SN Computer Science (2025) 6:383
https://doi.org/10.1007/s42979-025-03937-y

ORIGINAL RESEARCH

On the Completion of Partial Combinatorial Test Suites

Andrea Bombarda[1] · **Angelo Gargantini**[1]

Received: 21 November 2023 / Accepted: 28 March 2025

Abstract
Combinatorial Interaction Testing is a widely used method for testing intricate systems. In most cases, the test suites are generated from scratch. However, there are cases when testers may want to reuse existing tests, in order to include them in a new test suite, both for enhancing the performance of the generation process or because those tests are valuable for checking the functioning of the system under test in critical conditions. In this paper, we propose a general framework for dealing with existing test suites using combinatorial test generators. We also discuss the definition of partial tests and test suites, and the scenarios in which partial tests should or could be reused. Finally, we compare the most common tools for completing test suites, namely ACTS, PICT, and pMEDICI+, using different incompleteness levels in the seeds. ACTS with seeds generally performed the best in terms of test suite size and generation time. The other two tools, namely PICT and pMEDICI+, were slower and produced larger test suites on average. We have found that using seeds could sometimes come with a cost, especially in the scenario where test cases are partial and completing them is not always cost-effective in terms of generation time. The choice of re-using or throwing away existing tests must be based on use case-specific requirements. We do not recommend using seeds when they are composed of partial test cases, providing that they are not required for some other reason. On the contrary, we envision the use of partial test suites when a test suite with higher strength is needed.

Keywords Test seeds · Combinatorial testing · Test completion

Introduction

Combinatorial Interaction Testing (CIT) has been the subject of extensive investigation over numerous years, demonstrating its significant value in testing intricate systems, particularly those with numerous input or configuration parameters.

In most cases, the generation of a combinatorial test suite is done from scratch: given a model in a certain format, the state-of-the-art tool of choice can generate a test suite that achieves the desired coverage (and satisfies all the constraints present in the model). However, there are some cases in which some tests, sometimes *partial* tests, are already available and the tester wants to reuse them or at least evaluate if they can be reused. Combinatorial test suites can be *partial* in two different ways, i.e., they can be composed of complete test cases but some interaction is not covered (thus, additional test cases are needed), or some test case does not specify the value of all parameters (thus, they need to be completed). The reuse of possibly partial tests has not been extensively studied, although there are several tools that support the use of *seeds* which are nothing but parts of tests that can be reused.

For all these reasons, in this paper, we want to tackle the problem of the test generation by completing partial test suites, by exploiting combinatorial test generators supporting the test generation from seeds. One goal is to better define the meaning of *partial* tests and test suites and to define the roots for such incompleteness. For example, sometimes some tests could become partially invalid (for example if they were generated from a model that has changed) or a test suite could become incomplete (for example if the testing requirements have changed).

Another goal is to identify the scenarios in which these partial tests are better reused instead of discarded.

Moreover, we want to collect several tools and algorithms that support the completion of existing test suites, and try to compare them in order to assess their strong and weak

✉ Andrea Bombarda
andrea.bombarda@unibg.it

Angelo Gargantini
angelo.gargantini@unibg.it

[1] Department of Management, Information and Production Engineering, University of Bergamo, Bergamo, Italy

Published online: 15 April 2025

points and to evaluate the advantages and disadvantages of this approach.

This work extends that presented in [7] by adding additional tools and scenarios. In particular, in this paper, we compare three different tools supporting the completion of partial test suites, namely ACTS, PICT, and pMEDICI+ under different working scenarios. The objective is to identify which is the best-performing tool among them and, for each scenario, to establish which is the setting leading to the best performance, in terms of generation time and test suite size. We have identified three main scenarios in which completing test suites or test cases can be useful: when the tester wants to complete an existing incomplete test suite (TSCP scenario), when the tester wants to increase the strength of an existing test suite (SINC scenario), or when a test suite is composed of non-complete test cases, because, e.g., some parameter has been added to the combinatorial model (TCCP scenario). For these three scenarios, we have devised a set of research questions and designed experiments to give us the answers to those questions.

With our experiments, we have compared the performances of the three test generators in the three scenarios and we have investigated the impact of the percentage of missing tests or assignments (in the following referred to as *incompleteness level* or *incompleteness percentage*) on them. We have discovered that ACTS with seeds is, in general, the best-performing tool both in terms of test suite size and generation time. Instead, the other two tools, namely PICT and pMEDICI+, are on average slower or generate larger test suites. Interestingly, using seeds can also be counterproductive. In the SINC scenario, most of the time, seeds can help tools in working faster and producing more optimized test suites. Nevertheless, our experiments in the TCCP and TSCP scenarios have shown that, especially when looking at the generation time, completing the test cases or test suites is not always cost-effective.

The paper is structured as follows. Section Background and Definitions introduces basic definitions and background about combinatorial testing and partial test cases or test suites. Moreover, it introduces the scenarios we consider in our analysis. In Section Tools Supporting Completion, we present the tools we have analyzed in our work, namely ACTS, PICT, and pMEDICI+ which we have further extended for this work. Section Experiments presents the experimental methodology, together with the research questions we are interested in responding to, and the results we obtained with our experiments. Section Threats to Validity discusses the possible threats to the validity of our findings, while Section Related Work introduces the related work on the completion of test suites in the context of combinatorial testing. Finally, Section Conclusions concludes the paper.

Background and Definitions

Before analyzing the tools supporting the completion of partial test suites, we first introduce the basic concepts of CIT and we define when a test and a test suite can be considered *partial* and in which cases it may happen.

Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) is a systematic approach to testing the interactions between different features of a system. Combinatorial test generators are used to generate a test suite that covers all combinations of interest of input values for the features. The goal of CIT is to reduce the number of test cases needed to test the system, while still ensuring that all possible t-way interactions are covered.

A generic combinatorial model M, also known as input parameter model (IPM), is composed of a set of parameters for the system under test (SUT), their possible values, together with any additional constraints between values of distinct parameters. Combinatorial test generators are fed with an IPM as input and generate a combinatorial test suite with the desired strength t, meaning that every t-way interaction between parameters and their values is covered in at least one test case. Some tools optionally accept also *test seeds* that are discussed in this paper. In the following, the definition of *test case* or, more simply, *test* is given.

Definition 1 (*Test*) Let $M = \langle P, C \rangle$ be an IPM for a constrained combinatorial problem, where P is the set of parameters and C is the set of constraints. A test ts is a function that for every parameter p in P returns a valid value for p, such that ts satisfies the constraints C.

Partial Tests

Since we want to deal with incomplete tests, Definition 1 of test is rather restrictive because it requires that a test ts is *complete* and gives a valid value for each parameter in P. It can be easily extended in order to allow partially defined tests.

Definition 2 (*Partial test*) Let $M = \langle P, C \rangle$ be an IPM for a constrained combinatorial problem. A *partial test* ts is a function defined on a proper subset $P'_{ts} \subset P$ that for every parameter p in P'_{ts} returns a valid value, and such that exists an extension of ts to the entire P satisfying the constraints C.

Intuitively, a partial test is a test that assigns values only to some parameters, not all of them. However, the parameters that are not assigned by ts may not be free to

take any value they please as some of the values may lead the test to clash with one or more constraints.

We can briefly identify the cases in which partial tests are of interest and still worth considering for completion.

(a) Old test cases can become partial when the IPM changes. A typical example of partial tests occurs when the set of parameters P is extended by adding additional ones. In this case, all the old test cases will miss the value for the new parameters and have to be completed.
(b) Sometimes the IPM changes the constraints C among the parameters. In this case, some tests may become invalid, so the designers decide to remove all the assignments that cause this invalidity. This is particularly useful if the designer wants to reuse parts of the old test cases, and knows how to keep only the part of the tests that is still valid.
(c) Additionally, some partial test cases may have been manually written to test critical conditions of the systems which do not depend on the complete set of parameters. In this case, a partial test may be easier to write and the designer can ask the generator to complete it.

To be more precise, in this paper, we want to deal not only with incomplete tests but in general with incomplete test suites. We can extend the definition of partial test, also to entire test suites.

Definition 3 (*Partial test suite*) Let $M = \langle P, C \rangle$ be an IPM for a constrained combinatorial problem, where P is the set of parameters and C is the set of constraints. Let TS be a test suite for M. We say that TS is *partial* w.r.t. strength t, if there exists at least a t-tuple which is not covered by any of the $ts \in TS$ or if there exist at least a partial test ts in TS.

Partial test cases are not the only possible incompleteness for a test suite. A test suite may either contain partial test cases or do not cover some interactions. We can distinguish two orthogonal cases in which a test suite is incomplete.

(a) The simplest case occurs when every test is complete, but the TS does not achieve the desired combinatorial coverage. We say that TS is *vertically incomplete*.
(b) TS contains at least a partial test, but the desired combinatorial coverage is already achieved by TS. We say that in this case TS is *horizontally incomplete*.

In the most general case, a test suite is incomplete in both directions (vertically and horizontally): some tests are incomplete, and the coverage is not achieved.

Partial test cases fed to a combinatorial test generator are also referred to in the literature as *seed tests*, or, more simply, *seeds*. Seeds can be user-defined tests that are prescribed, for instance by a requirement specification [10]. They represent the first set of tests that must be included at the start into the test suite. For this reason, they constitute the *old* test suite and they have to be guaranteed to be included in the final test suite [12] produced by a test generator, which is then completed by adding new test cases (or by completing the partial ones) in order to achieve the desired t-way coverage.

Scenarios

Given the preliminary motivations and the definition of partial test cases and test suites, we have identified three different concrete scenarios that we report in Table 1. These scenarios represent three likely applications of test completion algorithms and they will constitute the base for our experiments presented in Section Experiments.

TSCP: ***Test Suite ComPletion*** In this scenario, we consider a test suite TS_1 to be partial, but the tests in it are complete. It may be composed of complete test cases that have been manually written or derived by applying other methodologies. Their goal may be testing critical or specific conditions of the SUT [10], and practitioners want to include those tests in the final test suite. Under this assumption, the test suite TS_1 is likely missing some of the t-tuples of interest.

If one considers the traditional approach in which incomplete test suites are not supported, the TSCP scenario would be tackled by generating a new TS_2 from

Table 1 Use case scenarios for partial tests

ID	Description	Goal	Incompleteness
TSCP	Test cases are themselves complete, but some test case is missing, so the test suite is not complete	Test suite completion, for achieving the t-wise coverage	Vertical
SINC	A complete test suite for the t-wise coverage is available and a new one for the $(t+1)$-wise is needed	Strength increasing	Vertical
TCCP	Test cases are not complete, i.e., some of the parameters do not have assigned values	Test suite completion, for achieving the t-wise coverage. New test cases might be added, or not (if the partial ones can be completed without adding new tests)	Horizontal/Mix

scratch and, at the end, by adding the tests in TS_1 to TS_2 (and removing possible duplicates). This may lead to having some t-way interaction covered multiple times and, thus, to a non-optimality of the test suite. On the contrary, if partial test suites are supported during generation, TS_1 can be considered as a baseline from which TS_2 can be built up.

In this scenario, the test suite is considered to be *vertically incomplete* since TS_1 is only composed of tests assigning values to all parameters of the IPM, but some test cases are missing in order to cover all the possible t-tuples.

SINC: ***Strength INCrease***

In this scenario, we consider a test suite TS_1, containing complete tests, to be partial because it has been generated for a lower strength t_1 and, after experiments, testers have decided to increase the strength of test generation to $t_2 > t_1$. As in the previous scenario, some of the (t_2)-interactions will not be covered by the tests in TS_1, and new tests are required in order to have a TS_2 achieving the (t_2)-wise coverage.

If one considers the traditional approach, the SINC scenario would be tackled by completely throwing away TS_1 and by generating a new test suite TS_2 with higher strength from scratch. On the contrary, if partial test suites are supported, TS_1 can be considered as a baseline from which TS_2 can be built up.

In this scenario, the test suite is considered to be *vertically incomplete* since TS_1 is only composed of tests assigning values to all parameters of the IPM.

TCCP: ***Test Case ComPletion***

In this scenario, we consider a test suite TS_1 to be partial because it is composed of partial test cases (as defined in Definition 2). Those test cases may originate from different sources. For example, they may have been manually written, but not all the parameters are set in all the tests. Alternatively, the test cases in TS_1 may be initially complete but, for some reason, the IPM which they have been derived from, may have changed and some assignments have to be removed from the tests in order to keep them valid. For this reason, the test suite TS_1 is likely missing some of the t-tuples of interest.

If one considers the traditional approach, the TCCP scenario would be tackled by generating a completely new test suite TS_2 from scratch and, at the end, by somehow completing the tests in TS_1 and adding them to TS_2. Instead, if partial tests are supported by the test generator, the partial test cases in TS_1 can be considered as a baseline from which TS_2 can be built up.

In this scenario, the test suite is considered to be, *horizontally incomplete*, since test cases require some parameter to be set in order to be completed. Moreover, in some cases, new test cases may be added to cover all the t-way interactions, thus the test suite is also *vertically incomplete*. To be more general, we refer to this condition as *mixed incompleteness*.

Tools Supporting Completion

In this section, we present the tools we analyze in this paper, which have been selected among those supporting partial test suite seeding. By analyzing the literature, we found that in recent years several tools supporting seeds have been proposed, such as pMEDICI+ [7], ACTS [21], PICT [2], CAgen [19], INWARD [11], and Jenny [1]. In this work, however, we will focus on the first three tools. Additionally, for each of these tools, we report the way in which partial (or not present) test cases have to be seeded to the tool.

pMEDICI+

During IWCT 2023, we presented pMEDICI+ [7], a tool that allows the generation of combinatorial test suites

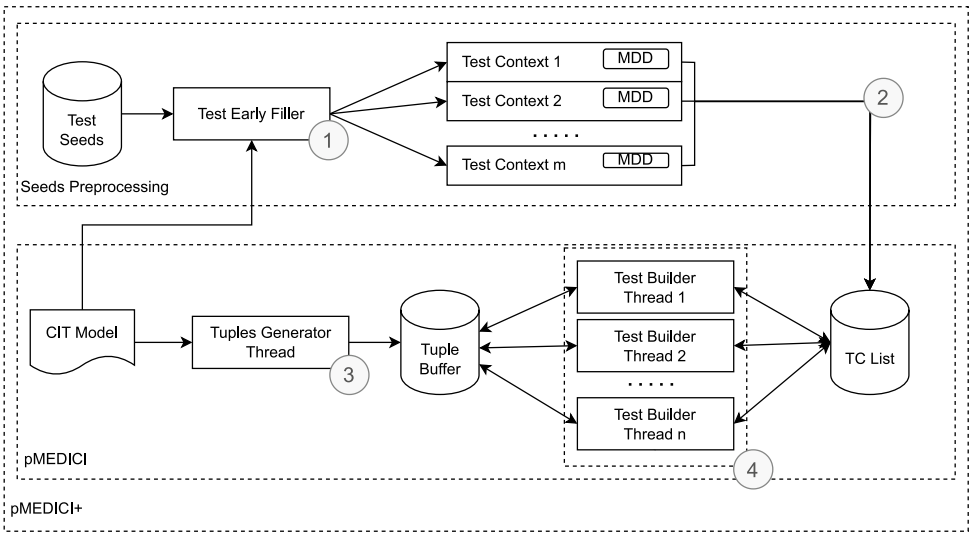


Fig. 1 Data flow in the pMEDICI+ tool

starting from seeds. It exploits the basic structure of pMEDICI [6], based on multi-threading, and extends it by adding an optional pre-processing stage in which the seeds are handled and initialized to be used as a starting point for generating the new test suite. The data-flow of pMEDICI+ is shown in Fig. 1 and described in the following.

Seeds Preprocessing

The preprocessing stage is composed of activities 1 and 2 in Fig. 1. Initially, the seeds (e.g., the old test suite), composed of *m* test cases, are processed and each test case is translated into the corresponding *Test Context* (as defined in [7]) by the Test Early Filler procedure reported in Algorithm 1.

Algorithm 1 Test Early Filler procedure: Preprocessing step of pMEDICI+

Require: $seedTests$, the seeds (e.g., an old test suite)
Require: $citModel$, the combinatorial model
Ensure: $tcList$, the list of test contexts created starting from the seeds

1: **for each** $seed \in seedTests$ **do**
2: $tc \leftarrow newTestContext(citModel)$
▷ Fill the test context with values taken
▷ from the seed
3: **for each** $p \in citModel.getParList()$ **do**
4: $val \leftarrow seed.getVal(p)$
5: **if** $val \neq NULL$ **then**
▷ Check if the new assignment is
▷ compatible with constraints
6: **if** $tc.isCompAssign(\langle p, val\rangle)$ **then**
7: $tc.addAssignment(\langle p, val\rangle)$
8: **end if**
9: **end if**
10: **end for**
▷ Check if at least one assignment has
▷ been added to the context
11: **if not** $tc.isEmpty()$ **then**
12: $tcList.add(tc)$
13: **end if**
14: **end for**

The preprocessing phase cyclically takes each *seed* from the *seedTests* list (line 1), and creates a new test context *tc*, which is initially empty with its own internal MDD (line 2) representing the parameters and the constraints of the combinatorial model. For each assignment in *seed*, the test context *tc* is continuously updated after having checked that the analyzed assignment is not empty (line 5) and compatible with the assignments committed to *tc* so far (line 6). If these two conditions are satisfied, then the test context *tc* is updated by adding the new assignment (line 7).

This operation is necessary and performed not at the test-case level but at the single-assignment level, i.e., one assignment at a time. This allows pMEDICI+ to accept seed tests that contain also invalid assignments: in this case only valid assignments are kept. Despite this feature is not employed in this work, it is a distinguishing characteristic of pMEDICI+ w.r.t. other tools. In this way, at the end of the pre-processing phase, the test contexts represent only (partial) valid test cases. Then, all test contexts created are stored in a *tcList* (line 12), which is the one used by the pMEDICI subcomponent. Note that this pre-processing phase is performed in single-thread mode since from our preliminary experiments we have not observed significant improvement by parallelizing the pre-processing stage.

Algorithm 2 Incremental test generation step of pMEDICI+

Require: $TupBuffer$, the buffer containing the tuple already produced and ready to be consumed
Require: $tcList$, the list of all the test contexts previously created during the pre-processing stage
Require: $citModel$, the CIT model

1: **while not** $TupBuffer.isEmpty()$ **do**
▷ Extract the tuple from the tuple buffer
2: $\langle par, val\rangle \leftarrow TupBuffer.extractFirst()$
▷ Try to find a test context which implies
▷ the tuple
3: $tc \leftarrow findImplies(tcList, \langle par, val\rangle)$
4: **if** tc is not $NULL$ **then**
5: **continue**
6: **end if**
▷ Try to find a test context that is
▷ compatible with the tuple $\langle par, val\rangle$
7: $tc \leftarrow findCompatible(tcList, \langle par, val\rangle)$
8: **if** tc is not $NULL$ **then**
9: $tc.addAssignment(\langle par, val\rangle)$
10: **continue**
11: **end if**
▷ Create a new empty test context
12: $tc \leftarrow newTestContext(citModel)$
13: **if** $tc.isCompAssign(\langle par, val\rangle)$ **then**
14: $tc.addAssignment(\langle par, val\rangle)$
15: **end if**
16: **end while**

Test Suite Completion

After having pre-processed and filtered all the seeds, new test cases are generated with the regular procedure executed by pMEDICI. Initially, given the strength t and the combinatorial model *citModel*, all tuples are generated by the Tuples Generator Thread (step 3 in Fig. 1) and stored into a tuple buffer.

Then, n test builder threads (step 4 in Fig. 1) execute in parallel the operations reported in Algorithm 2. Each builder inserts, if possible, the selected tuple in one of the test contexts stored in the *tcList*, by checking if the tuple is already implied or can be covered by one of them (*findImplies* and *findCompatible* methods at lines 3 and 7). If no compatible test context is found, the test builders build a new test context, containing only the constraints of the analyzed model and no assignments, and check if the considered tuple is compatible (lines 12 and 13). If the tuple is compatible, it is added to the newly created test context; otherwise, it means that the tuple can not be covered because it clashes with the constraints of the IPM.

Tool Extensions

For the work presented in this paper, we have further extended and fixed pMEDICI+, in order to make it applicable to more scenarios and to solve some of the issues we found thanks to the second edition of the CT-Competition .[1]

First, we found that invalid test suites were generated when the combinatorial models included parameters having a single value (e.g., representing constants used in constraints). This scenario was not supported by the mddlib[2] library we used to model constraints and parameters. For this reason, we have extended the library functionalities and, now, pMEDICI+ is able to deal with this kind of parameters without producing invalid test cases.

Then, in order to support all the scenarios listed in Section Background and Definitions, we have extended the seed pre-processing part which now allows seed tests to be horizontally partial. Indeed, pMEDICI+ previously supported only complete test cases as seeds (even if a part of them could be invalid for the considered IPM). Thus, with the updated version of pMEDICI+, if a test contains the `*` value for a parameter, that assignment will be skipped during the seed pre-processing, and that parameter will be set to a valid value by pMEDICI+ during the test suite completion phase (see Section Test Suite Completion).

Finally, pMEDICI+ is now enriched with an optimization that lets the user choose the order in which parameters have to be considered for generating the tuples to cover. In particular, pMEDICI+ now implements the parameter ordering strategy `IN_ORDER_SIZE_DESC` (OD) which allows the generation of tuples starting from the parameters assuming the highest number of values, `IN_ORDER_SIZE_ASC` (OA) which allows to first generate the tuples starting from the parameters assuming the lowest number of values, `RANDOM` (RD), which shuffles the list of parameters before starting the tuple generation process, and `AS_DECLARED` (AD), which considers the parameters in the order in which they are declared inside the IPM. We emphasize that in the following we will consider pMEDICI+ with the `IN_ORDER_SIZE_DESC` ordering strategy since our preliminary experiments have shown that this is the option leading to the best performance.

ACTS

ACTS [21] is one of the most used and well-tested combinatorial test generators. Starting from an initial covering array (CA), the IPO strategy [15] builds a combinatorial test set by adding columns one at a time and appending rows to cover any uncovered interactions if necessary. ACTS implements multiple different greedy IPO algorithms, such as IPOG, IPOG-F, and IPOG-F2 [13], which only differ in how the horizontal extension is implemented. IPOG is the default algorithm, is the fastest of the three, and iterates the rows from top to bottom. IPOG-F and IPOG-F2 both further optimize the order in which rows are iterated by applying dynamic programming techniques and a heuristic respectively.

In terms of constraint handling, ACTS also implements several techniques, ranging from Minimal Forbidden Tuples (MFTs) and Necessary Forbidden Tuples (NFTs) to CSP solvers, allowing the users to select the constraint handling method that best suits their needs.

With ACTS, users may set a seed test suite in `csv` format, which can be used as a starting point for the test generation process. The tool is able to deal with test suites both composed of test cases that are themselves complete or not. In the former case, ACTS will add new test cases to the test suite in order to achieve the intended t-wise coverage. In the latter, the `*` symbol marks those values that are not set, and ACTS will complete all the incomplete test cases during the generation process. We emphasize that, to the best of our knowledge, if a test case in the seeds does not comply with one of the constraints of the combinatorial model, it is completely discarded by ACTS and, thus, it will not be used as a seed test.

[1] https://fmselab.github.io/ct-competition/results/2023/Competition2023.html.

[2] https://github.com/colomoto/mddlib.

PICT

PICT [2] is a combinatorial test generator based on the one-test-at-a-time technique. It was mainly introduced for pairwise ($t = 2$) testing, but also supports the generation of general t-way test sets. It is developed by Microsoft and is written in C++ and is open source. The test generation algorithm behind PICT is based on a greedy and deterministic heuristic. It builds one test at a time by assigning values to each parameter that maximize the number of covered t-way interactions and do not violate any constraints, which is evaluated with a forbidden tuple approach.

As previously introduced for ACTS, also PICT is able to deal with partial test suites. Indeed, the test generation process can be started by feeding a test suite as a set of seeds, in the form of a tab-separated variant of the popular comma-separated values (CSV) format, including a header line containing the names of parameters. Seeds can be themselves complete or not. In the former case, PICT will add new test cases to the test suite in order to achieve the intended t-wise coverage. In the latter, the columns corresponding to not assigned parameters are left empty, and PICT will complete all the incomplete test cases.

As for ACTS (see Section ACTS), if a test case in the seeds clashes with one of the constraint of the combinatorial model, it is completely discarded by PICT and, thus, it will not be used as a seed test.

Experiments

In this section, we first describe the experimental methodology allowing the repeatability of our experiments aimed to evaluate the analyzed tools. Then, we describe the research questions we have used for guiding the experiments, we analyze the results we have obtained, and finally we compare the performance of the three analyzed tools (see Section Tools Supporting Completion) on the benchmark models reported in Table 2.

All benchmarks have been generated with the same generator[3] [8] used during the CT-Competition[4] which is held yearly during the International Workshop on Combinatorial Testing and, thus, they follow the same standards and terminology used for the competition. We emphasize that only the models supported by all three analyzed tools have been considered in our experiments and, for this reason, the `NUMC` instances (i.e., those with integer parameters and arithmetical constrains between them) are missing.

Each execution has been performed using a timeout of 300 seconds, as per CT-Competition rules.

Moreover, in each experiment in which the performance of the tools are analyzed in terms of test suite size and generation time, when a tool timed out on an instance, we have considered its generation time equal to 301 seconds, while for the size analysis each instance timing out has been set with size to 10^5, which is greater than any other size obtained by all tools.

When two or more generators are to be compared, we have decided to use the one-tailed Wilcoxon-Signed Rank test [20], a general test to compare the distributions in paired samples that does not require data to be normally distributed. Given x the measure to be compared between the two tools (e.g., generation time or test suite size), the test is performed using a significance level $\alpha = 0.05$, the null hypothesis H_0 stating that the mean values of x in the two compared tools are equal (i.e., $\bar{x}_1 = \bar{x}_2$), and the alternative hypothesis H_1 stating that the measure for the first tool is lower than that of the second one. In the following, when reporting the results of statistical tests we use tables sorted from the best-performing tool to the worst one in terms of the considered measure. We report, for each tool T_i, the comparison with the nearest competitor T_j for which the test confirms the better performance of T_i w.r.t. T_j, i.e., for which H_0 can be discarded. When T_i and T_j are not neighbors, we also report the comparison between T_i and all other T_k, with $i < k < j$ and we mark those comparisons with $\times$, meaning that H_0 cannot be discarded.

The replication package, containing the results of the experiments (including size and time of each tool, in each scenario and configuration, on each model) and the script we have used for analyzing them is available online at https://github.com/fmselab/ct-tools/tree/main/IncrementalCitGeneration.

Generation of the seeds All analyzed scenarios require to have seeds to test tools€™ performance. In principle, we could generate those seeds by using one of the three tools we analyze (i.e., ACTS, pMEDICI+, or PICT). However, this may imply favouring that specific tool, which could take advantage of a test suite generated by its own algorithm. For this reason, we decided to generate test seeds by using another tool, KALI [9]. As revealed by previous analyses [16], KALI is, in general, a tool producing larger test suites.[5] However, we do not see this non-optimization as a threat to validity of our approach, as seeds could be manually written by testers and not generated automatically, thus a low optimization level is somehow expected.

[3] https://github.com/fmselab/CIT_Benchmark_Generator.

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

[5] https://fmselab.github.io/ct-competition/results/2023/Competition2023.html.

Table 2 Characteristics of the benchmark models used for tool evaluation, where #P indicates the number of parameters and #C the number of constraints

Model name	#P	#C	Model name	#P	#C	Model name	#P	#C
UNIFORM_BOOL_0	25	0	UNIFORM_ALL_7	28	0	BOOLC_4	25	6
UNIFORM_BOOL_1	21	0	UNIFORM_ALL_8	7	0	BOOLC_5	11	5
UNIFORM_BOOL_2	6	0	UNIFORM_ALL_9	18	0	BOOLC_6	6	1
UNIFORM_BOOL_3	22	0	MCA_0	25	0	BOOLC_7	13	1
UNIFORM_BOOL_4	18	0	MCA_1	9	0	BOOLC_8	20	26
UNIFORM_BOOL_5	23	0	MCA_2	30	0	BOOLC_9	17	14
UNIFORM_BOOL_6	24	0	MCA_3	27	0	MCAC_0	26	28
UNIFORM_BOOL_7	29	0	MCA_4	13	0	MCAC_1	17	9
UNIFORM_BOOL_8	10	0	MCA_5	26	0	MCAC_2	17	16
UNIFORM_BOOL_9	13	0	MCA_6	12	0	MCAC_3	10	1
UNIFORM_ALL_0	18	0	MCA_7	7	0	MCAC_4	22	9
UNIFORM_ALL_1	28	0	MCA_8	30	0	MCAC_5	7	9
UNIFORM_ALL_2	17	0	MCA_9	10	0	MCAC_6	30	15
UNIFORM_ALL_3	11	0	BOOLC_0	24	14	MCAC_7	29	17
UNIFORM_ALL_4	11	0	BOOLC_1	22	2	MCAC_8	27	11
UNIFORM_ALL_5	20	0	BOOLC_2	17	13	MCAC_9	23	23
UNIFORM_ALL_6	23	0	BOOLC_3	30	3			

TSCP: Test Suite Completion

In this section, we report the experiments executed on the TSCP scenario, in which the test suite is *vertically* partial.

Experimental Methodology

The experimental methodology we have used for gathering data for the evaluation of the tools is depicted in Fig. 2 and described in the following. Given a combinatorial model, we aim to compare, for each tool, two different approaches: the one based on test generation from scratch and the one starting from seeds. Initially, by using pMEDICI, ACTS, and PICT, we generate the test suite with strength $t = 2$ starting from scratch (step 1 in Fig. 2), as normally done when generating combinatorial test sets. Then we apply the completion-based approach. In order to perform the generation by completion we generate an initial test suite TS_{KALI} with KALI [9] (step 2 in Fig. 2). This test suite is then used to obtain the test suite TS_{old} by randomly selecting a variable percentage of test seeds from TS_{KALI} (step 3 in Fig. 2). Thus, this step is repeated by keeping in TS_{old} from 0% to 100% (with step 10%) of the tests in TS_{KALI} to be used as seeds. As previously reported, each experiment is executed 5 times and a different random subset is taken at every repetition. Finally, we apply completion-based test generation using pMEDICI+, ACTS, and PICT (step 4 in Fig. 2) starting from the seeds in TS_{old}. We emphasize that TS_{old} represents the set of seeds we need to maintain, so in the traditional approach it is required to append TS_{old} to the test suites generated from scratch (step 5 in Fig. 2).

Research questions

In order to guide our experiments on the TSCP scenario, we have identified some research questions:

- **RQ1**: How do the tools performances depend on the percentage of removed tests in the TSCP scenario?
- **RQ2**: Which is the best-performing tool, in terms of time and size, in the TSCP scenario?

Experimental Results

In this section, we report the results of the experiments we conducted by following the experimental methodology reported in Section Experimental Methodology, and we

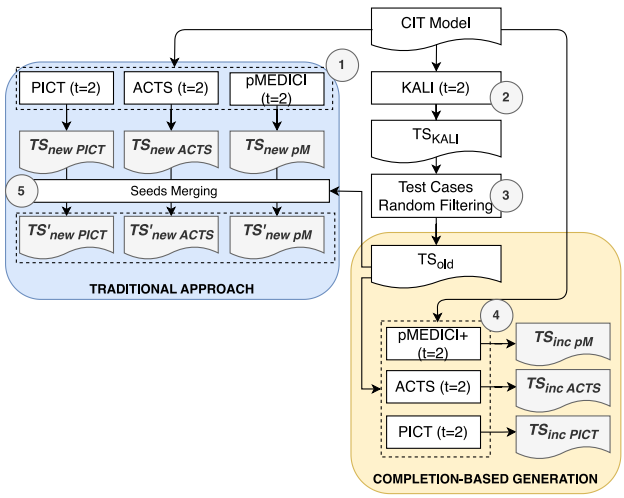


Fig. 2 TSCP: test suite completion

analyze them by means of the two research questions we have identified.

RQ1: Effects of the seed relative size (TSCP)

In this research question, we aim to investigate whether using seeds in the TSCP scenario is advantageous or not, both in terms of generation **time** and test suite **size**.

In Fig. 3a, b we report, respectively, the comparison between tools in terms of test suite generation time and size when different percentages of the test suites are removed in order to obtain the seeds. In this way, 0% means completely reusing the test suite TS_{KALI}, while 90% means keeping only few tests as seeds. In particular, in both plots, we report on the y-axis in a symmetric logarithmic scale the difference between the average time/size when a percentage x of the original test suite TS_{old} is removed and the average time/size when no seed is given. A positive difference means that using seeds increases time/size, while a negative difference means that using seed is advantageous. Note that in both plots the value 100% on the x-axis is the one used as a reference (i.e., the value obtained when seeds are not used) and, thus, it is not reported.

The number of **timeouts** does not change when a different seed size is used in the case of ACTS and PICT: ACTS never timed out, while PICT timed out 12 times. Instead, pMEDICI+ timed out 10 times when the incompleteness percentage is 0% (i.e., all seeds are kept) and only 5 times for all the other cases, meaning that significant amount of time is spent by the tool in seeds pre-processing.

In terms of test generation time (see Fig. 3a), using the complete test suite as seeds (i.e., removing 0% of the tests from TS_{KALI}) is advantageous only for PICT, while all other tools do not gain any advantage from this approach. When we increase the percentage of removed tests, PICT keeps taking advantage from using seeds, pMEDICI+ starts taking advantage when using a 10% incompleteness percentage, while ACTS is faster only when removing 20% and 30% (see Fig. 3a).

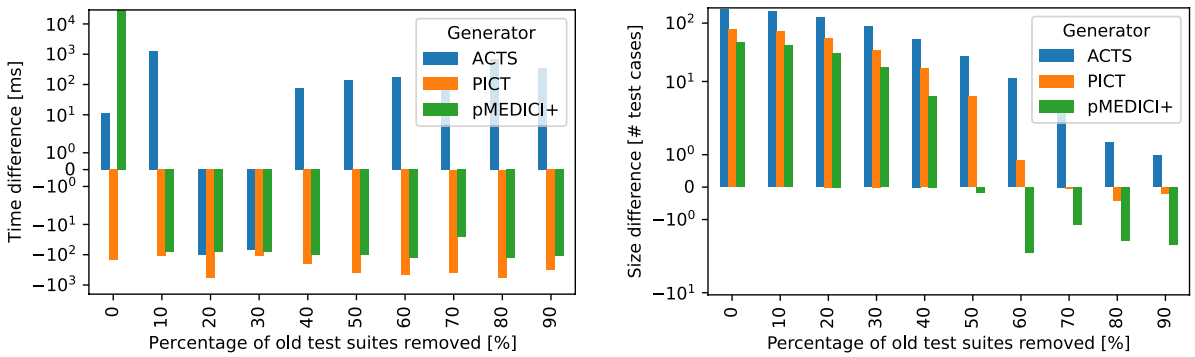
In terms of test suite size (see Fig. 3b), using the complete test suite as seeds (i.e., removing 0% of the tests from TS_{KALI}) does not represent an optimal solution for any tool, as all of them increase the size of the generated test suites. Only when increasing the incompleteness percentage up to 50% some tools start reducing the test suite size: PICT benefits from an incompleteness level higher than 70%, pMEDICI+ from 50%, while ACTS always produces larger test suites when seeds are used.

In conclusion, with the three analyzed tools, using seeds is not always effective, both in terms of size and generation time, especially for tools already producing optimized test suites in a short time even without seeds, as ACTS. However, some tool (such as PICT) which is normally quite slow may be made faster by starting from an already existing test suite. In general, different tools may take advantage by using different incompleteness percentages and the optimization of the test seeds strongly impacts on the size and time required to obtain the final test suite in an incremental way.

RQ2: Comparison among tools (TSCP)

In this research question, we are interested in comparing the three analyzed tools in the TSCP scenario.

Figure 4a, b report, respectively, the comparison among test suite generation time and size for all the tools (with the logarithmic scale on the y-axis) when different incompleteness percentages are used. Moreover, the dashed lines signal the minimum value (both for time and test suite size) for all tools. These results confirm ACTS being the fastest (when 20% of tests are removed from TS_{KALI}) generator as well as the one producing smallest test suites (when 80% of tests are removed from TS_{KALI}). pMEDICI+ is the second one both in terms of generation time (when 20% of tests are removed from TS_{KALI}) and test suite size (when 50% of tests are removed from TS_{KALI}). Finally, PICT is the slowest and the one producing, on average, larger test suites. However, for PICT the best generation time is measured when 90% of



(a) Test suite generation time comparison

(b) Test suite size comparison

Fig. 3 Tool comparison in the TSCP scenario using different size for seeds

tests are removed from TS_{KALI}, and the best test suite size when 80% of tests are removed from TS_{KALI}.

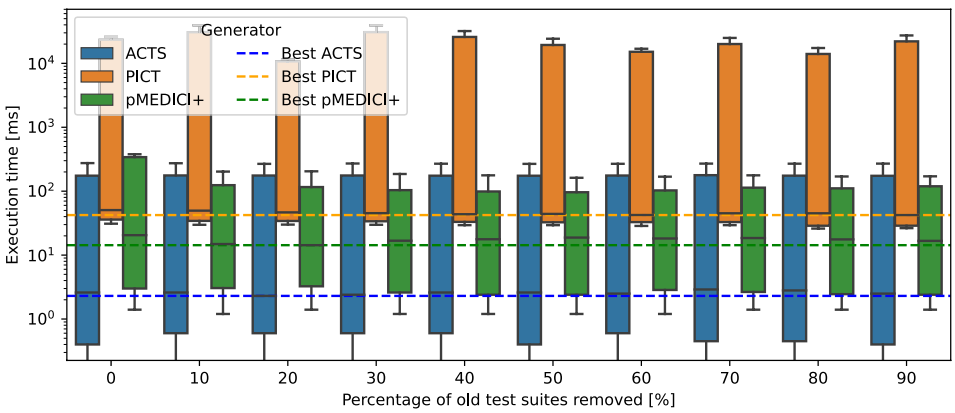
Given these considerations, we can now use for all the tools their best configurations and compare them. Figure 5a, b report, respectively, the comparison among test suite generation time and size for all the tools and model categories (with the logarithmic scale on the y-axis).

As reported in RQ1, PICT is the tool reporting the highest number of **timeouts**, followed by pMEDICI+. This may be due to the fact that some of the models, especially those appertaining to the MCAC category, were too complex and some of the tools timed out during the building of the structures used for generating test cases (e.g., of the MDDs in the case of pMEDICI+).

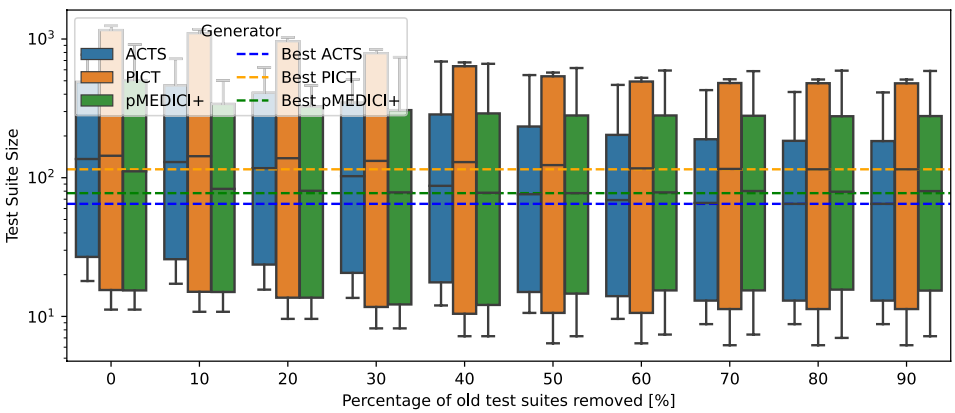
Let€™s first analyze the **generation time**. We here compare ACTS and pMEDICI+ when using 20%, and PICT using 90% as incompleteness percentage. ACTS is the tool leading in all categories, except for the BOOLC one, where pMEDICI+ is slightly faster. On constrained instances, PICT frequently times out (e.g., it does not complete any of the MCAC instances). On the other categories, moreover, PICT is the slowest tool, except in the MCA category, where it performs better than pMEDICI+ but worse than ACTS. These considerations are confirmed even when considering all the model categories as a whole, as shown by the Wilcoxon-Signed Rank Test reported in Table 3.

Moving to the **test suite size**, we here compare ACTS and PICT when using 80%, and pMEDICI+ using 50% as incompleteness percentage. ACTS and PICT performs comparably in all categories, except for the MCAC where no data is available for the latter as all executions timed out. Instead, pMEDICI+ produces, in all categories, slightly larger test suites than the other two tools. These considerations are confirmed even when considering all the model categories as a whole, as shown by the Wilcoxon-Signed Rank Test reported in Table 4. Note that the number of timeouts for PICT strongly influences the outcome of the statistical test for that tool.

In conclusion, among the three analyzed tools on the TSCP scenario, each of them configured in its optimal way, ACTS is the best-performing one both in terms of generation time and test suite size. PICT performs in a similar way in terms of size, but it is very much slower and reports several timeouts, while pMEDICI+ is faster than PICT but produces larger test suites.

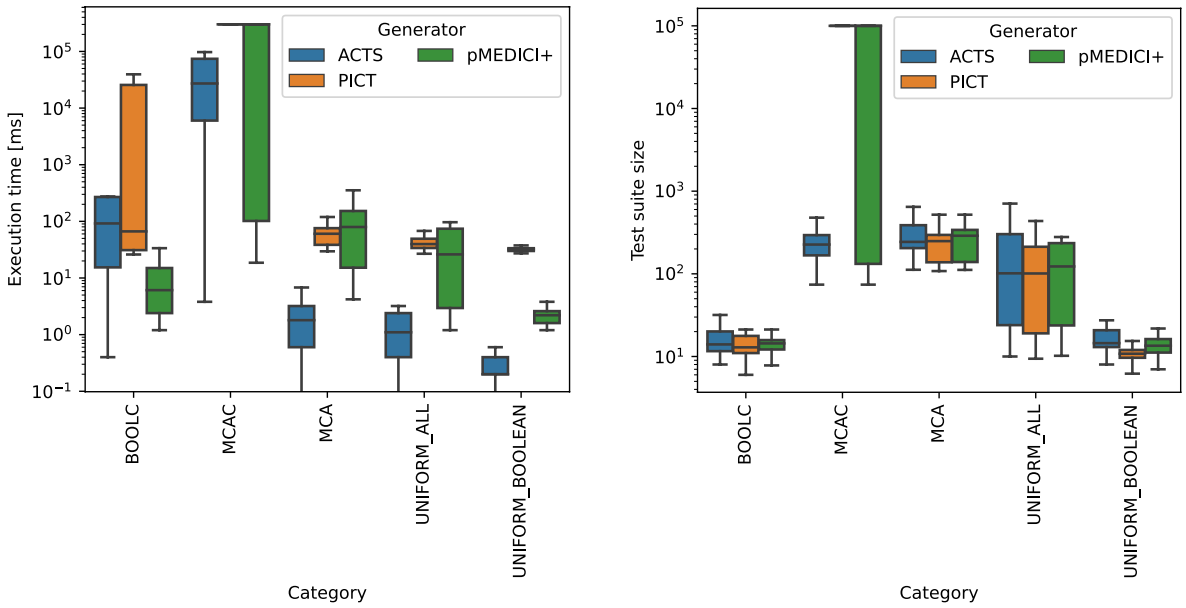


(a) Test suite generation time



(b) Test suite size

Fig. 4 Tool comparison in the TSCP scenario and different incompleteness percentages (only the completion-based version for each tool is reported)



(a) Test suite generation time comparison

(b) Test suite size comparison

Fig. 5 Tool comparison in the TSCP scenario (only the completion-based version for each tool is reported)

Table 3 Wilcoxon-Signed Rank Test on times for the TSCP scenario (only the completion-based version for each tool is reported)

		Wilcoxon-Signed Rank Test		
Tool	$\bar{t}$ [s]	w.r.t.	H_0 rejected?	p-value
ACTS	2.3	pMEDICI+	✓	3.73×10^{-17}
pMEDICI+	14.3	PICT	✓	1.53×10^{-21}
PICT	42.3			

Table 4 Wilcoxon-Signed Rank Test on sizes for the TSCP scenario (only the completion-based version for each tool is reported)

		Wilcoxon-Signed Rank Test		
Tool	$\bar{s}$	w.r.t.	H_0 rejected?	p-value
ACTS	64.8	pMEDICI+	×	0.17
		PICT	×	0.55
pMEDICI+	77.4	PICT	✓	5.26×10^{-9}
PICT	115.0			

SINC: Strength Increase

In this section, we report the experiments executed on the SINC scenario, in which the test suite is *vertically* partial, due to the request of increasing the strength of the test suite.

Experimental Methodology

For the SINC scenario (see Table 1), given a combinatorial model we aim to compare, for each tool, two different approaches generating test suites at a higher strength (in our experiments $t = 3$): the one based on test generation from scratch and the completion-based generation starting from a test suite with a lower strength (in our experiments $t = 2$). Initially, we use the traditional approach by generating the test suite achieving the desired final combinatorial coverage ($t = 3$) with pMEDICI, ACTS, and PICT (step 1 in Fig. 6). Then, we apply the completion-based approach based on test seeds. First, we generate a test suite TS_{old} for strength $t = 2$ using KALI (step 2 in Fig. 6). Finally, we execute pMEDICI+, ACTS, and PICT (step 3 in Fig. 6) in order to generate a test suite with strength $t = 3$ using the tests contained in TS_{old} as seeds.

Research questions In order to guide our experiments on the SINC scenario, we have identified some research questions:

- **RQ3**: Starting from a test suite for pairwise to obtain a test suite for 3-wise coverage: is it worthwhile, in terms of time and size?

- **RQ4**: Which is the best-performing tool, in terms of time and size, in the SINC scenario?

Experimental Results

In this section, we report the results of the experiments we conducted by following the experimental methodology reported in Section Experimental Methodology, and we analyze them by means of the two research questions we have identified. Figure 7a, b report, respectively, the comparison among test suite generation time and size for all the tools (with the logarithmic scale on the y-axis) with and without seeds, and we will use them both for RQ3 and RQ4.

RQ3: Effects of starting from a test suite with lower strength (SINC)

In this research question, we aim to compare each generator when it is asked to generate a 3-wise coverage with or without a test suite for pairwise used as seeds. In this way, we want to investigate whether using seeds in the SINC scenario is worthwhile or not.

As done in the previous RQs, we start comparing tools by analyzing the number of **timeouts**: all tools timed out the same number of times in their versions with and without seeds (7 times ACTS, 11 times PICT, and 9 times pMEDICI). This means that using seeds does not significantly impact, neither in a positive nor in a negative way, on the probability of a tool to complete the test generation when a higher strength of the test suite is needed.

Similar conclusions can be drawn by analyzing the data shown in Fig. 7a, regarding the generation **time** of the analyzed tools in each category and by comparing the results obtained by the basic version of each tool with those of the same tool when seeds are used. The plot shows that no relevant difference is visible for the three tools, in terms of average value, even if the use of seeds seems to lower the execution time for most model categories in the case of pMEDICI+. This data are confirmed by the statistical test, considering all the categories as a whole, reported in Table 5.

The only non relevant test is that between ACTS and ACTS w. SEEDS, demonstrating that for this tool no difference is visible when using seeds. Instead, both for PICT and pMEDICI, using a pairwise test suite as seed leads to enhanced performance in terms of generation time of 3-wise test suites. Nevertheless, for the former, the advantage of seeding is very limited (even if it is confirmed by the Wilcoxon-Signed Rank test€"see Table 5, where PICT w. SEEDS is statistically better performing than PICT), while for the latter the impact is much more evident, as confirmed by the statistical test between pMEDICI and pMEDICI+ in Table 5. ACTS, which is very fast in generating the 3-wise test suite, is not able to take advantage of the seeds, while pMEDICI and PICT which are normally slower in generating test suites, can avoid generating tests for covering the t-tuples already covered in the pairwise test suite, and this allows for saving time in test generation.

Finally, in terms of test suite **size** (see Fig. 7b) using seeds does not always imply obtaining smaller test suites, in all categories. Indeed, for ACTS using seeds does not influence the test suite size, for PICT the test suites are smaller when seeds are not used (in a statistically significant way€"see the Wilcoxon-Signed Rank Tests in Table 6), while with pMEDICI there is a slight improvement when seeding is used (i.e., with pMEDICI+), especially in the UNIFORM_BOOLEAN category, but it is not statistically significant, as denoted by the Wilcoxon-Signed Rank Tests in Table 6.

Thus, pMEDICI+ is the only tool which benefits from using seeds, in terms of test suite size but not in a statistically significant way. Instead, PICT tends to produce larger test suites if seeds are used in the SINC scenario.

In conclusion, for all the analyzed tools, using a pairwise test suite as seeds for generating a 3-wise test suite does not significantly impact the performance: ACTS€™s performance is not influenced by seeds (both in terms of time and size), PICT produces bigger test suites but in a shorter time when seeds are used, while pMEDICI+ benefits (both in terms of time and size) from seeds. For this reason, as

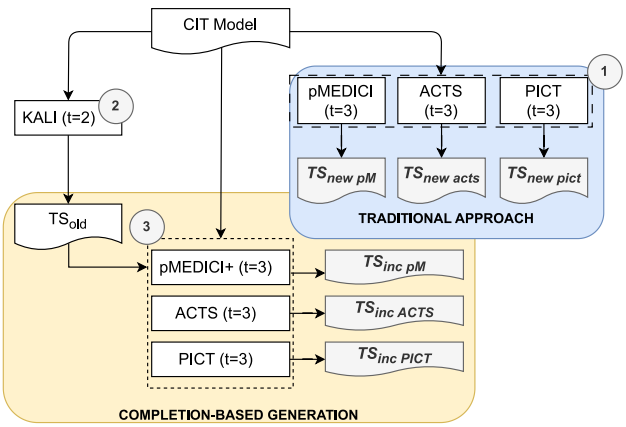


Fig. 6 SINC: test suite strength increase

Table 5 Wilcoxon-Signed Rank Test on times for the SINC scenario with the comparison of tools in their versions with or without seeds

		Wilcoxon-Signed Rank Test			
Tool	$\bar{t}$ [s]	w.r.t.	$\bar{t}$ [s]	H_0 rejected?	p-value
ACTS w. SEEDS	46.34	ACTS	46.34	×	0.20
PICT w. SEEDS	70.91	PICT	71.43	✓	4.22×10^{-3}
pMEDICI+	62.86	pMEDICI	64.96	✓	4.79×10^{-4}

performance are not significantly degraded for most tools, we favor the use of seeds in the SINC scenario if tests with lower strength can be reused.

RQ4: Comparison among tools (SINC)

In this research question, we aim to compare different generators in the SINC scenario when seeds are used.

The first comparison among tools can be made by considering the number of **timeouts**: ACTS w. SEEDS is the tool timing out the lowest number of times (only on 7 benchmarks), followed by pMEDICI+ (9 times) and by PICT w. SEEDS (11 times). Concerning the number of timeouts, ACTS_{SEEDS} is the tool with the best performance.

In terms of generation **time** (see Fig. 7a), ACTS w. SEEDS is the tool allowing the fastest test suite generation, while PICT w. SEEDS and pMEDICI+ are slower, and, in general, perform similarly, as confirmed by the statistical test results shown in Table 7.

Thus, in terms of generation time, ACTS w. SEEDS is the best fit.

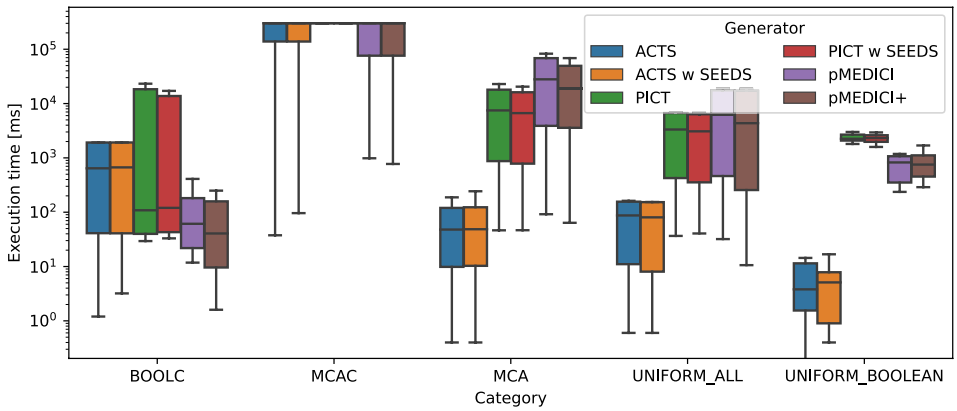
Finally, in terms of test suite **size** (see Fig. 7b), ACTS w. SEEDS is the tool generating the smallest test suites, followed by PICT w. SEEDS and pMEDICI+, as confirmed by the statistical test results shown in Table 8 which are all statistically relevant.

Thus, also in terms of test suite size, using ACTS w. SEEDS is the best option while dealing with the SINC scenario as it guarantees the generation of the smallest test suites.

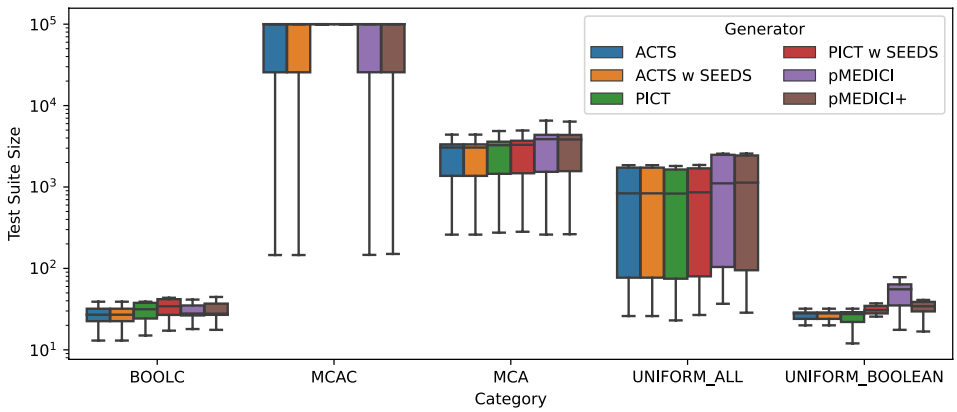
In conclusion, the best generator for the SINC scenario is ACTS w. SEEDS, both in terms of test suite size and

Table 6 Wilcoxon-Signed Rank Test on sizes for the SINC scenario with the comparison of tools in their versions with or without seeds

		Wilcoxon-Signed Rank Test			
Tool	$\bar{s}$	w.r.t.	$\bar{s}$	H_0 rejected?	p-value
ACTS w. SEEDS	864.47	ACTS	864.47	×	1.00
PICT w. SEEDS	934.72	PICT	913.87	✓	7.27×10^{-12}
pMEDICI+	1130.07	pMEDICI	1147.07	×	0.06



(a) Test suite generation time comparison



(b) Test suite size comparison

Fig. 7 Tool comparison in the SINC scenario

generation time. In terms of generation time PICT w. SEEDS and pMEDICI+ performs comparably, but the former generates more compact test suites than the latter when starting with a pairwise test suite for generating a 3-wise one.

TCCP: Test Cases Completion

In this section, we report the experiments executed on the TCCP scenario, in which the test suite is *horizontally* (or *mixedly*) partial, and some assignment in each test case is missing.

Experimental Methodology

For the TCCP scenario (see Table 1), given a combinatorial model we aim to compare, for each tool, two different approaches generating test suites which achieve the desired combinatorial coverage ($t = 2$, in our experiments): the one based on test generation from scratch and the completion-based generation starting from a test suite with partial test cases. Initially, by using pMEDICI, ACTS, and PICT, we generate the test suite starting from scratch (step 1 in Fig. 8), as normally done when generating combinatorial test sets. Then, we apply the completion-based approach. In order to perform the completion-based generation we generate an initial test suite TS_{KALI} with KALI [9] (step 2 in Fig. 8). This test suite is then used to obtain the test suite $TS_{partial\ old}$ by randomly adding don€™t cares (representing not assigned parameter values) or by removing some assignment (depending on the tool€"see Section Tools Supporting Completion) in a variable percentage of parameters in each test seed from TS_{KALI} (step 3 in Fig. 8). Thus, the process is repeated removing from 0% to 100% (with step 10%) of the parameter assignments in each test. As previously reported, each experiment is executed 5 times and a different random subset of assignments is taken at every repetition. Finally, we apply completion-based test generation using pMEDICI+, ACTS and PICT (step 4 in Fig. 8) starting from $TS_{partial\ old}$. We emphasize that $TS_{partial\ old}$ contains the seeds the user wants to maintain, so in the traditional approach we merge it with the test suites obtained by the three tools (step 5 in in Fig. 8). During this process, for each seed, we look for a test covering it. If it exists, we skip the seed (as it has already been covered); otherwise, we add it to TS_{new} and we leave it not complete.

Table 7 Wilcoxon-Signed Rank Test on times for the SINC scenario (only the completion-based version for each tool is reported)

Tool	$\bar{t}$ [s]	Wilcoxon-Signed Rank Test w.r.t.	H_0 rejected?	p-value
ACTS	46.07	pMEDICI+	✓	2.64×10^{-3}
pMEDICI+	62.89	PICT	×	0.80
PICT	70.91			

Table 8 Wilcoxon-Signed Rank Test on sizes for the SINC scenario (only the completion-based version for each tool is reported)

Tool	$\bar{s}$	Wilcoxon-Signed Rank Test w.r.t.	H_0 rejected?	p-value
ACTS	864.47	PICT	✓	2.40×10^{-10}
PICT	934.72	pMEDICI+	✓	6.57×10^{-9}
pMEDICI+	1130.07			

Research questions

In order to guide our experiments, we have identified some research questions:

- **RQ5**: How do the tools performances depend on the percentage of removed assignments from the seed tests in the TCCP scenario?
- **RQ6**: Which is the best-performing tool, in terms of time and size, in the TCCP scenario?

Experimental Results

In this section, we report the results of the experiments we conducted by following the experimental methodology reported in Section Experimental Methodology, and we analyze them by means of the two research questions we have identified.

RQ5: Effects of the seed relative size (TCCP) In this research question, we aim to investigate if using seeds in the TCCP scenario is always advantageous, both in terms of test suite **size** and generation **time**, or if there is an incompleteness percentage of assignments removed from the test

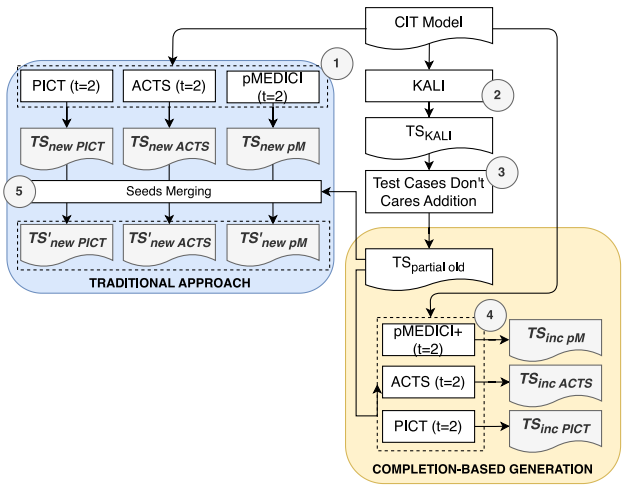


Fig. 8 TCCP: test suite completion

cases over which it is useless or counterproductive to use them as seeds.

In Fig. 9a, b we report, respectively, the comparison between tools in terms of test suite generation time and size when different percentages of the test suites are removed in order to obtain the seeds. As for RQ1, in both plots, we report on the y-axis the difference (the lower the better) between the average size/time when a percentage x of the parameter assignments in TS_{old} is removed and the average size/time when no seed is given.

Interestingly, our experiments show that using seeds composed of partial test cases (i.e., working with the TCCP scenario) is, in general, not very useful for reducing generation time or test suite size.

For what concerns the number of **timeouts**, ACTS did not time out in any of the executed experiments. pMEDICI+ timed out 5 times in every configuration while PICT timed out on 8 models in all configurations. In terms of **time** (see Fig. 9a), using the complete test suite as seeds is advantageous for PICT and pMEDICI+, while ACTS is not able to take advantage from seeds. When increasing the incompleteness level, PICT keeps being the tool gaining the most advantage, except in the case in which the incompeteness percentage is 70% or 80%. pMEDICI+, on the contrary, is slowed by using a non complete test suite as seeds. Finally, for ACTS the delta in generation time shown in Fig. 9a is quite swinging, meaning that handling partial test cases is sometimes costly for the tool and, depending on which assignments are removed, the performance can either improve or get worse.

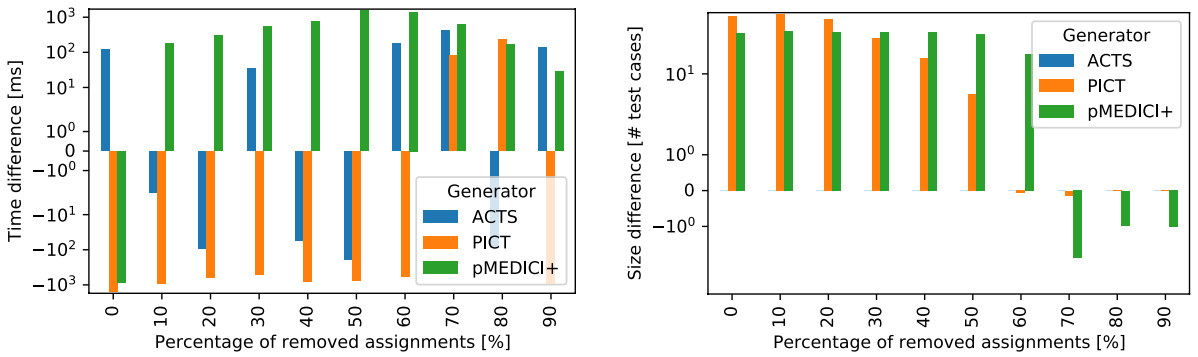
In terms of test suite **size** (see Fig. 9b), using the complete test suite as seeds (i.e., with an incompleteness level of 0%) does not imply any advantage for any of the analyzed tool. The only tool which does not worsen its performance is ACTS, while pMEDICI+ and PICT produce significantly larger test suites. When increasing the incompleteness percentage, ACTS does not either increase or decrease the test suite size. PICT tends to reduce the disadvantage of seeds with the increase of the incompleteness percentage and starts gaining a very small advantage over 60%. pMEDICI+ behaves very similar to PICT and over 70% it significantly benefits from the use of seeds.

The fact that using incomplete seeds does not reduce the size of the final test suite is expected. On the contrary, reusing existing seeds may require more tests, especially because $TS_{partial\ old}$ is generated by KALI which, in general, tends to generate less compact test suites (as confirmed by the results obtained during the past editions of the CT Competition [16].[6]

In conclusion, in the TCCP scenario, we don€™t see, in general, any reasonable advantage in using seeds, except when their use is required by the application domain. If the test suite size is analyzed, the only tool which significantly benefits from the use of seeds is pMEDICI+, while, in terms of test suite generation time, pre-processing test seeds seems to heavily impact on the performance of the tool. For what concerns ACTS and PICT, using partial test cases as seeds does not improve the test suite size and, for the former, neither does the generation time, while for the latter test suite generation is faster when seeds are used in the TCCP scenario.

RQ6: Comparison among tools (TCCP)

In this research question, we aim to investigate the performances of the three analyzed tools in the TCCP scenario. Fig. 10a, b report, respectively, the comparison among test suite generation time and size for all the tools (with the logarithmic scale on the y-axis) when different incompleteness percentages are used. Moreover, the dashed lines signal the

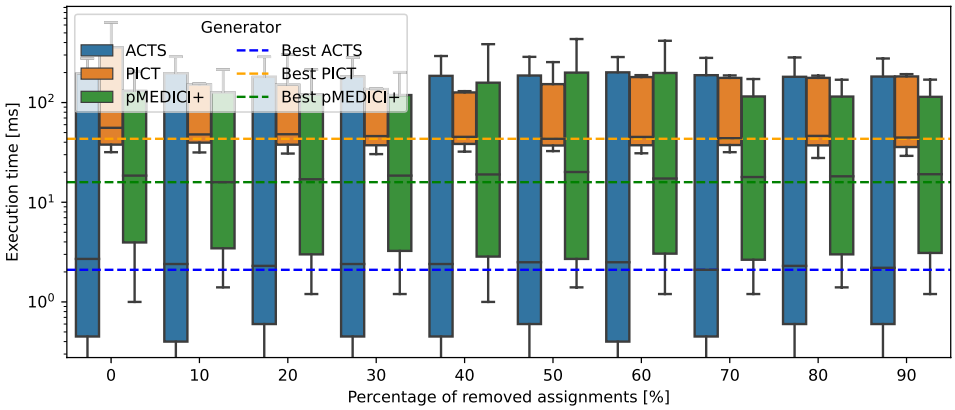


(a) Test suite generation time comparison

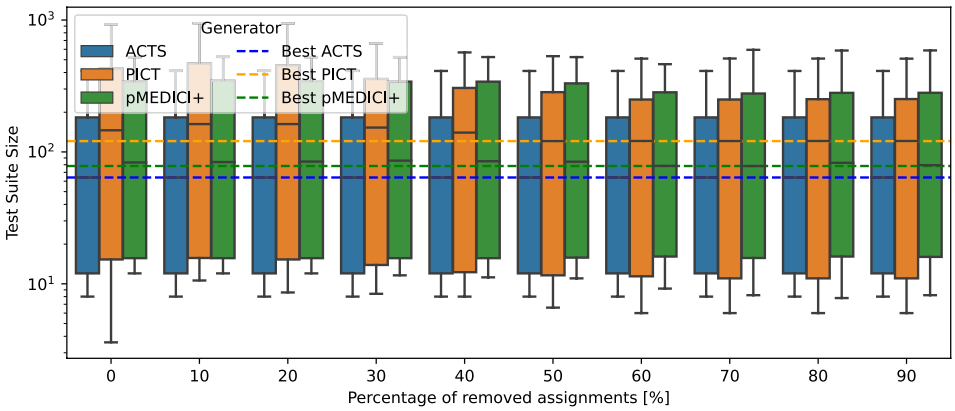
(b) Test suite size comparison

Fig. 9 Tool comparison in the TCCP scenario using different size for seeds

[6] https://fmselab.github.io/ct-competition/results/2023/Competition2023.html.



(a) Test suite generation time



(b) Test suite size

Fig. 10 Test suite generation time comparison

minimum value (both for time and test suite size) for all tools. These results confirm ACTS being the fastest (when 70% of assignments in tests are removed from TS_{KALI}) generator as well as the one producing smallest test suites (for all incompleteness percentages). pMEDICI+ is the second one both in terms of generation time (when 10% of assignments in tests are removed from TS_{KALI}) and test suite size (when 70% of assignments in tests are removed from TS_{KALI}). Finally, PICT is the slowest and the one producing, on average, larger test suites. However, its best generation time and test suite size are measured when 50% of assignments in tests are removed from TS_{KALI}.

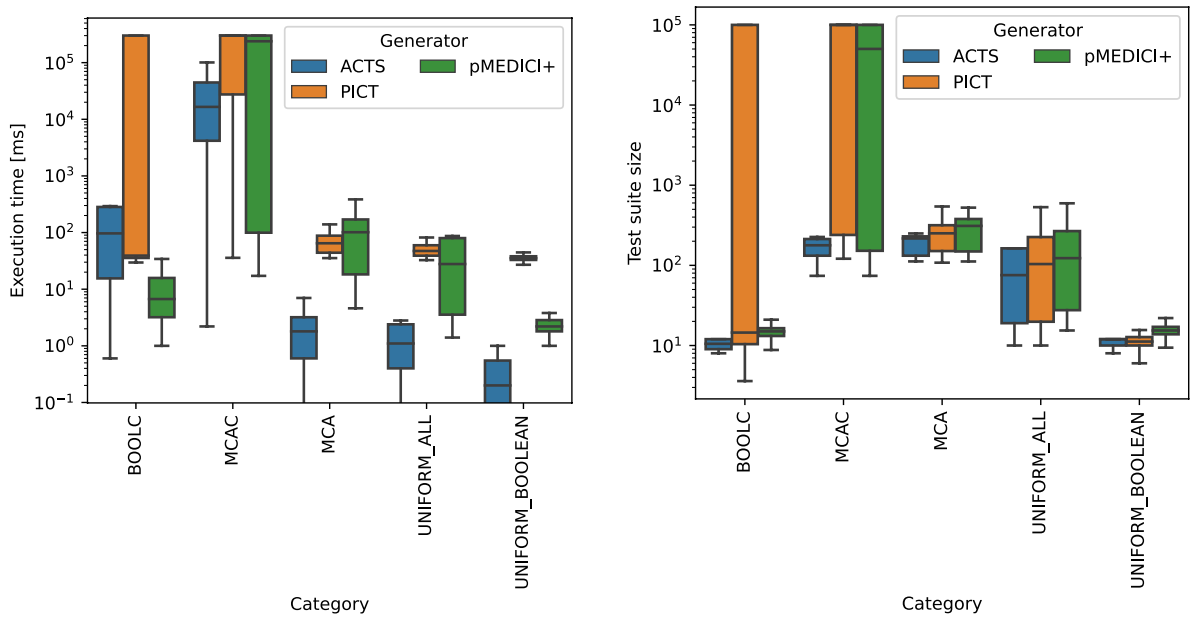
In the previous RQ, we have shown how using seeds is not a real advantage, in general, in terms of test suite size or generation time. However, just to compare the three tools, we choose the values reported in the previous discussion and we configure each tool to perform completion-based generation in its best configuration (i.e., for time, ACTS with incompleteness percentage 70%, PICT with 50% and pMEDICI+ with 10%, and for size, ACTS with incompleteness percentage 0%, PICT with 50% and pMEDICI+ with 70%).

Figure 11a, b report, respectively, the comparison among test suite generation time and size for all the tools in each category (with the logarithmic scale on the y-axis).

As discussed in the previous RQ, our experiments have highlighted that the number of **timeouts** does not change when a different seed size is used. Regarding the **time**, ACTS is the fastest tool in each category, except for the BOOLC one. pMEDICI+, instead, leads the BOOLC category (as already discussed for the TSCP scenario) but follows ACTS in all the others. Finally, PICT is the slowest tool in each category, either with and without constraints.

These results are confirmed by the Wilcoxon-Signed Rank test reported in Table 9, where ACTS is confirmed to be the best performing tool, followed by pMEDICI+ and PICT when completion-based generation is used in the TCCP scenario.

Considering the test suite **size**, ACTS is the tool producing the smallest test suites with the lowest variability in each category. PICT performed better than pMEDICI+, on average, on unconstrained instances and comparably in all the others, but with a consistent amount of timeouts. These considerations are confirmed by the results we report in Table 10.



(a) Test suite generation time comparison

(b) Test suite size comparison

Fig. 11 Test suite size comparison

Table 9 Wilcoxon-Signed Rank Test on times for the TCCP scenario (only the completion-based version for each tool is reported)

		Wilcoxon-Signed Rank Test		
Tool	$\bar{t}$ [s]	w.r.t.	H_0 rejected?	p-value
ACTS	2.1	pMEDICI+	✓	4.85×10^{-19}
pMEDICI+	15.9	PICT	✓	1.26×10^{-9}
PICT	43.3			

Table 10 Wilcoxon-Signed Rank Test on sizes for the TCCP scenario (only the completion-based version for each tool is reported)

		Wilcoxon-Signed Rank Test		
Tool	$\bar{s}$	w.r.t.	H_0 rejected?	p-value
ACTS	64.0	pMEDICI+	✓	2.59×10^{-77}
pMEDICI+	78.2	PICT	✓	1.55×10^{-8}
PICT	120.8			

In conclusion, among the three analyzed tools on the TCCP scenario, ACTS is the best-performing one both in terms of generation time and test suite size when completion-based generation is used and the tools are configured in their best setting. Instead, pMEDICI+ performs better in terms of generation time than PICT, but it generates larger test suites even if with fewer timeouts.

Threats to Validity

In this section, we discuss potential threats to validity [17] of our work and the actions we have taken to mitigate them.

Internal validity refers to the fact that the different outcomes obtained with the analyzed techniques and tools are actually caused by the different approaches themselves and by the way the experiments were carried out, and not by methodological errors. To mitigate this risk, we have carefully checked the code of the experiments to see if there could be other factors that have caused the outcome, such as errors in the tools or in the experimental code we wrote. Moreover, we have validated the obtained test suites in order to check whether they covered all the possible t-way interactions of interest, and no invalid test case was produced by the generators under analysis. The PC that has been used for performing the experiments has been carefully checked in order to verify whether other processes were executed in the meantime. However, we cannot completely exclude the influence of the operating system on the obtained results.

A possible threat to the *construct validity* comes from the assumption that the measures we considered for comparing tools and their configurations (i.e., time and size) are suitable to correctly compare different tools. We rely on the literature for this, where similar measures are often

used like in [5]. However, we recognize that comparing test suites with these measures does not take into account other properties, such as the differences between seeds and the obtained test suites [3] or their specificity [4] which can be of interest as well.

External validity is concerned with whether we can generalize the results outside the scope of the presented study. Under this lens, one threat to the external validity refers to scenarios we have used in the experiments. We recognize that other scenarios could be considered as well. For example, in our experiments, we always start with a valid test suite TS_{old} which is made partial by removing some test cases or assignments from the test suite generated by KALI. However, the results we have obtained are influenced by the validity of TS_{old}: if a partially invalid test suite was considered, the conclusions may have been different. For example, with ACTS and PICT, we did not find any evidence in the literature about their ability to repair invalid test cases, while pMEDICI+ is able to exclude non-valid assignments and to still consider the valid part as seeds. As future work, we may investigate further scenarios including partially invalid test cases or test suites. Moreover, in this work, we have used KALI to generate the test seeds for guaranteeing more fairness, as this tool is not included in those we have used for our evaluation. However, in future work, we may evaluate whether using a different tool than KALI for generating the test suites which seeds are extracted from could change the conclusions we have drawn with our experiments. Indeed, KALI is, in general, a tool producing poorly optimized test suites and this may influence the outcome of the experiments, as explained in Sect. 4. However, we do not consider this lack of optimization to be a limitation of our approach because, as previously introduced in Section Partial Tests, seeds are typically created manually by testers rather than being generated automatically, so a lower level of optimization is somehow expected. From preliminary experiments, however, we have seen that similar results are obtained even with other tools producing seeds.

Related Work

Incremental generation of test suites has been tackled in other papers by applying it in different contexts as well. Our work aspires to recap in a single framework different approaches and scenarios in which completion-based test generation is or is not advisable.

In the TCCP and TSCP scenarios, we consider seeds to be partial tests or test suites. This idea was originally proposed in [12]: The tester can also guarantee inclusion of their favorite test cases by specifying them as seed tests or partial seed tests for a relation. The seed tests are included in the generated test set without modification. The partial seed tests are seed test cases that have fields that have not been assigned values. This definition is compliant with what we report in Section 2. The problem of using seeds is tackled also by [10], in which the authors describe an algorithm for prioritized interaction testing aiming to achieve pairwise coverage by taking advantage of seeds.

As reported in the explanation of the TSCP scenario, partial tests can be used when the system under test evolves and tests need to be checked and, possibly, adapted in order to be still applicable to the new version of the system. In this scenario, in [18], the authors do not explicitly mention seeds, but propose to solve the problem of changing requirements in combinatorial models by combining three building blocks, allowing to minimally modify existing and not valid combinatorial tests, enhance them, or choose from them selected test cases for obtaining a new test suite, composed of only valid tests. This work also presented the FOCUS tool embedding this process. Unfortunately, we have not been able to find this tool, otherwise, we could have compared it with ACTS, PICT, and pMEDICI+ in our experiments.

Using a previously computed test suite for achieving higher strength coverage, such as in the SINC scenario has been proposed in [14], where the authors present an approach that incrementally builds covering array schedules: it begins at low strength, and then iteratively increases strength as resources allow using previous tests as seeds. However, the authors do not commit to any specific algorithm for generating test cases nor tools. Moreover, since a limit in resources is set, in that approach it is not guaranteed that the desired strength t is reached, and the t-wise coverage fully achieved. Another tool, called INWARD, for test generation from seeds is presented in [11]. It exploits the algebraic properties of Covering Arrays (CA) and allows the fast generation of a CA of strength t given an already CA of strength $t-1$, as we do in the SINC scenario we described in Section Scenarios. The algorithm is very fast, but it produces very large test suites, and it is not able to consider constraints. For this reason, we have not included INWARD in our comparison.

Conclusions

In this paper, we have addressed the important problem of completing partial test suites using combinatorial test generators supporting test generation from seeds. Through extensive experiments and analysis, we have compared three generators, namely ACTS, PICT, and pMEDICI+, in three distinct scenarios having different directions and percentages of removed tests or assignments from seeds: TSCP, SINC, and TCCP.

The results of our experiments revealed that ACTS with seeds consistently outperformed the other tools in terms of both test suite size and generation time. However, it is worth

noting that the use of seeds may not always be advantageous, particularly in the TCCP and TSCP scenarios where completing test cases or test suites may not always be cost-effective, due to the overhead imposed by seed treatment and their possible non optimization. These findings provide valuable insights into the practical application of combinatorial interaction testing and offer guidance on when and how to effectively utilize partial tests in testing processes.

In this work, we have focused on using only valid (partial) tests as seeds. However, as future work, in order to have a deeper understanding of the differences between these tools, we are planning to include invalid tests to be repaired in seeds and to see whether the conclusions we have drawn in this paper could change or not.

Acknowledgements This work has been partially supported by the National Plan for NRRP Complementary Investments (PNC)-ANTHEM (AdvaNced Technologies for Human-centrEd Medicine)-CUP: B53C22006700001-Spoke 1-Pilot 1.4.

Author Contributions AB (corresponding author): Conceptualization, Methodology, Software, Validation, Writing-Original Draft, Writing-Review & Editing; AG: Conceptualization, Methodology, Writing-Original Draft, Writing-Review & Editing.

Funding Open access funding provided by Università degli studi di Bergamo within the CRUI-CARE Agreement. Not applicable.

Data Availability The data sets used in our experiments and the scripts used for the evaluation of the tools can be found in our repository online at https://github.com/fmselab/ct-tools/tree/main/IncrementalCitGeneration.

Declarations

Conflict of Interest Andrea Bombarda and Angelo Gargantini are co-authors of pMEDICI and pMEDICI+.

Ethical Approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Jenny web page. https://burtleburtle.net/bob/math/jenny.html. Accessed 11 Apr 2025.
2. PICT GitHub page. https://github.com/microsoft/pict. Accessed 11 Apr 2025.
3. Bombarda A, Bonfanti S, Gargantini A. On the reuse of existing configurations for testing evolving feature models. In: Proceedings of the 27th ACM international systems and software product line conference (Volume B). ACM; 2023.
4. Bombarda A, Bonfanti S, Gargantini A. Testing the evolution of feature models with specific combinatorial tests. In: 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2024.
5. Bombarda A, Crippa E, Gargantini A. An environment for benchmarking combinatorial test suite generators. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2021.
6. Bombarda A, Gargantini A. 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, 2022.
7. Bombarda A, Gargantini A. Incremental generation of combinatorial test suites starting from existing seed tests. In: 2023 IEEE international conference on software testing, verification and validation workshops (ICSTW). 2023. p. 197–205.
8. Bombarda A, Gargantini A. Design, implementation, and validation of a benchmark generator for combinatorial interaction testing tools. J Syst Softw. 2024;209: 111920.
9. Bombarda A, Gargantini A, Calvagna A. Multi-thread combinatorial test generation with smt solvers. In: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC '23, page 1698-1705, New York, NY, USA, 2023. Association for Computing Machinery.
10. Bryce RC, Colbourn CJ. Prioritized interaction testing for pairwise coverage with seeding and constraints. Inf Softw Technol. 2006;48(10):960–70.
11. Calvagna A, Tramontana E. Incrementally applicable t-wise combinatorial test suites for high-strength interaction testing. In: 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops. IEEE, 2013.
12. Cohen D, Dalal S, Fredman M, Patton G. The AETG system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng. 1997;23(7):437€"44.
13. Forbes M, Lawrence J, Lei Y, Kacker RN, Kuhn DR. Refining the in-parameter-order strategy for constructing covering arrays. J Res Nat Inst Stand Technol. 2008;113(5):287.
14. Fouché S, Cohen MB, Porter A. Incremental covering array failure characterization in large configuration spaces. In: Proceedings of the eighteenth international symposium on software testing and analysis. ACM; 2009.
15. Lei Y, Tai K. In-parameter-order: a test generation strategy for pairwise testing. In: Proceedings third IEEE international high-assurance systems engineering symposium (Cat. No.98EX231). 1998. p. 254–61.
16. Leithner M, Bombarda A, Wagner M, Gargantini A, Simos DE. State of the CArt: evaluating covering array generators at scale. Int J Softw Tools Technol Transf. 2024;26:301–26.
17. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng. 2008;14(2):131–64.
18. Tzoref-Brill R, Maoz S. 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, 2018.
19. Wagner M, Kleine K, Simos D, Kuhn R, Kacker R. Cagen: a fast combinatorial test generation tool with support for constraints and higher-index. In: International Workshop on Combinatorial Testing (IWCT 2020), 3 2020.

20. Woolson RF. Wilcoxon signed-rank test. In: Encyclopedia of biostatistics. Wiley; 2005. https://doi.org/10.1002/0470011815.b2a15177.
21. Yu L, Lei Y, Kacker RN, Kuhn DR. ACTS: a combinatorial test generation tool. In: 2013 IEEE Sixth International Conference on software testing, verification and validation. IEEE, 2013.