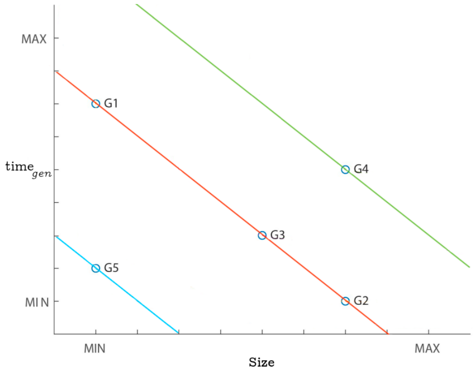


Fig. 1. Relation between generation time and test suite size for different generators G_i

However, sometimes it can happen that a tool is not able to manage a specific constrained combinatorial model (for example because it uses expression not supported by the generator, e.g. relational expressions). In this case, only the generators capable to handle the model must be considered and, between them, the less expensive one should be chosen. Moreover, this comparison must be performed only on tools generating valid, and possibly complete, test suites. In fact, completeness and validity are two of the desired characteristics of a test suite, together with the minimality [3], which is not analyzed in this paper. The analyses of these two aspects, for the tools used in this paper, are presented in Sect. V-C.

III. Benchmarking framework

This paper presents a benchmarking framework for combinatorial test generators based on CTWedge [7]. It provides, *i*) benchmark models with different features and complexities, *ii*) interfaces for generator integration, and *iii*) methods for benchmarking and validating new generators.

It has been designed to be extensible as more as possible, to allow users to add their own generators and benchmarking them, using the provided general models.

A. Benchmarks

In our benchmarking environment, we have collected 196 test models. Some of the benchmark models were already present in CTWedge as examples (previously taken from [17], [12], [18], [11], and [16]), others have been collected from the PICT GitHub page [2], and others have been extracted from the collection used in [19]. For the models which were not written using the CTWedge language, we have translated them in order to have all the benchmarks in the

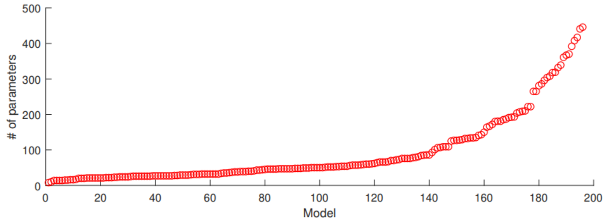


Fig. 2. Number of parameters in benchmarks

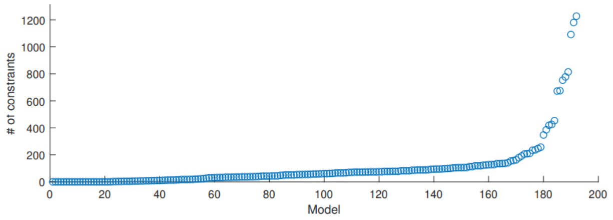


Fig. 3. Number of constraints in benchmarks

same format, both using automatic translations and manual intervention.

The benchmark models vary greatly in terms of number of variables (Fig. 2), number of constraints (Fig. 3), total size (Fig. 4), and tuple and test validity ratios (Fig. 5 and Fig. 6). Note that the total number of models shown in Fig. 5 and Fig. 6 is lower than 196 because we use Medici to compute these data and the tool is not able to process some models.

We have also tried to identify relevant features of the models and Tab. V shows them. We can see that the benchmarks cover all these features. Note that a model can exhibit more than one feature, while others none of them.

Identifying different features can help designers of test generators to better specify the targets of their tool and benchmark it only with models having those features. In the future, we plan to extend our framework in order to support *synthetic benchmarks*, i.e., for testing the behavior of a tool only

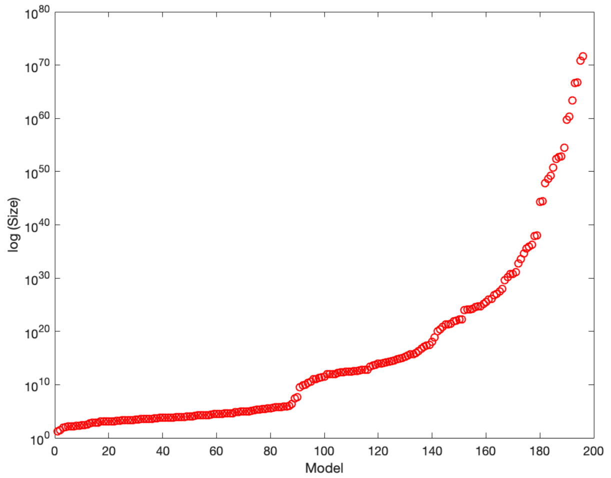


Fig. 4. Size distribution for benchmarks

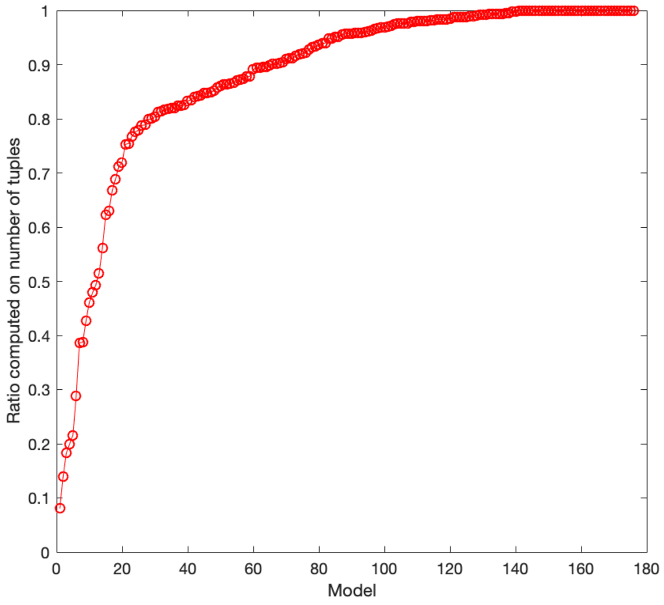


Fig. 5. Tuple Validity Ratio for benchmarks

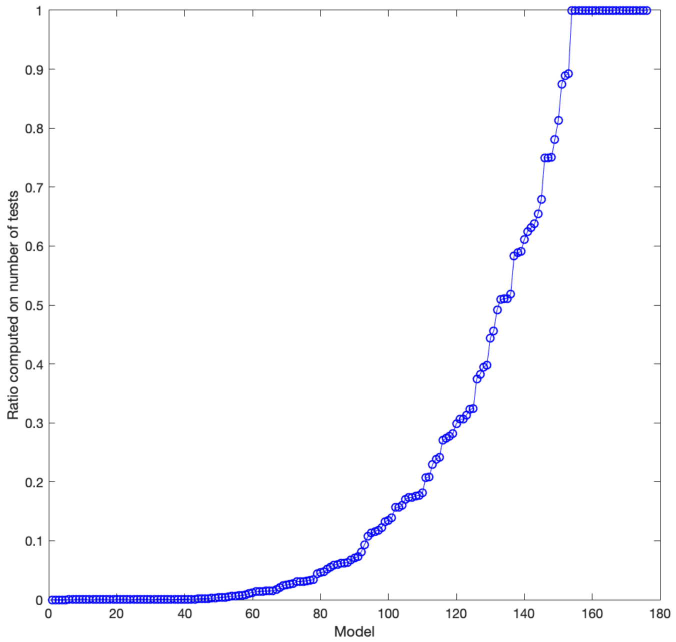


Fig. 6. Test Validity Ratio for benchmarks

TABLE V
SUMMARY OF THE BENCHMARKS FEATURES

Feature:		# models
Parameters		
	all with the same cardinality	5
	only booleans	4
	with also enumeratives	18
	with also integers	33
Constraints		
	without constraints	22
	as forbidden tuples	33
	in Clausal Normal Form	25
	containing relational operators (>, <, etc.)	6

w.r.t. a large set of models, possibly automatically generated, satisfying a defined characteristic, without considering all the others. For instance, a research group could target models with only booleans and ignore all the benchmarks containing other types of parameters.

B. Generator integration

To make the integration of new generators in the benchmarking environment easier we have deeply refactored the internal architecture of CTWedge (Fig. 7). This new structure exploits the Eclipse extension points, made available for Eclipse plugins development.

Each generator must extend the class `ICTWedgeTestGenerator`, implementing the `getTestSuite(...)` method, and extend the `ctwedge.util.ctwedgeGenerators` extension point[2]. For benchmarking purposes, the `getTestSuite(...)` method must return the generated test suite for a defined model, together with the required generation time.

This extension allows a loosely coupled structure: the benchmarking environment does not need modifications when new generators are added since they are automatically discovered by Eclipse as new plugins.

Each generator must be implemented as an Eclipse plugin, managed with Maven. Since we have used CTWedge as the hosting environment, we have chosen it also as *pivot language*, so all the benchmark models are written in CTWedge language. This means that each generator needs to include also a translator from CTWedge grammar to its own one. However, new generators can exploit the functionalities offered by CTWedge, for example, the ones included in the `util` package (e.g., to check the validity and the completeness, to convert the resulting test suite in CSV or XLS, and to analyze model features).

For example, Fig. 8 shows the internal structure of the Eclipse plugin for the CAgen generator. It includes the class performing the test suite generator and the two classes for translation, both for parameters and constraints, from the CTWedge grammar to the one used by CAgen.

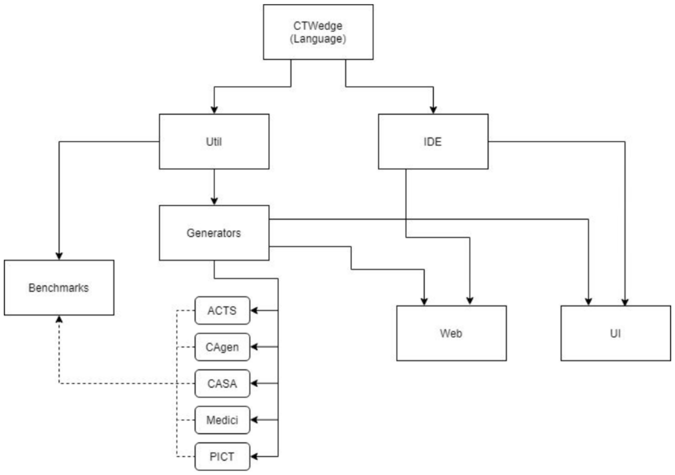
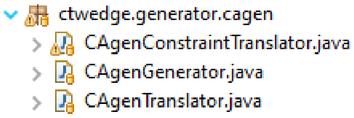


Fig. 7. Refactored architecture of CTWedge for benchmarking integration



ctwedge.generator.cagen
CAgenConstraintTranslator.java
CAgenGenerator.java
CAgenTranslator.java

Fig. 8. Structure of the Eclipse plugin for the CAgen generator

C. Validation, Completeness, and Benchmarking

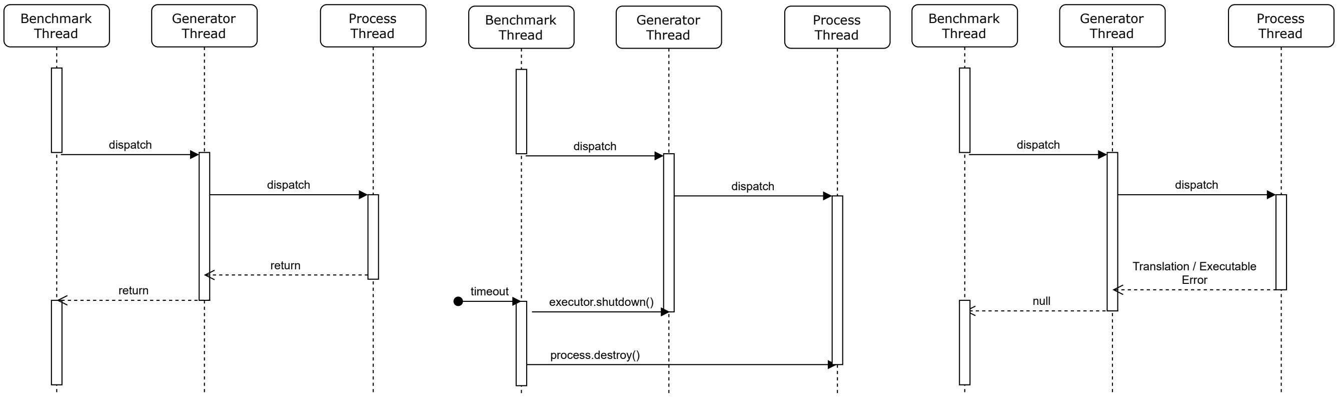
Our benchmarking framework provides completeness and validity checks, suitable to check if a tool generates a test suite violating the properties described in Sect. II. This APIs are contained in `ctwedge.util.validation` package and, in particular in the `SMTTestSuiteValidator` class exploiting a SMT solver[3]. Given a test suite `TS`, the main functionalities offered by the validation APIs are:

- `isValid()`, returning whether the test suite satisfies all the constraints contained in the CIT model;
- `howManyTestsAreValid()`, which counts how many tests contained in the test suite are valid;
- `isComplete()`, checking the completeness of the test suite. This method verify if some of the valid combinations of parameters are not covered by the test suite;
- `howManyTuplesCovers()`, returning the number of tuples covered by all the tests in the test suite.

After having verified that the tools generate valid and complete test suites, one can perform benchmarking, i.e. test the implemented generator versus the others, using a defined set of test models (see Sect. III-A). The benchmarking feature is contained in the package `ctwedge.generator.benchmarks`, in the class `BenchmarkTest`. This class exploits the *Eclipse extension point* `ctwedge.util.ctwedgeGenerators` to discover all the generators that have been defined in the environment and check the test suite sizes and generation times for each

[2] More detailed information about how to integrate new generators can be found at https://github.com/fmselab/ctwedge/wiki

[3] We use the following SMT solver: https://github.com/sosy-lab/java-smt



(a) Regular benchmarking execution

(b) Execution leading to a timeout

(c) Execution leading to an error

Fig. 9. Benchmark execution scenarios

generator on each benchmark model. The execution of a benchmark can lead to three different outcomes (see Fig. 9):

1) *success*, when the tests are correctly generated (Fig 9(a));
2) *timeout*, when the test generation overpass the set timeout (Fig 9(b));
3) *error*, when an error during the translation or the test generation occurs (Fig 9(c)).

IV. Available generators

As presented in Sect. III, we have integrated the benchmarking environment into CTWedge [7]. It already implemented two different generators:

- **ACTS** (Automated Combinatorial Testing for Software), developed by the NIST [22].
- **CASA** (Covering Arrays by Simulated Annealing), which generates combinatorial test suites using simulated annealing [10];

However, we have integrated into our benchmarking environment other generators selected from the ones presented at http://pairwise.org, to evaluate our environment. To select generators we have defined two criteria: 1) the generator must be freely available, as source code or executable; 2) the grammar to define the test models must be available.

Thus, using these criteria, we have selected the following generators:

- **CAGen** (Covering Array Generation), developed by SBAresearch [20];
- **Medici** (MultivaluEd Decision diagrams for Combinatorial Interaction testing), developed by University of Bergamo [8].
- **PICT** (Pairwise Independent Combinatorial Testing), developed by Microsoft [2];

Nevertheless, the environment we have developed has been designed to be extensible, so users can add their own generators, as previously explained in Sec. III-B, and perform benchmarking and validation checks.

V. Results

Using the framework presented in Sect. III and the generators listed in Sect. IV we have performed benchmarking tests over all the 196 models, using a strength $t = 2$. This approach has allowed us to compare the performance of the generators in terms of the number of test cases and test suite generation time.

In our tests, we have set a threshold of $150s$: each test generation exceeding this threshold has been marked as leading to a timeout. We have repeated each experiment 10 times and computed the average (for those with no errors nor timeouts). The experiments can be easily re-executed with a larger timeout and a large number of tries. A general summary of the results is shown in Tab. VI. The values regarding errors are not all depending on the lacks of the generators but can derive from a wrong translation of the CitModel for the specific generator as well. In the following, the detailed results on the comparison between generators performed with our benchmarking environment are shown.

A. Number of test cases and generation time

For each model, the number of tests generated by each tool has been divided by the average of tests generated by all the generators. Thus, comparing the number of test cases we have obtained the box-plot in Fig. 10. From this analysis, we can see that all the generators have similar performances in terms of the number of test cases since the average values are all in the interval $[-5\%, +5\%]$. However, CASA is the best performing one, even if it has great variability, and PICT the worst one.

In the same way, Fig. 11, representing the percentage difference in terms of generation time w.r.t. the average time required by all the generators, has been obtained. It shows that, even if CASA generates fewer tests than the other generators, always requires more time with respect to the others. On the other hand, PICT, which is the worst in terms of the number of tests, performs really well in terms of generation time. The fastest generator is CAgen, which performs better with respect to the average also in the number of tests.

TABLE VI
SUMMARY OF THE BENCHMARKING RESULTS

Generator	ACTS	CAgen	CASA	Medici	PICT
# best over test number	57.0	76.0	82.0	101.0	41.0
# best over generation time	66.0	116.0	0.0	7.0	8.0
Difference from avg. test cases [%]	-0.3	-1.1	-4.9	-3.1	7.3
Difference from avg. generation time [%]	-48.1	-80.0	234.0	66.5	-64.7
# timeouts	1.0	4.0	37.0	7.0	11.0
Timeouts [%]	0.5	2.0	18.9	3.6	5.6
# errors	9.0	0.0	57.0	19.0	3.0
Errors [%]	4.6	0.0	29.1	9.7	1.5

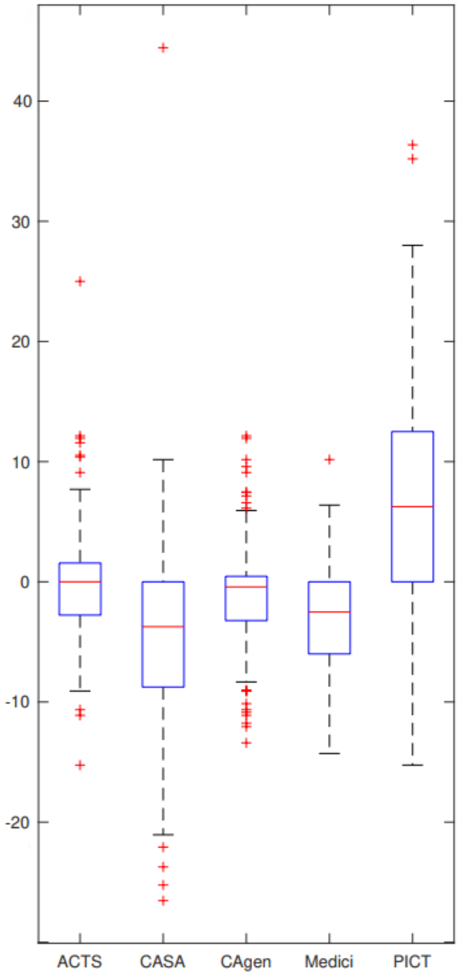


Fig. 10. Percentage difference in terms of number of test cases w.r.t. the average number

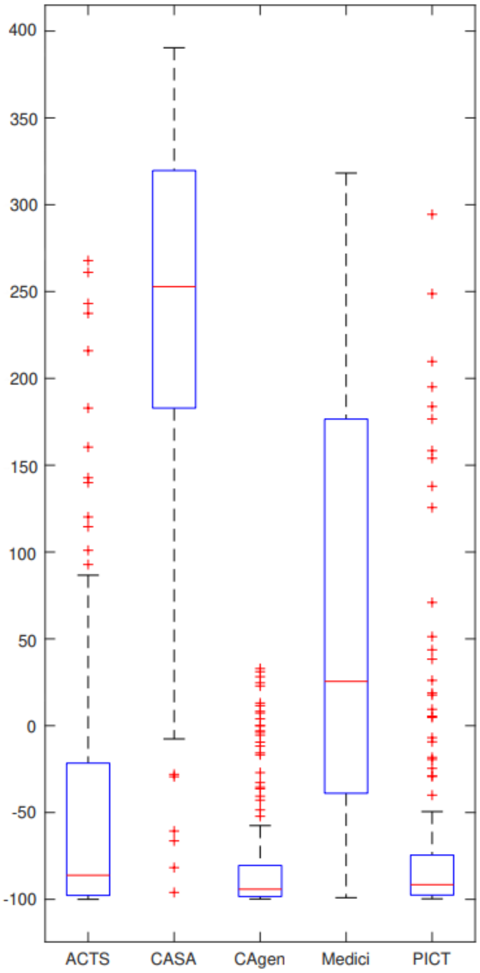


Fig. 11. Percentage difference in terms of generation time w.r.t. the average time

B. Errors

Another evaluation method for combinatorial test generators is the number of errors during test generation. Table VI reports the number of errors each generator makes. These errors can be due to several reasons. First, not all the generators support all the constraints. For example, while ACTS can deal with arithmetic and relational expressions in constraints, they are not supported in CASA nor in Medici. Furthermore, CASA and Medici do not allow a comparison between two different parameters (see example in Code 2), while the other generators do. Other errors can depend on the translation of the model from the CTWedge grammar to the one of the generator.

In our analyses, we have demonstrated that CAgen is the only generator able to deal with all the constraints and making no errors. On the other hand, CASA and Medici are subject

```
Model CASACounterexample

Parameters:
  P1 : {V1, V2}
  P2 : {V1, V2}
  P3 : [ 20 .. 30]

Constraints:
  # P1 != P2 #
  # P3 > 25 #
```

Code 2. Example of a model with constraints not supported by CASA and Medici

TABLE VII
COMPLETENESS AND VALIDITY RESULTS FOR THE BENCHMARKED GENERATORS

Generator	ACTS	CAgen	CASA	Medici	PICT
# Benchmarks	196	196	196	196	196
# Complete	185	189	100	155	178
# Valid	185	189	100	155	178
# Timeouts	2	7	39	22	15
# Errors	9	0	57	19	3

to various errors, mainly due to unsupported expressions.

C. Completeness and validity

Using the APIs for validation included in the benchmarking environment we have presented in Sect. III, we have investigated the completeness and validity of the test suites generated by each generator for all the benchmarks. Tab. VII reports the obtained results, using $150s$ as timeout for the entire operation, including validity and completeness checks, so the number of timeouts can be higher than the one presented in Tab. VI. These results confirm the considerations made for the comparison in terms of generation time and test suite size. CAgen has the highest number of complete and valid test suites, together with ACTS, while CASA and Medici have the lowest number of complete and valid test suites.

In terms of validity, the results are due to the number of timeouts and errors, since every tool generates only valid test suites for the benchmarks for which it does not fail and does not time-out. Moreover, Tab. VII shows that all the tools, when able to deal with the model and the constraints (i.e. do not have errors or timeouts), produce complete test suites.

D. Which is the best generator?

Choosing the best generator is not a trivial task, since different scenarios may require different generators. However, considering the cost model presented in Sec. II-B it is possible to note that, for a defined application scenario, the best generator is the one minimizing both the number of tests and the generation time. Thus, considering a set of generators $G = \{g_1, g_2, ..., g_n\}$, having respectively generation times $T = \{t_1, t_2, ..., t_n\}$ and test suite sizes $S = \{s_1, s_2, ..., s_n\}$ over a benchmark model, we can say that g_i is better than g_j for the considered benchmark if $t_i \leq t_j$ and $s_i \leq s_j$, regardless the test execution time $time_{test}$ (see Sect. II-B). Consequently, g_i is the best if it is better than g_j, $\forall j \in [1, ..., n]$.

TABLE VIII
GENERATORS COMPARISON

Generator	ACTS	CAgen	CASA	Medici	PICT	Any
ACTS	-	11	102	49	49	1
CAgen	51	-	100	55	116	16
CASA	9	4	-	6	16	0
Medici	17	7	81	-	19	4
PICT	19	3	85	29	-	0

To analyze deeply the performances of generators, we have compared each generator with the others, in a 1 vs 1 competition. In Tab. VIII, each cell reports the number of times in which the generator on the row has overperformed the second one reported in the column, i.e. has better performances in terms of the number of tests and generation time. The higher is this number, the darker is the cell background. Having a darker row means that the generator overperforms the others while having a darker column means that the generator is outperformed more times by the others. The last column, *Any*, reports the statistics regarding the number of times in which the generator g_i has outperformed all the other generators both in terms of generation time and test suite size.

By analyzing the table, we can see that CAgen has a higher number of wins over the other generators, so it can be considered the best one (on average) w.r.t. the ones analyzed. An exhaustive analysis must comprehend the number of errors made by each generator as well. Tab. VI shows that the best generator, under this aspect is CAgen which has the lowest number of errors, since it can deal with all kinds of constraints.

VI. RELATED WORK

Many tools have been proposed for combinatorial test generation [1], so choosing the best one can be of paramount importance. In fact, methods to select the most efficient generation algorithm or tool are useful, especially when the size and complexity of models increase, or when the time to be dedicated to testing activities is limited.

However, benchmarking CIT generators is not an easy task, so only a few researchers have found a solution. In [9], decision trees are exploited for an approach taking as input a distribution of combinatorial models and their test suites, generated using several tools to predict the algorithm performing better given the cost estimated to execute a single test and the model characteristics. The cost model we have used in this paper has been proposed by [11], which takes into account not only the generation time and the number of tests but the execution time too.

As well as the benchmarking environment, benchmark models are important, since having an exhaustive set of tests allows to conduct a complete comparison between tools. Thus, methods suitable to automatically generate benchmarks have been developed every year. For example, in [21], the authors propose a method for generating benchmarks, with known solutions, that does not suffer the usual limitations on the problem size or the sequence length, since it does not require the re-optimization phase.

In many fields competitions have been proposed, starting from SAT Solvers [13] to software testing [4]. The former has significantly contributed to the fast progress in SAT solver technology, allowing the community to provide robust, reliable, and generic purposes SAT solvers to other research communities. For instance, the CDCL-based Minisat [6] and Picosat [5] solvers, proposed in former editions of the competition, are widely reused within and outside the SAT community. The latter has stimulated researchers to create testing frameworks adaptable to general applications. It aims is to establish a set of test tasks for comparing automatic software testers.

In general, competitions have forced programmers and testers to develop tools with enhanced performances, together with higher usability and adaptability to different scenarios. Thus, we believe that our benchmarking framework can be used in this direction, **proposing a CIT competition**, and aiming to improve combinatorial test generators or to introduce new ones.

VII. Conclusions

In this paper, we have presented a benchmarking environment, integrated with CTWedge, for CIT test generators. Unlike the already available benchmarks, which are only based on the number of test cases produced, our framework allows testers to compare their own generators to the others already available, in terms of different features, namely number of test cases in the test suite, generation time, validity, and completeness. We have applied our framework to five different CIT generators (ACTS, CAgen, CASA, Medici, and PICT) using the benchmark models distributed together with the proposed framework, and we have shown how to integrate now generators. These benchmarks aim to aid the developers to test their own generators with different models, which vary greatly in terms of complexity, number and values of parameters, and number and types of constraints. CAgen has shown to be the best generator, among those analyzed, both in terms of the number of tests, generation time, completeness, and validity. As future work, we are planning to include in our framework a check of the minimality [3] for generated test suites as well. We believe that this framework can be used as a platform for tool competition, and this may bring several advantages, as shown in other fields, leading to the increase in generator performances, usability, and to the introduction of new ones.

References

[1] Pairwise Testing. http://pairwise.org/.

[2] PICT GitHub page. https://github.com/microsoft/pict.

[3] P. Arcaini, A. Gargantini, and P. Vavassori. Validation of models and tests for constrained combinatorial interaction testing. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, mar 2014.

[4] D. Beyer. International competition on software testing (test-comp). In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–175, Cham, 2019. Springer International Publishing.

[5] A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2–4):75–97, May 2008.

[6] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. Springer Berlin Heidelberg, 2004.

[7] A. Gargantini and M. Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, April 2018.

[8] A. Gargantini and P. Vavassori. Efficient combinatorial test generation based on multivalued decision diagrams. In E. Yahav, editor, *Hardware and Software: Verification and Testing*, pages 220–235, Cham, 2014. Springer International Publishing.

[9] A. Gargantini and P. Vavassori. Using decision trees to aid algorithm selection in combinatorial interaction tests generation. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2015.

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

[11] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, jul 2010.

[12] H. Jin and T. Tsuchiya. Constrained locating arrays for combinatorial interaction testing. *Journal of Systems and Software*, 170:110771, dec 2020.

[13] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international sat solver competitions. *AI Magazine*, 33(1):89–92, Mar. 2012.

[14] S. K. Khalsa and Y. Labiche. An orchestrated survey of available algorithms and tools for combinatorial testing. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, nov 2014.

[15] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys*, 43(2):1–29, jan 2011.

[16] J. Petke. Constraints: The future of combinatorial interaction testing. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. IEEE, may 2015.

[17] J. Petke, M. B. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Transactions on Software Engineering*, 41(9):901–924, 2015.

[18] I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*. ACM Press, 2011.

[19] R. Tzoref-Brill and S. Maoz. Modify, enhance, select: Co-evolution of combinatorial models and test plans. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 235–245, New York, NY, USA, 2018. Association for Computing Machinery.

[20] M. Wagner, K. Kleine, D. Simos, R. Kuhn, and R. Kacker. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index. In *International Workshop on Combinatorial Testing (IWCT 2020)*, 3 2020.

[21] A. Younes, P. Calamai, and O. Basir. Generalized benchmark generation for dynamic combinatorial problems. In *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation*, GECCO '05, page 25–31, New York, NY, USA, 2005. Association for Computing Machinery.

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