Efficient Combinatorial Test Generation based on Multivalued Decision Diagrams

Angelo Gargantini and Paolo Vavassori

Dip. di Ingegneria, Università di Bergamo, Italy
{angelo.gargantini,paolo.vavassori}@unibg.it

Abstract. Combinatorial interaction testing (CIT) is an emerging testing technique that has proved to be effective in finding faults due to the interaction among inputs. Efficient test generation for CIT is still an open problem especially when applied to real models having meaningful size and containing many constraints among inputs. In this paper we present a novel technique for the automatic generation of compact test suites starting from models containing constraints given in general form. It is based on the use of Multivalued Decision Diagrams (MDDs) which prove to be suitable to efficiently support CIT. We devise and experiment several optimizations including a novel variation of the classical greedy policy normally used in similar algorithms. The results of a thorough comparison with other similar techniques are presented and show that our approach can provide several advantages in terms of applicability, test suite size, generation time, and cost.

1 Introduction

Combinatorial Interaction Testing (CIT) helps tester to find defects due to the interaction of components or inputs. It is based on the assumption that faults are generally caused by *interactions* among parameters. CIT tests the interaction in a systematic way. For instance, *pairwise* testing requires that every pair of parameter value be tested at least once. It can be generalized by the *t-way* testing. CIT has been proved to be very effective in finding faults [20].

A major problem in CIT is the generation of compact test suites, especially when the cost of executing each test case is high. Suitable tools can produce very compact test suites. For instance [20], a manufacturing automation system that has 20 controls, each with 10 possible settings –a total of 10^{20} combinations – can be tested by a test suite for the pairwise testing with only 180 tests in it. Applying CIT to highly configurable software systems is complicated by the fact that, in many such systems, the parameters are rarely independent from each other. There exist constraints that model dependencies among parameters that render certain combinations invalid or some combinations mandatory [12]. The presence of constraints increases the complexity of the test generation task: if constraints on the input domain are to be taken into account, even finding a single test or configuration that satisfies the constraints is NP-complete [5], since it can be reduced in the most general case to a satisfiability problem. Several

works, like this, target explicitly the test generation for CIT in the presence of constraints, CCIT in brief. In this paper we focus on reaching a good trade-off between the size of the generated test-suite and its time of generation.

Our algorithm is a classical greedy algorithm which produces a test at the time [7]. When building a single test, it chooses an *optimal* parameter and assigns an *optimal* value to it until a test is complete. However, we advance with respect to the state of the art by adopting the following original approaches:

– We employ a data structure, called Multivalued Decision Diagram (MDD), which is particularly suitable to combinatorial problems in order to represent inputs, their domains, and constraints over those inputs; MDDs offer several advantages w.r.t. the classical Binary Decision Diagrams.

– We soften the classical greedy algorithm by reducing the importance of the number of tuples covered by the test currently built, by weighting parameters and tuples depending on the constraints in order to reduce the test suite size;

The paper is organized as follows. In Sect. 2 we present some introductory material about constrained combinatorial interaction testing, about the framework called CitLab, and about MDDs. Sect. 3 shows how MDDs are suitable to efficiently represent several aspects of CIT (models, tuples, tests, and constraints). In Sect. 4 we present our algorithm and several optimizations. Experiments are reported in Sect. 5. Section 6 presents relevant related work. Future works are discussed in Sect. 7, which concludes the paper.

2 Background

2.1 Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) systematically explores t-way feature interactions inside a given system, by effectively combining all t-tuples of parameter assignments in the smallest possible number of test cases. This allows to budget-constraint the costs of testing while still having a testing process driven by an effective coverage metric [19]. The most commonly applied combinatorial testing technique is *pairwise* testing, which consists in applying a test suite covering all *pairs* of input values (each pair in at least one test case). Many CIT tools (see [24] for an up to date listing) and techniques have already been developed [16,19] and are currently applied in practice [4,18].

Combinatorial testing can be applied to a wide variety of problems: highly configurable software systems, software product lines which define a family of software, hardware systems, and so on. As an example, Listing 1 reports the input domain model of a simple smart-phone product line using the CitLab [10]. The model contains three parameters: the display can have 16 or 8 million colors or be in black and white (BW), the frontCamera can have 1 or 2 megapixels (1MP and 2MP) or not be present (NOC). The phone can also have an emailViewer. We will use this simple example throughout the paper to explain our approach. While testing of all the possible configurations for the phone would require $3 \cdot 3 \cdot 2 = 18$ tests, pairwise coverage can be obtained by a test suite containing only 9 tests.

Listing 1: A mobile phone example

```
Model phone
Parameters:
  Enumerative display { 16MC 8MC BW };
  Enumerative frontCamera { 2MP 1MP NOC };
  Boolean emailViewer;
end
Constraints: # emailViever => display != BW # end
```

In most configurable systems, constraints or dependencies exist between parameters. Constraints were first described as being important to combinatorial testing in [11] and were introduced in the AETG system. In our approach, tests that do not satisfy the constraints are considered *invalid* and do not need to be produced. However, the generation of tests considering constraints is more challenging than the generation without them, and several test generation techniques still do not support constraints, at least not in a direct manner.

In CitLab testers are allowed to specify constraints in a general form. For instance, the constraint that a phone with an email viewer cannot have a black and white display can be modeled as shown in Listing 1.

2.2 Multivalued Decision Diagram

A decision diagram is a graph that represents a function $f : \mathcal{D} \rightarrow B$ where $\mathcal{D} = \mathcal{D}_1 \times \ldots \times \ldots \mathcal{D}_n$ and B is the Boolean domain, i.e., $B = \{\mathbf{F}, \mathbf{T}\}$. A decision diagram is used to evaluate the truth value of f when applied to the variables $x_1, \ldots, x_n$. If all the domains $\mathcal{D}_i$ are binary, then we use Binary Decision Diagrams (BDDs) to represent Boolean functions. BDDs are widely used within the domain of system design verification. Multi-Valued Decision Diagrams (MDD) extend BDDs by allowing every variable to have a different domain with different size. A MDD is a directed acyclic graph used to encode a function f. The graph has only two terminal nodes each labeled $\mathbf{F}$ or $\mathbf{T}$. Every non-terminal node is labeled by an input variable x_i and has $|D_i|$ outgoing labeled edges; one corresponding to each value. The diagram is ordered if the variables adhere to a single ordering on every path in the graph, and no variable appears more than once on any path from the root to a terminal node. An MDD can represent the values in $\mathcal{D}$ that are selected by f: if the values $x_1, \ldots, x_n$ for the variables in $\mathcal{D}$ are selected by f, then $f(x_1, \ldots, x_n) = \mathbf{T}$, otherwise $f(x_1, \ldots, x_n) = \mathbf{F}$.

Typical operations among MDDs include *unary* operations like complement and cardinality, and *binary* operations like union, intersection, and difference.

MDD operations can be mapped to logic operation between the Boolean functions represented by an MDD. Given an MDD m with function f, its complement $m^{\complement}$ represents the function $\neg f$. The union between two MDDs $m_1 \sqcup m_2$ represents the function $f_1 \vee f_2$. The intersection between two MDDs $m_1 \sqcap m_2$ represents the function $f_1 \wedge f_2$. Given the MDD m, its cardinality $|m|$ is the

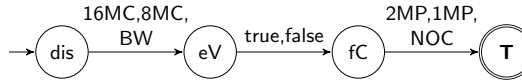


Fig. 1: MDD for the combinatorial problem of Listing 1

number of all the possible paths to the terminal node **T**. The cardinality can be used to check consistency among Boolean functions: if f_1 and f_2 are inconsistent, i.e. $f_1(x) \neq f_2(x)$ for any x, the intersection between their MDDs is empty.

MDDs can represent Boolean logic functions using less memory and shorter path then BDDs. From a theoretical point of view, Nagayama [22] demonstrated that the amount of memory used by mapping Boolean function with Boolean variables to heterogeneous MDD is lesser than using OBDD directly. This seems to suggest that MDDs are the preferred data structure when the domains are not simple Boolean values.

In order to achieve this performance improvement over BDDs, it is very important the use of techniques that can reduce the size of MDDs. To our knowledge, Meddly [3] is the only opensource C/C++ library that natively supports these DDs. According to our opinion, Meddly native support for MDDs and their variants, along with its performance makes it a good candidate for applications in areas where these DDs make sense.

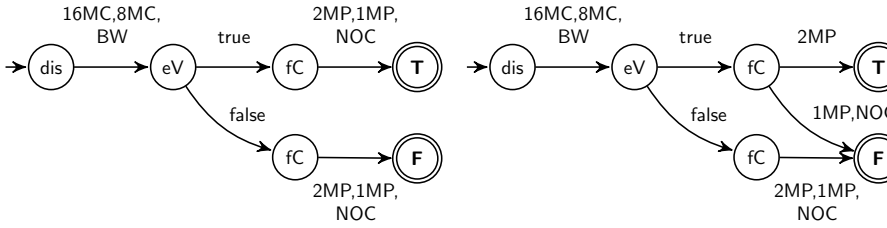
3 Using MDD for CCIT

If one ignores the constraints, a combinatorial model with n parameters each with cardinality p_i can be very easily represented by an MDD that has n non-terminal nodes labeled by the name of every parameter and each node for parameter P_i has p_i outgoing labeled edges to the node for P_{i+1} for $i < n$ and to the **T** terminal node for P_n. We call this MDD M_{TS}. For instance, the MDD in Fig. 1 represents the M_{TS} for the phone given in Listing 1. In the following figures, edges sharing the same starting and final node are shown with a unique arch and the list of labels. Every path from root to the terminal **T** is a syntactically correct configuration. The M_{TS} represents all the tests, i.e., all the possible paths from the start to the terminal node. The cardinality of M_{TS} is equal to $\prod_{i=1}^{n} p_i$ which is equal to the total number of possible tests.

The equality formula that associates parameter P_i to one of its values v, i.e., the assignment $P_i = v$ can be easily represented by the following function.

$$f(p_1, \ldots, p_n) = \begin{cases} \mathbf{T} & \text{if } P_i = v \\ \mathbf{F} & \text{if } P_i \neq v \end{cases}$$

Such function can be represented by an MDD in which all the paths, traversing the edge outgoing the node P_i with label v, terminate to the terminal **T** while all the other ones terminate in **F**. For instance, the equality $\mathsf{eV} = \mathsf{true}$ is shown in Fig. 2a. A similar MDD representation can be given for a tuple assigning



(a) Single value equality ($\mathsf{eV} = \mathsf{true}$) **(b)** Tuple ($\mathsf{eV} = \mathsf{true}, \mathsf{fC} = \mathsf{2MP}$)

Fig. 2: Representation by MDDs of assignments and tuples

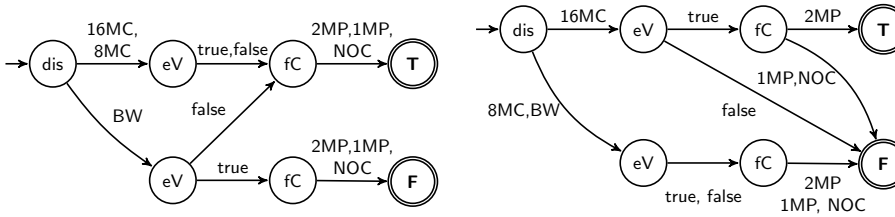
values to a list of parameters. A path terminates to the node **T** if and only if it contains an assignment contained also in the tuple. For instance, the tuple ($\mathsf{eV} = \mathsf{true}, \mathsf{fC} = \mathsf{2MP}$) is shown in Fig. 2b.

In most configurable systems, constraints or dependencies exist between parameters. Since we assume that the constraints corresponding to a CIT problem can be described by propositional logic with equality, we can describe every model constraint c_i using a Boolean general formula containing operators $\neg, \vee, \wedge$ over equalities among parameters and their values. Every constraint can be represented by an MDD modeling its truth function: it can be built using the representation of equality formulas proposed above and the operations between MDDs presented in Sect. 2.2.

In order to include the constraints in the MDD M_{TS} representing the unconstrained model, we can use the operations between MDDs. Let M_{TS} be the MDD representing the model and the whole test set from all the possible combinations. The conjunction of M_{TS} with all the constraints c_i restricts the set of satisfying interpretations of the function associated to M_{TS} such that it contains exactly those interpretations that correspond to valid test cases. Let m_{c_i} be the MDD for the constraint c_i, and the MDD M_{VS} be defined by the following formula: $M_{VS} = \prod_{i=1}^{n} m_{c_i} \sqcap M_{TS}$.

Integrating the constraints c_i into the MDD M_{TS} in order to obtain the MDD M_{VS}, changes the M_{TS} original topology by making one or more paths from valid to not-valid. In the original MDD there are n levels and n not-terminal nodes, where n is the number of parameter. In order to model not-valid paths it is necessary to duplicate some nodes. The MDD M_{VS} preserves the number n of levels but has some more not-terminal nodes. The M_{VS} represents all the *valid tests*, i.e. all the possible paths from the start to the terminal **T** node.

An example of the MDD M_{VS} representing the model and the constraint for the phone problem is shown in Fig. 3a. M_{VS} can be used to identify valid tests. For instance, the combination ($\mathsf{dis} = \mathsf{BW}, \mathsf{eV} = \mathsf{true}$) is not valid, regardless of the value of fC, as expected, since the requirements prohibit a BW display with the emailViewer. On the contrary, the test ($\mathsf{dis} = \mathsf{16MP}, \mathsf{eV} = \mathsf{true}, \mathsf{fC} = \mathsf{2MP}$) is a valid test, as shown by the corresponding path leading to the terminal **T** node in the MDD. An MDD with cardinality 1, i.e. with only one path to the **T**



(a) M_{VS} for the phone with the constraint

(b) An MDD representing one test case

Fig. 3: Representation by MDDs of models with constraints and test cases

terminal node, represents one valid test. The example shown in Fig. 3b identifies the test ($\mathsf{dis} = \mathsf{16MP}$, $\mathsf{eV} = \mathsf{true}, \mathsf{fC} = \mathsf{2MP}$).

4 An MDD-based algorithm for CCIT

We have devised an automatic algorithm for the generation of combinatorial test suites based on the use of MDDs. The algorithm takes as input the MDD M_{VS} representing the intersection between the model domain and the constraints, and produces as output the desired test suite R. It builds one test at the time until all the testing requirements are achieved. When building a single test, it proceeds in a greedy manner: it chooses one *optimal* parameter, which is not already set in the test, and its *optimal* value, according to our weighting criteria, and it adds this assignment to the test to be built. In the following we explain in details the algorithm that is reported in Alg. 1.

Firstly, we populate a list of tuples T_{TC} including all the combinations to cover based on a given coverage criterion C, usually *t-wise* coverage. We plan to use MDDs also to represent set of tuples like in [26]. Some tuples may be infeasible because of the constraints. In order to filter all the valid tuples, we use MDDs as well: the function *feasibleTuples* returns all the tuples required by the criterion C that have a non-empty intersection with the MDD M_{VS}.

We then start the iteration part where we generate, for each iteration, a test case M_{nc} represented by an MDD with final cardinality equal to 1. At the end of each iteration, we update T_{TC} removing the tuples covered by the generated M_{nc} until T_{TC} is empty.

In the single iteration we initialize M_{nc} to the valid set M_{VS}, we then sort all the parameters (*sortParamList*) by simply counting for every parameter p the number of tuples in T_{TC} that contain p. We then start assigning every P_i to the best value for it, by taking the *value* producing an assignment that is compatible with M_{nc} and that maximizes the coverage of tuples in T_{TC}.

This basic algorithm is a classical greedy algorithm that generates a test at the time and tries to cover as many uncovered tuples as possible. It can be improved in several directions, as explained in the following sections.

Algorithm 1 Generation of the test suite R

Input: M_{VS}: MDD for the model with the constraints
Output: R: set of MDDs representing the test set

```
T_TC ← feasibleTuples(M_VS)
R ← ∅
while T_TC ≠ ∅ do                                  ▷ Build single test M_nc
    M_nc ← M_VS
    P ← sortParamList(T_TC)
    for all P_i ∈ P do                             ▷ Fix every parameter in P
        value ← chooseBest(P_i, M_nc, T_TC)
        M_nc ← M_nc ⊓ P_i = value
        if |M_nc| = 1 then break end if
    end for
    T_TC ← removeCoveredTuples(M_nc)
    R ← R ∪ M_nc
end while
```

4.1 Optimization: Weighting compatibility

Although most greedy algorithms consider only the number of remaining tuples that will be covered in order to determine the best choice [7], it is well known that such greedy policy can lead to bigger test suites, even for unconstrained models[1]. In the presence of constraints, this greedy policy can be even more inefficient since it tends to leave at the end all the tuples that are "difficult" to cover, because the constraints limit the number of valid test cases that can cover them. In this way, the last generated tests cover only a few tuples not covered yet, leading to bigger test suites.

We propose to *weight* every tuple depending on its compatibility with respect to the other tuples not covered yet considering also the constraints. Heavy tuples are more difficult to cover and they should be fixed sooner than light tuples. To weight tuples, we introduce a dynamic function *weigth* that measures the weight of every tuple and we modify the Alg. 1 by calling the function in Alg. 2 that assigns the weights before ordering the parameters. We modify the functions *sortParamList* and *chooseBest* accordingly in order to consider tuple weights.

The function ASSIGNWEIGHT increases the **weight** (initially set to 0) for all the tuple pairs (T_i, T_J) with T_i and T_j in T_{TC} that are mutually exclusive by considering also the constraints. Checking if two tuples are compatible can be performed by using the usual intersection operator among MDDs. For instance the tuples $(\mathsf{dis} = \mathsf{BW}, \mathsf{fC} = \mathsf{2MP})$ and $(\mathsf{fC} = \mathsf{2MP}, \mathsf{eV} = \mathsf{true})$ would have their *weight* increased because they are incompatible due to the constraints and this can be easily computed using the MDD of Fig. 3a.

Although we can rely on the efficiency of MDDs for the computation of weights, Alg. 2 has complexity $N^2/2$ where N is the number of remaining tuples

[1] Bryce and Colbourn report in [6] the example in which a simple greedy algorithm provides a solution of 1,222 tests. Relaxing the greedy behavior or other algorithms can provide much smaller test suites till 910 tests

Algorithm 2 Computation of weights

```
function assignWeight(T_TC, M_VS)
    for all T ∈ T_TC do weight(T) ← 0 end for
    for all (T_i, T_j) ∈ T_TC × T_TC with i < j do
        if M_VS ⊓ T_i ⊓ T_j = ∅ then
            weight(T_i) ← weight(T_i) + 1
            weight(T_j) ← weight(T_j) + 1
        end if
    end for
end function
```

Algorithm 3 Approximate and faster computation of weights

```
function assignWeightFromParams(T_TC)
    for all T ∈ T_TC do weight(T) ← 0 end for
    for all P_i and T_i ∈ T_TC with P_i ∈ T_i do
        weight(P_i) ← weight(P_i) + 1
    end for
    for all T_i ∈ T_TC and P_i ∈ T_i do
        weight(T_i) ← weight(T_i) + weight(P_i)
    end for
end function
```

to cover (T_{TC}) and this can increase the computation time. For this reason, we define a simplified algorithm (Alg. 3) that is less precise but it is much faster than Alg. 2. This algorithm 3 first assigns a weight to every parameter depending on the number of remaining tuples to cover (T_{TC}) that contain it. Then, every tuple gets a weight that is the sum of the weights of the parameters in it. It does not consider the model and its constraints (M_{VS}), it does not need to perform any operation among MDDs, and for this reason is much faster.

We devised the following policy. If the number of tuples to be covered ($|T_{TC}|$) is greater than a **threshold**, Alg. 3 performs the weighting otherwise, the more precise Alg. 2 is used.

4.2 Optimization: Repetitions

Our algorithm produces non deterministic results, since when ordering the parameters and when identifying the best value for the chosen parameter, it may occur that two or more choices are equally valid. In this case the algorithm randomly chooses one possibility. The choice may affect the behavior of the test generation only much later (typically only in the last steps). One possibility is to repeat with a different random seed the entire algorithm (except the evaluation of tuple feasibility) in order to see if by chance a better solution is found. We call this optimization *repetition*, as defined in [7]. We manage the repetition policy by setting the following three parameters $repeat_{\min}$, $repeat_{\max}$, and $repeat_{\text{better}}$. When repetition is activated, the algorithm generates at least $repeat_{\min}$ times a

Table 1: Characteristics of the CCIT benchmarks.

	#Variable	#Constraints	Domain size	#Valid configurations	Ratio[3]
Minimum	3.00	0	8.00	1.00	2.44×10^{-29}
Maximum	259.00	388	$9.26\times10^{+77}$	$2.44\times10^{+62}$	1.00
Mean	44.85	27.46	$1.16\times10^{+76}$	$5.89\times10^{+60}$	0.25
Median	15.00	15	$8.35\times10^{+04}$	$2.60\times10^{+04}$	7.86×10^{-02}

new test suite. It keeps generating new test suites unless for $repeat_{\text{better}}$ the test suite is not smaller than the best found so far. In any case no more than $repeat_{\max}$ generation runs will be executed. The smallest test suite found is returned.

5 Experiments

We have implemented the algorithm presented in the previous section in a prototype tool called MEDICI (MultivaluEd Decision diagrams for Combinatorial Interaction testing). We have integrated MEDICI in CITLAB, an extensible framework for combinatorial testing [10]. MEDICI is written in C++ and is based on Meddly [3] for the MDDs. It has been embedded in CITLAB and it is freely available[2]. CITLAB simply exports the necessary input file for MEDICI and executes it. Note that MEDICI accepts constraints in general form and thanks to the fact that it uses MDDs, it avoids the time-consuming conversion to CNF .

As benchmarks for CCIT problems we have gathered 117 models with constraints taken from the literature (Casa [13,15,12], FoCuS [26], ACTS [1], and IPO-S [9]) and from SPLOT SPLs repository, and used (in subsets) also by many other papers. The benchmarks can be found on the CITLAB web site and can be used for further comparisons. For the sake of brevity, we show, in Tab. 1, only some useful statistical summary about the models. We run the experiments on a PC with two Intel(R) Xeon(R) CPU E5-2630 @ 2.30GHz and 64 GByte of RAM. We exploit the multi-core architecture by running 20 threads in parallel and we run all the experiments with the pairwise coverage and 50 runs.

Let $\overline{size}_m$ be the average of the test suite size for model m over all the runs and $\overline{time}_m$ be the average of the time for model m, we introduce *size* and *time* defined as: $size = \Sigma\overline{size}_m$ which is the sum of the averages of the test suite sizes and $time = \Sigma\overline{time}_m$ which is the sum of the averages of the executions times (in seconds). We will use *size* and *time* as performance indexes.

Optimal threshold value. We perform an experiment in order to discover the impact of the threshold introduced in Sect. 4.1 over the test generation size

[2] CitLab and its MEDICI plugin can be found at `http://code.google.com/a/eclipselabs.org/p/citlab/`

[3] Ratio=(#Valid configurations / Domain Size)

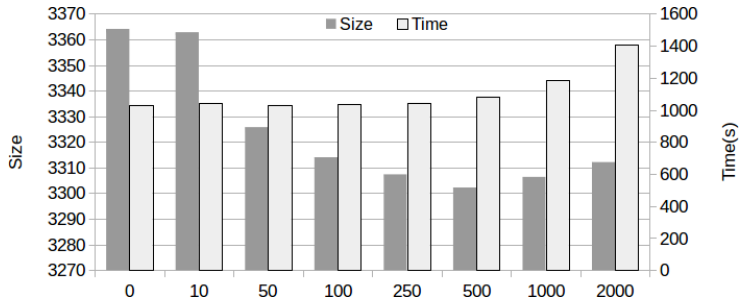


Fig. 4: Test suite *size* and *time* depending on the threshold

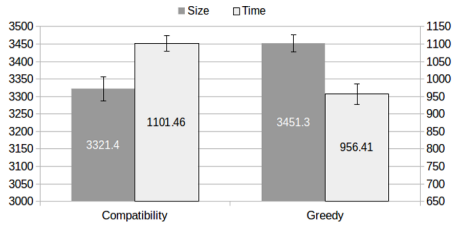
and time (with 1 repetition). Fig. 4 reports how the test suite size and time changes depending on the value of the threshold[4]. As the graph shows, the test suite size has a minimum for a threshold around 500, while it becomes sensibly greater with thresholds smaller than 250. The time becomes significantly greater for threshold greater than 250. From now on, we chose as optimal threshold the default value of 250.

Using compatibility We experiment the efficacy of the use of the compatibility and weights in order to choose the optimal parameter and value w.r.t. the classical greedy algorithm as explained in Sect. 4.1 by performing a comparison with a version of MEDICI that avoids this optimization and uses a greedy algorithm over the number of covered combinations. The results are shown in the chart of Fig. 5a.

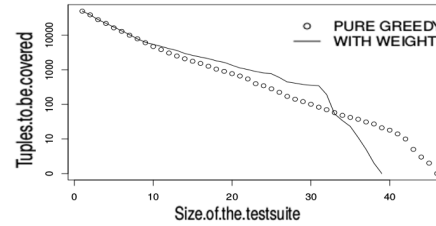
We observe that using the compatibility leads to smaller test suites (*size* is around 4% smaller on average) with an increase of the time (*time*) of around 15%. Using the proposed technique slows the rate in which uncovered tuples are covered but reduces the final test suite size. For instance, Fig. 5b reports the size of still uncovered tuples (y-axis) while generating tests for one model (the number of tests already generated is on the x-axis). By maximizing the coverage of tuples (dotted line), the test generation covers more tuples at the beginning but at the end it needs new tests to cover the residual uncovered tuples. By using compatibility and by weighting the tuples (continuous line), the algorithm covers fewer tuples at the beginning but at the end all the residual tuples are easily covered with few tests. The figure shows that the problem of finding minimal test suites is not easily solvable by using pure greedy algorithms, since only near the end our proposed approach outperforms the classical greedy approach.

Number of repetitions Regarding the number of repetitions (options $repeat_{\min}$, $repeat_{\max}$, $repeat_{\text{better}}$ introduced Sect. 4.2), the situation is more clear, since the use of these options is purely incremental and increasing the number of tries will

[4] Threshold values are in the set {0,10,50,100,250,500,1000,2000}.



(a) Greedy vs Compatibility comparison with optimization of Sect. 4.1



(b) Tuple coverage rate for *b_12* with optimization of Sect. 4.1

Fig. 5: Greedy vs Compatibility comparison

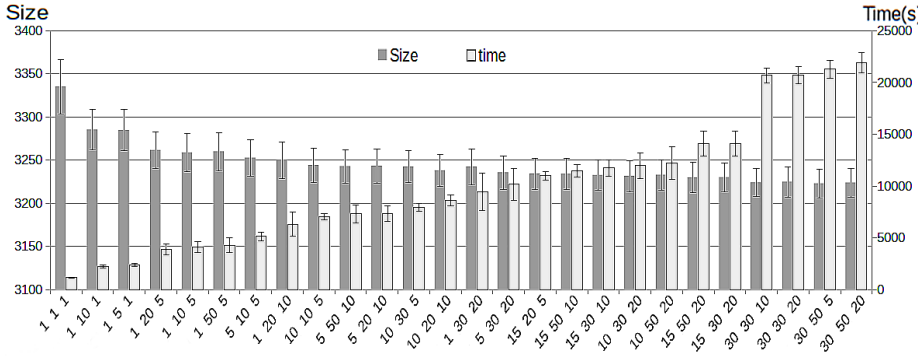


Fig. 6: Test suite *size* and *time* depending on the repetitions settings ($repeat_{\min}$ $repeat_{\max}$ $repeat_{\text{better}}$).

always increase the time and decrease (or keep equal) the number of tests. The choice of the optimal values for these options, is however a typical multi-objective optimization, in which we try to optimize the two conflicting objectives of a small test suite size *and* a small generation time.

We test for the *repeat* options the values {1, 5, 10, 15, 20, 30, 50} which give rise to 27 valid configurations. The data for the execution of all the configurations is shown in Fig. 6 (and later in Tab. 2). The graph confirms that the two objectives of minimizing both *size* and *time* are conflicting: it is possible to obtain smaller test suite but at the expense of the test generation time. Our technique allows the tester to decide of spending more time in order to have smaller test suites. From all the configurations, we select one with ($repeat_{\min}$, $repeat_{\max}$, $repeat_{\text{better}}$) equal to (10,30,5) which represents a good compromise between time and speed and it can be considered as a good candidate for a default use of MEDICI. From now on, we will use this version for further comparison.

Comparison with other tools in CitLab We perform a comparison of MEDICI with the other external tools supported by CITLAB, namely ACTS [1,21] and CASA [13,15]. ACTS is a tool developed by the NIST and implements several

	size	Σσ S	*time*	Σσ T
ACTS	3387.5	0.5	73.7	2.4
CASA	3185.4	4391.2	14781.2	14305.9
MEDICI	3214.4	6633.5	7871	965.4

Table 2: Comparison with ACTS and CASA

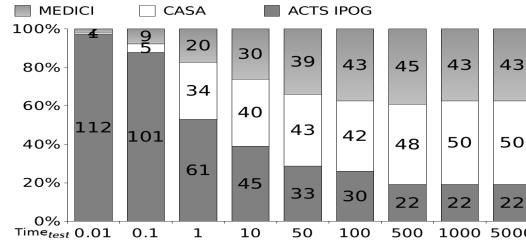


Fig. 7: Number of models that present the minimum cost for each generator for $time_{test}$ from 0.01 to 5000 secs.

variants of the In Parameter Order (IPO) strategy. CASA is a tool developed at the University of Nebraska and it is based on simulated annealing, a well-studied meta-heuristic algorithm. Both support constraints, are freely available, have a large user base, and are very often used in comparison studies. Using CitLab allows us to perform all the experiments in a very controlled environment on the same computer and using exactly the same examples.

Due to the high number of models and experiments, we can give only some cumulative results. Table 2 reports the results of the comparison: we have computed the mean, and the standard deviation (σ) of the size and time (in secs) among all the 115 runs for every model. Besides the sum of averages (*size* and *time*), the table displays the sum of the standard deviations.

Table 2 shows that ACTS is the fastest but it produces also the biggest test suites. ACTS has a deterministic algorithm and hence the standard deviation of its sizes is null. MEDICI is always slower than ACTS but it produces smaller test suites. MEDICI is around 200 times slower than ACTS, but it produces a test suite on the average 5.4% smaller than ACTS. On the other hand, CASA is the slowest of all, but it produces rather small test suites. CASA has a very high standard deviation both in time and in size (running CASA only once may not lead to the best solution of its). MEDICI is faster than CASA and it has a smaller standard deviation. CASA produces a test suite on the average 1% smaller than MEDICI, but its generation time is, on average, double that of MEDICI.

Overall, we can say that MEDICI performances are between CASA and ACTS. To better guide the user in the choice of the best test generator tool, we can roughly estimate the cost of testing (*cost*) as the total time for test generation ($time_{gen}$) plus test execution, which depends on the size of the test suite (*size*) and time necessary to execute every single test ($time_{test}$): $cost = time_{total} = time_{gen} + size \times time_{test}$. Using the data previously computed, we have also calculated the *cost* for each model and for each generator selecting a meaningful set of $time_{test}$. Fig. 7 shows the number of models that present the minimum average *cost* for each generator varying the $time_{test}$. ACTS outperforms both CASA and MEDICI if each test takes on average less than 10 seconds. This is in line with what was found by Garvin et al. [14]. If the time for executing a single

test increases, CASA and MEDICI cost less than ACTS in most models. Even for very costly test execution (e.g. tests that require some human intervention), MEDICI can still compete with CASA in a meaningful number of models.

5.1 Threats to validity

We have identified some threats to validity of the proposed study and we present some countermeasures we have employed. First, the benchmark data may be not representative. We have tried to collect models from many sources: to the best of our knowledge this is one of the biggest benchmark set of constrained combinatorial models used for test generation. The models represent a wide heterogeneous range of real life and academic models. Second, we are aware that our tool, MEDICI, may produce incomplete and incorrect test-suites that allow it to perform better than the other tools. To avoid this, besides performing unit testing we have used CITLAB "validator" [2] that checks that the resulting test suite actually cover all the required tuples (except those infeasible). We use this program for debugging MEDICI. In order to have confidence of the data obtained in the experiments, we have executed 50 runs for every configuration. Using multi-threads allows us to reduce the experimental time, but it may alter the running time, since an ordinary user will generally launch only one execution at the time. However, we believe that the comparison is still fair because we have treated all the generators in the same way.

6 Related work

Combinatorial interaction testing has been an active area of research for many years. In a recent survey [23] Nie and Leung count more than 12 research groups that actively work on CIT area and many other groups and tools are missing in the count. In a previous survey, Grindal et al. [16] presented 16 different combination strategies, covering more than 40 papers. There are several web sites listing tools and approaches (like [24]), and publishing benchmarks and evaluations of tools and algorithms. The most studied area in CIT is the test suite generation, where several research groups continuously challenge existing algorithms and tools in order to provide better approaches in terms of execution times, supported features, and minimality of the produced test suites. Finding an algorithm that improves over the current state of the art has become a hard research task.

There are several families of CIT test generation tools, including bio-inspired, algebraic, logic-based [8], and greedy. In [7], Bryce et al. presented a general framework of *greedy* construction algorithms, in order to study the impact of each *type* of decision on the effectiveness of the resulting heuristic construction process. To this aim, they designed the framework as a nested structure of four decision layers, regarding respectively: (1) the number of instances of the process to be run, (2) the size of the pool of candidate rows from which select each new row, (3) the factor ordering selection criteria and (4) the level ordering selection

criteria. The approach presented in this work fits exactly in the *greedy* category of algorithms modeled by that framework, and it is structured in order to be parametric with respect to the desired number of repetitions and the factor and level ordering strategies. Note that their study concluded that factor ordering is predominant on the resulting test suite size, and that *density*-based level ordering selection criteria was the best performing one out of those tested. In the present work, we explored original ways of redefining the *density* concept. In fact, while Bryce et al. compute it as the expected number of uncovered pairs, we weight tuple compatibility and we order parameters accordingly.

Comparison with BDD-based tools. Regarding the data structure we use, a comparison can be done with works using for CCIT binary decision diagrams (BDDs) which are similar to MDDs. Salecker et al. [25] developed a test set calculation algorithm which uses BDDs as efficient data structure to represent the combinatorial interaction testing problem with constraints. Both their and our approach are based on the modeling of the combinatorial interaction test problem with constraints as a single propositional logic formula. MDDs are a more efficient data structure for CCIT than BDDs: while modeling CCIT using BDDs requires a logic subformula corresponding to all possible alternatives for selecting values from each parameter P_i, MDDs permit to avoid the representation of these subformulas for single parameters; the benefit produced by this technique is the absence of the implicit constraints introduced to represent value selection. Unfortunately the tool presented in [25] is not available and a fair comparison is difficult. For sanity check, we found that on the same models presented in [25], MEDICI without repetitions was able to produce a smaller test suite (486.2 vs 547) and the time required in [25] was 2.3 times the time for MEDICI (687 vs 1606 secs), although our PC is only 1.8 times faster (considering the SPECint of around 42.6 vs 23.5).

Segall et al. [26] developed FoCuS, another BDD-based CCIT tool. In their approach each parameter is represented by one or more binary variables in the BDD. In order to build the BDD of valid tests, they first built for each constraint (called *restriction*) the BDD representing the set of tests allowed by it. A test is valid if and only if it is valid according to all restrictions, therefore the set of valid tests is exactly the intersection of the sets of tests allowed by the restrictions. This is computed by the conjunction of the BDDs representing these sets. Their approach is therefore very similar to ours in terms of problem representation, and we believe that also their approach would benefit from the use of MDDs instead of BDDs. Unfortunately FoCuS is not publicly available. However, again for sanity check, we found that on the same models presented in [26] MEDICI produced smaller test suites (923.5 vs 934) while published data for FoCuS do not include generation time.

7 Future work and Conclusions

We plan to work in several directions in order to improve our approach and the tool. MEDICI (as most other test generation tools, with the notable exception of

ACTS) does not support constraints containing arithmetic expressions. CitLab already adopts the language of propositional logic with equality and arithmetic to express constraints. To be more precise, it uses propositional calculus, enriched by the arithmetic over the integers and enumerative symbols. Although arithmetic expressions are quite rare in models published in the literature, we plan to extend MEDICI in order to deal with the arithmetic constraints expressed in CitLab, since we believe that industrial studies often use them.

Moreover, we have experimented only pairwise coverage, even if MEDICI, ACTS, and CASA support n-wise coverage. Initial experiments shows that MEDICI performs well also with n-wise coverage, but further experiments are needed.

Overall, we believe that the technique presented in this paper and implemented in a prototype tool is a viable alternative to other commonly used tools for tests generation of combinatorial tests in the presence of constraints. Our techniques exploits an efficient data structure (MDDs) that proved to be suitable to represent and solve constrained combinatorial models and promise to scale better than BDDs [17]. We have also devised several optimizations, like *weighting*, that combined with a classic greedy approach allow us to obtain very good results, as demonstrated by our experiments. The use of the framework CitLab has allowed us to define a wide body of benchmarks and to perform the comparison with other tools in a simple and fair way.

Acknowledgments. We thank Dario Corna for his valuable work on the implementation of MEDICI.

References

1. Advanced Combinatorial Testing System (ACTS). http://csrc.nist.gov/groups/SNS/acts/.
2. P. Arcaini, A. Gargantini, and P. Vavassori. Validation of models and tests for constrained combinatorial interaction testing. In *The 3rd International Workshop on Combinatorial Testing (IWCT 2014) In conjunction with International Conference on Software Testing ICSTW*, pages 98–107. IEEE, 2014.
3. J. Babar and A. Miner. Meddly: Multi-terminal and edge-valued decision diagram library. In *7th International Conference on the Quantitative Evaluation of Systems*. IEEE, 2010.
4. R. Brownlie, J.Prowse, and M. Phadke. Robust testing of AT&T PMX/starMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, 1992.
5. R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information & Software Technology*, 48(10):960–970, 2006.
6. R. C. Bryce and C. J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1082–1089, New York, NY, USA, 2007. ACM.
7. R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proc. of the 27th int. conf. on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.

8. A. Calvagna and A. Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010. Springer.
9. A. Calvagna and A. Gargantini. T-wise combinatorial interaction test suites construction based on coverage inheritance. *Software Testing, Verification and Reliability*, 22(7):507–526, 2012.
10. A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial interaction testing with CitLab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tool track*, 2013.
11. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions On Software Engineering*, 23(7):437–444, 1997.
12. M. Cohen, M. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Trans. on*, 34(5):633–650, 2008.
13. Covering Arrays by Simulated Annealing. http://cse.unl.edu/citportal/tools/casa/.
14. B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering*, SSBSE '09, pages 13–22, Washington, DC, USA, 2009. IEEE Computer Society.
15. B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.
16. M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
17. T. Hadzic, E. R. Hansen, and B. OSullivan. On automata, MDDs and BDDs in constraint satisfaction. In *Proceedings of the ECAI 2008 Workshop on Inference Methods based on Graphical Structures of Knowledge*, 2008.
18. D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In I. Society, editor, *27th NASA/IEEE Software Engineering workshop*, pages 91–95, 2002.
19. D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.
20. R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.
21. Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, Sept. 2008.
22. S. Nagayama and T. Sasao. Compact representations of logic functions using heterogeneous MDDs. In *Multiple-Valued Logic, 2003. Proceedings. 33rd International Symposium on*, pages 247–252, 2003.
23. C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.
24. Pairwise web site. http://www.pairwise.org/.
25. E. Salecker, R. Reicherdt, and S. Glesner. Calculating prioritized interaction test sets with constraints using binary decision diagrams. In *Proceedings of IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 278–285. IEEE Computer Society, 2011.
26. I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 254–264, New York, NY, USA, 2011. ACM.