Combining Satisfiability Solving and Heuristics to Constrained Combinatorial Interaction Testing

Andrea Calvagna[1] and Angelo Gargantini[2]

[1] University of Catania, Italy
Dip. Ingegneria Informatica e delle Telecomunicazioni
andrea.calvagna@unict.it
[2] University of Bergamo, Italy
Dip. Metodi Matematici e Ingegneria dell'Informazione
angelo.gargantini@unibg.it

Abstract. Combinatorial interaction testing aims at revealing errors inside a system under test triggered by unintended interaction between values of its input parameters. In this context we defined a new greedy approach to generate a combinatorial interaction test suites in the presence of constraints, based on integration of an SMT solver, and ordered processing of test goals. Based on the observation that the processing order of required combinations determines the size of the final test suite, this approach has been then used as a framework to evaluate a set of deterministic ordering strategies, each based on a different heuristic optimization criteria. Their performance has been assessed and contrasted also with those of random and dummy ordering strategies. Results of experimental assessment are presented and compared with well-known combinatorial tools.

1 Introduction

Verification and validation of highly-configurable software systems, such as those supporting many optional or customizable features, is a challenging activity. In fact, due to its intrinsic complexity, formal specification of the whole system may require a great effort. Modeling activities may become extremely expensive and time consuming, and the tester may decide to model (at least initially) only the inputs and require they are sufficiently covered by tests.

To this aim, combinatorial interaction testing (CIT) techniques [11,19,25] can be effectively applied in practice [1,29,24]. CIT consists in employing combination strategies to select values for inputs and combine them to form test cases. The tests can then be used to check how the interaction among the inputs influences the behavior of the original system under test. The most used combinatorial testing approach is to systematically sample the set of inputs in such a way that all t-way combinations of inputs are included. This approach exhaustively explores t-strength interaction between input parameters, generally in the smallest possible test executions. For instance, pairwise interaction testing aims at generating a reduced size test suite which covers all *pairs* of input values. Significant time savings can be achieved by implementing this kind of approach,

as well as in general with t-wise interaction testing. As an example, exhaustive testing of a system with a hundred boolean configuration options would require 2^{100} test cases, while pairwise coverage for it can be accomplished with only 10 test cases. Similarly, pairwise coverage of a system with twenty ten-valued inputs (10^{20} distinct input assignments possible) requires a test suite sized less than 200 tests cases only.

CIT requires just specification of the input model, thus it could be defined as an *input-based* testing technique. Since it is also possible to model the output of a system under test as a boolean variable describing only the (success or failure) outcome of each applied test. It is then clear that CIT actually implements a particular type of *functional* (I/O based) testing [26], focused just on the coverage of *interactions* between the inputs, at various degrees of depth. The importance of such functional testing criteria is more evident if we think at the inputs as actually modelling interactions between system components, as it is the case for testing of GUIs or protocols, where we aim at testing just combinations of possible interaction sequences. Nevertheless, several case studies [25,33] have proved the general effectiveness of such functional approach by showing that unintended interaction between optional features can lead to incorrect behaviors which may not be detected by traditional testing.

CIT is also widely recognized as effective in revealing software defects [23]. In particular, experimental research work [25] showed that usually 100% of faults in a software system are already triggered by just a relatively low degree of feature interaction, typically 4-way to 6-way. Dalal et al. [12], showed that testing of all pairwise interactions of a set of software system was able to detect a significant percentage of the existing faults (between 50% and 75%). Dunietz *et al.* [13] compared t-wise coverage to random input testing with respect to structural (block) coverage achieved, with results showing higher reliability of the former in achieving block coverage if compared to random test suites of the same size. Burr and Young [3] report 93% code coverage as a result from applying pairwise testing of a commercial software system. For these reasons combinatorial testing, besides being an active research area, is largely used in practice and supported today by many tools (see [27] for an up to date listing).

Combinatorial testing is applied to a wide variety of problems: highly configurable software systems, software product lines which define a family of softwares, hardware systems, and so on. As an example, Table 1 reports the input domain of a simple telephone switch billing system [26], which processes telephone call data with four call properties, each of which has three possible values: the `access` parameter tells how the calling party's phone is connected to the switch, the `billing` parameter says who pays for the call, the `calltype` parameter tells the type of call, and the last parameter, `status` tells whether or not the call was successful or failed either because the calling party's phone was busy or the call was blocked in the phone network. While covering all the possible combinations for the BBS inputs shown in Table 1 would require $3^4 = 81$ tests, the pairwise coverage of the BBS can be obtained by the test suite reported in Table 2 which contains only 11 tests.

Table 1. Input domain of a basic billing system (BBS) for phone calls.

access	billing	calltype	status
LOOP	CALLER	LOCALCALL	SUCCESS
ISDN	COLLECT	LONGDISTANCE	BUSY
PBX	EIGHT_HUNDRED	INTERNATIONAL	BLOCKED

Table 2. A test suite for pairwise coverage of BBS.

#	billing	calltype	status	access
1	EIGHT_HUNDRED	LOCALCALL	BUSY	PBX
2	CALLER	LONGDISTANCE	BLOCKED	LOOP
3	EIGHT_HUNDRED	INTERNATIONAL	SUCCESS	ISDN
4	COLLECT	LOCALCALL	SUCCESS	LOOP
5	COLLECT	LONGDISTANCE	BUSY	ISDN
6	COLLECT	INTERNATIONAL	BLOCKED	PBX
7	CALLER	LOCALCALL	SUCCESS	ISDN
8	CALLER	LOCALCALL	BUSY	PBX
9	EIGHT_HUNDRED	LONGDISTANCE	BLOCKED	ISDN
10	COLLECT	LONGDISTANCE	BUSY	LOOP
11	COLLECT	LONGDISTANCE	SUCCESS	LOOP

In most cases, constraints or dependencies exist between the system inputs. They normally model assumptions about the environment or about the system components or about the system interface and they are normally described in natural language. If constraints are considered, then the combinatorial testing becomes constrained combinatorial interaction testing (CCIT). However, as explained in sections 2 and 5, most combinatorial testing techniques either ignore the constraints which the environment may impose on the inputs or require the user to modify the original specifications and add extra information to take into account the constraints.

Based on a CCIT construction approach which extends our own previous work [4,5], in this paper we present a study focused on the use of heuristics to order the test goals and assess their impact on the size of generated test suites. Moreover, in contrast to our previous experience, in the proposed approach we experimented implementing the support for constraints by means of integrating the well known satisfiability solver Yices [14] in the construction process.

The paper is organized as follows: Sect. 2 presents our approach and its key points, Sect. 2.1 discusses how the Yices tool is used to generate constrained combinatorial tests, Sect. 3 introduces a set of heuristic strategies for which Sect. 4 reports and discusses results of their experimental evaluation. A comparison with other techniques and tools is reported in Sect. 5, and finally Sect.6 draws our conclusions.

2 Background

Our approach to combinatorial testing can be classified as *logic-based*, since it formalizes the combinatorial coverage in the presence of constraints by means

of logical predicates and applies techniques normally used for solving logical problems to it. To formalize pairwise testing, which aims at validating each possible pair of input values for a given system under test, we can formally express each pair as a corresponding logical expression, a *test predicate* (or test goal), e.g.: $p_1 = v_1 \wedge p_2 = v_2$, where p_1 and p_2 are two input variables and v_1 and v_2 are two possible values of $p1$ and $p2$ respectively. Similarly, the t-wise coverage can be modeled by a set of test predicates, each of the type:

$$p_1 = v_1 \wedge p_2 = v_2 \wedge \ldots \wedge p_t = v_t \equiv \wedge_{i=1}^{t} p_i = v_i$$

where $p_1, p_2 \ldots p_t$ are t inputs and $v_1, v_2 \ldots v_t$ are their possible values. We define a *test* as an assignment to all the inputs of the system. We say that a test *ts* covers a test predicate *tp* if and only if it is a model of *tp*, and we formally represent this fact as $ts \models tp$. Note that while a test binds *every* variable to one of its possible values, a test predicate binds only t variables. We say that a test suite achieves the t-wise combinatorial coverage if all the test predicates for the t-wise coverage are covered by at least one test in the suite. The main goal of combinatorial testing is to find a possibly small test suite able to achieve the t-wise coverage.

In order to generate the test predicates, we assume the availability of a formal description of the system under test. This description should include at least the input parameters together with their domains[3], but could also include constraints over the inputs expressed as axioms. By formalizing the t-wise testing by means of logical predicates, finding a test that satisfies a given predicate reduces to a logical problem of finding a complete[4] model for a logical formula representing the test predicate and the constraints. Formally, the first task is to find the test ts that solves the equation $ts \models tp \wedge c_1 \wedge \ldots \wedge c_n$, where c_i represent the constraints. To this aim, many techniques like constraint solvers and model checkers can be applied. In this approach the constraints become first class citizens and they can be represented by logical expressions.

To generate the complete test suite, one can generate all the test predicates by a simple combinatorial algorithm and then proceed *incrementally* to generate the tests, that is choosing one test predicate at the time and trying to generate a test that covers it and satisfies the constraints. In [4] we already proposed three enhancements of this approach, which have been applied also in the present context. The first consists in *monitoring* the test generation, i.e. marking the test predicates covered by the found test, and skipping them in the next runs. The second consists in conjoining as many as possible *compatible* test predicates, and using this bigger, extended test predicate to derive the test case, in order to increase coverage more quickly and also reduce the number of runs of the external solver. This stage of the construction process can be referred to as *composition* of the test predicates and precedes every run of the external test generation which in [4] was a model checker. The third enhancement consists in further reducing the size of the overall test suite by searching for existence of any redundant test case, that is a test whose predicates are all already covered by

[3] Currently, only finite, discrete enumerable domains are supported.
[4] We say that a model is *complete* if it assigns a value to every input variable.

other tests, and deleting such tests from the final suite. This optimization stage is performed a posteriori on the built test suite by a dedicated *reduction* greedy algorithm. Results from empirical studies [32] indicate that as minimization is applied (while keeping constant coverage), there is little or no reduction in the fault detection effectiveness. For this reason, although the reduction stage is optional, it is always applied in this paper.

The process proposed by our method is implemented by the *ASM Test Generation Tool* (ATGT)[5]. ATGT was originally developed to support structural [18] and fault based testing [16] of *Abstract State Machines* (ASMs), and it has been then extended to support also combinatorial testing.

In [4] we discussed the main advantages of our approach with respect to other approaches to CCIT, namely (1) the ability to deal with user specific requirements on the test suite, in form of both specific test predicates representing critical combinations and particular tests already generated, (2) the integration with other testing techniques like the structural coverage presented in [17] and the fault based coverage of [16], (3) the integration with the entire system development process, since the user can initially model only the input and later add also the behavior and other formal properties, and apply other analysis techniques like simulation, formal verification, and model checking, and (4) the complete support of constraints, which can be given as generic boolean predicates over the inputs. Indeed, as discussed again in Sect. 5, most methods for combinatorial testing focus only on finding very efficient algorithms able to generate small test suites, while they normally neglect all the other issues listed above. In [5] we presented an extension of this approach able to deal with temporal constraints.

2.1 Implementing support for constraints by Yices

One important difference introduced in this work over the methodology defined in our previous work [4,5] and summarized in Section 2 is the use of the SMT solver Yices [14] instead of the model checker SAL to support constraints over the input domain. Yices is an efficient SMT solver that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions. With respect to a SAT solver, Yices offers a more expressive language for the specification and for the properties and it uses several algorithms for satisfiability checking. For example, we are able to directly encode constraints over enumerative variables without the burden of their translation to SAT, which is itself an open research problem [31]. We plan in the future to exploit other features of Yices to deal with more complex systems, but for now we simply use booleans and enumerative variables since we apply our tool to case studies taken from other approaches which support only that. With respect to a model checker, Yices cannot deal with state transformations (although it can be used as front end for bounded model checking) and for this reason it cannot be used in the

[5] ATGT is available at http://cs.unibg.it/gargantini/software/atgt/.

```
;; (1) define the types and the variables for the BBS example
(define−type AccessT (scalar LOOP ISDN PBX))
(define−type BillingT (scalar CALLER COLLECT EIGHT_HUNDRED))
(define−type CallTypeT (scalar LOCALCALL LONGDISTANCE INTERNATIONAL))
(define−type StatusT (scalar SUCCESS BUSY BLOCKED))
(define access :: AccessT) (define billing :: BillingT)
(define callType :: CallTypeT) (define status :: StatusT)
;; add the constraints
(assert (=> (= billing COLLECT) (/= callType INTERNATIONAL)))
;; add the test predicate
(assert (and (= access LOOP) (= billing COLLECT)))
;; find and print a model (if any)
(set−evidence! true) (check)
```

```
;; (2) second BBS example using an uniterpreted function modelling the cost
;; Cost as uniterpreted type with a constant for FREE calls
(define−type Cost) (define FREE::Cost)
;; Two Cost functions of billing and of callType
(define CB::(−> BillingT Cost)) (define CCT::(−> CallTypeT Cost))
;; add some constraints about free calls
(assert (= (CB EIGHT_HUNDRED) FREE))
(assert (/= (CCT INTERNATIONAL) FREE))
(assert (= (CB billing) (CCT callType)))
;; add the test predicate
(assert (and (= billing EIGHT_HUNDRED) (= callType INTERNATIONAL)))
;; find and print a model (if any)
(set−evidence! true) (check) ;; −−−> unsat
```

Fig. 1. BBS in Yices (1) base version (2) using uniterpreted functions

presence of temporal constraints, where instead, model checkers can be efficiently employed [5]. Since Yices does not perform model checking and SAL uses Yices as default SMT solver, directly using Yices should be faster than using SAL. Moreover, we can use the Yices API library instead of the more expensive data exchange through files we used for SAL. Experimental work results presented in Section 4 confirmed that Yices is much faster than SAL.

The translation of the logical problem of finding a model for a given problem is straightforward. For instance, the translation of the BBS problem for the test predicate $access = LOOP \wedge billing = COLLECT$ is reported in Fig. 1. As second example, we show how the advanced features of Yices allow to easily model partial specifications by using uninterpreted functions.

While a model checker always finds and outputs a counter example, a SMT solver normally only checks if a formula is satisfiable or not, and it is not mandatory to print the model in case a model exists. Yices can print the model (if any) if explicitly requested by the user (with a set-evidence command). However, the model found by Yices may not bind all the variables: it specifies only the val-

ues of variables whose values are fixed by the found model, leaving the others unspecified. To be used as test, the model must be completed with values for all the variables and this can be done by choosing random values for unbound variables. In our case, the number of unbound variables should be very low, since we compose as many test predicates as possible, so to bind as many variables as possible and to leave free only the variables which are not referred by uncovered test predicates that could be composed. Therefore, the effect of this random completion of tests should negligibly affect the final test suite.

3 Sorting the test predicates

The order in which the test predicates are selected during test generation may affect the size of the final test suite. In our previous work [4], we found that processing the test predicates in random order generally leads to good results, provided that one runs the generation process several times and then he/she takes the best outcome as final test suite. This approach is widely used: existing tools based on greedy, non deterministic search heuristics (i.e. like AETG, SA-SAT, ATGT, PICT[6]) are commonly benchmarked over a series of fifty executions. The main disadvantage of the random based approach, is that the user must run several times the test generation process to be sure to obtain a *statistically* good result. In order to obtain a reasonably short test suite with just one run of the tool, a deterministic construction algorithm can be applied. In this case, the incremental construction algorithm will process the pool of test predicates always in the same order, determined with respect to some optimization criterion. Several orderings can be defined to this aim, based on the observation that in an optimal covering array all the combinations of variables to values assignments are evenly used. In fact, when adding a new test case to the test suite, it should cover as many new combinations as possible, that is, it should include test predicates which give a more original contribution to the test suite. To this aim, we defined several comparison criteria to evaluate which test predicate is more original between two candidates tp_1 and tp_2, given a test suite containing already several test cases. These are:

Counting explored assignments (touch) This counts the number of assignment $variable = value$ contained in the tps which are already contained in a test of the test suite. In this way, tp_1 is preferable if it contains fewer assignment already *touched* in the test suite than tp_2.

Least used assignment (min) In this case, the algorithm keeps track of the number each $variable = value$ assignment has already been included in the test suite, and tp_1 is preferable to tp_2 if its least used assignment is less used than the least used assignment of tp_2. The rationale behind this sorting criteria is that the novelty of a single assignment has here priority over the novelty of the whole test predicate.

[6] The PICT tool core algorithm does make pseudo-random choices but, unless a user specifies otherwise, the pseudo-random generator is always initialized with the same seed value, in order to purposely let two executions of the tool on the same input produce the same output.

Most used assignment (max) As opposite to the former, this comparison criteria prefers tp_1 to tp_2 if its most used combination has lower usage count than that of tp_2. Note that this produce a totally different ordering with respect to the former, and not just its reversal.

Even usage of assignments (dev) In this criteria, a test predicate tp_1 is preferable if the usage of its assignments is more evenly distributed than how it is for tp_2. This is actually quantified by computing the *standard deviation* of the usage counts for the assignments in the considered test predicate. The rationale is that the even usage of combinations, which is a global requirement of a good test suite, can be imposed also locally in each newly added test predicate, and throughout the incremental construction process this can help preventing the introduction of unbalanced test cases.

Accounting for tp composition Four additional ordering strategies have been defined (*touch/c, min/c, max/c, dev/c*), which are variants of their respective original ordering strategies, modified according to the test predicate composition principle, that is, they account also for all the assignments in the composed test predicate, instead of just for those already in the test suite.

4 Evaluation and discussion

In this section a comparison of experimental results is presented, obtained by applying the ATGT tool to a set of example tasks available from the literature [8,9], and listed in the leftmost columns of Table 3. While the tasks #[1..5,11..13] have been generated artificially with increasing size, all other tasks encode example combinatorial problems derived from real systems of various sizes. Third and fourth column of Table 3 report the input domain size and the complexity of the imposed constraints. The notation used to express the problem domain size is the exponential notation introduced in [21], while the constraints complexity is expressed by converting the constraints into DNF and then apply the following function δ:

$$\delta(a \wedge b) = \delta(a) \cdot \delta(b) \qquad \delta(a \vee b) = \delta(a) + \delta(b)$$
$$\delta(x = b) = range(x) - 1 \qquad \delta(x \neq b) = 1$$

For forbidden combinations, δ is equal to the constraints measure proposed in [9], which simply counts a forbidden combination of t variables as t and multiply all the forbidden combinations. For instance, the forbidden pair $x = a, y = b$, would be represented in our approach by the constraint $\neg(x = a \wedge y = b)$, which in DNF becomes $x \neq a \vee y \neq b$ which is evaluated by δ to 2. In case of forbidden combinations, quantities expressed with this criteria can take advantage of exponential layout, e.g., $2^5 \cdot 3^1$ will read also as five pairwise constraints plus one tree-wise. All the specifications shown in Table 3 contains constraints easily expressed as forbidden combinations, except task#8, which has only boolean variables and contains a single complex boolean constraint, which converted to DNF has 224 conjunctions, so the complexity is 224.

Table 3. Suite sizes and average times for random heuristic.

#	task: name	size	δ	size: *min*	*avg*	*1stQ*	*3rdQ*	*max*	time: *Yices*	*SAL*
1	CCA1	3^3	$2^5 3^1$	9	9.43	9	10	11	0,89	9,7
2	CCA2	4^3	$2^3 3^1$	17	19.16	19	20	22	1,43	15,58
3	CCA3	5^3	$2^5 3^1$	27	30.06	29	31	34	2,23	23,77
4	CCA4	6^3	$2^6 3^1$	40	42.92	42	44	47	3,13	33,67
5	CCA5	7^3	$2^5 3^1$	55	59.74	59	61	64	4,17	45,49
6	MobilePhone	$3^3 2^2$	$2^5 3^1 5^1$	10	11.48	11	12	14	1,8	19,95
7	CruiseControl	$4^1 3^1 2^4$	2^2	8	8.41	8	9	10	1,07	11,68
8	TCAS2Boolean	$10^2 4^1 3^2 2^7$	224	10	11.15	11	11	13	4,9	57,74
9	BBS	3^4	2^1	12	12.98	12	14	15	1,08	11,89
10	SpinSimulator	$4^5 2^{13}$	$2^{47} 3^2$	26	29.45	29	30	33	12,49	137,09
11	CCA6	5^4	$2^3 3^1$	34	36.76	36	38	40	3,15	32,66
12	CCA7	6^4	$2^3 3^1$	49	52.81	52	54	57	4,46	46,3
13	CCA8	7^4	$2^5 3^1$	68	73.14	72	74	79	6,09	70,36

A set of fifty instances of the test suite based on the policy of random[7] processing order have been generated for each of the tasks, and Table 3 reports the resulting best, average, and worst test suite size obtained, together with the values of the first and third quartiles computed from the gathered set of sizes. Table 3 also reports the computing times for the considered tasks when using Yices or SAL respectively. They show a performance improvement by a factor of about eleven times using Yices over SAL model checker. Although these computing times correspond to the random policy experiments only, the computing times observed for the other policies were similar, irrespective of the considered processing policy, and the performance improvement observed in all experiments has shown to be constant, irrespective the task too. Thus, we decided not to report them here.

All the previously introduced deterministic ordering policies have also been applied, and the resulting test suite sizes are reported in Table 4. In the performed experiments an additional deterministic ordering policy has been applied too, which consisted in processing all the test predicates just in the same order they where enumerated by straight nested loops. This dummy ordering policy, named *as generated (asg)* has been included to have a scenario where no ordering is applied at all and the corresponding outcomes are in the rightmost column of Table 4. All results reported in tables 3 and 4 are intentionally reported prior to eventual application of the suite reduction stage. Indeed, applying reduction would have improved the results but could have also masked the relative performance differences between the policies.

Figure 2 allows the reader to visually compare altogether the data in both tables 4 and 3, and especially to figure out how the considered deterministic policies perform with respect to random processing, before applying the test

[7] An uniform distribution among test predicates has been applied.

Table 4. Comparison of suite sizes for deterministic heuristics.

task#	*min*	*min/c*	*touch*	*touch/c*	*max*	*max/c*	*dev*	*dev/c*	asg
1	10	9	9	9	10	11	9	10	10
2	20	21	18	18	21	20	21	21	21
3	36	33	31	30	33	35	32	32	37
4	58	52	46	45	47	48	49	46	58
5	77	68	63	63	70	66	67	65	80
6	11	14	11	10	10	12	11	11	13
7	9	10	9	9	10	9	9	9	9
8	12	12	12	10	12	14	12	13	15
9	18	15	13	14	20	14	14	15	20
10	39	31	29	28	37	31	31	33	40
11	45	40	37	38	42	39	42	41	55
12	68	59	58	54	63	61	64	62	81
13	106	88	79	79	92	81	83	81	123

suite reduction algorithm. On the horizontal axis are the task numbers, split in two graphs with different scaling of the y axis, for improved readability.

While random policy has always reached better (smaller) sizes than all the considered deterministic policies, it is interesting to note that its worst (bigger) performance is also worse than many of the proposed deterministic policies in all the tasks in Figure 2(a), and in tasks #{3,4,5,10} of Figure 2(b). In many tasks there have been deterministic strategies performing even better than average random result, like i.e. in tasks # {1,2,7,8,3,4}, or comparable to average, like i.e. tasks #{9, 3, 5}. It can be observed that the *touch/c* processing policy is constantly the best performing among all the observed deterministic policies, with the exception of tasks #9 and #5, where it is only slightly outperformed by its sibling policy *touch*. Also, it is interesting to note that the *touch/c* policy is always performing better that the worst random policy performance in all tasks, with the sole exception of task #13, where they are equal, and as good as the best random result, in tasks #{1,7,8}. It is relevant to note that the time cost to achieve the best random performance has to be multiplied by the number of runs necessary to obtain it. This result encourages the use of a deterministic processing strategy, like *touch/c* over the random based alternative, at least in specific cases where time performance is a strict priority over the test suite size optimization.

Figure 2(b) also shows that the performance of deterministic policies degrades faster with respect to random policy average, when scaling up the task size. Note that the performances of the non deterministic strategy span in an interval which become wider for big specifications, from the best performance to the worse performance possible since such strategy has no constraints except those theoretical. The performance of a deterministic strategy will fall in this interval, but it will never perform better than the best result of the random policy. For this reason finding a deterministic strategy which performs at least better than the average is a challenging activity. Indeed, there is no guarantee about the

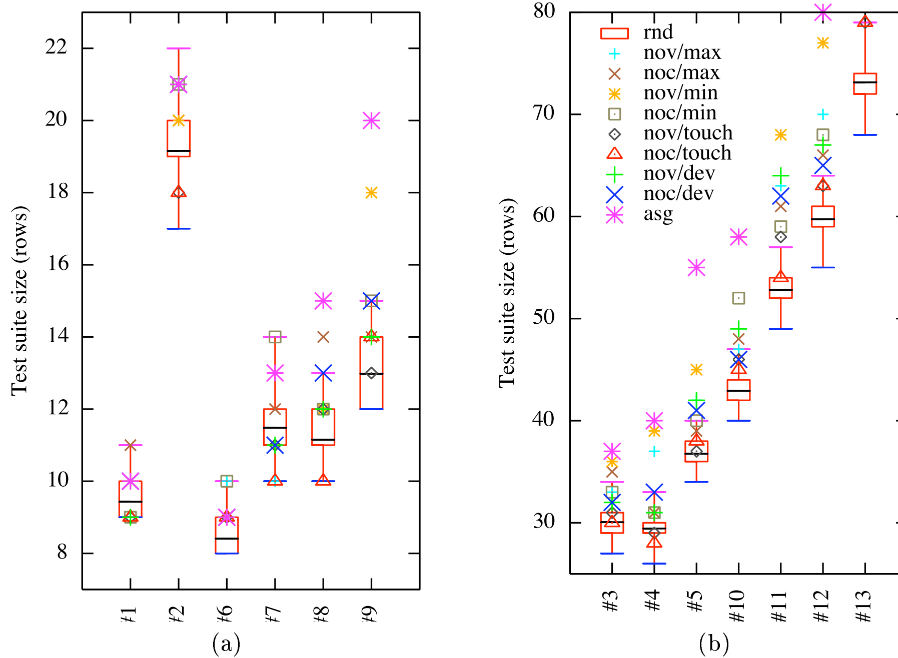

Fig. 2. Suite sizes computed for tasks #1-13, prior to reduction.

quality of a single performance of a non deterministic strategy, requiring for this reason multiple runs. After the application of reduction stage, the performance gap reduces significantly, as shown in Figure 3, which compares the suites sizes of random and the best deterministic policy. In this case, for all considered tasks the performance of deterministic is never worse than the random worst result, and in nine out of thirteen tasks is better than average random result, supporting the adoption of deterministic heuristic as a fairly good option. Finally, in Table 5 we contrasted the best results of our approach, from both random and best deterministic ordering policy, with those of some tools from the literature. The mAETG-SAT [8] and PICT [10] tools are AETG-like [6] greedy construction algorithms, with a random-heuristic based core, even though the PICT tool can be forced to behave deterministically by initializing its internal random seed always with the same constant value. The SA-SAT [8] tool is a meta-heuristic search algorithm inspired on simulated annealing [7], which exhibit random behavior too. Testcover [30][28] tool builds a test suite by recursively composing together sub-blocks which are already optimal orthogonal arrays or covering arrays. This latter is the sole tool which has no random heuristic inside. The results from our tool ATGT are here reported inclusive of the ex-post reduction optimization, showing that both for pairwise and for three-wise combinatorial tasks the difference between random and deterministic performance is always moderate, and

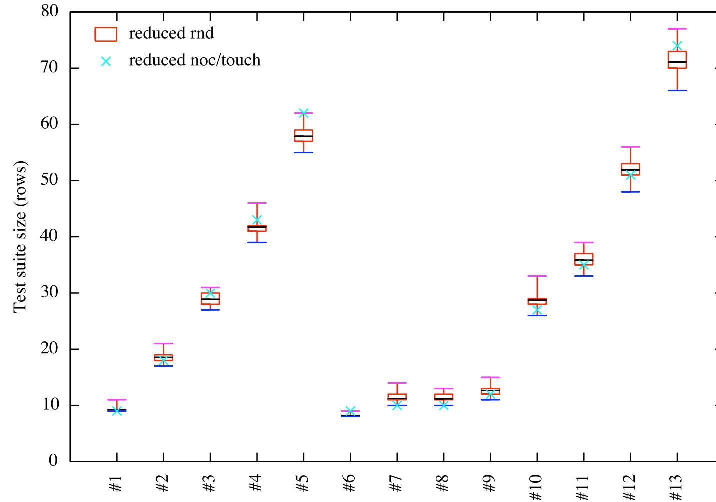

Fig. 3. Reduced suite sizes, of random and touch policies, for tasks #1-13.

often negligible. The shown performance compares also fairly good with that of the other available tools.

5 Related work

To the best of our knowledge, little work has already be done in literature about the ordering of test goals for test generation. In [15], the authors show that taking the test predicates in the same order in which they are built is generally a bad policy, while randomly choosing them leads to good results. They also define some ordering criteria, which however are not suitable to combinatorial testing. Although their approach differs with respect to that presented here, since they generate one test case for each test predicate while we collect as many test predicates as possible to generate one test, our experiments confirm that a random order is better than the order in which test predicates are generated. In [2], 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. The major contribution of this study is then the evaluation of the original strategies presented in Sect. 3, which actually implement variants of a novel *hybrid* heuristic based on defining

Table 5. Comparison of suite sizes for 2-wise and 3-wise constrained models.

Task #	ATGT $touch/c^{red}$	rnd^{red}_{min}	mAETG-SAT	SA-SAT	PICT	TestCover
			t = 2, pair wise			
1	9	9	10	10	10	10
2	18	17	17	17	19	17
3	30	27	26	26	27	30
4	45	40	37	36	39	38
5	63	55	52	52	56	54
10	28	26	25	24	16	19
			t = 3, three wise			
10	135	127	108	95		
11	164	159	138	140	143	-
12	283	282	241	251	250	-
13	502	449	383	438	401	-

ordering selection criteria for the test predicates, instead. Building the next row around a test predicate means that both a set of *fixed* factors and their levels (values) will be determined at the same time, in contrast with Bryce et al. study which focused on separate rules for the factor and level selection layers. 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, all the strategies proposed and tested actually implement this common optimization principle of controlling the *density* of feature levels, but we explored original ways of redefining the *density* concept. In fact, while Bryce et al. compute it as the expected number of uncovered pairs, in contrast we define many measures somewhat related to the current frequency of appearance of each predicate in the test suite, and lexicographically sort with respect to that.

As a second aspect, research on combinatorial testing approaches featuring support for constraints deserves further investigation. Some methods require to remodel the original specification, very few directly support constraints in an integrated manner. For instance, AETG [6,26] requires to separate the inputs in a way they become unconstrained, and only simple constraints of type **if then else** (or **requires** in [8]) can be directly modeled in the specification. Other methods [20] require to explicitly list all the forbidden combinations. As the number of input grows, the explicit list may explode and it may become practically infeasible to find for a user. Cohen et al. [8] found that just one tool, PICT [10], was able to handle *full* constraints specification, that is, without requiring remodeling of inputs or explicit expansion of each forbidden test cases. However, there is no detail on how the constraints are actually implemented in PICT, limiting the reuse of its technique. Others propose to deal with the constraints only after the test suite has been generated by deleting tests which violate the constraints and then regenerate the missing combinations. By this approach, any classical algorithm for CIT may be extended to support constraints. However,

this is usable only if the number of combinations to be regenerate is small and all the constraints are explicitly listed. Experiments show that the number of tests violating the constraints can be very high if they are simply ignored, and the number of implicit constraints can exponentially grow with the number of variables [9]. In our work we address the use of full constraints as suggested in [8] and a more expressive language for them through the expressions supported by Yices.

Several papers recently investigated the use of verification methods for combinatorial testing. Hnich et al. [22] translates the problem of building covering arrays to a Boolean satisfiability problem and then they use a SAT solver to generate their solution. In their paper, they leave the treatment of auxiliary constraints over the inputs as future work. Note that in their approach, the whole problem of finding an *entire* covering array is solved by SAT, while in our approach only the generation of a single test case is solved by Yices. To this respect, our approach is similar to that presented by Cohen et al. [8,9], where a mix of logical solvers and heuristic algorithms is used to find the final test suite. Kuhn and Okun [23] try to integrate combinatorial testing with model checking (SMV) to provide automated specification based testing, with no support for constraints. Conversely, Cohen et al. propose a framework to incorporating constraints into established greedy and simulating annealing combinatorial testing algorithm. They exclusively focus on handling constraints and present a SAT-based constraint solving technique that has to be integrated with external algorithms for combinatorial testing like IPO or AETG. Their framework is general and fully supports the presence of constraints, even if they can be modeled only in a canonical form of boolean formulae as forbidden tuples.

6 Conclusion

In this paper we have defined a pool of eight metrics to sort the test predicates prior to processing, in order to assess the impact of their processing order on the size of the resulting test suite and on the generation time. The investigation of such aspect of deterministic combinatorial construction algorithms is an original contribution. The results have been contrasted with those of random and dummy orderings and also with available results from some well-known combinatorial tools. It has been shown that even though random based heuristics can achieve better (lower) absolute results in terms of the size of the computed test suite, the performance of deterministic heuristics is still acceptable, it does not require multiple runs as the random policy, and thus it is preferable if the computing time requirements are an issue. In order to support our study we implemented a combinatorial construction technique that supports constrained combinatorial testing, by using the Yices SMT solver in order to generate models. The proposed approach is able to support not just pairwise but also n-wise CCIT, and the presented comparative evaluation with respect to other existing tools suggest that the presented methodology is fairly good approach to CCIT. To the best of our knowledge, this is also the first approach to CCIT exploiting an existing SMT

solver. Work is undergoing to integrate this technique with structural and fault based testing, and to extend it in order to support constraints with universal and existential quantifications, which would be very useful to express complex constraints in a very compact style.

References

1. 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.
2. R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 146–155, New York, NY, USA, 2005. ACM.
3. K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513, October 1998.
4. A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In B. Beckert and R. Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2008.
5. A. Calvagna and A. Gargantini. Using SRI SAL model checker for combinatorial tests generation in the presence of temporal constraints. In J. Rushby and N. Shankar, editors, *AFM'08: Third Workshop on Automated Formal Methods (satellite of CAV)*, pages 43–52, 2008.
6. 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.
7. M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting simulated annealing to build interaction test suites. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, page 394, Washington, DC, USA, 2003. IEEE Computer Society.
8. M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA International symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007.
9. M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: a greedy approach. *IEEE Transactions on Software Engineering*, to appear, 2008.
10. J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, 2006.
11. S. Dalal, A. Jain, N. Karunanithi, J. Leaton, and C. Lott. Model-based testing of a highly programmable system. In *Software Reliability Engineering, The Ninth International Symposium on*, pages 174–179, Los Alamitos, CA, USA, 1998. IEEE Computer Society.
12. S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *International Conference on Software Engineering ICSE*, pages 285–295, New York, May 1999. Association for Computing Machinery.
13. I. S. Dunietz, W. K. Ehrlich, B. Szablak, C. Mallows, and A. Iannino. Applying design of experiments to software testing. In I. Society, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 205–215, 1997.

14. B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI Available at http://yices.csl.sri.com/tool-paper.pdf, 2006.
15. G. Fraser, A. Gargantini, and F. Wotawa. On the order of test goals in specification-based testing. *Journal of Logic and Algebraic Programming*, In press, Available online, 2009.
16. A. Gargantini. Using model checking to generate fault detecting tests. In *International Conference on Tests And Proofs (TAP)*, number 4454 in Lecture Notes in Computer Science (LNCS), pages 189–206. Springer, 2007.
17. A. Gargantini and E. Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *JUCS*, 10(8), Nov 2001.
18. A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from ASM specifications. In *ASM 2003 - Taormina, Italy, March 2003. Proceedings, LNCS 2589*, 2003.
19. M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
20. A. Hartman. Ibm intelligent test case handler: Whitch.
21. A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *DMATH: Discrete Mathematics*, 284(1-3):149–156, 2004.
22. B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.
23. D. R. Kuhn and V. Okum. Pseudo-exhaustive testing for software. In *SEW '06: IEEE/NASA Software Engineering Workshop*, volume 0, pages 153–158, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
24. 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.
25. 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.
26. C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *A-MOST '05: Proceedings of the 1st international workshop on Advances in model-based testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.
27. Pairwise web site. http://www.pairwise.org/.
28. G. B. Sherwood. Optimal and near-optimal mixed covering arrays by column expansion. *Discrete Mathematics*, 308(24):6022–6035, 2008.
29. B. D. Smith, M. S. Feather, and N. Muscettola. Challenges and methods in validating the remote agent planner. In C. Breckenridge, editor, *Fifth International conference on Artificial Intelligence Planning Systems (AIPS)*, 2000.
30. TestCover tool. http://www.testcover.com/.
31. T. Walsh. SAT v CSP. In R. Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.
32. W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software*, 48(2):79–89, 1999.
33. C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Software Eng*, 32(1):20–34, 2006.