Test-Case Design by Feature Trees

Takashi Kitamura, Ngoc Thi Bich Do, Hitoshi Ohsaki,
Ling Fang, Shunsuke Yatabe

National Institute of Advanced Industrial Science and Technology (AIST)
{t.kitamura,do.ngoc,hitoshi.ohsaki,fang-ling,shunsuke.yatabe}@aist.go.jp

Abstract. This paper proposes a test-case design method for black-box testing, called "*Feature Oriented Testing* (FOT)". The method is realized by applying *Feature Models* (FMs) developed in software product line engineering to test-case designs. We develop a graphical language for test-case design called "*Feature Trees for Testing* (FTT)" based on FMs. To firmly underpin the method, we provide a formal semantics of FTT, by means of test-cases derived from test-case designs modelled with FTT. Based on the semantics we develop an automated test-suite generation and correctness checking of test-case designs using SAT, as computer-aided analysis techniques of the method. Feasibility of the method is demonstrated from several viewpoints including its implementation, complexity analysis, experiments, a case study, and an assistant tool.

Key words: black-box testing, combination testing, SAT-based analysis

1 Introduction

In black-box testing (BBT) test cases are designed by analysing the input domain of the system under test (SUT) often according to the system's specification. The *Classification Tree Method* (CTM) [5, 9–11] is one of the-state-of-the-art test-case design methods for BBT. It is a model-based and combination testing method; i.e., test cases are designed as a visual model with a given diagram-based language, and test cases are generated automatically from such a model using combination techniques. Due to its nice characteristics as a testing method, CTM is often used in industry including automotive industries [19]. However, for a better testing method improvements can be considered from several perspectives such as its theory, higher computer-aided analysis, efficiency of automated technologies, and modelling paradigms.

Feature-Oriented Domain Analysis/Feature Models (FODA/FMs) is an analysis method for software product lines (SPLs), first proposed by Kang et al. [14]. This method takes a model-based approach; i.e., an SPL is modelled with extended and-or logical trees called "Feature Models (FMs)", which enable systematic analysis in a top-down manner, together with their graphical representations of "Feature Diagrams". In addition, useful information about the SPL can be derived by applying analysis techniques to the models. A main characteristic of FMs is

its compact and visual representations by diagrams to capture SPLs as well as a variety of analysis techniques. So far fruitful research results of FMs have been made in research and industry, including various model designs [20], semantics [3, 17, 20, 21], various analysis operations such as consistency checking, diagnosis, validations, refactoring, and so on (as summarized in [2, 17]). Also such analysis operations are carried out on various logic paradigms such as propositional logic [1, 13, 16, 22], description logic [8] or constraint programming [3] as well as algorithmic approaches [4, 23].

In this work, we propose a test-case design method by applying FMs, called "*Feature Oriented Testing* (FOT)", identified as a model-based and combination testing method for BBT. The aims of the work are three-fold: (1) to develop a test-case design language based on the model designs of FMs, which are characterized with the compact and visual representations by diagrams of extended and-or logical trees, (2) to apply rich theories of FMs to the test-case design method focusing on semantics, and (3) to apply computer-aided analysis techniques of FMs to the test-case design method, to retrieve useful information for test-case designs.

The main contributions of this paper are two-fold: (A) to realize these aims as a test-case design method, and (B) to demonstrate the method's feasibility from several viewpoints. For (A), first we analyze the requirements for developing a test-case design language, and design such a language as "*Feature Tree for Testing* (FTT)" that suits the test-case design purpose based on various designs of FMs proposed in the literature [20]. Then we build a theoretical foundation of FTT by providing its formal syntax and semantics; which makes a basis of reliability and computer-aided analysis. Further, we develop two kinds of logic-based automated analysis techniques for FTT using a SAT solver: a test-suite generation and correctness checking of test-case designs by FTT.

For (B), first feasibility is shown from the viewpoint of reliability, which is an important property as a testing method, by building formal semantics and proving the correctness of the test-case generation algorithm w.r.t. the semantics. Feasibility is demonstrated from the viewpoint of computational cost on the automated analysis, by analyzing the computational complexity and by providing experimental results. A case study is presented, where we apply FOT to test-case design for OSEK/VDX-OS (OSEK-OS), a standard real-time OS for automotives [18]. We also explain our GUI-based assistant tool for FOT. This show not only how the method can be assisted by a tool, but also some essential techniques for test-case designs of FOT using this tool.

2 A motivating example

Borrowing an example in [5], we design test cases for BBT for a computer vision system. As seen in Fig. 1, this system determines the size of various blocks passing the camera of the system on a belt-conveyor. Fig. 2 shows a test-case design for BBT for the system by FOT. In FOT, an FM is used to design test cases; i.e, analysis for test-case design proceeds using FTT by splitting up the

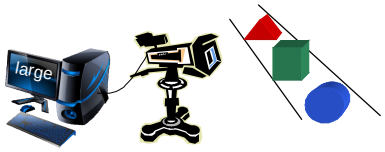

Fig. 1. Computer vision system for determining the size of building blocks

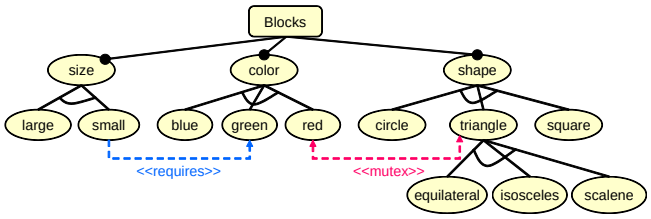

Fig. 2. A simple and small test-case design using an FTT

input/(output) domain of the SUT with various *test-relevant aspects*, which we also refer to as *features*.

The analysis of the input domain, i.e., the test-case design for the SUT, proceeds in a top-down manner with its root as the input domain of SUT, which is "Blocks" here. First, "Blocks" are decomposed into features of their "size", "color" and "shape". In decompositions (decomp.), we distinguish them with two kinds of *orthogonal* and *alternative* decomp. The "block" is decomposed with an orthogonal decomp., as we regard its three sub-features are orthogonal notions. We may also call such decomp. *and-decomp.*, following the convention of and-or trees and FMs. We clarify such decomp. explicitly in the diagram by the dot on top of each feature.

Next these three decomposed sub-features are further decomposed into smaller sub-features. For example, the "size" feature is decomposed into two sub-features: "small" and "large". In this case, it is done with an *alternative* decomp., which may be also called *xor-decomp.*, by regarding the sub-features as alternatives to one another. Compositions of this kind are clarified in the diagram in the way that the edges of a decomp., are tied up with a string. Similarly, the "color" features are decomposed into "blue", "green" and "red", and "shape" into "circle", "triangle" and "square" with *alternative*-decomp. The "triangle" feature is further decomposed into "equilateral", "isosceles" and "scalene" alternatively, to design test cases in a detailed way and hence inspect the system in more details.

Besides such decomp. relations between features, which form the parent-child relations of trees, "*mutex*" and "*requires*" relations drawn globally in the tree (i.e., crossing the tree) are found in the model in Fig 2. These represent constraints between features in the tree globally, which we call *cross-tree constraints* (CTCs), to exclude nonsense or undesired test cases according to given specifications. For example, assume in the above example, the following specifications are given: (1) "There are no blocks whose color is red and whose shape is a triangle."

1. small, red, circle	7. small, green, square	13. small, green, isosceles
2. small, blue, circle	8. large, green, square	14. small, blue, isosceles
3. large, green, circle	9. large, green, equilateral	15. small, blue, scalene
4. small, green, circle	10. small, blue, equilateral	16. large, green, scalene
5. small, blue, square	11. small, green, equilateral	17. small, green, scalene
6. small, red, square	12. large, green, isosceles	

Table 1. The test suite obtained from test-case design of Fig. 2

(2) "If the size of blocks is small, then the color of the blocks is green.". Due to these specifications, it is nonsense and undesirable to prepare test cases for such cases. The CTCs clarify such nonsense test cases. The *mutex* (an abbr. for "mutually exclusive with") constraint between the "red" and "triangle" features in Fig. 2 is drawn to cover specification (1), and *requires* between the "small" to "circle" feature to cover specification (2). Note that each CTC affects all the features in the sub-tree of the features it involves. E.g., the *mutex* constraint stipulates that "red" is *mutex* not only with "triangle" but also with all the sub-features of the sub-tree: "equilateral", "isosceles" and "scalene".

The test-case design, shown as a diagram in Fig. 2, captures a set of test cases; i.e., we can obtain a set of test cases (i.e., test suite) from the diagram. Here, a test case is defined as a set of features in the tree. Table 1 shows the test suite obtained from the test-case design of Fig. 2. That is, the test suite derived from the test-case design consists of fifteen test cases; for example, test-case 1 indicates blocks whose size is "small", color is "red", is shape "circle". Roughly, test cases are derived from such test-case designs by recursively applying the following standard interpretation of and-or logical tree; i.e., all the sub-features of *and*-decomp. or its descendants have to be in any test case, and exactly one of the sub-features of *xor*-decomp. or its descendants have to be in any test case. Besides, the test cases the CTCs are applied to are excluded. The rules to derive test cases from the diagrams should be more detailed in an exact way, and we formally explain these rules in Section 4.

3 Feature Trees for Testing

This section develops a test-case design language based on FMs, called *Feature Tree for Testing* (FTT), which we regard as the *modelling language* for test-case design in FOT. First we analyze requirements for such a language as a model-based and combination testing method for BBT. According to them we design such a language as FTT based on FMs, showing its design choices. Then based on the design of FTT, the syntax and semantics of FTT are provided formally.

3.1 Requirements and design choices

Requirements Though the basic idea of FTT was seen in the previous section, here we briefly summarize the requirements analysis for developing a test-case design language for our test-case design purpose of model-based and combination testing method for BBT, as follows:

1. The basic structure of FTT is designed as a *tree*; i.e., the tree structure is formed by an input-domain analysis of SUT by repeatedly decomposing it with features from the root, which facilitates systematic test-case design in a top-down manner.
2. Each decomp. of a feature (i.e., the input domain of SUT) should be distinguished by two kinds: *orthogonal decomp.*, i.e., all the sub-features are orthogonal notions to one another, and *alternative decomp.*, i.e., all the sub-features are alternative notions to one another.
3. Some kinds of constraint operators, imposed on globally (between any features crossing a tree), are equipped to exclude non-sense and undesired test cases according to given specifications.

Design choices We design a language for test-case design as FTT to meet the requirements based on various variants of FMs proposed in the literature [20]. By following [20] for a scheme of design choices of FMs, FTT is characterized as:

1. FTT are trees (, but not DAGs: Directed Acyclic Graphs).
2. FTT have the following two-kinds of decomp. operators:
   (a) *and*-decomp., to express *orthogonal*-decomp.
   (b) *xor*-decomp., to express *alternative*-decomp.
3. FTT have the following constraint representations drawn globally in a tree:
   (a) *requires*; if a feature f requires a feature g, the inclusion of f in a test case implies the inclusion of g in such a test case.
   (b) *mutex*; the two features related by the relation cannot be present simultaneously in a test-case.

Some other relations, often common in FMs such as "*optional*", "*or*-decomp." and "cardinality", are not included in FTT, since straightforward interpretations can not be given on the operators in our test-case design setting. The same is true for other relations such as "*generalization*", "*specialization*" and "*implemented-by*" found in [15]. The language design of FTT is not same as any of the FMs listed in [20], but similar to the original FM developed by Kang et al. [14]

3.2 Syntax of Feature Trees for Testing

We give a formal syntax of FTT as a basis for the formal developments:

Definition 1. *A feature tree is a tuple* $(F, r, L, \rightharpoonup, @, \overset{\text{req}}{\rightarrow}, \overset{\text{mex}}{\longleftrightarrow})$ *such that*

- $(F, r, \rightharpoonup)$ *is a tree, where* F *is a set of features (as the nodes of a tree),* r *is the root, and* $\rightharpoonup$ *is the parent-child relation on* F*,*
  - *we say "feature* f *is the parent of* g*" and "*g *is a child of* f*" if* $f \rightharpoonup g$*,*
- $L (\subset F)$ *is a set of leaf features,*
- $@$ *is a function from* $F \setminus L$ *to* $\{\text{and}, \text{xor}\}$*,*
- $\overset{\text{mex}}{\longleftrightarrow}$ *is a symmetric and irreflexive binary relation over* F*,*
- $\overset{\text{req}}{\longrightarrow}$ *is an asymmetric and irreflexive binary relation over* F*.* □

FTT are trees $(F, r, \rightharpoonup)$ extended with several notions. First FTT are a variant of and-or logical trees. We realize this with "node-based design", where each feature (i.e., node) of the tree except for leaf features is labeled with "and" or "xor". The function $@ : F \setminus L \to \{\text{and}, \text{xor}\}$, which labels each (non-leaf) features with *and* or *xor*, is equipped for this. We call features "and-feature" or "xor-feature" if it is associated with "and" and "xor" by @ respectively. Note that, due to the design, "and " and "xor"-edges shall not be mixed among the edges out-going from a feature. The two kinds of CTCs of *mutex* and *requires*, which are another extension of FTT, are expressed by the binary relations " $\overset{\text{mex}}{\leftrightarrow}$ " and " $\overset{\text{req}}{\rightarrow}$" on F.

3.3 Semantics

An FTT captures a set of test cases. In other words, the semantics of an FTT is defined by way of a set of test cases derived from it; i.e., given an FTT, we formally understand what it means by way of a set of test cases.

Definition 2 (Pre-model). *A pre-model $M'(\in \mathcal{M}')$ of an FTT t is a subset of its features: $M' \in \mathcal{P}F$, where $\mathcal{P}X$ denotes the power set of X.* □

Definition 3 (Model). *A model $M(\in \mathcal{M})$ of an FTT t is a pre-model that satisfies the following conditions, and is noted as $M \models' t$:*

1. *The root feature is in the model: $r \in M$,*
2. *If a feature is in a model, its parent is in the model too: $f \in M \Rightarrow \text{parent}(f) \in M$,*
3. *If an and-feature is in a model, all its children are in the model too: $f \in M \wedge @(f) = and \Rightarrow (\forall g. f \rightharpoonup g \rightarrow g \in M)$,*
4. *If an xor-feature is in a model, exactly one of its children is in the model too; $f \in M \wedge @(f) = \text{xor} \Rightarrow (\exists! g. f \rightharpoonup g \wedge g \in M)$,*
5. *The model must satisfy all formulas from the CTCs set $\Phi(= \overset{mex}{\leftrightarrow} \cup \overset{req}{\rightarrow})$: $\forall \phi \in \Phi. M \models' \phi$, where "$M \models' f \overset{\text{mex}}{\longleftrightarrow} g$" if f and g are not both in M, and "$M \models' f \overset{\text{req}}{\longrightarrow} g$" if f is in M, g is in M too.* □

The definition of test case and test suite are given by way of the *model*.

Definition 4 (Test case and test suite). *1. A test case c is a subset of leaves: $c \in \mathcal{P}L$. 2. A test case of the model M, noted M°, is $M \cap L$. 3. A test suite s is a set of test cases: $s = \mathcal{P}c \in \mathcal{P}\mathcal{P}L$. 4. The test suite derived from an FTT t is the set of test cases of models M satisfying t: $[\![t]\!] = \{M^\circ \mid M \models' t\}$* □

4 SAT-based automated analysis of FTT

This section explains several SAT-based automated analysis techniques of FTT, as computer-aided analysis techniques of FOT. An epoch in the research of FMs is the provision of encoding FMs to a propositional (prop.) formula, which brings many interesting logic-based analysis on FMs, often using technologies of SAT-solvers. Applying these techniques to our setting, we develop *SAT-based automated test-suite generation* and *correctness checking of test-case designs by FTT*.

	Feature model relation	Corresponding formula
(a)	r is the root feature	r
(b)	$p \rightharpoonup c_1$	$c_1 \rightarrow p$
(c)	$@(p) = and$ and $p \rightharpoonup c$	$p \rightharpoonup c$
(d)	$@(p) = xor$ and $p \rightharpoonup c_1, \cdots, p \rightharpoonup c_n$	$p \rightarrow \bigvee \begin{pmatrix} (c_1 \wedge \neg c_2 \wedge \cdots \wedge \neg c_n) \\ \cdots \\ (\neg c_1 \wedge \neg c_2 \wedge \cdots \wedge c_n) \end{pmatrix}$
(e)	$p \overset{mex}{\longleftrightarrow} q$	$\neg(p \wedge q)$
(f)	$p \overset{req}{\longrightarrow} q$	$p \rightarrow q$

Table 2. The encoding rules *trans* of an FTT into prop. formulas

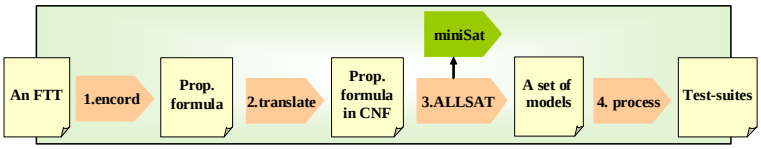

Fig. 3. The system for test-case generation

4.1 Propositional formulas encoding

Table 2 shows the encoding rules (*trans*) of an FTT to a prop. formula. Lemma 1 shows the rules are correct w.r.t. the semantics in Definition 4. The proof of this lemma is obvious, but it plays a critical role for guaranteeing the correctness of our automated analysis techniques.

Lemma 1. *For any* FTT t, $M \models' t$ *iff* $M \models trans(t)$ □

4.2 A SAT-based automated test-suite generation

An algorithm design and early implementation First we derive the following theorem from Lemma 1:

Theorem 1. *For any* FTT t, $[\![t]\!] = \{M^\circ \mid M \models trans(t)\}$. □

This theorem indicates that in order to obtain the test suite of a given FTT t according to Definition 4, it suffices to follow the procedures of: (1) to derive all the models that satisfy the prop. formula encoded from the FTT t i.t.o the classical logic, and (2) to process each of the models by taking one that intersects with the leaf nodes of the FTT and (3) to take the union of the processed models. And the algorithm design follows this scheme.

The test-suite generation algorithm is displayed in Fig. 3. The input is an FTT and the output is a set of corresponding test cases (i.e., a test suite). The algorithm mainly consists of the following four components.

1. The first component encodes an FTT to a prop. formula according to the encoding rules in Table 2.
2. The second is a *conjunctive normal form* (CNF) translator, which translates a prop. formula into it in a CNF.

3. The third is an all-solutions SAT-solver (ALLSAT), inputting the encoded formula of FTT in CNF, finds all models for it. We have implemented an ALLSAT using the blocking algorithm (which finds all models by iteratively calling a SAT solver while at each call blocking clauses which block finding a model already found is added) by extending MiniSAT[6].
4. The fourth processes a set of models obtained from the ALLSAT, by taking one that intersects with the leaf nodes of the FTT, and produces the test suite by collecting the processed models (i.e., test cases).

Computational complexity To analyze the complexity of the test-case generation algorithm, we analyze the complexity of each component 1-4. Given an FTT t, we denote the number of features as n.

1. The length of a formula derived by *trans* is the sum of sub-formulas by applying each rule of (a)–(f). Thus it suffices to analyze rule (d), which makes the longest sub-formula among of (a)–(f) in Table 2. The length of a sub-formula by (d) for an xor-decomp. with k-children is bound by $O(k \times k)$. Both the number of children of any feature and that of xor-decomp. in t are bound by $n-1$. Hence the length of a formula by *trans* is bound by $O(n^2)$.
2. We have implemented an algorithm to transform a prop. formula using the standard laws of logical equivalences, and have produced a clause set that is exponential w.r.t. the size of the original formula in the worst case.
3. The SAT-problem is NP-complete, and the worst time complexity of the algorithm we use (i.e., MiniSAT[6, 12]) is $O(2^n)$ where n indicates the number of the prop. variables. Also the number of models for a given formula is bound by 2^n. Hence, the complexity of ALLSAT is bound by $2^n \times O(2^n) \in O(4^n)$.
4. The complexity of the set intersection of two sets with size k is $O(k^2)$. The number of nodes and the leaf nodes of an FTT are bound by n. There are at most 2^n models. Hence, the complexity is bound by $2^n \times O(n^2) \in O(2^n \times n^2)$.

Hence the bottleneck of the algorithm is the component of CNF-transformation and ALLSAT, whose complexity are exponential to the input FTT t.

Experimental results Besides the complexity analysis, we provide experimental results to show feasibility of the implementation from the viewpoint of computational cost of FOT. According to the above analysis of computational cost of the test-suite generation algorithm, we know that its bottleneck lies on computing all the models using ALLSAT, which takes exponential time w.r.t. the size of FTT. But in practice the computational cost is cheaper than the theoretical analysis. One reason is that the off-the-shelf SAT-solver we use, i.e., MiniSAT, runs faster than the above analysis. Second, the number of the models for the formula encoded from an FTT is much less than 2^n in real settings. Also, the number of test cases varies depending on the structures of FTT. The ratio of *and/xor*-decomp. and the ratio of CTCs in an FTT mainly affect the number of test cases; i.e., the more *and*-decomp. and CTCs there are in an FTT, the less test cases are derived. Table 3 shows an experimental result, presenting the time and

ctcr(%)		Size of an FTT (n) 20	30	40	50	60
0	time (s)	0.09	0.67	2.04	9.65	20.43
	# test cases	120	960	6912	19008	43200
10	time (s)	0.04	0.35	0.82	3.65	4.71
	# test cases	92	432	2464	8580	9160
20	time (s)	0.03	0.09	0.53	1.52	2.01
	# test cases	75	238	916	2710	4244
30	time (s)	0.01	0.07	0.17	0.51	0.93
	# test cases	26	120	288	880	1666
40	time (s)	0.01	0.06	0.06	0.12	0.14
	# test cases	13	45	96	122	168

Table 3. Experimental results

the number of test cases, where *ctcr* stands for the CTCs ratio (i.e., the ratio of CTCs w.r.t the size of FTT). The experiments were conducted on a machine with an Intel Core2 Duo CPU P8700 @2.53 GHz, 2.96 GB of RAM and Windows 7.

4.3 SAT-based correctness checking of test-case designs

An important class of various computer-aided analysis techniques on FMs is *correctness checking*. Generally, *correctness checking* includes *consistency checking* and detecting *dead/common* features. These notions are interpreted in the setting of test-case design as follows: an FTT, i.e., a test-case design, is *inconsistent* if no test case can be derived from it; a feature is *dead* in an FTT if it does not appear in any of the test cases of the model derived from it; and a feature is *common* in an FTT if it appears in all the test cases derived from it.

Interestingly, these analysis operations on *correctness checking* can be reduced to a simple satisfiability checking problem of a prop. formula. The *consistency of a test-case design* by a FTT can be examined by checking the satisfiability of the formula ϕ encoded from the FTT (t), i.e., $\phi = trans(t)$. Existence of a *dead* feature f in an FTT can be examined by checking the satisfiability of the formula $\phi \wedge f$; i.e., f is a *dead* feature if $\phi \wedge f$ is unsatisfiable. Similarly, existence of a *common* feature f in a model can be examined by checking the satisfiability of $\phi \wedge \neg f$; i.e., f is a *common* feature if the formula is unsatisfiable.

We have introduced these analysis operations on correctness checking of test-case designs in FOT, which help validation of test-case designs by FTT. As shown in the next section, the consistency checking, especially detecting *dead* features, are quite useful for finding defects in test-case designs since they often enter test-case designs and their existences are undesirable.

5 A case study : A test-case design for OSEK-OS

To demonstrate feasibility of FOT in real practice, we show a case study where we applied FOT to test case design for the OSEK-OS[18, 19], a real-time OS for

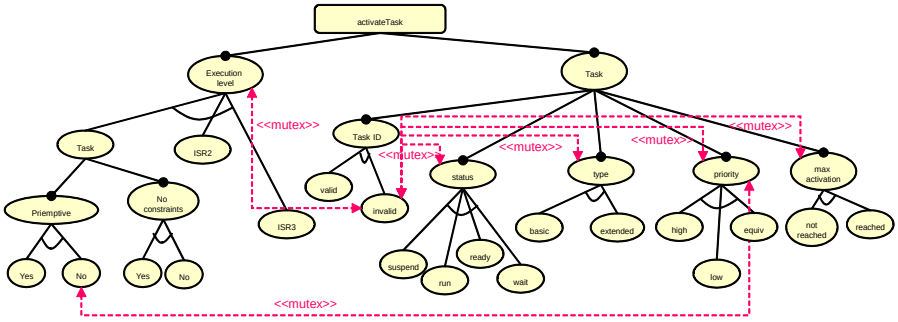

Fig. 4. A test-case design by FTT for API "activateTask" in OSEK-OS

automotives. Specifically, using FTT we make a test-case design for an API function "*activateTask*", which transfers a task specified with parameter "*TaskID*" from the "*suspended*" state into the "*ready*" state. We analyzed the specification [18], and made its test-case design as in Fig. 4. The figure shows the test-case design with an FTT that consists of 30 features and 6 CTCs, and 192 test cases are obtained from it.

Several observations obtained from the case-study are as follows:

1. Test-case designs with a variant of and-or tree are easily accepted by developers in practice because and-or trees are a common analysis technique and close to human thinking. Also this analysis technique using and-or trees can allow them to focus on designing test cases released from direct edits on logical formulas, which are often error-prone.
2. Efficiency of the automated analysis techniques of FOT, i.e, automated test-suite generation and correctness checking, whose experimental results are shown in Tab. 3, is practical enough in our case studies.
3. Unfortunately, FTT is not expressive enough to express any desired test suite in any settings, because test-case designs in real development are sometimes extremely detailed and beyond the expressiveness of FTT. As a result, manual arrangements of test cases such as to add, delete and modify test cases are required to cover some detailed cases. But this should not be taken as a critical defect of FOT, since CTM, which is the state-of-the-art method of test-case design for BBT often used in real developments, also inherently has this aspect of expressiveness. (See related discussions in Section 8.)
4. Detecting *dead* features for correctness checking of test-case designs by FTT is quite useful in practice. Test-case designs are often complex, and hence prone to contain deficiencies. In the test-case design in Fig. 4, the "*invalid*" and "*No*" (under "*Preemptive*") features are the *dead* features. Existence of *dead* features indicates that some errors may be contained in the test-case design, or these *dead* features may have to be taken care of by manual arrangements.
5. Test-case designs by FTT can be used for test documentations such as a system specification for testing. These designs can also be used as communication media among developers, and as evidence for certification. The high

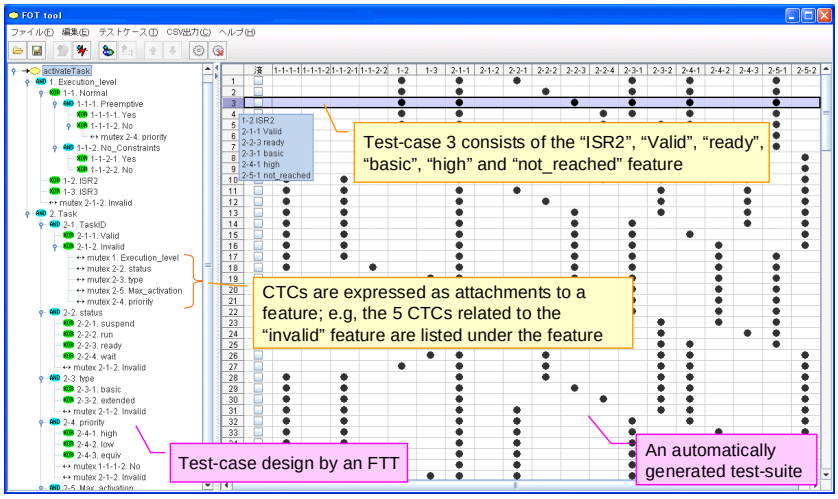

Fig. 5. The GUI tool for FOT

readability of FTT, achieved by the compact and visual representation by diagrams, and by the formal semantics to unify interpretation of FTT contributes to the aspect of documentations.

6. The readability of the diagram representations of FTT can be preserved, even with many CTCs drawn all over the tree, together with the GUI-based assistant tool. We explain the tool's support for readability in the next section.

6 Tool development

We have developed a GUI tool to assist FOT. This section will briefly explain this tool. The tool development shows not only our current status of the development of FOT, but is also essential in the test-case design method of FOT.

Describing FTT via the GUI. Fig. 5 shows the main GUI of the tool. It is separated into two panels: the left-hand-side panel where users describe and input an FTT, and the right-hand-side panel which displays the automatically generated test cases in a matrix form.

In designing test cases by describing an FTT via the GUI, several advantages ascribed to the properties of FTT become possible. First, the GUI prevents inputting illegitimate FTT w.r.t. the defined syntax in Definition 1. That is, the GUI lets users input only a legitimate FTT, which then allows them to concentrate on the logic of test-case designs. The second advantage centers on scalability w.r.t. the readability of the diagram representations of FTT. As shown in Fig. 5 in the FTT description of in the GUI, each CTC is expressed as an attachment to features involved in the description. Due to the GUI design, even with a number of CTCs, the diagram representation keeps readability.

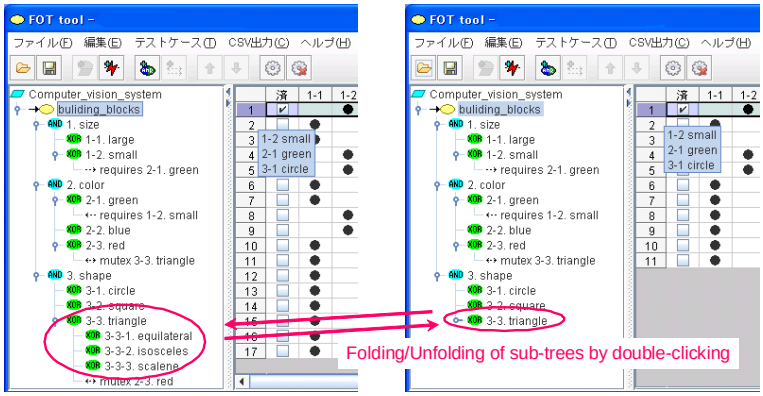

Fig. 6. The number of test-cases can be controlled by folding/unfolding sub-trees

Controlling the number of test cases flexibly In general, the system quality guaranteed by testing and its cost is a trade-off. That is, the more the cost for detailed testing is allowed, the higher the quality of the system is guaranteed. On the other hand, the resources for testing are limited in real developments. Therefore, it is desirable for a testing method to be able to flexibly control system quality guaranteed by testing by depending on its affordable resources.

FOT is equipped with such a mechanism; i.e., it is equipped with a device to flexibly control the number of test cases. The device is realized by using a notion of abstraction on tree structures in the FTT such as folding and unfolding sub-trees. Fig. 6 demonstrates this device in the tool, using the example of the computer vision system. The left side in Fig. 6 shows the test-case design for the computer vision system in Section 2 using the tool, where 17 test cases are obtained. The number can be flexibly reduced, for instance, by abstracting the "triangle" feature by folding its sub-tree; i.e., the number of the test cases obtained from the tree whose "triangle" sub-tree is folded, can be reduced to 11.

7 Discussions and Related Work

CTM (*Classification Tree Method*) [5, 9–11] is a model-based and combination-based test-case design method for BBT; i.e., a test case is designed as a model of a tree diagram, and test cases are obtained automatically from it using a combination technique. In CTM, the model to represent a test-case design consists of the three separate description components: (1) a "*classification-tree diagram*", which is a tree-based diagram to represent the basic structure of test-case design, (2) "*combination rules*" to define combination rules based on the classification-tree diagram, and (3) "*dependency rules*", written in prop. logic, to exclude nonsense test cases. Fig. 7 shows a descriptive example of a test-case design for the computer vision system using CTM to produce the same test suite in Tab. 1. FOT can be seen as a comparable test-case design method to CTM, but its advantages over CTM are the following:

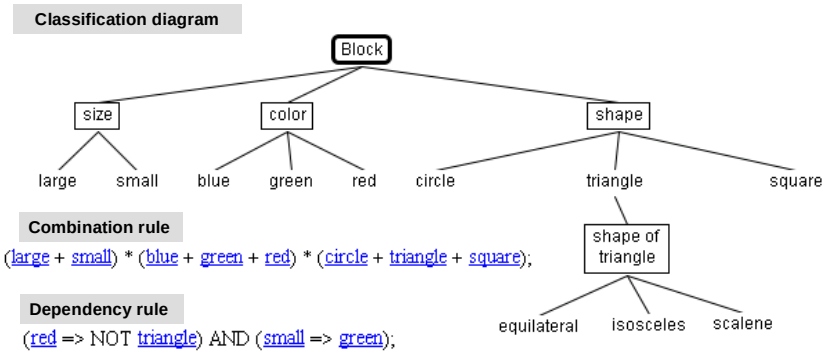

Fig. 7. Test-case design for the computer vision system in CTM

The first advantage is on the modelling paradigm for test-case designs. In FOT, a test-case design is represented as an FTT in a single diagram based on *and-or logical trees*. We inherit the single and compact design of FMs, often recognized as a characteristic of FMs, in FTT, which brings higher readability. The model design of FTT requires less complex descriptions than CTM, bringing higher productivity; i.e., complex descriptions together with direct edits of the logical formula is often a main barrier preventing wider adoption of such methods in real developments. Also, the single representations of FTT achieve higher maintainability; the separate descriptions in CTM often require efforts because such changes in a description component may affect the others. Also, the model design based on *and-or trees* achieves higher availability as a common and traditional analysis technique. In fact, the model design is highly inspired by *Fault Tree Analysis* (FTA) [7], which is an established analysis technique based on and-or trees in *reliability engineering*. Also, the logic-based model of FTT facilitates the logic-based analysis in FOT.

On the other hand, theoretically CTM is more expressive than FTT. The difference lies on the expressiveness of a description device to exclude nonsense test cases: i..e, CTCs in FTT, which consists of the two operators *mutex* and *requires*, and the *dependency rules* in CTM, which deals with a full prop. formula. But from our case studies, the advantage of the expressiveness of CTM is mostly in theory. From the case studies, we learned only simple rules are needed to realize such devices, and CTCs are expressive enough for this purpose. In place of expressiveness, FTT realizes the above-mentioned nice properties such as readability, productivity, and maintainability, etc. We have mentioned in Section 6 that FTT may not be expressive enough for some settings. But this is due to the tree structure of FTT rather than due to the expressiveness of CTCs, and hence CTM also has this aspect.

Second, FOT has a formal semantics, which is missing in CTM. The semantics makes a basis for reliability by preventing the "ambiguity problem" which causes faulty developments. In addition, FOT has the advantage that due to its compact model design it can be formalized with a small set of constructs. Conciseness is

not only important in a scientific sense, but also in an engineering sense since it requires less cost to learn the method and makes the method easy to extend.

Third, a SAT-based algorithm is designed and implemented for automated test-suite generation. An obvious advantage of the design is efficiency. The design can benefit from recent advances in theory and in the techniques of SAT-solvers [6]. For instance, FOT takes only about 20 seconds to generate 64200 test cases, while CTM tool[10, 11] takes 73 minutes in a similar setting. Another advantage is that the correctness of the algorithm is easy to prove as we did, making FOT more reliable; i.e., it is guaranteed the test suite generated by the algorithm is always correct (i.e., the test suite generated by the algorithm is sound and complete w.r.t a test-case design and the semantics in Definition 3.).

Fourth, FOT is equipped with several automated analysis operations for *correctness checking* for test-case designs such as *consistency checking* and detecting *dead/common* features, which are absent in CTM. These analysis operations are quite useful, and we find them in several case-studies for finding deficiencies in test-case designs by FTT, and validating test-case designs.

8 Conclusion and Future Research

Conclusion In this paper, we have developed a test-case design method for BBT called "FOT (Feature Oriented Testing)", by applying analysis and design methods of FMs originally developed for SPLs. We designed a test-case design language as a model-based and combination testing method for BBT based on FMs. A formal semantics of FTT is developed by means of test-cases; this makes a firm underpinning of the method. Also we have develop and implemented an automated test-suite generation and correctness checking of test-case designs using SAT, as computer-aided analysis techniques of the method. Furthermore, we have demonstrated feasibility of FOT with several dimensions of implementation, analysis of computational cost, experiments, a case study, and an assistant-tool development. We have also clarified the technical and practical advances of FOT to CTM, which is the-state-of-the-art testing method for BBT.

Future Research There are many directions for further research on the method. The first is to introduce to FOT other theories and computer-aided techniques of FMs, including refactoring, diagnosis and efficiency analysis. Another direction is to extend FOT with useful notions for test-case design such as the notion of priority. In addition, incorporating other testing methods for BBT such as combination testing methods (e.g., n-wise testing, etc) and input-domain analysis techniques (e.g., equivalent partitioning, boundary value analysis, etc) to FOT are important directions for our future research.

References

1. D. S. Batory. Feature models, grammars, and propositional formulas. In SPLC, pages 7–20, 2005.

2. D. Benavides, A. R. Cortes, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In XV Jornadas de Ingenieria del Software y Bases de Datos, 2006.
3. D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In CAiSE, pages 491–503, 2005.
4. F. Cao, B. R. Bryant, C. C. Burt, Z. Huang, R. R. Raje, A. M. Olson, and M. Auguston. Automating feature-oriented domain analysis. In Software Engineering Research and Practice, pages 944–949, 2003.
5. T. Y. Chen, P. L. Poon, and T. H. Tse. An integrated classification-tree methodology for test case generation. International Journal of Software Engineering and Knowledge Engineering, pages 647–679, 2000.
6. N. Eén and N. Sörensson. An extensible SAT-solver. In SAT, pages 502–518, 2003.
7. C. Ericson. Fault tree analysis - a history. In The 17th International Systems Safety Conference, 1999.
8. S. Fan and N. Zhang. Feature model based on description logics. In KES, pages 1144–1151, 2006.
9. M. Grochtmann. Test case design using classification trees. In The International Conference on Software Testing Analysis, 1994.
10. M. Grochtmann, K. Grimm, J. Wegener, and M. Grochtmann. Tool-supported test case design for black-box testing by means of the classification-tree editor. In The 1st European International Conference on Software Testing Analysis, pages 169–176, 1993.
11. M. Grochtmann and J. Wegener. Test case design using classification trees and the classification-tree editor cte. In QW, 1995.
12. R. Impagliazzo and R. Paturi. On the complexity of k-SAT. Comput. Syst. Sci., 62(2):367–375, 2001.
13. M. Janota. Do SAT solvers make good configurators? In ASPL, pages 191–195, 2008.
14. 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, Carnegie-Mellon University Software Engineering Institute, November 1990.
15. K. C. Kang, S. Kim, J. Lee, and K. Kim. FORM: a feature-oriented reuse method, annals of software engineering. Annals of Software Engineering, 5:143–168, 1998.
16. M. Mannion. Using first-order logic for product line model validation. In SPLC, pages 176–187, 2002.
17. M. Mendonca, A. Wsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In SPLC, pages 231–240, 2009.
18. OSEK/VDX operating system specification 2.2.3 http://www.osek-vdx.org/, 2005.
19. OSEK/VDX operating system test plan, version 2.0, 1999.
20. P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. Computer Networks, 51(2):456–479, Feb 2007.
21. P. Schobbens, P. Heymans, and J. C. Trigaux. Feature diagrams: A survey and a formal semantics. RE, pages 139–148, 2006.
22. J. Sun, H. Zhang, Y.F. Li, and H. H. Wang. Formal semantics and verification for feature modeling. In ICECCS, pages 303–312, 2005.
23. W. Zhang, H. Zhao, and H. Mei. Binary-search based verification of feature models. In ICSR, pages 4–19, 2011.