Incremental generation of combinatorial test suites starting from existing seed tests

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

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

***Abstract*—Many researchers have been focusing on building combinatorial test generators having the best possible performances, in terms of smaller test suites and shorter generation times. The majority of tools generates test suites from scratch. This means that when the test suite must be regenerated, the old tests are discarded and a new test suite is built. However, there are many cases in which old test cases, possibly written by hand, need to be (or could be) included in the final test suite, and the test suite completed with new tests in order to reach the desired level of combinatorial coverage. These existing tests that are reused are generally called *seed tests*. Seed tests could be important for testing domain-specific critical parts of the system, or they could represent old test suites that must be enriched to reach the desired (possibly higher) strength of coverage. In this paper, we propose a new architecture for incremental test generation that starts from existing test seeds. This new architecture is supported by the pMEDICI+ tool which extends our previous effort done for pMEDICI. We evaluate the proposed approach on the benchmarks given in the context of the second edition of the CT-Competition and w.r.t. two application scenarios. For each scenario, we automatically generate seed tests and then we apply pMEDICI+ to obtain the desired test suite. The experiments highlight that using incremental test generation can contribute significantly in the reduction of test generation time and, in many cases, in the reduction of the test suite size.**

***Index Terms*—Incremental test generation, Combinatorial testing, Test suite completion, Test strength increasing, Seed tests**

I. Introduction

Combinatorial Interaction Testing (CIT) has been a topic of intense research for many years and has proven very useful in the testing of complex systems, especially those with multiple input parameters. Combinatorial test suites are generated using test generators which exploit different algorithms. Searching for ever more powerful algorithms capable of generating fewer tests for complex models has led the community to introduce new combinatorial test generator tools every year. However, the majority of these tools suffer from the same limitation: the test generation builds a test suite from scratch and every time it is repeated, a new one is generated and old tests are lost. In practice, considering already existing tests may be needed for many reasons, such as: 1) some tests were generated by a different test generator that has not covered all the testing requirements; 2) some test cases (even partial) written by hand are available and testers want to complete them; 3) a new test suite with higher strength is needed; 4) the combinatorial model used for generating tests has changed and some tests must be discarded. In these cases, generating a new test suite from scratch is not the optimal solution. Reusing some of the old tests or their parts is desirable, since the old test suite may include some tests that are important in order to test specific and critical conditions of the system under test. Moreover, completing an existing test suite may require less time than building a new one from scratch.

For all these reasons, incremental generation of combinatorial tests from existing tests is important. There are already some tools supporting this (for example, ACTS [16], jenny [1], and PICT [2]). In this paper, we show how to extend an architecture that allows testers to generate combinatorial test suites by exploiting the boost in performance offered by multithreading and by taking into account test cases previously available. We present pMEDICI+, an extension of the tool pMEDICI [6], which is capable of producing combinatorial tests by starting from a non-empty test suite. It is able to deal with both tests that are still valid for the system under test and tests that are partially invalid. Old tests are given as input to pMEDICI+ in the `csv` format. They are automatically filtered and polished in order to keep only those that are still valid or only their parts that are still valid. Then, new tests are incrementally added until the desired coverage is reached.

We have identified two main scenarios in which such incremental generation can be useful: when the tester wants to complete an existing incomplete test suite (TSCP scenario) and when the tester wants to increase the strength of an existing test suite (SINC scenario). For these two scenarios, we have devised a set of research questions and designed experiments able to give us the answers to those questions.

The experiments confirm that using pMEDICI+ and starting from an already generated test set, in general, allows practitioners to generate test suites in a shorter time, w.r.t. generating them from scratch, and, in some cases, to reduce the total number of test cases.

The remainder of this work is structured as follows. Sect. II introduce the background concepts on using multivalued decision diagrams for generating combinatorial test suites and on the concept of seed tests. Sect. III presents the pMEDICI+ tool, the incremental generation of test suites approach implemented by the proposed tool, and the main scenarios in which incremental generation may be useful. In Sect. IV we present the experimental methodology we have used in the experiments we have carried out in order to evaluate our approach on a set of combinatorial models, and in Sect. V we present the obtained results. Sect. VI discusses the possible threats to the validity of our findings. Finally, Sect. VII introduces the related work on the incremental generation of combinatorial test suites and Sect. VIII concludes the paper.

II. Background

In this section, we report the background on combinatorial test generation using Multivalued Decision Diagrams (MDDs) with pMEDICI and we recall the concept of seed tests, used by the incremental test generation approach.

A. Combinatorial problems, MDDs, and pMEDICI

In [6], we presented pMEDICI and proposed to use MDDs for dealing with combinatorial test generation, i.e., to use them for representing both parameters and constraints, and for extracting test cases. In particular, with MDDs one can represent boolean functions. While dealing with combinatorial testing, it is possible to encode in an MDD the Boolean functions computing the validity of assignments to each parameter in the combinatorial model. In this way, when trying to insert a new assignment for a parameter in a test case (denoted by $\langle par, val \rangle$), if it is accepted by the MDD it means that the assignment is *compatible* with that test case and can be added. Otherwise, it may be compatible with a different test case or definitely incompatible with the combinatorial model, due to the constraints.

This approach has been used in pMEDICI and allows creating a complete test suite, which contains only *valid tests*, i.e., those that do not clash with the constraints of the combinatorial problem. Each test case is handled through a *test context*, defined as follows:

Definition 1 (Test context)**.** *We call $TC = \langle A, M_{TS} \rangle$ a* test context *for the combinatorial problem P, where A is a list of assignments to some parameters p_i to one of their possible values $v_{i,j}$ and M_{TS} is the MDD representing a combinatorial problem P and the assignments committed to the context so far.*

The operations on the context can be: $isCompAssign$ to check if an assignment is compatible, $addAssignment$ to add an assignment to the context, and $isImplied$ if the test context contains already or implies a given assignment.

In this work, we extend the concept of test context, allowing A to be not empty when the test generation process starts.

B. Seed tests

The incremental approach proposed in this paper is based on *seed tests* or simply *seeds*. Seeds can be user-defined tests that are prescribed, for instance by a requirement specification [8]. They represent a first set of tests that must be included at the start in the test suite; for this reason, they constitute the *old* test suite and they are guaranteed to be included in the final test suite [10]. Then, a new test suite is created by taking the old test suite (the seeds) and adding the necessary tests covering the required t-way interactions not already covered by the seeds.

Note that, in this work, we consider the seed tests to be only the valid tests. However, the architecture and tool we propose in Sect. III can be used also with seeds that are no longer valid for the system to be tested and need to be polished or filtered.

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 $par \in citModel.getParamList()$ do
4:          $val \leftarrow seed.getVal(par)$
5:          if $val \neq NULL$ then
                ▷ Check if the new assignment is compatible
                ▷ with model constraints
6:              if $tc.isCompAssign(\langle par, val \rangle)$ then
7:                  $tc.addAssignment(\langle par, val \rangle)$
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
```

III. pMEDICI+: Incremental test generation

In this paper, we present pMEDICI+, a tool that allows the incremental generation of combinatorial test suites. It exploits the basic structure of pMEDICI [6] and extends it by adding an optional preprocessing stage in which the seeds are handled and initialized to be used as a starting point for generating the new test suite. The dataflow of pMEDICI+ is shown in Fig. 1 and described in the following.

A. Seeds preprocessing

The preprocessing stage is composed of the 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 Sect. II-A) by the Test Early Filler procedure reported in Alg. 1. The preprocessing phase cyclically takes each test case from the $seedTests$ list (line 1), and creates a new test context tc, which is initially empty with its own internal MDD (line 2); it is initialized with the parameters and the constraints

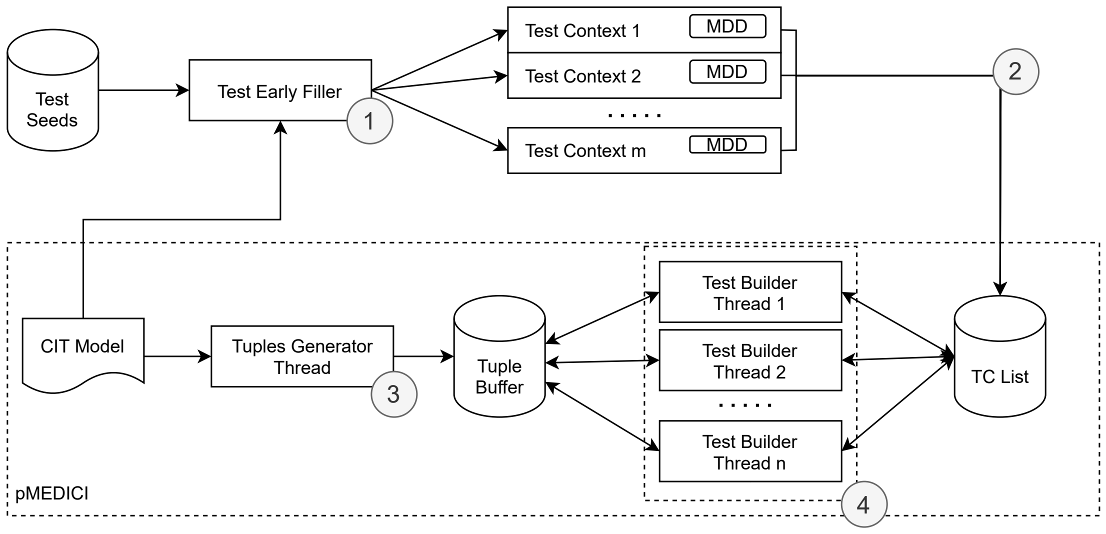


Fig. 1: Data flow in the pMEDICI+ tool

of the combinatorial model and continuously updated when each assignment of the analyzed test case is checked for compatibility (line 6) and added (line 7).

This operation is necessary and performed not at the test-case level but at the single-assignment level since the seed test may not be completely compatible with the combinatorial model and, thus, only valid assignments must be kept. Then, all test contexts created are stored in a $tcList$ (line 12), which is the one used by the pMEDICI subcomponent. Note that this preprocessing phase is performed in single-thread mode since from our preliminary experiments we have not observed significant improvement by parallelizing the preprocessing stage.

B. 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 Alg. 2. Each builder inserts, if possible, the selected tuple in one of the test contexts stored in the $tcList$ ($findImplies$ 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.

C. Tool limitations

Being based on pMEDICI and MDDs, pMEDICI+ shares the same limitations when dealing with combinatorial models. In fact, it is not able to deal with models containing constraints with arithmetical operators (such as $+$, $-$, $\times$ and $/$) or comparisons between parameters (such as $p_1 = p_2$ or $p_1 \neq p_2$). Note that the presented approach may be applied in other tools based on the same architecture but different solvers, e.g., KALI [7], allowing testers to overcome these limitations.

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 preprocessing 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
```

TABLE I: Use case scenarios for incremental test generation

ID	Description	Goal	Traditional approach
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	A new t-wise test suite is generated and the old test cases, or those that are still applicable, are added
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	A new $(t+1)$-wise test suite is generated

D. Typical application scenarios of the approach

We have identified two typical scenarios in which the incremental approach is useful; they are reported in Tab. I and better described in the following. They are suitable for two different types of experiments as we will see in Sect. IV.

The TSCP scenario typically occurs when an incomplete test suite (let's call it TS_{old}) is available and the tester wants to build a test suite that reaches the desired coverage and include the seeds. This case can occur for several reasons: some tests are written by hand by the tester in order to test critical conditions of the SUT, or the model has been modified and some old tests may have become invalid and need to be discarded. Invalid tests due to model evolution are studied in [14], while the use of seeds to represent tests requested by the user is presented in [8], [10]. In this scenario, traditionally, a completely new test suite TS_{new} is generated from scratch and, after that, the old tests are checked in order to filter only those that are applicable and merged to the new test suite, obtaining the final test suite $TS'_{new} = TS_{new} \cup TS_{old}$. Instead, with the incremental approach proposed in this paper, testers can build test suites incrementally, by starting from TS_{old}.

The second scenario we have identified, the SINC scenario, typically occurs when practitioners already have a complete test suite TS_{old} achieving the t-wise coverage, but a new one covering all $t+x$ (where $x \geq 1$) interactions is needed. This can occur when the tester has found no fault with interactions at strength t, and he/she wants to increase the test strength to exclude faults due to higher interaction levels. This is the scenario presented and addressed also in [11] with $x = 1$. Traditionally, this problem is tackled by generating a completely new test suite with the desired higher strength and dumping existing tests. However, this may be not the optimal solution, especially if the test suite generation requires a significant amount of time or if the existing tests have particular importance (for example they have been already manually executed). The existing test suite instead could be reused in order to discard all the $t+x$-uples that are already covered by TS_{old}. In this scenario, with the incremental generation, the new test suite is built starting from a non-empty TS_{old}.

IV. Experimental methodology

The experimental methodology we have used for gathering data for the evaluation of the approach is depicted in Fig. 2 and described in the following.

For the TSCP scenario (see Tab. I), given a combinatorial model we aim at comparing two different approaches: the one based on test generation from scratch and the incremental generation. Initially, by using pMEDICI, we generate the test suite starting from scratch (step 1 in Fig. 2a), as normally done when generating combinatorial test sets. Then we apply the incremental approach. In order to avoid possible influences given by test suites generated with the same tool, the incremental generation is performed starting from an initial test suite, representing TS_{acts}, generated by a different tool, namely ACTS [16] (step 2 in Fig. 2a). This test suite then is used to obtain the test suite TS_{old} by randomly selecting half of the test cases from TS_{acts} (step 3 in Fig. 2a). Finally, we apply incremental test generation using pMEDICI+ (step 4 in Fig. 2a) starting from TS_{old}.

On the other hand, for the SINC scenario (see Tab. I), given a combinatorial model we aim at comparing two different approaches generating test suites at higher strength (in our experiments 3): the one based on test generation from scratch and the incremental generation starting from a test suite with lower strength (in our experiments 2). Initially, we use the traditional approach by generating the test suite achieving the desired final combinatorial coverage ($t = 3$) with pMEDICI (step 1 in Fig. 2b). Then, we apply the incremental approach. First, we generate a test suite TS_{old} for strength $t = 2$ using pMEDICI (step 2 in Fig. 2b). Finally, we execute pMEDICI+ (step 3 in Fig. 2b) in order to generate a test suite with strength $t = 3$ starting from TS_{old}.

Research questions

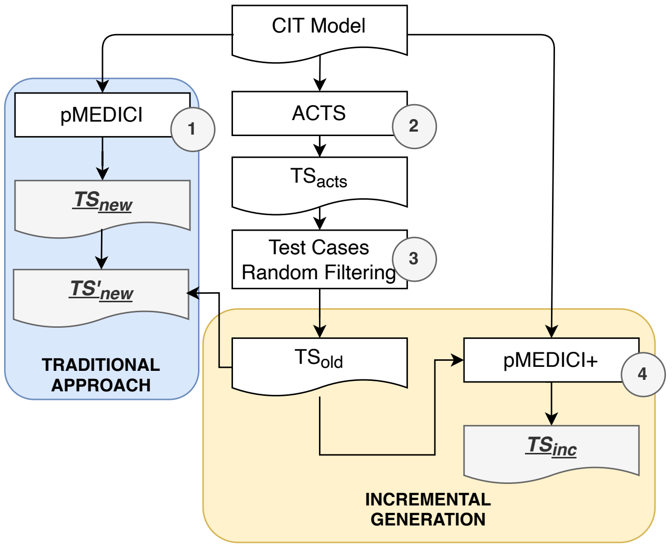
For comparing the traditional approach with that based on incremental generation, we have considered two measures:

- **Size**: the final size of TS_{new} vs TS_{inc}
- **Time**: test generation time required by pMEDICI+ compared with the time required by pMEDICI

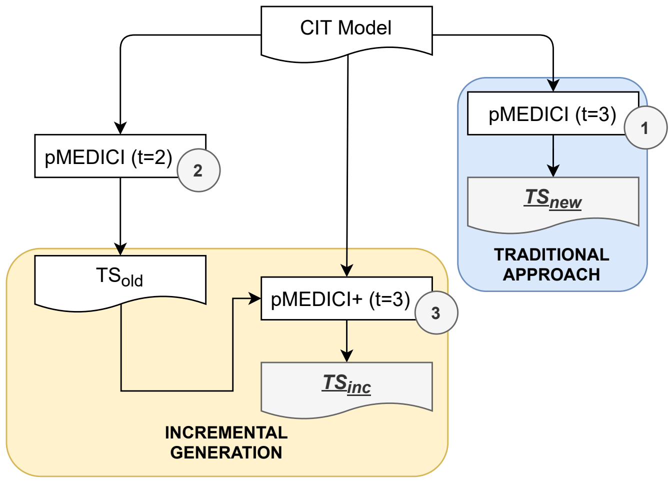
By combining measures and scenarios, we have devised the following four Research Questions:

	Size	Time
TSCP	RQ1	RQ2
SINC	RQ3	RQ4

Note that, for a higher statistical relevance and better generalization, we have decided to repeat each experiment 5 times and to consider the average results when answering the research questions. We emphasize that for the TSCP, in which the half of the test suite produced by ACTS is selected, a different random subset is taken at every repetition.



(a) TSCP: test suite completion



(b) SINC: test suite strength increase

Fig. 2: Experimental methodology for the two analyzed scenarios

Moreover, in order to compare the results obtained by each test generation strategy, we have decided to use the one-tailed Wilcoxon-Signed Rank test [15], 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 techniques (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 techniques are equal (i.e., $\bar{x}_1 = \bar{x}_2$), and the alternative hypothesis H_1 stating that the measure for the traditional technique is higher than that of the incremental one.

V. Experimental Results

In this section, we answer the research questions previously identified by executing the experiments as presented in Sect. IV. The experiments have been performed using the example benchmarks[1] given by the organizers of the second edition of the CT-Competition [4] at IWCT 2023 (with the exception of `NUMC` models with which pMEDICI+ can not deal) on a machine using a Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (16 physical cores, 32 logical cores) with 256 GB RAM. As per the CT-Competition rules, we have set a timeout of 300 seconds.

Tab. II shows the results we have obtained with the experiments described above, both for TSCP and SINC. In particular, the column referenced as **pMEDICI** reports the results obtained using the traditional approach, while that referenced as **pMEDICI+** reports the results obtained with the incremental approach.

[1]Benchmarks are available online at: https://github.com/fmselab/CIT_Benchmark_Generator/tree/main/Benchmarks_CITCompetition_2023

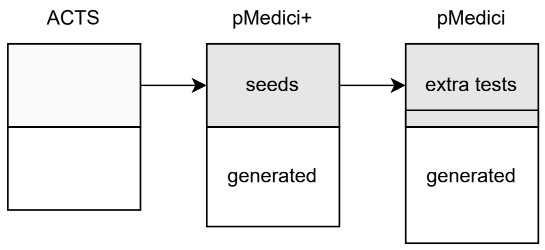


Fig. 3: Expected size of the test suites

RQ1: TSCP- Test suite size

In this research question, we analyze the impact of using the incremental test generation process with pMEDICI+ on the resulting test suite size. For this purpose, we compare the test suite size obtained when the generation is performed from scratch and the one obtained with incremental generation. The expected behavior is represented in Fig. 3. Let's suppose to have a former test suite (the one generated with ACTS), where half of the tests should be kept even after the new test generation. If we use pMEDICI+ with the incremental approach, the kept tests are used as seeds and only the missing interactions have to be covered in the `generated` part. On the contrary, if we generate the new test suite from scratch with pMEDICI, the probability of generating the same tests we wrote manually is very low, thus we need to append them to the generated test suite (those in the middle) and, possibly, to remove duplicates. For this reason, the number of tests is expected to be higher with pMEDICI than with pMEDICI+.

Our expectations have been confirmed by the results reported in Tab. II and Fig. 4: pMEDICI+ always produces test suites with a size lower (or equal) than those of pMEDICI. Note that these results are valid in the context of the TSCP,

TABLE II: Average of the results obtained in our experiments. In red those measures in which pMEDICI+ is outperformed by pMEDICI, in yellow those in which the two tools perform equally, and in green those in which pMEDICI+ performs better than pMEDICI

	$t = 2$ **(TSCP)**				$t = 3$ **(SINC)**			
	pMEDICI		**pMEDICI+**		**pMEDICI**		**pMEDICI+**	
Model	**Size**	**Time** $[s]$	**Size**	**Time** $[s]$	**Size**	**Time** $[s]$	**Size**	**Time** $[s]$
UNIFORM_BOOLEAN_0	20.8	0.03	10.4	0.02	32.8	0.03	29.8	0.03
UNIFORM_BOOLEAN_1	22.0	0.04	10.8	0.02	35.8	0.05	36.0	0.05
UNIFORM_BOOLEAN_2	18.0	0.06	10.6	0.06	28.4	0.03	25.0	0.02
UNIFORM_BOOLEAN_3	21.8	0.03	10.6	0.02	32.6	0.04	34.0	0.04
UNIFORM_BOOLEAN_4	15.2	0.02	8.8	0.02	22.6	0.03	19.8	0.03
UNIFORM_ALL_0	116.8	0.03	83.8	0.03	735.0	0.85	741.4	0.84
UNIFORM_ALL_1	339.6	0.60	236.4	0.48	–	–	–	–
UNIFORM_ALL_2	190.0	0.17	136.2	0.14	1519.8	45.65	1529.2	41.99
UNIFORM_ALL_3	433.2	0.51	300.8	0.36	5139.8	256.15	5126.8	259.95
UNIFORM_ALL_4	14.4	0.02	8.2	0.02	19.4	0.06	20.2	0.06
MCA_0	294.0	0.26	198.6	0.12	2542.4	30.25	2537.8	27.74
MCA_1	124.2	0.04	83.2	0.04	192.6	0.27	192.0	0.22
MCA_2	230.2	0.05	155.2	0.05	1443.4	1.33	1459.8	1.29
MCA_3	318.2	0.24	215.6	0.15	2882.4	44.28	2881.2	41.42
MCA_4	357.4	0.28	248.0	0.14	2926.4	32.61	2923.4	32.29
BOOLC_0	26.8	0.06	22.2	0.05	33.0	0.16	37.4	0.18
BOOLC_1	30.4	0.05	23.2	0.04	41.8	0.12	45.0	0.12
BOOLC_2	34.6	0.06	28.4	0.04	40.2	0.08	48.8	0.08
BOOLC_3	14.2	0.05	8.8	0.05	17.4	0.03	17.2	0.03
BOOLC_4	4.0	0.02	4.0	0.02	4.0	0.02	4.0	0.02
MCAC_0	–	–	–	–	–	–	–	–
MCAC_1	532.0	61.39	356.2	66.42	–	–	–	–
MCAC_2	206.4	0.51	140.6	0.48	894.2	9.49	896.6	8.87
MCAC_3	145.8	17.97	106.0	15.94	579.8	93.05	586.6	81.72
MCAC_4	–	–	–	–	–	–	–	–
FM_0	24.6	0.19	20.8	0.20	40.4	3.40	39.6	2.78
FM_1	23.2	0.03	15.2	0.02	31.0	0.06	34.4	0.05
FM_2	27.6	0.07	19.6	0.05	42.6	0.94	43.6	0.76
FM_3	24.6	0.06	24.0	0.06	43.0	0.97	44.2	0.76
FM_4	14.8	0.05	10.8	0.04	19.6	0.19	20.2	0.12
CNF_0	329.2	10.37	225.4	6.90	–	–	–	–
CNF_1	209.2	0.19	143.0	0.11	1333.6	5.28	1342.2	4.91
CNF_2	282.6	0.23	195.2	0.17	2030.6	9.83	2041.0	9.06
CNF_3	481.6	0.95	336.0	0.57	5564.2	224.33	5571.2	206.59
CNF_4	–	–	–	–	–	–	–	–
INDUSTRIAL_0	41.0	0.13	33.8	0.09	76.6	1.09	76.2	1.15
INDUSTRIAL_1	–	–	–	–	–	–	–	–
INDUSTRIAL_2	34.6	0.05	23.0	0.04	58.2	0.17	56.0	0.15
INDUSTRIAL_3	83.4	0.09	60.4	0.09	170.0	0.42	170.2	0.36
INDUSTRIAL_4	20.4	0.03	17.2	0.02	25.0	0.09	25.0	0.09
HIGHLY_CONSTRAINED_0	440.0	5.08	294.4	3.32	1485.0	235.43	1490.2	175.14
HIGHLY_CONSTRAINED_1	–	–	351.8	284.81	–	–	–	–
HIGHLY_CONSTRAINED_2	85.6	0.07	67.6	0.05	180.0	0.20	180.0	0.17
HIGHLY_CONSTRAINED_3	289.2	8.94	194.6	6.35	2002.4	104.76	2014.8	108.29
HIGHLY_CONSTRAINED_4	259.4	1.27	180.0	1.49	1326.4	24.49	1329.2	24.07

in which the tests in TS_{old} are needed even after the new test generation. Indeed, the size of pMEDICI includes the portion of tests generated by ACTS to be kept (excluding repetitions). However, as previously introduced, in order to statistically verify our considerations, we have performed a Wilcoxon-Signed Rank test and we have obtained $pvalue = 2.63 \cdot 10^{-8}$ which has allowed us to reject the null hypothesis and claim that applying the incremental generation leads to a lower test suite size than the non-incremental approach.

RQ2: TSCP- Test suite generation time

In this research question, we analyze the impact of using the incremental test generation process with pMEDICI+ on the test suite generation time. For this purpose, we compare the time required for the generation, for each model, of the test suite from scratch and that required when using the incremental generation. Given that the incremental generation is based on seeds possibly generated by another tool (ACTS in our case), the impact depends on the tuple density of the tests in the seeds. However, when a new tuple has to be covered by pMEDICI, it is required to check the compatibility of the tuple

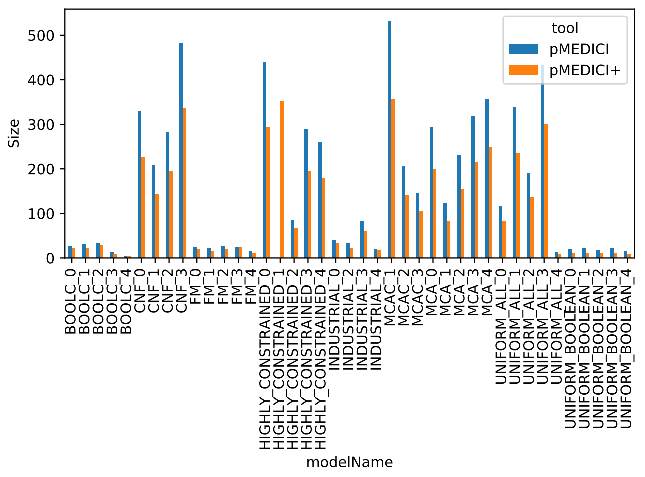


Fig. 4: Test suite size comparison between pMEDICI and pMEDICI+ in the TSCP scenario

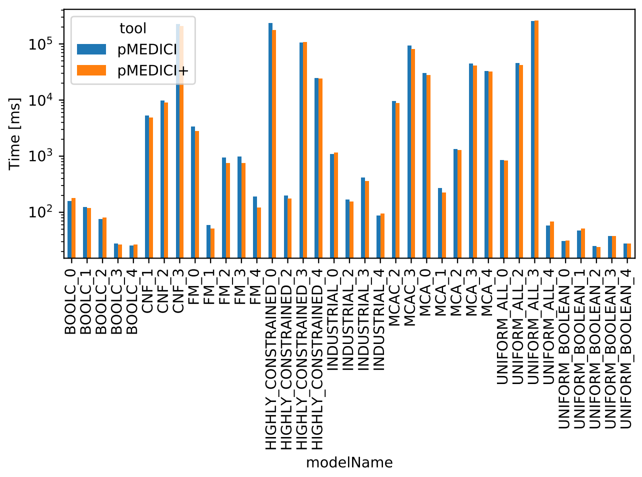


Fig. 5: Test suite generation time comparison between pMEDICI and pMEDICI+ in the SINC scenario

with the constraints by computing intersections among MDDs and, in general, it is one of the most expensive operations performed by the test generation tool. On the contrary, starting from a non-empty test suite allows avoiding that costly check (since comparing the values of the parameters in the tuple with those in the seeds is enough) and to saving lot of time.

These preliminary observations have been confirmed by the results reported in Tab. II: pMEDICI+ performs worse than pMEDICI in only three benchmarks (i.e., `MCAC_1`, `FM_0` and `HIGHLY_CONSTRAINED_4` - those highlighted in red) and is able to complete the test generation process also for the model `HIGHLY_CONSTRAINED_1` which is not completed by pMEDICI.

As previously introduced, in order to statistically verify our considerations, we have performed a Wilcoxon-Signed Rank test and we have obtained $pvalue = 5.71 \cdot 10^{-5}$ which has allowed us to reject the null hypothesis and claim that applying the incremental generation leads to a lower generation time than the non-incremental approach.

RQ3: SINC- Test suite size

In this research question, we analyze the impact of incremental test generation on the test suite size when testers want to increase the strength of the test suite. As a preliminary consideration, we can suppose that using a test suite produced for a lower strength may contribute in increasing the final test suite size. Indeed, in that case, the former test suite may not be optimized for the desired new strength and generating the new test suite from scratch may be advised.

These considerations have been confirmed by the results reported in Tab. II: 22 out of the 37 solved benchmarks have an higher test suite size with pMEDICI+ than with pMEDICI.

As previously introduced, we have statistically verified our considerations by performing a Wilcoxon-Signed Rank test. The analysis produced $pvalue = 0.99$ which has not allowed us to reject the null hypothesis. Thus, we cannot draw any conclusion from this test. In order to analyze whether the incremental approach actually performs worse or equally than the most traditional one, we have also executed Wilcoxon-Signed Rank test by inverting the data. In this way, the alternative hypothesis H_1 states that the non-incremental technique is lower than that of the incremental one. This additional analysis produced $pvalue = 0.01$ which has allowed us to reject the null hypothesis and to determine, as stated in the preliminary observation of this RQ, that the incremental technique is outperformed by the traditional approach, in terms of test suite size, in the SINC scenario.

RQ4: SINC- Test suite generation time

With this research question, we analyze the impact of using the incremental test generation process with pMEDICI+ on the test suite generation time, when a test suite with lower strength (e.g., $t = 2$) is available and a new one with higher strength (e.g., $t = 3$) is needed. As highlighted in RQ2, being based on seeds allows pMEDICI+ to start from a non-empty test suite and, thus, to avoid checking the compatibility of the already covered tuples with the MDD internal to the test contexts. Considering that updating the MDD and checking the compatibility are the two most expensive operations, avoiding them for a set of tuples allows reducing the generation time.

The results obtained with our experiments are reported in Fig. 5 and Tab. II. Those results confirm our preliminary considerations. In fact, pMEDICI+ (with the incremental approach) requires more time for generating test suites in only 5 instances, while in all the others, it performs equally or better than pMEDICI.

The statistical confidence of these considerations has been investigated by performing a Wilcoxon-Signed Rank test that produced $pvalue = 1.1 \cdot 10^{-3}$. In this way, we have been able to reject the null hypothesis and claim that applying the incremental generation leads to a lower generation time than the non-incremental approach, even when the strength of the test suite needs to be increased.

VI. Threats to validity

This work presents our technique for incremental generation of combinatorial test cases and preliminary experiments show its viability. However, we are aware that this study has several threats to validity [13] we discuss in this section together with the actions we have taken to mitigate them and possible future works.

A first threat to validity regards how we have generated the seeds in the experiment for the first scenario TSCP. Normally we can assume that these seeds are generated by humans, looking at the specifications or by some domain experts that design a required set of tests. In our experiments, instead, we have generated the initial seeds using another tool (ACTS) and taken only half of them. The use of a tool is required because we wanted to apply our technique to several randomly generated case studies (those for the CT-Competition), so writing manually the initial tests would have been time-consuming without any real benefit. We have preferred to use another tool and not ours because using the same tool for the generation of the seeds and then for the rest of the test suite could imply benefits for pMEDICI+. We plan in the future to generate the initial test suite using other techniques (also randomly) or other tools. Furthermore, an additional threat to validity regards the impact of seeds' size on the performance of the generation algorithm. In our experiments, we have kept half of the tests in TS_{ACTS} as seeds, but different percentages may lead to different results. We plan to investigate the impact of this aspect in future work.

Another threat to internal validity regards the correctness of our implementation of the incremental generation. To mitigate this risk, we have validated all the test suites [3] in order to be sure that also pMEDICI+ generates test suites that are valid and complete.

An external threat to the validity of the entire approach is that incremental generation of tests may be not so useful in practice. For this reason, we have identified two scenarios in which we believe incremental generation is useful, as demonstrated by the other incremental approaches available in the literature (see Sect. VII). As future work we plan to find other use cases, for instance when a test suite contains some tests that are incomplete, i.e., they do not assign values to all the parameters, or scenarios in which some tests or part thereof must be discarded because some new constraints are added. We believe that incremental generation can be useful in many cases when test suite and combinatorial models co-evolve [14].

VII. Related work

Incremental generation of test suites has been tackled in other papers by applying it in different contexts as well. Our framework aspires to recap in a single framework different approaches and scenarios in which incremental test generation is advisable.

Incremental generation of combinatorial test suites is often applied 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. The problem is highlighted, in the case of combinatorial test suites, in [14] where the authors combine three building blocks, allowing to minimally modify existing tests, enhance them, or choose from them selected test cases for obtaining a new test suite, composed of only valid tests. Unfortunately, the tool FOCUS presented in [14] is not available, otherwise we could have compared pMEDICI+ with FOCUS. Similar considerations are presented in [12], where an approach for automatically repairing and generating test cases during software evolution is devised. In our scenarios, we assume that the model does not change and that the seed tests are valid, but the entire approach and the algorithm would work even if the seed tests are partially invalid or even if the seeds are incomplete.

The idea of seeding we exploit in pMEDICI+ was originally proposed in [10]: "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". The problem of using seeds is tackled also by [8], in which an algorithm for prioritized interaction testing for 2-wise coverage is described. It supports mainly soft constraints, but it is shown to be extensible to allow seeds.

Using a previously computed test suite for achieving higher strengths coverage has been proposed in [11], 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 as we, instead, do in this paper being based on the *one-test-at-a-time* approach with pMEDICI+. 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 incremental generation is presented in [9]. It exploits some algebraic properties of Covering Arrays (CA) and allows the fast generation of a CA of strength t given an already CA of strength $t-1$. The algorithm is very fast, but it produces very large test suites and it is not able to consider also constraints. Generating test sets from seeds is also supported by other tools such as ACTS [16], jenny [1], and PICT [2]. However, none of them is based on a multithread algorithm as pMEDICI+. Moreover, with pMEDICI+ we are able to deal with test cases that are not valid anymore, but can still be partially reused. Instead, at the best of our knowledge, other available tools completely discard non valid test cases.

VIII. Conclusion

Many CIT generators have been proposed in the last decades. However, with the majority of them, when the test generation is repeated, all the old tests are generally lost.

This may be an inefficient approach, especially when part of the former test suite is still applicable. For this reason, in this paper, we propose an architecture, supported by the pMEDICI+ tool, allowing testers to reuse a former test suite during the test generation process. This approach can be used in several different scenarios and, in this paper, we analyze two different scenarios of interest: when some of the former tests have been manually written and are still required, and when a new test suite with higher strength is to be generated.

With our experiments, carried out on the training set given in the context of the second edition of the CT-Competition, we have verified that using the incremental test generation plays an important role in reducing the test suite generation time and, for many benchmarks, the size. As a consequence, the cost of the test generation process is reduced as well.

As a future work, we are working on letting users choose the solver to be used in test contexts (to be substituted to the MDDs), in order to overcome the limitations highlighted for pMEDICI+ while still keeping the same tool structure and functionalities. This may be useful also in terms of performance, since more powerful structures, or different MDDs implementations, may be used for constraint handling. Moreover, we are working on a preprocessing activity on the test cases in order to randomly generate the seeds and start the test generation from those. In this way, we could exploit the boost in performance given by incremental generation of pMEDICI+ without the need of asking users to provide a former test suite. Additionally, we believe that incremental generation may be applied to other architectures and problems, given that they are based on *collecting* tuple and producing one test at a time, such as the one presented in [5]. Finally, we plan to compare pMEDICI+ with the tools already supporting test generation from seeds, and to investigate the impact of using multithreading for incremental test generation.

References

[1] Jenny website. http://burtleburtle.net/bob/math/jenny.html.

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

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

[4] A. Bombarda, E. Crippa, and A. Gargantini. An environment for benchmarking combinatorial test suite generators. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, apr 2021.

[5] A. Bombarda and A. Gargantini. An automata-based generation method for combinatorial sequence testing of finite state machines. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, oct 2020.

[6] A. Bombarda and A. Gargantini. Parallel test generation for combinatorial models based on multivalued decision diagrams. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 74–81, Los Alamitos, CA, USA, 4 2022. IEEE Computer Society.

[7] A. Bombarda, A. Gargantini, and A. Calvagna. Multi-thread combinatorial test generation with smt solvers. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[8] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pairwise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, oct 2006.

[9] A. Calvagna and E. Tramontana. Incrementally applicable t-wise combinatorial test suites for high-strength interaction testing. In *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*. IEEE, jul 2013.

[10] D. Cohen, S. Dalal, M. Fredman, and G. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, jul 1997.

[11] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*. ACM Press, 2009.

[12] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, Apr. 2012.

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

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

[15] R. F. Woolson. Wilcoxon signed-rank test, Sept. 2008.

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