Combinatorial Testing for Tree-Structured Test Models with Constraints

Takashi Kitamura*, Akihisa Yamada*†, Goro Hatayama‡, Cyrille Artho*, Eun-Hye Choi*,
Ngoc Thi Bich Do§, Yutaka Oiwa*, Shinya Sakuragi‡
* National Institute of Advanced Industrial Science and Technology (AIST), Japan,
Email: {t.kitamura, c.artho, e.choi, y.oiwa}@aist.go.jp
† University of Innsbruck, Austria, Email: akihisa.yamada@uibk.ac.at
‡ Omron Social Solutions Co., Ltd., Japan, Email: {goro_hatayama, shinya_sakuragi}@oss.omron.co.jp
§ Posts & Telecommunications Institute of Technology, Vietnam, Email: ngocdtb@ptit.edu.vn

Abstract—**In this paper, we develop a combinatorial testing technique for tree-structured test models. First, we generalize our previous test models for combinatorial testing based on AND-XOR trees with constraints limited to a syntactic subset of propositional logic, to allow for constraints in full propositional logic. We prove that the generalized test models are strictly more expressive than the limited ones. Then we develop an algorithm for combinatorial testing for the generalized models, and show its correctness and computational complexity. We apply a tool based on our algorithm to an actual ticket gate system that is used by several large transportation companies in Japan. Experimental results show that our technique outperforms existing techniques.**

I. Introduction

Combinatorial testing (CT) is expected to reduce the cost and improve the quality of software testing [16]. Given a *test model* consisting of a list of parameter-values and constraints on them, a CT technique called *t-way testing* requires that all combinations of values of t parameters be tested at least once. Test generation for t-way testing is an active research subject. Consequently, various algorithms and tools with different strengths have been proposed so far, e. g., AETG [7], ACTS [23], [22], CASA [10], PICT [8], CIT-BACH [19], and CALOT [21].

The *Classification Tree Method* (CTM) [12], [17], [5], [14], [15] is a structured technique for test modeling in CT. The method uses *classification trees* and propositional logic constraints to describe test models. The effectiveness of CT in practice heavily depends on the quality of test modeling, while it is a difficult task requiring creativity and experience of testers [11], [1]. CTM is expected to be an effective technique to the important task, and is a key technique in CT [15].

Algorithms that generate t-way tests for such tree-structured models deserve further investigation. Tool CTE-XL [14] generates 2- or 3-way tests for CTM. In that work [14], the mechanism of test generation is explained for a simple tree without constraints; however, our interest is in mechanisms for general trees with constraints. Inspired by Oster et al. [20], we [9] took a *transformation approach.* This approach transforms tree-structured test models to non-structured ones and feeds them to standard CT tools such as the aforementioned ones [7], [8], [10], [19], [22], [21]. This approach has a clear advantage: We can leverage recent and future advances in standard CT tools. However, the technique [9] inherits the limitation from previous transformation approaches [20]; they confine the constraints to a syntactic subset of propositional logic.

The goal of this paper is to provide a transformation technique for tree-structured test models with constraints in full propositional logic, which we call T_{prop}. To this aim, we provide the following contributions:

1) We first examine a direction of translating test models of T_{prop} to one which [9] can handle (called T_{mr}). Unfortunately, we conclude this direction is not feasible; we prove that T_{prop} is strictly more expressive than T_{mr}.
2) Motivated by this fact, we develop a transformation algorithm dedicated to T_{prop}. We prove the correctness of our algorithm, showing that the semantics of test models are preserved, and the t-way coverage is ensured.
3) We further analyze the runtime complexity of our algorithm. We show that our algorithm achieves complexity $O(|N| \cdot |\phi|)$, significantly improving $O(|N|^4)$ stated by earlier work [9], where $|N|$ is the number of nodes and $|\phi|$ is the length of the constraints in a test model.
4) We implement the algorithm and conduct experiments in an industrial setting showing our technique outperforms the test generation tool for CTM [14].

This paper is organized as follows. Section II gives an overview of the proposed technique. Section III defines T_{prop}. Section IV investigates the expressiveness of T_{mr} and T_{prop}. In Section V, we explain the transformation algorithm dedicated to T_{prop}. Section VI shows experimental results of our technique in comparison with CTE-XL. Section VII discusses related work, and Section VIII concludes.

II. Transformation Approach

In this section, we overview the transformation approach for tree-structured test models. Fig. 1 shows a CALOT test model of charging IC cards in a ticket gate system for railway stations. The test model consists of two parts: an AND-XOR tree describing parameter-values, and propositional-logic constraints on them.

This work was done when the 2nd and 6th authors were in AIST.

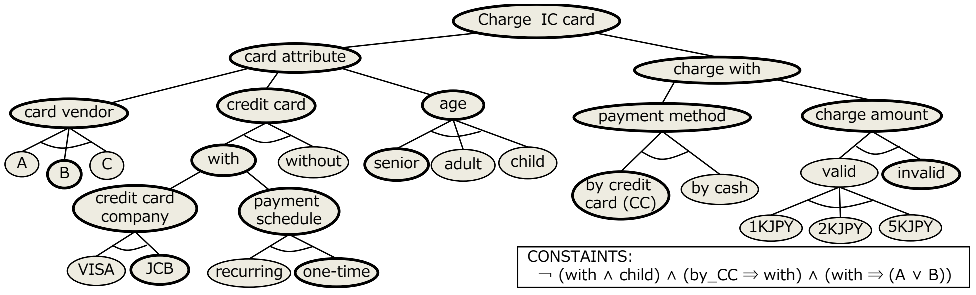

Fig. 1. A test model for an IC Card charge function.

```
# List of parameters and values
card vendor: A, B, C
credit card: with, without
age: senior, adult, child
payment method: by CC, by cash
charge amount: valid, invalid
payment schedule: recurring, one-time, -
credit card company: VISA, JCB, -
valid: 1KJPY, 2KJPY, 5KJPY, -

# CONSTRAINTS
IF [payment method] = "credit card"
    THEN [credit card] = "with";
(NOT ([age] = "child" AND [credit card] = "with"));
IF [credit card] = "with"
    THEN ([card vendor] = "A" OR [card vendor] = "B");
IF [credit card company] = "-"
    THEN (NOT [credit card] = "with")
    ELSE [credit card] = "with";
...
```

Fig. 3. PICT code for Fig. 2

The AND-XOR tree describes the basic structure of the test model by decompositional analysis of the input domain. In Fig. 1, the input domain is first decomposed into two orthogonal test aspects: card_attribute and charge_with. Further analysis decomposes the former into three aspects: the card_vendor, the availability of a credit_card function, and the age of the card holder. An arced edge denotes an XOR-composition, while its absence represents an AND-composition. Such trees specify *test models* by regarding each XOR-node as a parameter (*classification* in CTM) and its children as the values (*classes*) of the parameter.

The propositional logic constraints describe dependency among parameter-values in a test model. The constraints in Fig. 1 express the following:

1) ¬(with ∧ child): A child cannot have a card with credit_card functionality.
2) by_CC ⇒ with: To pay by credit, the card must have credit_card functionality.
3) with ⇒ (A ∨ B): credit_card functionality is available only when the IC card vendor is A or B, but not C.

Our transformation approach, using *flattening algorithm*, transforms the tree-structured test model of Fig. 1 to the *flat* test model of Fig. 2. Note that several extra nodes are introduced and constraints are manipulated, in order to keep the semantic equivalence.

The flattened tree is converted to a standard format of CT, e. g., the PICT code in Fig. 3. Correspondence between Fig. 2 and the PICT code is straightforward: Each XOR-node at the second level of the tree is a parameter in the PICT model, and children of an XOR-node are values of the parameter. Table I shows 2-way test suite for this model generated by PICT.

III. Syntax and Semantics of T_{prop}

The test modeling language T_{prop}, which is a general form of tree-structured test models including classification trees, describes AND-XOR trees and constraints using full propositional logic. The syntax of T_{prop} is defined as follows:

Definition 1 (T_{prop})**.** *The language T_{prop} is the set of tuples $(N, r, \downarrow, @, \phi)$ s. t.*

- *$(N, r, \downarrow)$ is a rooted tree, where N is the set of nodes, r is the root node, and $\downarrow$ assigns each node a the set of its children $\downarrow a$;*
- *@ assigns each node a its* node type *$@a \in \{\text{AND}, \text{XOR}, \text{LEAF}\}$;*
- *$@a = \text{LEAF}$ if and only if $\downarrow a = \emptyset$;*
- *ϕ describes constraints as a propositional formula given by the following BNF: $\phi ::= true \mid false \mid a (\in N) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi$*

We call $a \in N$ an AND-node, XOR-node, or LEAF-node if it is associated by @ with AND, XOR, or LEAF, respectively. We denote the parent of a by $\uparrow a$, if a is not the root node r. We call an element of T_{prop} a test tree/model *(of T_{prop}).*

The semantics of a test tree in T_{prop} is defined as a set of test cases. By defining a notion of a *configuration* of T_{prop}, we can derive test cases from it.

Definition 2 (Configuration)**.** *A (valid)* configuration *of a test tree $s = (N, r, \downarrow, @, \phi)$ is a subset $C \subseteq N$ of nodes that satisfies the following conditions:*

1) *The root node is in C: $r \in C$.*
2) *If a non-root node is in C, then so is its parent: $a \in C \wedge a \neq r \Rightarrow \uparrow a \in C$.*
3) *If an AND-node is in C, then so are all of its children: $(a \in C \wedge @a = \text{AND}) \Rightarrow (\forall b \in \downarrow a.\ b \in C)$.*
4) *If an XOR-node is in C, exactly one of its children is in C: $(a \in C \wedge @a = \text{XOR}) \Rightarrow (\exists! b \in \downarrow a.\ b \in C)$.*
5) *C satisfies the constraint ϕ, i. e., $C \models \phi$.*

We denote the set of all configurations of s by $\mathcal{C}(s)$.

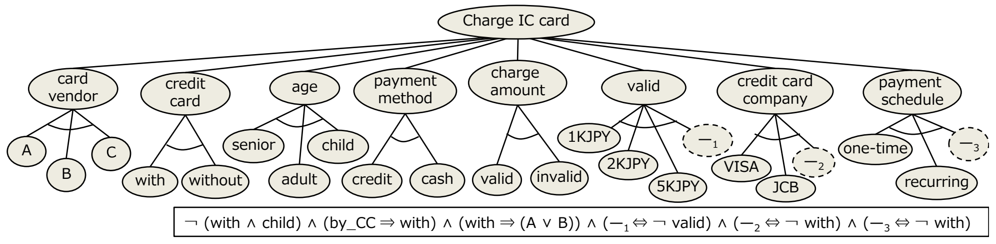

Fig. 2. The flattened tree for charge_IC_card.

TABLE I
A pair-wise test suite obtained from the test model in Fig. 1.

No.	vendor	credit card	age	payment method	charge amount	credit card company	payment schedule	valid
1	B	with	senior	by CC	invalid	JCB	one-time	-
2	A	with	adult	by cash	valid	VISA	recurring	1KJPY
3	C	without	child	by cash	valid	-	-	3KJPY
4	B	with	senior	by cash	valid	VISA	one-time	3KJPY
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

The definition of test cases assumes the basic setting of CT. Let P be a set of *parameters (classifications)* where each $p \in P$ is associated with a set V_p of *values (classes)* of p, then a *test case* is a *value assignment* γ to P, i. e., $\gamma(p) \in V_p$ for every $p \in P$. For a test tree $s \in T_{prop}$, we interpret each xor-node p of s as a parameter ($p \in P$) and its children as the values of the parameter. Some parameters may be absent in a test case, since some xor-nodes may so in a configuration. To express this, we add a special value $\perp$ to V_p for such a parameter p, representing "–" in Table I.

Definition 3 (Test cases for T_{prop})**.** *Let $s = (N, r, \downarrow, @, \phi)$ be a test tree, and C a configuration of s. The* test case tc_C *of C is the mapping on $P = \{p \in N \mid @p = \text{XOR}\}$ defined as follows:*

$$\mathsf{tc}_C(p) = \begin{cases} v \quad s.t. \quad v \in \downarrow p \cap C & \text{if } p \in C \\ \perp & \text{if } p \notin C \end{cases}$$

Note that $\mathsf{tc}_C(p)$ is uniquely defined due to condition 4 of Definition 2. The set of all the test cases of s is called the test suite *of s and denoted by $[\![s]\!]$; i. e., $[\![s]\!] = \{\mathsf{tc}_C \mid C \in \mathcal{C}(s)\}$.*

Example 1. *There are 165 configurations that satisfy the test tree in Fig. 1 according to Definition 2. The highlighted nodes in Fig. 1 constitute such a configuration, say C. By Definition 3, this configuration induces the following test case tc_C, which corresponds to the first test case in Table I:*

$$\mathsf{tc}_C = \begin{cases} \text{card\_vendor} \mapsto \text{B}, \\ \text{credit\_card} \mapsto \text{with} \\ \text{age} \mapsto \text{senior}, \\ \text{payment\_method} \mapsto \text{by\_CC}, \\ \text{charge\_amount} \mapsto \text{invalid} \\ \text{credit\_card\_company} \mapsto \text{JCB}, \\ \text{payment\_schedule} \mapsto \text{one-time} \\ \text{valid} \mapsto \perp \end{cases}$$

Definition 4 (t-tuples and t-way test suite)**.** *Let s be a test model and t a positive integer. A* t-tuple *(of values) is a value assignment on t parameters, i. e., a mapping $\tau : \pi \to N$ such that π is a set of t xor-nodes and $\tau(p) \in \downarrow p \cup \{\perp\}$. A t-tuple is* possible *if it appears in $[\![s]\!]$ and* forbidden *otherwise. A* t-way test suite *of s is a set of test cases that covers all possible t-tuples of s at least once.*

IV. Expressiveness of T_{mr} and T_{prop}

The aim of this paper is to develop a transformation technique for test generation for T_{prop} models. There are two plausible options for it. Option (a) is to convert a T_{prop} model to an "equivalent" T_{mr} model and then apply the previous technique in [9] (i. e., using the flattening algorithm for T_{mr}). Option (b) is to develop a new flattening algorithm dedicated to T_{prop}. This section shows that approaches with option (a) is infeasible or disadvantageous.

A. Correspondence and Expressiveness

First, we provide the notion of expressiveness of test modeling languages. We define it referring to a similar notion provided in [13][1]; however here, we consider it based on correspondence of test suites up to renaming.

Definition 5 (Correspondence)**.** *Let Γ and Γ' be sets of test cases on parameters P and P', respectively. We say that Γ* corresponds to Γ', *denoted by $\Gamma \simeq \Gamma'$, if and only if there exist bijections $par : P \to P'$ and $val_p : V_p \to V_{par(p)}$ for all $p \in P$ that induce a bijection $map : \Gamma \to \Gamma'$ which is defined as follows: $map(\gamma) = \gamma'$ s. t. $\gamma'(par(p)) = val_p(\gamma(p))$.*

Example 2. *Consider the test models s and $s' \in T_{prop}$ in Fig. 4. By Definition 3, $[\![s]\!] = \{\gamma_1, \gamma_2, \gamma_3\}$ and $[\![s']\!] = \{\gamma'_1, \gamma'_2, \gamma'_3\}$, where*

$$\begin{aligned} \gamma_1 &= \{a \mapsto foo,\ b \mapsto 1\} & \gamma'_1 &= \{x \mapsto 2,\ y \mapsto \alpha\} \\ \gamma_2 &= \{a \mapsto foo,\ b \mapsto 2\} & \gamma'_2 &= \{x \mapsto 2,\ y \mapsto \beta\} \\ \gamma_3 &= \{a \mapsto bar,\ b \mapsto 3\} & \gamma'_3 &= \{x \mapsto 1,\ y \mapsto \perp\} \end{aligned}$$

[1]In [13], it is defined based on correspondence of configurations without renaming.

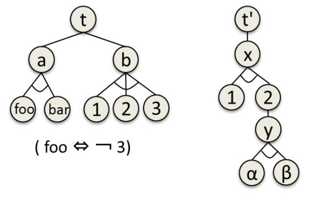

Fig. 4. A small example for $[\![s]\!] \simeq [\![s']\!]$.

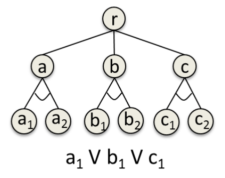

Fig. 5. A test model that cannot be expressed in T_{mr}.

Then $[\![s]\!] \simeq [\![s']\!]$. That is, we can find bijections par and val that induce a bijection map defined in Definition 5, as follows:

$$par(p) = \begin{cases} x & (\text{if } p = a) \\ y & (\text{if } p = b) \end{cases}$$

$$val_a(v) = \begin{cases} 2 & (\text{if } v = foo) \\ 1 & (\text{if } v = bar) \end{cases} \qquad val_b(v) = \begin{cases} \alpha & (\text{if } v = 1) \\ \beta & (\text{if } v = 2) \\ \bot & (\text{if } v = 3) \end{cases}$$

One may think that correspondence should be checked regarding the t-way tests for all t ($\leq$ the number of parameters). However, this is not needed, as the next theorem states:

Theorem 1. *Suppose that $[\![s]\!] \simeq [\![s']\!]$ is derived by a bijection map, and Γ a t-way test suite of s. $\Gamma' = \{map(\gamma) \mid \gamma \in \Gamma\}$ is a t-way test suite of s'.*

Proof. We show that any possible t-tuple τ' of s' is covered by Γ'. Take any possible t-tuple τ' of s'. Let τ be the t-tuple of s which corresponds to τ', i. e., $map(\tau) = \tau'$ (here, *map* is naturally extended for t-tuples). First, we show that τ is possible in s. Since τ' is possible, it appears in some test case $\delta' \in [\![s']\!]$. By assumption, we have a corresponding test case $\delta \in [\![s]\!]$ s. t. $map(\delta) = \delta'$. Because $\delta \in [\![s]\!]$, all tuples in δ are possible including the corresponding t-tuple τ of s. Next, we show that τ' is covered by Γ'. Since Γ covers all t-tuples, there must exist a test case $\gamma \in \Gamma$ that covers τ. It is obvious that $map(\gamma)$ covers τ', and hence τ' is covered by Γ'. □

Definition 6 (Expressiveness)**.** *Let T and T' be languages. We say T is* at least as expressive as *T', denoted by $T \gtrsim T'$, if for any $s' \in T'$ there exists $s \in T$ s. t. $[\![s]\!] \simeq [\![s']\!]$. We say T is* (strictly) more expressive than *T', denoted by $T > T'$, if $T \gtrsim T'$ and $T \not\lesssim T'$; and T is* as expressive as *T', denoted by $T \simeq T'$, if $T \gtrsim T'$ and $T \lesssim T'$.*

Proposition 1. *The relations $\lesssim$, $\simeq$ and $<$ are transitive.*

B. T_{prop} is more expressive than T_{mr}, i. e., $T_{mr} < T_{prop}$

Here, we compare the expressiveness of T_{mr} and T_{prop}. We first revisit T_{mr} [9].

Definition 7 (T_{mr})**.** *The language T_{mr} is the subset of T_{prop} consisting of tuples of form $s = (N, r, \downarrow, @, \phi_{mr})$, where ϕ_{mr} is a conjunction of formulas of form "$\neg(a \wedge b)$" or "$a \Rightarrow b$" with $a, b \in N$. We call a constraint of the former or latter form a $\overset{\text{MEX}}{\longleftrightarrow}$- or $\overset{\text{REQ}}{\longrightarrow}$-constraint, and write "$a \overset{\text{MEX}}{\longleftrightarrow} b$" or "$a \overset{\text{REQ}}{\longrightarrow} b$", respectively.*

Here we provide three lemmas, whose proofs are found in Appendix. The first lemma tells that an AND-node coincides with all its children in a configuration.

Lemma 1. *Let $s = (N, r, \downarrow, @, \phi)$ be an AND-XOR tree and C be a configuration of s. For any $n \in N$ and $n' \in \downarrow n$ s. t. $@n =$ AND, $n \in C$ if and only if $n' \in C$.*

The next lemma states that nodes that do not occur in any configuration can be removed from a test tree.

Lemma 2. *Let $t_1 = (N, r, \downarrow, @, \phi_{mr}) \in T_{mr}$, and $N' \subset N$ be the set of nodes which never appear in $C(t_1)$. Let $t_2 \in T_{mr}$ be the tree obtained from t_1 by removing all nodes in N' and all $\overset{\text{REQ}}{\longrightarrow}$- and $\overset{\text{MEX}}{\longleftrightarrow}$-constraints which involve nodes in N', i. e., $a \overset{\text{MEX}}{\longleftrightarrow} b$ and $a \overset{\text{REQ}}{\longrightarrow} b$ for some $a \in N'$ or $b \in N'$. Then $C(t_1) = C(t_2)$.*

The following lemma ensures that any subtree that does not contain XOR-nodes can be reduced to a single node by manipulation of constraints.

Lemma 3. *Let $t_1 \in T_{mr}$, and t'_1 be a subtree of t_1 that does not contain XOR-nodes. We denote the root node of t'_1 by n and the set of nodes of t'_1 by N'. Then $[\![t_1]\!] = [\![t_2]\!]$, where t_2 is obtained by sequentially applying to t_1 the following:*

1) *For each $\overset{\text{REQ}}{\longrightarrow}$-constraint $a \overset{\text{REQ}}{\longrightarrow} b$ in t_1, replace it with $n \overset{\text{REQ}}{\longrightarrow} b$ if $a \in N' \setminus \{n\}$ and by $a \overset{\text{REQ}}{\longrightarrow} n$ if $b \in N' \setminus \{n\}$.*
2) *For each $\overset{\text{MEX}}{\longleftrightarrow}$-constraint $a \overset{\text{MEX}}{\longleftrightarrow} b$ in t_1, replace it with $n \overset{\text{MEX}}{\longleftrightarrow} b$ if $a \in N' \setminus \{n\}$ and by $a \overset{\text{MEX}}{\longleftrightarrow} n$ if $b \in N' \setminus \{n\}$.*
3) *Remove all the nodes in $N' \setminus \{n\}$ from t_1.*

The following theorem shows that the transformation technique developed for T_{mr} [9] is not applicable to T_{prop} in general. The proof exemplifies a test model in T_{prop} and shows that it cannot be expressed in T_{mr}.

Theorem 2. *T_{prop} is more expressive than T_{mr}, i. e., $T_{mr} < T_{prop}$.*

Proof. It is easy to show that $T_{mr} \lesssim T_{prop}$, since $a \overset{\text{MEX}}{\longleftrightarrow} b$ and $a \overset{\text{REQ}}{\longrightarrow} b$ are merely $\neg(a \wedge b)$ and $a \Rightarrow b$ respectively. In the following, we show $T_{prop} \not\lesssim T_{mr}$ by giving a tree $s \in T_{prop}$ which cannot be expressed in T_{mr}.

Take the tree in Fig. 5 as s. There are the following seven test cases in $[\![s]\!]$:

$$\begin{aligned}
\gamma_1 &= \{\ a \mapsto a_1,\ b \mapsto b_1,\ c \mapsto c_1\ \} \\
\gamma_2 &= \{\ a \mapsto a_1,\ b \mapsto b_1,\ c \mapsto c_2\ \} \\
\gamma_3 &= \{\ a \mapsto a_1,\ b \mapsto b_2,\ c \mapsto c_1\ \} \\
\gamma_4 &= \{\ a \mapsto a_1,\ b \mapsto b_2,\ c \mapsto c_2\ \} \\
\gamma_5 &= \{\ a \mapsto a_2,\ b \mapsto b_1,\ c \mapsto c_1\ \} \\
\gamma_6 &= \{\ a \mapsto a_2,\ b \mapsto b_1,\ c \mapsto c_2\ \} \\
\gamma_7 &= \{\ a \mapsto a_2,\ b \mapsto b_2,\ c \mapsto c_1\ \}
\end{aligned}$$

Assume that there exists a tree $s' \in T_{mr}$ such that $[\![s]\!] \simeq [\![s']\!]$ is derived by bijections *par* and *val*. Let us write $par(p) = p'$, $val(v) = v'$, and $map(\gamma) = \gamma'$. Note that b'_1 cannot be an ancestor of a', since in that case Definitions 2 and 3 impose "$\gamma'(b') \neq b'_1 \Rightarrow \gamma'(a') = \bot$ for every $\gamma' \in [\![s']\!]$", which contradicts either the test case γ_3 or γ_7. Similarly, b'_2 cannot be

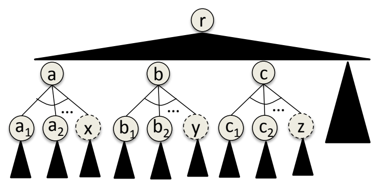

Fig. 6. Required tree structure for test models that correspond to Fig. 5.

an ancestor of a'. Thus, we know $a' \neq LCA(a', b') \neq b'$, where $LCA(a', b')$ expresses the *least common ancestor* of a' and b'. Analogously, $b' \neq LCA(b', c') \neq c'$ and $c' \neq LCA(c', a') \neq a'$. Moreover, the XOR-nodes in s' are exactly a', b', and c', since there can be only these three parameters in test cases of s'. This also entails that a'_i, b'_i, and c'_i are not $\perp$.

Since $\gamma'_1, \gamma'_5 \in [\![s']\!]$ with $\gamma'_1(a') = a'_1$ and $\gamma'_5(a') = a'_2$, it follows that $a'_1, a'_2 \in {\downarrow}a'$ in s'. Analogously, $b'_1, b'_2 \in {\downarrow}b'$ and $c'_1, c'_2 \in {\downarrow}c'$.

We conclude that the structure of s' is as shown in Fig. 6, where black subtrees contain only AND- or LEAF-nodes. Dashed nodes x, y, and z must not appear in any test case, as $[\![s']\!]$ contains only $a'_1, a'_2, b'_1, b'_2, c'_1$, and c'_2. Thus, we consider the tree in Fig. 5 instead of Fig. 6, as ensured by Lemma 2 and Lemma 3.

If s' has no constraint, then $[\![s']\!]$ contains eight test cases. On the other hand, $[\![s]\!]$ has seven test cases; thus $[\![s]\!] \neq [\![s']\!]$. If s' includes at least one constraint, then the size of $[\![s']\!]$ is either 8 or at most 6. Why? Let α be the number of removed test cases by adding one constraint to s'. In the case for $u \overset{\text{REQ}}{\to} v$,

$$\alpha = \begin{cases} 0 & \text{if } v \in Y \lor u = v \\ 2 & \text{if } v \in X \land u \in X \land {\uparrow}u \neq {\uparrow}v \\ 4 & \text{if } v \in X \land u \in Y \\ 8 & \text{if } v \in X \land u \in X \land {\uparrow}u = {\uparrow}v \land u \neq v \end{cases}$$

where $X = \{a'_1, a'_2, b'_1, b'_2, c'_1, c'_2\}$ and $Y = \{r, a', b', c'\}$. For a $\overset{\text{MEX}}{\leftrightarrow}$-constraint, α is either 2, 4, or 8. Note that by adding $\overset{\text{MEX}}{\leftrightarrow}$- and/or $\overset{\text{REQ}}{\to}$-constraints, the number of test cases cannot increase. Hence the size of $[\![s']\!]$ cannot be 7, and $[\![s]\!] \neq [\![s']\!]$. □

Theorem 2 shows the impossibility of translation of a T_{prop} model to an equivalent T_{mr} model based on the defined notion of "equivalence" in Definition 6, and hence the infeasibility of option (a). Here, one may think of another elaborated technique for such a translation, that may enable the transformation approach of option (a). The technique is to introduce extra "dummy" nodes to realize translation of T_{prop} to T_{mr} preserving equivalence. That is, additional nodes are introduced in test models of T_{mr} to realize such a translation of T_{prop} to T_{mr} models, and these nodes are also taken into account also for the generation of t-way combinatorial test suites, but finally are filtered out in the resulting test suite.

However, in this paper we do not tackle this approach, at least as the first attempt to achieve our goal. It is since we guess this approach with such an elaboration for sticking to option (a) is less advantageous than option (b). The first reason for this is that additional "dummy" nodes introduced for translation in option (a) become "dummy" parameters and values in standard test models processed for generation of t-way combinatorial test suites. These extra elements may cause undesirable side effects of generation of larger sized test suites than necessary as well as higher computation costs for test generation. Contrarily, the technique of option (b), that develops a flattening algorithm for T_{prop}, can do without such an elaboration, and thus can avoid such side effects. The second reason is that, as we will show in Section V-D, the flattening algorithm for T_{prop} developed in option (b) achieves complexity $O(|N| \cdot |\phi|)$, which is lower than that of the algorithm for T_{mr} [9] analyzed as $O(|N|^2 \cdot |\phi|)$.

Algorithm 1: *remove-and-seq*$(N, r, \downarrow, @, \phi; a)$

```
    Input: A tree $s = (N, r, \downarrow, @, \phi)$ in $T_{prop}$ and a target $a \in N$
    Output: A tree $s$ in $T_{prop}$, without AND-sequences below $a$
1   foreach $b \in {\downarrow}a$ do
2       $(N, r, \downarrow, @, \phi) \leftarrow$ remove-and-seq$(N, r, \downarrow, @, \phi; b)$
3       if $@a = @b =$ AND then
4           $\phi \leftarrow \phi|_{b \to a}$
5           ${\downarrow}a \leftarrow {\downarrow}a \setminus \{b\} \cup {\downarrow}b$; $N \leftarrow N \setminus \{b\}$
6   return $(N, r, \downarrow, @, \phi)$
```

V. Flattening Algorithm for T_{prop}

Motivated by the result in the previous section, we develop a *flattening algorithm* for T_{prop} that inputs a test tree in T_{prop} and then transforms it into an equivalent *flat one* in T^f_{prop}. We also show its correctness proof and complexity analysis.

A. Outline of the flattening algorithm

First we define the *flat* test trees as follows:

Definition 8 (T^f_{prop}). *Language T^f_{prop} is a subclass of T_{prop} s. t. (1) the root node is an* AND*-node, (2) all the nodes in the second level are* XOR*-nodes, and (3) all the nodes in the third level are* LEAF*-nodes.*

The height of a tree in T^f_{prop} is always two; hence we call such trees '*flat*'. Flat test trees have the same structure as test models in existing combinatorial testing tools, such as PICT [8], ACTS [23], [22], CIT-BACH [19], AETG [7], and CALOT [21]. Thus, tests can be generated by feeding a flattened test model to these tools.

Next, we show that every test tree in T_{prop} can be transformed to a flat one. In other words, we prove the following:

Theorem 3. $T_{prop} \lesssim T^f_{prop}$.

The proof of this theorem is to demonstrate the correctness of the transformation, achieved by algorithm *flatten* we develop. The algorithm applies three sub-algorithms, *remove-and-seq*, *remove-xor-seq*, and *lift* in this order, but the order of the first two can be swapped.

Note that we only consider test trees whose root node is an AND-node here; test trees whose root node is an XOR-node can be handled simply by inserting an AND-node above the root [9].

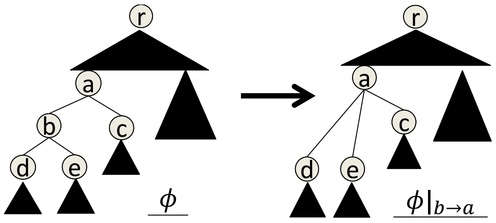

Fig. 7. remove-and-seq.

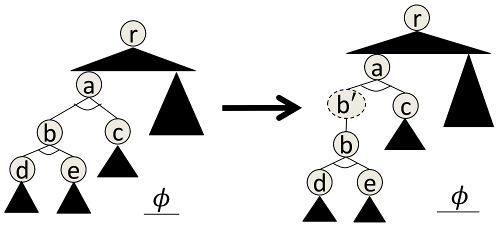

Fig. 8. remove-xor-seq.

Algorithm 2: *remove-xor-seq*$(N, r, \downarrow, @, \phi; a)$

```
    Input: A tree $s = (N, r, \downarrow, @, \phi)$ in $T_{prop}$ and $a \in N$
    Output: A tree $s'$ in $T_{prop}$, without XOR-sequences below $a$
1   foreach $b \in \downarrow a$ do
2       $(N, r, \downarrow, @, \phi) \leftarrow$ remove-xor-seq$(N, r, \downarrow, @, \phi; b)$
3       if $@a = @b =$ XOR then
4           $N \leftarrow N \cup \{b'\}$ where $b'$ is a fresh node
5           $@b' \leftarrow$ AND; $\downarrow b' \leftarrow \{b\}$; $\downarrow a \leftarrow \downarrow a \setminus \{b\} \cup \{b'\}$
6   return $(N, r, \downarrow, @, \phi)$
```

Algorithm *remove-and-seq* (see Algorithm 1 and Fig. 7), removes all AND-sequences below a given target node a in a tree s. When consecutive AND-nodes a and b are found, it deletes the second AND-node b. Then, all the occurrences of the second AND-node b in ϕ are replaced by its parent a. By recursively visiting all the nodes of the input tree starting from the root, *remove-and-seq*$(N, r, \downarrow, @; r)$ removes all AND-sequences from the input tree $(N, r, \downarrow, @)$.

Algorithm *remove-xor-seq* (see Algorithm 2 and Fig. 8) removes XOR-sequences in a given tree, but it does so differently from *remove-and-seq*. When it finds a consecutive XOR-nodes a and b, a fresh AND-node b' is inserted in between. This approach differs from others [20], [9], which remove the second XOR-node in an XOR-sequence. As the correspondence between two test suites requires a bijection between their parameters, deleting XOR-nodes (i. e., parameters) is not appropriate in this setting. It is not obvious if a t-way test suite is preserved, even when the number of parameters has changed. On the other hand, our algorithm is shown to preserve this equivalence (see Theorem 1 and Theorem 4).

Algorithm *lift* is applied after the previous two algorithms; it requires that the input tree be free of AND- and XOR-sequences (i. e., trees where AND-nodes and XOR-nodes appear alternately). If there exists an XOR-AND-XOR-sequence of nodes a, b, and c, then the second XOR-node c is 'lifted' to a direct child of the root node r. However, this operation causes c to always appear in a configuration, which should not be the case if its parent b is not in the configuration. Thus, every occurrence of c in constraint ϕ is replaced by b, which is equivalent according to Lemma 1. Furthermore, in order to express the case where b is not in the configuration, an extra leaf node $\tilde{b}$ is added as a child of c (see Fig. 9). The new option $\tilde{b}$ should be chosen if and only if b is not in the configuration, which is ensured by adding constraint $b \Leftrightarrow \neg\tilde{b}$ as a conjunction to constraint ϕ.

Algorithm 3: *lift*(s).

```
1   function lift$(s)$
        Input: A tree $s = (N, r, \downarrow, @, \phi)$ in $T_{prop}$ without AND- and
               XOR- sequences
        Output: A tree $s'$ in $T^{f}_{prop}$
2       $O \leftarrow \emptyset$; $H \leftarrow \emptyset$; $\psi \leftarrow$ TRUE // global variables
3       foreach $a \in \downarrow r$ do  lift-sub$(N, r, \downarrow, @, \phi; a)$
4       $\downarrow r \leftarrow \downarrow r \cup O$
5       return $(N \cup H, r, \downarrow, @, \phi \wedge \psi)$
6   subfunction lift-sub$(N, r, \downarrow, @, \phi; a)$
7       foreach $b \in \downarrow a$ do
8           foreach $c \in \downarrow b$ do
9               $(N, r, \downarrow, @, \phi) \leftarrow$ lift-sub$(N, r, \downarrow, @, \phi; c)$
10              $H \leftarrow H \cup \{\tilde{b}\}$ where $\tilde{b}$ is a fresh node
11              $@\tilde{b} \leftarrow$ LEAF; $\downarrow c \leftarrow \downarrow c \cup \{\tilde{b}\}$; $O \leftarrow O \cup \{c\}$
12              $\psi \leftarrow \psi \wedge (b \Leftrightarrow \neg\tilde{b})$; $\phi \leftarrow \phi|_{c\to b}$
13          $\downarrow b \leftarrow \emptyset$; $@b \leftarrow$ LEAF
```

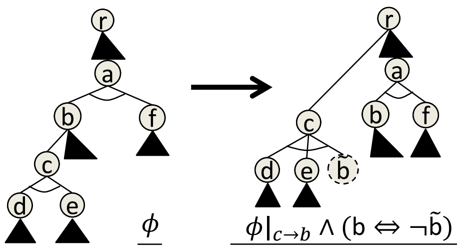

Fig. 9. The main process of lift(s).

Algorithm 3 uses the sub-algorithm *lift-sub*, which takes a tree s and an XOR-node a, indicating the target of "lifting". It recursively applies itself to grandchildren of a, which are again XOR-nodes due to the pre-condition. We prepare three global variables O, H, and ϕ, in order to store "lifted" XOR-nodes (e. g. c in Fig. 8), newly-created LEAF-nodes (e. g. $\tilde{b}$), and formulas added in the procedure (e. g. $b \Leftrightarrow \neg\tilde{b}$), respectively.

B. Correctness proof

The correctness of our new algorithm is stated formally as follows:

Theorem 4. *For $s \in T_{prop}$, $[\![s]\!] \simeq [\![flatten(s)]\!]$.*

The theorem is proved by showing the correctness of the three sub-algorithms *remove-and-seq*, *remove-xor-seq*, and *lift*. Here we present a correctness proof only for the *lift* algorithm. Correctness proofs of the other two, which are not as obvious as one may expect, can be found in Appendix.

Lemma 4. *For $s \in T_{prop}$ without* AND- *and* XOR-*sequences,* $[\![s]\!] \simeq [\![lift(s)]\!]$.

Proof. The process of algorithm *lift* is an iteration of lifting each XOR-node at the third level to the first level (see lines 7–15 in Algorithm 3). We prove that every time the procedure is applied, the set of test cases remains unchanged. Suppose that the procedure is applied to an XOR-AND-XOR sequence of nodes a, b, and c in a tree $t_1 \in T_{prop}$ (left part of Fig. 9), yielding a tree t_2 (right part of Fig. 9).

Let us split the set of configurations for t_1 into the following two subsets: The set of configurations (1) without b: $\{C \in \mathcal{C}(t_1) \mid b \notin C\}$ and (2) with b: $\{C \in \mathcal{C}(t_1) \mid b \in C\}$. We split t_2 analogously: The set of configurations (3) without b: $\{C \in \mathcal{C}(t_2) \mid b \notin C\}$ and (4) with b: $\{C \in \mathcal{C}(t_2) \mid b \in C\}$. Based on this case analysis, the following can be induced:

(i) (2) and (4) are equivalent. ∵ In both sets (2) and (4), b is included in the configurations. This means that c is included in both sets of configurations as well. The only difference that may arise in such a situation is that $\tilde{b}$ may be included in the configurations in (4), but not in the configurations in (2). However, this difference never occurs, due to the added constraint $b \Leftrightarrow \neg\tilde{b}$.

(ii) The set of configurations obtained by adding c and $\tilde{b}$ to each configuration in (1) is equivalent to (3). ∵ Since b is not included in any of these configurations, we can safely remove the sub-trees below b in both t_1 and t_2. Then, the only difference that may arise between such trees is that a child d_i of c may appear in a configuration C in (3) but not in (1). However, due to the added constraint $b \Leftrightarrow \neg\tilde{b}$, $\tilde{b} \in C$. Hence, d_i cannot appear in C due to Definition 2. On the other hand, c and $\tilde{b}$ are always included in the configurations.

Test cases are preserved in both (i) and (ii). For (i), the sets of configurations in t_1 and t_2 are equivalent, and the differences in the tree structures between t_1 and t_2 do not affect test cases (since they do not affect the relationship between XOR-nodes and their children). For (ii), the only difference between the sets of configurations (2) and (4) is that $\tilde{b}$ is included in (4) but not in (2).

Since c is not in (2), $\mathsf{tc}_C(c) = \bot$ for every C in (2). Hence, we conclude the claim by considering in Definition 3 the identity *par* and the following val_p:

$$val_p(x) = \begin{cases} \tilde{b} & \text{if } p = c \text{ and } x = \bot \\ x & \text{otherwise} \end{cases}$$ □

C. Complexity analysis

In this section, we analyze the runtime complexity of our new flattening algorithm. We state the main result first.

Theorem 5. *The computational complexity of flatten(s) is $O(|N| \cdot |\phi|)$ for $s = (N, r, \downarrow, @, \phi)$.*

The proof is done by showing the complexity of the three sub-algorithms. Here we only show the complexity analysis for *lift*, which is the most interesting part. The other two can be analyzed in a straightforward manner.

Lemma 5. *The complexity of lift$(N, r, \downarrow, @, \phi)$ is $O(|N| \cdot |\phi|)$.*

Proof. Let $T(s)$ denote the complexity of *lift*(s) and $T'(s, a)$ that of *lift-sub*$(s; a)$.

First, we show that $T'(s, a)$ is $O(|N'| \cdot |\phi|)$ by induction on N', where N' denotes the set of the nodes below a.

The algorithm *lift-sub* iteratively applies the operations of lines 11–14 to all grandchildren $c_1, \ldots, c_k$ of the given node a. The worst-case complexity of line 14 is $O(|\phi|)$, while lines 11–13 can be done in constant time. Accordingly, line 14 dominates the complexity of this procedure. We obtain the following, where each N_i is the set of the nodes below c_i:

$$\begin{aligned} T'(s, a) &= T'(s, c_1) + O(|\phi|) + \cdots + T'(s, c_k) + O(|\phi|) \\ &= T'(s, c_1) + \cdots + T'(s, c_k) + t \cdot O(|\phi|) \end{aligned}$$

By applying the induction hypothesis to each $T(s, c_i)$, we proceed as follows:

$$\begin{aligned} &= O(|N_1| \cdot |\phi|) + \cdots + O(|N_k| \cdot |\phi|) + t \cdot O(|\phi|) \\ &= (O(|N_1|) + \cdots + O(|N_k|)) \cdot O(|\phi|) + t \cdot O(|\phi|) \\ &= O(|N_1| + \cdots + |N_k| + t) \cdot O(|\phi|) \end{aligned}$$

Finally, *lift* calls *lift-sub* for all children $a_1, \ldots, a_k$ of the root node r. Thus, the complexity of *lift* is as follows, where N_i denotes the set of nodes below a_i:

$$\begin{aligned} T(s) &= T'(s, a_1) + \cdots + T'(s, a_k) \\ &= O(|N_1|) \cdot O(|\phi|) + \cdots + O(|N_k|) \cdot O(|\phi|) \\ &= O(|N_1| + \cdots + |N_k|) \cdot O(|\phi|) \end{aligned}$$

Since $N_1 \cup \cdots \cup N_k = N \setminus \{r, a_1, \ldots, a_k\}$, we conclude $T(s) = O(|N| \cdot |\phi|)$. □

The complexities of the other two sub-algorithms are proved in a similar manner. In summary, we have

- the complexity of *remove-and-seq*: $O(|N| \cdot |\phi|)$,
- the complexity of *remove-xor-seq*: $O(|N|)$, and
- the complexity of *lift*: $O(|N| \cdot |\phi|)$.

Here we assume set operations that add or remove one element are computable in constant time.

Note that *remove-and-seq* and *remove-xor-seq* keep the number of nodes within $O(|N|)$ and the length of constraints within $O(|\phi|)$ for input tree $s = (N, r, \downarrow, @, \phi)$. This is required to guarantee the complexity of the entire procedure.

D. Complexity analysis in comparison

The computational complexity of the new algorithm, concluded as $O(|N| \cdot |\phi|)$, is significantly lower than that of our previous work [9] as stated $O(|N|^4)$. There are two reasons for this. One reason is that our new algorithm has a clear recursive structure that allows an accurate complexity analysis. After a similar refinement, the complexity of the algorithm in [9] would be $O(|N|^2 \cdot |\phi|)$.

The other reason is an improvement of the algorithm *remove-xor-seq*. Fig. 10 illustrates the basic process of the *remove-xor-seq* counterpart in [9] (using notations of the current paper). We can observe that it collapses XOR-sequences,

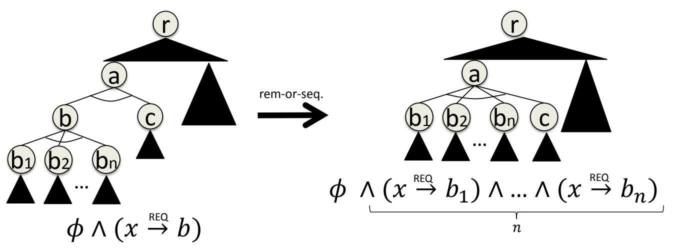
r
a
b
c
b1
b2
bn
rem-or-seq.
$\phi \wedge (x \overset{\text{REQ}}{\rightarrow} b)$
$\phi \wedge (x \overset{\text{REQ}}{\rightarrow} b_1) \wedge \ldots \wedge (x \overset{\text{REQ}}{\rightarrow} b_n)$
n

Fig. 10. remove-xor-seq(s) in [9].

but this comes at the cost of making the size of constraints larger by the order of the size of input test trees. More specifically, this results in an output test tree with constraints ϕ' whose size is $O(|N| \cdot |\phi|)$.[2] Trees output by *remove-xor-seq* is passed to the downward sub-algorithm *lift*$(N, \downarrow, @, \phi')$, whose complexity is $O(|N| \cdot |\phi'|)$. Hence, we have $O(|N| \cdot |\phi'|) = O(|N| \cdot (|N| \cdot |\phi|)) = O(|N|^2 \cdot |\phi|)$. On the other hand, the current version of *remove-xor-seq* of the flattening algorithm for T_{prop} keeps the length of constraints within $O(|\phi|)$ as well as the number of nodes within $O(|N|)$ for input tree $s = (N, r, \downarrow, @, \phi)$.

VI. Experimental Results

We have implemented the flattening algorithm (in C++) in our tool calot, which is then combined with pict (ver. 3.3) and acts (ver. 2.9). Experiments compare calot with *Classification Tree Editor* (cte-xl, ver. 3.5) [12], [17], [14], a test generator for ctm and the only competitor that can process tree-structured test models as far as we know (see Section VII).

As a benchmark set, we described test models in calot and in ctm for 18 API functions of an actual ticket gate system. Experiments were performed on a machine with Intel Core i7-4650U CPU 1.70G Hz with 8 GB Memory and Windows 7.

Table II shows the results of our experiments for 2-way and 3-way testing. Columns 2 and 3 show the sizes of the input models in T_{prop}, in terms of the number of nodes and the size of constraints ($|N|$ and $|\phi|$). Columns 4 and 5 show the sizes of the flattened models, measured by the size of the model and the constraints ($|M|$ and $|\phi'|$). For example, test model 1 written in T_{prop} contains 120 nodes and the size of the constraints is 50. It is flattened to a model $|M| = 2^5 3^{20} 4^2 5^2 6^1 10^1$ with constraints of size $|\phi'| = 90$; here x^y represents that the model has y parameters with x values. The right part compares calot with pict, calot with acts, and cte-xl, in terms of number of generated test cases and execution times (in seconds). For the numbers of test cases, the smallest ones are highlighted for each test model.

Execution time for calot includes the time for flattening and that for test generation by either pict or acts, although the time for flattening was less then 0.03 seconds for all models. For cte-xl, since it only has a GUI, we measured the execution time with a stopwatch. Timeout is set to 3,600 seconds.

The results show that calot outperforms cte-xl w. r. t. the number of generated test cases, with one interesting exception of No. 3 for 2-way testing. In some test models, the difference is considerable; e. g. in test model 4 for 3-way testing, the number of test cases calot generates is six times smaller than what cte-xl yields. Regarding execution time, although the results for cte-xl are inaccurate, calot excels over cte-xl by an order of magnitude in quite few examples. Again there is one exception (No. 2 for 2-way testing), but in this case cte-xl generates three times as many test cases as calot does.

In addition, it is worth noting that cte-xl does not support t-way testing with $t \geq 4$, while calot supports any t that the back-end engine admits. Moreover, little implementation effort enables further extensions in calot for other coverage criteria, e. g., see [8].

VII. Related Work

Test generation for t-way testing is an active research subject in CT. Consequently, various algorithms and tools with different strengths have been proposed so far, e. g., aetg [7], acts [18], [23], casa [10], pict [8], cascade [24], citbach [19], and calot [21]. For example, the algorithm in [22] can efficiently handle test models with complex constraints. The algorithm in [21] specializes in minimizing the t-way test suite within an allowed time. Our transformation approach can enjoy such various strengths of different algorithms. Note also that all of these algorithms and tools, except for cte-xl, cannot directly handle tree-structured test models.

cte-xl [14] is the only technique, except for calot, that can process tree-structured test models for t-way testing. Their test models and our T_{prop} are essentially equivalent; cte-xl uses so-called *classification trees*, while calot uses and-xor trees. The major difference appears in the algorithms for test generation. Although the algorithm in cte-xl has not been revealed in detail, from a partial explanation in [14], we can observe that it directly constructs test cases from a tree-based test model. Thus, unlike calot, cte-xl cannot benefit from the various algorithms and tools for standard t-way test case generation. Superiority of the calot approach is demonstrated through experiments in Section VI.

We have previously developed a restricted transformation algorithm for tree-structured test models with constraints [9]. The contribution of this paper over that work [9] is threefold. (1) We develop a transformation algorithm for tree-structured test models with constraints written in full propositional logic (called T_{prop}), motivated by the fact that the permitted constraints in previous work [9] (called T_{mr}) are limited to a syntactic subset of propositional logic. The proof of $T_{mr} < T_{prop}$ in Theorem 2 indicates that previous algorithms cannot be used for our goal. (2) We refine our complexity analysis. In [9], we stated that the complexity of the previous algorithm is $O(|N|^4)$. This paper shows that of the new algorithm to be $O(|N| \cdot |\phi|)$. (3) Our case study is another important contribution. By relaxing

[2]Here, note that we cannot have the following derivation, since the derived logical formula is not a legitimate constraint of T_{mr}:

$$\phi \wedge ((x \overset{\text{REQ}}{\rightarrow} b_1) \wedge (x \overset{\text{REQ}}{\rightarrow} b_2) \wedge \cdots \wedge (x \overset{\text{REQ}}{\rightarrow} b_n))$$
$$\Longleftrightarrow \phi \wedge (x \overset{\text{REQ}}{\rightarrow} (b_1 \wedge b_2 \wedge \cdots \wedge b_n))$$

TABLE II
TEST CASE GENERATION USING CALOT AND CTE-XL FOR THE TRAIN TICKET GATE SYSTEM.

	T_{prop}		Flattened model		2-way tests						3-way tests					
					Calot/PICT		Calot/ACTS		CTE		Calot/PICT		Calot/ACTS		CTE	
	$\lvert N\rvert$	$\lvert\phi\rvert$	$\lvert M\rvert$	$\lvert\phi'\rvert$	size	time	size	time	size	time	size	time	size	time	size	time
1	120	50	$2^5 3^{20} 4^2 5^2 6^1 10^1$	90	time out		**95**	91.8	time out		time out		time out		time out	
2	153	26	$2^{24} 3^{20} 7^1$	66	66	34.8	**63**	16.0	199	8.0	237	40.2	**226**	155.6	810	1103.4
3	167	22	$2^{16} 3^7 9^1 14^1 16^1 33^1$	34	964	979.8	958	5.9	**893**	128.7	9693	1125.6	**8646**	33.7	time out	
4	97	40	$2^{20} 3^7$	50	**13**	0.3	**13**	3.2	51	<1.0	35	0.4	**35**	6.6	217	274.2
5	78	12	$2^{12} 3^6 4^2 5^1$	22	26	0.3	**22**	2.5	25	<1.0	107	0.6	**102**	2.9	115	23.9
6	57	0	$2^{13} 3^2 4^1$	2	**14**	0.3	**14**	1.8	15	<1.0	47	0.3	**47**	1.9	48	2.1
7	109	20	$2^{10} 3^{13} 4^2$	44	37	0.7	**36**	7.5	80	<1.0	119	1.3	**119**	48.4	341	78.1
8	83	52	$2^{17} 3^9$	70	**15**	0.3	17	3.9	65	<1.0	48	0.5	**46**	10.5	246	54.2
9	48	6	$2^{10} 3^3$	8	14	0.3	**13**	2.0	**13**	<1.0	41	0.3	**39**	2.1	36	<1.0
10	42	0	$2^{10} 3^1$	0	**9**	0.3	10	1.0	**9**	<1.0	**24**	0.3	26	1.2	**24**	<1.0
11	62	18	$2^{12} 3^4 6^1$	26	**20**	0.3	**20**	3.1	34	<1.0	**66**	0.4	**66**	5.4	150	20.3
12	41	10	$2^9 3^2$	12	**9**	0.3	**9**	2.0	11	<1.0	**24**	0.3	26	2.1	28	<1.0
13	38	12	$2^7 3^2 5^1$	14	**15**	0.3	**15**	2.4	**15**	<1.0	**44**	0.3	45	3.2	50	<1.0
14	31	0	$2^5 3^5$	8	**18**	0.3	**18**	2.3	**18**	<1.0	**43**	0.3	49	2.6	45	<1.0
15	25	0	$2^4 3^4$	6	**12**	0.3	**12**	2.2	**12**	<1.0	**30**	0.3	36	2.5	36	<1.0
16	33	12	$2^6 3^4$	18	**12**	0.3	**12**	2.3	14	<1.0	31	0.3	**27**	2.7	38	<1.0
17	23	0	$2^3 8^1$	0	**16**	0.3	**16**	1.0	**16**	<1.0	**32**	0.3	**32**	1.1	34	<1.0
18	58	0	$3^{15} 6^1$	30	**30**	14.1	**30**	12.6	error		**36**	22.3	**36**	143.8	error	

limitations on constraints, we are now able to compare the transformation approach with CTE-XL.

Our work is inspired by Oster et al. [20], who applied CT to Software Product Lines (SPLs). An SPL is expressed as a *feature model*, an AND-OR tree with constraints. Their flattening algorithm transforms a feature model to a *flat* one, (a feature model whose height is two) and a set of products is obtained as a t-way test suite for the SPL. The class of feature models in [20] restricts the constraints to the same syntactic subset as T_{mr}. Also, we can observe that a similar technique to our flattening algorithm is used in [3] for the setting of SPL testing. Compared to these work, our contribution reveals several important facts of the proposed technique such as expressiveness, correctness, and complexity analysis. We expect that our contributions can be also used to advance their approach for SPL.

VIII. Conclusions and Future Work

This paper presented a transformation approach to generating t-way tests for tree-structured test models with propositional logic constraints. Practicality of our technique was evaluated by experiments in an industrial setting; they demonstrated superiority of our technique over the state-of-the-art tool CTE-XL. Also, the paper contains several theoretical contributions. We proved $T_{mr} < T_{prop}$; to our knowledge, there is no published proof for this non-trivial result. We also proved the correctness and complexity of the algorithm.

Prioritized t-way testing extends t-way testing with the priority notion to express importance of different test aspects [2], [6], [15]. For tree-structured test models, CTE-XL has already introduced priorities [14]. For future work, we plan to connect the notion of priority for standard t-way testing and that for tree-structured testing via our transformation approach. We will also investigate the relation between tree-structured test modeling and combinatorial testing with *shielding parameters* [4]. Shielded parameters may not appear in some test cases, depending on whether some other parameters have specified values. This feature may open up new ways to model hierarchical dependencies in tree-structured test models, but it has so far not been extensively analyzed.

Acknowledgment

We are grateful to the anonymous reviewers of the paper for their constructive and careful comments. The first author would like to thank Hitoshi Ohsaki for his support and insightful comments at an early stage of this work. This work is in part supported by JST A-STEP grant AS2524001H.

References

[1] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of ACTS: A case study. In *Proc. of ICST'12*, pages 591–600. IEEE, 2012.

[2] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Inform. Software Tech.*, 48(10):960–970, 2006.

[3] A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial testing for feature models using citlab. In *Proc. of ICST 2013 Workshops*, pages 338–347, 2013.

[4] B. Chen, J. Yan, and J. Zhang. Combinatorial testing with shielding parameters. In *Proc. of APSEC'10*, pages 280–289, 2010.

[5] T. Y. Chen, P.-L. Poon, and T. H. Tse. An integrated classification-tree methodology for test case generation. *Int. J. Softw. Eng. Know.*, 10(6):647–679, 2000.

[6] E. Choi, T. Kitamura, C. Artho, and Y. Oiwa. Design of prioritized N-wise testing. In *Proc. of ICTSS'14*, LNCS 8763, pages 186–191. Springer, 2014.

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatiorial design. *IEEE Trans. Software Eng.*, 23(7):437–444, 1997.

[8] J. Czerwonka. Pairwise testing in real world: Practical extensions to test case scenarios. In *Proc. of PNSQC'06*, 2006.

[9] N. T. B. Do, T. Kitamura, N. V. Tang, G. Hatayama, S. Sakuragi, and H. Ohsaki. Constructing test cases for N-wise testing from tree-based test models. In *Proc. of SoICT'13*, pages 275–284. ACM, 2013.

[10] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Softw. Eng.*, 16(1):61–102, 2011.

[11] M. Grindal and J. Offutt. Input parameter modeling for combination strategies. In *in Proc. of IASTED'07 on Software Engineering*, pages 255–260. ACTA Press, 2007.
[12] M. Grochtmann. Test case design using classification trees. In *Proc. of STAR 1994*, 1994.
[13] P. Heymans, P. Schobbens, J. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *Proc. of IET Software*, 2(3):281–302, 2008.
[14] P. M. Kruse and M. Luniak. Automated test case generation using Classification Trees. *Software Quality Professional*, pages 4–12, 2010.
[15] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. CRC press, 2013.
[16] 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.
[17] E. Lehmann and J. Wegener. Test case design by means of the CTE XL. In *Proc. of EuroSTAR*, 2000.
[18] Y. Lei and K.-C. Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proc. of HASE'98*, pages 254–261. IEEE, 1998.
[19] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions*, 95-A(9):1501–1505, 2012.
[20] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proc. of SPLC'10*, LNCS 6287, pages 196–210. Springer, 2010.
[21] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere. Optimization of combinatorial testing by incremental SAT solving. In *Proc. of ICST'15*, pages 1–10. IEEE, 2015.
[22] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, and D. R. Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *Proc. of ICST'13*, pages 242–251. IEEE, 2013.
[23] L. Yu, Y. Lei, R. Kacker, and D. R. Kuhn. ACTS: A combinatorial test generation tool. In *Proc. of ICST'13*, pages 370–375. IEEE, 2013.
[24] Y. Zhao, Z. Zhang, J. Yan, and J. Zhang. Cascade: A test generation tool for combinatorial testing. In *Proc. of ICST 2013 Workshops*, pages 267–270, 2013.

Appendix

A. Omitted proofs

Proof of Lemma 1. The proof is straightforward from the condition (3) in Definition 2. □

Proof of Lemma 2. Let t_1' be the tree obtained by adding a $\overset{\text{MEX}}{\leftrightarrow}$-constraint $r \overset{\text{MEX}}{\leftrightarrow} n$ for each $n \in N'$ where r is the root node. Let t_1'' be the tree obtained by removing all $\overset{\text{MEX}}{\leftrightarrow}$- and $\overset{\text{REQ}}{\rightarrow}$-constraints that involve any nodes in N' except for the $\overset{\text{MEX}}{\leftrightarrow}$-constraints added in t_1'. We prove the following:

1) $C(t_1) = C(t_1')$. Since $n \in N'$ is a node which never appears in any configuration of t_1 and the root node is always in a configuration of t_1, adding such constraints makes no difference between $C(t_1)$ and $C(t_1')$.
2) $C(t_1') = C(t_1'')$. We consider the following three forms of the constraint:
   a) $a \overset{\text{REQ}}{\rightarrow} b$ with $a \in N'$ and b is any node in t_1': Since $a \in N'$, this constraint does not affect b. Further, since $a \overset{\text{MEX}}{\leftrightarrow} r$, a still does not appear in any configuration after removing $a \overset{\text{REQ}}{\rightarrow} b$.
   b) $a \overset{\text{REQ}}{\rightarrow} b$ with a is any node in t_1' and $b \in N'$: Since $b \in N'$, $a \in N'$. Hence, the case follows case a).
   c) $a \overset{\text{MEX}}{\leftrightarrow} b$ with $a \in N'$ or $b \in N'$ and both are not r: Analogous to a).
3) $C(t_1'') = C(t_2)$. By removing all the nodes in N' and all the $\overset{\text{MEX}}{\leftrightarrow}$-constraints between the root node and nodes in N' from t_1'', we obtain t_2. The removal makes no difference between $C(t_1'')$ and $C(t_2)$, since nodes in N' are not referred to by any other constraints in t_1''. □

Proof of Lemma 3. We show that these procedures do not change the test suite. For 1) and 2), the claim follows Lemma 1. For 3), it is because (i) the subtree contains no xor-nodes and hence any nodes in $N' \setminus \{n\}$ do not appear in the test suite according to Definition 3, and (ii) no constraint involves any node in $N' \setminus \{n\}$ after applying 1) and 2). □

Lemma 6. *Let $s = (N, r, \downarrow, @, \phi) \in T_{prop}$ and $t =$ remove-and-seq$(N, r, \downarrow, @, \phi; r)$. Then, $[\![s]\!] = [\![t]\!]$ and thus $[\![s]\!] \simeq [\![t]\!]$.*

Proof. Algorithm 1 applies to every and-sequence in s the following two steps: 1) substituting the second and-node of the and-sequence in ϕ by its parent (line 4), and 2) deleting the second and-node from the tree (line 5). We show that each of these steps preserves the test suite of an input tree.

1) This step yields $s' = (N, r, \downarrow, @, \phi')$ where $\phi' = \phi|_{b \to a}$. To prove $[\![s]\!] \subseteq [\![s']\!]$, suppose $\mathsf{tc}_C \in [\![s]\!]$. Due to Lemma 1, we have $C \models \phi|_{b \to a}$ and thus $C \in C(s')$. Hence by definition, $\mathsf{tc}_C \in [\![s']\!]$. Analogously we have $[\![s]\!] \supseteq [\![s']\!]$, concluding $[\![s]\!] = [\![s']\!]$.
2) This step yields $s'' = (N \setminus \{b\}, r, \downarrow', @, \phi')$ where
$$\downarrow'(x) = \begin{cases} \downarrow(a) \setminus \{b\} \cup \downarrow(b) & \text{if } x = a \\ \downarrow(x) & \text{otherwise} \end{cases}$$
It is easy to show that $C \in C(s')$ if and only if $C \setminus \{b\} \in C(s'')$, since b does not occur in condition ϕ', which is used by both s' and s''. Below we show that $\mathsf{tc}_C = \mathsf{tc}_{C'}$ where $C' = C \setminus \{b\}$.
   - Suppose $\mathsf{tc}_C(p) = v$. Then, $p \neq b$ since b is not an xor-node, and $v \neq b$ since b is not child of an xor-node. Hence, $p, v \in C'$ and also $v \in \downarrow' p$. Accordingly, $\mathsf{tc}_{C'}(p) = v$.
   - Suppose $\mathsf{tc}_{C'}(p) = v$. Then, $v \in \downarrow' p$ and $p \neq a$ since a is not an xor-node. Thus, $v \in \downarrow p$. Also, $p, v \in C$, since $v \in C'$. Hence $\mathsf{tc}_C(p) = v$.

   This concludes $[\![s']\!] = [\![s'']\!]$. □

Lemma 7. *Let $s = (N, r, \downarrow, @, \phi) \in T_{prop}$ and $t =$ remove-xor-seq$(N, r, \downarrow, @, \phi; r)$. Then $[\![s]\!] \simeq [\![t]\!]$.*

Proof. Algorithm 2 is an iteration of the procedure shown in Fig. 8. Each time it is applied to an xor-sequence of $a \in N$ and $b \in \downarrow a$ in a tree s yielding s', we can prove $[\![s]\!] \simeq [\![s']\!]$ by regarding the identity *par* and the following val_p in Definition 5:
$$val_p(v) = \begin{cases} b & \text{if } p = a \text{ and } v = b' \\ v & \text{otherwise} \end{cases}$$
□

Proof of Theorem 4 . From Lemmas 6, 7, and 4, each of the three steps in the algorithm *flatten* preserves test cases up to correspondence. □