Combinatorial Test Generation For Software Product Lines Using Minimum Invalid Tuples

Linbin Yu, Feng Duan, Yu Lei
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX 76019, USA
{linbin.yu, feng.duan}@mavs.uta.edu, ylei@cse.uta.edu

Raghu N. Kacker, D. Richard Kuhn
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{raghu.kacker, kuhn}@nist.gov

***Abstract*—A software product line is a set of software systems that share some common features. Several recent works have been reported that apply combinatorial testing, a very effective testing strategy, to software product lines. A unique challenge in these efforts is dealing with a potentially large number of constraints among different features. In this paper, we propose a novel constraint-handling strategy that uses minimum invalid tuples (MITs) as an alternative to traditional constraint solvers. Our approach systematically derives all MITs from a software product line, and uses them to quickly determine the validity of a test configuration during test generation. We implemented a test generation research tool called LOOKUP that integrates the proposed constraint-handling strategy with a general test generation algorithm called IPOG-C. Experimental results show that LOOKUP performs considerably better than two existing test generation tools in terms of test size and execution time.**

Keywords—Feature Model; Combinatorial Testing; Constraint Handling

I. INTRODUCTION

As an emerging software development paradigm, software product lines [1] have been adopted by many companies. A software product line is a set of software systems that share a set of common features. Different configurations of a software product line are typically represented by a feature model [2], in which a compact tree structure is used to capture the relationships among different features. Such relationship must hold in order to create a valid product configuration. There are four types of relationships, i.e., *mandatory*, *optional*, *or*, and *alternative*. Furthermore, a feature model may include cross-tree constraints that are explicitly specified by the user.

Fig. 1 shows an example feature model with 13 features drawn by a tool named FeatureIDE [3]. In Fig. 1, each node represents a feature which can be configured as either *included* (*true*) or *excluded* (*false*). Restrictions or constraints on which features can be combined with each other are denoted using different notations in the tree. The root feature Aircraft is always included. The root contains three sub-features, in which Wing and Materials are *mandatory*. It means that these features must be included. Feature Engine is *optional*, which means it can be either included or excluded. Detailed notations of the feature model will be explained later.

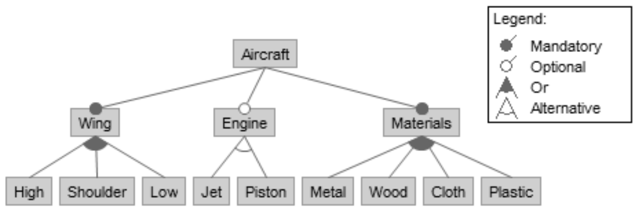
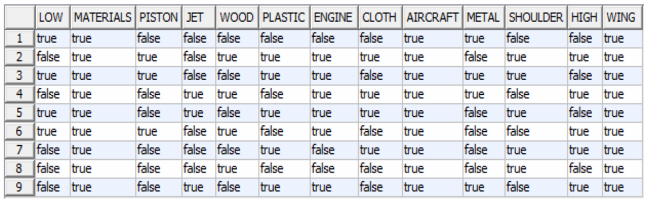


Fig. 1. An example feature model

Since the number of all configurations increases exponentially with the number of features, it is often impractical to test all possible configurations exhaustively. Several recent works have been reported that apply combinatorial testing to software product lines. Combinatorial testing has been shown to be a very effective strategy for general software testing [4] [5] [6]. Given a system with n parameters, t-way combinatorial testing or simply t-way testing, where *t* is referred to as test strength, requires that all t-way combinations that consist of *t* parameter values be covered by at least one test. A widely cited NIST study suggests that software faults in practical applications are typically caused by interactions between only a few parameters, usually no more than 6 [7].

Consider the feature model in Fig. 1. Assume *t* is 2. We need to cover all possible configurations for all 2-way feature groups such as {Aircraft, Wing}, {Aircraft, Engine} and {Engine, Materials}. The number of 2-way feature groups is $C_{13}^{2} = 78$. Each 2-way group has 4 configurations. So the total number of different 2-way configurations is 78*4 = 312. However, some configurations may not be allowed by the feature model constraints. Those configurations are invalid and should not be covered. A test set shown in the following Fig. 2 covers all valid 2-way configurations of the feature model shown in Fig. 1.



	LOW	MATERIALS	PISTON	JET	WOOD	PLASTIC	ENGINE	CLOTH	AIRCRAFT	METAL	SHOULDER	HIGH	WING
1	true	true	false	false	false	false	false	false	true	true	false	false	true
2	false	true	true	false	true	true	true	true	true	false	true	true	true
3	true	true	true	false	false	true	true	false	true	true	true	false	true
4	false	true	false	true	true	false	true	true	true	true	false	true	true
5	true	true	false	true	false	true	true	true	true	false	true	false	true
6	true	true	true	false	true	false	true	false	true	false	false	true	true
7	false	true	false	false	false	true	false	true	true	false	true	true	true
8	false	true	false	false	true	false	false	false	true	false	true	false	true
9	false	true	false	true	false	true	true	false	true	true	false	true	true

Fig. 2. An example 2-way test set

Assuming that test parameters are modeled properly, faults involving at most *t* parameters are guaranteed to be exposed by t-way testing. Pairwise testing is a special case of t-way testing where t is 2. Exhaustive testing is also a special case of t-way testing where t equals the number of parameters. To apply t-way testing on a feature model, one common approach is to model each feature as a Boolean parameter, where true (or false) indicates that a feature is included (or excluded) in a test configuration. Moreover, a feature model imposes restrictions on which features can be combined with each other. These restrictions need to be modeled as constraints, which are usually handled by constraint solvers like SAT solvers during test generation.

Compared to general software systems, software product lines have two unique characteristics. First, test parameters derived from a feature model are Boolean parameters. Second, a large number of constraints are often derived from a feature model. Constraint handling can be a compute-intensive process especially when there are a large number of constraints. These two unique characteristics can be exploited to optimize the performance of the test generation process.

In this paper, we present an approach that uses the notion of minimum invalid tuples (MITs) to handle constraints. One important task of constraint handling is validity checking, i.e., to check whether a test configuration violates any constraint. We first formally define the notion of MIT and report an efficient algorithm that systematically derives all possible MITs from a feature model. These MITs represent the same constraint space as the feature model tree notation, and can be used to quickly determine the validity of a test configuration, i.e., a test is valid if and only if it contains no MIT. This approach is different from traditional constraint-solving approaches such as the ICPL algorithm [8]. In traditional constraint-solving approaches, numerous solving processes could be performed during test generation, but they are almost independent with each other and very little information can be shared among different constraint solvings. In contrast, the process of MIT generation works as a preprocess step before test generation. Once all MITs are found, validity checking can be performed in a very efficient way, i.e., checking if a test contains any known MIT. The performance of MIT generation highly depends on the complexity of a feature model. If the model contains a large number of MITs, the constraint solving approach may perform better.

We built a test generation tool called LOOKUP [9] that integrates our constraint-handling approach with a general t-way test generation algorithm called IPOG [10]. LOOKUP can be downloaded at [9]. In order to evaluate the proposed constraint-handling strategy and the LOOKUP tool, we use 12 largest feature models from the SPLOT feature model repository [11] as subject systems. The number of features in these 12 models ranges from 71 to 290. We compared our tool with two existing tools, including a general combinatorial test generation tool called PICT [12], and a feature model-specific test generation tool called SPLCA [13]. Experimental results show that our tool performed considerably better than SPLCA and PICT in terms of test set size and execution time.

The rest of this paper is organized as follows. Section II gives some background knowledge about feature models and constrained combinatorial test generation. Section III introduces the notion of MIT. Section IV presents the complete test generation algorithm for feature models. Section V reports experimental results. Section VI discusses related work. Section VII concludes this paper and discusses future work.

II. PRELIMINARIES

In this section, we give some formal definitions that are used by our approach.

Definition 1 (Feature) A feature p is a Boolean variable where true (or false) indicates this feature is included in (or excluded from) a test configuration.

For ease of notation, we use p to denote that the value of feature p is true, and $!p$ to denote that the value of feature p is false, when there is no ambiguity. In the rest of this paper, we assume that a feature model contains a set P of n features, i.e., $P = \{p_1, p_2, \ldots, p_n\}$.

Definition 2 (Test Configuration) A test configuration is a function that assigns a Boolean value to each feature. Formally, a test configuration is a function f: $P \rightarrow \{\text{true}, \text{false}\}$.

A test configuration represents a specific version in a software product line.

Definition 3 (Tuple) A tuple u is a test configuration f restricted to a subset of features. Formally, $u = f \mid M$, where $M \subseteq P$.

A tuple is a set of feature configurations. We will use $dom(u)$ to represent the domain of u, which is the set of features involved in u. We define the size of a tuple as the number of features in $dom(u)$. A tuple of size t is also denoted as a t-tuple. A tuple u can also be considered as a set of values $\{u(p_i) \mid p_i \in dom(u)\}$. For example, assume a feature model contains 3 features {a, b, c}. A 2-tuple {a, !b} represents a partial configuration in which feature a is selected, and b is not selected. We will use this notation in the rest of this paper.

Note that a tuple contains at most one value for the same feature, since otherwise the feature configuration is not meaningful. For example, {a, b, !a} is not a meaningful configuration since it contains contradicting values a and !a.

Definition 4 (Containment) A tuple u is said to be contained (or covered) by another tuple u', denoted as $u \subseteq u'$, if and only if $dom(u) \subseteq dom(u')$ and $\forall$ p $\in$ dom(u), u(p) = u(p').

Definition 5 (Constraint) A constraint c is a function: F $\rightarrow$ {true, false} that maps a test configuration to true or false.

A constraint is in essence a restriction that must be satisfied when different features are combined to create a product configuration. A constraint may be explicitly defined using logic expressions, or may be implicitly encoded by the feature model tree structure. A valid software product (test configuration) must satisfy all constraints.

Definition 6 (Feature Model) A feature model M = <P, C> consists of a set of features $P = \{P_1, P_2, \ldots, P_n\}$, and a set of constraints $C = \{c_1, c_2, \quad, c_r\}$.

Constraint can be represented in different ways. One may use logic expressions to specify constraints, or use a list of unwanted combinations explicitly.

Definition 7 (Configuration Validity) A test configuration *f* of a feature model M is valid if and only if *f* satisfies all the constraints of M, i.e., $\forall c \in C, c(f) = \text{true}$.

Definition 8 (Tuple Validity) A tuple *u* is valid if it can be contained by a valid test *f*. Otherwise *u* is invalid.

A tuple is valid implies that it can be extended to a valid test. Otherwise, it is an invalid tuple.

Definition 9 (T-way Test Set). Let M = <P, C> be a feature model. Let Σ be the set of all valid t-tuples. A t-way test set is a set Ω of tests such that, $\forall \sigma \in \Sigma, \exists \tau \in \Omega$ such that τ is valid and $\sigma \subseteq \tau$.

Intuitively, a t-way test set is a set of valid test configurations such that each valid t-tuple is covered by at least one valid test.

III. Minimum Invalid Tuples

In this section, we introduce the notion of minimum invalid tuples (MITs) that can be used for validity checking. We first discuss how to represent constraints in a feature model using a set of invalid tuples. Then we formally define MIT and explain how MIT can be used for validity checking. Last, we present an algorithm that can effectively generate all MITs from a feature model.

A. Invalid Tuples in Feature Model

As mentioned earlier, constraints can be represented by unwanted combinations, i.e., invalid tuples. An invalid tuple is a tuple that is not allowed to appear in a test configuration. Thus an invalid tuple is equivalent to a conjunctive normal form (CNF) constraint. For example, an invalid tuple {a, !b} means a product cannot include feature a when feature b is excluded. This is equivalent to a logic expression "$\neg(a \wedge \neg b)$" or "$\neg a \vee b$", i.e., either feature *a* is excluded or feature *b* is included. Similarly, an invalid tuple {a, !b, c, d} is equivalent to "$\neg a \vee b \vee \neg c \vee \neg d$". Constraints encoded in feature model can be easily converted to invalid tuples. Fig. 3 summarizes all 6 types of relations in a feature model and the equivalent invalid tuples. A special case is that, the root feature is always true in order to make a test configuration meaningful.

- **Optional relation**: A parent feature *p* must be true if any child feature *c* is true.
- **Mandatory relation**: A child feature *c* must has the same value as its parent.
- **Or relation**: A parent feature *p* must be true if any child feature c_i is true; at least one child feature is true if the parent feature *p* is true.
- **Alternative relation**: A parent feature *p* must be true if any child feature c_i is true; at most one child feature can be true.
- **Require**: The selection of feature *a* requires the selection of feature *b*.
- **Exclude**: Features *a* and *b* cannot be both true.

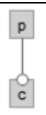
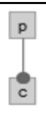
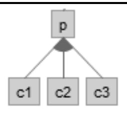
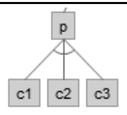
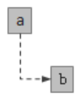
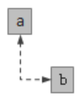
Given a feature model, we can find a set of invalid tuples, denoted as *input invalid tuples*, using these rules.

B. Minimum Invalid Tuples

We define the notion of a minimum invalid tuple (MIT).

Definition 10 (Minimum Invalid Tuple) A minimum invalid tuple (MIT) is an invalid tuple that can not contain any other invalid tuple.

Intuitively, a MIT is an invalid tuple of minimum size. That is, an MIT will become valid if any element is removed from this tuple. This also means that, given any invalid tuple *u*, we can generate a MIT $u' \subseteq u$. Note that an invalid tuple may contain more than one MIT.

Type	Optional	Mandatory	Or	Alternative	Require	Exclude
Notation						
Semantics	$c \rightarrow p$	$p \rightarrow c$ $c \rightarrow p$	$c_i \rightarrow p$ $p \rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n)$	$c_i \rightarrow p$ $\neg(c_i \wedge c_j)$	$a \rightarrow b$	$\neg(a \wedge b)$

Invalid Tuples	{!p, c}	{p, !c}
		{!p, c}
			{!p, c_i}
			{p, !c_1, !c_2, ...!c_n}
				{!p, c_i}
				{c_i, c_j}
					{!b, a}	{a, b}

Fig. 3. Invalid Tuples in Feature Model

From Definition 10, we have an important observation: **A tuple is valid if and only if it contains no MIT.** If a tuple u contains no MIT, then it also contains no invalid tuples. Otherwise we can generate at least one MIT from an invalid tuple, which contradicts our assumption. This observation suggests a new approach of validity check, that is, checking if a tuple contains any MIT. The main challenge of this approach is to generate all possible MITs from a feature model, which will be discussed in the next section.

C. *The MIT Generation Algorithm*

Generating all MITs is an important step. In this section we propose an effective algorithm that can generate all the MITs from a feature model. As discussed in Section III.A, constraints in a feature model can be represented using a set of invalid tuples, which we refer to as the set of *given invalid tuples*. Obviously, a tuple that contains any given invalid tuple must be invalid. However if a tuple contains no given invalid tuple, it may still not be valid. For example, assuming we have 2 input invalid tuple {a, b} and {!b, c}. A tuple {a, c} is invalid even it does not contain {a, b} or {!b, c}. This is because tuple {a, c} cannot be extended to a valid test: if we extend it by adding *b*, then the resulting tuple contains the first input invalid tuple; if we extend it by adding !*b*, then the other input invalid tuple will be contained. This example shows that input invalid tuples cannot be used directly for validity checking. Note that generating all MITs from input invalid tuples is similar to find prime implicants from CNF/DNF formulas.

In order to generate all MITs from a set of invalid tuples, we first show an operation that can derive a new invalid tuple from two invalid tuples.

Derivation Rule: Given two invalid tuples u and u', if there exists exactly one feature p for which one of the two tuples contains true and the other contains false, then $v = (u \cup u') \setminus \{p, !p\}$ is a new invalid tuple.

```
Algorithm: Generate-All-MITs
Input: a set $I$ of input invalid tuples $u_1, u_2, \dots u_n$
Output: a set $S$ consisting of all MITs that can be derived
from $I$

1.  initialize $S = I$
2.  do{
3.      let $S' = S$ and $E = \varnothing$
4.      for each pair of invalid tuples $(u, u')$ in $S$ {
5.          if (a new tuple $v$ can be derived from $u$ and $u'$
               using the Derivation Rule)
6.              $E = E \cup \{v\}$
7.      }
8.      $S = S \cup E$
9.      for each invalid tuple $u$ in $S$ {
10.         if ($u$ contains another tuple in $S$)
11.             $S = S \setminus \{u\}$
12.     }
13. }
14. while ($S \neq S'$)
15. return $S$
```

Fig. 4. Algorithm Generate-All-MITs

The reason is simple: we cannot add p or $!p$ into v since otherwise the resulting tuple must contain either u or u', which is invalid. For example, from invalid tuples {a, b} and {!b, c}, we can derive a new invalid tuple {a, b, !b, c}\{b, !b} = {a, c}. The newly derived tuple is also an invalid tuple. Inspired by the derive operation, we propose an effective algorithm shown in Fig.4 that can derive all MITs from a set of invalid tuples.

The algorithm starts with all input invalid tuples (line 1), then it tries to derive new invalid tuples from existing invalid tuples, and then adds them to set S (lines 4 to 9). In the next step, a tuple that contains another tuple in S is removed from S (lines 10 to 14). The deriving and removing processes are repeated until set S converges. At last, set S consists of all MITs that can be derived from input invalid tuples. An example shown in Fig 5 illustrates each step of MIT generation. The input invalid tuples are {a, b}, {a, c}, {!b, !c} and {b, !c, d}, and all the MITs are {!b, !c}, {!c, d} and {a}.

Input	**M = {{a, b}, {a, c}, {!b, !c}, {b, !c, d}}**
Iteration1, derive	{a, b} ∪ {!b, !c} \ {b, !b} = {a, !c} {a, c} ∪ {!b, !c} \ {c, !c} = {a, !b} {a, c} ∪ {b, !c, d} \ {c, !c} = {a, b, d} {!b, !c} ∪ {b, !c, d} \ {b, !b} = {!c, d} **E = {{a, !c} , {a, !b} , {a, b, d}, {!c, d} }**
Iteration1, remove	M = {{a, b}, {a, c}, {!b, !c}, ~~{b, !c, d}~~, **{a, !c} , {a, !b} ,** ~~**{a, b, d}**~~**, {!c, d}**}
Iteration2, derive	{a, b} ∪ {a, !b} \ {b, !b} = {a} {a, c} ∪ {a, !c} \ {c, !c} = {a} {a, c} ∪ {!c, d} \ {c, !c} = {a, d} **E = {{a}, {a, d}}**
Iteration2, remove	M = {~~{a, b}~~, ~~{a, c}~~, {!b, !c}, ~~{a, !c}~~ , ~~{a, !b}~~, {!c, d}, **{a},** ~~**{a, d}**~~}
Iteration3, derive	No new tuples can be derived. **M = {{!b, !c}, {!c, d}, {a}}** are all MITs.

Fig. 5. An example of generating all MITs

This algorithm is guaranteed to generate all MITs that can be derived from input invalid tuples. It is not hard to see that at any time, S represents the same constraints as represented by the set of I of invalid tuples. In line 8, a set E of invalid tuples is added into S. An invalid tuple in E is in essence a tuple that can be derived from two existing tuples in S, thus $S \cup E$ still represents the same constraints. In line 11, we remove invalid tuples that contain another invalid tuple in S. It is easy to see that S still represents the same constraints, since the removed tuples implied by existing formulas in S. Thus the final set S represents the same constraints as inputs. Furthermore, the removing step (line 9 to 11) guarantees all tuples in S are MITs.

IV. The Test Generation Algorithm

In this section, we present the complete t-way test generation algorithm named FMTG (Feature Model Test Generation) for feature models. The pseudo-code is shown in Fig. 6.

The FMTG algorithm contains two major parts, i.e., MIT generation and test generation. For MIT generation, we first convert a feature model into a parameter model while features are modeled as Boolean parameters and constraints are modeled as invalid tuples (Section III.A). Then we generate all MITs using the Generate-All-MITs (Section III.C). These MITs are then used for validity checking.

```
Algorithm: FMTG (Feature Model Test Generation)
Input: feature model $M$, test strength $t$
Output: a t-way test set $S$ for $M$

1.  model every feature in $M$ as a Boolean parameter
2.  model every constraint in $M$ as a set of invalid tuples $I$
3.  generate the set $I_m$ of all the MITs from $I$
4.  sort all the parameters in a non-increasing order of the number
        of their appearances in $I_m$ and denote them as $P_1, P_2, \ldots, P_n$
5.  find a valid t-way test set $S$ for the first $t$ parameters
6.  for ($i$ from $t$+1 to $n$) {
7.      let $\pi$ be the set of all the valid t-tuples involving parameter
            $P_i$ and any $t$-1 parameters before $P_i$
8.      for each partial test $\tau$ in $S$ { //horizontal growth
9.          add a value $v_i$ for $P_i$ such that the resulting test contains
                no MIT and covers the most uncovered t-tuples in $\pi$
10.         remove from $\pi$ the covered t-tuples
11.     } //finish horizontal growth
12.     for each t-tuple $\sigma$ in $\pi$ { //vertical growth
13.       if $\sigma$ contains any MIT, remove $\sigma$ from $\pi$
14.       else {
15.           cover $\sigma$ by adding new values to an existing test or
                  adding into $S$ a new partial test that contains no MIT
16.       }
17.   } //finish vertical growth
18. }
19. return $S$
```

Fig. 6. Algorithm FMTG

During test generation, we first sort parameters according to how many MITs are involved. Then we build a t-way test set for the first *t* parameters, which are actually all valid combinations of these parameters (line 5). Next we extend this test set for one more parameter, and continue to do so until it builds a t-way test set for all the parameters. For each new parameter, we need to cover all the t-way combinations involving the new parameter and any group of (*t*-1) parameters among previous parameters. These combinations are covered in two steps, i.e., horizontal growth (lines 8 to 11) and vertical growth (lines 12 to 17). Horizontal growth adds a new parameter for each existing test. Each value is chosen such that it covers the most uncovered combinations and covers no MIT. In vertical growth, the remaining combinations are covered either by changing an existing test or by adding a new test (line 15).

The test generation part adopts a general IPOG-C test generation algorithm studied in our recent work [14]. One major difference is that, validity checking in [14] is handled by a constraint solver, while in algorithm FMTG, we use MITs for validity checking. Another difference is in line 4, we sort all parameters in a nonincreasing order according to the number of their appearances in the set of all MITs. This is a heuristic that can reduce the size of a generated test set. In [14] we sort parameters according to their domain sizes. However all parameters in the feature model have the same domain size, thus the original sorting approach cannot apply to feature models.

V. EXPERIMENTS

We implemented the proposed constraint-handling strategy and the test generation algorithm FMTG into a tool named LOOKUP [9]. LOOKUP takes a feature model in the Simple XML Feature Model (SXFM) format [15] as input and generates a t-way test set as output. To evaluate the proposed test generation strategy, we choose 12 largest[1] real-life feature models from the SPLOT feature models repository [11] [15]. The number of features in these models ranges from 71 to 290. These systems and the LOOKUP tool are made publicly available at our website [9].

Our experiments have two parts. In Section V.A, we evaluate the performance of MIT generation. In Section V.B, we compare our test generation algorithm to two existing test generation tools SPLCA [13] and PICT [12]. All these experiments were performed on a laptop with i5-2450M 2.5GHz CPU and 4GB memory. The generated t-way test sets are verified by an independent process.

A. Results of MIT Generation

We generated MITs for 12 subject systems and recorded the number of MITs and generation time in TABLE I. We also recorded the number of cross-tree constraints, i.e., constraints that are explicitly added to a feature model. Input invalid tuples are invalid tuples directly extracted from feature model.

TABLE I. RESULTS OF MIT GENERATION

Feature Model	# of Features	# of Cross-tree Constraints	# of given invalid tuples	# of MITs	Time (s)
Video Player	71	0	82	81	0.028
Car Selection	72	21	146	156	0.046
Eclipse1-Reuso	72	1	104	183	0.025
J2EE web arch	77	0	111	135	0.021
Transformation	88	0	140	276	0.110
Billing	88	59	153	52	0.017
Coche ecologico	94	2	155	255	0.079
UP estructural	97	2	146	314	0.090
xtext	137	1	173	453	0.075
FM_Test	168	46	294	4801	2.516
Printers	172	0	262	401	0.126
E-Shopping	290	21	399	9995	2114

[1] The SPLOT repository is continuously updated. These 12 largest features were selected in January 2013.

TABLE I shows that, for all 12 feature models except FM_test and E-Shopping, LOOKUP generated less than 1000 MITs within 1 second. The number of MITs generated by LOOKUP depends on factors such as the tree structure of a feature model and the number and type of cross-tree constraints. This supports our belief that for many practical systems, the number of MITs is small and can be generated very fast.

For FM_test, it requires more time to generate a set of 4801 MITs. For E-Shopping, it takes much long time and generates 9995 MITs. The main reason is due to the nature of cross-tree constraints in the model and the tree structure. A cross-tree constraint may connect one or more sub-trees in the feature model, leading to a large number of implied constraints between features in these sub-trees. Thus the number of MIT also becomes large.

In general, MIT generation is relatively fast in practice. For feature models with moderate size, it usually take a few seconds to generate all MITs. Furthermore, this process is independent from test generation and has nothing to do with the test strength used in test generation. The list of MITs is an alternative representation of feature model constraints, and can be used for t-way test generation with any test strength as well as other general purposes regarding the validity of test configurations.

B. Results of T-way Test Generation

We compared LOOKUP to two other test generation tools, SPLCA [16], an implementation of the ICPL algorithm [13] for t-way test generation, and PICT [12], a publicly available tool for general combinatorial testing. The SPLCA tool is by far the fastest test generation tool for feature models as shown in their evaluation. Thus we did not compare with other tools that are already compared in [8].

While LOOKUP and SPLCA can use a XML file in the SXFM format as input, PICT requires a plain-text file. We used a parser to get all the parameters and covert invalid tuples into PICT constraints.

We applied 2-way and 3-way testing in this experiment. Note that the maximal test strength supported by SPLCA is 3, while our tool supports any test strength. TABLE II shows the results of 2-way test set generation and TABLE III shows the results of 3-way test set generation. In these tables, N/A means the test set for a given feature model was not generated within one hour. Note that PICT failed to generate test sets for three feature models, i.e., Billing , Printers and E-Shopping, SPLCA failed to generate 3-way test sets for one feature model, i.e., E-shopping, and LOOKUP was able to generate test sets for all the feature models. The best sizes and times are highlighted in tables II and III.

Results of 2-way test generation show that LOOKUP generates smaller test sets for most systems. PICT generate larger test sets because it's not specially designed for feature models. Regarding the execution time, PICT and LOOKUP are faster than SPLCA but the difference is small, since 2-way test generation is relatively fast. LOOKUP is very slow on E-Shopping since most time are spent on MIT generation.

TABLE II. COMPARISON OF TEST GENERATION (2-WAY)

Feature Model	PICT 3.3		SPLCA 0.3		LOOKUP	
	size	time (s)	size	time (s)	size	time (s)
Video Player	16	13.49	18	0.62	**13**	**0.48**
Car Selection	50	**0.19**	**24**	0.74	**24**	0.61
Eclipse1-Reuso	47	**0.23**	**19**	0.75	21	0.52
J2EE web arch	36	**0.18**	18	0.71	**17**	0.52
Transformation	74	**0.3**	28	0.79	**26**	0.95
Billing	N/A	N/A	15	0.72	**13**	**0.48**
Coche ecologico	115	1.93	92	1.24	**90**	**0.87**
UP estructural	110	**0.34**	36	0.93	**34**	0.73
xtext	40	**0.33**	24	1.17	**17**	0.78
FM_Test	100	2.76	43	**1.94**	**40**	3.23
Printers	N/A	N/A	184	**2.37**	**180**	2.90
E-Shopping	N/A	N/A	26	**2.95**	**23**	2152.68

TABLE III. COMPARISON OF TEST GENERATION (3-WAY)

Feature Model Name	PICT 3.3		SPLCA 0.3		LOOKUP	
	size	time (s)	size	time (s)	size	time (s)
Video Player	47	14.03	47	3.18	**39**	**0.97**
Car Selection	243	6.14	107	4.70	**91**	**1.43**
Eclipse1-Reuso	177	5.00	96	6.68	**86**	**1.45**
J2EE web arch	132	4.43	73	4.25	**67**	**1.39**
Transformation	457	18.23	132	7.61	**131**	**3.00**
Billing	N/A	N/A	46	3.69	**39**	**1.27**
Coche ecologico	543	24.30	375	10.61	**363**	**4.72**
UP estructural	689	37.42	191	12.08	**178**	**3.44**
xtext	195	39.24	102	45.90	**80**	**6.58**
FM_Test	563	278.59	**222**	549.47	243	**29.34**
Printers	N/A	N/A	566	174.51	**547**	**58.85**
E-Shopping	N/A	N/A	N/A	N/A	**111**	**2244.16**

TABLE III shows that in terms of execution time, LOOKUP outperforms the other two tools. For some large feature models, e.g., FM_Test and Printers, our tool is faster than SPLCA by one order of magnitude. Also, LOOKUP produced the smallest test set on all the feature models except FM_Test. Comparing 2-way and 3-way results, one may find that the execution time of LOOKUP increases much slower that other tools. This is because the most time-consuming step of LOOKUP, i.e., MIT generation, is independent with test strength.

In summary, our tool is considerably better than PICT and SPLCA, in terms of test size and execution time. Note that the execution time of LOOKUP contains both MIT generation time and test generation time. The first part is independent with test strength *t*, while the second part increases with *t*. The advantage of LOOKUP can be even more significant for higher test strengths.

VI. RELATED WORK

In this section we discuss related work on modeling and testing of software product lines, combinatorial testing, and constrained combinatorial test generation.

To efficiently test software product lines, many testing techniques can be used, e.g., reusable component testing [17] and incremental testing [18]. Reusable component testing is a testing strategy where unit tests for the core assets are reused for each product. This strategy does not test for interaction faults between different components in the software product lines. Incremental testing tries to automatically adapt a test case from one version to the next version based on similarities and differences between the two versions. There is also a scenario-based method called ScenTED [19] [20]. ScenTED models extend UML activity diagrams for software product lines by introducing explicit representation of variability and then derive application test cases from the extended diagrams.

Recently several algorithms have been developed that apply combinatorial testing to software product lines. Perrouin et al. [21] [22] introduced strategies for t-wise test generation of software product lines. Machado et al [23] reviewed strategies for testing products in software product lines from 1998 to 2012. The key challenge in combinatorial testing of software product lines is how to deal with constraints. A common constraint-handling approach is using a constraint solver [24]. In this approach, validity checking is performed by constraint solvers [25]. Mendonca et al. [26] discussed the SAT-based analysis of feature models, and Johansen et al. [27] investigated covering array generation for feature models based on SAT solving.

Hadzic et al. [28] reported an approach that first constructs a Binary Decision Diagram (BDD) [29] to represent the solution space of all valid configurations, and then calculates valid domains for the remaining unassigned variables by extracting values from the BDD. Given a combination of value-assigned variables, if the BDD shows no valid domain for any remaining unassigned variable, the combination is considered invalid. The size of BDD can vary dramatically depending on the order of the assigned variables plus unassigned variables. In contrast, we derive all MITs from a feature model and use them directly for validity checking. Unlike BDD, the number of MITs is independent from the order of parameters.

Most combinatorial test generation algorithms use constraint solvers for constraint handling, such as ICPL [13] and IPOG-C [30]. The only test generation tool that systematically uses a similar constraint-handling strategy like MIT is PICT [12]. PICT uses forbidden tuples for validity checking. It first generates all necessary forbidden tuples from input constraints, and then uses them for validity checking during test generation. However, the definition of forbidden tuples and the details of how to generate them are not reported.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present an efficient combinatorial test generation algorithm for software product lines based on a novel approach of validity checking using minimum invalid tuples (MITs). Constraints in a feature model are converted into a set of MITs, and are then used for quick validity-checking during test generation. Experiments show that the performance of test generation is greatly improved while the test size is very competitive as well.

In the future, we will conduct more experiments on large feature models to evaluate our approach. We also plan to apply the constraint-handling approach proposed in this paper on general systems which may contain non-Boolean parameters and more complex constraints.

VIII. ACKNOWLEDGMENTS

This work is partly supported by three grants (70NANB9H9178, 70NANB10H168, 70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

REFERENCES

[1] K. Pohl, G. Böckle and F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University Software Engineering, 1990.

[3] T. Thüm, C. Kästner, F. Benduhn and J. Meinicke, "FeatureIDE: An extensible framework for feature-oriented software development," *Science of Computer Programming,* 2012.

[4] D. R. Kuhn, D. R. Wallace and A. J. Gallo Jr., "Software fault interactions and implications for software testing," *Software Engineering, IEEE Transactions on,* vol. 30, pp. 418-421, 2004.

[5] D. R. Wallace and D. R. Kuhn, "Failure modes in medical device software: An analysis of 15 years of recall data," *International Journal of Reliability, Quality and Safety Engineering,* vol. 8, pp. 351-371, 2001.

[6] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker and R. Kuhn, "Combinatorial Testing of ACTS: A Case Study," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 2012.

[7] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of Experiments to Software Testing," in *27th NASA/IEEE Software Engineering Workshop*, 2002.

[8] M. F. Johansen, Ø. Haugen and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *ACM*, 2012.

[9] "http://barbie.uta.edu/~lyu/lookup," [Online].

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

[11] M. Mendonca, M. Branco, Cowan and Donald, "SPLOT: software product lines online tools," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009.

[12] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, 2006.

[13] M. F. Johansen, Ø. Haugen and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*, 2012.

[14] L. Yu, Y. Lei, R. Kacker and D. R. Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013 Tools Track)*, 2013, in press.

[15] "http://gsd.uwaterloo.ca:8088/SPLOT," [Online].

[16] "http://heim.ifi.uio.no/martifag/splc2012/," [Online].

[17] M. F. Johansen, O. Haugen and F. Fleurey, "A survey of empirics of strategies for software product line testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011.

[18] E. Uzuncaova, S. Khurshid and D. Batory, "Incremental test generation for software product lines," *Software Engineering, IEEE Transactions on,* vol. 36, pp. 309-322, 2010.

[19] A. Reuys, S. Reis, E. Kamsties and K. Pohl, "The scented method for testing software product lines," in *Software Product Lines*, Springer, 2006, pp. 479-520.

[20] K. Pohl and A. Metzger, "Software product line testing," *Communications of the ACM,* vol. 49, no. 12, pp. 78-81, 2006.

[21] G. Perrouin, S. Sen, J. Klein, B. Baudry and Y. Le Traon, "Automated and scalable t-wise test case generation strategies for software product lines," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010.

[22] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry and Y. Le Traon, "Pairwise testing for software product lines: Comparison of two approaches," *Software Quality Journal,* vol. 20, pp. 605-643, 2012.

[23] I. do Carmo Machado, J. D. McGregor and E. Santana de Almeida, "Strategies for testing products in software product lines," *ACM SIGSOFT Software Engineering Notes,* vol. 37, pp. 1-8, 2012.

[24] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *The journal of logic programming,* vol. 19, pp. 503-581, 1994.

[25] D. Batory, "Feature models, grammars, and propositional formulas," *Software Product Lines,* pp. 7-20, 2005.

[26] M. Mendonca, A. Wąsowski and K. Czarnecki, "SAT-based analysis of feature models is easy," in *Proceedings of the 13th International Software Product Line Conference*, 2009.

[27] M. F. Johansen, Ø. Haugen and F. Fleurey, "Properties of realistic feature models make combinatorial testing of product lines feasible," in *Model Driven Engineering Languages and Systems*, Springer, 2011, pp. 638-652.

[28] T. Hadzic, R. M. Jensen and H. R. Andersen, "Calculating valid domains for BDD-based interactive configuration," *arXiv preprint arXiv:0704.1394,* 2007.

[29] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers,* vol. 100, pp. 677-691, 1986.

[30] L. Yu, M. Nouroz Borazjany, Y. Lei, R. Kacker and D. R. Kuhn, "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, 2013, in press.

[31] N. Andersen, K. Czarnecki, S. She and A. Wąsowski, "Efficient synthesis of feature models," in *Proceedings of the 16th International Software Product Line Conference*, 2012.