Constraint Handling In Combinatorial Test Generation Using Forbidden Tuples

Linbin Yu
Facebook Inc.
Menlo Park, CA 94025, USA
linbin@fb.com

Feng Duan, Yu Lei
University of Texas at Arlington
Arlington, TX 76019, USA
feng.duan@mavs.uta.edu,
ylei@cse.uta.edu

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

***Abstract*—Constraint handling is a challenging problem in combinatorial test generation. In general, there are two ways to handle constraints, i.e., constraint solving and forbidden tuples. In our earlier work, we proposed a constraint handling approach based on forbidden tuples for software product line systems consisting of only Boolean parameters. In this paper, we generalize this approach for general software systems that may consist of other types of parameter. The key idea of our approach is using the notion of minimum forbidden tuples to perform validity checks on both complete and partial tests. Furthermore, we propose an on-demand strategy that only generates minimum forbidden tuples for validity checks as they are encountered, instead of generating all of them up front. We implemented our generalized approach with and without the on-demand strategy in our combinatorial testing tool called ACTS. We performed experiments on 35 systems using ACTS and PICT. The results show that for these 35 systems, our generalized approach performed faster than PICT and the constraint solving-based approach in ACTS. For some large systems, the improvement on test generation time is up to two orders of magnitude.**

Keywords*—Combinatorial Testing; Constraints; Forbidden Tuples;

I. Introduction

Real-life applications often have constraints on how parameter values can be combined in a test [1]. Constraints can be specified in two different forms, including logical expressions and forbidden tuples. These two forms of specification can be converted from one to the other. In general, there are two types of approach to handle constraints, i.e., constraint solving-based and forbidden tuple-based approaches. A constraint solving-based approach usually employs a constraint solver to ensure test validity. A test is valid if and only if it satisfies all the constraints. A forbidden tuple-based approach derives a set of forbidden tuples and uses them to ensure test validity. A test is valid if and only if it does not contain any forbidden tuple.

Both constraint solving-based and forbidden tuple-based approaches have their advantages and disadvantages. The former is preferred when constraints are complex and there exist a large number of forbidden tuples. The latter is more efficient when the number of forbidden tuples is small, which seems to be the case for many real-life systems based on our experience and as shown in our experiments. There are, however, few forbidden tuple-based approaches reported in the literature. In [2], we reported a forbidden tuple-based approach to handle constraints for software product line systems consisting of Boolean parameters. To our best knowledge, PICT [3] is the only other work that uses a forbidden tuple-based approach to handle constraints. However, the details of how to derive forbidden tuples are not discussed in [3].

In this paper, we generalize our earlier work in [2] to systems that may consist of parameters of an arbitrary domain size. We focus on constraints specified in forbidden tuples. The main challenge in a forbidden tuple-based approach is how to determine the validity of a partial test in which the value of one or more parameters has not been assigned. Greedy algorithms such as IPOG [4] and AETG [5] construct a test in an incremental manner in terms that the parameter values in a test are assigned one at a time. Each time a parameter value is assigned, the validity of a (partial) test is checked before the next parameter value is assigned. Given a complete test in which every parameter has a value, validity of this test can be determined by checking if it contains any user-specified forbidden tuple. However, this approach does not work for a partial test. This is because there may exist some implicit forbidden tuple that are not explicitly specified by the user [4]. Thus, a partial test containing no user-specified forbidden tuple may still be invalid.

Fig. 1 shows an example system consisting of three Boolean parameters A, B, C and two user-specified forbidden tuples {A=0, C=0} and {B=0, C=1}. A partial test {A=0, B=0} is invalid even though it does not contain any user-specified forbidden tuples. This is because no value can be set for parameter C to make the test complete. If C is set to 0, we obtain test {A=0, B=0, C=0} which is invalid as it contains the first user-specified forbidden tuple {A=0, C=0}. Similarly if C is set to 1, we obtain test {A=0, B=0, C=1}, which is invalid as it contains the second user-specified forbidden tuple {B=0, C=1}. Therefore tuple {A=0, B=0} is invalid. Later we will show that {A=0, B=0} is actually an implicit tuple that can be derived from {A=0, C=0} and {B=0, C=1}.

2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). 4th International Workshop on Combinatorial Testing (IWCT 2015).
978-1-4799-1885-0/15/$31.00 ©2015 IEEE.

Parameters: A = {0,1}, B={0,1}, C={0,1} Forbidden Tuples: {A=0, C=0}, {B=0, C=1}	Tests: {A=0, B=0} (Invalid)

Fig.1 Example of Invalid Partial Test

To address this challenge, we introduce a notion called minimum forbidden tuple (or MFT). This notion was originally defined as minimum invalid tuple in [2]. A MFT is a forbidden tuple of minimum size. We develop an approach to generate all the MFTs from a set of user-specified forbidden tuples. We show that a partial test is valid if and only if it contains no MFTs. Once all MFTs are generated, validity check of a partial test can be performed in a way that is similar to validity check of a complete test. Note that our earlier approach for software product line systems only deal with Boolean parameters. As a result, it cannot be directly applied to general systems that may consist of other types of parameters. This approach is generalized in this paper in order to handle general systems with parameters of an arbitrary domain size.

When the number of forbidden tuples is small, the MFT generation process is very fast. However, when the number of forbidden tuples is large, it can be very expensive to generate all the MFTs. A key observation is that not all the MFTs are needed in every validity check. We propose an on-demand MFTs generation strategy that only generates MFTs that are needed to perform validity checks as they are encountered. Experiment results show that this on-demand approach can reduce the time spent on MFTs generation when the number of forbidden tuples is large.

The main contributions of this paper are:

- We generalize our earlier work on constraint handling based on forbidden tuples for software product line systems consisting of only Boolean parameter to systems that may consist of other types of parameters.
- We propose an on-demand MFTs generation approach that only generates necessary MFTs for individual validity checks.
- We implement our generalized forbidden tuple-based approach with and without the on-demand strategy. The implementation is publicly available at [5].
- We conduct an experimental evaluation of the proposed approach. The results show that the proposed approach performs better, in several cases significantly better, than the constraint solving-based approach in ACTS [6] and PICT [3].

The rest of this paper is organized as follows. Section II gives several formal definitions used in this paper. Section III introduces the generalized MFTs generation approach. Section IV introduces our approach for on-demand forbidden tuple generation. Section V reports experimental results. Section VI briefly discusses the related works. Section VII concludes this paper and discusses our future work.

II. Preliminaries

In this section, we present some formal definitions that are used in this paper. Definitions 1 to 3 were first presented in our early work [7].

Definition 1 (Parameter) A parameter P is a set of values, i.e., $P = \{v_1, v_2, \ldots, v_p\}$

The value set of a parameter is also denoted as the domain of the parameter. According to Definition 1, a parameter value belongs to only one parameter. Thus, the association between a value and the parameter to which this value belongs to is implicit. This allows us to refer to a parameter value by itself, i.e., without mentioning which parameter it belongs to. For example, we use value v instead of value $P.v$ if there is no ambiguity.

Definition 2 (Tuple) Let $G = \{P_1, P_2, \ldots, P_m\}$ be a set of parameters. A tuple $U = \{v_1, v_2, \ldots, v_m\}$ of G is a set of values where $v_i \in P_i$. That is, $U \in P_1 \times P_2 \ldots \times P_m$.

A tuple of size m represents the value assignment of m parameters. Thus, a tuple can only have one value from the same parameter.

Definition 3 (Covering) A tuple U is said to be covered or contained by another tuple U' if U is a subset of U', i.e., $U \subseteq U'$.

Let Σ be a system consisting of a set Δ of parameters. A tuple of Σ is said to be complete if it contains a value for every parameter in Δ.

Definition 4. (Tuple Validity) Let Σ be a system. Let F be a set of forbidden tuples $\{U_1, U_2, \ldots, U_n\}$ of Σ. The validity of a tuple U of Σ can be determined as follows:

- U is complete: U is valid if U covers no forbidden tuple in F, i.e. $\forall T \in F, T \not\subseteq U$.
- U is not complete: U is valid if U can be covered by a valid, complete tuple.

In the rest of the paper, we will refer to a tuple whose validity needs to be checked as a test. We will refer to a complete or partial test simply as a test when there is no ambiguity. We will also refer to forbidden tuples in F as explicit forbidden tuples. Later we will show that more forbidden tuples could be derived from explicit forbidden tuples in F; we will refer to these derived forbidden tuples as implicit forbidden tuples.

We will use the following naming conventions in the rest of the paper.

System	Σ
Parameter	Pi
Parameter value	v_i
Parameter Set	Δ
Tuple	U_i
Test	T_i
Set of tuples (tests)	Ω

III. Minimum Forbidden Tuples Generation

In this section, we first introduce the concept of minimum forbidden tuples (MFTs). Then we show how to check validity using MFTs, and how to generate all MFTs from explicit forbidden tuples for general systems.

A. *Minimum Forbidden Tuples*

As we discussed in Introduction, forbidden tuples can be used to verify if a complete test is valid or not. However, a partial test covering no forbidden tuples may still be invalid. In the example shown in Fig. 1, a partial test {A=0, B=0} is invalid even it covers no forbidden tuples.

Usually we cannot directly use forbidden tuples to check the validity of a partial test. This is because user-specified forbidden tuples may imply more forbidden tuples that are not explicitly specified [4]. A partial test that covers no explicit forbidden tuple may cover some implicit forbidden tuples. Take the system in Fig. 1 as an example, {A=0, B=0} is actually an implicit forbidden tuple.

It is not practical for the user to specify all forbidden tuples in a system. If we can generate all implicit forbidden tuples, the validity of a partial test can be easily determined using the same way as validity check for a complete test. However, as pointed out in [4], the number of implicit forbidden tuples could be very large. To address this challenge, we propose the concept of minimum forbidden tuple. We first present the formal definition of minimum forbidden tuple [2].

Definition 5 (Minimum Forbidden Tuple) An invalid tuple U is a minimum forbidden tuple (MFT) if $\forall U' \subseteq U$ is valid.

Intuitively, a MFT is a forbidden tuple of minimum size and covers no other forbidden tuples. All proper subsets of a MFT are valid tuples. As a special case, a forbidden tuple of size one must be a MFT.

Theorem If a tuple T is invalid, then it covers at least one MFT.

Proof. If an invalid tuple T contains only one value, then it must be a MFT. Assume T contains two or more values. If all its proper subsets are valid, then according to the definition, T itself is a MFT. Otherwise, we can find a tuple of minimal size in those invalid proper subsets, and that tuple must be a MFT by definition.

Note that an invalid tuple may contain more than one MFTs. Based on this theorem we know that a partial test is valid if and only if it covers no MFT. Therefore, the validity of a partial can be determined by checking if it covers any MFT.

B. *The Derive and Simplify Process*

An MFT finding algorithm has been proposed in our early work [2] for product line systems that have only Boolean parameters. The algorithm iteratively applies two processes, i.e., *derive* and *simply*, on the set of forbidden tuples until it converges.

We first introduce a rule that can be used to derive implicit forbidden tuples from existing ones. This rule is generalized from the original rule for Boolean parameters [2].

(Derive) Assume a parameter P has n values, and there are n forbidden tuples in which each forbidden tuple contains a different value of parameter P. A new forbidden tuple can be constructed by combining all values in these n forbidden tuples other than values from parameter P.

This process is the key step of MFT generation. First, given a parameter P of domain size n, we have to find n forbidden tuples such that each forbidden tuple contains a different value of parameter P. If a value in parameter P is not contained by any forbidden tuple, then no new forbidden tuples can be derived using this parameter. Second, the new tuple is constructed by combining all values in these forbidden tuples except values in P. It's possible that this value set contains more than one value for the same parameter. In this case, this value set is not a tuple, and we cannot derive a new forbidden tuples from them.

There are some examples shown as follows.

Parameters: A = {0,1,2}, B = {0,1,2}, C = {0,1,2}, D = {0,1,2}			

(1)	{ A=0, C=0 }
	{ A=1, B=1, C=0 }
	{ A=2, D=2 }
		(derive using A)	{B=1, C=0, D=2}

(2)	{ A=0, B=0 }
	{ A=1, B=1, C=0 }
	{ B=2, D=0 }
		(derive using B)	~~{A=0, A=1, C=0, D=0}~~
			(not a tuple as it contains 2 values for A)

(3)	{ C=0 }
	{ C=1 }
	{ C=2 }
		(derive using C)	{ } (an empty tuple)

Fig.2 Examples of Deriving New Invalid Tuples

In Fig. 2(1), a new forbidden tuple is derived from three forbidden tuples using parameter A. In Fig. 2(2), there are two conflict values A=0 and A=1 in the combined values set, thus it's not even a tuple. In Fig. 2(3) we show a special case that an *empty* tuple is generated from 3 forbidden tuples of size 1. It actually means we cannot assign a valid value for parameter C. We will discuss more about empty tuples in the section of on-demand MFT generation.

We now explain why the tuple derived as above is invalid. Given n forbidden tuples $\{U_i\}$ and a parameter $P = \{v_1, v_2, \ldots, v_n\}$, where $v_i \in U_i$, $i = 1, 2, \ldots n$. The new derived forbidden tuple U consists of parameter values in these forbidden tuples excluding values from parameter P, i.e., $U = U_1\backslash\{v_1\} \cup U_2\backslash\{v_2\} \cup \ldots \cup U_n\backslash\{v_n\}$. Notice that U contains no value from P. If U is a tuple (it contains only one value from the same parameter), it is easy to see that we cannot assign a valid value for parameter. In particular, assigning value v_i to U will cover forbidden tuple U_i, i.e., $U_i \subseteq U \cup \{v_i\}$. Therefore, U must be an invalid tuple.

Another process called simplify is used to remove forbidden tuples that cannot be MFTs. This process is the same as the original one for Boolean parameters [2].

(Simplify) Given a set Ω of forbidden tuples, tuple U can be removed if $\exists\ U' \in \Omega$ such that $U' \subseteq U$.

By definition, a forbidden tuple that covers another forbidden tuple cannot be a MFT. For example, assuming we have two forbidden tuples {B=2, D=0} and {B=2, C=1, D=0}. The latter cannot be a MFT as it covers the former. The result of validity check will not be affected if we remove the latter, because any partial test covering that forbidden tuple must also cover the former forbidden tuple.

C. *The MFT Generation Process*

The MFT generation process is similar to the one proposed in [2]. The main difference is in the derive process described in the last section.

If the same parameter value appears in multiple forbidden tuples, we may derive multiple forbidden tuples using this parameter. Given a parameter $P = \{v_1, v_2, \ldots, v_m\}$, we group forbidden tuples according to the value of P. Let tuple set Ω_i be $\{U \backslash \{v_i\} \mid v_i \in U\}$, multiple forbidden tuples could be derived using Cartesian product $\Omega_1 \times \Omega_2 \ldots \times \Omega_m$. An example is shown in the following figure where there are two forbidden tuples that have A=0, and 2 other forbidden tuples that have A=1. There are four value sets can be deriving using parameter A. The first one is not a tuple because it contains two different values for parameter B. The other three are newly derived forbidden tuples. Note that {B=2, C=0, D=0} will be removed in the simplify process because it covers another forbidden tuple {B=2, D=0}.

{ A=0, B=0 }		{{B=0}, {B=2}}		~~{ B=0, B=2, D=0 }~~
{ A=0, B=2 }		×		{ B=0, C=0, D=0 }
{ A=1, B=2 }	→	{{B=2}, {C=0}}	→	{ B=2, D=0 }
{ A=1, C=0 }		×		{ B=2, C=0, D=0 }
{ A=2, D=0}		{{D=0}}		

Fig.3 Examples of Deriving Multiple New Invalid Tuples

The MFTs generation algorithm starts from the set of explicit forbidden tuples. It iteratively derives new forbidden tuples and simplifies the set of forbidden tuples, until no new forbidden tuples could be derived. The final set of forbidden tuples consists of all MFTs. More details about this approach can be found in [2].

We use an example to show it works. In the first two steps, three new forbidden tuples are derived using parameter A and B. There are no new forbidden tuples that can be derived using parameter C and D, so we move forward to the simplify process. In step 3, no tuples can be removed at this time. The next iteration starts with three new forbidden tuples that are marked with *. In step 4 we derive a new tuple {D=0} using parameter A. In step 5, six forbidden tuples containing {D=0} are removed. Finally, there are only three forbidden tuples in the sets and no new tuples could be derived. These forbidden tuples are MFTs and can be used for the validity check.

Parameters: A={0,1,2}, B={0,1,2}, C={0,1,2}, D={0,1,2}			

Step 1	{ A=0, B=0 }
	{ A=0, B=2 }
	{ A=1, D=0 }
	{ A=2, D=0 }
	{ B=1, D=0 }
		(derive using A)	{ B=0, D=0 }
			{ B=2, D=0 }

Step 2	{ A=0, B=0 }
	{ A=0, B=2 }
	{ A=1, D=0 }
	{ A=2, D=0 }
	{ B=1, D=0 }
		(derive using B)	{ A=0, D=0 }

Step 3	(simplify skipped)		

Step 4	{ A=0, B=0 }
	{ A=0, B=2 }
	{ A=1, D=0 }
	{ A=2, D=0 }
	{ B=1, D=0 }
	{ B=0, D=0 }*
	{ B=2, D=0 }*
	{ A=0, D=0 }*
		(derive using A)	{ D=0 }

Step 5	~~{ A=1, D=0 }~~ ~~{ A=2, D=0 }~~ ~~{ B=1, D=0 }~~ ~~{ B=0, D=0 }~~ ~~{ B=2, D=0 }~~ ~~{ A=0, D=0 }~~	(simplify due to D=0)	

Step 6	{ A=0, B=0 }
	{ A=0, B=2 }
	{ D=0 }
		Terminated with 3 MFTs	

Fig.4 Example of MFT Generation Process

IV. On-demand Forbidden Tuples Generation

The MFT generation process is fast when the number of forbidden tuples is small. However, if there are too many forbidden tuples, it could be very expensive to generate all MFTs from them. Given a partial test, not all MFTs are actually needed in validity check. Based on this observation we propose an on-demand MFTs generation strategy. Instead of generating all MFTs up front, this approach generates necessary forbidden tuples (NFTs) for a particular validity check.

A. *Necessary Forbidden Tuples*

The process of validity check for a test T is essentially to find MFTs that can be covered by T. However, a MFT that meets condition (1) or (2) cannot be covered by T:

(1) It does not contain any value from T.

(2) It contains a different value with what T has for the same parameter.

Those MFTs are not necessary in the validity check of T. A forbidden tuple meets either of these two conditions is denoted as a necessary forbidden tuple (NFT) *w.r.t* test T. Using these two conditions, we can identify the initial set of NFTs from a set of forbidden tuples.

When derive new forbidden tuples, we combin all values from a set of existing forbidden tuples (except values of one parameter). Therefore, a tuple derived from non-NFTs cannot be a NFT. In other words, a NFT can only be derived from a set of forbidden tuples containing at least one NFT. The derive process can be slightly modified such that only NFTs are derived.

Given a test *T*, we want to derive NFTs for validity check. For this particular problem, all parameters appearing in *T* are fixed. We show that if a forbidden tuple *U* contains the same parameter value with *T*, we can remove that value from *U.* Assume we removed some values in this way from *U* and denote the new tuple as *U'*. We can see that either *T* covers both *U* and *U'*, or *T* covers either of them. This means removing values from *T* does not affect the result of validity check. It also implies that an initial NFT contains no parameters in test *T*. For example, given a test {A=0, B=1}, the forbidden tuple {A=0, C=1} can be simplified to {C=1}, since {A=0} is contained in the test.

B. The NFT Generation Process

The NFT generation process is similar to MFT generation. The main differences are (1) the simplifying process used to obtain the initial NFT set; (2) the modified derive process that only generates NFTs. Note that input forbidden tuples that are not NFTs are also kept for deriving new NFTs during NFT generation process. Similarly, we iteratively apply these two processes until convergence.

An example is shown in the following Fig 5. The test is {A=0, B=1} and there are 7 forbidden tuples. We first obtain the initial NFTs (marked with star) by removing {A=0} and {B=1} from input forbidden tuples. Forbidden tuple {A=1, D=0} is remove as it contains A=1. Then we start deriving new NFTs. Note that in step 2, we may derive a new tuple {E=0, H=1} from 3 forbidden tuples {F=2, H=1}, {F=1, E=0} and {F=0, E=0} using parameter F. However, {E=0, H=1} is not a NFT since none of these three forbidden tuples is a NFT. In step 3 we remove two forbidden tuples which covers the new derived forbidden tuple {D=0}. Finally we obtained 5 forbidden tuples that can be used in validity check.

Parameters: A/B/C/D/E/F/H = {0,1,2}			
Test {A=0, B=1}, NFTs are marked with *			

Step 1	{ A=0, C=0, D=0 }
	{ B=1, C=2 }
	{ A=1, D=0 }
	{ F=2, H=1}
	{ F=1, E=0}
	{ F=0, E=0}
	{ C=1, D=0}
		(initial set)	{ ~~A=0,~~ C=0, D=0 }*
			{~~B=1,~~ C=2 }*
			~~{ A=1, D=0 }~~
			{ F=2, H=1}
			{ F=1, E=0}
			{ F=0, E=0}
			{ C=1, D=0}

Step 2	{ C=0, D=0 }*
	{ C=2 }*
	{ F=2, H=1}
	{ F=1, E=0}
	{ F=0, E=0}
	{ C=1, D=0 }
		(derive using C)	{ D=0 }*

Step 3	~~{ C=0, D=0 }*~~ ~~{ C=1, D=0 }~~	(simplify due to D=0)	

Step 4	{ C=2 }
	{ D=0 }
	{ F=2, H=1}
	{ F=1, E=0}
	{ F=0, E=0}
		(finished)	

Fig.5 Example of NFTs Generation

C. Validity Check Using NFTs

Similar to the MFT approach, we want to find if any NFT that can be covered by *T*. However, in section IV.A we know that the initial set of NFTs does not contain any parameters in *T*, nor the final set of NFTs do. Actually, in the NFT-based approach, the validity check is implicitly performed during the NFT generation process. A test is invalid only if an empty tuple is generated during NFT generation, otherwise it must be valid.

For example, the final set of NFTs in Fig.5 contains no parameter from test {A=0, B=1}, and there is no empty tuple. Therefore test {A=0, B=1} is valid.

D. Discussion

We summarize differences between the original MFT-based constraint handling approach and the on-demand NFT-based constraint handling approach. The MFT-based approach requires all MFTs to be generated up front while the NFT-based approach does not. MFTs generation is a one-time process and has nothing to do with test strength. For validity check, the MFT-based approach searches all MFTs to find if any MFT can be covered by the test. In the NFT-based approach, we only check if an empty tuple is derived during NFTs generation.

The NFT-based approach avoids the one-time cost of MFT generation, but the NFT generation is required for every validity check. The NFT-based approach may be faster than the MFT-based approach if (1) it's very expensive to generate all MFTs; (2) there are only a few validity checks to perform during test generation.

V. EXPERIMENTS

In order to evaluate the performance of the proposed constrain handling approaches based on MFT and NFT, we integrated them into an existing test generation algorithm IPOG [8], and denoted them as IPOG-MFT and IPOG-NFT. The implementations are available online [5].

We first evaluate the performance of MFT generation in section A. Then we compare IPOG-MFT and IPOG-NFT with algorithm IPOG-C [7] (implemented in ACTS [6] version 2.82) and another widely used combinatorial test generation tool, PICT [3] (version 3.3). IPOG-C uses a CSP solver to handle constraint while PICT uses a forbidden tuple based approach to handle constraints. Both of them are greedy constraint generation algorithms. Subject systems include Apache, Bugzilla, Gcc, Spin-S, Spin-V and other 30 systems studied in [9].

We note that the number of forbidden tuples in these systems is very small. For better comparison between IPOG-MFT and IPOG-NFT, we use two real-life systems that are reported by ACTS users. For proprietary reasons, we will refer to these two systems as RL-A and RL-B. Both of them have very complex constraints, in the form of logical expressions. We convert these expressions to equivalent forbidden tuples in order to use the proposed constraint handling approaches..

All the experiments were performed on a laptop with Intel i5-2450M 2.5GHz CPU and 4GB memory, running 64-bit Windows 7 and 64-bit Java 7. To ensure correctness of the proposed constraint handling approaches, we verified the validity of the complete tests after test generation.

A. *Results of MFT Generation*

We generated all MFTs for 37 subject systems and recorded the number of MFTs and the generation time in TABLE I.

MFT generation is very fast for the first 35 subject systems. It takes only a few milliseconds to find all MFTs. This is because the number of forbidden tuples for these systems is very small (less than 50) and most of them are of size 2 or 3. The number of generated MFTs is also very small, ranging from 5 to 219. This makes the validity check process very fast comparing to constraint solving approaches.

Systems RL-A and RL-B have a large number of forbidden tuples, and the MFT generation time is relatively longer. It takes about 10 seconds to generated all 824 MFTs for system RL-A, and takes about 21 minutes to generate 2085 MFTs for system RL-B.

B. *Results of T-way Test Generation*

In Table II we compares PICT, IPOG-C, IPOG-MFT, and IPOG-NFT in terms of test generation time (including MFT/NFT generation time) and the number of generated tests. Note that methods IPOG-C, IPOG-MFT, and IPOG-NFT share the same test generation engine, and use different constraint handling approaches. Constraint handling does not affect the test generation result. Therefore, these methods generate exactly the same test set. We set the test strength to 3, because 2-way test generation for most systems is very fast and the difference is not significant.

The results show that IPOG-MFT and IPOG-NFT are faster than IPOG-C and PICT. For many systems, the improvement is one to two orders of magnitude. Take system 5 as an example, IPOG-C and PICT take more than 300 seconds, while IPOG-MFT and IPOG-NFT only take about 20 seconds.

The difference between IPOG-MFT and IPOG-NFT is small. This is because the number of forbidden tuples is very small and the MFTs generation process is fast for these systems. In the next section, we will compare IPOG-MFT and IPOG-NFT using systems RL-A and RL-B that have a large number of forbidden tuples.

TABLE I. RESULTS OF MFTS GENERATION

System	**Parameter**	**Constraint**	**Number of forbidden tuples**	**Number of generated MFTs**	**MFT generation time (ms)**
Apache	$2^{158}\,3^{8}\,4^{4}\,5^{1}\,6^{1}$	$2^{3}3^{1}4^{2}5^{1}$	7	7	2
Bugzilla	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	5	5	1
GCC	$2^{189}3^{10}$	$2^{37}3^{7}$	40	39	3
Spin-S	$2^{13}4^{5}$	2^{13}	13	13	1
Spin-V	$2^{42}3^{2}4^{11}$	$2^{47}\,3^{2}$	49	66	2
1	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	24	67	4
2	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	22	46	3
3	$2^{27}4^{2}$	$2^{9}3^{1}$	10	19	1
4	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	17	21	2
5	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	39	112	6
6	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	30	12	5
7	$2^{29}3^{1}$	$2^{13}3^{2}$	15	5	1
8	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	37	57	4
9	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	37	15	3
10	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	47	117	8
11	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	32	69	4
12	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	27	34	3
13	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	26	24	1
14	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	15	11	2
15	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	22	91	6
16	$2^{81}3^{3}4^{2}6^{1}$	$2^{30}3^{4}$	34	15	5
17	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	29	99	1
18	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	28	35	4
19	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	43	219	3
20	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	48	76	8
21	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	46	17	4
22	$2^{72}3^{4}4^{1}6^{2}$	$2^{31}3^{4}$	22	19	3
23	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	15	15	1
24	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	29	47	2
25	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{3}4^{1}$	27	74	6
26	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	32	79	5
27	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	20	54	1
28	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	37	157	4
29	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	22	52	3
30	$2^{73}3^{3}4^{3}$	$2^{20}3^{2}$	35	24	8
RL-A	$1^{2}2^{5}3^{4}4^{7}5^{4}$ $6^{5}7^{4}8^{1}12^{3}$	-	848	824	9726
RL-B	$1^{47}2^{8}3^{2}4^{3}5^{3}6^{1}9^{1}$ $10^{1}12^{2}14^{3}20^{1}$ $24^{1}37^{1}$	-	6279	2085	1276306

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

System	PICT [3]		IPOG-C [7]		IPOG-MFT		IPOG-NFT	
	# of Tests	Time (s)	# of Tests	Time (s)	# of Tests	Time (s)	# of Tests	Time (s)
Apache	*202*	135.373	*173*	25.732	*173*	**15.312**	*173*	15.38
Bugzilla	*70*	0.649	*68*	0.197	*68*	**0.095**	*68*	0.111
GCC	*134*	136.174	*108*	27.107	*108*	**16.655**	*108*	17.851
Spin-S	*113*	0.063	*98*	0.894	*98*	**0.016**	*98*	**0.016**
Spin-V	*345*	4.553	*286*	3.252	*286*	**0.531**	*286*	0.75
1	*332*	26.59	*293*	30.343	*293*	**1.825**	*293*	1.856
2	*200*	13.046	*174*	15.83	*174*	**1.126**	*174*	1.171
3	*71*	0.115	*71*	0.365	*71*	**0.016**	*71*	**0.016**
4	*116*	1.587	*102*	2.77	*102*	**0.2**	*102*	0.219
5	*454*	319.629	*386*	359.642	*386*	**20.602**	*386*	20.969
6	*133*	4.156	*119*	9.019	*119*	**0.562**	*119*	0.672
7	*36*	0.065	*35*	0.219	*35*	**0.016**	*35*	0.031
8	*361*	56.263	*326*	122.496	*326*	**4.462**	*326*	4.22
9	*80*	1.169	*84*	7.253	*84*	**0.297**	*84*	0.328
10	*389*	139.347	*329*	193.421	*329*	12.607	*329*	**11.171**
11	*403*	31.053	*318*	46.024	*318*	2.372	*318*	**1.983**
12	*315*	107.192	*263*	81.109	*263*	11.177	*263*	**8.567**
13	*234*	54.209	*200*	24.18	*200*	5.767	*200*	**5.136**
14	*269*	17.341	*244*	5.898	*244*	1.454	*244*	**1.407**
15	*206*	3.004	*173*	3.538	*173*	**0.267**	*173*	0.281
16	*131*	6.039	*117*	13.88	*117*	**0.905**	*117*	0.958
17	*307*	79.883	*265*	67.139	*265*	6.601	*265*	**6.253**
18	*380*	122.715	*344*	50.049	*344*	9.055	*344*	**8.418**
19	*440*	472.994	*373*	281.289	*373*	38.74	*373*	**36.38**
20	*523*	249.383	*463*	897.651	*463*	**17.568**	*463*	18.044
21	*247*	11.124	*235*	18.846	*235*	**1.235**	*235*	1.358
22	*178*	6.023	*164*	6.257	*164*	**0.735**	*164*	0.75
23	*48*	0.079	*48*	0.329	*48*	0.016	*48*	**0.015**
24	*393*	58.671	*341*	62.462	*341*	4.185	*341*	**4.181**
25	*478*	120.012	*404*	75.911	*404*	**7.358**	*404*	7.629
26	*226*	14.915	*207*	31.139	*207*	**1.407**	*207*	1.421
27	*242*	4.135	*204*	2.793	*204*	0.351	*204*	**0.346**
28	*494*	497.876	*420*	157.953	*420*	**33.953**	*420*	36.81
29	*165*	48.501	*154*	10.609	*154*	6.132	*154*	**6.014**
30	*104*	3.543	*100*	7.43	*100*	**0.579**	*100*	0.61

C. *Comparison between IPOG-MFT and IPOG-NFT*

We compare IPOG-MFT and IPOG-NFT using systems RL-A and RL-B. The test strength is set 3 for RL-A. For RL-B, we use the mixed strength provided by the user. As mentioned earlier, the constraints in both systems are represented using logical expressions. In order to use the forbidden tuples based constraint handling approach proposed in this paper, we first convert these constraints to equivalent forbidden tuples.

TABLE III. COMPARISON OF TEST GENERATION (RL-A AND RL-B)

System	IPOG-MFT			IPOG-NFT	
	# of Tests	MFT generation time (s)	Time (s)	# of Tests	Time (s)
RL-A	1132	9.726	16.785	1132	10.16
RL-B	98	1276.306	1277.248	98	206.802

Table III shows the results of test generation for RL-A and RL-B using IPOG-MFT and IPOG-NFT. For the system RL-A, more than 57% of the total test generation time is spent on MFT generation. For the same system, the on-demand approach IPOG-NFT takes 10.16 seconds, which is 60% faster. For the system RL-B with 6279 forbidden tuples, MFT generation takes 1276 seconds to finish. Note that the total test generation time is 1277 seconds. That means most of execution time is spend on MFT generation. For this system, IPOG-NFT is 6 times faster (207 seconds). This demonstrates the advantage of the on-demand strategy.

E. *Discussion*

In summary, the proposed forbidden tuples based constraint handling approaches outperform the traditional constraint solving based approach on 35 subject systems. When the number of forbidden tuples is small, constraint handling based on MFT and NFT are both very efficient. When the number of forbidden tuples is large, MFT generation could be expensive and in this case, the NFT-based approach may be faster.

VI. Related Works

We first review existing works on constraint solving-based approaches. SAT solvers are used in many approaches for constraint handling. Garvin et al. integrated a SAT solver into a meta-heuristic search algorithm for constrained combinatorial test generation [10]. Cohen et al. integrated a SAT solver into an AETG-like test generation algorithm [4] [9]. Software product line systems usually have a large number of constraints. Perrouin et al. [11] [12] addressed the scalability of a SAT solver in combinatorial testing for software product lines. Specifically, they address the problem that a limited number of clauses can be solved at once. In our early work [7], we integrated a CSP solver into the IPOG algorithm and proposed several optimizations to improve the performance of constraint handling.

For forbidden tuple-based strategies, some approaches such as [13] require that all forbidden tuples are explicitly listed. This requirement simplifies constraint handling but is not practical when the number of forbidden tuples is large. PICT [3] translates constraints to a set of exclusions (i.e. forbidden tuples) and uses them to ensure that generated tests are valid. For each constrained set of parameters, it explicitly marks invalid t-way target tuples and excludes value combinations of higher strength as necessary. However, the technical details of how to generate higher strength exclusions are not discussed in the paper. In comparison, we derive forbidden tuples of minimum size and use them for validity check. Our method does not rely on invalid t-way tuples.

We note that some approaches, e.g., [14] [15], model the whole test generation problem as a constraint satisfaction problem. In this way, constraints are naturally handled during test generation. These approaches typically take longer than greedy test generation methods such as IPOG [8] and AETG [16].

VII. Conclusion

In this paper, we first presented a generalized forbidden tuple-based constraint handling approach for combinatorial test generation. Our approach is based on the notion of minimum forbidden tuple (MFT). A complete or partial test is valid if and only if it does not contain any MFT. We also presented an on-demand strategy that only generates necessary forbidden tuple (NFT) for a particular validity checks. The experiment results show that the proposed constraint handling approaches performed better than the constraint solving-based approach IPOG-C and another forbidden tuple-based approach PICT. Furthermore, when the number of forbidden tuples is large, the MFT-based approach with on-demand generation performed better than the original MFT-based approach.

We plan to continue our work in the following two directions. First, we plan to optimize the performance of our approach further. In particular, we want to explore the use of advanced data structures to manage intermediate states such that they can be efficiently stored and searched. Second, we plan to conduct more experiments to evaluate the performance of our approach, especially using systems with constraints that are at different complexity levels. Two interesting problems to investigate include are 1) how to measure the complexity of a constraint and 2) how to generate constraints of different complexity levels.

Acknowledgments

This work is partly supported by a research grant (70NANB12H175) from Information Technology Laboratory of National Institute of Standards and Technology (NIST).

References

[1] Mats Grindal, Jeff Offutt, and Jonas Mellin, "Managing conficts when using combination strategies to test software," in *the 2007 Australian Software Engineering Conference (ASWEC 2007)*, 2007.

[2] Linbin Yu, Feng Duan, Yu Lei, Raghu Kacker, and D. Richard Kuhn, "Combinatorial Test Generation for Software Product Lines Using Minimum Invalid Tuples," in *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE)*, 2014, pp. 65-72.

[3] Jacek Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, 2006, pp. 419-430.

[4] M.B. Cohen, M.B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *5th international symposium on software testing and analysis*, 2007.

[5] http://barbie.uta.edu/~lyu/ft.

[6] Linbin Yu, Yu Lei, Raghu Kacker, and D. Richard Kuhn, "ACTS: A Combinatorial Test Generation Tool," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013 Tools Track)*, 2013.

[7] Linbin Yu, Mehra Nouroz Borazjany, Yu Lei, Raghu Kacker, and D. Richard Kuhn, "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation," in *IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, 2013.

[8] 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, pp. 549-556.

[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*, vol. 34, pp. 633–650, 2008.

[10] B.J. Garvin, M.B. Cohen, and M.B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *1st International Symposium on Search Based Software Engineering*, 2009.

[11] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves 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, pp. 459-468.

[12] Gilles Perrouin et al., "Pairwise testing for software product lines: Comparison of two approaches," *Software Quality Journal*, vol. 20, pp. 605-643, 2012.

[13] Andrea Calvagna and Angelo Gargantini, "T-wise combinatorial interaction test suitees construction based on coverage inheritance," in *Software Testing, Verification and Reliability*, 2009.

[14] Andrea Calvagna and Angelo Gargantini, "A Logic-Based Approach to Combinatorial Testing with Constraints," in *Proceedings of the 2nd international conference on Tests and proofs*, 2008, pp. 66-83.

[15] Andrea Calvagna and Angelo Gargantini, "A Formal Logic Approach to Constrained Combinatorial Testing," *Journal of Automated Reasoning*, pp. 45:331-358, 2010.

[16] 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*, vol. 23, no. 7, pp. 437–444, 1997.