# Parallel Test Generation for Combinatorial Models Based on Multivalued Decision Diagrams

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*—Combinatorial interaction testing (CIT) is a testing technique that has proved to be effective in finding faults due to the interaction among inputs, and in reducing the number of test cases. One of the most crucial parts of combinatorial testing is the test generation for which many tools and algorithms have been proposed in recent years, with different methodologies and performances. However, generating tests remains a complex procedure that can require a lot of effort (mainly time). Thus, in this paper, we present the tool pMEDICI which aims to reduce the test generation time by parallelizing the generation process and exploiting the recent multithread hardware architectures. It uses Multivalued Decision Diagrams (MDDs) for representing the constraints and the tuples to be tested and extracts from them the t-wise test cases. Our experiments confirm that our tool requires a shorter amount of time for generating combinatorial test suites, especially for complex models, with a lot of parameters and constraints.**

***Index Terms*—combinatorial testing, multithread test generation, multivalued decision diagrams**

## I. Introduction

Combinatorial interaction testing (CIT) has been an active area of research in the last years since it has proven to be very effective to test complex systems with multiple input parameters, and to help testers in finding defects due to the interaction of different inputs. This interaction is tested in a systematic way. In general, in t-way testing, every t-uple of parameter values must be tested at least once. However, generating test suites that guarantee the desired t-way coverage is not always an easy task, since it may require a lot of time or may produce a test suite with a high number of test cases. This is the rationale behind several research works, presenting tools for test generation[1]. In most cases, testers want to minimize the test suite size, since the time for executing tests on the actual systems is usually higher than the one required for their generation. Nevertheless, in some cases (especially for small and not really complex systems), a high generation time can discourage testers to test their software.

Thus, considering that modern PCs are all equipped with multicore and multithread CPUs, we have tried to investigate whether was possible to develop a tool exploiting multiple threads in order to speed up the test generation process.

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

In this paper, we present the pMEDICI combinatorial test generator tool. It exploits MDDs (Multivalued Decision Diagrams) and multiple threads for generating combinatorial test suites with a lower generation time. As other test generation tools (such as MEDICI [7]) it is able to deal with combinatorial models with parameters, both boolean or enumeratives, and constraints expressed as logical formulas. With our experiments, pMEDICI has proved to be very effective in terms of test suite generation time, especially for models without constraints.

The paper is structured as follows. In Sect. II, we present the general background on CIT and MDDs. Then, Sect. III presents the pMEDICI combinatorial test generator, its structure and algorithm, and the basic idea underlying the use of MDDs for CIT. The performances of the tool are evaluated in Sect. IV by using the benchmark examples provided in the context of the CT-Competition. In Sect. V we present relevant related work and, finally, Sect. VI presents some future work and concludes the paper.

## II. Background

### A. Combinatorial Interaction Testing

With Combinatorial Interaction Testing (CIT), the tester systematically explores the t-way interaction between all the features inside a given system. This is obtained by combining all the t-tuples of parameter assignments in the smallest possible number of test cases and aims to reduce the time required for testing, by decreasing the number of test cases. The most commonly applied combinatorial testing technique is pairwise testing, which consists in generating a test suite covering all pairs of input values (each pair in at least one test case).

Listing 1 represents a simple example of a combinatorial model, where a list of parameters and a list of constraints are specified using the CTWedge format [6]. In particular, the model contains three parameters (one boolean parameter and two enumeratives) and two constraints, which express the logical relation between the parameter and their values.

If a tester wants to exhaustively test the system modeled as per Listing 1, $2 \cdot 2 \cdot 2 = 8$ tests should be performed, if the constraints are ignored. Instead, using pairwise testing,

**Model** Example1

**Parameters**:
P1: { v1 v2 };
P2: { v3 v4 };
P3: Boolean;

**Constraints**:
# ( P3!=true OR P2!=v3 OR P1!=v1) #
# ! (P3 = true AND P1 = v2) #

Listing 1. Example of a combinatorial model

only $4$ tests are sufficient to cover all the possible couples of parameter values.

A combinatorial model, such as the one presented in Listing 1, is usually given as input to a tool, called *Test Generator*, that generates a test suite with combinatorial coverage.

### B. Multivalued Decision Diagrams

The tool presented in this paper is built on the top of the concept of the *decision diagrams*, as defined in the following.

**Definition 1** (Decision diagram)**.** *A decision diagram is a graph that represents a function* $f : D \rightarrow B$ *where* $D = D_1 \times \cdots \times \cdots D_n$ *and* $B$ *is the Boolean domain, i.e.,* $B = \{F, T\}$.

In general, a decision diagram is used to evaluate the truth value of $f$ when applied to the variables $x_1, \cdots, x_n$. If all the domains $D_i$ are binary, then we use Binary Decision Diagrams (BDDs) to represent Boolean functions. BDDs are widely used within the domain of system design verification. Multivalued Decision Diagrams (MDD), instead, extend BDDs by allowing every variable to have a different domain with a different size. Each MDD respects the following properties:

- only two terminal nodes are available, which are labeled as `F` and `T`;
- every non-terminal node is labeled by an input variable $x_i$ and has $|D_i|$ outgoing labeled edges, i.e. one per each possible value of the domain;
- every variable appears only once in the MDD, in any path from the root to a terminal node;

Given these properties, an MDD can select which values of the domain $D$ are selected by the function $f$. In fact, if the values $x_1, \ldots, x_n$ for the variables in $D$ are selected by $f$, then $f(x_1, \ldots, x_n) = T$, otherwise $f(x_1, \ldots, x_n) = F$.

Typically, among MDDs, it is possible to perform unary operations such as *complement*, computation of the *cardinality*, or the most classical binary operations like *union*, *intersection*, and *difference*. In particular, since MDDs can represent logic functions, operations among MDDs are equivalent to logic operations. Given an MDD $M$ representing the function $f$, its complement $\neg m$ represents the function $\neg f$. The union between two MDDs $M_1 \cup M_2$ represents the function $f_1 \vee f_2$. The intersection between two MDDs $M_1 \cap M_2$ represents the function $f_1 \wedge f_2$. Finally, given an MDD $M$, its cardinality $|M|$ represents the number of all the possible paths to the terminal node `T`. The value of cardinality of an MDD can be used to check the consistency between boolean functions, i.e. if $f_1(x)$ and $f_2(x)$ are inconsistent, the intersection between the MDDs representing the two functions is empty, for all the values of $x$.

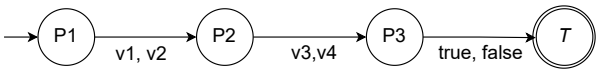


Fig. 1. MDD structure for the combinatorial problem in Listing 1 if the constraints are ignored

For the work presented in this paper, we have used mddlib [2], by the Consortium for Logical Models and Tools, which is the only open-source Java library that natively supports MDDs and allows the computation of operations between them.

## III. pMEDICI: A PARALLEL MDD-BASED ALGORITHM FOR CIT

In this paper, we present pMEDICI, a Java tool exploiting MDDs for generating test suites with combinatorial coverage. It implements multithreading strategies for reducing the time required for test generation.

### A. How to deal with CIT models with MDDs

As explained in Sect. II, MDDs allow expressing boolean functions. Hence, an MDD can be used to represent the boolean function computing the validity of the assignments to each parameter in the combinatorial model. If ignoring the constraints, a combinatorial model with $n$ parameters with cardinality $p_i$ can be easily represented using an MDD $M_{TS}$ with $n$ non-terminal nodes (with the name of the corresponding parameter each) and with $p_i$ outgoing labeled edges for all the parameters except the last one, which has only one edge connected to the terminal node $T$ - since each assignment is valid if constraints are ignored. Fig. 1 represents the MDD $M_{TS}$ associated with the combinatorial model shown in Listing 1. Every path from the root to the terminal `T` is a syntactically correct assignment of values to the parameters. Thus, the MDD $M_{TS}$ represents all the tests, i.e., all the possible paths from the start to the terminal node. Thus, its cardinality is equal to $\prod_{i=1}^{n} p_i$ and represents the total number of possible tests.

If also the constraints are considered, some of the possible paths, i.e. those that will violate the constraints, will lead to the terminal node `F`. In fact, the constraints corresponding to a CIT problem can be described using propositional logic[3]. Every constraint can be represented as a boolean formula containing the operators $\neg, \vee, \wedge$ and equality between the parameters and their values. Then, each constraint can be represented by an MDD modeling its truth function: it can be built using the equivalence between MDDs and boolean formulas proposed, and the operations among MDDs presented in Sect. II. In particular, the intersection between the MDD $M_{TS}$ and the

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

[3]There are some limits that we will discuss later.

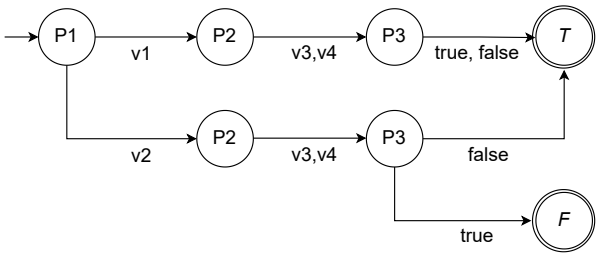


Fig. 2. MDD structure when a constraint is included

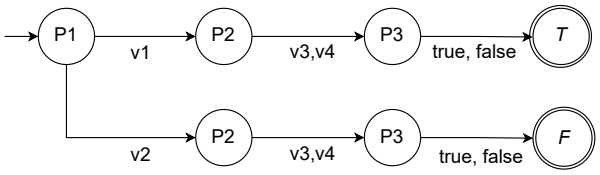


Fig. 3. MDD structure with the assignment P1 = v1

MDDs derived from each constraint is a new MDD that accepts only valid tests which comply with all the constraints. For example, Fig. 2 shows how the MDD in Fig. 1 evolves when the second constraint (!(P3 = true AND P1 = v2)) of Listing 1 is included.

### B. How to represent (partial) tests with MDDs

First, note that an MDD can be modified by adding a tuple to it. In this way, an MDD can keep track of the values assigned to the parameters in a given test. Formally, the MDD can be updated by computing its intersection with the MDD representing the assignments $p_i = v_{i,j}$ contained in the tuple. For instance, by adding the assignment P1 = v1, the original MDD of Fig. 2 becomes the MDD shown in Fig. 3.

As we will discuss later in detail, pMEDICI builds the tests incrementally, by collecting suitable tuples in order to obtain valid tests. In order to store the information about the model and its constraints, together with the tuples added so far to a (partial) test, we introduce the notion of *test context*.

**Definition 2** (Test context)**.** *We call $TC = \langle A_i, M_{TS} \rangle$ a* test context *for $P$, where $A_i$ 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.*

Each test context $TC$ contains a list of assignments that represents a *partial* test case $T$ together with the MDD representing all the information about the model and the test itself. A test context is complete, i.e. it represents a complete test case if the list of assignments $A_i$ includes all the parameters of the model.

Given a test context $TC$, and be $M_{TS}$ its MDD, a tuple $tp$ can be:

- *implied*, if all the assignments of the tuple $tp$ are already contained in the test case $T$. In this case, the check of the compatibility of $tp$ with the MDD $M_{TS}$ is not required;

**Algorithm 1** Tuple building procedure

**Require:** $TupBuilder$, the thread building the tuples
**Require:** $P$, the map containing all the parameters of the CIT model and their possible values
**Ensure:** $TupBuffer$, the buffer containing the tuples built by $TupBuilder$

▷ Create the tuple builder thread
1: $TupBuilder.create()$
▷ Generate the tuples
2: $TupBuilder.run($
3: **for each** $t_i \in cartesian\_product(P)$ **do**
4: $TupBuffer.add(t_i)$
5: **end for**
6: )

- *compatible*, if the tuple $tp$ contains only assignments which are not in conflict with those of the test context $TC$, i.e. in the test $TC$, each parameter contained in the tuple $tp$ is still not valorized or has an equal value, and $t_p$ does not clash with the constraints of the combinatorial problem.
- *uncoverable*, if the assignments contained in the tuple $tp$ clash with the constraints of the combinatorial problem. This check requires the use of the MDD $M_{TS}$ containing only the constraints.

### C. Tool structure and algorithm

The structure of the pMEDICI tool is shown in Figure 4. A single thread generates all the tuples starting from a CIT model (such as the one in Listing 1), exploiting the cartesian product, and stores them in a shared buffer with a limited capacity, as shown in Algorithm 1. The tuple generation thread sleeps until some of the tuples are consumed and more space for other tuples becomes available. This process allows to avoid storing all the tuples and, thus, guarantees a consistent save in memory utilization, especially for complex combinatorial models. When at least a tuple is available, $n$ threads (where n can be either specified by the user or automatically selected by the tool depending on the execution environment architecture) start consuming each tuple. Each tuple $t_j$ in the shared buffer is consumed by one thread which searches the best test context and drops $t_j$ in it. Test contexts are continuously updated as each tuple is managed by a thread. This process is described in Algorithm 2 and is repeated until all the tuples have been consumed. In particular, given a tuple $t_j$:

- if a thread finds a test context $tc_i$ in which the tuple $t_j$ is already *implied*, then $t_j$ is consumed and marked as covered;
- if a thread finds a test context $tc_i$ in which the tuple $t_j$ is *compatible*, $t_j$ is passed to $tc_i$, which updates its MDD structure (if any constraint is present). Then, $t_j$ is consumed, and marked as covered;
- if a thread can not find a test context $tc_i$ in which the tuple $t_j$ is compatible or implied, a new test context is created

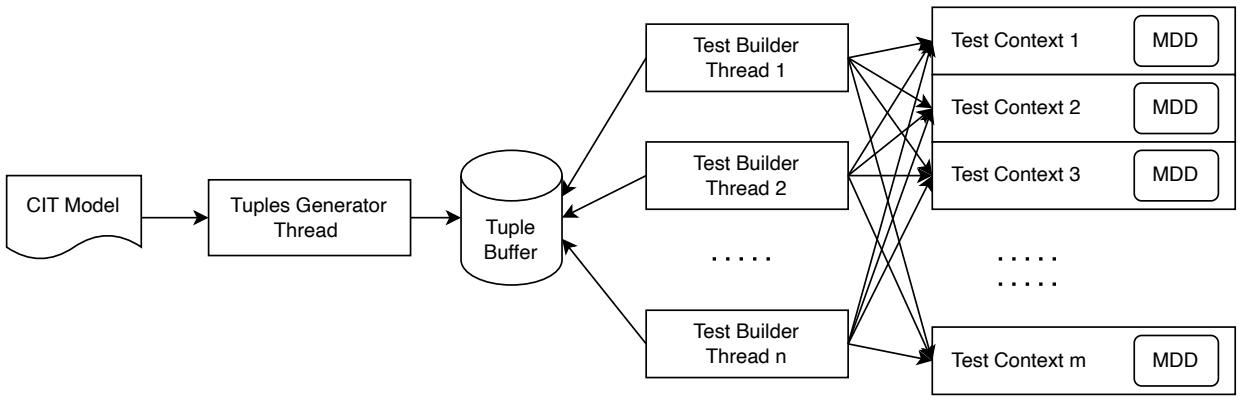


Fig. 4. Structure of the pMEDICI tool

**Algorithm 2** Tuple consumption procedure

**Require:** $TupBuffer$, the buffer containing the tuple already produced and ready to be consumed
**Require:** $TC$, the list of all the test contexts
**Require:** $M_C$, the CIT model

▷ Extract the tuple from the tuple buffer
1: $t_j \leftarrow TupBuffer.extractFirst()$
▷ Try to find a test context which implies the tuple
2: $tc \leftarrow findImplies(TC, t_j)$
3: **if** $tc$ is not $NULL$ **then**
4: $t_j.setCovered()$
5: **return**
6: **end if**
▷ Try to find a test context which is compatible with $t_j$
7: $tc \leftarrow findCompatible(TC, t_j)$
8: **if** $tc$ is not $NULL$ **then**
9: $tc.updateTC(t_j)$
10: $t_j.setCovered()$
11: **return**
12: **end if**
▷ Create a new empty test context
13: $tc \leftarrow createTestContext()$
14: $tc.addConstraints(M_C.getConstraintsList())$
15: **if** $tc.isCompatible(t_j)$ **then**
16: $tc.updateTC(t_j)$
17: $t_j.setCovered()$
18: **else**
19: $t_j.setUncovered()$
20: **end if**
21: **return**

together with its MDD that initially contains only the constraints. If $t_j$ is compatible with the newly created test context, the tuple is consumed and marked as covered, otherwise, it means that the tuple is not compatible with the constraints, is marked as *uncoverable* and skipped.

Note that pMEDICI could perform the compatibility check with the constraints even during the tuple generation, before trying to add the tuple $t_j$ to any test context. We plan to add this feature in a future version of the tool.

When all the tuples are consumed, each test context provides the resulting test case. Thus, the test suite is composed of the tests generated by all the test contexts. Algorithm 3 reports the process used for collecting all the tests from the test contexts.

**Algorithm 3** Tests collection

**Require:** $TC$, the list of all the test contexts
**Require:** $Threads$, the list of all the test builder threads
**Require:** $TupBuilder$, the thread building the tuples
**Ensure:** $TS$, the vector containing the test cases

▷ Join all the threads
1: **for each** $thread \in Threads$ **do**
2: $thread.join()$
3: **end for**
4: $TupBuild.join()$
▷ Gather all the tests generated by the test contexts
5: **for each** $tc \in TC$ **do**
6: $TS.add(tc.getTest())$
7: **end for**

### *D. Algorithm optimizations*

Several optimizations can be performed over the pMEDICI algorithm shown in the previous section. In particular, one can optimize three main aspects in the process:

- test context selection: the threads building the tests, can select the test context according to some policies (for instance, by giving precedence to those having some relations with the tuple to be added)
- management of the constraints: the test context could optimize the storage of constraints, or in case there are none, simplify the process
- tuple selection: the test builders could choose particular tuples from the tuple buffer.

In terms of context selection, the *findImplies* and *findCompatible* (see Algorithm 2, lines 2 and 7) operations can be optimized by ordering the test contexts. Preferable tests

contexts are those in which the cardinality of the MDD (number of possible paths from the first parameter to the `true` leaf) after the addition of the tuple is higher, since it allows to create fewer tests with higher variability. The current version of pMEDICI has already implemented this optimization. However, as shown in the experiments in Sect. IV, ordering the test contexts is not the best idea when working on models with a lot of parameters or constraints, since the time required for the ordering process overpass that of the test generation algorithm. For this reason, the user can choose to exclude this optimization.

Moreover, pMEDICI can avoid the use of MDDs if no constraint is present in the combinatorial model under analysis. In this way, we can avoid computing intersections among MDDs, which is one of the most expensive operations in terms of time. The instruction $tc.updateTC(t_j)$ can be substituted by the following conditional statement:

---

$tc.updateAssignments(t_j)$
**if** $tc.getConstraintsList().size() > 0$ **then**
    ▷ Update the MDD of the test context with $t_j$
  $tc.updateMDD(t_j)$
**end if**

---

As future work, we are planning to include another optimization, in terms of the selection of the tuples by the test builder threads from the tuple buffer. Since the update of an MDD require a lot of time to be performed (as an intersection between two different MDDs has to be computed), a thread could select in a smarter way the tuple to be managed by a test context. In fact, a tuple $t_i$ can be easily covered by a test context $tc$ which is similar to the tuple. This operation can be easily implemented by modifying line 1 of Algorithm 1.

### *E. Tool limits*

Despite being based on the powerful structure of MDDs, pMEDICI still has some limits which are directly connected to the limitations of MDDs. In particular, pMEDICI is not able to deal with models containing constraints with:

- Arithmetical operators, such as $+$, $-$, $\cdot$ and $/$;
- Comparisons between parameters, such as $p_1 = p_2$ or $p_1 \neq p_2$.

In principle, MDDs could deal also with arithmetic and variable comparison by converting the constraints to pure boolean expressions. However, in practice, this conversion is not easy to be done and risks to generate constraints with exponential length.

Moreover, the use of MDDs requires the translation of each constraint in RPN (Reverse Polish Notation) which may decrease the readability of the models.

SMT (Satisfiability Modulo Theories) solvers could be used for CIT in alternative to MDDs since they allow to check if a formula (the tuple) is satisfiable w.r.t. the constraints of the model. However, using MDDs allows extracting several metrics from the test suite, such as its complexity, or cardinality.

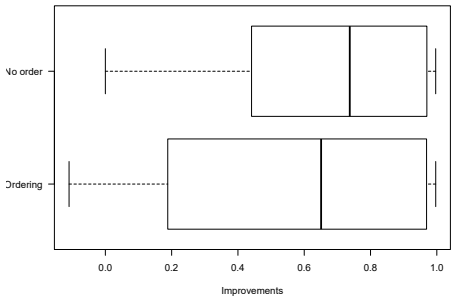


Fig. 5. Improvements of time for pMEDICI

We plan in the future to take advantage of SMT in order to solve some of the limitations of MDDs for test generation.

## IV. Experiments

In this section, we report the results we have obtained using the pMEDICI tool over the benchmark examples provided by the organizers of the CT-Competition [1]. All the experiments are performed using strength $t = 2$. As explained in Sect. III-E, due to the structure of MDDs, our tool pMEDICI is able to deal only with logical operators and not with relational and mathematical ones. Thus, we intend to compete only in the `UNIFORM`, `MCA`, `BOOLC`, and `MCAC` categories. Table I reports the results, in terms of size and generation time for all the benchmarks of these categories using the number of threads automatically chosen by the tool, depending on the execution environment architecture. We have executed the experiments on a PC with Windows 11, Intel i7-3770 with 3.4 GHz, 8 threads, and 32 GB RAM. As required by the competition regulation we have used a maximum execution time of 300 seconds. Note that for the benchmarks with size 0 and generation time 0, we have verified that the generated test suites are always empty (even with other generators like ACTS [16] and MEDICI [7]) since the constraints are too strict. However, in order to further test pMEDICI over models with constraints, we have generated some additional benchmark models (using the tool provided by competition organizers[4]), which have been verified and must produce not empty test suites, and we have reported the obtained results in Tab. II.

### *A. Comparison between pMEDICI and MEDICI*

In Tab. I and Tab. II, we have gathered the results obtained with the comparison between the performance of pMEDICI and MEDICI [7], from which our tool derives.

From our experiments, we have noticed that for each model, either with or without constraints, pMEDICI always requires less time for the test suite generation. The improvements in terms of time with respect MEDICI, computed as $(T_{MEDICI} - T_{pMEDICI})/T_{MEDICI}$, is shown in Fig. 5. As

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

TABLE I
PERFORMANCE OF PMEDICI OVER THE BENCHMARKS AND COMPARISON WITH MEDICI

| Benchmark name | # Params | # Constraints | pMEDICI with ordering | | pMEDICI without ordering | | MEDICI | |
|---|---|---|---|---|---|---|---|---|
| | | | Size | Generation time $[s]$ | Size | Generation time $[s]$ | Size | Generation time $[s]$ |
| UNIFORM_BOOLEAN_0 | 11 | 0 | 14 | 0.033 | 15 | 0.027 | 7 | 0.039 |
| UNIFORM_BOOLEAN_1 | 4 | 0 | 8 | 0.015 | 9 | 0.012 | 5 | 0.040 |
| UNIFORM_BOOLEAN_2 | 15 | 0 | 19 | 0.048 | 16 | 0.040 | 8 | 0.076 |
| UNIFORM_BOOLEAN_3 | 20 | 0 | 27 | 0.063 | 21 | 0.059 | 9 | 0.195 |
| UNIFORM_BOOLEAN_4 | 17 | 0 | 19 | 0.056 | 19 | 0.050 | 8 | 0.109 |
| UNIFORM_ALL_0 | 10 | 0 | 445 | 0.689 | 437 | 0.669 | 259 | 22.283 |
| UNIFORM_ALL_1 | 20 | 0 | 774 | 3.182 | 745 | 3.195 | 432 | 90.331 |
| UNIFORM_ALL_2 | 2 | 0 | 144 | 0.033 | 144 | 0.037 | 144 | 0.058 |
| UNIFORM_ALL_3 | 13 | 0 | 139 | 0.219 | 155 | 0.194 | 88 | 8.846 |
| UNIFORM_ALL_4 | 3 | 0 | 604 | 0.303 | 593 | 0.320 | 423 | 40.897 |
| MCA_0 | 4 | 0 | 1,503 | 0.611 | 1,491 | 0.627 | 1,419 | 186.291 |
| MCA_1 | 11 | 0 | 2,197 | 4.855 | 2,235 | 5.396 | 1,342 | 140.041 |
| MCA_2 | 17 | 0 | 1,406 | 1.669 | 1,395 | 1.701 | 1,376 | 279.026 |
| MCA_3 | 8 | 0 | 63 | 0.042 | 63 | 0.050 | 63 | 0.222 |
| MCA_4 | 8 | 0 | 1,532 | 1.240 | 1,522 | 1.255 | 1,496 | 225.149 |
| BOOLC_0 | 14 | 1 | 11 | 0.245 | 12 | 0.166 | 8 | 0.297 |
| BOOLC_1 | 8 | 9 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| BOOLC_2 | 3 | 55 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| BOOLC_3 | 7 | 83 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| BOOLC_4 | 20 | 56 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| MCAC_0 | 11 | 49 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| MCAC_1 | 20 | 30 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| MCAC_2 | 5 | 47 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| MCAC_3 | 8 | 97 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |
| MCAC_4 | 19 | 27 | 0 | 0.000 | 0 | 0.000 | 0 | 0.000 |

TABLE II
PERFORMANCE OF PMEDICI OVER THE ADDITIONAL BENCHMARKS AND COMPARISON WITH MEDICI

| Benchmark name | # Params | # Constraints | pMEDICI with ordering | | pMEDICI without ordering | | MEDICI | |
|---|---|---|---|---|---|---|---|---|
| | | | Size | Generation time $[s]$ | Size | Generation time $[s]$ | Size | Generation time $[s]$ |
| ADD_BOOLC_0 | 20 | 13 | 16 | 0.403 | 14 | 0.395 | 11 | 1.061 |
| ADD_BOOLC_1 | 20 | 11 | 16 | 0.907 | 17 | 1.022 | 11 | 4.772 |
| ADD_BOOLC_2 | 15 | 16 | 9 | 0.239 | 8 | 0.220 | 3 | 0.220 |
| ADD_BOOLC_3 | 14 | 16 | 4 | 0.169 | 6 | 0.106 | 1 | 0.178 |
| ADD_BOOLC_4 | 15 | 5 | 16 | 0.138 | 12 | 0.097 | 11 | 0.170 |
| ADD_MCAC_0 | 4 | 7 | 1 | 0.152 | 2 | 0.106 | 1 | 0.137 |
| ADD_MCAC_1 | 5 | 5 | 9 | 0.145 | 9 | 0.060 | 8 | 0.181 |
| ADD_MCAC_2 | 9 | 3 | 2,768 | 171.043 | 2,617 | 27.030 | 1,481 | 169.651 |
| ADD_MCAC_3 | 9 | 3 | 2,139 | 26.396 | 2,108 | 8.994 | 1,556 | 216.583 |
| ADD_MCAC_4 | 14 | 3 | 582 | 0.869 | 587 | 0.597 | 580 | 53.548 |

shown in the figure, pMEDICI can save till 99% of time w.r.t. the original MEDICI.

However, we have observed that the saving is in general higher when no ordering optimization is used, except for 8 benchmarks (see Sect. IV-B for further details).

In terms of size, not surprisingly MEDICI always performs better or equally to pMEDICI. This was expected since considering multiple test contexts at once does not ensure to always choose the best context in which a tuple can be inserted. In fact, it is possible that the best context is already occupied managing another tuple and, thus, a different context is chosen by the algorithm. On the contrary, MEDICI manages only one tuple at a time and this assures to always choose the best MDD in which inserting the new tuple, minimizing the test suite size.

### *B. Impact of the ordering optimization on the test suite*

As shown in Tab. I and Tab. II, we have found that ordering the test contexts is not always an efficient optimization, especially in terms of generation time.

This is evident especially when models with a lot of parameters or constraints are analyzed since the time required for the ordering process overpasses that of the test generation algorithm (see, for example, the `ADD_MCAC_2` benchmark). In general, we can state that for the majority of the models, the performances in terms of time when the ordering optimization is activated are always worst or equal to those obtained when the optimization is not activated.

However, we have observed that in some models, the ordering optimization allows to reduce the test suite size up to

10%, no matter what is the complexity of the model. This is because if the tests contexts are ordered, the algorithm chooses in which test context insert every new tuple for minimizing the size and avoids creating useless test contexts. However, this heuristic can be less effective if multithread is enabled, since the "best" test context in which insert a new tuple may be locked by other threads.

Thus, in our opinion, the user should choose whether to use the ordering optimization or not depending on the complexity of the analyzed model. Moreover, it should be used when smaller test suites are preferable and avoided when the generation time is more important than the size.

### C. Impact of number of threads on the generation time

In our experiments, we have investigated the impact of the number of threads on the test suite generation time and observed that the gain in terms of generation time depends on the complexity of the combinatorial problem. In particular, our tests highlight that for models with fewer constraints and parameters, increasing the number of threads is counterproductive or, sometimes, meaningless. In fact, for those models, the overhead implied by the construction and coordination between threads overpass the time required for the actual test generation. On the other hand, when it comes to models with a lot of constraints or parameters, the advantages of using more threads overcome the thread management overhead. For example, here we report in Fig. 6 and Fig. 7 how the test generation time changes for the models which require the highest generation time (`ADD_MCAC_2` and `MCA_1`, respectively with and without constraints) and the lowest one (`ADD_BOOLC_4` and `UNIFORM_BOOLEAN_1`, respectively with and without constraints), when the ordering optimization is not active. In particular, the plots of Fig. 6 show that for the most complex models (`ADD_MCAC_2` and `MCA_1`), increasing the number of threads from 2 to 6 leads to a decrease in generation time, that in the case of the `ADD_MCAC_2` passes from 30 to 27 seconds. However, as well known, further increasing the number of threads is not always the best solution and, over a certain limit, having more threads means introducing a lot of coordination overhead and does not yield any advantages. On the other hand, for the simplest models (`ADD_BOOLC_4` and `UNIFORM_BOOLEAN_1`), introducing new threads only gives minor improvements (Fig. 7).

### D. Impact of number of threads on the test suite size

From our experiments, we have noticed that, as for the generation time, the impact of the number of threads on the test suite size varies, but generally by increasing the number of threads the test suite gets smaller. In particular, our experiments highlight that for models with fewer constraints and parameters, increasing the number of threads can be meaningless (or even counterproductive), since is some cases a thread starts when all the other threads have already completed the computation, and thus, it does not contribute in decreasing or increasing the size (see for instance `UNIFORM_BOOLEAN_1`). On the contrary, for complex models, increasing the number

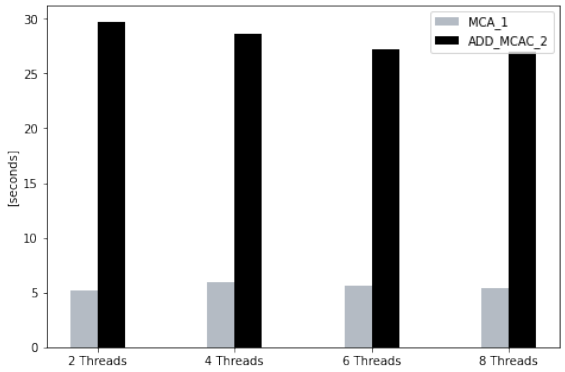


Fig. 6. Test suite generation time for models with the highest generation time

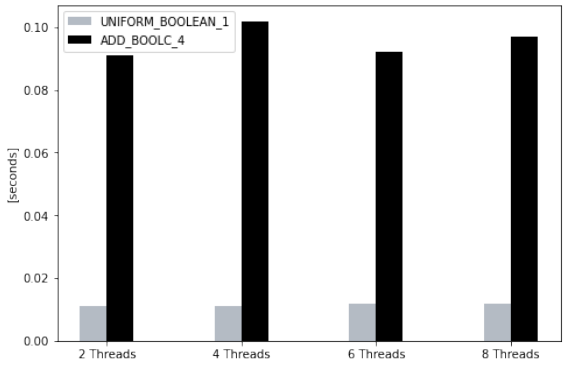


Fig. 7. Test suite generation time for models with the lowest generation time

of threads contributes in decreasing the test suite size. For example, here we report in Fig. 8 and Fig. 9 how the test generation time changes for the models which require the highest generation time (`ADD_MCAC_2` and `MCA_1`, respectively with and without constraints) and the lowest one (`ADD_BOOLC_4` and `UNIFORM_BOOLEAN_1`, respectively with and without constraints), when the ordering optimization is not active.

This is contrary w.r.t. what we expected: we initially thought that by increasing the number of threads, the test suite would have become bigger. Instead, the experiments show that the higher is the number of threads, the smaller are the test suites. We plan to investigate the reasons as future work.

## V. Related work

This paper presents the pMEDICI tool which exploits the MDDs and multithreading for generating combinatorial test cases. The core functionalities of the tool are inherited from the tool MEDICI [7], even if it is a single-threaded test generator.

Nowadays, most of the available works on Combinatorial Interaction Testing exploit sequential algorithms for solving the problem such as AETG [5], IPO, IRPS [14], DDA [2], IPOG [10], and G2Way [8]. Some attempts to develop a multithread combinatorial tests generator have been done, but the majority of them are able to deal only with combinatorial models without constraints, and mainly exploit the IPOG

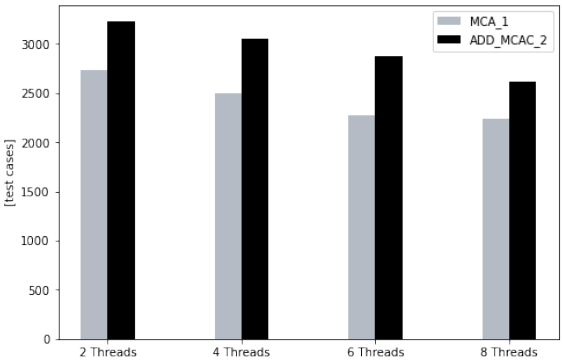


Fig. 8. Test suite size for models with the highest generation time

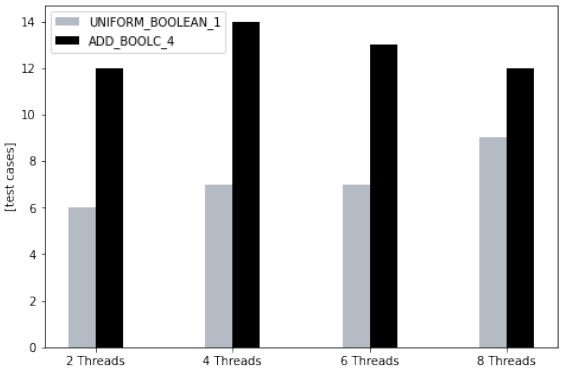


Fig. 9. Test suite size for models with the lowest generation time

algorithm. Examples are GMIPOG [15] and MC-MIPOG [13]. The former distributes the test sets generation process into the grid by partitioning the work based on parameter value, while the latter adopts a novel approach by removing control and data dependency to permit the harnessing of multicore systems. Other approaches, such as [12], exploit the search method with map and reduce framework for generating test suites, or a parallel tree structure, such as [9], or as in [11] which utilizes vast amount of parallelism provided by graphics processing units (GPUs). Other attempts to parallelize generation algorithms, even on the larges scale of grid computing are presented in [3], [4]. Although very promising, these algorithms require a powerful infrastructure while multi-threading like that implemented by pMEDICI is more accessible.

## VI. Conclusions

In this paper, we have presented pMEDICI which is a combinatorial test suite generator exploiting MDDs and the multithreading capabilities of recent PCs in order to reduce the time required for the generation of a test suite. The proposed tool is still a prototype but has proved to be very effective in terms of generation time, both for models with and without constraints. However, the size of the generated test suite is almost never the lowest. Thus, as future work, we are planning to introduce some heuristics which allow reducing the size without requiring too much additional time. Moreover, pMEDICI is not able to deal with constraints containing arithmetical operations. Although they are quite rare in models published in the literature, we plan to extend our tool in order to support them. All the analyses conducted in this paper are based on pairwise testing, Nevertheless, our experiments show that our tool performs well also with t-wise coverage, but further experiments are needed. In general, we believe that the approach we devised for pMEDICI, based on the parallelization and on the use of several test contexts, is promising and can be further extended in order to overpass almost all the limitations that this first prototype version of our tool still has.

## References

[1] 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.

[2] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, 17(3):159–182, 2007.

[3] A. Calvagna, A. Gargantini, and E. Tramontana. Building t-wise combinatorial interaction test suites by means of grid computing. In *2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises*. IEEE, jun 2009.

[4] A. Calvagna, G. Pappalardo, and E. Tramontana. A novel approach to effective parallel computing of t-wise covering arrays. In *2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, jun 2012.

[5] 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.

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

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

[8] M. F. Klaib, K. Z. Zamli, N. A. M. Isa, M. I. Younis, and R. Abdullah. G2way a backtracking strategy for pairwise test data generation. In *2008 15th Asia-Pacific Software Engineering Conference*. IEEE, 2008.

[9] M. F. J. Klaib, S. Muthuraman, N. Ahmad, and R. Sidek. A parallel tree based strategy for test data generation and cost calculation for pairwise combinatorial interaction testing. In *Networked Digital Technologies*, pages 509–522. Springer Berlin Heidelberg, 2010.

[10] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, mar 2007.

[11] H. Mercan, C. Yilmaz, and K. Kaya. CHiP: A configurable hybrid parallel covering array constructor. *IEEE Transactions on Software Engineering*, 45(12):1270–1291, dec 2019.

[12] Z. H. C. Soh, S. A. C. Abdullah, K. Z. Zamli, and M. I. Younis. Distributed t-way test suite data generation using exhaustive search method with map and reduce framework. In *2010 IEEE Symposium on Industrial Electronics and Applications (ISIEA)*. IEEE, oct 2010.

[13] M. I. Younis and K. Z. Zamli. MC-MIPOG: A parallel t-way test generation strategy for multicore systems. *ETRI Journal*, 32(1):73–83, feb 2010.

[14] M. I. Younis, K. Z. Zamli, and N. A. M. Isa. IRPS – an efficient test data generation strategy for pairwise testing. In *Lecture Notes in Computer Science*, pages 493–500. Springer Berlin Heidelberg, 2008.

[15] M. I. Younis, K. Z. Zamli, and N. A. M. Isa. A strategy for grid based t-way test data generation. In *2008 First International Conference on Distributed Framework and Applications*. IEEE, oct 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, 2013.