

Article

Locating Causes of Inconsistency in a Variability Model for Software Product Line

Younghun Han ¹, Sungwon Kang ¹  and Jihyun Lee ^{2,*} 

¹ School of Computing, Korea Advanced Institute of Science & Technology (KAIST), 291 Daehakro, Daejeon 34141, Republic of Korea; younghun.han@kaist.ac.kr (Y.H.); sungwon.kang@kaist.ac.kr (S.K.)

² Department of Software Engineering, Jeonbuk National University, Jeonju 54896, Republic of Korea

* Correspondence: jihyun30@jbnu.ac.kr

Abstract

One of the central activities of software product line development is variability modeling for a product family. Because variability models are needed at various stages of software product line development, determining whether a variability model has been modeled correctly is an essential activity for successful software product line development. Existing studies proposed various methods for analysis of various aspects of correctness of a variability model. In particular, analyzing whether a variability model is consistent or not is considered the most important analysis perspective since it is impossible to configure products from such a model. There are few studies in the software product line field that locate causes of inconsistency in a variability model. Furthermore, these existing methods cannot locate the exact causes of inconsistency due to the fact that the feature model they are based on allows ambiguity in its parent–child relationship or due to the fact that they are designed to produce explanations rather than locations of causes, resulting in producing long and complex explanations as the size of the feature model increases. In this work, we propose a method that determines whether or not a variability model has an inconsistency and identifies the exact locations of its causes if it has an inconsistency. To evaluate the proposed method, we developed a tool that automatically performs all the steps of the method and used it to conduct experiments with 49 models, including real-world variability models. As a result, the proposed method accurately identified all models with an inconsistency and located all causes of inconsistency in them.



Academic Editor: Christos Bouras

Received: 10 October 2025

Revised: 12 November 2025

Accepted: 14 November 2025

Published: 20 November 2025

Citation: Han, Y.; Kang, S.; Lee, J. Locating Causes of Inconsistency in a Variability Model for Software Product Line. *Appl. Sci.* **2025**, *15*, 12328. <https://doi.org/10.3390/app152212328>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: software product line; variability modeling; inconsistency in a variability model; inconsistency cause locating

1. Introduction

Software product line development is used to produce families of software products. By this approach, development costs can be dramatically reduced while productivity and product quality are improved [1,2]. In software product line development, the variabilities associated with a family of products are abstracted at various stages and defined explicitly in an integrated manner to play a pivotal role in configuring and creating artifacts for each product. Ultimately, variability modeling for a product family and analysis of these variability models to determine whether they are correct are both important activities crucial to the success of software product line development.

To date, a number of methods have been presented to analyze variability models. To ensure that a variability model is correctly modeled, the non-quantitative analysis perspectives such as inconsistency analysis and dead feature analysis and the quantitative analysis

perspectives such as analysis of possible configurations and analysis of variability ratio have been studied. Among these, inconsistency analysis is considered the most important, as products cannot be generated from models with an inconsistency [3]. Several possible means to analyze whether a variability model has an inconsistency have been proposed in [4–18]. However, there are few studies that locate the causes of the inconsistency, and none of them can identify the exact locations of its causes due to the fact that the feature model they are based on allows ambiguity in its parent–child relationship or due to the fact that they are designed to produce explanations rather than locations of causes, making explanations longer and more complex as the size of feature model increases. As a variability model becomes more complex or larger, more effort and time are required to analyze it manually, resulting in an increased likelihood of errors [10,19]. Thus, an analysis of variability models must be accurate, efficient, and (preferably) automated.

To fill this gap, we propose an ICL (*Inconsistency Cause Locating*) method that analyzes whether a variability model has an inconsistency or not and locates all its causes if so. This method consists of the Inconsistency Analysis phase followed by the Inconsistency Cause Locating phase. The first of these phases determines whether a variability model has an inconsistency. If an inconsistency is found, its cause is identified during the Inconsistency Cause Locating phase. The key to our method is to distinguish all the clauses associated with inconsistency of the variability model in the process of analyzing the Conjunctive Normal Form (CNF) formula representing the variability model and locate all causes of inconsistency in the variability model through tracking from these clauses to the elements of the variability model.

To evaluate the proposed method, we developed a tool that automatically performs all of these steps. We used this tool to conduct experiments with 49 models, including real-world variability models and experimental variability models. The ICL tool accurately identified all models with an inconsistency and located all causes of inconsistency in them.

The contributions of our work are as follows. First, to our knowledge, it is the first method that locates the causes of inconsistency in SVM (*Systematic Variability Modeler*) variability models. Second, the proposed method determines whether or not a variability model has an inconsistency and identifies the exact locations of its causes if it has an inconsistency. Third, when tested, the proposed method accurately identified all inconsistencies and located their causes of inconsistency in the SVM models that we experimented with. Fourth, this method is fully automatable and has an associated tool that we developed based on it.

The research in this paper that locates the causes of inconsistency in a variability model is challenging for the following reasons. If a variability model is converted to an intermediate representation, i.e., a CNF formula, in a naive way, then much of the information in the variability model is lost. Therefore, (1) capturing information related to the causes of inconsistency in an intermediate representation is not straightforward. Moreover, once the necessary information related to causes is captured in the intermediate representation, (2) analyzing inconsistency from the intermediate representation while preserving traceability to the causes is not straightforward. Our method has solved both of these challenges by building traceability between the SVM variability model and its CNF formula representation and capturing key information related to the causes of inconsistency from the CNF formula.

The remainder of this paper is organized as follows. In Section 2, we discuss related work on consistency analysis of variability models. Section 3 presents the background information that is necessary to understand our proposed method. Section 4 describes our proposed ICL method in detail. In Section 5, we evaluate our method and discuss the experimental results. In Section 6, we conclude this paper.

2. Related Work

In this section, we discuss prior studies on the consistency analysis perspective and consistency analysis methods for variability models.

A number of variability models have been proposed, including Feature Models [20–22] and the Orthogonal Variability Model (OVM) [2]. The analysis perspectives applicable to these variability models vary widely. Our discussion focuses on the analysis perspectives associated with feature models, the most widely used type of variability model.

Benavides et al. [23] investigated, compared, and evaluated a wide range of studies on the analysis of feature models, identifying approximately 30 analysis perspectives proposed between 1990 and 2009. There are analysis perspectives that validate the correct model is modeled and quantitative analysis perspectives such as an analysis of the number of potential products that can be configured from the feature model. According to these authors, the study of analysis perspectives that validate whether the correct model has been created has been more active than the study of quantitative analysis perspectives.

Among these analysis perspectives, consistency analysis is considered the most important analysis perspective [3]. This is because if there is an inconsistency in the feature model, a potential product cannot be configured from that model. On the other hand, other analysis perspectives target a model that can configure potential products from a feature model.

Consistency analysis is also called “Model Validation”, “Model Consistency Checking”, “Model Satisfiability Checking”, “Model Solvability Checking”, and “Model Constraints Checking” by other researchers. However, their roles are the same, including (1) analysis of a model that cannot configure potential products from a feature model or (2) analysis of a model that has zero potential number of products that can be configured from a feature model.

A number of methods have been developed to perform consistency analysis of feature models. Most of these [4,7–9,12,14,15] use Propositional Logic as the basis for their analysis.

Mendonca et al. [8] introduced Software Product Lines Online Tools (SPLOT) [9], a web-based online tool for software product line development. The tool provides feature modeling, analysis of a feature model, and configuration of potential products. In analysis of a feature model, the tool provides a variety of analysis perspectives, including consistency analysis, dead feature analysis, common feature analysis, analysis of the number of potential products, and so on. Mendonca et al. also demonstrate the practical aspects of the SAT Solver-based feature model analysis tool by presenting in [24,25] that it is very efficient to analyze over SAT Solver even when the feature model grows in size. From a practical point of view, there is also a study to evaluate SAT solvers using industrial feature models. Chico et al. [26] empirically evaluated nine publicly available #SAT solvers in 127 industrial feature models, showing that highly efficient analysis is possible in 125 models. However, in this tool, it is possible to analyze whether the feature model has an inconsistency, but it does not locate the causes of the inconsistency.

Zhang et al. [7] proposed a propositional logic-based method to validate a feature model at different binding times. The most important characteristic of their method is that it integrated logical validation and binding time, and thus it is capable of validating only part of a feature model to configure specific bindings. In this work, the validation of the feature model is transformed into a Satisfaction Problem, which requires the analysis of three properties: Satisfiability, Usability, and Suitability. Satisfiability analysis determines whether the feature model has an inconsistency without locating the cause.

Mannion [12] developed a technique that uses propositional logic connectives and calculus to validate product line models. In this work, product line models are represented as logical expressions, which are instantiated by the requirements selected for a single

system and are also used to analyze whether the selected instance is valid or invalid. In other words, there is no inconsistency in a product line model if any valid single system is instantiated from the model. This work utilizes First-Order Logic to analyze whether a variability model contains an inconsistency, but it does not locate associated causes.

Batory [4] conducted research to reveal the fundamental associations between feature models, grammar, and propositional formulas. This connection enabled feature model analysis by lightweight, efficient, and easy-to-build Logic Truth Maintenance Systems (LTMSs) and off-the-shelf SAT solvers. By equating feature models with propositional formulas, the analysis was substantially simplified. The propositional formula was analyzed with an SAT solver to determine whether a feature model contained an inconsistency. Once again, no causes were located.

Thüm et al. [14] proposed FeatureIDE [15], a scalable framework for feature-oriented software development. Feature-oriented software development is a paradigm for the development, customization, and synthesis of software systems. The FeatureIDE framework encompasses the entire process of software product line development and provides an Eclipse-based integrated development environment (IDE) for implementation. Domain analysis performed in FeatureIDE uses SAT4j to perform, for example, dead feature analysis, false optional feature analysis, and inconsistency analysis through dead feature analysis [16,27] to check for anomalies. Specifically, the study of [16] provides a way of explaining an inconsistency for FeatureIDE. Then, based on the provided explanation, users may manually resolve the cause of inconsistency. However, as the number of the points of a variability model that cause inconsistency, the number of nodes of a variability model, or the height of a variability model increases, the explanation becomes longer and manually resolving the cause of inconsistency based on the provided explanation becomes more difficult.

Alternative methods use the Constraint Satisfaction Problem (CSP) [5,6], description logic [10,11], first-order logic [13,28] and the dynamic-priority-based approach [17]. Trinidad et al. [5] presented an analysis framework that enabled automated error analysis to create error-free feature models. Wang et al. [10,11] presented a Semantic Web environment for feature modeling and analysis of feature models in the absence of formal semantics of feature models. The approach of Sun et al. [13] used the formal inference technique for the formalization and validation of feature models. Wang et al. [17] presented a dynamic-priority-based approach to interactively fixing inconsistencies in feature models. Since this approach removes constraints that are lower than those of high interest depending on the pre-determined user's interest when fixing inconsistencies, it is different from locating all the causes of the inconsistencies proposed in this paper.

The commonality between prior approaches is that they indirectly analyze a variability model by converting it to an intermediate representation without directly analyzing the variability model to detect inconsistency in the variability model. On the other hand, the differences between these approaches are that they use various different intermediate representations ranging from SAT formulas, description logic formulas, and first-order logic formulas to the constraint satisfaction problem. In addition, the commonality between the prior approaches and ours is that both the prior approaches and our method conduct inconsistency analysis to determine whether a variability model is consistent or not.

Our approach differs from the prior ones mainly in two respects.

First, the prior approaches and ours have different base models. Most prior approaches perform inconsistency analysis for feature models, whereas our method performs inconsistency analysis for the SVM variability model that overcomes many limitations of the existing feature modeling methods, as explained in Section 3.2 and in [29]. Notably, in the SVM variability model, the variability relationships and composed-of relationships

are clearly separated. Analysis of the prior approaches could produce ambiguous results because the feature model on which their approaches are based represents variability relationships and composed-of relationships ambiguously in the same parent–child relationships and consequently makes them unable to pinpoint which relationship has actually caused inconsistency.

Second, our method identifies the exact locations that are the causes of inconsistency in SVM variability models, whereas the prior approaches cannot identify them. The reason is that when a feature model is converted into an easy-to-analyze form, i.e., CNF formulas, this process involves the loss of information of which parts of the feature model correspond to which parts of the CNF formulas, making it difficult to track the causes of inconsistencies in the feature model. Some studies [5,11,13,16] produce explanations which are useful in fixing inconsistencies, but they do not identify the exact locations that are the causes of inconsistency. As the size of the feature model increases, explanations become long (since the height of the feature model would increase) and complex (since the number of causes is likely to increase), which makes it more difficult to locate the causes of inconsistency even when it is performed manually. Furthermore, these studies have the inherent problem of containing ambiguity in their feature models as mentioned above. For these reasons, we use with our method the term “location of cause”, which is more exact than the term “explanation”.

There are some seemingly related works such as [18,30], which are, however, not closely related to our work for the following reasons. Hentze et al. [18] proposed the notion of “hyper explanation” to solve the multiple dead features problem, which extends the existing notion of “explanation” that solves the single dead feature problem. Explanation is a concept useful for solving the dead feature problem, whereas cause locating, the subject of our study, finds the exact point at which an inconsistency has occurred. Le et al. [30] proposed a feature model testing and debugging approach that proactively supports feature model designers. This approach focuses on analysis of anomalies such as dead features and false optionals, whereas our method focuses on inconsistency.

3. Background Information

This section presents background information necessary to understand proposed method.

3.1. Definitions

In this paper, we use the following terms. A propositional variable is a variable that can take the truth value of True or False, called constants. Propositional variables, their negations and the two constants, True and False, are called literals. A clause is an expression in which a finite number of literals form a disjunction. A Conjunctive Normal Form (CNF) formula is an expression in which a finite number of clauses form a conjunction. If a clause of a CNF formula is a literal, then the clause is called a literal clause.

Example 3.1. *In the following CNF formula*

$$F1 = (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee \neg B)$$

two propositional variables (i.e., A , B) occur; there are six literals (i.e., $\neg A$, B , A , $\neg B$, $\neg A$, $\neg B$); and there are three clauses (i.e., $(\neg A \vee B)$, $(A \vee \neg B)$, $(\neg A \vee \neg B)$). If $A = \text{True}$, then $F1$ evaluates to a conjunction of clauses as follows.

$$F2 = (B) \wedge (\text{True}) \wedge (\neg B)$$

because the first clause evaluates to '(B)', the second clause evaluates to '(True)' and the third clause to '¬(B)'. In F2, '(B)', '(True)', and '¬(B)' are literal clauses.

If there are two or more literal clauses in a CNF formula such that one is the negation of the other, we say that there are conflicting literal clauses in the CNF formula, and the CNF formula has a contradiction. Each conflicting literal clause is represented by <literal clause: ¬literal clauses>.

Example 3.2. In F2 of the example above, '(B)' and '¬(B)' are conflicting literal clauses, so we say that there is a contradiction in F2, hence there is a contradiction in F1.

3.2. Variability Modeling with SVM

Systematic Variability Modeler (SVM) [31,32], developed in our study, is a tool based on Compositional Feature Modeling (CFM) [32] and overcomes many of the limitations of the existing feature modeling methods [20–22], which were presented in Kang et al. [29], by supporting systematic variability modeling and analysis of variability models. With feature models, it is difficult, in particular, to explicitly model the composed-of relationship and variability relationships as two different kinds of relationships due to the absence of variation points. SVM solves this problem by supporting the notions of variation point and variant and making them be explicitly modeled, which were also the main concepts supported by Orthogonal Variability Modeling (OVM) [2], an innovative variability modeling method introduced by Pohl et al. We think that these concepts are essential to model a user's product line requirements. In the remainder of this paper, when a variability model is mentioned, it denotes a variability model modeled with SVM.

SVM models a variability model using SVM elements. SVM elements consist of inclusion relationship, two variability dependencies, and two variability constraints as shown in Table 1.

The inclusion relationship between a parent node and a child node requires that the child node must be included when the parent node is included. The graphical notation of inclusion relationship connects the parent node and the child node in a solid line as shown in the second column of Table 1. The textual notation of it is I<parent node, child node>, and I is an abbreviation for inclusion relationship.

The two variability dependencies are alternative relationship and optional relationship. The alternative relationship between a parent node and its child nodes requires that one of the child nodes must be included when the parent node is included. The textual notation of alternative relationship is A <parent node,{child node1, ..., child node_n}>, and A is an abbreviation for alternative relationship. The optional relationship between a parent node and its only child node requires that the child node may or may not be included when the parent node is included. If the child node is NULL, it means that the child node is not selected. The textual notation of the optional relationship is O <parent node,{child node, NULL}>, and O is an abbreviation for the optional relationship. A special case of the alternative relationship occurs when the child nodes of the alternative relationship are obtained as a subset of a set of child nodes. We will call this relationship a collection alternative relationship. The textual notation of a collection alternative relationship is C <parent node,{child node1, ..., child node_n}>, and C is an abbreviation for a collection alternative relationship. In these relationships, the parent node is called a variation point [2], and the child nodes are called a variant [2].

Table 1. Graphical and textual notations of SVM elements.

SVM Elements		Graphical Notation ¹	Textual Notation
Variability Dependencies	Inclusion Relationship		$I \langle A, B \rangle$
	Alternative Relationship		$\langle vpA, \{B1, B2, \dots, Bn\} \rangle$
	Collection Alternative Relationship		$\langle vpA, \{B1, B2, \dots, Bn\} \rangle$
	Optional Relationship		$O \langle vpA, \{B, NULL\} \rangle$
Variability Constraints	Requires Constraint		$R \langle A, B \rangle$
	Excludes Constraint		$E \langle A, B \rangle$

¹ To clearly distinguish the alternative relationships in graphical notation, SVM displays green color in the variation point (vp) of the collection alternative relationship.

To clearly distinguish between inclusion relationship and variability dependencies, SVM uses the notions of variation point and variant to represent variability, and the name of the node for a variation point is prefixed with “vp”.

The two variability constraints are the requires constraint and excludes constraint. The requires constraint between a source node and a target node requires that the target node must be included when the source node is included. The textual notation for the requires constraint is $R \langle \text{source node}, \text{target node} \rangle$, and R is an abbreviation for the requires constraint. On the other side, the excludes constraint between a source node and a target node requires that the target node must not be included when the source node is included. The textual notation for the excludes constraint is $E \langle \text{source node}, \text{target node} \rangle$, and E is an abbreviation for the excludes constraint. In SVM, the graphical notation of the requires constraint is represented by a dotted line annotated with “REQ”, while the graphical notation for excludes constraint is represented by a dotted line annotated with “EXCL”.

With these five SVM elements, all the elements of feature models [23] can be expressed.

Example 3.3. (Example SVM variability model).

An example feature model for a mobile phone [23] is shown in Figure 1. Figure 2 shows an SVM variability model that is equivalent to the feature model in Figure 1.

In a feature model, the inclusion relationship and variability dependencies are not explicitly differentiated. For example, in the case of the relationship between the mobile phone and GPS in Figure 1, the inclusion relationship and variability dependency are mixed because it represents that the mobile phone may include GPS, which means that the mobile

phone includes something that is GPS or does not include anything. Such expressions with mixed relationships are hard to understand and reason with. In contrast, in an SVM variability model, these two concepts are clearly distinguished through the explicit use of the notions of variation point and variant as exemplified with vpGPS and GPS in Figure 2. In Figure 2, vpGPS is a variation point, as the prefix “vp” indicates, and GPS is a variant. According to this model, the mobile phone includes vpGPS that is either GPS or nothing. So, the relationship between the mobile phone and vpGPS is an inclusion relationship, and the relationship between vpGPS and its variants GPS and NULL is an optional relationship. In an SVM variability model, if a node has internal variability, i.e., variability within the node itself, the name of the ancestor node is underlined.

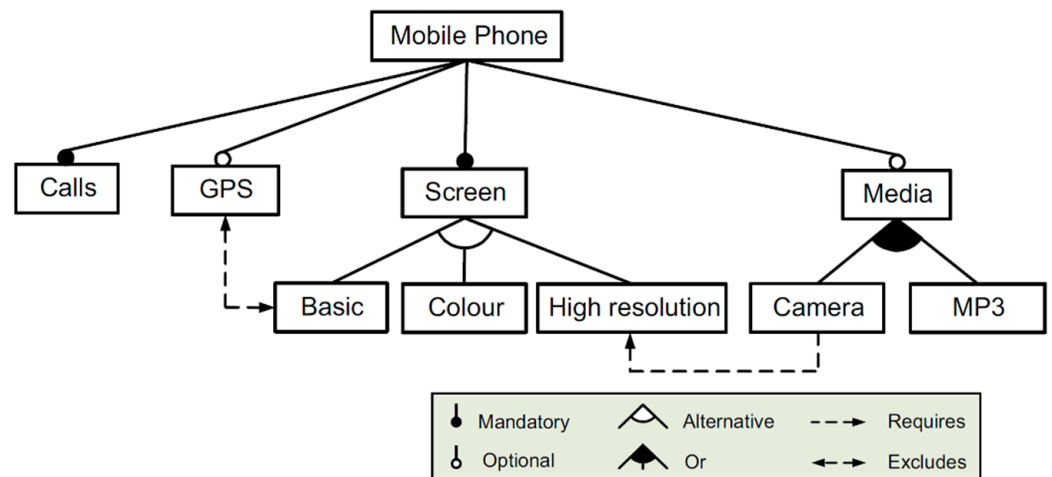


Figure 1. Example feature model for mobile phone [23].

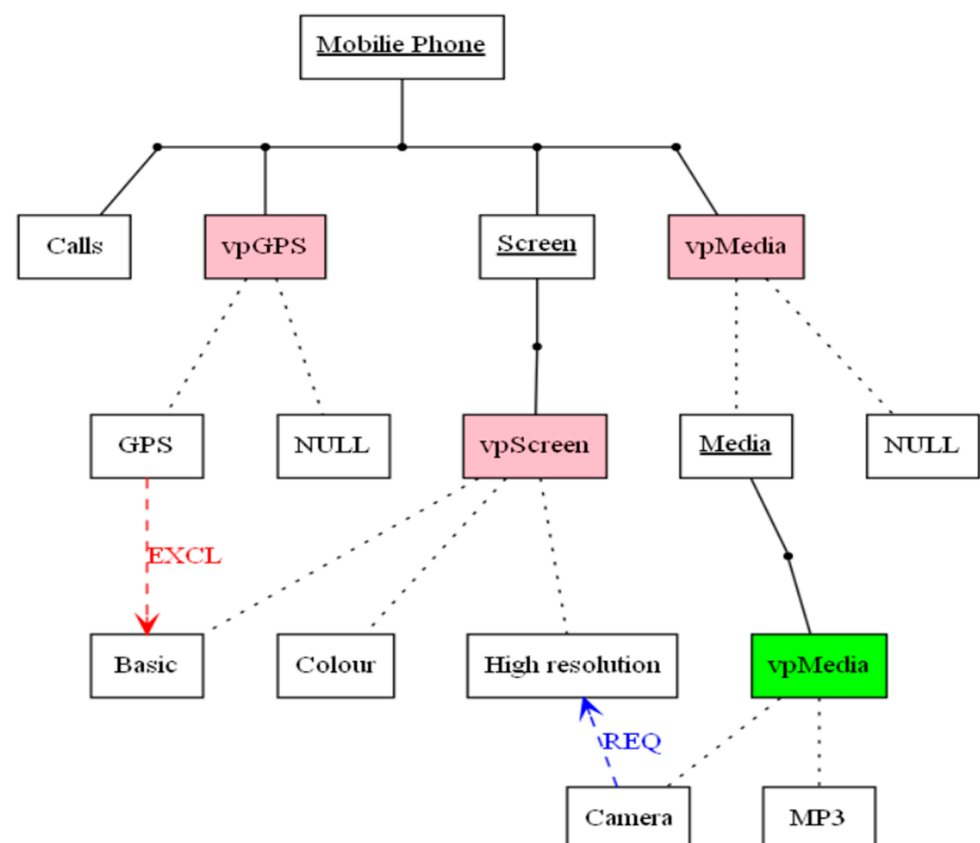


Figure 2. SVM variability model equivalent to the feature model in Figure 1.

3.3. Inconsistent Model

In logic, a set of statements is said to be consistent if they can be true together and inconsistent otherwise ([33] p. 3). So, for a consistent set of statements Σ , there is at least one truth value assignment to each atomic statement, constituting the statements which make each statement of Σ true. The purpose of the variability model is to define a set of potential products from this model for software product line development. Therefore, from the perspective of the variability model, a variability model can be said to be consistent if at least one potential product can be configured from the model and inconsistent otherwise. Specifically, a conflicting SVM component is a pair of SVM elements such that one element requires inclusion of a node of the variability model while the other element requires exclusion of the same node. If a variability model contains a conflicting SVM component, we say that the model has inconsistency, and we call such a model an inconsistent model (that is, the same as an inconsistent variability model). Therefore, when there is a conflicting SVM component in a variability model, products cannot be generated from the model because it is impossible to configure products from such a model due to the conflicting inclusion requirements on a node. We call a variability model that does not have inconsistency a consistent model.

According to David Benavides et al. [23], “A feature model is void if it represents no products.” and “A feature is dead if it cannot appear in any of the products of the software product line.” Therefore, if all features of a feature model are dead, it is a void model. Also, if the root of a feature model is dead, it is a void model. The two examples in our work, Figure 3b and Figure 8b (It is described in Section 5.5.), show the case where all features are dead and thus the variability model is void. The example of Figure 8a (It is described in Section 5.5.) shows the case where some features are dead but the variability model is not void. Our method works by locating a conflicting SVM component and, upon finding one, declaring that the variability model is inconsistent. So, although our method does not directly work with void models, it can determine whether a model is void by locating a conflicting SVM component.

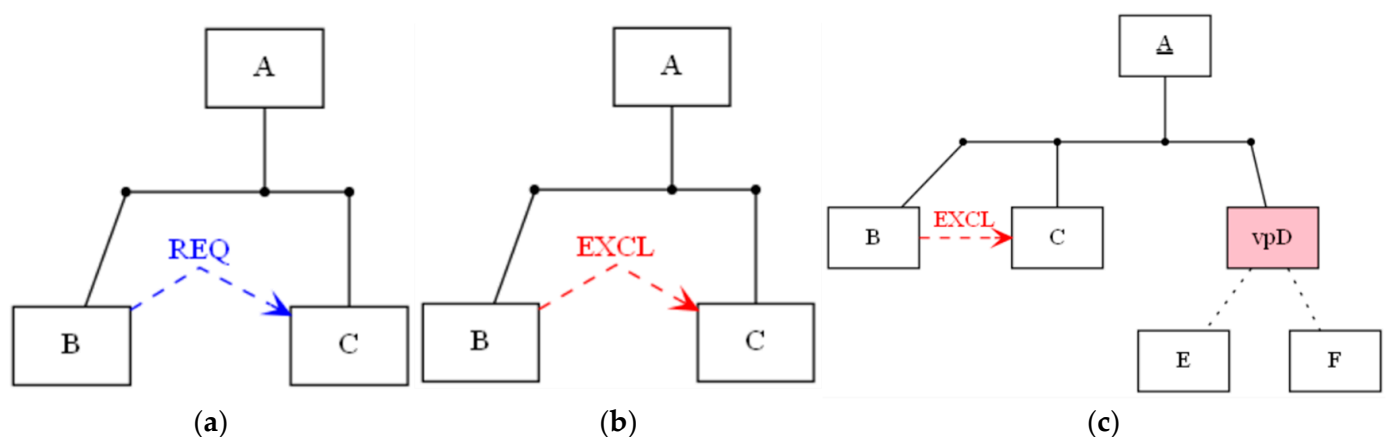


Figure 3. Examples of consistent model and inconsistent model. (a) consistent model; (b) inconsistent model; (c) inconsistent model with variation point.

This paper complies with the classification scheme for feature model analysis presented in [3], and only the most important analysis perspective, “inconsistency analysis,” is the scope of the paper. The anomalies analysis perspectives such as dead feature analysis and false optional feature analysis are not within the scope of this paper. Although the various anomalies analysis perspectives are not in the scope of this paper, we plan to enhance the proposed method so that various causes of anomalies analysis perspectives such as

dead feature analysis and false optional feature analysis can be located. In particular, by extracting and analyzing the information related to the cause of a dead feature from the CNF formula, it will be possible to accurately locate the cause of the dead feature in the variability model.

Figure 3 shows examples of a consistent model and an inconsistent model. Figure 3a,b has three nodes A, B, and C, and A includes B and C. In Figure 3a, B and C are constrained by the requires constraint, which is represented by the arrow annotated with 'REQ'. The model is consistent because C must be included if B is included due to the requires constraint, so there is no conflict and the variability model in Figure 3a can produce products as its instances.

In contrast, in Figure 3b, B and C are constrained by the excludes constraint, which is represented by the arrow annotated with 'EXCL'. This model is inconsistent because C cannot be included if B is included due to the excludes constraint, so the inclusion of C required by the inclusion relationship and the exclusion of C required by the excludes constraint conflict with each other. In Figure 3b, $I \langle A, C \rangle$ and $E \langle B, C \rangle$ make a conflicting SVM component and $I \langle A, B \rangle$ and $E \langle B, C \rangle$ make another conflicting SVM component.

We call two SVM elements that together make a conflicting SVM component a cause of inconsistency. We call two SVM elements that are located by our ICL method as a conflicting SVM component a located cause of inconsistency. In Figure 3b, there are two located causes of inconsistency, and they are:

Cause 1: $I \langle A, C \rangle$ and $E \langle B, C \rangle$ are a cause of inconsistency because they make a conflicting SVM component.

Cause 2: $I \langle A, B \rangle$ and $E \langle B, C \rangle$ are a cause of inconsistency because they make a conflicting SVM component.

In order to describe a located cause of inconsistency, the following notation is used:

(an SVM element E :{the set of SVM elements in conflict with E })

The two located causes of inconsistency above are described using this notation as follows:

Cause 1: ($I \langle A, C \rangle$:{ $E \langle B, C \rangle$ })

Cause 2: ($I \langle A, B \rangle$:{ $E \langle B, C \rangle$ })

For the same cause of inconsistency, different descriptions are possible. For example, Cause 1 could be described as ($E \langle B, C \rangle$, { $I \langle A, C \rangle$ }) too. In our method, the SVM element described in the left side of the colon of a located cause of inconsistency is called the Major Element, and a set of SVM elements described in the right side of the colon, which are in conflict with the Major Element, are called the Minor Elements. In Cause 1 of the example above, the Major Element can be any SVM element in the conflicting SVM components if there are only two SVM elements in conflicting SVM components.

The difference between the examples in Figure 3b,c is the presence and absence of variability. The example in Figure 3c is an inconsistent model, and the cause of inconsistency is the same as that of the example in Figure 3b. In contrast, Figure 3c is an example focusing on variation point rather than inconsistent model and clearly shows the distinction between the composed-of relationship and the variability relationships by explicitly modeling variation points. FM differs from SVM in that it has no explicit notation for variability relationship and thus edges from a parent node to child nodes are ambiguous in the sense that they can be interpreted as a composed-of relationship or a variability relationship or a mixture of these two relationships.

Prior approaches can also detect inconsistency for two variability models shown in Figure 3. However, they do not locate the causes of inconsistency in the variability model

shown in Figure 3b. The reason is that, in the prior approaches, when the variability model is converted to an intermediate representation, i.e., a CNF formula, much of the information in the variability model is lost. On the other hand, our proposed method carries in our intermediate representation the information about the elements that cause inconsistency. The explanation is a concept that facilitates solving inconsistency, whereas inconsistency cause locating, the subject of our study, finds the exact location at which an inconsistency has occurred. If repair is conducted from the cause located from our method, the repair should be performed manually but can be performed immediately because the exact location to repair has been pinpointed by our method. In contrast, the method of [16] may not point to the exact location and may provide only broad guidelines that could lead to the exact location. So, for the purpose of repairing, our method will be superior to [16].

This description of a located cause of inconsistency, however, only shows the very place where a conflict occurs by exhibiting the conflicting SVM components involved in the conflict. However, a cause of inconsistency alone does not provide all relevant SVM elements when the conflicting SVM components are imbedded in a variability model. See Section 5.5 Figure 8b for such an example. A complete description of a cause of inconsistency, to be explained in detail in Section 5.5, shows not only the conflicting SVM components but also all the SVM elements that participate in the creation of a conflict.

3.4. Davis–Putnam–Logemann–Loveland (DPLL) Algorithm

In logic, the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [34,35] is a recursive backtracking-based search algorithm for determining the satisfiability of a propositional formula in conjunctive normal form. Figure 4 shows the algorithm. Using the two rules, (1) unit-propagation rule and (2) splitting rule, it runs by choosing a variable, assigning a truth value to it, simplifying the CNF formula, and then recursively checking if the simplified CNF formula is satisfiable. If the simplified CNF formula is found to be satisfiable, then it declares that the original CNF formula is satisfiable; otherwise, the same recursive check is performed assuming the opposite truth value.

The DPLL Algorithm

Given a set of clauses Δ defined over a set of variables Σ , this algorithm returns “satisfiable” if Δ is satisfiable and returns “unsatisfiable” otherwise. This algorithm should be called with $\text{DPLL}(\Delta)$.

Then, it runs as follows:

1. If $\Delta = \emptyset$, then return “satisfiable”
2. If $\square \in \Delta$, then return “unsatisfiable” // $\square$ is the empty clause
3. **(Unit-propagation rule)** If Δ contains a unit-clause C , assign a truth value to the variable in C that satisfies C , simplify Δ to Δ' , and return $\text{DPLL}(\Delta')$.
4. **(Splitting rule)** Select from Σ a variable v which has not been assigned a truth value.
 - Assign one truth value true to it, simplify Δ to Δ' , and call $\text{DPLL}(\Delta')$.
 - 4.1. If the call returns “satisfiable”, then return “satisfiable”.
 - 4.2. Otherwise, assign the opposite truth value to v in Δ , simplify to Δ'' , and return $\text{DPLL}(\Delta'')$.

Figure 4. The DPLL algorithm [34].

4. The Proposed Method: ICL

This section describes our proposed method: Inconsistency Cause Locating. Section 4.1 describes the workflow of the ICL method, while Section 4.2 describes the steps of the ICL method and provides examples.

4.1. The Workflow of the ICL Method

Figure 5 shows the workflow of the ICL method depicted using the activity diagram, which provides a methodological description and a visual representation of the ICL method. The input to the ICL method is an SVM Variability Model, and the outputs are Contradiction Analysis Result and List of Causes. The ICL method consists of an Inconsistency Analysis phase (light gray part in Figure 5) and an Inconsistency Cause Locating phase (dark gray part in Figure 5).

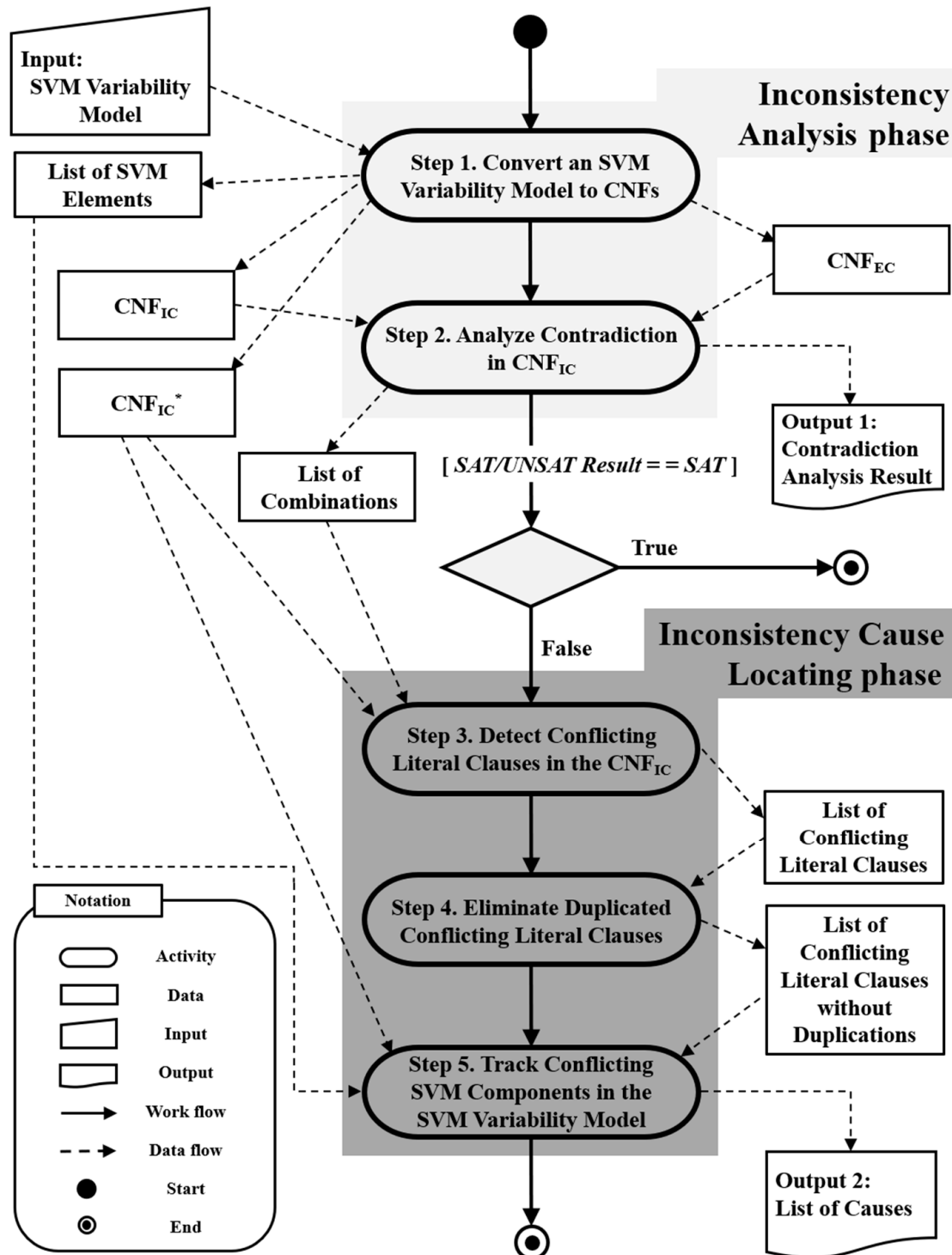


Figure 5. The workflow of the ICL method.

In the Inconsistency Analysis phase, the SVM Variability Model is converted to the Conjunctive Normal Form (CNF) formula (Step 1 of Figure 5). Whether the variability model from which the CNF formula was derived is consistent is then determined, analyzing whether the CNF contains a contradiction (Step 2). If the variability model is inconsistent, the Inconsistency Cause Locating phase is triggered; otherwise, the whole process of the ICL method terminates in this phase. The Contradiction Analysis Result presents the conclusion reached as to whether the SVM variability model is consistent or not.

In the Inconsistency Cause Locating phase, the ICL method locates causes of any inconsistencies in the variability model. Step 3 of Figure 5 shows where conflicting literal clauses associated with conflicting SVM components in a variability model are detected. After removal of any duplicated conflicting literal clauses detected in Step 4, the conflicting SVM components are tracked by analyzing the conflicting literal clauses (Step 5). Removal of duplicate causes is performed to prevent inefficiencies in the tracking of conflicting SVM components in Step 5.

4.2. Steps of the ICL Method

This section provides a detailed description of each step of the ICL method.

4.2.1. Step 1: Convert an SVM Variability Model to CNFs

In Step 1 of the ICL method, an SVM variability model is converted to CNFs. Analyzing the variability model itself is a time-consuming task and is prone to errors in the results. The variability model is thus converted to an easy-to-analyze form such as CNFs and then analyzed, and the reason why the ICL method converts the variability model to CNFs is to analyze the variability model using SAT Solver-based algorithms.

The ICL method converts the variability model to the following three formulas (IC is short for Including Constraints, and EC is short for Excluding Constraints):

CNF_{IC} : the formula obtained by converting a variability model.

CNF_{EC} : the formula obtained by converting a variability model excluding variability constraints.

CNF_{IC}^* : the formula obtained by assigning a unique subscript index to each of the literals in CNF_{IC} .

Prior studies only converted a variability model to CNF_{IC} . Our method uses CNF_{EC} to detect conflicting literal clauses in the event that CNF_{IC} contains contradictions associated with inconsistency. There is no possibility of a contradiction associated with an inconsistency in CNF_{EC} , but there may be contradictions associated with an inconsistency in CNF_{IC} . When a variability model without constraints is converted to the CNFs, CNF_{IC} and CNF_{EC} are identical. Thus, the existence of a contradiction associated with the inconsistency of a variability model in CNF_{IC} can be determined using CNF_{EC} . CNF_{IC}^* is used to track where the analysis results of CNF_{IC} affect the variability model.

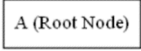
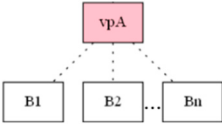
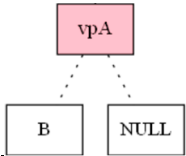
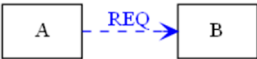
Table 2 lists the conversion rules used to convert a variability model to CNF formulas. A variability model is converted to CNFs by using the rules in Table 2 with the following four substeps.

Step (1-1) A variability model for a product family has the tree structure (a variability model that does not have the tree structure requires a process to convert the model to an SVM variability model). In this substep, the variability model is decomposed into SVM elements in a top-down manner, starting from the root node to the leaf nodes. When decomposed, each element of the variability model corresponds to one of the six elements of Table 2, which includes the root node as an SVM element.

Step (1-2) The SVM elements are converted into components of a CNF formula. The conversion rules in Table 2, with the exception of the conversion rule for the root node, follow those in Figure 15 of [23]. The CNF variable that corresponds to the root node in a

variability model is always True because the root node must be included in all products. Thus, the root node is converted to a literal clause expressed as (A).

Table 2. Conversion rules from SVM variability model to CNF.

SVM Elements	Components of CNF Formula	
	(A)	
	$(\neg A \vee B) \wedge (A \vee \neg B)$	
	When Bi's, $1 \leq i \leq n$, are non-collection elements	$(\neg A \vee B1 \vee B2 \vee \dots \vee Bn) \wedge (\neg B1 \vee \neg B2) \wedge \dots \wedge (\neg B1 \vee \neg Bn) \wedge (\neg B1 \vee A) \wedge (\neg B2 \vee \neg B3) \wedge \dots \wedge (\neg B2 \vee \neg Bn) \wedge (\neg B2 \vee A) \wedge (\neg Bn-1 \vee \neg Bn) \wedge (\neg Bn-1 \vee A) \wedge (\neg Bn \vee A)$
	When Bi's, $1 \leq i \leq n$, are collection elements	$(\neg A \vee B1 \vee B2 \vee \dots \vee Bn) \wedge (\neg B1 \vee A) \wedge (\neg B2 \vee A) \wedge \dots \wedge (\neg Bn \vee A)$
	$(A \vee \neg B)$	
	$(\neg A \vee \neg B)$	

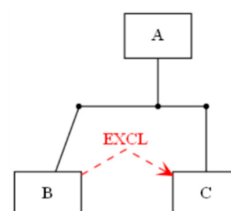
Step (1-3) CNF_{IC} and CNF_{EC} are generated by connecting with conjunction the components of the CNF formula obtained in Step 1-2. CNF_{IC} is generated by connecting with conjunction all the components of the CNF formula obtained in Step 1-2, while CNF_{EC} is generated by connecting with conjunction all the components of the CNF formula, with the exception of the components for constraints.

Step (1-4) CNF_{IC}^* is generated by assigning a unique subscript index to each literal of CNF_{IC} .

The following is an application example for Step 1 reflecting a conversion of the SVM variability model in Figure 3b to the associated CNF formulas (referred to as CNF1).

Example 4.1. (Step 1. Convert an SVM variability model in Figure 3b to CNFs).

Input:



Step (1-1) The SVM variability model in Figure 3b is decomposed into the SVM elements in the first column of Figure 6.

SVM Elements	Components of CNF1
	(A)
	$(\neg A \vee B) \wedge (A \vee \neg B)$
	$(\neg A \vee C) \wedge (A \vee \neg C)$
	$(\neg B \vee \neg C)$

Figure 6. SVM elements of Figure 3b and components of CNF1.

Step (1-2) Each SVM element is converted to the corresponding component for CNF1 on the second column of Figure 6 using the conversion rules in Table 2.

Step (1-3) CNF1_{IC} and CNF1_{EC} are generated by connecting with conjunction the components of CNF1 on the second column of Figure 6.

$$\text{Output 1: CNF1}_{IC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C) \wedge (\neg B \vee \neg C)$$

$$\text{Output 2: CNF1}_{EC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C)$$

Step (1-4) CNF1_{IC}^{*} is generated by assigning a unique subscript index to each literal of CNF1_{IC} obtained in Step 1-3.

$$\text{Output 3: CNF1}_{IC}^* = (A_1) \wedge (\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2) \wedge (\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2) \wedge (\neg B_3 \vee \neg C_3)$$

The example in Figure 3c can be converted to a CNF formula as follows.

$$\text{CNF}_{IC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C) \wedge (\neg B \vee \neg C) \wedge \underline{(\neg A \vee D) \wedge (A \vee \neg D) \wedge (\neg D \vee E \vee F) \wedge (\neg E \vee \neg F) \wedge (\neg E \vee D) \wedge (\neg F \vee D)}$$

In the CNF_{IC} above, the part that differs from Output 1 in Example 4.1 is underlined. The underlined part is the result of converting an alternative relationship.

The following is an application example for Step 1 reflecting a conversion of the SVM variability model in Figure 3a to the associated CNF formulas (referred to as CNF2).

Example 4.2. (Step 1. Convert an SVM variability model in Figure 3a to CNFs).

Further details of Example 4.2 can be found in Appendix C.

CNF_{IC} , CNF_{EC} , and CNF_{IC}^* are CNF formulas with variable A, B, and C. Each variable can have a True or False truth value, and the truth values of the CNF_{IC} , CNF_{EC} , and CNF_{IC}^* are determined by the combinations of truth values of A, B, and C.

If no combination of truth values for the variables in a CNF formula makes the formula true, then the variability model from which the CNF formula has been obtained by applying Steps 1-1 through 1-4 of our method is an inconsistent model.

4.2.2. Step 2: Analyze Contradiction in CNF_{IC}

To analyze contradiction in CNF_{IC} , Step 2 evaluates the truth values of CNF_{IC} and CNF_{EC} using all combinations of truth values of all of the variables in the CNF_{IC} formula. In this step, the combinations are associated with any inconsistencies in the variability model, and then a list of combinations is generated. A list of combinations is a finite sequence of combinations, and a combination is a triple $\langle (a), (b), (c) \rangle$, where (a), (b), and (c) are as follows:

- (a) A combination of truth values of all variables that make “ CNF_{IC} is False and CNF_{EC} is True”.
- (b) The result of an evaluation of the truth value of a CNF_{IC} is False.
- (c) The result of an evaluation of the truth value of a CNF_{EC} is True.

$\langle \{A = \text{True}, B = \text{True}, C = \text{True}\}, CNF_{IC} = \text{False}, CNF_{EC} = \text{True} \rangle$ is an example of a combination. This definition is recursive and the first component of a combination can be arbitrarily long, but its length is bounded by the number of nodes in a variability model.

The results of an evaluation of the truth value of a CNF are represented as E [CNF], which is called the Evaluation Result. If one or more True exists among the evaluation result, the SAT/UNSAT Result of the CNF is referred to as SAT (Satisfiable). If no True exists, the CNF is referred to as UNSAT (Unsatisfiable).

This step consists of the following three substeps:

Step (2-1) This step evaluates all combinations of truth values of all variables in CNF_{IC} and CNF_{EC} formulas to generate evaluation results of CNF_{IC} and CNF_{EC} .

Step (2-2) In this substep, the SAT/UNSAT results of CNFs are analyzed using the evaluation results of CNF_{IC} and CNF_{EC} .

Step (2-3) This step analyzes the differences between the evaluation results of CNF_{IC} and CNF_{EC} when the SAT/UNSAT result of CNF_{IC} is UNSAT and then generates a List of Combinations.

In Step 2-1, for each variable, if a truth value is True, it means that the corresponding node is selected in the variability model, and if a truth value is False, it means that no corresponding node is selected in the variability model. If the truth value of CNF_{IC} and CNF_{EC} is False, then there is contradiction in the CNF_{IC} and CNF_{EC} formula. This occurs when a product is configured with a combination of nodes that cannot be established in a variability model. The truth value of CNF_{EC} that is false represents a combination of nodes that cannot be configured with a potential product from a variability model. The truth value of CNF_{IC} that is false represents a combination of nodes that cannot be configured with a potential product from a variability model, as in CNF_{EC} , but some of those combinations are associated with conflicting SVM components in the variability model.

In Step 2-2, the SAT/UNSAT result of CNF_{IC} can be either SAT or UNSAT. On the other hand, the SAT/UNSAT result of CNF_{EC} can only be SAT. According to Mendonca [36], CNF_{EC} has proven that there is no contradiction associated with inconsistency in a variability model, so there is one or more True in the evaluation result of CNF_{EC} . If the SAT/UNSAT result of CNF_{IC} is SAT, it is possible to configure the potential products from the variability model, and the number of True in the evaluation result of CNF_{IC} is the number of potential products. For example, one product can be configured in a variability model corresponding

to CNF_{2IC} . If the SAT/UNSAT result of CNF_{1IC} is UNSAT, the evaluation result of CNF_{1IC} has all False, and the product cannot be configured in the variability model corresponding to CNF_{1IC} .

In Step 2-3, if the SAT/UNSAT result of CNF_{1IC} is UNSAT, there is contradiction in the CNF_{1IC} associated with the conflicting SVM components in the variability model. So, the ICL method analyzes the differences in the truth values of CNF_{1IC} and CNF_{EC} in the evaluation results to analyze contradiction associated with conflicting SVM components in the variability model and generates a list of combinations. In this case, when applying the same combination of truth values of all variables, the truth value of CNF_{1IC} is False and the truth value of CNF_{EC} is True only. Thus, each element of the list of combinations consists of a combination of truth values of all variables, when the truth value of CNF_{1IC} is False and the truth value of CNF_{EC} is True. If the SAT/UNSAT result of CNF_{1IC} is SAT in Step 2-2, the corresponding variability model is a consistent model. Thus, the ICL method is terminated at the Inconsistency Analysis phase.

The Contradiction Analysis Result reflects whether the SVM variability model is consistent or inconsistent as a result of the Inconsistency Analysis phase.

The following is an application example for Step 2 that detects Contradiction in CNF_{1IC} obtained from Step 1 associated with the conflicting SVM components in the variability model.

Example 4.3. (Step 2. Analyze Contradiction in CNF_{1IC}).

Input 1: $CNF_{1IC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C) \wedge (\neg B \vee \neg C)$

Input 2: $CNF_{1EC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C)$

Step (2-1) This step evaluates all combinations of truth values of all variables in CNF_{1IC} and CNF_{1EC} formulas to generate evaluation results of CNF_{1IC} and CNF_{1EC} . There are eight combinations of the truth values of the three variables including A, B, and C. The evaluation results are generated by evaluating the eight combinations. The evaluation results for CNF_1 are shown in columns 4 and 5 of Table 3.

Table 3. The SAT/UNSAT result of CNF_1 .

Variables			CNF1	
A	B	C	E [CNF_{1IC}]	E [CNF_{1EC}]
True	True	True	False	True
True	True	False	False	False
True	False	True	False	False
True	False	False	False	False
False	True	True	False	False
False	True	False	False	False
False	False	True	False	False
False	False	False	False	False
SAT/UNSAT Result			UNSAT	SAT

Step (2-2) In this substep, the SAT/UNSAT results of CNFs are analyzed using the evaluation results of CNF_{1IC} and CNF_{1EC} . Table 3 shows the SAT/UNSAT result for CNF_1 . In Table 3, the CNF_{1IC} becomes UNSAT because a combination of the truth values of all variables where the truth value of CNF_{1IC} is true does not exist.

Step (2-3) This step analyzes the differences between the evaluation results of $CNF1_{IC}$ and $CNF1_{EC}$ when the SAT/UNSAT result of $CNF1_{IC}$ is UNSAT. There is only one case where there is a difference in the truth values of $E[CNF1_{IC}]$ and $E[CNF1_{EC}]$ in evaluation results, where the list of combinations is as follows:

Output 1: List of Combinations

$$\langle A = \text{True}, B = \text{True}, C = \text{True} \rangle, CNF1_{IC} = \text{False}, CNF1_{EC} = \text{True} \rangle$$

This list is used to analyze the conflicting literal clauses in the $CNF1_{IC}$ associated with conflicting SVM components in the variability model in Step 3.

The following is an application example for Step 2 that detects contradiction in $CNF2_{IC}$ obtained from Step 1 associated with the conflicting SVM components in the variability model.

Example 4.4. (Step 2. Analyze Contradiction in $CNF2_{IC}$).

$$\text{Input 1: } CNF2_{IC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C) \wedge (\neg B \vee C)$$

$$\text{Input 2: } CNF2_{EC} = (A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C)$$

Step (2-1) This step evaluates all combinations of truth values of all variables in $CNF2_{IC}$ and $CNF2_{EC}$ formulas to generate evaluation results of $CNF2_{IC}$ and $CNF2_{EC}$. There are eight combinations of the truth values of the three variables including A, B, and C. The evaluation results are generated by evaluating the eight combinations. The evaluation results for CNF2 are shown in columns 4 and 5 of Table 4.

Table 4. The SAT/UNSAT result of CNF2.

Variables			CNF2	
A	B	C	E [$CNF2_{IC}$]	E [$CNF2_{EC}$]
True	True	True	True	True
True	True	False	False	False
True	False	True	False	False
True	False	False	False	False
False	True	True	False	False
False	True	False	False	False
False	False	True	False	False
False	False	False	False	False
SAT/UNSAT Result			SAT	SAT

Step (2-2) In this substep, the SAT/UNSAT results of CNFs are analyzed using the evaluation results of $CNF2_{IC}$ and $CNF2_{EC}$. Table 4 shows the SAT/UNSAT result for CNF2. In Table 4, the $CNF2_{IC}$ becomes SAT because there exists a combination of the truth values of all variables where the truth value of $CNF2_{IC}$ is true.

Step (2-3) This step analyzes the differences between the evaluation results of $CNF2_{IC}$ and $CNF2_{EC}$ when the SAT/UNSAT result of $CNF2_{IC}$ is UNSAT. However, the ICL method does not perform this step because the SAT/UNSAT result of $CNF2_{IC}$ is SAT. Thus, the ICL method determines that this variability model is a consistent model, and it terminates in Step 2.

4.2.3. Step 3: Detect Conflicting Literal Clauses in CNF_{IC}

In Step 3, the ICL method identifies conflicting literal clauses in CNF_{IC} associated with conflicting SVM components in the variability model using the list of combinations and CNF_{IC}^* . The CNF_{IC}^* is simplified by evaluating the truth value of each variable in the list of combinations according to the rules set forth in [34,35]. If there is contradiction between two or more literal clauses in CNF_{IC}^* , it is no longer simplified. This CNF_{IC}^* is called the Simplified CNF_{IC} (S_CNF_{IC}). Because each literal in CNF_{IC}^* contains a unique subscript index, it is possible to track the clauses of S_CNF_{IC} that correspond to an associated clause in CNF_{IC}^* .

In this step, the truth values of variables in combination are evaluated in the order set forth [34,35]. According to these rules, the truth value of the variable in the clause that has smallest number of literals among all clauses in CNF_{IC}^* is the first to be evaluated. In other words, the truth value of a variable in the literal clause is evaluated first. In the event that multiple clauses with the same number of literals exist, a branch occurs. After one S_CNF_{IC} is generated by evaluating the truth value of one of the variables at the point where the branch occurs, another S_CNF_{IC} is generated by evaluating the truth value of another variable. Thus, S_CNF_{IC} relies on the order of evaluating the true values of the variables. If there are clauses that have the same number of literals, the proposed method first evaluates the leftmost clause to CNF_{IC}^* ,

In the case of the list of combinations for CNF_{IC} (Example 4.3), since the clause of (A_1) has the smallest number of literals in CNF_{IC}^* , the truth value of variable A is evaluated first. Subsequently, because the clauses of (B_1) and (C_1) have an identical number of literals, branch occurs. Accordingly, it is possible to evaluate $C = \text{True}$ after determining that $B = \text{True}$ or evaluate $B = \text{True}$ only after determining that $C = \text{True}$.

The clauses in S_CNF_{IC} are the conflicting literal clauses in CNF_{IC} associated with the conflicting SVM components in the variability model. The proposed method generates a list of conflicting literal clauses by collecting all S_CNF_{IC} s.

The following is an application example for Step 3 that detects conflicting literal clauses in CNF_{IC} .

Example 4.5. (Step 3. Detect Conflicting Literal Clauses in CNF_{IC}).

Input 1: $CNF_{IC}^* = (A_1) \wedge (\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2) \wedge (\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2) \wedge (\neg B_3 \vee \neg C_3)$

Input 2: List of Combinations

$\langle \{A = \text{True}, B = \text{True}, C = \text{True}\}, CNF_{IC} = \text{False}, CNF_{EC} = \text{True} \rangle$

Steps (3) In Example 4.3, CNF_{IC} is UNSAT, and the only difference between the evaluation results of CNF_{IC} and CNF_{EC} is $(A = \text{True}, B = \text{True}, \text{ and } C = \text{True})$. In this case, the detection of conflicting literal clauses in CNF_{IC} is carried out as follows:

When “ $A = \text{True}$ ” is assigned to CNF_{IC}^* according to the given rules, the CNF_{IC}^* is simplified as below.

$$CNF_{IC}^* = B_1 \wedge C_1 \wedge (\neg B_3 \vee \neg C_3)$$

And then, because each of B_1 and C_1 has literal clauses, branching occurs.

Branch 1:

“ $B = \text{True}$ ” is assigned first according to the given rule

$$CNF_{IC}^* = C_1 \wedge \neg C_3$$

And then, “C = True” is assigned to the $CNF1_{IC}^*$, the truth value of $CNF1_{IC}^*$ becomes “False” as below.

$$CNF1_{IC}^* = \neg True_3$$

That is, “ $CNF1_{IC}^* = (C_1) \wedge (\neg C_3)$ ” is S_CNF1_{IC} . Thus, when “A = True, B = True, and C = True” are assigned to the $CNF1_{IC}^*$, $\{<C_1: \neg C_3>\}$ is one of the conflicting literal clauses in $CNF1_{IC}$ associated with conflicting SVM components in the variability model.

Branch 2:

Next, “C = True” is assigned first according to a given rule.

$$CNF1_{IC}^* = B_1 \wedge \neg B_3$$

And then, “B = True” is assigned to the $CNF1_{IC}^*$, the truth value of $CNF1_{IC}^*$ becomes “False” as below.

$$CNF1_{IC}^* = \neg True_3$$

That is, “ $CNF1_{IC}^* = (B_1) \wedge (\neg B_3)$ ” is S_CNF1_{IC} . Thus, when the order of “A = True, C = True, and B = True” is assigned to the $CNF1_{IC}^*$, $\{<B_1: \neg B_3>\}$ is one of the conflicting literal clauses in $CNF1_{IC}$ associated with conflicting SVM components in variability model.

Output 1: List of Conflicting Literal Clauses

$$\{<C_1: \neg C_3>, <B_1: \neg B_3>\}$$

4.2.4. Step 4: Eliminate Duplicated Conflicting Literal Clauses

Step 4 identifies and eliminates duplicated conflicting literal clauses among all conflicting literal clauses in $CNF1_{IC}^*$ detected in Step 3. The identification method compares the conflicting literal clauses in the list of conflicting literal clauses, and if there are identical conflicting literal clauses, it keeps only one of them, removing the rest. By doing so, the ICL method generates a list of conflicting literal clauses without duplications.

The following is an application example for Step 4.

Example 4.6. (Step 4. Eliminate Duplicated Conflicting Literal Clauses).

Input: List of conflicting literal clauses

$$\{<C_1: \neg C_3>, <B_1: \neg B_3>\}$$

Output: List of conflicting literal clauses without duplications

$$\{<C_1: \neg C_3>, <B_1: \neg B_3>\}$$

4.2.5. Step 5: Track Conflicting SVM Components in the SVM Variability Model

Step 5 tracks which SVM elements of the SVM variability model are affected by the analysis results of the $CNF1_{IC}^*$. In other words, the conflicting SVM components responsible for the inconsistency in the SVM variability model are tracked.

This step consists of the following two substeps:

Step (5-1) This substep tracks which clauses in $CNF1_{IC}^*$ include each literal clause in the list of conflicting literal clauses without duplications. Because $CNF1_{IC}^*$ has a unique subscript index of each literal, tracking is easy.

Step (5-2) This substep tracks which SVM elements are mapped to the clauses that were tracked in Step 5-1.

These SVM elements are responsible for creating a variability model with inconsistency.

The following is an application example for Step 5 that tracks conflicting SVM components in a variability model in Figure 3b

Example 4.7. (Step 5. Track Conflicting SVM Components in a Variability Model in Figure 3b).

$$\text{Input 1: } \text{CNF1}_{\text{IC}}^* = \{(A_1)\} \wedge \{(\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2)\} \wedge \{(\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2)\} \\ \wedge \{(\neg B_3 \vee \neg C_3)\}$$

Input 2: List of conflicting literal clauses without duplications

Input 3: SVM elements in Figure 6

Step (5-1) This substep tracks which clauses in $\text{CNF1}_{\text{IC}}^*$ include each literal clause in the list of conflicting literal clauses without duplications.

The tracking results of $\langle C_1: \neg C_3 \rangle$ are underlined as follows:

$$\text{CNF1}_{\text{IC}}^* = \{(A_1)\} \wedge \{(\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2)\} \wedge \{(\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2)\} \wedge \{(\neg B_3 \vee \neg C_3)\}$$

The tracking results of $\langle B_1: \neg B_3 \rangle$ are underlined as follows:

$$\text{CNF1}_{\text{IC}}^* = \{(A_1)\} \wedge \{(\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2)\} \wedge \{(\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2)\} \wedge \{(\neg B_3 \vee \neg C_3)\}$$

Step (5-2) This substep tracks which SVM elements in Figure 6 are mapped to the clauses that tracked in Step 5-1.

Now, the tracking results of the SVM elements in Figure 6 are as follows:

As shown in Figure 6, $(\neg A_4 \vee C_1)$ is included in the component of $(\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2)$, and $(\neg B_3 \vee \neg C_3)$ is included in the component of $(\neg B_3 \vee \neg C_3)$.

In addition, $(\neg A_2 \vee B_1)$ is included in the component of $(\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2)$, and $(\neg B_3 \vee \neg C_3)$ is included in the component of $(\neg B_3 \vee \neg C_3)$. The components of $(\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2)$, $(\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2)$, and $(\neg B_3 \vee \neg C_3)$ are mapped to the corresponding SVM elements.

Output 1: List of causes

Cause 1:

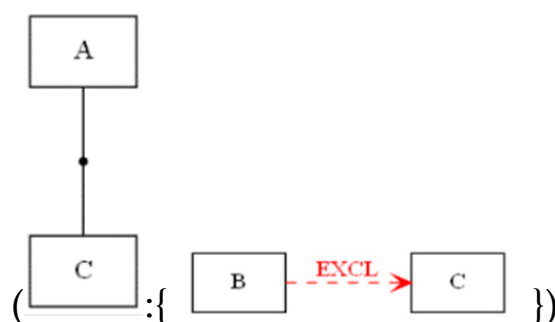
$$(I\langle A, C \rangle: \{E\langle B, C \rangle\})$$

Cause 2:

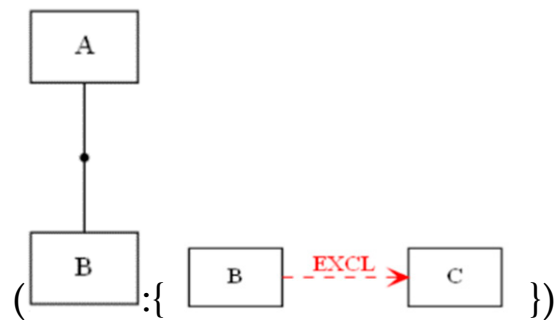
$$(I\langle A, B \rangle: \{E\langle B, C \rangle\})$$

If the graphical notation in Table 1 is used, Cause 1 and Cause 2 are expressed as follows:

Cause 1:



Cause 2:



In Figure 3b, because A includes B and A includes C, B and C must be included in all products. However, C cannot be included in a product if B is included due to the constraint that B excludes C. Cause 1 represents a conflict between the inclusion requirement of C by the inclusion relationship and the exclusion requirement of C by the excludes constraint. On the other hand, Cause 2 represents a conflict between the inclusion requirement of B by the inclusion relationship and the exclusion requirement of B by the excludes constraint.

Whether the representation of causes located by the ICL method is adequate or not is discussed in Section 5.5.

5. Evaluation

To evaluate the ICL method, we implemented it as an ICL tool and conducted two experiments. The objective of the evaluation is to assess whether the ICL method accurately locates all causes of inconsistency in a variability model. Our first experiment is conducted on the 24 variability models in [37–39]. Although the works of [37–39] do not propose approaches analyzing variability models or locating the causes of inconsistency in a variability model, they provide a rich collection of datasets. The second experiment is conducted on variability models provided by the SPLOT repository (hereinafter referred to as real-world variability models) larger than those used in the first experiment. The following subsections are organized as follows. Section 5.1 introduces research questions for the experiments. Section 5.2 describes the implementation of the ICL method. In Sections 5.3 and 5.4, the two experiments are conducted. Finally, Sections 5.5 and 5.6 discuss the results of the experiments. All the models used in the experiments, including the derived models, and the ICL tool developed in this paper (with its source code) have been posted at <https://url.kr/fnk7uq> (accessed on 9 October 2025).

5.1. Research Questions

The research questions for evaluating the ICL method are formulated as follows.

(RQ1) Does the proposed method accurately determine whether a variability model is consistent or not?

(RQ2) For an inconsistent model, does the proposed method accurately locate all causes of inconsistency in a variability model?

RQ1 and RQ2 can be further broken down into the following questions:

(RQ1a) Does the proposed method erroneously ever tell that a consistent model is an inconsistent model?

(RQ1b) Does the proposed method erroneously ever tell that an inconsistent model is a consistent model?

(RQ2a) (Completeness) For an inconsistent model, does the proposed method find out all the causes of inconsistency without missing any true cause?

(RQ2b) (Soundness) For an inconsistent model, does the proposed method ever report a false cause of inconsistency?

Because there is a dependency between causes, solving one cause can affect another. For this reason, the number of located causes may vary depending on how the cause is located, and other causes may be removed simultaneously when one cause is removed. Therefore, making the ICL tool find one cause would make it efficient and convenient for the users by guiding them to analyze and fix causes one by one.

Therefore, instead of asking “Does the proposed method find all causes of inconsistency?” because “all causes” are not fixed, we can confirm that the proposed method does not miss the cause of inconsistency by asking “Does the proposed method find at least one cause of inconsistency if there is a cause of inconsistency in an inconsistent model?”. Thus, to answer (RQ2a), we replace (RQ2a)’s answer by answering (RQ2a’).

(RQ2a’) (Completeness) Does the proposed method find at least one cause of inconsistency if there is a cause of inconsistency in an inconsistent model?

If the answer to RQ1b is No, then the answer to RQ2a’ is Yes because the proposed method does not answer as a consistent model if there is a cause of inconsistency in an inconsistent model. For the same reason, if the answer to RQ1b is Yes, then the answer to RQ2a’ is No.

5.2. The ICL Tool

The ICL tool automated the entire process of the ICL method. Figure 7 shows the architecture of the ICL tool, which consisted of four components: (1) Converter Component, (2) Contradiction Analysis Component, (3) Traceability Component, and (4) Cause Detection of Inconsistency Component. The input to the ICL tool was an SVM variability model, and the outputs from the tool were a contradiction analysis result (Output 1) and a list of causes (Output 2).

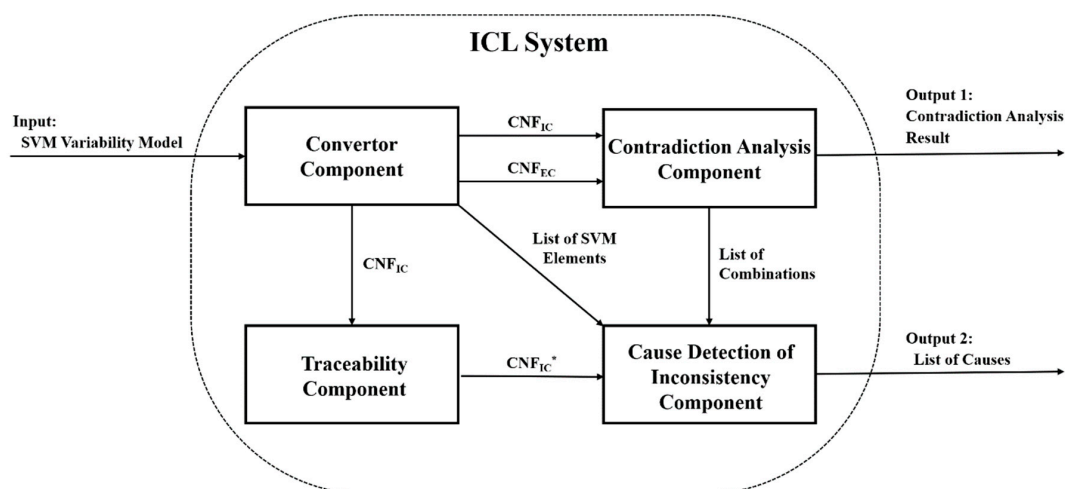


Figure 7. Architecture of ICL tool.

The Converter Component and Traceability Component (Figure 7) performed Step 1 of the ICL method together. The Contradiction Analysis Component performed Step 2. This component was implemented by extending the SAT Solver-based DPLL algorithm [34,35] to accurately detect all contradictions associated with the inconsistency of a variability model in CNF_{IC} . The DPLL algorithm is capable of efficiently finding the combinations of truth values associated with the variables of a CNF formula that make it True. In our ICL tool, the DPLL algorithm was extended to find combinations of truth values of the variables that rendered the CNF_{IC} formula False while rendering the CNF_{EC} formula True. In the event of an inconsistency in an SVM variability model, the result from the Contradiction Analysis Component was “Inconsistent”, otherwise it was “Consistent”.

The Cause Detection of Inconsistency Component performed inconsistency cause locating (Steps 3~5). This component outputted a list of causes containing any identified causes of inconsistency within an SVM variability model. But the list of causes produced by the ICL tool may not contain all the causes produced by the ICL method, as a dependent relationship between the causes exists. If one cause is resolved, other causes may be resolved with it; otherwise, it means that there exists no dependent relationship between the causes. If there exists a dependent relationship between causes, the ICL method locates all dependent causes, while the ICL tool locates only one of all dependent causes.

In Example 4.7, the ICL method located both Cause 1: ($I < A, C > : \{E < B, C > \}$) and Cause 2: ($I < A, B > : \{E < B, C > \}$). But, because there exists a dependent relationship between the two causes, the ICL tool locates only one of them. Because from the nodes of a variability model that are identified as the conflicting SVM components in a cause of inconsistency located by the ICL tool, it is possible to find other SVM elements that participate in creating the inconsistency by navigating towards the root of the model from the conflicting SVM components. Thus, the list of causes located by the ICL tool is sufficient to fix the inconsistency in the variability model.

It is not difficult to implement the ICL tool so that it can report all causes it finds. However, even if it reports all causes it finds, it is not guaranteed that the reported causes are the final ones in the sense that they are independent from each other and by removing all of them all inconsistencies are gone at once. Since there can be dependencies between causes in a variability model, solving one cause can introduce or remove inconsistencies in other locations. Therefore, making the ICL tool find one cause would make it efficient and convenient for the users by guiding them to analyze and fix causes one by one.

5.3. Experiment 1: Applying to Experimental Variability Models

5.3.1. Subject Models

Experiment 1 used 24 out of the 192 experimental variability models presented in Segura et al. [37–39] (the 24 models are described in detail on pages 18 to 21 of [38]), which were especially developed for their inconsistency analysis experiment. These models were presented with seven analysis perspectives (as shown in the left two columns of Table 5). Each model was systematically developed through the design, evaluation, and refinement process described in [37–39]. The 192 models, including 24 models used in an Inconsistency Analysis Perspective experiment, as well as their results, are shown in the right two columns of Table 5. For Experiment 1, we used these 24 models, of which 20 are consistent and 4 are inconsistent.

Table 5. Models and analysis results from [37–39].

Analysis Perspectives	The Number of Models	24 Types of Models	Analysis Results (from [37–39])
Inconsistency Analysis Perspective	24	VM-1, VM-2, VM-3, VM-4, VM-5, VM-6, VM-7, VM-8, VM-9, VM-10, VM-12, VM-13, VM-14, VM-15, VM-16, VM-17, VM-18, VM-19, VM-20, VM-22	Consistent
		VM-11, VM-21, VM-23, VM-24	Inconsistent
Other Six Perspectives ²	168		
Total	192		

² The other six analysis perspectives are Valid Product, All Products, Number of Products, Variability, Commonality, and Dead Features.

5.3.2. Experimental Design

To answer RQ1a, the Inconsistency Analysis phase of the ICL tool is carried out to see if the ICL method determines each of the 20 consistent models to be consistent in agreement with the analysis results in the second row of Table 5.

To answer RQ1b, the Inconsistency Analysis phase of the ICL tool is carried out to see if the ICL method determines each of the four inconsistent models to be inconsistent in agreement with the analysis results in the third row of Table 5.

To answer RQ2a', we will revert to the answer provided in response to RQ1b, as earlier explained in Section 5.1.

To answer RQ2b, the Inconsistency Cause Locating phase of the ICL tool is carried out to see if the ICL method does not produce false causes of inconsistency for the four inconsistent models. We will manually check each cause for whether it is a false cause or not.

5.3.3. Experiment Results

The results of our Inconsistency Analysis of Experiment 1 for RQ1a and RQ1b are provided in Table 6.

Table 6. Results of the inconsistency analysis phase of the ICL tool in Experiment 1.

24 Types of Models	The Number of Models	Inconsistency Analysis Results (from the ICL Tool)
VM-1, VM-2, VM-3, VM-4, VM-5, VM-6, VM-7, VM-8, VM-9, VM-10, VM-12, VM-13, VM-14, VM-15, VM-16, VM-17, VM-18, VM-19, VM-20, VM-22	20	Consistent
VM-11, VM-21, VM-23, VM-24	4	Inconsistent

The answer to RQ1a was No. As shown in the second row of Table 6, the ICL tool determined that each of the 20 models was consistent, matching the previously determined results listed in the second row of Table 5.

The answer to RQ1b was No. As shown in the third row of Table 6, the ICL tool determined that each of the four models was inconsistent, matching the previously determined results listed in the third row of Table 5.

The answer to RQ2a' is Yes because the answer to RQ1b was No. The reason is that in Experiment 1, the ICL tool did not answer consistent model if there is a cause of inconsistency in an inconsistent model.

The results of our Inconsistency Cause Locating phase of Experiment 1 for RQ2b are listed in Table 7. VM-11, VM-21, VM-23, and VM-24 (Table 6) were all inconsistent models, and Inconsistency Cause Locating was therefore conducted on them.

Table 7. Results of inconsistency cause locating phase of the ICL tool in Experiment 1.

Inconsistent Models	Located Causes (from the ICL Tool)
VM-11	(E <B,C>:{I <A,C>})
VM-21	(R <E,D>:{A <vpB,{C,D}>})
VM-23	(E <E,B>:{A <vpB,{C,D}>})
VM-24	(R <H,G>:{A <vpB,{C,D,E,F,G}>})

The answer to RQ2b was No because the causes located by the ICL tool in Table 7 were all true causes.

The located cause $(E \langle B, C \rangle : \{ I \langle A, C \rangle \})$ for VM-11 is a true cause because the inclusion of C required by the inclusion relationship and the exclusion of C required by the excludes constraint conflict each other, which we could manually check from VM-11.

The located cause $(R \langle E, D \rangle : \{ A \langle \text{vp}B, \{C, D\} \rangle \})$ for VM-21 is a true cause because the inclusion of D required by the requires constraint and the exclusion of D required by the alternative relationship conflict each other. The case of VM-24 is similar to VM-21.

Finally, the located cause $(E \langle E, B \rangle : \{ A \langle \text{vp}B, \{C, D\} \rangle \})$ for VM-23 is a true cause because the inclusion of B required by the alternative relationship and the exclusion of B required by the excludes constraint conflict each other. Specifically, in the VM-23 model, node A is a root node. If node A is included in the products, node E must be included by the inclusion relationship between nodes A and E. If node E is included in the products, node B must not be included by the exclude constraint between nodes E and B. On the other hand, if node E is included in the products, node B must be included by the alternative relationship between nodes B and D. So, the inclusion of B required by the alternative relationship and the exclusion of B required by the excludes constraint conflict with each other.

5.4. Experiment 2: Applying to Real-World Variability Models

5.4.1. Subject Models

Experiment 2 targeted the 1146 real-world variability models that are provided on the SPLOT-Software Product Line Online Tools website [9] as of 17 October 2019. For analysis subjects, we examined the size (i.e., the number of features) and CTCR (Cross-Tree Constraint Ratio) (i.e., the ratio of the number of features in the constraints to the number of features in a feature model [24]) and finally selected five consistent models, which are the Alarm System, Smart Hotel, Mobile Media_Lenita, Smart Home, and Automotive System models. To create inconsistent models as well as consistent models that consider various scenarios, we generate four derived models by adding constraints to each selected model. Therefore, altogether 25 models (5 original models and 20 derived models) were selected as subjects of Experiment 2 (Table 8). Detailed information on the subject models can be found in Appendix A.

Table 8. Results of inconsistency analysis phase of the ICL tool in Experiment 2.

Model Name	Subject Models	Inconsistency Analysis Results from the SPLOT Tool	Inconsistency Analysis Results from the ICL Tool
Alarm System	Alarm System_Original	Consistent	Consistent
	Alarm System_D1	Consistent	Consistent
	Alarm System_D2	Consistent	Consistent
	Alarm System_D3	Inconsistent	Inconsistent
	Alarm System_D4	Inconsistent	Inconsistent
Smart Hotel	Smart Hotel_Original	Consistent	Consistent
	Smart Hotel_D1	Consistent	Consistent
	Smart Hotel_D2	Inconsistent	Inconsistent
	Smart Hotel_D3	Consistent	Consistent
	Smart Hotel_D4	Inconsistent	Inconsistent

Table 8. Cont.

Model Name	Subject Models	Inconsistency Analysis Results from the SPLOT Tool	Inconsistency Analysis Results from the ICL Tool
Mobile Media_Lenita	Mobile Media_Lenita_Original	Consistent	Consistent
	Mobile Media_Lenita_D1	Inconsistent	Inconsistent
	Mobile Media_Lenita_D2	Consistent	Consistent
	Mobile Media_Lenita_D3	Consistent	Consistent
	Mobile Media_Lenita_D4	Consistent	Consistent
Smart Home	Smart Home_Original	Consistent	Consistent
	Smart Home_D1	Inconsistent	Inconsistent
	Smart Home_D2	Consistent	Consistent
	Smart Home_D3	Inconsistent	Inconsistent
	Smart Home_D4	Consistent	Consistent
Automotive System	Automotive System_Original	Consistent	Consistent
	Automotive System_D1	Inconsistent	Inconsistent
	Automotive System_D2	Consistent	Consistent
	Automotive System_D3	Consistent	Consistent
	Automotive System_D4	Inconsistent	Inconsistent

The selection criteria are as follows, and detailed information on the subject models can be found in Appendix A.

- (1) Selection Criterion 1. All models in the repository were analyzed in terms of feature size and CTCR (Cross-Tree Constraint Ratio), and models were selected in the area with the most distribution.
- (2) Selection Criterion 2. Very small models and incomplete models were excluded from the experiment. Complete models were selected such that the information of the organization that developed them and their developers were clearly specified, and models with ten or less features and incomplete models whose feature names are arbitrarily given, for example, A, B, and C, were excluded from the experiment.
- (3) Selection Criterion 3. If the number of features is the same, then variability models with a higher CTCR were selected as subjects for the experiment. Since inconsistency is caused by conflicts of relationships and constraints between components in the variability model, there is a high probability of inconsistency in the variability model with a high CTCR. Inconsistency does not occur in the variability models with 0% CTCR.
- (4) Selection Criterion 4. Since the feature size can be large, it affects performance, so a variability model with a large number of features in the SPLOT repository was included in the experiment. In the process of converting a variability model into a CNF formula, even if the number of constraints increases, the number of literals in the CNF formula does not necessarily increase. However, when the number of features increases, the number of literals increases proportionally, affecting performance.

No preexisting report of analysis results for the 25 variability models existed. Accordingly, to perform the Inconsistency Analysis of Experiment 2 for RQ1a and RQ1b, we first generated analysis results by performing the Inconsistency Analysis with the SPLOT tool.

A total of 16 models were found to be consistent, while 9 were found to be inconsistent (as shown in the third column of Table 8).

5.4.2. Experimental Design

To answer RQ1a, the Inconsistency Analysis phase of the ICL tool is carried out to see if the ICL method determines each of the 16 consistent models to be consistent in agreement with the analysis results in the third column of Table 8.

To answer RQ1b, the Inconsistency Analysis phase of the ICL tool is carried out to see if the ICL method determines each of the nine inconsistent models to be inconsistent in agreement with the analysis results in the third column of Table 8.

To answer RQ2a', we will revert to the answer provided in response to RQ1b, as earlier explained in Section 5.1.

To answer RQ2b, the Inconsistency Cause Locating phase of the ICL tool is carried out to see if the ICL method does not produce false causes of inconsistency for nine inconsistent models. We will manually check each cause for whether it is a false cause or not.

5.4.3. Experiment Results

From the Inconsistency Analysis of Experiment 2 for RQ1a and RQ1b, the results obtained are presented in the fourth column of Table 8.

The answer to RQ1a was No. As shown in the fourth column of Table 8, the ICL tool determined each of the 16 models marked "Consistent" in the third column of Table 8 to be consistent.

The answer to RQ1b was No. As shown in fourth column of Table 8, the ICL tool determined each of the nine models marked "Inconsistent" in the third column of Table 8 to be inconsistent.

The answer to RQ2a' was Yes because the answer to RQ1b was No. The reason is that in Experiment 2, the ICL tool did not answer as a consistent model if there is a cause of inconsistency in an inconsistent model.

Table 9 shows the results of the Inconsistency Cause Locating phase of Experiment 2 applying the ICL tool to the nine inconsistent models in Table 8. The results in the fourth column (for detected causes) of Table 9 answer RQ2b.

The answer to RQ2b was No because the located causes in the ICL tool in fourth column of Table 9 were all true causes.

The located cause (R <Siren, Warning_Light>:{A <vpAlarm_System, {Siren, Warning_Light}>}) for the Alarm_System_D3 variability model was a true cause because the inclusion of the Warning_Light node required by the requires constraint (Figure A2 in Appendix B), which was indicated by "R <Siren, Warning_Light>" in the located cause, conflicted with the exclusion of the Warning_Light node required by an alternative relationship, which was indicated by "A <vpAlarm_System, {Siren, Warning_Light}>" in the located cause. This was manually checked in the Alarm_System_D3 variability model. Similar results are found for the located causes for Smart_Hotel_D2 and Mobile_Media_Lenita_D1.

The located cause (R <Engine, Roof_Control_with_Rain_Sensor>:{A <vpRoof_Control, {Manual_Roof_Control, Roof_Control_with_Rain_Sensor}>, R <Roof_Control_with_Rain_Sensor, Rain_Sensor>}) for the Automotive_System_D4 variability model was a true cause because the inclusion of the Roof_Control_with_Rain_Sensor node required by a requires constraint (Figure A3 in Appendix B), which was indicated by "R <Engine, Roof_Control_with_Rain_Sensor>" in the located cause, conflicted with (1) the inclusion of the Roof_Control_with_Rain_Sensor node required by a requires constraint, which was indicated by "R <Roof_Control_with_Rain_Sensor, Rain_Sensor>" in the located cause and (2) the exclusion of the Roof_Control_with_Rain_Sensor node required by an alterna-

tive relationship, which was indicated by “A <vpRoof_Control, {Manual_Roof_Control, Roof_Control_with_Rain_Sensor}>” in the located cause. This was manually checked in the Automotive System_D4 variability model. Similarly, the located causes for Smart Hotel_D4, Smart Home_D1, Smart Home_D3, and Automotive System_D1 were true causes.

Table 9. Results of inconsistency cause locating phase of the ICL tool in Experiment 2.

Model	Size of Model		Located Causes	Time (ms ³)
	# Features	CTCR		
Alarm System_D3	13	46%	(R <Siren, Warning_Light>: {A <vpAlarm_System, {Siren, Warning_Light}>})	0.99
Alarm System_D4	13	62%	(E <Online_Access, User_Interface>: {I <Alarm_System, User_Interface>}) (R <Siren, Warning_Light>: {A <vpAlarm_System, {Siren, Warning_Light}>})	2.02
Smart Hotel_D2	13	38%	(R <SilentAlarm, VisualAlarm>: {A <vpAlarm, {SilentAlarm, Siren, VisualAlarm}>})	1.01
Smart Hotel_D4	13	46%	(R <SilentAlarm, VisualAlarm>: {A <vpAlarm,{SilentAlarm, Siren, VisualAlarm}>, R <PipedMusic, SilentAlarm>})	1.00
Mobile Media_Lenita_D1	15	40%	(E <MediaManagement, ScreenSize>: {I <Mobile_Media_Lenita, ScreenSize>, A <vpScreenSize,{Screen1}>})	1.00
Smart Home_D1	22	27%	(E <Lighting, ControlSystem>: {I <SmartHome, ControlSystem>, C <vpControlSystem, {CellPhone, ControlPanel}>})	1.01
Smart Home_D3	22	41%	(E <Lighting, MoviePlayers>: {C <vpMoviePlayers, {HDTV42, PCPlays, HDTV32}>, R <ControlSystem, MoviePlayers>})	2.01
Automotive System_D1	31	48%	(E <Chassis, Energy_Reservoir>: {I <Automated_System, Energy_Reservoir>, C <vpEnergyReservoir, {Accumulator, Gasline_Tank}>})	6.02
Automotive System_D4	31	58%	(R <Engine, Roof_Control_with_Rain_Sensor>: {A <vpRoof_Control, {Manual_Roof_Control, Roof_Control_with_Rain_Sensor}>, R <Roof_Control_with_Rain_Sensor, Rain_Sensor>})	6.01

³ Milliseconds.

The located cause (E <Online_Access, User_Interface>:{I <Alarm_System, User_Interface>}) for the Alarm System_D4 variability model was a true cause because the inclusion of the User_Interface node required by an inclusion relationship (Figure A4 in Appendix B), which was indicated by “I <Alarm_System, User_Interface>” in the located cause, conflicted with the exclusion of the User_Interface node required by an excludes constraint, which was indicated by “E <Online_Access, User_Interface>” in the located cause. This was manually checked in the Alarm System_D4 variability model. Similarly, the located cause (R <Siren, Warning_Light>:{A <vpAlarm_System, {Siren, Warning_Light}>}) for Alarm System_D4 was a true cause.

5.5. Adequacy of the Located Cause of Inconsistency

Each node in an SVM variability model belongs to one or more SVM elements that constitute the SVM variability model. A node may thus be affected by more than one pair of SVM elements when the pairs are conflicting SVM components. Figure 8a,b shows variability models that include the inconsistent model of Figure 3b as a subtree.

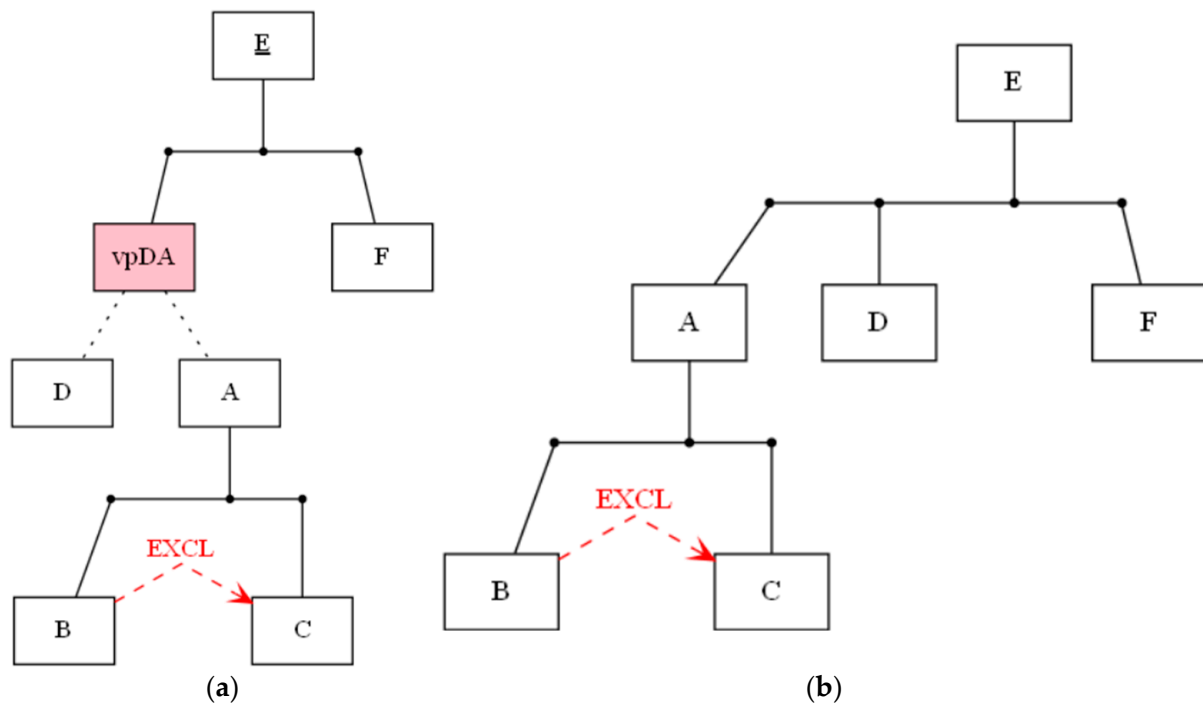


Figure 8. Variability models including the variability model of Figure 3b as a subtree. (a) consistent model; (b) inconsistent model.

In the case of Figure 8a, the ancestors of (the root of) the subtree do not affect inconsistency. Due to inclusion relationships and exclude constraint, nodes A, B, and C are always dead features, so if the variability model is instantiated, nodes A, B, and C can be not included in the product, but node D can be included. Therefore, Figure 8a is a consistent model that can be instantiated to {E, D, F}.

In contrast, in the case of Figure 8b, the ancestor of (the root of) the subtree affected inconsistency because, when the variability model was instantiated, node A must have been included by the inclusion relationship between root node E and node A. Therefore, Figure 8b is an inconsistent model that can be instantiated to {NULL}.

Inconsistency occurs because, when a variability model is instantiated, a specific node is simultaneously required and excluded by more than one pairs of SVM elements. In Figure 8b, node C is required by the inclusion relationship between node A and C and excluded by the excludes constraint between node B and C simultaneously.

As we explained in Section 3.3, a complete description of a cause of inconsistency is necessary to show the conflicting SVM components and also all the SVM elements that participate in the creation of a conflict. A complete description of a cause of inconsistency is stated as follows:

(Located Cause, Contributing SVM elements)

where *Located Cause* is a point where the inconsistency occurred and *Contributing SVM elements* is a set of the SVM elements that contribute to the *Located Cause* of inconsistency. The notation for *Located Cause* was defined in Section 3.3 as the notation for a located cause of inconsistency. So, for example, complete descriptions of the causes of inconsistency for the inconsistent model in Figure 8b are stated as:

Cause 1: ((I<A, C>:{E<B, C>}), {I<E, A>})

Cause 2: ((I<A, B>:{E<B, C>}), {I<E, A>})

However, navigating from the conflicting SVM components towards the root of the model allows for the identification of other SVM elements that participate in creating the inconsistency. Developers that are informed of the located causes can fix therefore the inconsistency based on the conflicting SVM components described. For that reason, our ICL method was designed to provide the located cause of the inconsistency rather than a complete description of its cause.

Since the target models are different, it is difficult to directly compare our method and explanation-based approaches. If we assume that they are applied to their own target model, i.e., SVM models and feature models, respectively, then indirect comparison is possible and reveals the following differences.

In terms of precision, our method identifies only the exact location where inconsistency has occurred, as shown in Tables 9 and 10, while the explanation-based approach provides a wide range of descriptions from root feature related to inconsistency to child features related to inconsistency, as illustrated in Table 3 of [16]. In the case of the explanation-based approach, as the size of the feature model increases, the description becomes longer and more complex. Moreover, the user has to analyze the description in detail to precisely locate the root cause. Therefore, with our method, the user can quickly and efficiently fix inconsistency by directly knowing the exact location where the inconsistency exists.

Table 10. Results of inconsistency cause locating.

Model ID	Size of Model		Detected Causes	Time (Milliseconds)
	# Features	CTCR		
Banking Software_D1	176	2%	(E <OpenAccount, ATMLogin>: {I <CoreBanking, OpenAccount>, I <CoreBanking, ATMLogin>})	42

5.6. Threats to Validity

5.6.1. Validity of Subject Models for Experiment 1

In the process of selecting variability models, there is a risk that the subject models may not be representative enough. To mitigate this threat, we selected all 24 models that were developed by Segura et al. [37–39] for their inconsistency analysis experiment. These models were systematically developed through design, development, evaluation and refinement processes and cover various variability models with typical settings including inconsistent and consistent ones. Segura et al. additionally developed 168 models for other analysis experiments such as dead feature analysis. But since these models were not developed for inconsistency analysis experiment, we did not include them for our Experiment 1.

5.6.2. Validity of Subject Models for Experiment 2

There was a risk that the subject models in Experiment 2 may not have been representative enough to enable analysis of whether the ICL tool is accurate in identifying inconsistent models. The two reasons are as follows:

First, there was a risk that the real-world variability models we selected may not have been representative enough to enable analysis of whether the ICL tool is accurate in identifying inconsistent models. To mitigate this threat, as shown in Appendix A, we systematically analyzed all of the real-world variability models in the SPLOT repository to select representatives of variability models. In Appendix A, we examined 1146 real-world variability models in terms of the size of model, i.e., the number of features, and the CTCR (Cross-Tree Constraint Ratio), that is, the ratio of the number of features in the constraints to the number of features in a feature model [24]. Then, we selected the variability models

to be used in Experiment 2 from the representative groups. Since it was introduced in 2009, the SPLOT tool has been available at <http://52.32.1.180:8080/SPLOT/index.html> (accessed on 9 October 2025). The real-world variability models in this SPLOT repository were developed by a variety of individuals and organizations that are not even relevant to SPLOT researchers and the authors of this paper.

Second, there was a risk that the subject models in Experiment 2 cannot be representative models for real-world inconsistent models. To mitigate this threat, the subject models in Experiment 2 were created from the inconsistent and the consistent cases that were used by Segura et al. [37–39] for their inconsistency analysis experiment. Since the subject models cover all critical aspects of inconsistent cases, these models can be representative models for real-world inconsistent models. In addition, Segura’s method is also consistent with the way in which many software product line experts including us analyze cases of inconsistency. As the result, the representative subject models created in this way consist of 16 consistent models and 9 inconsistent models.

5.6.3. Validity of the Scalability of the ICL Tool

Between 11 and 40 features were selected from the real-world variability models selected in Experiment 2. Less than one millisecond was spent performing an Inconsistency Analysis of each model, and accordingly this was omitted from Table 8. As shown in the fifth column of Table 9, the time spent locating the causes of inconsistency in the models ranged from 0.99 to 6.02.

The ICL tool may not be practical for performing inconsistency analysis of large-size variability models. Despite Experiments 1 and 2, there is still a question as to whether the ICL tool is scalable. To mitigate this threat, we conducted additional experiments using the third largest real-world variability model, in the SPLOT repository, the Banking Software model, with the ICL tool. The model consisted of 176 features, with more than 1 billion potential product configurations.

Since this model was consistent, we manually generated a derived model of Banking Software, Banking Software_D1, by adding a constraint of $E < \text{OpenAccount}, \text{ATMLogin} >$ to the model that has original constraints. Now this derived model is inconsistent. So, we performed an Inconsistency Analysis and Inconsistency Cause Locating using the ICL tool. The ICL tool determined that Banking Software_D1 was inconsistent and located the cause of inconsistency. We manually checked that the cause located by the ICL tool is indeed the true cause.

To perform the Inconsistency Analysis on Banking Software_D1, the ICL tool needed less than 1 millisecond. To locate the cause of Banking Software_D1, it took 42 milliseconds, as is shown in Table 10. For all 50 variability models, including Banking Software_D1, which we experimented with, the time the ICL tool needed to locate causes of inconsistency was within milliseconds.

We used the datasets systematically developed for inconsistency analysis in [37–39] for Experiment 1 and the real-world datasets in [9] for Experiment 2. In addition, we used a large real-world dataset in [9] for experiments in Section 5.6.3. Because our experiments cover such a wide spectrum of scale from medium to large real-world sizes, we think that the proposed method and the current ICL tool will also work competently for real-world projects.

The studies in [24,25] stated that SAT-based analysis is very efficient even if the CNF formula becomes very large, as it mainly consists of clauses with two literals when converting a feature model into a CNF formula. Since the analysis of our method is based on the CNF formula and our CNF formula mainly consists of clauses with two literals, we think that it is very likely that our method is scalable.

Because the algorithm of the ICL tool is deterministic, the output results for the same input are always the same. Therefore, it is important to discuss whether various situations in locating the cause of inconsistency have been considered as much as possible rather than discussing “statistical signatures or variability across multiple runs”. By considering the following four points for our experimental data, we think that our experiments reflect a variety of situations to ensure robustness of the results: (1) use of all the data provided by the study [38] that developed the inconsistency analysis dataset; (2) use of real models (available from SPLOT-Software Product Line Online Tools); (3) use of large real models, which includes a real model with 176 features that produces more than 1 billion potential product configurations; and (4) use of a model with an average Cross-Tree Constraint Ratio (CTCR) of 40% or more is used so as to increase the likelihood of discrepancies.

5.6.4. Validity of the Time Complexity of the ICL Tool

The time complexity required to analyze a variability model using the CNF Solver can be expressed as the sum of (1) the time to convert a variability model into a CNF formula and (2) the time to analyze a CNF formula using the CNF Solver.

Both our method and existing methods convert a variability model into a CNF formula in polynomial time. We believe that the existing methods based on propositional logic are similar to our method in terms of time complexity because they are analyzed using the SAT Solver or the CNF Solver. Theoretical worst-case time complexity of these solvers is $O(2^n)$, but in practice, its performance is often much better due to the use of optimization techniques like unit propagation and pure literal elimination, and its efficiency heavily depends on the heuristics used to choose which variables to branch on. In our experiments, the time it takes to analyze the variability model was within the milli-seconds level and increased linearly.

Figure 9 shows the results of our Experiment 2. As can be seen in Figure 9, even if the number of features increases, the time it takes to analyze remains within the milliseconds level and increases linearly. Based on our experimental results and the experimental results of [24,25], we believe that the performance of our method will remain efficient as the number of features increases.

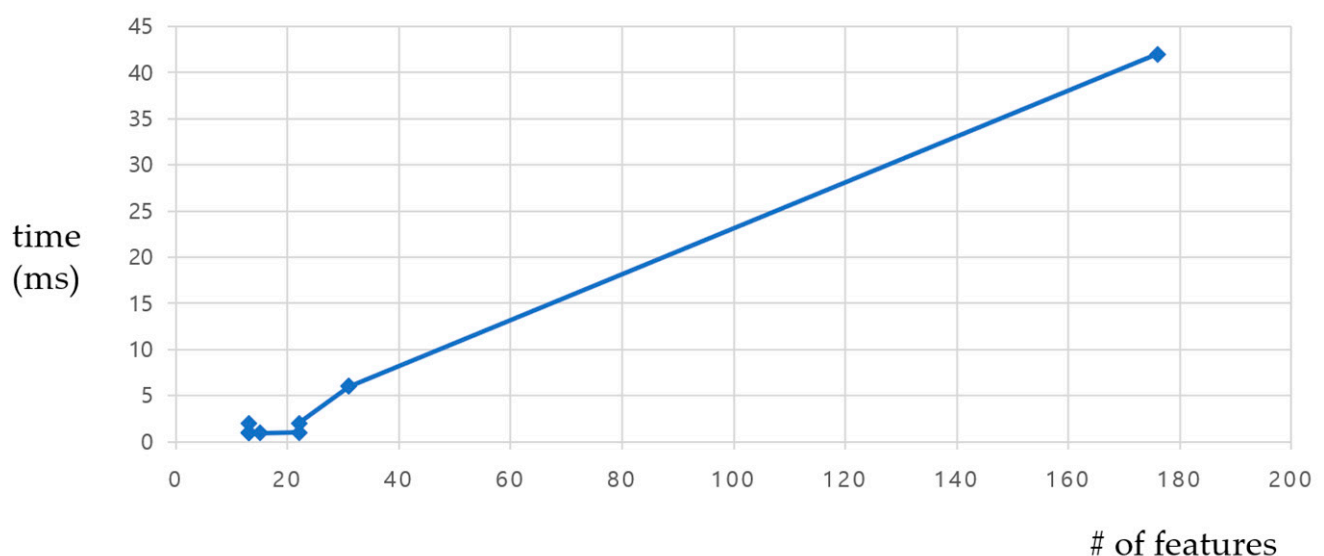


Figure 9. Time taken as the number of features increases.

6. Conclusions

In this paper, we presented an ICL method that accurately locates all causes of inconsistency in a variability model. We have developed a prototype tool, the ICL tool, that automates the entire process of the ICL method and conducted experiments using it to evaluate the ICL method. As a result, for each of the 49 subject variability models, the ICL tool accurately determined whether it is consistent or not and also accurately located all causes of inconsistency in it.

We developed for the first time an inconsistency analysis technique for variability models in the context of software product line development. Such a method did not exist for the feature models. The new results in our work are the ICL method and the ICL tool. They determine whether or not a variability model has an inconsistency and identify the exact locations of its causes if it has an inconsistency.

Our ICL method and ICL tool can be used to model and analyze commonality and variability of a product family in the software product line development process including the requirements, design, and implementation phases. Inconsistency analysis in these phases can be very useful for users because it can proactively address potential issues. Currently, we are planning to disseminate our method to major local industries that require software product line development including the automotive sector, and therefore so far there exists no feedback from users, hence no reported confusion between theoretical and practical issues.

Unlike a single product software development cycle, the software product line development process consists of two phases: the domain engineering phase and the application engineering phase. Readers who want to use the ICL method and ICL tool can use them to model and analyze commonality and variability of a product family during the domain engineering phase as long as variability models are constructed. The processes of the domain engineering phase and the application engineering phase can follow the waterfall model or its variations such as the V-model, an iterative model or an agile process.

As the follow-up research of our work in this paper, researchers can tackle the problems below among others:

(1) They can explore ways of providing more comprehensive information about identified causes of inconsistency. The ICL method currently provides just the immediate conflicting SVM components without providing information of all relevant parts. However, if an automatically located cause shows all relevant parts related to inconsistency (full chain of causes), the developer's modification work will be much facilitated.

(2) They can extend the ICL method to include other analysis perspectives. While the ICL method provides inconsistency analysis and inconsistency cause locating, it lacks the ability to perform, for example, dead feature analysis and associated cause locating.

(3) They can develop translators so that various variability models can be subject to the ICL method. We presented various problems and limitations of the existing feature models in this paper [29]. One critical limitation of feature models is that they can contain ambiguities, which require human interventions to remove. But, before ambiguities are resolved, it is not possible to correctly translate a feature model to an SVM model and apply the ICL method. That is, a translator cannot be meaningfully applied if the source model can be ambiguous.

(4) They can extend the ICL method so that the SVM tool can support various extra constraints supported by various variability modeling tools including FeatureIDE.

(5) They can extend the ICL method so that the SVM tool can automatically repair the causes of inconsistency that are located by our method.

We think that our work has opened up an important direction for software product line variability modeling and analysis. Among others, software product line engineers or

researchers studying analysis of variability models will most benefit from our results by using them as a basis or extending them for the study to improve existing inconsistency analysis studies or the study that detects causes from different analysis perspectives, such as dead feature analysis, false optional variable analysis, and so on. Currently, we are planning to disseminate our method to some major companies which could use software product line development, including an automotive company producing software product families for gasoline, electric vehicle, and hydrogen vehicles.

With future research, we can extend the ICL method and implement it as a fully automated tool. The ICL method helps provide a consistent and rigorous specification for modeling and analyzing the commonality and variability of a product family in software product line development by pinpointing the locations of the causes of inconsistencies and allowing us to remove them.

Author Contributions: Conceptualization, Y.H. and S.K.; methodology, Y.H. and S.K.; software, Y.H. and S.K.; validation, Y.H., S.K. and J.L.; writing—original draft preparation, Y.H., S.K. and J.L.; writing—review and editing, Y.H., S.K. and J.L.; supervision, S.K.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data developed in this paper (with its source code) have been posted at <https://url.kr/fnk7uq> (accessed on 9 October 2025).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. How to Select Subject Models Among Real-World Variability Models

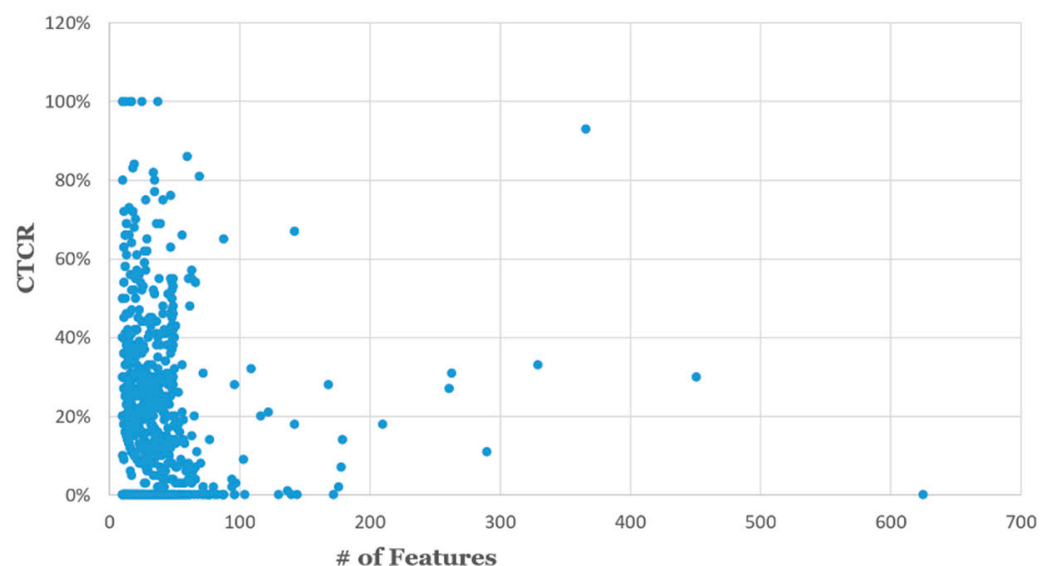


Figure A1. Distribution of models from the number of features and CTCR perspectives.

There are 1146 models as of 17 October 2019 that are managed in the SPLOT repository (splot-research.org), which have been developed by various people and organizations. To select the models to be used in Experiment 2, these models were analyzed from two perspectives, including the number of features and CTCR (Cross-Tree Constraint Ratio).

The analysis results of the number of features and CTCR for 1146 models are shown in Figure A1, Tables A1 and A2. Figure A1 shows distribution of models from the number of features and CTCR perspectives. The SPLOT repository provides information for analysis such as the number of features, CTCR, and so on. Through this information, it is possible to determine the size and complexity of the model. Figure A1 shows that most of the 1146 models are distributed in less than 100 features and less than 40% CTCR.

Table A1. Distribution of models from the number of features perspective.

# Feature	# Model	Ratio
1~10	100	8.73%
11~20	463	40.40%
21~30	205	17.89%
31~40	140	12.30%
41~50	121	10.56%
51~60	41	3.58%
61~70	23	2.01%
71~80	14	1.22%
81~90	8	0.70%
91~	30	2.62%
Sum	1146	100%

Table A2. Distribution of models from CTCR perspective.

CTCR	# Model	Ratio
0%~10%	573	50.00%
11%~20%	186	16.23%
21%~30%	148	12.91%
31%~40%	96	8.38%
41%~50%	62	5.41%
51%~60%	32	2.79%
61%~70%	25	2.18%
71%~80%	11	0.96%
81%~90%	5	0.44%
91%~100%	8	0.70%
Sum	1146	100%

Tables A1 and A2 show more detailed analysis results in Figure A1. As shown in Table A1, 463 models that have features of 11 to 20 accounts for 40.4% of all models. And 205 models that have features of 21 to 30 accounts for 17.89% of all models. As a result, 808 models that have 11 to 40 feature accounts for 70.59% of all models (gray part in Table A1). As shown in Table A2, 573 models that have CTCR of 0% to 10% accounts for 50% of all models. As a result, 1003 models that have CTCR of 0% to 40% accounts for 87.52% of all models (gray part in Table A2).

We presented two conditions for the selection of representative experimental models based on the analysis results in Tables A1 and A2.

- Condition 1: model that has 11 to 40 features
- Condition 2: model that has 0% to 40% CTCR

As shown in Table A3, we selected five models that meet both Condition 1 and Condition 2. The subject models are Alarm System, Smart Hotel, Mobile Media_Lenita, Smart Home, and Automotive System. To evaluate the efficiency of the proposed method for a large-size variability model in Section 5.6.3, Banking Software, the third largest variability model managed by the SPLOT repository, was also selected.

Table A3. Subject models.

Subject Models	# Features	CTCR	Creator
Alarm System	13	30%	Florian Wartenberg
Smart Hotel	13	15%	Jefferson da Silva Barbosa
Mobile Media_Lenita	15	26%	Lenita Ambrosio
Smart Home	22	18%	Jefferson da Silva Barbosa
Automotive System	31	41%	Jean-Vivien Millo
Banking Software	176	2%	Ganesh Khandu Narwane

If subject models are consistent models, we add constraints to each model to generate four derived models per model for the experiment of Inconsistency Analysis. Thus, 5 models are real-world variability models, and the other 20 models are derived models generated by adding constraints to the real-world variability models. Table A4 shows 20 derived models that add constraints to each subject model.

Table A4. Derived models from subject models.

Subjected Model	Derived Models		
	Model ID	CTCR	Add Constraints
Alarm System	Alarm System_Original	30%	N/A
	Alarm System_D1	46%	E <Siren, Warning Light>
	Alarm System_D2	46%	R <Siren, Warning Light>
	Alarm System_D3	46%	R <Siren, Warning Light>, R <Warning Light, Siren>
	Alarm System_D4	62%	R <Siren, Warning Light>, R <Warning Light, Siren>, E <Online Access, User Interface>
Smart Hotel	Smart Home_Original	15%	N/A
	Smart Home_D1	31%	R <SilentAlarm, VisualAlarm>
	Smart Home_D2	38%	R <SilentAlarm, VisualAlarm>, R <AutomatedIllumination, SilentAlarm>
	Smart Home_D3	46%	R <SilentAlarm, VisualAlarm>, E <InfraredSensor, VolumetricSensor>
	Smart Home_D4	46%	R <SilentAlarm, VisualAlarm>, R <AutomatedIllumination, PipedMusic>, R <PipedMusic, SilentAlarm>
Mobile Media_Lenita	Mobile Media_Lenita_Original	26%	N/A
	Mobile Media_Lenita_D1	40%	E <MediaManagement, ScreenSize>
	Mobile Media_Lenita_D2	33%	R <Video, CopyMedia>
	Mobile Media_Lenita_D3	47%	R <Video, CopyMedia>, E <ReceivePhoto, SendPhoto>
	Mobile Media_Lenita_D4	47%	R <Video, CopyMedia>, R <SetFavourites, ViewFavourites>

Table A4. Cont.

Subjected Model	Derived Models		
	Model ID	CTCR	Add Constraints
Smart Home	Smart Home_Original	18%	N/A
	Smart Home_D1	27%	E <Lighting, ControlSystem>
	Smart Home_D2	32%	E <HDTV42, ControlPanel>, R <Lighting, ControlSystem>
	Smart Home_D3	41%	R <ControlSystem, MoviePlayers>, E <Lighting, MoviePlayers>, R <PCPlayers, HDTV32>
	Smart Home_D4	45%	E <Lighting, MoviePlayers>, R <PCPlayers, HDTV32>, R <CellPhone, HDTV42>
Automotive System	Automotive System_Original	41%	N/A
	Automotive System_D1	48%	E <Chassis, Energy Reservoir>
	Automotive System_D2	48%	R <GearBox, Engine>
	Automotive System_D3	48%	E <Climate Control, Manual Control>
	Automotive System_D4	58%	R <GearBpx, Manual Roof Control>, R <Engine, Roof Control with Rain Sensor>

To create an inconsistent model, we manually added constraints to the original model based on all inconsistency and consistency cases provided by Segura et al. [37–39] because they already provide all cases that can be used in inconsistency analysis experiments, which have been developed through a systematic process and include both inconsistency and consistency cases. In this way, it was possible for us to create an inconsistency in the real-world variability model without introducing bias.

The subject models cover all critical aspects of inconsistent cases for the following two reasons:

First, the inconsistent and consistent cases provided by Segura et al. were systematically developed through the design, evaluation, and refinement process described in [37–39]. To cover all critical aspects, these cases were comprehensively evaluated through (1) first evaluation using mutations testing (10th page of [38]), (2) second evaluation using real faults (13th page of [38]), and third evaluation using their tool called FaMa (14th page of [38]).

Second, Segura’s method is also consistent with how several software product line experts, including the authors, analyze cases of inconsistency. When inconsistency occurs in a variability model, it was proven that the semantics between the relationships and constraints conflict [36]. An example of this is described in detail in Figure 3.

We analyzed in advance whether the model is an inconsistent or a consistent model using the SPLOT tool, as previously mentioned in Section 5.4.1. Since there are no methods and tools to locate the cause of the inconsistency, our software product line experts manually verified the causes by confirming compliance with all cases provided by Segura et al. In addition, the experts manually analyzed the select model through the combinations of various relationships and constraints that make up the variability model. Finally, the experts checked whether there was an inconsistency in the variability model that eliminated the cause.

Appendix B. Variability Models Drawn with the SVM Tool for Experiment 2

Appendix B shows 3 of the 25 models used in Experiment 2. As shown in Figures A2–A4, the black dotted line indicates the point where the inconsistency occurred. Figure A2 shows the variability model for Alarm System_D3 that is drawn with the SVM tool. Instead of red color in Alternative Relationship and Optional Relationship, it is indicated as (VP_ALT). And, instead of green color in Collection Alternative Relationship, it is indicated as (VP_OR).

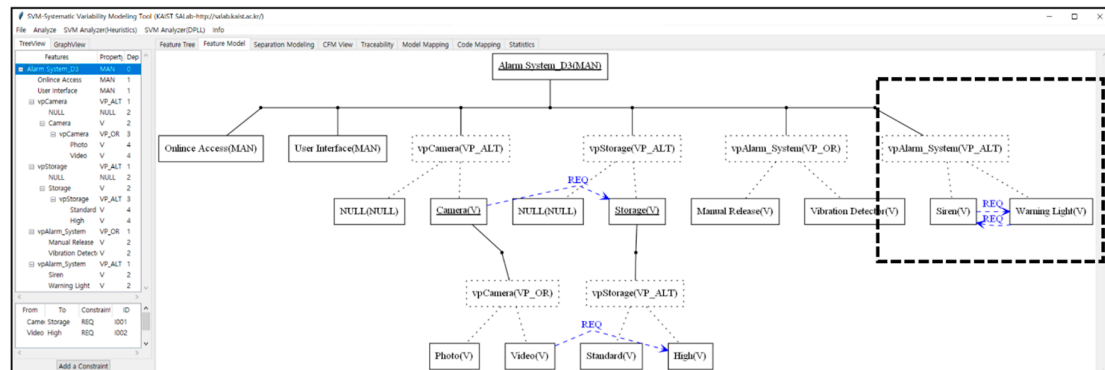


Figure A2. Variability model for Alarm System_D3.

Figure A3 shows an Automotive System_D4 variability model with the SVM tool.

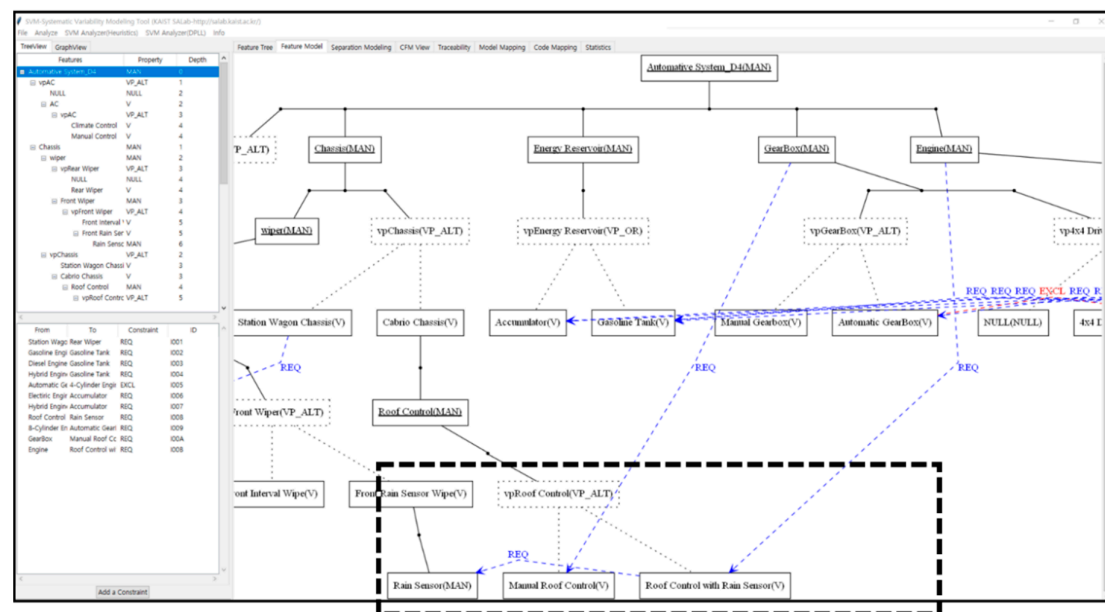


Figure A3. Variability model for Automotive System_D4.

Figure A4 shows the variability model for Alarm System_D4 that is drawn with the SVM tool.

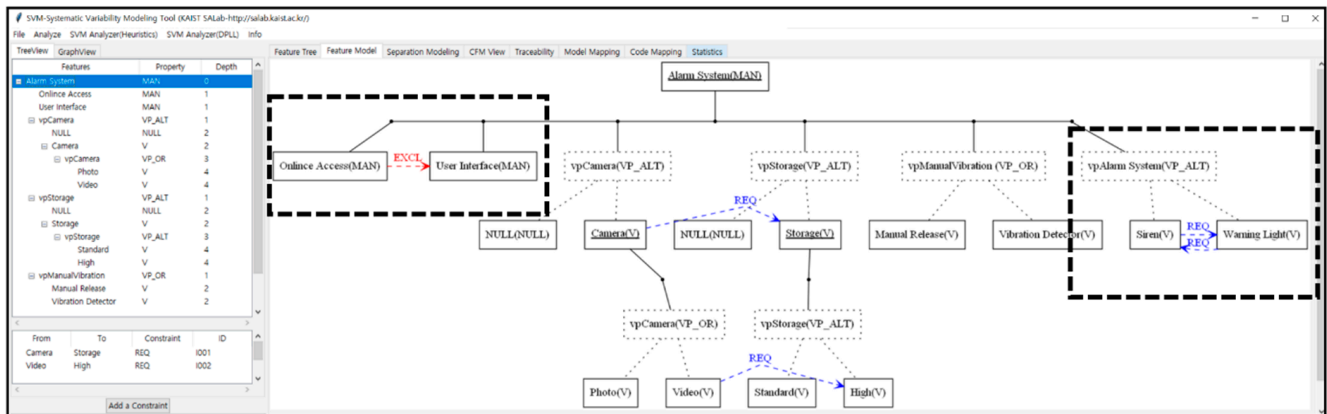


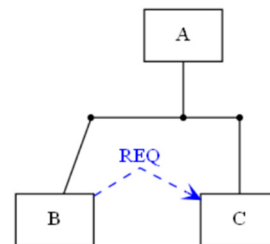
Figure A4. Variability model for Alarm System_D4.

The images listed in Appendix B were reduced due to limited space. Therefore, all data developed in this paper have been posted at <https://url.kr/fnk7uq> (accessed on 9 October 2025).

Appendix C. Example 4.2 in Section 4.2.1

Example 4.2 (Step 1. Convert an SVM variability model in Figure 3a of Section 3.3 to CNFs)

Input:



Step (1-1) The SVM variability model in Figure 3a of Section 3.3 is decomposed into SVM elements in the first column of Figure A5.

SVM Elements	Components of CNF2
A (Root Node)	(A)
	$(\neg A \vee B) \wedge (A \vee \neg B)$
	$(\neg A \vee C) \wedge (A \vee \neg C)$
	$(\neg B \vee C)$

Figure A5. SVM elements of Figure 3a of Section 3.3 and components of CNF2.

Step (1-2) Each SVM element is converted to the corresponding component of CNF2 on the second column of Figure A5 using the conversion rules in Table 2 of Section 4.2.1.

Step (1-3) CNF2_{IC} and CNF2_{EC} are generated by connecting with conjunction the components of CNF2 on the second column of Figure A5.

Output 1: CNF2_{IC} = $(A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C) \wedge (\neg B \vee C)$

Output 2: CNF2_{EC} = $(A) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (A \vee \neg C)$

Step (1-4) CNF2_{IC}^{*} is generated by assigning a unique subscript index to each literal of CNF2_{IC} obtained in Steps 1-3.

Output 3: CNF2_{IC}^{*} = $(A_1) \wedge (\neg A_2 \vee B_1) \wedge (A_3 \vee \neg B_2) \wedge (\neg A_4 \vee C_1) \wedge (A_5 \vee \neg C_2) \wedge (\neg B_3 \vee C_3)$

References

1. Linden, F.; Schmid, K.; Rommes, E. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2007.
2. Pohl, K.; Böckle, G.; Linden, F. *Software Product Line Engineering: Foundations, Principles and Techniques*; Springer: Berlin/Heidelberg, Germany, 2005.
3. Maßen, T.; Lichter, H. Deficiencies in feature models. In Proceedings of the Workshop on Software Variability Management for Product Derivation-Towards Tool Support, Boston, MA, USA, 30 August 2004; Springer: Berlin/Heidelberg, Germany, 2004; Volume 44.
4. Batory, D. Feature models, grammars, and propositional formulas. In Proceedings of the Software Product Lines Conference, Rennes, France, 26–29 September 2005; pp. 7–20.
5. Trinidad, P.; Benavides, D.; Durán, A.; Ruiz-Cortés, A.; Toro, M. Automated error analysis for the agilization of feature modeling. *J. Syst. Softw.* **2008**, *81*, 883–896. [\[CrossRef\]](#)
6. Benavides, D.; Segura, S.; Trinidad, P.; Ruiz-Cortés, A. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In Proceeding of the First International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS), Limerick, Ireland, 16–18 January 2007; pp. 129–134.
7. Zhang, W.; Zhao, H.; Mei, H. A propositional logic-based method for verification of feature models. In Proceedings of the International Conference on Formal Engineering Methods, Seattle, WA, USA, 8–12 November 2004; pp. 115–130.
8. Mendonca, M.; Branco, M.; Cowan, D. SPLOT: Software product lines online tools. In Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, Orlando, FL, USA, 25–29 October 2009; pp. 761–762.
9. Software Product Lines Online Tools. Available online: <http://52.32.1.180:8080/SPLOT/index.html> (accessed on 9 October 2025).
10. Wang, H.; Li, Y.; Sun, J.; Zhang, H.; Pan, J. A semantic web approach to Feature modeling and verification. In Proceedings of the Workshop on Semantic Web Enabled Software Engineering, Galway, Ireland, 6 November 2005; pp. 46–60.
11. Wang, H.; Li, Y.; Sun, J.; Zhang, H.; Pan, J. Verifying feature models using OWL. *J. Web Semant.* **2007**, *5*, 117–159. [\[CrossRef\]](#)
12. Mannion, M. *Using First-Order Logic for Product Line Model Validation*. *International Conference on Software Product Lines*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 176–187.
13. Sun, J.; Zhang, H.; Li, Y.; Wang, H. Formal semantics and verification for feature modeling. In Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, Shanghai, China, 16–20 June 2005; pp. 303–312.
14. Thüm, T.; Kästner, C.; Benduhn, F.; Meinicke, J.; Saake, G.; Leich, T. FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.* **2014**, *79*, 70–85. [\[CrossRef\]](#)
15. Website, FeatureIDE. Available online: <https://featureide.github.io/> (accessed on 9 October 2025).
16. Kowal, M.; Ananieva, S.; Thüm, T. Explaining anomalies in feature models. *ACM SIGPLAN Not.* **2016**, *52*, 132–143. [\[CrossRef\]](#)
17. Wang, B.; Xiong, Y.F.; Hu, Z.J. Interactive Inconsistency Fixing in Feature Modeling. *J. Comput. Sci. Technol.* **2014**, *29*, 724–736. [\[CrossRef\]](#)
18. Hentze, M.; Pett, T.; Thüm, T.; Schaefer, I. Hyper explanations for feature-model defect analysis. In Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems, Krems, Austria, 9–11 February 2021.
19. Deelstra, S.; Sinnema, M.; Bosch, J. Experiences in software product families: Problems and issues during product derivation. In Proceedings of the International Conference on Software Product Lines, Boston, MA, USA, 30 August–2 September 2004; pp. 165–182.
20. Choi, H.; Lee, K.; Lee, J.; Kang, K.C. Multiple Views of Feature Models to Manage Complexity. In Proceedings of the Workshop on Scalable Modeling Techniques for Software Product Lines (SCALE), San Francisco, CA, USA, 24–28 August 2009.
21. Kang, K.C.; Cohen, S.G.; Hess, J.A.; Novak, W.E.; Peterson, A.S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*; SEI Technical Report; Defense Technical Information Center: Fort Belvoir, VA, USA, 1990.
22. Kang, K.C.; Lee, J.; Donohoe, P. Donohoe Feature-Oriented Product Line Engineering. *IEEE Softw.* **2002**, *19*, 58–65. [\[CrossRef\]](#)
23. Benavides, D.; Segura, S.; Ruiz-Cortés, A. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* **2010**, *35*, 615–636. [\[CrossRef\]](#)

24. Mendonca, M.; Wąsowski, A.; Czarnecki, K. SAT-based Analysis of Feature Models is Easy. In Proceedings of the 13th International Software Product Line Conference, San Francisco, CA, USA, 24–28 August 2009; pp. 231–240.
25. Liang, J.; Ganesh, V.; Czarnecki, K.; Raman, V. SAT-based Analysis of Large Real-world Feature Models is Easy. In Proceedings of the 19th International Conference on Software Product Line, Nashville, TN, USA, 20–24 July 2015; pp. 91–100.
26. Sundermann, C.; Thüm, T.; Schaefer, I. Evaluating# SAT solvers on industrial feature models. In Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg, Germany, 5–7 February 2020.
27. Meinicke, J.; Thüm, T.; Schröter, R.; Benduhn, F.; Leich, T.; Saake, G. *Mastering Software Variability with FeatureIDE*; Springer: Cham, Switzerland, 2017.
28. Elfaki, A.O.; Phon-Amnuaisuk, S.; Ho, C.K. Modeling Variability in Software Product Line Using First Order Logic. In Proceedings of the 2009 Seventh ACIS International Conference on Software Engineering Research, Management and Applications, Haikou, China, 2–4 December 2009; pp. 227–233.
29. Kang, S.; Han, Y.; Ahn, H.; Lee, J. Critical analysis of feature model evolution. In Proceedings of the 8th International Conference on Software Engineering and Service Science (ICSESS), Beijing, China, 24–26 November 2017.
30. Le, V.-M.; Felfernig, A.; Tran, T.N.T.; Atas, M.; Uta, M.; Benavides, D.; Galindo, J. DirectDebug: A software package for the automated testing and debugging of feature models. *Softw. Impacts* **2021**, *9*, 100085. [[CrossRef](#)]
31. Han, Y.; Kang, S. *SVM: A Variability Modeling Tool for Software Product Line*; CS-TR-2022-425; School of Computing, KAIST: Daejeon, Republic of Korea, 2022; Available online: <https://cs.kaist.ac.kr/research/techReport> (accessed on 8 April 2022).
32. Kang, S.; Han, Y.; Ahn, H.; Lee, J. Compositional Feature Model CFM for Software Product Line Development and a Variability Modeling Tool SVM that Supports CFM. *Commun. Korean Inst. Inf. Sci. Eng.* **2018**, *36*, 9–18. (In Korean)
33. Leblanc, H.; Wisdom, W.A. *Deductive Logic*, 2nd ed.; Allyn and Bacon: Boston, MA, USA, 1976.
34. DPLL Algorithm. Available online: http://ais.informatik.uni-freiburg.de/teaching/ss17/ki/slides/ai08_satisfiability_and_model_construction.pdf (accessed on 9 October 2025).
35. Russell, S.J.; Norvig, P. *Artificial Intelligence a Modern Approach*, 3rd ed.; Pearson: London, UK, 2017.
36. Mendonca, M. Efficient Reasoning Techniques for Large Scale Feature Models. Ph.D. Thesis, University of Waterloo, Waterloo, ON, Canada, 2009; pp. 1–170.
37. Segura, S.; Benavides, D.; Ruiz-Cortés, A. Functional testing of feature model analysis tools. a first step. In Proceedings of the 5th Software Product Lines Testing Workshop (SPLiT 2008), Limerick, Ireland, 8–12 September 2008; IEEE: New York, NY, USA, 2008; pp. 36–39.
38. Segura, S.; Benavides, D.; Ruiz-Cortés, A. *FaMa Test Suite v1.2*; Technical Report ISA-10-TR-01; ISA Research Group: Arlington, VA, USA, 2010; pp. 1–52.
39. Segura, S.; Benavides, D.; Ruiz-Cortés, A. Functional testing of feature model analysis tools: A test suite. *IET Softw.* **2011**, *5*, 70–82. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.