The Journal of Systems and Software 209 (2024) 111920

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Design, implementation, and validation of a benchmark generator for combinatorial interaction testing tools[☆]

Andrea Bombarda [*], Angelo Gargantini

Department of Engineering, University of Bergamo, Bergamo, Italy

ARTICLE INFO

Keywords:
Combinatorial testing
Benchmarks
Test generators
Validation

ABSTRACT

Combinatorial testing is a widely adopted technique for efficiently detecting faults in software. The quality of combinatorial test generators plays a crucial role in achieving effective test coverage. Evaluating combinatorial test generators remains a challenging task that requires diverse and representative benchmarks. Having such benchmarks might help developers to test their tools, and improve their performance.

For this reason, in this paper, we present BenCIGen, a highly configurable generator of benchmarks to be used by combinatorial test generators, empowering users to customize the type of benchmarks generated, including constraints and parameters, as well as their complexity. An initial version of such a tool has been used during the CT-Competition, held yearly during the International Workshop on Combinatorial Testing. This paper describes the requirements, the design, the implementation, and the validation of BenCIGen. Tests for the validation of BenCIGen are derived from its requirements by using a combinatorial interaction approach. Moreover, we demonstrate the tool's ability to generate benchmarks that reflect the characteristics of real software systems.

BenCIGen not only facilitates the evaluation of existing generators but also serves as a valuable resource for researchers and practitioners seeking to enhance the quality and effectiveness of combinatorial testing methodologies.

1. Introduction

Combinatorial Interaction Testing (CIT) (Petke et al., 2015) has been an active area of research in the latest years and has proven to be very effective to test complex systems, having multiple inputs or configuration parameters. The main purpose of CIT is to help testers in finding defects due to the interaction of different inputs or parameters, by testing this interaction systematically and by assuring that every t-uple of parameter values (i.e., an array of t elements, where each element is one of the parameters of the system under test with one of its possible values Niu et al., 2013) is tested at least once (Kuhn et al., 2004). In practice, testers provide an input parameter model (IPM) of a system under test (SUT), containing the possible values for each parameter, as well as any additional constraints between values of distinct parameters, and ask a test generator to produce a test suite.

During the years, several test generators have been proposed[1] by the community: research groups that actively work on the CIT area have been listed in Nie and Leung (2011), but many other recent groups and tools are not considered in that paper, while in Khalsa and Labiche (2014) a lot of algorithms and tools available for CIT are analyzed. Despite so many algorithms and tools for CIT have been developed with the intent of improving testing of software systems, paradoxically little attention has been given to testing and systematically and fairly evaluating those tools and algorithms. One major issue is the absence of a collection of benchmarks to be used for testing the correctness and evaluating the performance of the generators themselves. Many tools have only been evaluated on ad-hoc or unrealistic models, or small examples, missing some important and common problem characteristics. This becomes especially evident when dealing with problems that involve constraints, as they pose a greater challenge for test generators, and obtaining representative test IPMs from real scenarios can be difficult.

While evaluating CIT test generators, every research group has established its own procedure and benchmarks, and this can be limiting for many reasons: (a) some specific features, which may be common in practice, are not considered while testing the test generator; (b) on the contrary, uncommon features may be considered and, thus, bias the test outcome; (c) a limited amount of test IPMs may be available.

[☆] Editor: Professor Yan Cai.

[*] Corresponding author.
E-mail addresses: andrea.bombarda@unibg.it (A. Bombarda), angelo.gargantini@unibg.it (A. Gargantini).

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

https://doi.org/10.1016/j.jss.2023.111920
Received 5 July 2023; Received in revised form 21 October 2023; Accepted 4 December 2023
Available online 14 December 2023

Moreover, having a high number of benchmarks may foster the improvement of the performance of test generators, since they can be tested (and, thus, adapted) against different IPMs. This is the rationale behind the CT-Competition which is organized every year during the International Workshop of Combinatorial Testing.[2]

To address all these issues, in this paper, we present BenCIGen, a benchmark generator of IPMs that can be used by practitioners to generate synthetic IPMs for testing CIT generators. First, we design BenCIGen by building a feature model describing its configuration parameters and possible constraints among them. BenCIGen is built on the top of the CTWedge environment (Bombarda et al., 2021), and allows practitioners to generate a set of different benchmarks, with a configurable type, amount, and cardinality of parameters and constraints. In order to make the benchmarks as challenging as desired, BenCIGen allows users to configure the ratio of the generated IPMs, i.e., the fraction of the number of valid tests (or t-uples) over the total number of tests (or t-uples). We believe that this aspect is crucial for assessing the performance of a test generator under different (also in terms of complexity) use case scenarios. Lastly, BenCIGen only produces solvable IPMs, i.e., IPMs from which at least a test case can be generated. This is of paramount importance for making the use of generated benchmarks valuable for evaluating test generators: assessing the performance (time and test suite size) of test generators requires models that allow at least a test case. Non solvable models could be useful as well in order to test the correctness of test generators but not in evaluating their performance, and we may add the feature of generating also non solvable IPMs in future releases of BenCIGen.

We investigate the correctness of BenCIGen by using combinatorial test cases derived from its model, and we show how models available in the literature can fit inside those that can be generated from our tool. By demonstrating this aspect, we can state that BenCIGen can generate realistic IPMs, as challenging and complex as those used in practice for real systems, and, thus, that the benchmarks we generate are valuable for effectively testing CIT test generators.

The remainder of the paper is structured as follows. Section 2 describes the background on combinatorial testing and the measures we perform on each generated IPM. In Section 3 we present the requirements we set for the development of BenCIGen, while Section 4 introduces the design of our tool and the possible approaches for computing the two types of ratio and the solvability of an IPM. Section 5 shows BenCIGen and how we have implemented it, while in Section 6 we validate our tool by generating combinatorial tests from its requirements, and by showing how the majority of CIT models available in the literature can fit in those that our tool can generate. Finally, Section 7 presents related works on benchmarking combinatorial test generators, and Section 8 concludes the paper.

2. Background

Combinatorial test generators are tools used to generate test suites suitable for testing a system that has been modeled using an Input Parameter Model (IPM). It specifies parameters of a system under test (SUT), their possible values, as well as any additional constraints between values of distinct parameters. Formally, it can be defined as follows.

Definition 1 (*Input Parameter Model*)**.** Let S be the system under test, $P = \{p_1, \dots, p_n\}$ be a set of n parameters, where every parameter p_i assumes values in the domain $D_i = \{v^i_1, \dots, v^i_j\}$, let D be the set of all the D_i, i.e., $D = \{D_1, \dots, D_k\}$ and $C = \{c_1, \dots, c_m\}$ be the set of constraints over the parameters p_i and their values v^i_j. We say that $M = (P, D, C)$ is an *Input Parameter Model* for the system S.

```
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 a constrained combinatorial model

Table 1
An example of a pairwise ($t = 2$) test suite.

P1	P2	P3	P4
V2	False	V1	2
V2	True	V3	2
V2	False	V3	3
V2	False	V1	4
V2	True	V3	4
V2	False	V1	5
V2	True	V3	5
V2	True	V3	3

Given an IPM, test generators build a *test suite* TS, composed of several test cases tc_i, in which every parameter $p \in P$ has its own value. The main objective of a TS is to cover all the feasible interactions between t parameters, where t is the strength of the test suite.

Definition 2 (*T-wise Coverage*)**.** Let TS be the test suite for the IPM $M = (P, D, C)$, as defined in Definition 1, and be t its strength. We say that TS achieves the *t-wise coverage* if all the feasible t-uples among the parameters $p_i \in P$ and their values are covered by at least a test case in TS.

Based on the system to be modeled, the parameters may be of different types. In the work presented in this paper, we consider *Boolean* parameters, that can assume only the true and false values, *Enumerative* parameters, assuming values in a finite set, and *Integer ranges* parameters, assuming values between a lower and an upper bound (both Integers).

An example of IPM, in the CTWedge format (Gargantini and Radavelli, 2018), is given in Listing 1. It contains two enumerative parameters (P1 and P3), a single Boolean parameter (P2), and an integer range parameter (P4). Furthermore, it contains a set of three constraints, defined over the set of parameters. Table 1 shows the test suite achieving the pairwise (i.e., $t = 2$) coverage for the IPM in Listing 1.

In every IPM, for each constraint, it is possible to compute a complexity, which roughly measures the effort required by the combinatorial test generator when checking the satisfiability of the constraint. Formally, it can be defined as follows.

Definition 3 (*Complexity*)**.** Let $M = (P, D, C)$ be an IPM as defined in Definition 1. The *Complexity* of a constraint $c \in C$ is the number of binary logical operators and connectors in c, i.e., the number of AND, OR, implies (=>), and double implies (<=>). More formally, the complexity is represented by a function $Comp : C \rightarrow \mathbb{N}$.

For example, the complexity of the constraint

```
# P1 = true AND P2 = false #
```

is equals to 1, as only a binary logical operator or connector (i.e., the AND) is available. Instead, if we consider the constraint

[2] https://fmselab.github.io/ct-competition/.

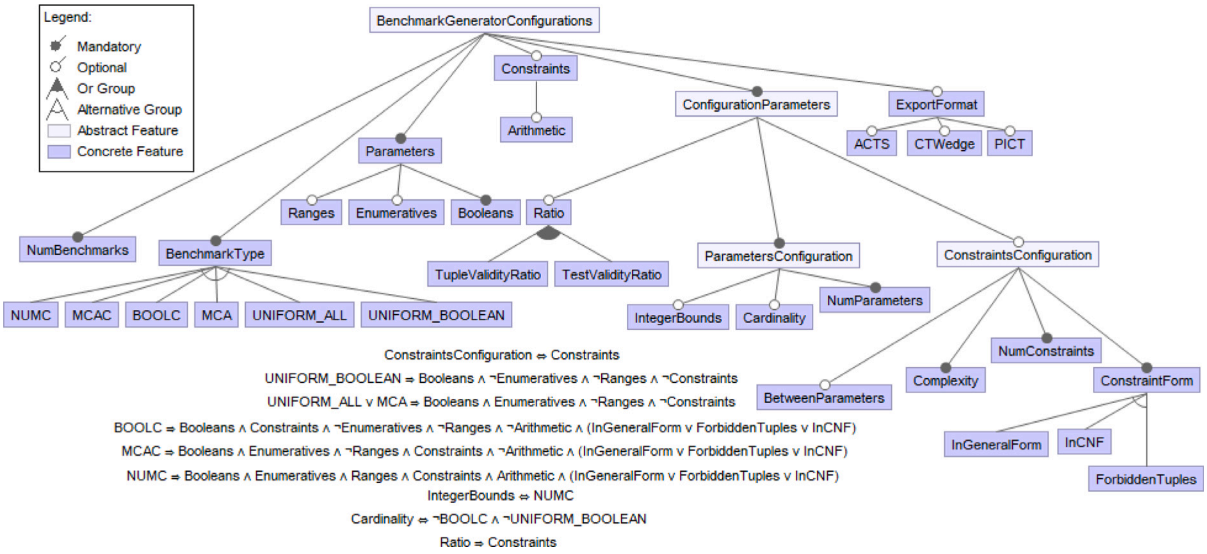

Legend:
Mandatory
Optional
Or Group
Alternative Group
Abstract Feature
Concrete Feature
BenchmarkGeneratorConfigurations
Constraints
Arithmetic
ConfigurationParameters
ExportFormat
ACTS
CTWedge
PICT
Parameters
Ranges
Enumeratives
Booleans
Ratio
TupleValidityRatio
TestValidityRatio
NumBenchmarks
BenchmarkType
NUMC
MCAC
BOOLC
MCA
UNIFORM_ALL
UNIFORM_BOOLEAN
ParametersConfiguration
IntegerBounds
Cardinality
NumParameters
ConstraintsConfiguration
NumConstraints
Complexity
ConstraintForm
BetweenParameters
InGeneralForm
InCNF
ForbiddenTuples
ConstraintsConfiguration ⇔ Constraints
UNIFORM_BOOLEAN ⇒ Booleans ∧ ¬Enumeratives ∧ ¬Ranges ∧ ¬Constraints
UNIFORM_ALL ∨ MCA ⇒ Booleans ∧ Enumeratives ∧ ¬Ranges ∧ ¬Constraints
BOOLC ⇒ Booleans ∧ Constraints ∧ ¬Enumeratives ∧ ¬Ranges ∧ ¬Arithmetic ∧ (InGeneralForm ∨ ForbiddenTuples ∨ InCNF)
MCAC ⇒ Booleans ∧ Enumeratives ∧ ¬Ranges ∧ Constraints ∧ ¬Arithmetic ∧ (InGeneralForm ∨ ForbiddenTuples ∨ InCNF)
NUMC ⇒ Booleans ∧ Enumeratives ∧ Ranges ∧ Constraints ∧ Arithmetic ∧ (InGeneralForm ∨ ForbiddenTuples ∨ InCNF)
IntegerBounds ⇔ NUMC
Cardinality ⇔ ¬BOOLC ∧ ¬UNIFORM_BOOLEAN
Ratio ⇒ Constraints

Fig. 1. The Feature Model representing the possible configurations and features of the benchmarks generator. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
# P1 => (P2 AND P3)#
```

the complexity is 2, as we have an AND connector and an implication.

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). In that case, none of the tests generated from an IPM will cover those t-uples and we say that they are *not feasible* or *invalid*. In order 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 4 (*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, due to the constraints, some of the tests that can be generated from an IPM by a combinatorial test generator may be not valid, i.e., they may violate one or more constraints; instead, tests complying with the constraints of the IPM are considered as *valid*. For this reason, 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 5 (*Test Validity Ratio*)**.** Let $M = (P, D, C)$ be the IPM for a system S, TS be the set of all possible test cases that can be generated when the constraints C of M are ignored. Let $TS_v \subseteq TS$ be the set of valid test cases, i.e., 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_v) over the total number of possible tests N (i.e., the cardinality of TS).

3. Requirements

During the development of BenCIGen, we aimed at creating a tool allowing users to generate a wide spectrum of IPMs, by specifying all the features and characteristics we have found in the other models available in the literature (see Section 6.2).

The possible configurations we wanted to include in BenCIGen generator are reported in the feature model in Fig. 1. In the following, we better describe the features and their meaning in detail:

- Each generation run may generate multiple benchmarks with the same characteristics. The number of benchmarks is configurable through the feature NumBenchmarks;
- Several different categories (BenchmarkType) of models may be generated, depending on the type of parameters and constraints, as reported in Table 2. In case one of the categories containing constraints is chosen, the Constraints may be selected and, possibly, contain Arithmetic operators;
- Depending on the benchmark type, different types of Parameters (Booleans, Enumeratives, or integer Ranges) can be present in the generated IPM;
- Depending on the selected benchmark type, the user may specify the following ConfigurationParameters:
  - the maximum accepted Ratio, as described in Section 2, which can be set as the TupleValidityRatio and/or TestValidityRatio;
  - regarding the parameters (Parameters Configuration), the user can select:
    - the Cardinality, limited between a lower and upper bound, for the parameters in the generated IPMs, only if not BOOLC neither UNIFORM_BOOLEAN are selected;
    - the integer ranges (IntegerBounds) in the models of the NUMC category;
    - the number of parameters NumParams to be present in the generated IPMs, included between a lower and upper bound;
  - regarding the constraints (Constraints Configuration), the user can select:
    - the number of constraints NumConstraints (whether applicable), included between a lower and upper bound;
    - the Complexity, included between a lower and upper bound, for the constraints in the generated IPMs, as described in Section 2;
    - whether to have constraints comparison BetweenParameters (e.g., PAR1 = PAR2) and not only comparisons between parameters and values (e.g., PAR1 = true);

Table 2
Types of benchmarks supported by the BenCIGen benchmark generator.

Benchmark Type	Parameters	Constraints
UNIFORM_BOOLEAN (UB)	Only Booleans	NO
UNIFORM_ALL (UA)	Uniform	NO
MCA (M)	MCA (Booleans and Enumeratives)	NO
BOOLC (BC)	Only Booleans	Randomly chosen between AND, OR, ⇔, NOT, ⇒
MCAC (MC)	MCA (Booleans and Enumeratives)	Randomly chosen between AND, OR, ⇔, NOT, ⇒, = (both $x = C$ and $x = y$, where x and y are parameters and C a constant of x), ≠
NUMC (NC)	Booleans, Enums and Integer ranges	Randomly chosen between AND, OR, ⇔, NOT, ⇒, = (both $x = C$ and $x = y$, where x and y are parameters and C a constant of x), ≠, mathematical and relational operations

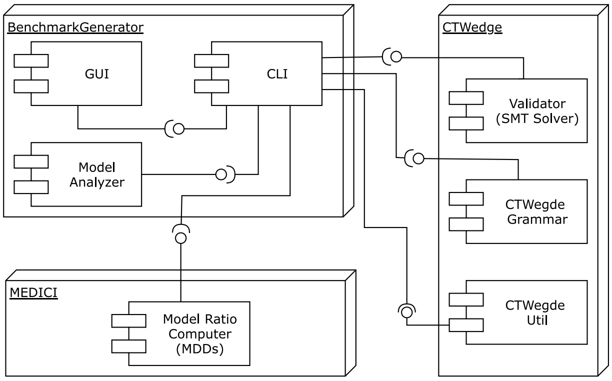


Fig. 2. Software architecture of BenCIGen.

  - whether the constraints (if they are applicable — see Table 2) need to be InCNF,[3] expressed as ForbiddenTuples or InGeneralForm. In the first case, each constraint is a conjunction (an AND) of one or more clauses, where a clause is a disjunction (an OR) of atomic predicates. In the second case, each constraint must express a forbidden tuple, i.e., in the form of NOT (P1=v1 AND P2=v2 AND ...), or (P1!=v1 OR P2!=v2 OR ...). Finally, in the third, an arbitrary composition of each constraint is allowed, i.e., a mix between conjunctions, disjunctions, implications, equivalences and negations can be used in any arbitrary order and combination;

- The generated benchmarks may be exported in different formats, such as ACTS (Yu et al., 2013), PICT (Microsoft Inc, 2023) and CTWedge (Gargantini and Radavelli, 2018). We decided to support these three different formats because they are the most used ones and, moreover, they allow for representing the same type of constraints and operators. Other formats, such as the CASA one, would require the transformation of the constraints and this would make the benchmarks not comparable.

All these configuration parameters may be set by the user prior to the benchmark generation. Moreover, considering that in real scenarios one may want to test its combinatorial test generator with models similar to those he/she already has, BenCIGen must provide an interface for extracting the configuration from a former IPM and generating models having similar characteristics. Finally, BenCIGen shall allow users to load a JSON file, such as the one in Listing 2, representing the dictionary of parameter name, type, and values to be used in the randomly created IPMs when tests of a specific domain are required.

Listing 2: Example of a dictionary JSON file for the Smartphone domain for BenCIGen

```
[
  {
    "name": "ScreenSizeInch",
    "type": "Integer",
    "lowerBound": 4,
    "upperBound": 7
  },
  {
    "name": "OS",
    "type": "Enum",
    "values" : [
      "android",
      "ios"
    ]
  },
  {
    "name": "WirelessCharge",
    "type": "Boolean"
  }
]
```

4. Design

In this section, we describe the architecture we have designed for BenCIGen, together with the strategies and approximations we used for computing relevant measures. The tool architecture is reported in Fig. 2. BenCIGen features a GUI and a CLI. The former aims at increasing the usability of the benchmark generator, but the business logic is completely implemented in the CLI. The latter includes all the functionalities of the benchmark generator, such as the pure generation, the check for the existence of at least a test derivable from the generated IPM (see Section 4.1), the computation of the tuple validity ratio (see Section 4.2) and test validity ratio (see Section 4.3).

The basic functionalities used by BenCIGen are offered by the CTWedge environment (Gargantini and Radavelli, 2018), including the CTWedge grammar definition, the utility functions (such as those generating the tuples, converting a CTWedge model in other formats, etc.), and the validation functionalities (exploited for checking the solvability of an IPM). In the following, we describe in detail the role of each component of the architecture, by explaining the method we have implemented for checking the solvability of an IPM and computing relevant ratios.

4.1. Existence of at least a test

When a benchmark is generated, it is important to check its solvability, i.e., the existence of at least a test case derivable from the IPM. This check is done by the CTWedge *validator* module in Fig. 2, which exploits an SMT Solver,[4] as presented in Bombarda et al. (2021). In particular, an SMT solver is a tool aiming to determine whether a mathematical formula is satisfiable or not, by using some modulo

[3] We support constraints in CNF as some generator, such as in the case of CASA (Garvin et al., 2009), may require constraints to be defined in that form.

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

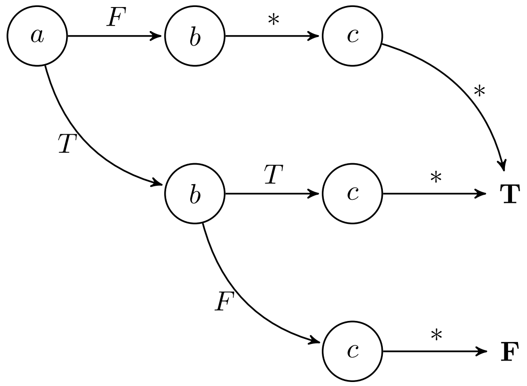


Fig. 3. MDD structure for the combinatorial problem in Listing 3. With $*$ we mean all the possible values.

theories. In our case, the formula we want to check is a Boolean formula composed by the conjunction of all the constraints and defined on the Cartesian product of all the domains of the parameters of the IPM under analysis.

The process to be followed for determining if at least a test case can be derived from an IPM is very straightforward. Each IPM generated by BenCIGen is translated in its own SMT context, containing all the variables and constraints of the IPM. More in detail, the parameters of an IPM are translated into SMT variables depending on their type:

- *Booleans* are translated into SMT Boolean variables;
- *Integer ranges* are translated into SMT integer variables. Furthermore, since ranges in combinatorial models are limited between a lower and an upper bound, it is necessary to add to the context an additional constraint specifying these limits. For example, if a range is defined in the combinatorial model as $P1 : [-4..3]$, in addition to the $P1$ integer variable, the following constraint is added: $P1 \geq -4$ AND $P1 \leq 3$;
- *Enumeratives* are translated into SMT integer variables. As for the normal integer ranges, when translating enumeratives, it is necessary to add to the SMT context a group of constraints limiting the values that can be taken by each enumerative. Furthermore, in this kind of transformation, it is necessary to use unambiguous numbers between parameters, in order to avoid different parameters assuming the same value. For example, if two enumeratives are defined in the combinatorial model as $P1 : \{A, B\}$ and $P2 : \{C, D\}$, the following is a valid mapping: $A \rightarrow 1$, $B \rightarrow 2$, $C \rightarrow 3$, and $D \rightarrow 4$. Moreover, for the parameter $P1$ the following constraint need to be added: $P1 \geq 1$ AND $P1 \leq 2$.

Additionally, all the other constraints of an IPM can be easily mapped to SMT formulas, exploiting the variables previously defined. Note that a combinatorial model may contain relational, mathematical, or comparison operators (between parameters or values) in general propositional formulas. All these aspects can be easily represented with operations between variables and values defined in an SMT context.

Then, if the context is SAT, it means that at least one test can be derived from the IPM and, thus, it can be accepted as benchmark.

4.2. Computation of the tuple validity ratio

To compute the tuple validity ratio r_{tp} we exploit the same formalism presented in Section 4.1, i.e., the CTWedge *validator* module in Fig. 2, similarly as done in Bombarda et al. (2023b). First, we build a complete SMT context ctx, containing all the parameters and constraints of the IPM, properly translated in SMT notation. Then, we iterate over all the t-uples $tp_i \in TP$ and we check if adding tp_i to ctx makes the context still satisfiable. In that case, it means that tp_i is valid, otherwise it is not. By doing so, we compute the number of valid t-uples v and, consequently, the tuple validity ratio r_{tp} as follows:

$$r_{tp} = \frac{v}{\#TP}$$

```
Model example2

Parameters:
a : Boolean
b : Boolean
c : {V1, V2, V3}

Constraints:
# a => b #
```

Listing 3: Example of a simple IPM in CTWedge format

4.3. Computation of the test validity ratio

One of the desired characteristics of the benchmark models is the test validity ratio r_{ts}, introduced and defined in Section 2. Only for small models the calculation of r_{ts} could be done by simply enumerating all the possible configurations and checking how many of them are valid. For large models, we have devised two techniques, one that is precise, but it is not suitable for any model, while the other is approximate, but it can be used even when the model contains arithmetic constraints.

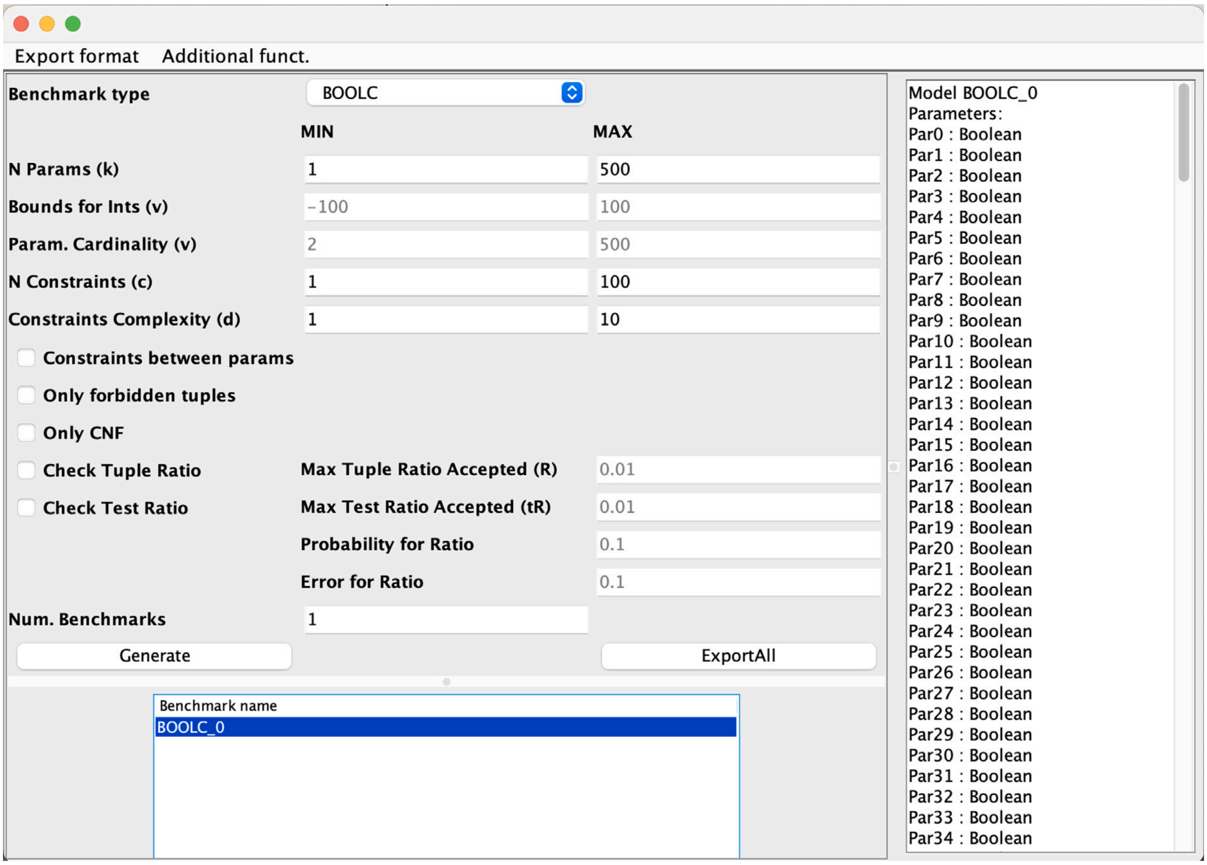
4.3.1. Using MDDs

To count how many combinations are valid, we rely on a data structure, called Multi-Valued Decision Diagrams, on which the MEDICI (Gargantini and Vavassori, 2014) test generator (see Fig. 2) is based. Indeed, most combinatorial problems can be easily represented by using an MDD identifying valid combinations that comply with the constraints of the IPM under analysis. Let us consider the IPM in Listing 3, which represents a combinatorial model with three parameters and a very simple constraint between a and b. With an MDD, as reported in Fig. 3, we can represent the validity of different parameter combinations. By counting how many paths lead to the T leaf, we can simply determine the number of valid tests without the need to generate each possible configuration and check if it is valid or not.

More in details, after having generated an IPM M, we can execute MEDICI with the option `--donotgenerate`. In this way, MEDICI translates M into its MDD representation, by starting with the definition of the nodes corresponding to the variables of the combinatorial model. The cardinality of the MDD (i.e., the number of paths starting from the root node to the true leaf) is the number of all the possible tests N. Then, we incrementally add all the constraints of M, and we compute again the cardinality of the MDD after all the constraints have been added. This second cardinality corresponds to the number of valid tests V for M when the constraints are considered. Thus, the test validity ratio is computed as follows:

$$r_{ts} = \frac{V}{N}$$

We emphasize that the cardinality of an MDD is not computed by enumerating all the possible assignments leading to the true leaf (although this would be possible Toda and Soh, 2016), but recursively visiting the MDD and computing the final cardinality by sums and products of the cardinality of partial MDDs, thus the complexity of this operation is much lower than that of path enumeration. However, although MDDs are very efficient in subset counting, not all combinatorial problems can be easily represented by an MDD. Indeed, as presented in Bombarda and Gargantini (2022, 2023a), MDDs allow users to represent in an optimized and memory-effective way only combinatorial problems not containing arithmetical or comparison operations between parameters and values (e.g., +, −, >, <, etc.), or



Benchmark type: BOOLC

	MIN	MAX
N Params (k)	1	500
Bounds for Ints (v)	-100	100
Param. Cardinality (v)	2	500
N Constraints (c)	1	100
Constraints Complexity (d)	1	10

☐ Constraints between params

☐ Only forbidden tuples

☐ Only CNF

☐ Check Tuple Ratio — Max Tuple Ratio Accepted (R): 0.01

☐ Check Test Ratio — Max Test Ratio Accepted (tR): 0.01

Probability for Ratio: 0.1

Error for Ratio: 0.1

Num. Benchmarks: 1

Benchmark name
BOOLC_0

Model BOOLC_0
Parameters:
Par0 : Boolean
Par1 : Boolean
Par2 : Boolean
Par3 : Boolean
Par4 : Boolean
Par5 : Boolean
Par6 : Boolean
Par7 : Boolean
Par8 : Boolean
Par9 : Boolean
Par10 : Boolean
Par11 : Boolean
Par12 : Boolean
Par13 : Boolean
Par14 : Boolean
Par15 : Boolean
Par16 : Boolean
Par17 : Boolean
Par18 : Boolean
Par19 : Boolean
Par20 : Boolean
Par21 : Boolean
Par22 : Boolean
Par23 : Boolean
Par24 : Boolean
Par25 : Boolean
Par26 : Boolean
Par27 : Boolean
Par28 : Boolean
Par29 : Boolean
Par30 : Boolean
Par31 : Boolean
Par32 : Boolean
Par33 : Boolean
Par34 : Boolean

Fig. 4. The BenCIGen GUI.

constraints comparing two different parameters (e.g., PAR1 = PAR2). Indeed, even if using MDDs would be technically feasible in those cases, we may likely have the problem of the combinatorial explosion of the number or complexity of constraints, thus leading to the impossibility of completely representing the combinatorial problem.

4.3.2. *Using a Monte Carlo approach*

When the MDD-based technique presented before is not applicable, we can rely on one of the basic approximate set counting algorithms that are based on the classical Monte Carlo method. These methods can be applied because we have a finite set, containing all the possible tests, U of known size N, and an efficient method for randomly choosing elements in U. We have also an efficient method to discover if a random test is valid or not (without using the solver, but simply by checking the truth value of each constraint when the assignments contained in the tests are set).

To estimate the ratio r_{ts}, we can simply take a sequence of n independent random tests by assigning a random value to each parameter in the model. Then we check if every test ts_i is valid or not, and we assign to x_i the value 0 if the ith test is not valid, otherwise, we assign to x_i the value 1. The total number of valid tests is $\sum_1^n x_i$.

The Monte Carlo-based estimator for r_{ts} that we indicate as $\tilde{r}_{ts}$ is simply:

$$\tilde{r}_{ts} = \frac{\sum_1^n x_i}{n}$$

It can be easily proved that this estimator is *unbiased*, i.e., as the sample size n increases the variance of the estimator decreases, improving the confidence of the estimation. If we could take all the possible N tests and count how many of them are valid, then we would get the right estimation. In most cases, we can only guarantee that the approximation is good enough if we take enough elements. In particular, the Zero-One Estimator Theorem (Karp et al., 1989) gives us a lower bound for the number of elements to be considered in order to make a correct prediction with probability $p > 0$ and a maximum error of $\varepsilon > 0$, i.e.

$$n \geq \frac{1}{r_{ts}} \cdot \frac{4 \cdot \ln \frac{2}{1-p}}{\varepsilon^2}$$

If this requirement is satisfied, then, our prediction $\tilde{r}_{ts}$ is a correct approximation of the ratio r_{ts} with probability:

$$Pr[(1-\varepsilon) \cdot r_{ts} \leq \tilde{r}_{ts} \leq (1+\varepsilon) \cdot r_{ts}] \geq p$$

When the user sets the desired ratio r_{ts}, we ask him/her to insert the desired probability p and to set the acceptable error ε, so we can compute for every model M the number n of samples needed for making a prediction which is a correct approximation. Then, after having estimated $\tilde{r}_{ts}$ we check whether it is included in the range $(1-\varepsilon) \cdot r_{ts} \leq \tilde{r}_{ts} \leq (1+\varepsilon) \cdot r_{ts}$. If the answer is yes, then we consider the model as satisfying the desired ratio r_{ts}. Otherwise, a new IPM has to be generated.

Example 1. For a desired IPM M, the user asks for $r_{ts} = 0.1$, $p = 75\%$, and $\varepsilon = 0.1$. The generator computes the number of required samples for making a correct approximation

$$n \geq \frac{1}{r_{ts}} \cdot \frac{4 \cdot \ln \frac{2}{1-p}}{\varepsilon^2} = 8,317.77$$

Thus, the generator takes 8318 random tests, and let us assume that 825 of them are valid, while 7493 are invalid. The estimated ratio is $\tilde{r}_{ts} = 0.099$, which is included in the range $[(1-\varepsilon) \cdot r_{ts}, (1+\varepsilon) \cdot r_{ts}] = [0.09, 0.11]$. Therefore, we can say that M has ratio $r_{ts} = 0.1$ with probability $p \geq 75\%$.

Algorithm 1 Algorithm for the generation of NUMC benchmarks

```
Require: $nBenchmarks$, the number of IPMs to be generated
Require: $\langle kMin, kMax \rangle$, the min. and max. number of parameters for each IPM
Require: $\langle lInt, uInt \rangle$, the lower and upper bounds for integer ranges
Require: $\langle vMin, vMax \rangle$, the min. and max. cardinalities
Require: $\langle cMin, cMax \rangle$, the min. and max. number of constraints for each IPM
Require: $\langle dMin, dMax \rangle$, the min. and max. constraint complexities
Require: $useCBtwP$, whether to use constraints between parameters
Require: $FT$, whether to use only forbidden tuples in constraints
Require: $CNF$, whether to use only constraints in CNF
Require: $r_{tp}$, the max. tuple validity ratio
Require: $useTupleRatio$, whether to consider $r_{tp}$ during IPMs generation
Require: $\langle r_{ts}, p, \varepsilon \rangle$, the max. test validity ratio, the probability, and the maximum error for the ratio
Require: $useTestRatio$, 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 $nB < nBenchmarks$ do
 3:     for $nAttempts \leftarrow 1$ to 10 do
            ▷ Randomly define parameters
 4:         $nParams \leftarrow$ randomBetween($kMin$,$kMax$)
 5:         $pList \leftarrow$ defineParams($nParams$, $lInt$, $uInt$, $vMin$, $vMax$)
            ▷ Randomly define constraints
 6:         $nC \leftarrow$ randomBetween($cMin$,$cMax$)
 7:         $cList \leftarrow$ defineCnstr($pList$, $nC$, $dMin$, $dMax$, $useCBtwP$, $FT$, $CNF$)
            ▷ Check that the generated IPM complies with the requirements
 8:         $model \leftarrow$ buildModel($pList$, $cList$)
 9:         if $model$.isSolvable() then
10:             if not $useTupleRatio$ or $model$.getTupleValidityRatio() $< r_{tp}$ then
11:                 if not $useTestRatio$ or $model$.getTestValidityRatio($p$, $\varepsilon$) $< r_{ts}$ then
12:                     $modelsList$.add($model$)
13:                     $nB \leftarrow nB + 1$; $nAttempts \leftarrow 0$
14:                     break
15:                 end if
16:             end if
17:         end if
18:     end for
19: end while
```

5. Implementation

This section describes the implemented tool for generating benchmark IPMs, available as a command-line tool and with a GUI (see Fig. 4). In both versions, the generator allows the user to work in two different ways:

- The parameters of interest, depending on the chosen benchmark type, can be manually set;
- The parameters of interest, including the benchmark type, can be automatically set by giving a baseline model in CTWedge format, which is analyzed by BenCIGen that extracts all the configuration parameter values (see Section 5.3 for further details).

After having fixed the parameters of interest, the benchmarks are randomly generated by BenCIGen. To give an intuition on how the benchmark generator produces the models, in Algorithm 1, we report the algorithm used for generating NUMC benchmarks. Note that the procedure is the same for the other benchmark categories, except for the type of constraints and parameters chosen.

The algorithm aims at producing $nBenchmarks$ IPMs with the desired characteristics. For each benchmark, initially, the tool extracts a random number of parameters (line 4) with bounds $kMin$ and $kMax$. Then, the set of the parameters to be included in the IPM is generated by the function defineParams, which randomly extracts the types and values for each parameter (line 5). The same approach is followed for constraints definition (lines 6 and 7). In Section 5.1 we will explain in detail the algorithm defining the parameters, and in Section 5.2 that defining the constraints. In this way, BenCIGen produces a single IPM (line 8) which now needs to be checked to see whether it is solvable (line 9) and, if the tuple validity ratio and/or the test validity ratio have to be met, possibly has the required ratios (lines 10 and 11). In that case, the model is added to the $modelsList$, otherwise, a new model is generated.

This process can last for a long time, especially if some check on the ratio is required. For this reason, we set a maximum number of $nAttempts$ of 10 trials for the single IPM. Note that different approaches may be used, especially when considering the ratio of IPMs, for producing only benchmarks complying with the requirements, such as adding one constraint per time and building incrementally the model. However, this may cause to be stuck in models where no constraint making the model solvable or complying with the ratio required can be added. As a future work, we may investigate this approach in order to solve its limitations (e.g., by using a backtracking strategy) and to avoid completely throwing away the generated IPM every time it is not compliant with the characteristics requested by the user.

5.1. Parameters definition

For every benchmark IPM, after having fixed the number of parameters, BenCIGen defines the type and values for each of them randomly.

Algorithm 2 Algorithm for the definition of parameters in the case of NUMC benchmarks

```
Require: $nParams$, the number of parameters
Require: $\langle lInt, uInt \rangle$, the lower and upper bounds for integer ranges
Require: $\langle vMin, vMax \rangle$, the min. and max. cardinalities
Ensure: $paramsList$, the list of the random parameters

 1: function DEFINEPARAMETERS($nParams$, $lInt$, $uInt$, $vMin$, $vMax$)
        ▷ Initially, no parameters have been defined
 2:     $paramsList \leftarrow \emptyset$; $nP \leftarrow 0$;
 3:     while $nP < nParams$ do
            ▷ Extract the type of the current parameter
 4:         $pType \leftarrow$ CHOOSERANDOM(Boolean, Enumerative, Range)
 5:         if $pType$ = Boolean then
                ▷ Boolean variables require only to set their name
 6:             $param \leftarrow$ CREATENEWBOOLEAN("PAR" + nP)
 7:         else if $pType$ = Enumerative then
                ▷ Enumeratives require to set all possible values, which are random as well
 8:             $nValues \leftarrow$ RANDOMBETWEEN($vMin$, $vMax$)
 9:             $param \leftarrow$ CREATENEWENUM("PAR" + nP)
10:             $i \leftarrow 0$
11:             for $i < nValues$ do
12:                 $param$.values.add("PAR" + nP + "_" + $i$)
13:                 $i \leftarrow i + 1$
14:             end for
15:         else
                ▷ For integer ranges, we need to set the lower and upper bound
                ▷ but the cardinality must be considered as well
16:             $param \leftarrow$ CREATENEWRANGE("PAR" + nP)
17:             $\langle l, u \rangle \leftarrow$ RANDOMRANGE($lInt$, $uInt$, $vMin$, $vMax$)
18:             $param$.setRange($l$, $u$)
19:         end if
20:         $paramsList$.add($param$)
21:         $nP \leftarrow nP + 1$
22:     end while
23:     return $paramsList$
24: end function
```

The type, and consequently the values, of each parameter depends on the requested type of benchmarks (e.g., for UNIFORM_BOOLEAN only Boolean parameters are chosen, for MCA and MCAC, the type of each parameter is chosen between Boolean and enumeratives, while for NUMC also integer ranges are considered). As previously done for the general algorithm, we here give an explanation of the parameters' definition algorithm for NUMC IPMs in Alg. 2, but for the other categories, the procedure is the same, except that fewer types of parameters are used.

In general, for each parameter, at line 4, the algorithm randomly defines the parameter type (among Booleans, enumeratives, and integer ranges). If the parameter is Boolean (line 5) no additional setting is required. On the other hand, if an enumerative or range has to be created, additional information has to be set. In the former case (line 7), the number of values is randomly set within the bounds given by $vMin$ and $vMax$. In the latter case (line 15), the bounds of the range have to be set by the function randomRange. At this stage, BenCIGen considers both the integer bounds $lInt$ and $uInt$, but observes the cardinality bounds ($vMin$ and $vMax$) as well.

The described process is repeated for the number of parameters required and, then, at the end, a full $paramList$ is produced, containing parameters with different types and values.

5.2. Constraints definition

After having set the parameters of the IPM, it is necessary to add (whether applicable), the constraints. As for parameters, the constraints are randomly defined, both in terms of number and complexity. Instead, unlike the parameters, the constraints are all composed in a very similar way, regardless of the benchmark type. For NUMC IPMs, relational and mathematical operations are possible as well.

The constraints definition process is based on Alg. 3. After having defined the number of constraints, as shown in Alg. 1, the algorithm randomly chooses the complexity of every single constraint (line 4). Then, the composition of the constraint is performed by the generateConstraint recursive function. It is designed for composing the constraint as an AND, OR, implication, or double implication of atomic constraints (line 12). This process is recursively repeated while the remaining complexity is greater than 1 and populates, for each constraint, the left and the right part (line 14 and 15). Then, when the complexity reaches the value 1, a single atomic constraint is created (line 17), in the form of $PAR = val$ or $PAR_X = PAR_y$ (or $\neq$, $>$, $\geq$, $<$, $\leq$, depending on the type of the IPM being generated). Note that the decision on the operator to be used in the atomic predicate, as well as the decision on whether to compare two parameters or a parameter and a value is randomly made by the benchmark generator thanks to the function createAtomicConstraint.

5.3. Model analyzer

In this section, we analyze the *Model analyzer* component, which is used by BenCIGen for automatically extracting the configuration depending on an already available CTWedge IPM M.

For what concerns the benchmark type (as reported in Table 2), first, BenCIGen looks for the constraints in M. If no constraint is found, then, the decision on the benchmark type is taken depending on the

Algorithm 3 Algorithm for the definition of constraints in the case of NUMC benchmarks

```
Require: $pList$, the list of parameters
Require: $nCnstr$, the number of constraints to be generated
Require: $\langle dMin, dMax \rangle$, the min. and max. constraint complexities
Require: $CBtwP$, whether to use constraints between parameters
Require: $FT$, whether to use only forbidden tuples in constraints
Require: $CNF$, whether to use only constraints in CNF
Ensure: $cList$, the list of the constraints generated

function defineCnstr($pList$, $nCnstr$, $dMin$, $dMax$, $CBtwP$, $FT$, $CNF$)
    ▷ Initially, no constraints have been defined
    $cList \leftarrow \emptyset$; $nC \leftarrow 0$;
    while $nC < nCnstr$ do
        ▷ Randomly define the complexity
        $compl \leftarrow$ randomBetween($dMin$, $dMax$)
        ▷ Generate the constraint
        $cList$.add(generateCnstr($pList$, $compl$, $CBtwP$, $FT$, $CNF$)
        $nC \leftarrow nC + 1$
    end while
    return $cList$
end function

function generateCnstr($pList$, $compl$, $CBtwP$, $FT$, $CNF$)
    if $compl > 1$ then
        ▷ Recursively define the constraint
        $cType \leftarrow$ chooseRandom(AND, OR, IMPL, DBLIMPL)
        $c \leftarrow$ createConstraintByType($cType$)
        ▷ Set the left and right part
        $c$.setLeft(generateCnstr($pList$, $(compl - 1)/2$, $CBtwP$, $FT$, $CNF$))
        $c$.setRight(generateCnstr($pList$, $(compl - 1)/2$, $CBtwP$, $FT$, $CNF$))
    else
        ▷ Define an atomic constraint
        $c \leftarrow$ createAtomicConstraint($pList$, $CBtwP$, $FT$, $CNF$)
    end if
    return $c$
end function
```

type of parameters. When all parameters are Booleans, the model is considered as an `UNIFORM_BOOLEAN` instance; if all parameters are all with the same size, the model is considered as an `UNIFORM_ALL` instance, while in all the other cases it is an `MCA` instance. On the other hand, if constraints are present in M, then the benchmark type is within `BOOLC`, `MCAC`, or `NUMC`. The first category is chosen when all parameters are Booleans; the second is assigned when not all parameters are Boolean but no integer ranges are available in M, while the last benchmark type is chosen in all the other cases.

Depending on the benchmark type, identified by the Model analyzer, following actions are taken by BenCIGen. The number of parameters (both minimum and maximum — k in the BenCIGen GUI) are automatically set by counting the parameters in M, as well as done for the constraints (c in the BenCIGen GUI). The minimum and maximum cardinality for the parameters (v in the BenCIGen GUI), or the bounds for integer ranges, are computed by enumerating all the parameters in M and identifying the one with the lowest and the one with the highest cardinality. Regarding the minimum and maximum constraints complexity (v in the BenCIGen GUI), they are computed by applying Definition 3 to all the constraints in an iterative way, in order to find the lowest and highest values. More specifically, the Model analyzer, extracts the complexity from each single constraint by recursively visiting it and identifying the number of binary logical operators or connectors.

The Model analyzer can also extract from M the type of the constraints, i.e., if all of them are expressed as forbidden tuples or in CNF. This analysis is done by iteratively visiting all the constraints and exploiting the modelanalyzer utility in the CTWedge framework (Bombarda et al., 2021).

Finally, the *tuple validity ratio* and the *test validity ratio* are extracted by M by applying the same strategies presented in Sections 4.2 and 4.3, and choosing the most suitable approach depending on the benchmark type, i.e., on the type of the parameters and constraints in M.

5.4. BenCIGen *usage*

In this section, we delve into the workflow and usage of BenCIGen and its GUI for generating a set of benchmarks. Additional instruction for the CLI version of BenCIGen are available at https://github.com/fmselab/CIT_Benchmark_Generator/tree/main/BenchmarkGenerator.

First, the user needs to set the *Benchmark type*, by choosing one of those proposed by BenCIGen. In this way, the configuration parameters of interest in the left column (as shown in Fig. 1 and explained in Section 3) are automatically enabled, filled with default values, and can be set by the user. After having set all the parameters, the *Generate* button allows for generating the benchmark IPMs complying with the chosen configuration parameters.

When the generation process terminates, the names of the IPMs are shown in the list in the lower part of the left column of BenCIGen, and the full model is shown when the user clicks on one of them. The exporting process is very straightforward: first, the formats of interest are set through the *Export format* button in the menu bar; then the IPMs are exported in the chosen formats when the user clicks on the *ExportAll* button.

Table 3
The test suite derived from the feature model of the requirements.

Param	ts_1	ts_2	ts_3	ts_4	ts_5	ts_6	ts_7	ts_8	ts_9	ts_{10}	ts_{11}	ts_{12}	ts_{13}	ts_{14}	ts_{15}	ts_{16}	ts_{17}
#B	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Type	NC	NC	NC	MC	MC	MC	BC	BC	BC	M	UA	UB	BC	UA	M	UB	MC
Ratio	–	X	–	X	–	X	–	X	X	–	–	–	X	–	–	–	X
r_{tp}	–	X	–	X	–	X	–	X	–	–	–	–	X	–	–	–	–
r_{ts}	–	X	–	–	–	X	–	X	X	–	–	–	X	–	–	–	X
Int.Bounds	X	X	X	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Card.	X	X	X	X	X	X	–	–	–	X	X	–	–	X	X	–	X
#P	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CnstrConf	X	X	X	X	X	X	X	X	X	–	–	–	X	–	–	–	X
BtwParam	–	X	X	–	X	–	X	–	X	–	–	–	X	–	–	–	–
Complx.	X	X	X	X	X	X	X	X	X	–	–	–	X	–	–	–	X
#C	X	X	X	X	X	X	X	X	X	–	–	–	X	–	–	–	X
CnstrForm	G	C	F	G	C	F	G	C	F	–	–	–	G	–	–	–	F
Params.	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Ranges	X	X	X	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Enums.	X	X	X	X	X	X	–	–	–	X	X	–	–	X	X	–	X
Booleans	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Cnstr.	X	X	X	X	X	X	X	X	X	–	–	–	X	–	–	–	X
Arithmetic	X	X	X	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Ex.Format	–	X	X	–	X	–	X	–	–	X	–	X	X	X	–	–	X
ACTS	–	X	X	–	X	–	–	–	–	X	–	X	X	X	–	–	–
CTWedge	–	X	–	–	X	–	X	–	–	X	–	X	–	X	–	–	X
PICT	–	X	X	–	–	–	X	–	–	X	–	X	X	X	–	–	X
Outcome	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 4
Summary of the values set for each non-boolean feature during test execution.

	#B	r_{tp}	r_{ts}	Int.Bounds	Card.	#P	Complx.	#C
Min	–	–	–	−50	2	2	1	1
Max	–	–	–	50	30	30	15	20
p	–	–	75.0%	–	–	–	–	–
ε	–	–	0.1	–	–	–	–	–
Value	10	0.1	0.1	–	–	–	–	–

If a baseline IPM is available and the user wants to generate new IPMs with the same characteristics, the model analyzer component introduced in Section 5.3 can be triggered by clicking on the *Set baseline IPM* button under the *Additional funct.* menu in the menu bar.

Finally, the use of a domain specific dictionary[5] is allowed by the *Set dictionary* button under the *Additional funct.* menu in the menu bar. When a dictionary is set, the name and values of the parameters are chosen among those provided in the dictionary, if available. Otherwise, the regular naming strategy is adopted.

6. Validation

In this section, we report how we have validated BenCIGen by testing its functionalities and ensured that generated benchmarks reflect real-world software systems' characteristics by showing that the majority of the benchmarks available in the literature can be generated by our tool.

6.1. CIT for validation

In order to validate and test BenCIGen, we have applied a dogfooding technique: we derive from the feature model in Fig. 1, describing the requirements of our tool, a combinatorial test suite with strength $t = 2$. The test suite has been generated, after having automatically translated the feature model in a CTWedge model, using ACTS, and, with only 17 tests it allowed us to effectively test BenCIGen. The test cases are reported in Table 3, where $\#B$ indicates the number of benchmarks, the $Type$ is expressed with the abbreviations introduced in Table 2, $\#P$ represents the number of parameters, $\#C$ the number of constraints, and the ConstraintForm is C if constraints need to be in CNF, G if the general form is required, or F if forbidden tuples are used. Note that abstract features (those in light blue in the feature model in Fig. 1) are not reported in the test suite, since they are not actual features of the generators, but they are only used for grouping other features.

Some of the parameters that can be selected or unselected in the generated test suite actually correspond to many parameters that have to be set during test execution (e.g., the number of parameters $\#P$ requires to set the maximum and minimum number). Therefore, in Table 4 we report the values we set in each test case for each non-boolean feature, but we emphasize that these values are reported only for completeness and replicability of the tests, and the same results would be obtained with every other values. Note that we decided to use $\#B = 10$ in order to have multiple examples to check for every test case, considering that models are generated randomly by the benchmark generator. Then, for each test case ts_i, after having set all the configuration parameters, we generate the benchmarks and check that every generated IPM conforms to its expected properties, in terms of parameters, constraints, ratio, and complexity.

The code executing the tests is available online in BenCIGen's official repository https://github.com/fmselab/CIT_Benchmark_Generator, while the outcome of each test execution is reported in the last row of Table 3.

6.2. External validation

Ensuring the similarity between artificially generated benchmarks and real models is of utmost importance when evaluating generators. This is crucial to avoid bias in the evaluation process, as models that do not accurately represent real systems can introduce distortions in the assessment of generator performance and correctness. For this reason, in this section, we show that a significant number of models taken from the literature can be obtained by at least one configuration of our CIT benchmark generation.

Table 5[6] shows the 767 models we have considered and the characteristics extracted from them by the *modelanalyzer* part of our CIT

[5] Examples of dictionaries are available at: https://github.com/fmselab/CIT_Benchmark_Generator/tree/main/BenchmarkGenerator/dictionaries.

[6] For the models in the NUMC category, r_{ts} is an estimation computed with the Monte Carlo-Based approach, as explained in Section 4.3, with $n = 1000$. For some models (those with the *) it was not possible to compute both r_{ts} and r_{tp} because of their high complexity (Thüm, 2020).

Table 5
Summary values for the IPMs taken from the literature. In columns where two values are reported, they represent the lower and the upper bounds. FT reports the number of models in that category having forbidden tuples, while CF those with constraints in CNF.

Src	#Ms	Type	#	#P	Int bnd.	Card.	#C	Comp.	FT	CF	r_{tp}	r_{ts}
Petke et al. (2015)	7	BC	1	10	–	–	1	5	0	1	0.99	0.75
		MC	6	7–14	–	2–10	6–83	1–38	0	6	0.75–0.92	0.002–0.250
Jin et al. (2020)*	11	BC	7	65–1639	–	–	108–4664	1–100	0	0	0.70–0.93	0.000–0.000
		MC	4	72–6295	–	2–27	94–9842	1–352	0	0	0.63–0.82	0.000–0.000
Segall et al. (2011)	18	BC	1	5	–	–	7	1–6	0	1	0.90	0.250
		MC	17	4–35	–	2–13	3–388	1–8	13	17	0.75–1.00	10^{-5}–0.654
Garvin et al. (2010)	35	MC	35	30–199	–	2–6	5–49	1–9	0	35	0.80–0.99	10^{-5}–0.324
Microsoft Inc (2023)	28	M	7	6–18	–	1–7	–	–	–	–	–	–
		BC	1	7	–	–	2	1	0	0	0.98	0.625
		MC	20	2–33	–	1–11	1–36	1–9	0	0	0.80–0.99	10^{-7}–0.813
Tzoref-Brill and Maoz (2018)	112	M	15	3–61	–	1–500	–	–	–	–	–	–
		BC	1	23	–	–	19	1–4	0	0	0.94	0.024
		MC	93	4–118	–	1–166	1–381	1–252	4	4	0.08–0.99	10^{-13}–1.000
		NC	2	8–8	0–3	2–4	7–11	1–4	0	0	0.35–0.91	0.088–1.000
Johansen et al. (2011)*	16	BC	8	28–1397	–	–	34–3633	1–13	0	0	0.61–0.93	10^{-12}–0.001
		MC	8	7–6295	–	2–27	7–9842	1–352	0	0	0.47–0.82	10^{-13}–0.103
Bombarda et al. (2022)	300	UB	53	2–20	–	–	–	–	–	–	–	–
		UA	50	2–20	–	2–20	–	–	–	–	–	–
		M	47	2–20	–	2–50	–	–	–	–	–	–
		BC	54	2–20	–	–	1–23	1–15	0	0	0.25–1.00	10^{-5}–0.937
		MC	60	2–19	–	2–50	1–38	1–14	0	4	0.01–1.00	10^{-16}–1.000
		NC	36	2–17	(−99)–100	1–199	1–13	1–14	1	1	0.01–1.00	0.001–1.000
Bombarda et al. (2023)	240	UB	15	7–29	–	–	–	–	–	–	–	–
		UA	15	7–25	–	2–15	–	–	–	–	–	–
		M	30	7–30	–	1–15	–	–	–	–	–	–
		BC	36	6–43	–	–	1–46	1–14	0	1	0.33–1.00	10^{-9}–0.906
		MC	114	4–199	–	1–15	1–37	1–15	0	44	0.02–1.00	10^{-12}–1.000
		NC	30	6–30	(−100)–111	1–16	1–24	1–14	0	0	0.06–1.00	0.001–0.830

benchmark generator, i.e., the part meant to extract the configuration from a given IPM where the generation from a baseline model is chosen (see Section 5). All models and data extracted from their analysis are available at https://github.com/fmselab/CIT_Benchmark_Generator/blob/main/BenckmarkGenerator/external_validation.

Data reported in Table 5 show that in all the considered cases, we have been able to classify the models from the literature in the categories handled by BenCIGen, and all the categories we can generate with BenCIGen have been found in the literature. The only limit we found is dealing with very complex models having thousands of parameters and constraints (derived from Software Product Lines and not natively representing IPMs, though), for which computing the ratio is not feasible in an exact way. Considering the data obtained by analyzing the IPMs available in the literature, we can conclude that by setting BenCIGen in the same way as in those benchmarks, we can obtain plausible models with the same features as real-world IPMs.

7. Related work

Benchmarking combinatorial test generators is of paramount importance since it allows both for assessing the correctness of the tools (i.e., their ability to produce valid and complete test suites, covering all the desired t-way interactions) and for evaluating their performance. Several works have been presented in the past, trying to evaluate test generators and identifying those having the best performance, both in terms of generation time and test suite size. For example, in Bombarda et al. (2021), the authors presented a benchmarking environment, based on CTWegde (Gargantini and Radavelli, 2018) which allows the comparison between test generators that can be easily included by extending some selected Eclipse extension points. In that work, the authors compared some of the most well-known generators (ACTS Yu et al., 2013, MEDICI Gargantini and Vavassori, 2014, CAgen Wagner et al., 2020, PICT Microsoft Inc, 2023, and CASA Garvin et al., 2009), but only on a limited set of 196 IPMs taken from the literature.

In this paper, instead, we focus more on generating benchmark models and not on their execution for comparing test generators. Indeed, finding real IPMs is not so easy in the literature, since many of those used in research works are not distributed due to IP limitations. Some analyses, when real highly configurable system models are needed, have been conducted by deriving combinatorial models from software product lines, such as in Johansen et al. (2011) and Jin et al. (2020). This is not always the optimal approach, since the translation of an SPL into an IPM requires some assumption (such as the way in which alternative groups are translated, or the way in which abstract or hidden features are treated) that may vary the complexity of the generated IPMs.

This is the reason why we focus on benchmark generation. This problem is not completely new and it is tackled also by other works. For example, in Younes et al. (2005), the authors proposed 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. This approach is different w.r.t. that we use in this paper since we do not require any solution to be known and, thus, we can generalize better test models. Moreover, in Ansotegui and Torres (2023), the authors propose a generator for benchmark IPMs, but only a limited set of features is addressed. For example, when considering constraints, only models containing Boolean parameters can be generated, while the tool presented in our paper supports also enumeratives and integer ranges.

Benchmark generation is a common approach for comparing different methods, techniques, and tools (Hasselbring, 2021). It has been widely adopted especially in the context of competitions but not only. For instance, in Derks et al. (2023) the authors introduce `vpbench`, which simulates the evolution of a variant-rich system. The tool generates an evolution together with metadata that explains it — like in our case we generate a benchmark together with its type. In Ferrer et al. (2011), an automatic benchmark generator of java programs is presented. As done in our work, it is configurable by the user which can include in the generated code interesting features, and the reachability of each branch is assured (as we do for the validity of the IPMs). The application of benchmarks is not limited only to pure software systems, but sometimes is applied even in systems embedding hardware. For example, benchmarks generated by exploiting machine learning are used to test computer networks (Cerquitelli et al., 2023).

8. Conclusions

Testing and comparing combinatorial test generators is of paramount importance for the improvement, both in terms of performance and correctness, of the tools developed by practitioners in combinatorial testing. However, this process requires the availability of a high number of benchmarks representing real-world examples and grasping all the aspects of interest.

For reducing this gap in evaluating test generators, in this paper, we have presented BenCIGen, a generator of benchmark IPMs. It is fully configurable by users, that can decide the type of parameters and constraints to be included in each model, their number and complexity, as well as the properties of the IPMs themselves (e.g., the ratios and the existence of at least one valid test case).

Its applicability has already been demonstrated by its use during all the past editions of the CT Competition, held yearly during the International Workshop on Combinatorial Testing. Moreover, in this paper, we have further extended the tool and unit-tested it by using a combinatorial test suite directly derived from its requirements. As shown by the external validation activity, in which we have compared the IPMs available in the literature with those generable with BenCIGen, we believe that our tool can be profitably used for evaluating test generators with synthetically generated benchmarks having the same characteristics as real-world systems.

As a future work, we may investigate approaches allowing BenCIGen to solve the limitation of throwing away the generated IPM every time it is not solvable or compliant with the ratios requested by the user. Moreover, we may include the generation of not solvable IPMs. This would allow users to test their generators not only when the model can be solved, but also in negative cases, and to verify that the generators are actually able to identify that condition.

CRediT authorship contribution statement

Andrea Bombarda: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Angelo Gargantini:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The paper reports the link to all data and code used in the experiments.

Acknowledgement

This work has been partially funded by PNRR - ANTHEM (AdvaNced Technologies for HumancentrEd Medicine) - Grant PNC0000003 — CUP: B53C22006700001 - Spoke 1 - Pilot 1.4.

References

Ansotegui, Carlos, Torres, Eduard, 2023. A Benchmark Generator for Combinatorial Testing. Techreport, arxiv.org.

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

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

Bombarda, Andrea, Gargantini, Angelo, 2023a. Incremental generation of combinatorial test suites starting from existing seed tests. In: 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE.

Bombarda, Andrea, Gargantini, Angelo, Calvagna, Andrea, 2023b. Multi-thread combinatorial test generation with SMT solvers. In: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing. SAC '23, Association for Computing Machinery, New York, NY, USA.

Bombarda, Andrea, Wagner, Michael, Leithner, Manuel, 2022. Ct-Competition 2022 page. https://github.com/fmselab/CIT_Benchmark_Generator/tree/main/Benchmarks_CITCompetition_2022.

Bombarda, Andrea, Wagner, Michael, Leithner, Manuel, 2023. Ct-Competition 2023 GitHub page. https://github.com/fmselab/CIT_Benchmark_Generator/tree/main/Benchmarks_CITCompetition_2023.

Cerquitelli, Tania, Meo, Michela, Curado, Marilia, Skorin-Kapov, Lea, Tsiropoulou, Eirini Eleni, 2023. Machine learning empowered computer networks. Comput. Netw. 230, 109807.

Derks, Christoph, Strüber, Daniel, Berger, Thorsten, 2023. A benchmark generator framework for evolving variant-rich software. J. Syst. Softw. 203, 111736.

Ferrer, Javier, Chicano, Francisco, Alba, Enrique, 2011. Benchmark generator for software testers. In: IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, pp. 378–388.

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

Gargantini, Angelo, Vavassori, Paolo, 2014. Efficient combinatorial test generation based on multivalued decision diagrams. In: Hardware and Software: Verification and Testing. Springer International Publishing, pp. 220–235.

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

Garvin, Brady J., Cohen, Myra B., Dwyer, Matthew B., 2010. Evaluating improvements to a meta-heuristic search for constrained interaction testing. Empir. Softw. Eng. 16 (1), 61–102.

Hasselbring, Wilhelm, 2021. Benchmarking as empirical standard in software engineering research. In: Evaluation and Assessment in Software Engineering. ACM.

Jin, Hao, Kitamura, Takashi, Choi, Eun-Hye, Tsuchiya, Tatsuhiro, 2020. A comparative study on combinatorial and random testing for highly configurable systems. In: Testing Software and Systems. Springer International Publishing, pp. 302–309.

Johansen, Martin Fagereng, Haugen, Øystein, Fleurey, Franck, 2011. Properties of realistic feature models make combinatorial testing of product lines feasible. In: Model Driven Engineering Languages and Systems. Springer Berlin Heidelberg, pp. 638–652.

Karp, Richard M., Luby, Michael, Madras, Neal, 1989. Monte-Carlo approximation algorithms for enumeration problems. J. Algorithms 10 (3), 429–448.

Khalsa, Sunint Kaur, Labiche, Yvan, 2014. An orchestrated survey of available algorithms and tools for combinatorial testing. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering. IEEE, pp. 324–334.

Kuhn, D.R., Wallace, D.R., Gallo, A.M., 2004. Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. 30 (6), 418–421.

Microsoft Inc, 2023. PICT GitHub page. https://github.com/microsoft/pict.

Nie, Changhai, Leung, Hareton, 2011. A survey of combinatorial testing. ACM Comput. Surv. 43 (2), 1–29.

Niu, Xintao, Nie, Changhai, Lei, Yu, Chan, Alvin T.S., 2013. Identifying failure-inducing combinations using tuple relationship. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. IEEE.

Petke, Justyna, Cohen, Myra B., Harman, Mark, Yoo, Shin, 2015. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. IEEE Trans. Softw. Eng. 41 (9), 901–924.

Segall, Itai, Tzoref-Brill, Rachel, Farchi, Eitan, 2011. Using binary decision diagrams for combinatorial test design. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM.

Thüm, Thomas, 2020. A BDD for Linux? In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume a - Volume a. ACM.

Toda, Takahisa, Soh, Takehide, 2016. Implementing efficient all solutions SAT solvers. ACM J. Exp. Algorithmics 21, 1–44.

Tzoref-Brill, Rachel, Maoz, Shahar, 2018. 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. ACM.

Wagner, Michael, Kleine, K., Simos, Dimitris, Kuhn, R., Kacker, R., 2020. Cagen: A fast combinatorial test generation tool with support for constraints and higher-index. In: International Workshop on Combinatorial Testing (IWCT 2020).

Younes, Abdunnaser, Calamai, Paul, Basir, Otman, 2005. Generalized benchmark generation for dynamic combinatorial problems. In: Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation. ACM.

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

Andrea Bombarda is a research fellow in the FOSELab (Formal Methods and Software Engineering Laboratory) at the University of Bergamo (Italy). His research topics are mainly in the context of quality assurance for medical software and systems, by applying rigorous methods and suitable software engineering processes, aiming at improving the effectiveness and the rapidness of the medical software certification process. He received his Ph.D. in Engineering and Applied Sciences from the University of Bergamo (Italy).

More information is available at https://cs.unibg.it/bombarda/.

Angelo Gargantini is a full professor in Computer Science and Engineering at the University of Bergamo (Italy), and he is the director of the FOSELab (Formal Methods and Software Engineering Laboratory). He received his Ph.D. in computer engineering from the Politecnico of Milan. Before joining the University of Bergamo, he has worked for the Politecnico of Milan, the Naval Research Laboratory in Washington DC, and the University of Catania.

His research focuses on automated testing techniques, model-based testing, mutation testing, and the application of formal methods in software validation and verification. More information is available at https://cs.unibg.it/gargantini/.