Flattening or Not the Combinatorial Interaction Testing Models?

Christopher Henard, Mike Papadakis, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg, Luxembourg, Luxembourg
christopher.henard@uni.lu, michail.papadakis@uni.lu, yves.letraon@uni.lu

***Abstract*—Combinatorial Interaction Testing (CIT) requires the use of models that represent the interactions between the features of the system under test. In most cases, CIT models involve Boolean or integer variables and constraints among them. Thus, applying CIT requires solving the involved constraints which can be directly performed using Satisfiability Modulo Theory (SMT) solvers. An alternative practice is to flatten the CIT model into a Boolean model and use Satisfiability (SAT) solvers. However, the flattening process artificially increases the size of the employed models, raising the question of whether it is profitable or not in the CIT context. This paper investigates this question and demonstrates that flattened models, despite being much larger, are processed faster with SAT solvers than the smaller original ones with SMT solvers. These results suggests that flattening is worthwhile in the CIT context.**

I. Introduction

Combinatorial Interaction Testing (CIT) is a widely used technique for uncovering interactions faults between parameters or features of software systems [1]. This approach is especially useful for systems involving a large number of parameters, configuration options or events, such as operating systems or Software Product Lines (SPLs) [2]. Applying CIT requires modeling all the parameters and their constraints that may influence the system under test. This results in a model that is called CIT model.

Most of the existing CIT approaches deal only with Boolean parameters and constraints. For instance, the approach proposed by Henard *et al.* [3], which has been shown to outperform the current state-of-the-art tools in terms of scalability, operates in these models type of models. These types of Boolean CIT model are called feature model. The approach is promising since it is both scalable and capable of generating and prioritizing t-wise sets efficiently. Since many real world problems have more complex properties than Boolean ones, such as program inputs [4], [5], it is legitimate to wonder whether approaches based on Boolean models are really useful in practice. Otherwise, they are limited to the scope of simple problems. In other words, apart from Boolean models, we need to consider using integer or more complex CIT models, as they are more realistic.

In practice, any CIT model containing a finite set of values for the variables can be transformed to a model where all its constraints and variables are Boolean. This process is called flattening and it has the particularity of making the model artificially larger both in terms of number of variables and constraints. For instance, a feature model of more than 6,000 parameters has been reverse-engineered for Linux [6]. However, many configuration parameters in Linux are non-Boolean and have thus been flattened, meaning that a non-Boolean model for Linux would contain much less variables and constraints. It raises the question of whether we should prefer flattened CIT models over the original and smaller CIT models.

In this context, we investigate whether flattening CIT models to Boolean ones is slowing down the constraint solving process. To this end, we employ CIT models from the literature that we flatten to Boolean ones. We then use Satisfiability Modulo Theory (SMT) solvers to solve the CIT models and Satisfiability (SAT) solvers to solve the flattened CIT model. In other words, we try to answer the question of whether the use of SMT over the SAT solvers is profitable in the context of CIT. Our experiments show that, despite the larger amount of variables and constraints to handle, flattened CIT models are processed faster with SAT solvers than the smaller original ones with SMT solvers.

The remainder of this paper is organized as follows. Section II presents the flattening of CIT models by first describing it on an example and then formalizing the general methodology. Then, Section III describes the conducted experiments and examines threats to its validity. Finally, Section IV discusses related work before Section V concludes the paper.

II. The Flattening of Combinatorial Models

This section introduces the flattening process of CIT models through an example. This process follows the lines suggested by Cohen *et al.* [7] and Papadakis *et al.* [5]. Then, the general flattening methodology is formalized.

We represent a CIT model M as a tuple $M = (O, C)$, where O is the set of n options (or variables) and C is the set of k constraints between the options. Each constraint is written as a disjunctive clause. The objective is to flatten M into a Boolean model denoted as $M_f = (O_f, C_f)$.

A. A Running Example

We present a small example for flattening a CIT model $M = (O, C)$ into $M_f = (O_f, C_f)$.

1) A Simple CIT Model: As a running example, consider the following CIT model with two options and one constraint: $M = (\{o_1 \in \{v_1, v_2\}, o_2 \in \{v_3, v_4, v_5\}\}, \{(o_1 = v_1 \vee o_2 \neq v_5)\})$. The first option can take the two values v_1 and v_2 while the second one can take the three values v_3, v_4 and v_5. Thus, the domain of o_1 is $\{v_1, v_2\}$ and the domain of o_2 is

$\{v_3, v_4, v_5\}$. Note that the CIT model options of this example can be represented as $2^1 3^1$, such as in the work of Petke *et al.* [4]. It means that there is one option that can take two different values and one option that can take three values. The example model also involves the constraint $(o_1 = v_1 \vee o_2 \neq v_5)$, meaning that we require either o_1 taking the value v_1 or o_2 taking a value different from v_5. Note that this constraint can be written as $(v_1 \vee \neg v_5)$.

2) Flattening the Options: To flatten the options, we simply create one option per option value. Thus, flattening o_1 results in two Boolean options o_{v_1} and o_{v_2} and flattening o_2 leads to three Boolean options o_{v_3}, o_{v_4} and o_{v_5}. As a result, flattening the model increased the number of variables from 2 to 5, and we have $O_f = \{o_{v_1}, o_{v_2}, o_{v_3}, o_{v_4}, o_{v_5}\}$, with each $o_{v_i} \in \{true, false\}$.

3) Flattening the Constraints: Flattening the constraints of M is quite immediate. We simply match the values in the constraint to the new options. Thus, the constraint $(o_1 = v_1 \vee o_2 \neq v_5) = (v_1 \vee \neg v_5)$ is transformed to $(o_{v_1} = true \vee o_{v_5} = false)$, which can be written more simply as $(o_{v_1} \vee \neg o_{v_5})$. In the following, we will use this simplified notation.

Besides to the constraint of M, we also need to add additional constraints due to the flattening of the options. These constraints prevents a given option to take two different values at the same time. Thus, since we cannot have at the same time $o_1 = v_1$ and $o_1 = v_2$, we add the following constraints: $(o_{v_1} \vee o_{v_2})$ and $(\neg o_{v_1} \vee \neg o_{v_2})$. We process similarly for o_2, thus adding the four following constraints: $(o_{v_3} \vee o_{v_4} \vee o_{v_5}), (\neg o_{v_3} \vee \neg o_{v_4}), (\neg o_{v_3} \vee \neg o_{v_5})$ and $(\neg o_{v_4} \vee \neg o_{v_5})$. As a result, the flattening increased the number of constraints from 1 to 7 and we have $C_f = \{(o_{v_1} \vee \neg o_{v_5}), (o_{v_1} \vee o_{v_2}), (\neg o_{v_1} \vee \neg o_{v_2}), (o_{v_3} \vee o_{v_4} \vee o_{v_5}), (\neg o_{v_3} \vee \neg o_{v_4}), (\neg o_{v_3} \vee \neg o_{v_5}), (\neg o_{v_4} \vee \neg o_{v_5})\}$.

B. General Flattening Methodology

The flattening methodology applied to the example is formalized as follows:

1) Flattening the Options: If $O = \{o_1, ..., o_n\}$, where each option o_i can take up to m values in a domain $D_i = v_1^i, ..., v_m^i$, then $O_f = \{o_{v_1^1}, ..., o_{v_m^1}, ..., o_{v_1^n}, ..., o_{v_m^n}\}$, where each $o_{v_y^x}$ is Boolean.

2) Flattening the Constraints: If $C = \{c_1, ..., c_k\}$, where each constraint c_j is a disjunctive clause of the form $[\neg]v_y^x \vee ... \vee [\neg]v_y^x$, with $x \in \{1, ..., n\}$, $y \in \{1, ..., m\}$ and $[\neg]$ meaning the presence of a negation or not, then $C_f = \{c'_1, ..., c'_k\}$ with each c'_j of the form $[\neg]o_{v_y^x} \vee ... \vee [\neg]o_{v_y^x}$.

The, additional constraints, which prevent options to take several values at the same time, are added to C_f. For the sake of simplicity, we now denote $O_f = \{o_{v_1^1}, ..., o_{v_m^1}, ..., o_{v_1^n}, ..., o_{v_m^n}\}$ as $O_f = \{o_{f_1}, ..., o_{f_s}\}$. We first add to C_f the constraint $(o_{f_1} \vee o_{f_2} \vee ... \vee o_{f_s})$. Then, for each o_{f_i} such as $1 \leq i < s$, the following constraints are added to $C_f : (\neg o_{f_i} \vee \neg o_{f_{i+1}}), (\neg o_{f_i} \vee \neg o_{f_{i+2}}), ..., (\neg o_{f_i} \vee \neg o_{f_s})$.

III. Experiments

In this section, we compare the execution time of SMT solvers on CIT models against the execution time of SAT solvers on the flattened version of each CIT model. The objective is to determine whether the flattening of CIT models has an impact on the time required by a solver to solve the model. Generally, there are two types of solvers that are compared, i.e., the SAT and the SMT, on two instances of the same problem, i.e., the flattened and the unflattened CIT model. Despite solving the same problem, the flattened model requires from the SAT solver to handle a considerably higher number of options and constraints than what it is required by the unflattened one from the SMT solver. Thus, we aim at answering the following research question (RQ):

TABLE I. The subject CIT models and their corresponding flattened versions.

	CIT Model [4], [8]		Flattened CIT Model	
	Options	Constraints	Options (Bool.)	Constraints
Flex	$9\,(2^6 3^2 5^1)$	12	23	43
Sed	$11\,(2^7 3^1 4^1 6^1 10^1)$	50	37	137
Grep	$9\,(2^1 3^3 4^2 5^1 6^1 8^1)$	83	38	167
Make	$10\,(2^{10})$	1	20	21
Gzip	$14\,(2^{13} 3^1)$	61	29	91
TCAS	$12\,(2^7 3^2 4^1 10^2)$	6	44	127
Storage5	$22\,(2^5 3^8 5^3 6^1 8^1 9^1 10^2 11^1)$	246	109	567
Services	$13\,(2^3 3^4 5^2 8^2 10^2)$	404	64	598
Insurance	$14\,(2^6 3^1 5^1 6^2 11^1 13^1 17^1 31^1)$	0	104	797
GCC	$199\,(2^{189} 3^{10})$	15,705	408	16123

[RQ] Does the flattened CIT models are processed slower than the unflattened ones when using SAT and SMT solvers?

In the following, the setup and results of the experiments are presented. Then, the research question is answered and possible threats to the validity of the experiments are presented.

A. Setup

We use 10 CIT models presented in the work of Petke *et al.* [4] and [8]. These experimental subjects were chosen because they are both real and of different sizes and complexity. For each of these models, we flattened them into Boolean ones. The details of these models are recorded in Table I. For each subjects, it presents the options and constraints it contains. For instance, the flex CIT model has a total of 9 options, where 6 options can take two different values, 2 options can take 3 values and one option can take 5 values. It also involves 12 constraints. The corresponding flattened model encompasses 23 Boolean options and 43 constraints.

We employ 4 SMT solvers to solve the CIT models and 4 SAT solvers to solve the corresponding flattened models. The SMT solvers are Yices v2.2.2, Z3 v4.3.3, veriT v201410, and CVC4 v1.4. The SAT solvers are Yices-sat v2.2.2, MiniSat v2.2.0, Glucose v4.0 and PicoSAT v960. The solvers were selected according to their popularity or performances in SMT/SAT competitions. For instance, veriT was declared the overall winner of the 2014 SMT competition[1]. All these solvers are written in C or C/C++. The version used are the latest versions available at the moment of writing.

[1] `http://smtcomp.sourceforge.net/2014/`.

TABLE II. AVERAGE (AVG) AND TOTAL TIME TO SOLVE THE SUBJECT CIT MODELS. EACH SOLVER HAS SOLVED EACH MODEL 10,000 TIMES.

	SMT Solvers (on CIT models)						SAT Solvers (on flattened CIT models)					
					SMT time per model						SAT time per model	
	Yices	Z3	veriT	CVC4	Avg	Total	Yices-sat	MiniSat	Glucose	PicoSAT	Avg	Total
Flex	2.22 ms	11.14 ms	3.24 ms	10.68 ms	6.82 ms	272.78 s	0.69 ms	2.70 ms	3.01 ms	1.27 ms	1.91 ms	76.67 s
Sed	4.13 ms	12.13 ms	5.90 ms	15.60 ms	9.44 ms	377.62 s	0.73 ms	2.76 ms	3.34 ms	1.51 ms	2.09 ms	83.68 s
Grep	4.32 ms	12.65 ms	6.68 ms	16.98 ms	10.16 ms	406.32 s	0.73 ms	2.77 ms	3.51 ms	1.47 ms	2.12 ms	84.90 s
Make	1.91 ms	10.78 ms	2.65 ms	09.22 ms	6.14 ms	245.57 s	0.70 ms	2.70 ms	2.98 ms	1.24 ms	1.90 ms	76.18 s
Gzip	3.55 ms	12.03 ms	5.31 ms	15.00 ms	8.98 ms	359.00 s	0.70 ms	2.71 ms	3.15 ms	1.31 ms	1.97 ms	78.64 s
TCAS	2.37 ms	12.60 ms	3.69 ms	12.15 ms	7.70 ms	308.10 s	0.83 ms	3.24 ms	3.77 ms	1.63 ms	2.37 ms	94.70 s
Storage5	9.55 ms	16.01 ms	12.06 ms	29.45 ms	16.77 ms	670.65 s	0.99 ms	3.45 ms	4.72 ms	2.30 ms	2.87 ms	114.45 s
Services	9.07 ms	17.80 ms	15.19 ms	38.82 ms	20.22 ms	808.78 s	1.04 ms	3.46 ms	4.94 ms	2.28 ms	2.93 ms	117.26 s
Insurance	2.18 ms	12.46 ms	3.06 ms	11.15 ms	7.21 ms	288.54 s	1.12 ms	3.64 ms	5.21 ms	2.45 ms	3.11 ms	124.26 s
GCC	192.4 ms	198.3 ms	279.5 ms	903.8 ms	393.5 ms	15,740 s	9.41 ms	8.07 ms	14.37 ms	14.86 ms	11.68 ms	467.14 s
Avg all models	23.17 ms	31.6 ms	33.73 ms	106.285 ms			1.69 ms	3.55 ms	4.90 ms	3.03 ms		
Total all models	2,317 s	3,159 s	3,373 s	10,629 s			169.39 s	354.93 s	490.22 s	303.393 s		

The experiments are performed on a Intel Core i7-2720QM CPU@2.20GHz with 4GB of RAM running Linux 3.11.0-18-generic. The solvers are compiled with the Gnu Compiler Collection (GCC) v4.8.1. We run each SMT solver on each CIT models 10,000 times, and we record the time required to solve the model, i.e., the time to decide whether it is satisfiable or not[2]. We process similarly for the SAT solvers by using the flattened models. The execution time has been recorded using the `perf-stat` command of Linux. The solvers have been used "out of the box", with no specific option or setting.

B. Results

Table II records, for each model, the average time out of 10,000 executions each solver required to solve it. The table also records the average and total SMT and SAT solving time per model as long as the average and total time required by each solver to solve all the models for the 10,000 runs. For instance, the Yices SMT solver took an average time of 2.22 ms to solve the flex CIT model. The last two lines of Table II indicate that Yices took 23.17 ms in average to solve the 10 CIT models for a total time of 2,317 s. The last two columns of the SMT side indicates that the flex CIT model was solved in 6.82 ms in average or 272.78 s in total by the SMT solvers.

[2]In order to decide whether a model is satisfiable (sat) or not (unsat), the solver tries to find an assignment for the variables which is satisfying the constraints of the model. When such an assignment is found, the model is considered as satisfiable, and unsatisfiable otherwise.

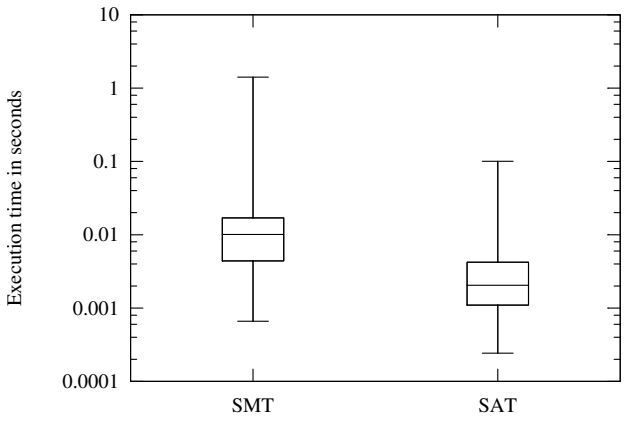

Fig. 1. Distribution of the recorded execution times. It encompasses 400,000 values for SMT and 400,000 for SAT (10,000 runs × 4 solvers × 10 models).

Following Table II, we can observe that the time required by SAT solvers on the flattened CIT models is practically always smaller than the solving time of SMT solvers. In most cases, SAT solving is 2 to 3 times faster. This is happening despite the additional constraints and options resulting from the flattening. For instance, with reference to Table I, the flattening of the grep model increases the number of options by more than 300% and the number of constraints by more than 100%. Still, whatever the SMT solver considered, any of the SAT solver investigated is performing faster on the flattened model. In some rarely cases, SMT solvers can be faster than SAT, such as the Yices and veriT solvers which perform faster than MiniSAT and Glucose on the make model. However, this is only valid for specific solvers. The average and total SAT time is always lower than the SMT ones. This is due to the fact that when SMT is faster than SAT, the differences observed are less important than when SAT is faster. Finally, the biggest difference is observed for the GCC model, which is more than 30 times faster with SAT solvers.

Finally, Figure 1 shows the variation among the execution times recored for the SMT and SAT solvers. For each type of solver, it represents 400,000 execution time values (in seconds) which are represented through a box plot (10,000 runs × 4 solvers × 10 models). From this figure, we can see that the overall execution times for SAT are lower than the SMT ones. Indeed, the minimum, 1st quartile, median, 3rd quartile and maximum values for SAT are lower than the corresponding ones for SMT. The median values for SMT is 0.01 seconds while it is only 0.002 for SAT, demonstrating that generally, SAT performs faster on the flattened models. Finally, the analysis of the two samples of 400,000 values

with the Mann-Whitney U test[3] results in a p-value lower than 0.0001, indicating that the difference in the execution times between SMT and SAT is statistically highly significant.

C. Answer to the Research Question

Based on the presented results and analysis we performed, and with the current level of technology, we have demonstrated that the overall processing time required by the SAT solvers is faster than the one required by the SMT solvers. This is true on all the considered models. Therefore, our experiment leads us to the conclusion that the flattening process does not make the solving process slower.

D. Threats to Validity

We can identify several threats to the validity of this work. First, we cannot ensure that the reported results do generalize to other models. To reduce this threat, we used 10 real CIT models from the literature. Each of these models has a different number of options and constraints. Additionally, the chosen subjects have a varying number of values that can be related with each considered parameter. Our results are consistent and they show that the faster SAT solvers are always faster than all the SMT ones. Even the slower SAT solvers are almost as faster than the fastest SMT ones. It thus gives confidence that the reported results can, to some extent, be generalized.

Another threat is attributed to the influencing factors of the the execution times and the way they are measured. Indeed, these times can be altered or biased by the way the operating system is executing the processes. To reduce this threat, we used the `perf-stat` tool of Linux, which is designed for collecting and analyzing the performance of a command. We also repeated the experiments 10,000 times. Our results were analyzed with statistical tests to ensure that all the reported differences are statistically significant. Finally, potential error in the models can affect the reported results. To reduce this threat, we manually and carefully checked all the models involved in the experiments. We also enable the reproducibility of the conducted experiments by making publicly available all the models used in the experiments.

IV. Related Work

CIT models have been used in a wide variety of work. For instance, Petke *et al.* [4] used CIT models to show that test suites exercising a higher number of interactions find more faults. Several test frameworks have been developed for defining CIT models and performing testing strategies based on CIT, such as [9] and [10]. Similarly, Boolean CIT models were also used by many researchers for the purpose of generating or prioritizing interaction test suites, like in the work of Henard *et al.* [3] or Cohen *et al.* [7]. In this work, we studied both Boolean and non-Boolean CIT models in the perspective of the execution time required to be processed.

[3]The Mann-Whitney U Test is a non-parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other. This test outputs a probability called p-value which represents the probability that the two samples are equal. It is conventional in statistics to consider that the difference is significant if the p-value is lower than the 5% level.

Regarding the flattening methodology, we used a similar process as the one described in the work of Papadakis *et al.* [5] Cohen *et al.* [7]. For the case of feature models where the features are represented hierarchically with a tree, flattening has been used to reduce the depth of the tree in order to facilitate the application of CIT on such models [11]. In this paper, we flatten the variables (or the logic) of the CIT model in order to make each variable Boolean instead of multiple values. We do not handle graphical model. Finally, regarding the transformation of non-Boolean logic formulas, general rules can be found in the work of Frisch *et al.* [12].

V. Conclusion and Future Work

In this paper, we conducted an experiment on 10 CIT models and 8 constraint solvers to evaluate whether flattened CIT models slow down the solving process. Surprisingly, and despite the higher number of options and constraints, the results show that flattened models handled by SAT solvers are not slower to process than unflattend ones handled by SMT solvers. Therefore, it shows that existing tools operating on Boolean models do not have any disadvantage on their efficiency compared to those operating on non-Boolean ones.

In future work, we plan to investigate the findings of the present paper to a larger set of CIT models. In particular, we will use very large models and analyze how the complexity of the models impacts the solving times.

Finally, we make available all the models we used at `http://research.henard.net/SPL/IWCT_2015`.

References

[1] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software fault interactions and implications for software testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 418–421, 2004.

[2] D. Blue, I. Segall, R. Tzoref-Brill, and A. Zlotnick, "Interaction-based test-suite minimization," in *ICSE*, 2013, pp. 182–191.

[3] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 650–670, 2014.

[4] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *ESEC/FSE*, 2013, pp. 26–36.

[5] M. Papadakis, C. Henard, and Y. L. Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," in *ICST*, 2014, pp. 1–10.

[6] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*, 2011, pp. 461–470.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *ISSTA*, 2007, pp. 129–139.

[8] Y. Jia, M. Cohen, and M. H. J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *ICSE*, 2015.

[9] A. Calvagna, A. Gargantini, and P. Vavassori, "Combinatorial interaction testing with CITLAB," in *ICST*, 2013, pp. 376–382.

[10] L. Yu, Y. Lei, R. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *ICST*, 2013, pp. 370–375.

[11] S. Oster, F. Markert, and P. Ritter, "Automated incremental pairwise testing of software product lines," in *SPLC*, 2010, pp. 196–210.

[12] A. M. Frisch, T. J. Peugniez, A. J. Doggett, and P. Nightingale, "Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings," *J. Autom. Reasoning*, vol. 35, no. 1-3, pp. 143–179, 2005.