Multi-thread Combinatorial Test Generation with SMT solvers

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

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

Andrea Calvagna
andreamario.calvagna@unict.it
University of Catania, Computer Science Department
Catania, Italy

ABSTRACT

Combinatorial interaction testing (CIT) is a testing technique that has proven to be effective in finding faults due to the interaction among inputs and in reducing the number of test cases, without losing effectiveness. Several tools have been proposed in the literature; however, generating tests remains a challenging task. In this paper, we present a technique for generating combinatorial test suites that uses a multi-thread architecture and exploits Satisfiability Modulo Theory (SMT) solvers to represent model parameters, constraints, and tuples, and it builds from SMT solver contexts the desired test suite. This technique is implemented by the tool KALI. The main advantage of using SMT solvers is that combinatorial models can contain all kinds of parameters and constraints. To evaluate our approach, we tested the impact of several optimizations and compared the performance of KALI with those of some existing tools for test generation. Our experiments confirm that the use of multi-threading is a promising technique but still requires some optimization for being more effective than the already available ones.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

combinatorial testing, multi-thread test generation, satisfiability modulo theories, software testing

ACM Reference Format:
Andrea Bombarda, Angelo Gargantini, and Andrea Calvagna. 2023. Multi-thread Combinatorial Test Generation with SMT solvers. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555776.3577703

1 INTRODUCTION

In recent years, Combinatorial Interaction Testing (CIT) has been widely studied because it has proven to be very effective in reducing the complexity when testing complicated systems with multiple input parameters and in helping testers in finding defects due to the interaction of different inputs. In general, this interaction is checked in such a way that every tuple of parameter values must be tested at least once. However, especially for complex systems with many parameters, generating combinatorial tests is not always an easy task and may require a great number of resources (either time or computational power) to be completed. The complexity of the test generation process is even higher when systems are described with not only many parameters, but also complex constraints among them.

This is the reason why several groups of researchers have proposed many test generators that exploit different strategies to generate combinatorial test suites in the most effective possible way. An emerging approach aimed at reducing the generation time is that of implementing multi-threading algorithms. However, due to the increased complexity, currently available combinatorial multi-thread tools either do not cope with constraints or offer only limited support for them.

In this paper, we present a technique for generating combinatorial test suites on multi-threads architectures and exploiting Satisfiability Modulo Theories for generating constraints-compliant test suites with the desired combinatorial coverage. To test the proposed technique, we have also implemented it in a publicly available prototype tool called KALI.

Thanks to the use of SMT solvers, KALI can deal with all types of constraints among those expressible with the CTWedge grammar [11]: both logical and mathematical, and with all types of parameters. The tool has been evaluated against well known benchmarks provided in an international combinatorial testing tool contest. Based on the results, we also provide answers to several research questions related to combinatorial testing tool development.

The paper is structured as follows. In Sect. 2, we present the general background on CIT and SMT solvers, together with the basic idea underlying the use of this kind of solvers for CIT. Then, Sect. 3 presents the proposed solution, the KALI test generator, its structure and data flow. The performance of the tool is evaluated in Sect. 4 by using the benchmark examples provided in the context of a combinatorial test tool competition. In Sect. 5 we present relevant related works on combinatorial test generators and their parallel applications, and finally, Sect. 6 presents some future work and concludes the paper.

2 BACKGROUND

2.1 Combinatorial Interaction Testing

With Combinatorial Interaction Testing (CIT), the tester systematically explores the t-way interaction between all the features inside a given system under test, where t is commonly called *strength*. To achieve this, chosen any t parameters, every combination of values among those parameters is tested at least one (unless there exist

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC '23, March 27-March 31, 2023, Tallinn, Estonia
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577703

```
Model Example1                     Constraints:
Parameters: P1: { v1 v2 };           # ( P3!=true OR P2!=v3 OR P1!=v1) #
            P2: { v3 v4 };
            P3: Boolean;             # ! (P3 = true AND P1 = v2) #
            P4: [0 .. 3]             # (P3 = true) => P4 > 2 #
```

Figure 1: Example of a combinatorial model

constraints that make a combination *unfeasible*). The aim of CIT algorithms is generating in the shortest time the smallest possible test suite achieving the desired coverage. The most often applied combinatorial testing technique is pairwise testing ($t = 2$), which consists in generating a test suite that covers all input values pairs.

Fig. 1 shows a simple example of a combinatorial model, where a list of parameters and a list of constraints are specified using the CTWedge format [11]. In particular, the model contains four *parameters* (one Boolean parameter, two enumeratives, and an integer range) and three *constraints*, which express the logical relation between the parameters and their values. A model such as the one in Fig. 1 can be passed as input to a *test generator* that, given a defined strength t, generates all test cases covering every t-tuple. Note that the CTWedge grammar does not support for now rational numbers, but we plan to implement its grammar in order to allow their use. In this case, generating tests using SMT solvers, as we do in this paper, could be even more suitable.

2.2 SMT solvers

Satisfiability modulo theories (SMT) are used for generalizing, by using first-order formulas, Boolean satisfiability (SAT) by adding several aspects, among which there are equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other first-order theories. An SMT solver is a tool for deciding the satisfiability (or validity) of formulas in these theories. SMT solvers are exploited for different tasks, including extended static checking, predicate abstraction, test case generation, theorem proving, and bounded model checking over infinite domains, to mention a few.

Many solvers have been proposed in the literature, such as Z3 [9], Yices [10], MathSAT5 [8], and SMTInterpol [7]. Since they are generally capable not only of checking the satisfiability of a formula but also of generating a model for it, SMT solvers are often used for test generation [16]. In this paper, we have decided not to commit to any specific solver, and all the algorithms and encodings we propose work for a variety of solvers. In the future, we may decide to adopt a specific tool and tailor all algorithms to it.

2.3 How to represent CIT models, constraints, tuples, and tests

Having introduced in Sect. 2.2 the main features of SMT solvers, we can now explain how we use them for CIT. First, we recall that a combinatorial model is composed of two different sets of elements, namely a set of *parameters* and a set of *constraints*.

2.3.1 Parameters encoding. Given an SMT solver context, the parameters can be represented as variables in that context. In particular, the parameters in a combinatorial model can be of different types, and thus their translation into variables in a solver context has to be done accordingly:

- *Boolean parameters* are represented as SMT Boolean variables;
- *Integer ranges* are represented as SMT integer variables. However, since when defining ranges in combinatorial models one must specify the lower and the upper threshold, we must add another constraint that specifies these limits when converting a range to an SMT variable. For example, if a range is defined in the combinatorial model as `P1 : [0 .. 3]`, in addition to the `P1` integer variable, the following constraint is added: $P1 \geq 0$ `AND` $P1 \leq 3$;
- *Enumerative parameters* can be represented in different ways, depending on the support given by the SMT solver chosen. Since our aim is to propose a solution compatible with all SMT solvers and with the simple SMT theory for Boolean formulas, we convert each enumeration with n values into a set of n SMT Boolean variables and a set of constraints that ensure that only one of the possible Boolean variables must be true. This process is generally defined as *flattening* [13]. For example, the enumeration `P2 : {v1 v2 v3}` is translated into three different SMT Boolean variables `P2v1`, `P2v2`, and `P2v3`. Furthermore, the following constraint must be added for `P2v1`: `P2v1 <=> (not P2v2 and not P2v3)`, and similar ones for the other two variables.

2.3.2 Model constraints. The constraints of a combinatorial model can be easily mapped to SMT constraints, exploiting the variables previously defined. In particular, a combinatorial model may contain relational, mathematical, or comparison operators (between parameters or values) in general propositional formulas. All these aspects can be easily represented with operations between variables and values defined in an SMT context.

2.3.3 Tuples. Once the combinatorial model has been converted to SMT structures, they can be used for generating tests, by checking which of the possible tuples are compatible with the SMT solver context. In particular, given a tuple tp, it can be represented in the SMT solver context by adding a new constraint that limits the values of the parameters to those specified by the tuple. For example, a tuple $tp = \langle P1, v1\rangle\langle P2, v2\rangle$ is translated in the following SMT constraint $P1 = v1$ `AND` $P2 = v2$.

As we will discuss in detail later, our approach builds the tests incrementally, collecting suitable tuples to obtain valid tests. The tests are derived from a combinatorial problem $P = \{X, C\}$, where $X = \{x_i, v_i\}$ is a set of parameters (each x_i with the set of possible values v_i) and C is the set of constraints (as shown in Fig. 1). To store information on the model and its constraints, together with the tuples added so far to a (partial) test, we modify the definition of *test context*, previously defined in [3], as follows.

Definition 2.1 (Test context). We call $TC = \langle A, LC_{TS}\rangle$ a *test context* for a 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 LC_{TS} is the SMT solver context representing P and the assignments committed to the context so far.

Thus, each test context TC contains a list of assignments A that represents a partial test case T together with the solver context containing all the information about the model (parameters and constraints) and the test itself. A test context is complete, i.e., it

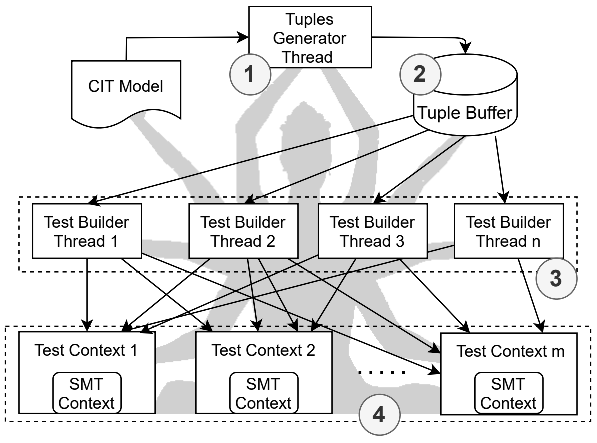


Figure 2: Data flow for the KALI tool

represents a complete test case if the assignment list A includes all the parameters of the model. Given a test context TC and LC_{TS} being its SMT solver context, a tuple tp can be:

- *implied*, if all assignments of the tuple tp are already contained in the assignments A, i.e., the possibly partial test case T.
- *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 does not yet have any value or has an equal value, and tp 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.

While checking whether a tuple tp is implied by a test context TC does not require any call to the SMT solver, to check if tp is compatible or uncoverable, the use of the SMT solver context LC_{TS} is necessary.

3 KALI: USING SMT SOLVERS FOR TEST GENERATION

In this paper, we present KALI, a Java tool that exploits an SMT solver, through the library `java-smt`[1], to generate test suites with combinatorial coverage. It implements multi-threading strategies for reducing the test generation time.

The source code, together with the tests, data, and evaluation scripts used in Sect. 4 are available in the replication package [4].

3.1 Tool architecture and algorithm

The data flow we devised for KALI to generate all the tests is shown in Figure 2. All the tuples to be covered are generated by a single thread (step 1 in Fig. 2) from a CIT model such as the one in Fig. 1. The generated tuples are stored into a shared buffer (step 2 in Fig. 2) with limited capacity (40 tuples in our experiments, but users may configure it with a different capacity). Then, when the shared buffer is fully filled, the tuple generation thread stops and waits until a new free slot is available. This process allows avoiding storing all tuples at the same time, and thus, guarantees a consistent saving in memory utilization, especially for complex combinatorial models, in which the number of tuples can be significantly high. Then, n test builder threads (step 3 in Fig. 2) start consuming the tuples in the tuple buffer. Note that n can be automatically selected by the tool depending on the hardware architecture or decided by the user. Each test builder can consume a tuple tp_i and try to add tp_i to any of the available *test contexts* (step 4 in Fig. 2), which are continuously updated and created if necessary. This process is described in Algorithm 1 and is repeated by each thread until all the tuples have been consumed. In particular, given a tuple tp_i (extracted from the buffer in line 1):

- the function `findImplies` at line 2 finds, in the list of all test contexts TC, the first test context tc_j, which already implies the tuple tp_i, if there exists one. Then, if tc_j has been found, tp_i is consumed;
- the function `findCompatible` at line 6 finds, in the list of all test contexts TC, the first test context tc_j that is compatible with the tuple tp_i (i.e., the tuple does not clash with the assignments already performed by the test context and with the constraints of the combinatorial model), if there exists one. Then, if tc_j has been found, tp_i is passed to tc_j, which updates its SMT solver context, and tp_i is consumed;
- if a thread cannot find a test context tc_j in which the tuple tp_i is compatible or implied, a new test context is created. It is initialized by the function `createTestContext` (line 11) which builds a new SMT solver context tc_j and adds all the variables and constraints of the combinatorial problem to tc_j. If tp_i is compatible with the newly created test context, the tuple is consumed, added to the context; otherwise, it means that the tuple is not compatible with the constraints, it is considered as uncoverable and skipped, and the empty test context is discarded.

When all tuples are consumed, each test context provides the resulting test case, which can be complete (if all parameters have been assigned) or incomplete. In the former case, the test case is derived from the list of assignments A contained in the test case, while in the latter case the test case is derived by computing a model that satisfies the SMT context.

3.2 Algorithm optimizations

The proposed tool implements several optimizations that can be activated to reduce either the generation time or the size of the test suite. In particular, the possible optimizations are in the following directions:

1. *tuple selection or generation*: tuples can be generated (or extracted from the tuple buffer) in an optimized manner, by following specific heuristics,
2. *constraints management*: when no constraints are available in the combinatorial model, no solver context should be created and updated,
3. *selection of test context*: the threads building the tests can give precedence to certain test contexts during their selection (for example, based on the grade of completeness of the context, i.e., the number of assignments already performed).

The first optimization can be performed by modifying the tuple selection or generation process. In fact, from the literature, it is often

[1]https://github.com/sosy-lab/java-smt

Algorithm 1 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:  $tp_i \leftarrow TupBuffer.extractFirst()$
    ▷ Try to find a test context which implies the tuple
2:  $tc_j \leftarrow findImplies(TC, tp_i)$
3:  if $tc_j$ is not $NULL$ then
4:      return
5:  end if
    ▷ Try to find a test context which is compatible with $tp_i$
6:  $tc_j \leftarrow findCompatible(TC, tp_i)$
7:  if $tc_j$ is not $NULL$ then
8:      $tc_j.updateTC(tp_i)$
9:      return
10: end if
    ▷ Create a new empty test context
11: $tc_j \leftarrow createTestContext(M_C)$
12: if $tc_j.isCompatible(tp_i)$ then
13:     $tc_j.updateTC(tp_i)$
14: else
15:     $tp_i.setUncoverable()$
16: end if
17: return
```

reported that the best way for generating the tuples in sequential algorithms consists in starting from the ones containing the parameters with the widest ranges (see the IPO algorithm [18]). For this reason, KALI natively implements the parameter ordering strategy `IN_ORDER_SIZE_DESC` (OD) which allows the generation of tuples starting from the parameters assuming the highest number of values. However, one may want to try different strategies. Thus, other ordering strategies are implemented, such as `IN_ORDER_SIZE_ASC` (OA) which allows to first generate the tuples starting from the parameters assuming the lowest number of values, `RANDOM` (RD), which shuffles the list of parameters before starting the tuple generation process, and `AS_DECLARED` (AD), which considers the parameters in the order in which they are declared inside the combinatorial model. In Sect. 4 we report further details about the impact of the tuple selection strategy on size and generation time.

One of the most costly activities during test generation is the update of the SMT solver context and the computation of the model satisfying all the constraints. This activity can be avoided when the combinatorial model has no constraints: the test can be extracted only by looking at the assignment list A included in each test context. This can be easily implemented by modifying the line 8 in Algorithm 1, which should be changed in the conditional statement reported in Algorithm 2.

Finally, the last optimization regards the test contexts that can be selected by giving higher priority to those that have a lower *completeness grade*, i.e., those that have fewer parameters already assigned. In this way, the number of test cases is reduced since the compatibility and coverability check is first performed on test

Algorithm 2 Optimized test context update

```
$tc.updateAssignments(tp_i)$
if $tc.getConstraintsList().size() > 0$ then
    ▷ Update the SMT solver context of the test context with $tp_i$
    $tc.updateSolverContext(tp_i)$
end if
```

contexts that are more likely able to accept the new tuple. On the other hand, introducing this optimization is not the best idea when the objective is to reduce the generation time, because, especially for combinatorial models with a lot of parameters and values (and thus, implicitly, a lot of test cases), the sorting procedure always adds a significant amount of time (see Sect. 4 for more details).

4 EXPERIMENTS

In this section, we report the results that we have obtained applying KALI to the benchmarks provided by the organizers of the CT-Competition[2], proposed in [2]. These benchmarks have been explicitly introduced for evaluating CIT tools and include models with varying characteristics:

(1) `U_BOOL`: uniform models, with Boolean parameters, and no constraints;

(2) `U_ALL`: uniform models, with all the parameters assuming the same number of values, and no constraints;

(3) `MCA`: with Boolean and enumerative parameters, but no constraints;

(4) `BOOLC`: with Boolean parameters, and logical constraints between them;

(5) `MCAC`: with Boolean and enumeratives parameters, and logical constraints between them;

(6) `NUMC`: with Boolean, enumeratives, and integer ranges parameters, and logical, relational, and mathematical constraints between them.

Using these benchmarks, we have analyzed KALI through a list of research questions as reported in Tab. 1. First, in order to find the best configuration for KALI, from RQ1 to RQ4, we have used the benchmarks given during the training phase of the competition[3] (*limited* model set in Tab. 1). Then, in RQ5 and RQ6, we have compared KALI with other approaches available in the literature using all the 300 benchmarks released for the evaluation phase of the competition.

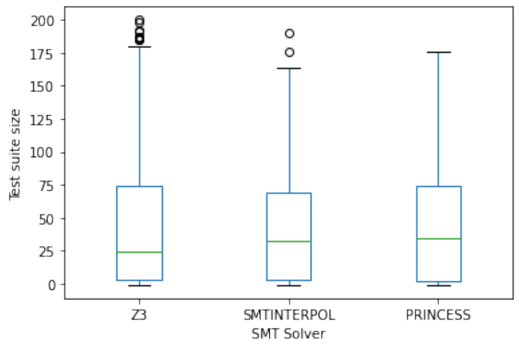
We have executed the experiments on a PC with Ubuntu 20.04.3, 2 CPUs Intel(R) Xeon(R) E5-2620 v4 with 2.10 GHz, 32 threads, and 128 GB RAM. In order to increase the statistical confidence in the results, each test has been executed multiple times (see column *Num Exec.* in Tab. 1) . All data used to answer the research questions, together with the Jupyter Notebook used for the analysis, are available in the replication package [4].

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

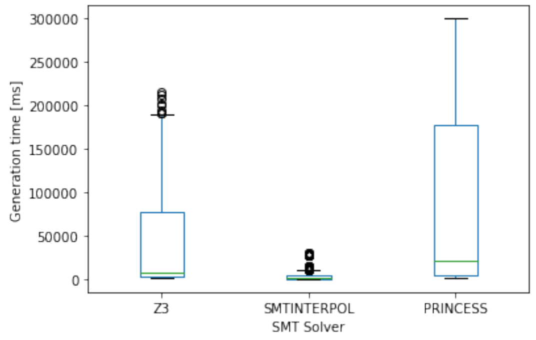
[3]https://github.com/fmselab/ct-competition/raw/gh-pages/examples/ADD_CTWedge.zip

Table 1: Design of the experiments

RQ	RQ Goal	Model Set	Solver	N Threads	Par. Ordering	TC Ordering	Num Exec.	Other Tools
RQ1	SMT solver	limited	all	32	all	all	3	-
RQ2	N threads	limited	SMTInterpol	1,2,4,5,16,32	all	all	10	-
RQ3	Par. Ordering	limited	SMTInterpol	32	all	all	10	-
RQ4	TC Ordering	limited	SMTInterpol	32	all	all	10	-
RQ5	SMT vs MDD	complete	SMTInterpol	32	OD	yes	3	pMEDICI
RQ6	SMT vs IPO	complete	SMTInterpol	32	OD	yes	3	ACTS



(a) Test suite size



(b) Test suite generation time

Figure 3: Impact of the SMT solver used for test generation

4.1 RQ1: Is the choice of the SMT solver important?

Since KALI is implemented using the `java-smt` library, the user can choose which solver to adopt for test generation. Therefore, we have performed the experiments with KALI using three different solvers, namely Z3 [9], SMTInterpol [7], and PRINCESS [17]. Fig. 3a and Fig. 3b report the impact of the SMT solver chosen, respectively, on test suite size and generation time. As expected, the choice of the SMT solver to be used does not significantly influence the test suite size. On the other hand, the SMT solver has a great impact in terms of generation time. In fact, in our case, SMTInterpol has been demonstrated to be the one producing test suites in the shortest amount of time[4].

From now on, we test all the optimizations using only the SMT-Interpol solver and 32 threads, unless otherwise specified. In the following, we run the experiments to collect the data and evaluate the impact of the optimizations by performing a set of *Wilcoxon Signed-Rank tests* [21], in which the null hypothesis is rejected if the p-value[5] is smaller than $\alpha = 0.05$.

4.2 RQ2: What is the gain (if any) in parallelizing the generation algorithm?

In order to assess whether the introduction of multi-threading has improved the performance of the test generation tool, we have executed the experiments on the limited model set using 1, 2, 4, 8, 16, and 32 threads. Fig. 4a and Fig. 4b report the impact of the number of threads used, respectively, on the test suite size and generation time. In particular, Fig. 4a shows that the sum of test suite sizes initially grows when the number of threads increases. This is the price of having multiple threads working on test generation, since more test context can be created. However, this effect may be mitigated (but not overcome) by using a higher number of threads. On the other hand, Fig. 4b evaluates the impact on the test suite generation time. We can observe that introducing a higher number of threads allows for reducing the test suite generation time. Thus, we can conclude, as expected, that the main gain in parallelizing the process of test generation is the significant reduction of the time required for it.

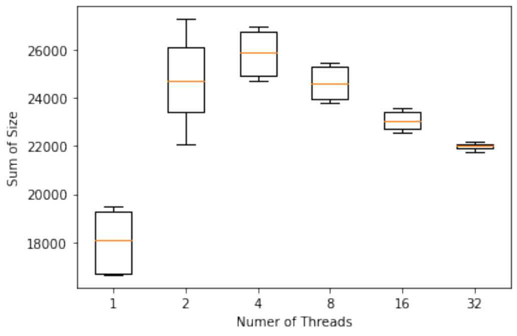
4.3 RQ3: Is the order of the parameters important during tuple generation?

KALI can be configured for considering the parameters in a different order during tuple generation. In particular, they can be considered *as declared* (AD) in the combinatorial model, in an *ascending* (OA) order in terms of the number of possible values, in a *descending* (OD) order in terms of the number of possible values, or in a *random* (RD) order. To evaluate the impact of the chosen parameter ordering strategy, we have performed a *Wilcoxon Signed-Rank test* with the following hypothesis:

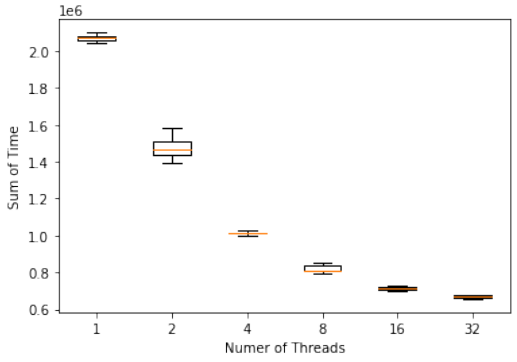
- H_0, the size (or generation time) of the test suites produced with a parameter ordering strategy O_1 is equal to the one of a strategy O_2;
- $H_A{}^+$, the size (or generation time) of the test suites produced with a parameter ordering strategy O_1 is higher than the one of a strategy O_2;

[4]The optimal performance of the SMTInterpol tool are confirmed also by the competition among SMT solvers https://ultimate.informatik.uni-freiburg.de/smtinterpol/news.html

[5]With the p-value we indicate the probability of obtaining test results at least as extreme as the result actually observed, under the assumption that the null hypothesis is correct.



(a) Test suite size



(b) Test suite generation time

Figure 4: Impact of the number of threads used for test generation

- $H_A{}^-$, the size (or generation time) of the test suites produced with a parameter ordering strategy O_1 is lower than the one of a strategy O_2;

Tab. 2a and Tab. 2c report which hypothesis is accepted based on the comparison between an ordering strategy O_1 (on the rows) and a second one O_2 (on the columns), respectively, in terms of test suite size and generation time. Surprisingly, the results obtained suggest that the order of the parameters does not influence the size of the test suite as expected. In fact, using the parameters in the same order in which they are declared in the combinatorial model leads to the smallest test suites (see Tab. 2b) and to results comparable to those obtained using the ordering in a descending way (OD). Note that the models on which we have performed the experiments are randomly generated by the CT-Competition organizers, and thus, the advantages in using the AD ordering strategy may be related to this random origin. In the following, we prefer OD over AD because it is more predictable, since it does not depend on the order in which the parameters are declared in the combinatorial model. Our experiments do not show a considerable advantage in using OD instead of AD, while in the literature OD performs much better than AD. This is probably due to the nature of algorithms, where, for IPO and variants, the order of parameters seems to significantly affect the test suite size. On the contrary, in terms of test suite generation time, the results report that the null hypothesis has to be always accepted, thus, it is not possible to define a method that is the most rapid.

Table 2: Statistical analysis of the impact of parameter order

(a) Impact on size

	AD	OD	OA	RD
AD	–	H_0	$H_A{}^-$	$H_A{}^-$
OD	H_0	–	H_0	$H_A{}^-$
OA	$H_A{}^+$	H_0	–	H_0
RD	$H_A{}^+$	$H_A{}^+$	H_0	–

(b) Average Size

	Size
AD	731.62
OD	732.15
OA	734.78
RD	733.94

(c) Impact on generation time

	AD	OD	OA	RD
AD	–	H_0	H_0	H_0
OD	H_0	–	H_0	H_0
OA	H_0	H_0	–	H_0
RD	H_0	H_0	H_0	–

4.4 RQ4: Is the order of the test contexts important during test generation?

KALI implements an option that allows ordering the test contexts, aiming to choose the best one, i.e. the one that is more likely able to cover the new tuple. To evaluate the impact of test context ordering on test generation, we have performed a *Wilcoxon Signed-Rank test* with the following hypothesis:

- H_0, the size of the test suites produced when the ordering optimization is activated is the same as the one obtained when the optimization is not activated;

The data allowed us to reject the hypothesis H_0 and claim that the size of the test suites is smaller when the ordering optimization is activated. Moreover, we have performed the same analysis on the test suite generation time:

- H_0, the generation time for the test suites produced when the ordering optimization is activated is the same as the one obtained when the optimization is not activated;

The data allowed us to reject the hypothesis H_0 and claim that the generation time for the test suites is, in general, higher when the ordering optimization is activated. In conclusion, looking at the evaluation results, we can observe that ordering the test contexts before trying to assign them a tuple allows producing smaller test suites, with a consistently lower variance, but requires more time.

Once we have tested all the optimizations, we performed an evaluation of KALI w.r.t. other tools using all the 300 benchmarks of the evaluation phase of the CT-competition, by executing 3 times each model and keeping only the best test execution. From the results of the previous research questions, we have configured KALI for using the best configuration possible, i.e., 32 threads (since it is the number of threads - greater than 1 - leading to the lowest total size), the SMTInterpol solver, the ordering optimization for test contexts enabled, and the OD parameter ordering. Moreover, we have set a timeout of 300 seconds for each execution, in order to cap the test generation time.

4.5 RQ5: Is the SMT solver also efficient when no constraints would require its use?

The main advantage of using an SMT solver for generating tests from combinatorial models is that it allows dealing with all types of parameters (including enumeratives) and constraints (including mathematical operations and comparisons between parameters). However, SMT solvers exploit complex sets of heuristics (depending also on the type of the variables contained in the solver context), which can slow down the generation process.

Thus, we have wondered if other solutions are better performing than SMT-based generation tools when the constraints do not require the advanced functionalities offered by the SMT solvers. For this reason, we have compared the performance of KALI and that of pMEDICI [3], which is based on the same concept of parallel test generation but uses multivalued decision diagrams (MDDs) instead of SMT solvers.

By analyzing the results, we could observe that pMEDICI performs better than KALI in most executions, especially in terms of test suite generation time. Also in terms of timeouts, pMEDICI fails only 9 times, whereas KALI 10.

In order to validate our preliminary observations, we have performed a *Wilcoxon Signed-Rank test* comparing the results of KALI and pMEDICI, in which the unsupported and timed-out models have been excluded, with H_0 stating that the two tools have the same test suite size and test suite generation time. For the generation time, the null hypotesis can be rejected and, as we preliminarly observed, we can conclude that KALI requires more time to generate test suites. On the other hand, for the test suite size, the obtained *pvalue* is too high and no conclusions can be drawn from the experiments, so the performances of the two tools can be considered comparable. We suspect that the low time performance is due to the overhead of the SMT solvers w.r.t. the one of MDDs and to the way in which enumeratives are to be encoded by using the java-smt library. For this reason, a specific SMT solver, natively allowing the use of enumeratives, should be investigated. Thus, we believe that the SMT-based solution implemented in KALI is justified, at least for now, only when constraints require the use of SMT solvers, i.e., when comparisons between parameters or mathematical operations are needed.

4.6 RQ6: Is our approach as efficient as others already known in the literature?

As described in Sect. 2.3, through the use of SMT solvers, KALI is able to support all the kinds of parameters and constraints. However, in the literature, other tools supporting the same type of constraints and parameters are available, among which ACTS [24] is one of the most used. By analyzing the data collected with KALI and ACTS on the same benchmarks (including those not supported by pMEDICI), we could observe that ACTS performs better than KALI in most executions. Also in terms of timeouts, ACTS never fails, whereas KALI fails 26 times.

In order to validate these considerations, we have performed a *Wilcoxon Signed-Rank test*, excluding all the instances in which KALI timed out, with H_0 stating that the two tools have the same test suite size and test suite generation time. In both cases, we can reject the null hypothesis, and confirm our preliminary observations that KALI is, in general, slower in generating test suites than ACTS and produces test suites with more test cases.

Advantages of using KALI. RQ5 and RQ6 clearly indicate that the use of the approach implemented by KALI comes with a price. However, we can identify two main advantages w.r.t. other tools and algorithms:

- pMEDICI outperforms KALI especially for generation time, but it cannot deal with complex constraints that cannot be represented in MDDs. For example, even simple constraints, such as $x = y$ where x and y are parameters, cannot be directly represented in an MDD. More complex constraints that include mathematical functions and numbers, such as $x > y + 3$, require the use of a more sophisticated constraint solver, which is more demanding in terms of time.
- ACTS supports a large set of constraints, and it is very efficient. However, ACTS and similar approaches build the whole test suite by adding a parameter one by one, and this means that no test is complete until the generation is completed. KALI instead completes one test by one, and after a short period of time a test case may be already available. This makes approaches like KALI more suitable for online testing [19], where test execution starts during test generation. In this case, having tests immediately available can reduce the time required to discover faults and can drive test generation to improve the fault detection capability.

5 RELATED WORK

While covering array generation and application has been an active field of research for several decades, very few attempts at parallel CIT construction algorithms have been proposed so far, to the best of our knowledge. In [5], the authors were the first to propose the exploitation of high-performance, parallel computing environments for combinatorial optimization problems. Particularly, they used the GRID to run a multi-threaded algorithm that, given a sub-optimal covering array, explores the spanning tree of available moves to reduce its size while preserving coverage. In [6] a slightly different refinements-based combinatorial algorithm is presented, designed to build a covering array in any MPI compliant shared memory, parallel computing environment. Both these and later works from the same authors focus solely on proposing viable solutions to cope with the scalability issues of larger combinatorial problems, i.e., by means of parallel computing or incremental constructions, and do not take constraints management into account.

Grieskamp et al. [12] propose an approach where the whole constrained covering array computing problem is formalized as an SMT constraint resolution task, for which the Z3 solver has been integrated into an iterative test suite construction algorithm. It provides many desirable features, like i.e. support of seeded test cases, generic propositional constraints, and variable strength coverage, but the proposed approach is implemented as a sequential algorithm. Younis et al. [22] proposed a new strategy called multi-core modified input parameter order (MC-IPOG) that allows running the IPOG algorithm [14] on multi-core systems. Avila et al. [1] also presented a strategy to run parallel instances of a simulated annealing metaheuristic combinatorial algorithm on the GRID.

Yilmaz et al. [15] approach is the first to leverage the computing power of thousands of GPU cores for the parallel computation of several instances of a meta-heuristic (again, simulated annealing) algorithm. However, its support for constraints over the inputs is limited to forbidden tuples, in turn managed by the integrated *Sugar* CSP solver. Moreover, as the same authors noted, the effective implementation of a combinatorial heuristic algorithm aimed at GPU hardware is not a straightforward task. Careful design of the algorithm computational steps is required in order to ensure the efficacy of the approach, as the CUDA API environment provides very specific and low-level parallel computing primitives that were not primarily designed for combinatorial optimization tasks. Recent work by Wang et al. [20] adopts *Java Parallel Streams* multi-threading technology to improve the performance of the fitness function computation required by a *tabu-search* based combinatorial optimization algorithm. In this approach, constraints are supported as a set of minimal forbidden tuples (MFT) [23] against which a SAT solver validates candidate test cases.

Our present work derives from the previous effort presented in [3] and differentiates from it for a new heuristic strategy and the integration of an SMT solver for constraints management, based on the java SMT library *java-smt*. To contrast with existing related works, our contribution is originally being the first combinatorial testing approach that, to the best of our knowledge, implements a combinatorial algorithm with a bottom-up parallel design and, at the same time, fully supports constraints expressed as general propositional formulas. No other reviewed approaches integrate both features: in most of them, either parallel execution is applied to just a single processing step or to run multiple instances of the whole (sequential) heuristic algorithm. The remaining approaches, even though feature a parallel algorithm, do not support constraints or provide basic support by means of forbidden tuples listing.

6 CONCLUSIONS

In this paper, we have presented a technique, implemented in the KALI tool, to generate combinatorial test suites. It exploits SMT Solvers and the multi-threading capabilities of recent PCs in order to reduce the time required for the generation of a test suite. Our analysis has confirmed that KALI allows overcoming all the limitations of tools already available in the literature (e.g., the inability of those tools to deal with particular types of constraints). However, the other side of the coin for this ability of dealing with more complex models is, in general, the higher demand in terms of generation time and the larger test suite size we produce with KALI. Thus, as future work, we are planning to introduce some mechanisms which allow reducing the size and the coordination effort between threads, in order to optimize also the generation time. All the analyses conducted in this paper are based on pairwise testing. Nevertheless, our experiments show that our tool also works with t-wise coverage, but further experiments are needed. In general, we believe that the approach we devised for KALI, based on the parallelization and on the use of several test contexts, is promising and can be further extended with more functionalities, heuristics, and coverage criteria.

REFERENCES

[1] H. Avila George. 2012. *Constructing Covering Arrays using Parallel Computing and Grid Computing.* Ph. D. Dissertation. Universitat Politècnica de València.

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

[3] A. Bombarda and A. Gargantini. 2022. Parallel Test Generation for Combinatorial Models Based on Multivalued Decision Diagrams. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE.

[4] A. Bombarda, A. Gargantini, and A. Calvagna. 2022. Replication package for the paper "Multi-thread Combinatorial Test Generation with SMT solvers". https://github.com/fmselab/ct-tools/tree/main/KALI

[5] A. Calvagna, A. Gargantini, and E. Tramontana. 2009. 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.* 213–218.

[6] A. Calvagna, G. Pappalardo, and E. Tramontana. 2012. 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.* 149–153.

[7] J. Christ, J. Hoenicke, and A. Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Model Checking Software*, Alastair Donaldson and David Parker (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 248–254.

[8] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. 2013. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 93–107.

[9] L. De Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[10] B. Dutertre. 2014. Yices 2.2. In *Computer-Aided Verification (CAV'2014) (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 737–744.

[11] A. Gargantini and M. Radavelli. 2018. Migrating Combinatorial Interaction Test Modeling and Generation to the Web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 308–317.

[12] W. Grieskamp et al. 2009. Interaction coverage meets path coverage by SMT constraint solving. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5826 LNCS (2009), 97–112.

[13] C. Henard, M. Papadakis, and Y. Le Traon. 2015. Flattening or not of the combinatorial interaction testing models?. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 1–4.

[14] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. 2007. 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.

[15] H. Mercan, C. Yilmaz, and K. Kaya. 2019. CHiP: A Configurable Hybrid Parallel Covering Array Constructor. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1270–1291.

[16] J. Peleska, E. Vorobev, and F. Lapschies. 2011. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11).* Springer-Verlag, Berlin, Heidelberg, 298–312.

[17] P. Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proc., 15th Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LNCS, Vol. 5330).* Springer, 274–289.

[18] K. C. Tai and Y. Lie. 2002. A Test Generation Strategy for Pairwise Testing. *IEEE Trans. Softw. Eng.* 28, 1 (jan 2002), 109–111.

[19] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. 2005. Online Testing with Model Programs. *SIGSOFT Softw. Eng. Notes* 30, 5 (sep 2005), 273–282. https://optdoi.org/10.1145/1095430.1081751

[20] Y. Wang et al. 2022. An Adaptive Penalty based Parallel Tabu Search for Constrained Covering Array Generation. *Information and Software Technology* 143 (2022).

[21] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering.* Springer Berlin Heidelberg.

[22] Mohammed I. Younis and Kamal Z. Zamli. 2010. MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems. *ETRI Journal* 32, 1 (feb 2010), 73–83.

[23] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. 2015. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 1–9.

[24] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn. 2013. ACTS: A Combinatorial Test Generation Tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* 370–375.