Constructing Test Cases for N-wise Testing from Tree-based Test Models

Thi Bich Ngoc Do
Posts & Telecommunications Institute of Technology, Vietnam
ngocdtb@ptit.edu.vn

Takashi Kitamura
National Institute of Advanced Industrial Science and Technology, Japan
t.kitamura@aist.go.jp

Van Tang Nguyen
Hanoi University of Industry, Vietnam

Goro Hatayama
Omron Social Solutions Co., Ltd, Japan

Shinya Sakuragi
Omron Social Solutions Co., Ltd, Japan

Hitoshi Ohsaki
National Institute of Advanced Industrial Science and Technology, Japan

ABSTRACT

In our previous work [17], we proposed a model-based combinatorial testing method, called FOT. It provides a technique to design test-models for combinatorial testing based on extended logic trees. In this paper, we introduce pair-wise testing (and by extension, n-wise testing, where n = 1,2,..) to FOT, by developing a technique to construct a test-suite of n-wise strategies from the test models in FOT. We take a "transformation approach" to realize this technique. To construct test suites, this approach first transforms test-models in FOT, represented as extended logic trees, to those in the formats which the existing n-wise testing tools (such as PICT [9], ACTS [30], CIT-BACH [31], etc.) accept to input, and then applies transformed test-models to any of these tools. In this transformation approach, an algorithm, called "flattening algorithm", plays a key role. We prove the correctness of the algorithm, and implement the algorithm to automate such test-suite constructions, providing a tool called FOT-nw (FOT with n-wise). Further, to show the effectiveness of the technique, we conduct a case study, where we apply FOT-nw to design test models and automatically construct test suites of n-wise strategies for an embedded system of stationary services for real-use in industry.

Categories and Subject Descriptors

D.2.5 [**Software**]: Software Engineering—*Testing and Debugging*

General Terms

Algorithm, Design, Performance

Keywords

Automated test case generation, black box testing, n-wise testing

1. INTRODUCTION

N-wise testing [19, 21] is a combinatorial technique for software testing. It is a testing strategy, or coverage criteria, for constructing a test-suite using combinatorial techniques, i.e., given a test model[1], which consists of a list of parameters with a set of its values and often with constraints (a.k.a., *forbidden rules*; expressed usually as a propositional formula), it stipulates that all combinations of values of n parameters should be covered at least once by test cases. The rationality of the technique is proved by empirical studies [19, 21], which show that most faults are caused by interactions among six or fewer parameters. A number of tools for n-wise testing has been developed so far, including PICT [9], ACTS [30], CIT-BACH [31] and AETG [5, 6]. These tools automatically construct a test-suite (a collection of test cases) of the n-wise testing for a given test model.

Adopting n-wise testing in real-world developments is one of the most important and difficult tasks in designing high-quality, i.e., correct, test-models. It is an important task, since the correctness of test-suites depends on the correctness of the test model. The correctness of a test-model can be measured by how much it satisfies the MECE (Mutually Exclusive and Collectively Exhaustive) principle. An omission in a test-model causes a lack of test-cases, which may outflow faults. A redundancy in a test-model causes duplicate of test-cases, which may increase the costs for testing. On the other hand, designing a correct test-model is a difficult task because the task is about a design, which requires iterative design decisions according to the creativity and experience of the testers. Hence, the qualities of tests often cannot be stabilized depending on testers, as different testers come up with different (qualities of) test models. Similar problems have been pointed out in [3, 12]. Despite the effectiveness of n-wise testing, the difficulty of the technique to design correct test-models creates a significant barrier to preventing wider adoption of the technique in real-world de-

[1]This is called the "Input Parameter Model (IPM)" in [12]

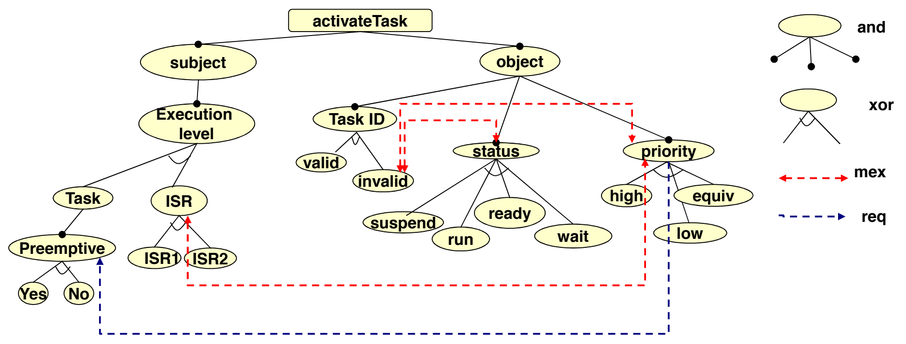

Figure 1: A test model for API function "activateTask" of OSEK-OS

velopments.

Toward this problem, in [17] we propose a model-based combinatorial testing method, called FOT (Feature Oriented Testing). Aiming to assists testers to design correct test-models, this method provides a design method to systematically design test-models using extended logical trees, named $\mathcal{T}_{(mex,\ req)}$. Figure 1 shows a simplified test model for an API function "activateTask(taskID)"[2] with $\mathcal{T}_{(mex,\ req)}$, with its graphical representation. Test models by $\mathcal{T}_{(mex,\ req)}$ are basically designed as *and-xor logic trees.* The basic structures of the test models form *trees* as a result of the analysis of the *recursive partitioning*, which repeatedly breaks down the test object (here, the input domain of "activateTask") by various aspects of test-concerns with two kinds of partioning notions (*and*, and *xor*-partitions). Also, the logic trees of $\mathcal{T}_{(mex,\ req)}$ are extended with two kinds of constraints "*mutex (shortly, mex)*" and "*requires (shortly, req)*" called *cross-tree constraints* (CTCs), which specify detailed possible and/or forbidden relations between any aspects in the tree. (The account of the test-model represented by a tree of $\mathcal{T}_{(mex,\ req)}$ is given in the next section.) Such a tree in $\mathcal{T}_{(mex,\ req)}$ is considered as a test-model, by regarding each *xor*-node in the test model as a parameter and its children as the values of the parameter. Hence test-cases can be constructed from such a test model, using combination techniques. For example, 24 test cases are constructed from the test model in Figure 1 for the full coverage. Logic trees are a mental model, often used for analysis techniques. They assist us to systematically reason about an analysis object in a top-down and recursive manner; i.e., the analysis process proceeds in a top-down manner by recursively partitioning the analysis target into sub-notions, called *recursive partitioning.* FOT aims to provide a design method which assists us to systematically design test-models. Hence, the quality of test-models is enhanced.

In this paper, we develop a technique to apply n-wise testing in the model-based combination testing method, FOT. That is, we develop a technique to construct a test-suite for the n-wise testing from the test model represented by $\mathcal{T}_{(mex,\ req)}$ in FOT. To do so, we take a *transformation approach.* In this approach, test cases for the n-wise strategy are constructed, by first transforming a test-model represented as a tree of $\mathcal{T}_{(mex,\ req)}$ to that represented in the format of the existing pair-wise tools such as PICT, ACTS, and CIT-BACH acceptable to input, and then feeding the transformed test-model to any of the these tools. To realize this approach, we develop a so-called *flattening algorithm.* This algorithm transforms test-models in the form of $\mathcal{T}_{(mex,\ req)}$ in FOT, to a "flattened tree", preserving the semantics equivalence of the trees. The *flattened trees* are a special kind of trees of $\mathcal{T}_{(mex,\ req)}$, whose height is 3-levels and the root node is an *and*-node and all the nodes in the second level are *xor* nodes. Figure 2 shows an example of a flattened tree of the tree obtained by applying the flattening algorithm to the tree in Figure 1. Test-models represented as flattened trees are regarded as the input format of the existing pair-wise tools. For example, 24 test cases are constructed with full coverage, and 12 test cases are generated for the pair-wise testing using PICT as shown in Table 1. We prove the correctness of the algorithm, and implement the algorithm to automate such test-suite constructions, providing a tool called FOT-nw (FOT with n-wise). Further, to show the effectiveness of the technique, we conduct a case study, where we apply FOT-nw to design test models and automatically construct test suites of n-wise strategies for a system of station services, which is in real-use in industry.

The remainder of this paper is organized as follows. Section 2 recalls basic notions of $\mathcal{T}_{(mex,\ req)}$ and explains how to design test models using $\mathcal{T}_{(mex,\ req)}$. In Section 3, we develop a technique to construct n-wise test cases from test models represented by $\mathcal{T}_{(mex,\ req)}$. The core of the technique is the flattening algorithm, therefore, most of this section is devoted to the algorithm. Section 4 presents the proof for correctness (sound and completeness) of the algorithm. In section 5, we show the experimental results. Section 6 discusses related works and ection 7 concludes this paper.

2. A LANGUAGE FOR TEST-MODEL DESIGN: $\mathcal{T}_{(mex,\ req)}$

This section briefly reviews $\mathcal{T}_{(mex,\ req)}$. First, we explain an example of test-model design by $\mathcal{T}_{(mex,\ req)}$, which is also used as a running example in this paper. Then, we recall the definition of $\mathcal{T}_{(mex,\ req)}$, defined with formal syntax and semantics. Readers are encouraged to refer to [17] for the details.

2.1 An example of test models by $\mathcal{T}_{(mex,\ req)}$

[2]This is one of API functions in OSEK-OS (a standard real-time operating systems for automotive systems) [23] that control the task scheduler in a operating system; the function transfers the status of a specified task with parameter "taskID" from "suspend" to "ready" state.

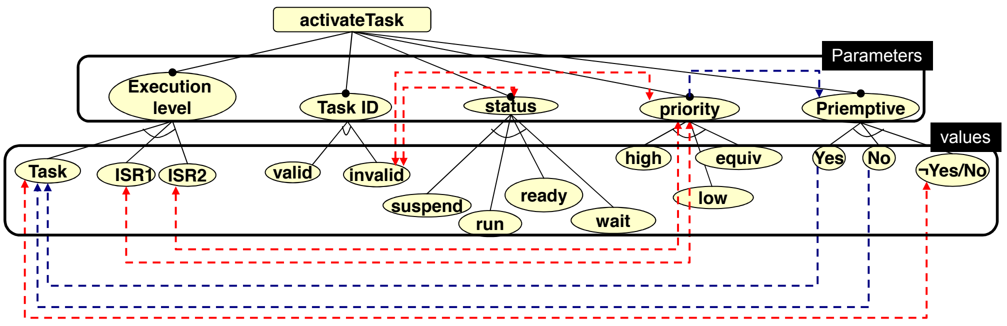

Figure 2: A flattened tree of the test-model in Figure 1

No.	1	2	3	4	5	6	7	8	9	10	11	12
Preemptive	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N
Execution level	T	T	T	T	T	T	T	T	T	T	T	T
TaskID	V	V	V	V	V	V	V	V	V	V	V	V
Status	S	S	R	W	A	A	S	R	R	W	W	A
Priority	L	E	L	H	L	E	H	H	E	L	E	H

Y: yes, N: no, T: task, V: valid, S: suspend, R: run, A: re$\underline{a}$dy, W: wait, E: equiv, H: high, L: low

Table 1: A test suite for pair-wise testing constructed from the test model in Figure 2

We design a test model for "activateTask(taskID)", a API function in OSEK-OS (A standard for real-time operating systems for automotive systems) [23]. This API function is one of the functions in OSEK-OS that controls the task schedule in OSEK-OS, which transfers the status of a specified task with the parameter "taskID" from the "suspend" to "ready" state. Figure 1 shows a simplified test-model for the function. The test-model is designed by $\mathcal{T}_{(mex,\ req)}$, which is an extension of *and-xor* logic trees. Specifically, $\mathcal{T}_{(mex,\ req)}$ is used to design test models by recursively partitioning the test object (i.e., in this example, the input domain of "activateTask") with various test-relevant aspects in a top-down manner.

We design the test model in Figure 1 by first partioning the input domain of "*activateTask(taskID)*" into two aspects of the "*subject (task)*", which is the task that calls the function, and the "*object (task)*", which is the task specified by the parameter "*taskID*" of the function as the object of the function call. Next, these two aspects are further analyzed for partitions separately. The analysis for the aspect of "*subject*' is made by the aspect of the "*Execution Level*". Then the aspect of the "*Execution Level*" is partitioned into two aspects of the "*(Normal) Task*" and the "*Interruption Service Routines (ISR)*". Further, the aspect of the"*ISR*" is partitioned into "*ISR1*" and "*ISR2*", to distinguish the two types of ISRs. The aspect of "*Task*" is next analyzed by a single aspect, i.e., "*Preemptive*" in order to consider the two cases where the API function works in "*Preemptive* (Yes)" or "*Preemptive(No)*". The test aspect of the "*object*" is analyzed in a similar manner. It is partitioned into three aspects: "*taskID*", "*status*" and "*priority*", each of which is analyzed into cases of two, four, and three test-relevant aspects, respectively. In such a way, the basic analysis with $\mathcal{T}_{(mex,\ req)}$ proceeds by *recursive partitioning*, which forms a tree structure as a test model.

In addition to such an analysis of partitions, each partition is distinguished by "and" or "xor" according to the two types of partitions. If an aspect is partitioned into aspects with orthogonal notions then it is distinguished by an *and*-partition. For example, the partitions of the test aspect of the "*object (task)*" with three aspects of "*taskID*", "*status*" and "*priority*" are assigned as an *and*-partition, since these three notions are orthogonal to each other. If an aspect is partitioned into aspects with alternative notions, then it is distinguished as an *xor*-partition. For example, the partitions of the test aspect of the "*execution level*" by the two aspects is distinguished as an *xor*-partition, since these notions are alternative to each other. All the other partitions must be also distinguished by these two types of the partitions in the same way. In such a way , the basic structure of a test model with $\mathcal{T}_{(mex,\ req)}$ forms an *and-xor* logic tree. Also, this tree forms the basic structure of a test model for combinatorial testing, by regarding each *xor*-node in the test model as a parameter and its children as the values of the parameter.

Furthermore, two kinds of constraints mex and req called *cross-tree constraints* are provided in $\mathcal{T}_{(mex,\ req)}$. A mex ("mutually exclusive") constraint, which is an undirected binary relation over nodes in a logic tree, specifies forbidding combinations of the two test-aspects (i.e., the two nodes) in constructing test cases. Let's consider the mex constraint between "*invalid*" and "*priority*". The constraint specifies avoiding combinations of these two test aspects of "*invalid*" and "*priority*" in constructing each test case. This expresses a fact induced by specifications in [23], which stipulates that it makes sense to consider the aspect of "*status*" and "*priority*", only when calling the "activateTask(TaskID)" with some "*valid*" ID number for "*taskID*". A *requires* constraint (say, "*a* req *b*"), which is a directed binary relation over nodes in a logic tree, specifying that test-aspect *a* is considered only when test-aspect *b* is considered.

Take the req constraint from "*priority*" to "*Preemptive*", for example. This expresses a fact induced by specifications in [23], which stipulates that the combination of "*priority*" and "*Preemptive*" is allowed in constructing each test case, only when "*preemptive*" is considered (i.e., the OS works with the preemptive mode). The notion of the cross-tree constraints is not quite new in fact, as a similar device is equipped with the modern tools for pair-wise testing including PICT, and ACTS, etc., as *constraints.* FOT equips the device in an integrated way with the graphical representations of the logic trees.

2.2 Syntax and Semantics of $\mathcal{T}_{(mex,\ req)}$

One characteristic of FOT is that it is formally founded. $\mathcal{T}_{(mex,\ req)}$ is defined as a *language*, with which testers design test models, with formal syntax and semantics. This subsection briefly reviews the definition of $\mathcal{T}_{(mex,\ req)}$.

Definition: [$\mathcal{T}_{(mex,\ req)}$] A language $\mathcal{T}_{(mex,\ req)}$ is a set of trees with several extensions, whose elements are expressed as a tuple $(N, r, \uparrow, @, \overset{\text{mex}}{\leftrightarrow}, \overset{\text{req}}{\rightarrow})$, such that

- $(N, r, \uparrow)$ is a rooted tree [29], where N is a set of nodes, r is the root, and $\uparrow$ is the parent-child relation such that node f is the parent of g if and only if $\uparrow(g) = f$,
- $@$ is a mapping from N to $\{\text{and}, \text{xor}, \text{leaf}\}$,
- $\overset{\text{mex}}{\leftrightarrow}$ is a symmetric binary relation on N,
- $\overset{\text{req}}{\rightarrow}$ is a binary relation on N.

We may call an element of $\mathcal{T}_{(mex,\ req)}$ a *test model* of $\mathcal{T}_{(mex,\ req)}$. We may call it a *test tree* of $\mathcal{T}_{(mex,\ req)}$, to avoid confusion with the notion of a *model* of a test tree, given in the Definition Models of $\mathcal{T}_{(mex,\ req)}$.

$\mathcal{T}_{(mex,\ req)}$ is a set of *trees* [29] extended with several notions. First, we consider $\mathcal{T}_{(mex,\ req)}$ a set of and-xor logic trees. In order to express this, we adopt a node-based design; i.e., all the nodes of trees except for leaf nodes are labelled with an"and" or "xor". The function $@ : N \rightarrow \{\text{and}, \text{xor}, \text{leaf}\}$, which labels each node with "*and*", "*xor*", or "*leaf*", is equipped for this purpose. We call a node an "and-node", "xor-node", or "leaf-node" if it is associated by @ with "and", "xor", or "leaf", respectively. Note that, due to the design, "and " and "xor"-edges shall not be mixed among the edges out-going from a node. The two kinds of CTCs, "*mutex*" and "*requires*" are expressed by the binary relations " $\overset{\text{mex}}{\leftrightarrow}$ " and " $\overset{\text{req}}{\rightarrow}$" on N.

A test tree $t \in \mathcal{T}_{(mex,\ req)}$ expresses a set of test cases. That is, the semantics of t is defined as a set of test cases. Its definition is as follows:

Definition: [Models of $\mathcal{T}_{(mex,\ req)}$] A model M of a test tree $t = (N, r, \uparrow, @, \overset{\text{mex}}{\leftrightarrow}, \overset{\text{req}}{\rightarrow}) \in \mathcal{T}_{(mex,\ req)}$ is a subset of nodes N that satisfy the following conditions:

1. The root node is in the model: $r \in M$,
2. If a node is in the model, its parent is in the model too: $(f \in M) \Rightarrow (\uparrow(f) \in M)$,
3. If an and-node is in the model, all its children are in the model too: $\Big((f \in M) \wedge (@(f) = \mathit{and})\Big) \Rightarrow \Big(\forall g.(\uparrow(g) = f) \Rightarrow (g \in M)\Big)$,
4. If an xor-node is in the model, exactly one of its children is in the model too; $\Big((f \in M) \wedge (@(f) = \text{xor})\Big) \Rightarrow \Big(\exists! g.(\uparrow(g) = f) \wedge (g \in M)\Big)$,
5. If $f \overset{\text{req}}{\rightarrow} g$ and f is in the model, g is in it too: $\Big((f \overset{\text{req}}{\rightarrow} g) \wedge (f \in M)\Big) \Rightarrow (g \in M)$,
6. If $f \overset{\text{mex}}{\leftrightarrow} g$, f and g are not both in the model: $(f \overset{\text{mex}}{\leftrightarrow} g) \Rightarrow \Big((f \notin M) \vee (g \notin M)\Big)$.

We write $M \models t$ if and only if M is a model of t.

The definitions of a test case and the test-suite of a test tree $t \in \mathcal{T}_{(mex,\ req)}$ are given by means of *models* of t.

Definition: [test case and test-suite] Let $t = (N, r, \uparrow, @, \overset{\text{mex}}{\leftrightarrow}, \overset{\text{req}}{\rightarrow}) \in \mathcal{T}_{(mex,\ req)}$, then

- a test case of t is $M \cap L$ for some M such that $M \models t$,
- the test-suite of t, denoted as $[\![t]\!]$, is the set of all the test cases of t; i.e., $[\![t]\!] = \{M \cap L \mid M \models t\}$,

where $L = \{f \in N \mid @(f) = \text{leaf}\}$.

3. CONSTRUCTING TEST CASES FOR N-WISE TESTING FROM THE TEST MODELS OF $\mathcal{T}_{(mex,\ req)}$

Exhaustive testing of computer software is intractable, but empirical studies of software failures suggest that testing can in some cases be effectively exhaustive. Data reported in [19] showed that software failures in a variety of domains were caused by combinations of relatively few conditions. That observation paved a new way for software testing called pairwise (or n-wise in general) testing. In particular, n-wise testing is a combinatorial method of software testing that, for each n-tuple of input parameters, tests all possible discrete combinations of those parameters. Using carefully chosen test vectors, this can be done much faster than an exhaustive search of all combinations of all parameters, by "parallelizing" the tests of parameter pairs. Readers are encouraged to refer to [10, 19] for details of an n-wise testing method.

In this section, we introduce n-wise testing to FOT. That is, we develop a technique to construct a test-suite of n-wise testing from the test models in FOT. We take a "transformation approach" to realize this. To construct test suites, this approach first transforms test-models in FOT, represented as extended logic trees, to those in the formats which the existing n-wise testing tools (such as PICT [9], ACTS [30], etc) accept to input, and then applies transformed test-models to any of these tools.

To realize this transformation approach, we propose an algorithm, called "flattening algorithm". This algorithm transforms test-models in the form of $\mathcal{T}_{(mex,\ req)}$ in FOT, to a "flattened tree" (or, *flat trees*), preserving the semantics equivalence of the trees. *Flattened trees* are a special kind of trees of $\mathcal{T}_{(mex,\ req)}$, whose height is tree-level, and the root node is an *and* node and the all the nodes in the second level are *xor* nodes. Figure 2 shows an example of a flattened tree of the tree of $\mathcal{T}_{(mex,\ req)}$ in Figure 1 by applying the flattening algorithm. The format of test models of the existing n-wise tools consists of a list of parameters with a set of its values and often with constraints (a.k.a., *forbidden rules*; expressed usually as a propositional formula). The flat trees can be interpreted as the format of test models of the existing n-wise tools, by regarding the *xor*-nodes at the second level of a flat tree as parameters, their children as values, and cross-tree constraints as constraints. Hence it is possible to construct test-cases for n-wise testing by feeding test-models represented by flat trees to the existing tools for n-wise testing. For example, the pairwise (2-wise) test suite, which can be constructed from the test model in Figure 1, is given in Table 1, which contains only 12 test cases.

In the following, we explain the details of the flattening algorithm. We start with some essential definitions that we will use in the rest of this paper.

Definition: [and-sequence] Given a test tree $T = (F, r, E, @, \overset{mex}{\leftrightarrow}, \overset{req}{\rightarrow})$, a pair of nodes (a, b) is called and-sequence if b is a child of a (i.e., $\uparrow(b) = a$) and $@(a) = @(b) =$ "*and*".

Definition: [xor-sequence] Given a test tree $T = (F, r, E, @, \overset{mex}{\leftrightarrow}, \overset{req}{\leftrightarrow}$, a pair of nodes (a, b) is called xor-sequence if b is a child of a (i.e., $\uparrow(b) = a$) and $@(a) = @(b) =$ "*xor*".

Definition: [Flattened Tree] A flattened tree is a tree of $\mathcal{T}_{(mex,\ req)}$ that satisfies the following conditions: (1) Its height is three; (2) the root node is an "and" node, i.e., $@(r) =$ "*and*"; and (3) the children of the root are "xor" nodes or leaves, i.e., $\forall x \cdot (r, x) \in E : @(x) =$ "*xor*" or $@(x) =$ "*leaf*".

Our procedure of generating test cases from test trees is described in Algorithm 1. The algorithm has the following main steps. First, the function **Flattening-Xor-Sequence** is called in order to transform the input test tree of $\mathcal{T}_{(mex,\ req)}$ to a tree which does not have any xor-sequences. Second, the function **Flattening-And-Sequence** is invoked to transform it to a test tree which has no and-sequences. Third, after line 3, the tree T that does not contain any xor-sequences and and-sequences, we apply the function **Lift-Tree** to transform T to a *flattened* tree T_1. Fourth, the function **transformFlattenedTreetoPICTInput** is applied to T_1 to extract and represent its leaves as the input format of the PICT system. The algorithms **Flatten-Xor-Sequence**, **Flatten-And-Sequence** , **Lift-Tree** are given in the Appendix. The PICT system is invoked to generate the set *temp* of all n-wise input parameters. Finally, the function **decodeToTestCaseofFOT** is called to transform the result *temp* back to the test cases of the FOT.

Algorithm 1: Algorithm of test cases Generation

```
Data: A general test tree $\mathbf{T} = (F, r, E, @, \overset{mex}{\leftrightarrow}, \overset{req}{\rightarrow})$
Result: A set of test cases, testcases
begin
    $T \leftarrow$ Flatten-And-Sequence($T$);
    $T \leftarrow$ Flatten-Xor-Sequence($T$);
    $T_1 \leftarrow$ Lift-Tree($T$);
    inputform-for-n-wise $\leftarrow$
    transformFlattenedTreetoPICTInput($T_1$);
    temp $\leftarrow$ callToPICT(inputform-for-n-wise);
    testcases $\leftarrow$ decodeToTestCaseofFOT(temp);
    return testcases;
end
```

Example

To illustrate how the algorithms work, we show an example of a test tree in Figure 1. This tree analyzes the input domain of activateTask [17] function in the OSEK-OS automotive operating system. The tree in Figure 3 is the result of applying function **Flatten-And-Sequence** to the test tree of $\mathcal{T}_{(mex,\ req)}$ in Figure 1. Let's consider and-sequence *(activateTask, Subject)* in Figure 1. We have: if "*activateTask*" is in the model then *Subject* is in the model too; if "*Subject*" is in the model then "*Execution_level*" is also in the model. Thus, if "*activateTask*" is in the model then all of "*Execution_level*", "*Subject*"' are in the model too. Besides, there is no constraint (e.g., $\overset{\text{req}}{\rightarrow}$and $\overset{\text{mex}}{\leftrightarrow}$), which contains "*Subject*". That means we can remove "*Subject*" and let the parent of "*Execution_level*" be "*activateTask*" without changing the set of constructed test cases.

The result of applying function **Flatten-Xor-Sequence** to the test tree (without and-sequence) of $\mathcal{T}_{(mex,\ req)}$ in Figure 3 is given in Figure 4. In Figure 3, we have an xor-sequence ("*execution_level*", "*ISR*"). We have: if "*execution_level*" is in the model then one of {"*task*', "*ISR*"} is in the model; if "*ISR*" is in the model then one of {"*ISR1*", "*ISR2*" } is in the model. Thus, if "*execution_level*" is in the model then one of {"*task*", "*ISR1*, "*ISR2*"} is in the model. Further, the constraint "*ISR* $\overset{\text{mex}}{\leftrightarrow}$*priority*" means "*ISR*", "*priority*" can not be in the model simultaneously. This constraint corresponds with the two constraints "*ISR1* $\overset{\text{mex}}{\leftrightarrow}$*priority*" and *ISR2* $\overset{\text{mex}}{\leftrightarrow}$*priority.* That means we can remove *ISR* and set the parent of "*ISR1*"; "*ISR2*" is "*execution_level*". Because "*ISR*" is removed, we also remove constraint *ISR* $\overset{\text{mex}}{\leftrightarrow}$*priority* and add the corresponding constraints *ISR1* $\overset{\text{mex}}{\leftrightarrow}$*priority* and *ISR2* $\overset{\text{mex}}{\leftrightarrow}$*priority.*

Finally, we perform lifting the tree by the function **Lift-Tree** and the flattened tree in Figure 2 is obtained. Because the level of the tree in Figure 4 is 4, we need to lift "*preemptive*" to be the child of the root "*activateTask*". To ensure the test suite does not change after lifting, we need to (1) add the constraints *yes* $\overset{\text{req}}{\rightarrow}$*Task* and *no* $\overset{\text{req}}{\rightarrow}$*Task* to ensure if "*yes*" or "*no*" is in the model then "*task*" is already in the model; (2) add one more special node, i.e., "¬ *yes/no*" and add a constraint ("*task*" $\overset{\text{mex}}{\leftrightarrow}$"¬ yes/no") to ensure if "*task*" is not in the model then "*yes*", "*no*" are also not in the model (or "¬ *yes/no*" is in the model).

4. PROVING THE CORRECTNESS

We have the following theorem as a criteria for the soundness and completeness of the flattening algorithm:

Theorem: Let $\mathbf{T} = (F, r, E, @, \overset{mex}{\leftrightarrow}, \overset{req}{\rightarrow})$ be a general test-tree of $\mathcal{T}_{(mex,\ req)}$. Let L be set of leaves of $\mathbf{T}$, and $T_1 = (F_1, r_1, E_1, @_1, \overset{mex}{\leftrightarrow}_1, \overset{req}{\rightarrow}_1)$ be the flattened tree obtained after performing algorithms **Flatten-Xor-Sequence**, **Flatten-And-Sequence** , **Lift-Tree**. The following properties hold:

1. If t_1 is a test case of T_1 then $t = t_1 \cap L$ is a test case of T, i.e., $\forall t_1 \in T_c(T_1) \exists t \in T_c(T) \cdot t = t_1 \cap L$.
2. If t is a test case of T then there exists a test case t_1 of T_1 such that $t_1 \cap L = t$, i.e., $\forall t \in T_c(T) \exists t_1 \in T_c(T_1) \cdot t_1 \cap L = t$.

Let $\mathbf{M}, \mathbf{M}_1$ be sets of all the models of T, T_1 respectively. Due to construction of algorithms **Flatten-Xor-Sequence**, **Flatten-And-Sequence** , **Lift-Tree**., there exists a bijection:

$$\phi : \mathbf{M} \rightarrow \mathbf{M}_1$$

satisfying the following conditions:

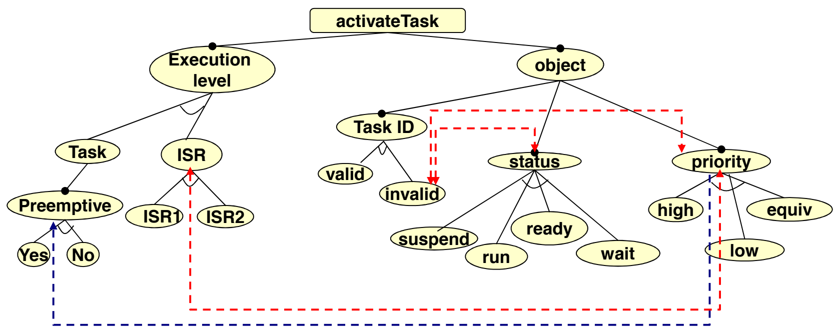

Figure 3: A test tree for “activateTask” after flattening and-sequences

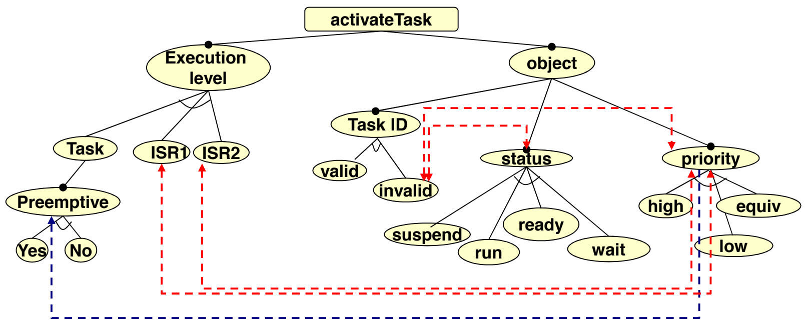

Figure 4: A test tree for “activateTask” after flattening xor-sequences and and-sequences

1. $\forall M \in \mathbf{M}$ and $\forall x \in F \cap F_1$:

$$x \in M \rightarrow x \in \phi(M)$$
$$x \notin M \rightarrow x \notin \phi(M)$$

2. $\forall M_1 \in \mathbf{M}_1$ and $\forall x \in F \cap F_1$:

$$x \in M_1 \rightarrow x \in \phi^{-1}(M_1)$$
$$x \notin M_1 \rightarrow x \notin \phi^{-1}(M_1)$$

We also use the following two properties during the proof:

- $L \subseteq L_1$ (because all leaves of T are also leaves of T_1)
- $L \subseteq F \cap F_1$ (because $L \subseteq F$, $L \subseteq L_1 \subseteq F_1$)

Proof $t = t_1 \cap L$

Let t_1 be a test case of T_1, there exists a model M_1 of T_1 such that $t_1 = M_1 \cap L_1$. Let $t = \phi^{-1}(M_1) \cap L$, t will be a test case of T. We will show $t = t_1 \cap L$.

We first show $t \subseteq t_1 \cap L$:
If $x \in t$ then $x \in \phi^{-1}(M_1) \cap L$. Thus, $x \in \phi^{-1}(M_1)$ and $x \in F \cap F_1$. From Condition 1 of ϕ, we have $x \in \phi(\phi^{-1}(M_1))$, or $x \in M_1$. Therefore, $x \in M_1 \cap L_1 \cap L$, or $x \in t_1 \cap L$.
Thus, $t \subseteq t_1 \cap L$.
We now need to prove $t_1 \cap L \subseteq t$:
If $x \in t_1 \cap L$ then $x \in F \cap F_1$ and $x \in M_1$. From Condition 2 of ϕ, we have $x \in \phi^{-1}(M_1)$. Thus, $x \in \phi^{-1}(M_1) \cap L (= t)$.
Thus, $t_1 \cap L \subseteq t$.

Proof $t_1 \cap L = t$

Let t be a test case of T. Then, there exists $M \in \mathbf{M}$ such that $t = M \cap L$. Let $t_1 = \phi(M) \cap L_1$, t_1 will be a test case of T_1. We will show $t_1 \cap L = t$.

We first show $t_1 \cap L \subseteq t$:
If $x \in t_1 \cap L$ then $x \in \phi(M) \cap L_1 \cap L$. Thus, $x \in \phi(M) \in \mathbf{M}_1$, $x \in F \cap F_1$. From Condition 2 of ϕ, we have $x \in \phi^{-1}(\phi(M)) = M$. Thus, $x \in M \cap L (= t)$.
Thus, $t_1 \cap L \subseteq t$.
We now need to prove $t \subseteq t_1 \cap L$:
If $x \in t$ then $x \in M \cap L$. Thus, $x \in F \cap F_1$ and $x \in \mathbf{M}$. From Condition 1 of ϕ we have $x \in \phi(M)$; from $x \in L$ we have: $x \in L_1$(because $L \subseteq L_1$). Thus, $x \in \phi(M) \cap L_1 \cap L = t_1 \cap L$.
Thus $t \subseteq t_1 \cap L$.

The theorem means that there is a correspondence (map) between the test suite of T and that of T_1. Therefore, in order to compute the test suite of a test tree T, we apply the flattening algorithm to transform T to the flattened tree T_1. Next, we apply the PICT algorithm to generate n-wise test cases for T_1. Finally, from the test suite of T_1, we compute the corresponding test suite of the original tree T.

Complexity analysis:.
The complexities of algorithms **Flatten-Xor-Sequence**, **Flatten-And-Sequence**, **Lift-Tree** are $O(|N|^4)$. In Algorithm 1, we need one more step call to the PICT system (to find n-wise test suite). In case n is equal to the number of parameters, PICT has to find all the test cases. Thus, the complexity of this step must be exponential time. As a result, the complexity of the whole proposed procedure is exponential time. However, in practic, we often find a pairwise (2-wise) test suite, and in this case, the complexity becomes polynomial time.

5. EXPERIMENTS

We have implemented the proposed n-wise testing for test tree algorithms in a tool called FOT-nw. The package is implemented by Java.

We applied the tool FOT-nw to generate test cases for a real IC-card system developed by our industrial partner, OMRON company. The experimental results show that, n-wise algorithms can help reduce the number of test cases. Table 2 shows the details of the experiments. Spec code column list 18 input test models given by OMRON company. Tree Nodes and CTC columns show the size of the input test model (number of nodes and number of constraints in the test tree). The SAT (FOT) column shows the number of all test cases using FOT. The 2-wise (3-wise, 4-wise, 5-wise, 6-wise) column shows the number of 2-wise (3-wise, 4-wise, 5-wise, 6-wise) test cases using FOT-nw. The experiments show that the number of test-case generated by n-wise algorithms are much smaller than the number of all test-case for test trees.

6. S

The CTM (Classification Tree Method)[14, 15, 18] is a similar technique to FOT, in the sense that it is a model-based approach to combinatorial testing technique. CTMcannot construct n-wise test-suite automatically. Besides, handling test-case generation of n-wise testing in the presence of complex constraints is not a subtle problem in CTM. As identified by [26], many of the current research directions on n-wise testing examine the specialised problem of handling constraints [4, 7, 8, 13, 22, 26]. In FOT, the algorithm is realized using the flattening algorithm; i.e., a hierarchically-structured test-model is converted to a test model in the format of the existing tools, such as PICT [9], via a flattening algorithm, and test-cases are constructed automatically feeding converted test-models to these tools.

Secondly, this work is related to the work by Oster et al. [24, 25], where [24, 25] also proposed a flattening algorithm used to construct pair-wise products for efficiently testing a software product line (SPL) by applying the technique of pair-wise testing. Oster's *flattening algorithm* is used in a similar way as ours; their flattening algorithm transforms a *feature model*, which is represented as an extended logic-tree to express a SPL to a *flat feature model* (which is extended logic trees with the height of 3-levels) to obtain a product line, which covers all pair-wise features (which are components of systems) of the product line. But differences between Oster's and our *flattening algorithms* are several-fold.

First, according to the different purposes of using extended logic trees (i.e., feature models aim to model SPLs, while our test models aim to model hierarchically structured test models for general and traditional pair-wise testing techniques extending the pair-wise tools, the extended logic trees which Oster's and our algorithms deal with are different. Extended logic-trees to represent feature models of SPLs in Oster's work [24, 25] consist of 4 operators (*mandatory, optional, alternative, or*). On the other hand, test models in our work consist of 2 kinds of *and* and *xor*-partitions to form logic trees, as an natural extension of test-models of traditional pair-wise algorithms and tools with hierarchical structures. Accordingly, this difference makes our flattening algorithm different from Oster's at a detailed level; that is, the logic trees are different; hence, their algorithm also naturally become different.

Also, our work is devoted to and contributes to developing a technique for constructing test cases for n-wise testing from tree-based test models, which also contributes to the related technique of CTM as discussed previously. In this paper, we not only show the general technique of how to construct test cases for n-wise testing from tree-based test models using the flattening algorithm, but also prove the correctness of the algorithm, develop a tool (FOT-nw) by implementing the algorithm, and conduct a case study where we apply the technique and tool to an embedded system for stationary services which is in real use in industry, showing its experimental data derived from the case study.

Logic trees are a mental model, often used for an analysis technique. They assist us in systematically reasoning about an analysis object in a top-down and recursive manner; i.e., the analysis process proceeds in a top-down manner by recursively partitioning the analysis target into sub-notions, called *recursive partitioning*. The analysis technique using logic trees has been widely applied to various analysis techniques in various fields, such as *FTA (Fault Tree Analysis)* [11] in reliability engineering, *Goal Modelling* [28] in requirements engineering, *Attack trees* [20] in security engineering, *Feature models* [16] in software product line engineering, and *Decision trees* [27] in operations research, etc. FOT aims to provide a design technique which assists in systematically designing test-models and hence, enhancing the quality of test-models, using the nice nature of the analysis technique using logic trees. To our best knowledge, this work is the first to apply analysis technique based logic trees to a design technique to assist in the systematic design of a test-model for combinatorial testing.

7. CONCLUSION

In this paper, we equipped a device to automatically construct test-cases for n-wise testing to FOTby developing a technique to construct test-cases for n-wise testing from a test model represented as an extended logic tree of $\mathcal{T}_{(mex,\ req)}$. We adopted a "transformation approach" to realize this technique. This approach constructs test suites, by first transforming test-models represented as $\mathcal{T}_{(mex,\ req)}$ in FOTto those in the formats which general n-wise testing tools (such as PICT [9], ACTS [30], CIT-BACH [31], etc) accept to input, and then feeding the transformed test-models to any of these tools. For this transformation approach, we develop an algorithm, called the "flattening algorithm". The algorithm plays the main role in our the approach. Comprehensive studies have been done on this technique in this work, by proposing the algorithm in order to evaluate this approach. Since correctness is the main concern of this technique used as a testing method, we proved the correctness of the algorithm. We also we implemented the algorithm to automate such test-suite constructions as a tool called FOT-nw. Further, to demonstrate the effectiveness of the technique, we conducted a case study, where we applied FOT-nw to design test models and automatically construct test suites of n-wise testing for an embedded system of stationary services, which is in a real-use in industry, thus showing experimental results derived from the case study.

There are several directions for further development on this testing method of FOT. First, we are currently extending FOT with several more cross-tree constraints, to enrich the expressiveness of the test-model represented by $\mathcal{T}_{(mex,\ req)}$. A second important direction is the introduction of the notion of *priority* to FOT to construct prioritized test suites. Software testing in practice often needs to

Specification			Number of test cases by					
Spec code	Tree Nodes	CTCs	SAT (FOT)	2-wise	3-wise	4-wise	5-wise	6-wise
Spec. 1	111	21	out of memory	90	559	3000	-	-
Spec. 2	146	10	out of memory	61	233	761	-	-
Spec. 3	158	7	out of memory	957	-	-	-	-
Spec. 4	92	18	out of memory	13	35	93	243	587
Spec. 5	75	5	out of memory	26	109	416	1400	4153
Spec. 6	56	0	147456	14	44	129	335	812
Spec. 7	52	7	21504	36	119	338	826	-
Spec. 8	70	20	10752	15	51	128	327	735
Spec. 9	45	2	4608	13	40	99	246	514
Spec. 10	41	0	3072	10	25	59	130	272
Spec. 11	57	7	2304	24	68	154	327	641
Spec. 12	38	3	768	12	32	67	152	277
Spec. 13	35	5	704	15	45	103	215	368
Spec. 14	30	0	288	18	48	105	172	288
Spec. 15	24	0	96	12	35	62	96	96
Spec. 16	26	3	84	12	31	54	84	84
Spec. 17	22	0	64	16	36	64	64	64
Spec. 18	57	0	36	25	36	36	36	36

Table 2: Experiments of test case generation

deal with limited resources, quick deadlines, requirements changes, etc., and hence, it is helpful to construct test-suites where test-cases are sorted in a prioritized order. Some of the recent research on n-wise testing investigates such a priority aspect [4]. We are developing a technique that integrates weighting factors to express priorities into tree-based test models (represented by logic trees) of FOT, and automatically constructs prioritized test suites from such prioritized test models. We are also considering to integrate FOT-nw with model-based testing methods for state-transition based specifications such as [1, 2].

Acknowledgement

Do Thi Bich Ngoc and Takashi Kitamura thank Prof. Tatsuhiro Tsuchiya at Osaka University for sharing insightful discussions which improved the research.

8. REFERENCES

[1] P. Ammann, J. Offutt, and W. Xu. Coverage criteria for state based specifications. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 118–156. Springer, 2008.

[2] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Proc. 9th Haifa Verification Conference (HVC 2013)*, volume 8244 of *LNCS*, pages 112–128, Haifa, Israel, 2013. Springer.

[3] 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, 2012.

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

[5] 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 Trans. Software Eng.*, 23(7):437–444, 1997.

[6] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proc. of ISSTA'07*, pages 129–139, 2007.

[8] 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 Trans. Software Eng.*, 34(5):633–650, 2008.

[9] J. Czerwonka. Pairwise testing in the real world. practical extensions to test case generators. *Microsoft Corporation, Software Testing Technical Articles*, 2008.

[10] Y. L. D. Richard Kuhn, Raghu N. Kacker. Practical Combinatorial Testing. Technical Report NIST/SP-800-142, National Institute of Standards and Technology, 2010.

[11] C. Ericson. Fault tree analysis - a history. In *The 17th International Systems Safety Conference*, 1999.

[12] M. Grindal and J. Offutt. Input parameter modeling for combination strategies. In *Proc. of IASTED'07 on Software Engineering*, pages 255–260. ACTA Press, 2007.

[13] M. Grindal, J. Offutt, and J. Mellin. Handling constraints in the input space when using combination strategies for software testing. Technical report, School of Humanities and Informatics, University of Skovde, 2006.

[14] M. Grochtmann. Test case design using classification trees. In *Proc. of The International Conference on Software Testing Analysis '94*, 1994.

[15] M. Grochtmann and J. Wegener. Test case design using classification trees and the classification-tree

editor cte. In *Proc. of QW'95*, 1995.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, 1990.

[17] T. Kitamura, T. B. N. Do, H. Ohsaki, L. Fang, and S. Yatabe. Test-case design by feature trees. In *Proc. of ISoLA'12 (1)*, volume 7609 of *LNCS*, pages 458–473. Springer, 2012.

[18] P. Kruse and M. Luniak. Automated test case generation using classification trees. *Software Quality Professional*, 13(1), 2010.

[19] 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.

[20] S. Mauw and M. Oostdijk. Foundations of attack trees. In *ICISC*, pages 186–198, 2005.

[21] J. D. McCaffrey and J. Czerwonka. An empirical study of the effectiveness of pairwise testing. In *Proc. of SERP'09*, pages 186–191, 2009.

[22] 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.

[23] OSEK/VDX operating system specification 2.2.3 http://www.osek-vdx.org/, 2005.

[24] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proc. of SPLC'10*, pages 196–210, 2010.

[25] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise feature-interaction testing for spls: potentials and limitations. In *Proc. of SPLC'11 (2)*, page 6, 2011.

[26] J. Petke, S. Yoo, M. B. Cohen, and M. Harman. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proc. of ESEC/FSE '13*, 2013.

[27] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[28] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proc. of RE*, page 249, 2001.

[29] D. B. West. *Introduction to Graph Theory*. Pearson, 2000.

[30] ACTS, available from: http://csrc.nist.gov/groups/SNS/acts/index.html.

[31] CIT-BACH, available from: http://www-ise4.ist.osaka-u.ac.jp/ t-tutiya/CIT/.

APPENDIX

Algorithm 2: Flatten-And-Sequence(T)

```
Data: A general test tree $\mathbf{T} = (F, r, E, @, MEX, REQ)$
Result: A test tree without and-sequences
begin
    if (T has no and-sequences) then
        return $T$
    else
        for $a \in F \wedge @(a) =$ "and" do
            while $\exists b \in F \cdot (a, b) \in E \wedge @(b) =$ "and" do
                for $x \in F$ do
                    //change $b$ to $a$ in all constraints
                    if $(b, x) \in REQ \wedge (b \neq x)$ then
                        $REQ \leftarrow REQ \setminus \{(b, x)\}$;
                        $REQ \leftarrow REQ \cup \{(a, x)\}$;
                    if $(x, b) \in REQ \wedge (b \neq x)$ then
                        $REQ \leftarrow REQ \setminus \{(x, b)\}$;
                        $REQ \leftarrow REQ \cup \{(x, a)\}$;
                    if $(b, x) \in MEX \wedge (b \neq x)$ then
                        $MEX \leftarrow MEX \setminus \{(x, b)\}$;
                        $MEX \leftarrow MEX \cup \{(x, a)\}$;
                if $(b, b) \in REQ$ then
                    $REQ \leftarrow REQ \setminus \{(b, b)\}$;
                    $REQ \leftarrow REQ \cup \{(a, a)\}$;
                if $(b, b) \in MEX$ then
                    $MEX \leftarrow MEX \setminus \{(b, b)\}$;
                    $MEX \leftarrow MEX \cup \{(a, a)\}$;
                // remove b and set parent of b's
                children is a
                $E \leftarrow E \setminus \{(a, b)\}$;
                for (c is a child of b) do
                    $E \leftarrow E \setminus \{(b, c)\}$;
                    $E \leftarrow E \cup \{(a, c)\}$;
                $F \leftarrow F \setminus \{b\}$;
    return $T$;
end
```

Algorithm 3: Flatten-Xor-Sequence(T)

```
Data: A general test tree $\mathbf{T} = (F, r, E, @, MEX, REQ)$
Result: A test tree without xor-sequences
begin
    if (T has no xor-sequences) then
        return $T$
    else
        for $a \in F \wedge @(a) = $ "xor" do
            while $\exists b \in F \cdot (a,b) \in E \wedge @(b) = $ "xor" do
                for $x \in F$ do
                    // change constraints contain b to constraints of other nodes
                    if $(b,x) \in REQ \wedge (b \neq x)$ then
                        $REQ \leftarrow REQ \setminus \{(b,x)\}$;
                        for (c is a child of b) do
                            $REQ \leftarrow REQ \cup \{(c,x)\}$;
                    if $(x,b) \in REQ \wedge (b \neq x)$ then
                        $REQ \leftarrow REQ \setminus \{(x,b)\}$;
                        $REQ \leftarrow REQ \cup \{(x,a)\}$;
                        for ($b'$ is child of a) $\wedge (b' \neq b)$ do
                            $MEX \leftarrow \cup \{(b',x)\}$;
                    if $(b,x) \in MEX \wedge (b \neq x)$ then
                        $MEX \leftarrow MEX \setminus \{(b,x)\}$;
                        for (c is a child of b) do
                            $MEX \leftarrow MEX \cup \{(c,x)\}$;
                if $(b,b) \in REQ$ then
                    $REQ \leftarrow REQ \setminus \{(b,b)\}$;
                    for (c is a child of b) do
                        $REQ \leftarrow REQ \cup \{(c,c)\}$;
                if $(b,b) \in MEX$ then
                    $MEX \leftarrow MEX \setminus \{(b,b)\}$;
                    for (c is a child of b) do
                        $MEX \leftarrow MEX \cup \{(c,c)\}$;
                // remove b and set parent of b's children is a
                $E \leftarrow E \setminus \{(a,b)\}$;
                for (c is a child of b) do
                    $E \leftarrow E \setminus \{(b,c)\}$;
                    $E \leftarrow E \cup \{(a,c)\}$;
                $F \leftarrow F \setminus \{b\}$;
    return $T$;
end
```

Algorithm 4: Lift-Tree(T)

```
Data: A test tree without and-sequence and xor-sequence $\mathbf{T} = (F, r, E, @, MEX, REQ)$
Result: A flattened tree
begin
    if $@(r) = $ "xor" then
        //add a new node and change @(r) to "and"
        $F \leftarrow F \cup \{r_1\}$;
        $@(r_1) \leftarrow$ "xor";
        $@(r) \leftarrow$ "and";
        for $x \in F$ do
            if $(r,x) \in E$ then
                $E \leftarrow E \setminus \{(r,x)\}$;
                $E \leftarrow E \cup \{r_1, x\}$;
        $E \leftarrow E \cup \{(r, r_1)\}$;
    while (the height of T is greater than or equal to 3, i.e., $\|T\| \geq 3$) do
        for $a, b, c \in F$ do
            if $(r,a) \in E \wedge (a,b) \in E \wedge (b,c) \in E$ then
                while $\exists x \in F \cdot (x,c) \in REQ$ do
                    //change c to b in all constraints
                    $REQ \leftarrow REQ \setminus \{(x,c)\}$;
                    $REQ \leftarrow REQ \cup \{(x,b)\}$;
                while $\exists x \in F \cdot (c,x) \in REQ$ do
                    $REQ \leftarrow REQ \setminus \{(c,x)\}$;
                    $REQ \leftarrow REQ \cup \{(b,x)\}$;
                while $\exists x \in F \cdot (c,x) \in MEX$ do
                    $MEX \leftarrow MEX \setminus \{(c,x)\}$;
                    $MEX \leftarrow MEX \cup \{(b,x)\}$;
                if (c has children $d_1, .., d_n$) then
                    //create a dummy node and add new edge
                    $newNode \leftarrow$ Node($\neg(d_1..d_n)$);
                    $F \leftarrow F \cup \{newNode\}$;
                    $E \leftarrow E \cup \{(c, newNode)\}$;
                    //remove edge $(b,c)$ and lift node $c$
                    $E \leftarrow E \setminus \{(b,c)\}$;
                    $E \leftarrow E \cup \{(r,c)\}$;
                    //update constraints
                    $REQ \leftarrow REQ \cup \{(d_i, b) \mid i = 1 \cdot \cdot n\}$;
                    $MEX \leftarrow MEX \cup \{(newNode, b)\}$;
                if c is a leaf then
                    //Create two dummy nodes and add new edges
                    $newNode1 \leftarrow$ Node($\neg(c)$);
                    $newNode2 \leftarrow$ Node($c_0$);
                    $F \leftarrow F \cup \{newNode1, newNode2\}$;
                    $@(newNode2) \leftarrow$ "xor";
                    $E \leftarrow E \cup \{(newNode2, c), (newNode2, newNode1)\}$;
                    //remove edge $(b,c)$ and lift new subtree $newNode2$
                    $E \leftarrow E \setminus \{(b,c)\}$;
                    $E \leftarrow E \cup \{(r, newNode2)\}$;
                    //update constraints
                    $REQ \leftarrow REQ \cup \{(c,b)\}$;
                    $MEX \leftarrow MEX \cup \{(newNode1, b)\}$;
    return $T$;
end
```