Early Consistency Checking between Specification and Implementation Variabilities

Xhevahire Tërnava, Philippe Collet

▶ **To cite this version:**

Xhevahire Tërnava, Philippe Collet. Early Consistency Checking between Specification and Implementation Variabilities. the 21st International Systems and Software Product Line Conference - Volume A, Sep 2017, Sevilla, France. 10.1145/3106195.3106209 . hal-01699878

HAL Id: hal-01699878
https://hal.science/hal-01699878

Submitted on 2 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Early Consistency Checking between Specification and Implementation Variabilities

Xhevahire Tërnava and Philippe Collet
Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France
{ternava,collet}@i3s.unice.fr

ABSTRACT

In a software product line (SPL) engineering approach, the addressed variability in core-code assets must be consistent with the specified domain variability, usually captured in a variability model, *e.g.*, a feature model. Currently, the support for checking such consistency is limited, mostly when a single variability implementation technique is used, *e.g.*, preprocessors in C. In realistic SPLs, variability is implemented using a combined set of traditional techniques, *e.g.*, inheritance, overloading, design patterns. An inappropriate choice and combination of such techniques become the source of variability inconsistencies. In this paper, we present a tooled approach to check the consistency of variability between the specification and implementation levels, when several variability implementation techniques are used together. The proposed method models the implemented variability in terms of variation points and variants, in a forest-like structure, and uses slicing to partially check the resulting propositional formulas at both levels. As a result, it offers an early and automatic detection of inconsistencies. We implemented and successfully applied the approach in four case studies. Our implementation, publicly available, detects inconsistencies in a very short time, which makes possible to ensure consistency earlier in the development process.

1 INTRODUCTION

In a Software Product Line (SPL) engineering approach, the specified domain variability is commonly captured in a variability model, *e.g.*, a feature model (FM) using the concept of *feature* as a reusable unit [11, 15]. The FM is a tree structure consisting of *mandatory*, *optional*, *or*, and/or *alternative* relation logic between features, while their cross-tree constraints are expressed in propositional logic. Semantically, an FM represents the valid software products (*i.e.*, feature configurations) within an SPL.

The variability specified in terms of features in an FM has to be addressed by different stakeholders in different software models and in core-code assets. As the addressed variability must conform to the specified variability in the FM, it becomes the potential source for *inconsistencies* [28] within an SPL. For example, when an *alternative* group of features is realized as an *or* relation logic between the variable units in core assets. Respectively, when a *mandatory* feature is realized as an *optional* unit in core assets. The occurrence of such inconsistencies is of a major importance, *e.g.*, for a given feature configuration it is not possible to derive the respective software product from the developed core assets.

According to a recent survey about consistency checking in SPLE, there are three major approaches to address consistency issues: *(i)* within the variability models (*i.e.*, FMs), *(ii)* between the FM and other software models, or *(iii)* between the FM and its implementation (*i.e.*, core-code assets). These also correspond to the locations where inconsistencies can happen, as mentioned by Vierhauser et. al [32].

Currently, the support for checking variability consistency between the FM and core-code assets is limited. The existing approaches are mostly conceived for resolving inconsistencies within a specific software, *e.g.*, the Linux kernel [22, 29], or when the variability is implemented by a single variability implementation technique, *e.g.*, using preprocessors in C [19]. However in realistic SPL settings, variability is implemented by using a combined set of traditional techniques, *e.g.*, inheritance, overloading, design patterns. An inappropriate choice and combination of such techniques become the source of variability inconsistencies that cannot be detected by existing approaches

In this context, there are several challenges to be addressed regarding the consistency checking of variability between the FM and core-code assets.

C1. Checking the consistency of variability between the specification and implementation levels.

Originally, in the FORM method [16] [8, Ch.8], the need to model separately the variability at the specification and realization levels is already present. While the variability at a realization level is more about the software variability, the specification one represents the variability between the software products themselves within an SPL [24]. In most variability management approaches, it is up to the reader to understand whether an FM is used to describe the variability at the specification or realization levels [23]. Moreover, the mapping of features to variable units in implementation is 1 – to – 1, *e.g.*, between features and preprocessor directives in C [19, 22, 29], although a directive can be scattered in core-code assets. In such cases, the FM is used to model only the implemented variability, or from both levels into a single model. In reality, the mapping of features from the specification level to their implementation is n – to – m [25, Ch.4], which represents a serious challenge during the consistency checking of variability.

C2. Checking the consistency of variability when a combined set of traditional variability implementation techniques is used.

The implemented variability by several traditional techniques can be modeled in terms of variation points (*vp*-s) and *variants* [9, 11, 14, 27] (they will be defined in Section 2). These *vp*-s are diverse compared to the case when a single variability implementation technique is used to implement the whole variability, *e.g.*, preprocessors in C. In case a *vp* is implemented using an improper technique, several inconsistencies may appear, for example, when an *alternative* relation logic between features in an FM is implemented by a *vp*

with an *or* relation logic between its variants. In such case, the number of possible products in the FM is inconsistent with the possible products that can be derived from the core-code assets. Therefore, in complement to C1, a consistency checking approach should be able to check whether the right technique to implement a *vp* and its *variants* is used.

C3. Achieving an early detection of variability inconsistencies. The consistency of variability can be checked only after the variability is realized, but not necessarily only after the whole specified variability is addressed. Commonly, some variability is deferred to be implemented later or during the application engineering phase. In addition it becomes harder to fix the inconsistencies after all of them are shown at the same time at the end [32]. In particular, it has been shown that trying to change the implementation technique for a *vp*, only after the whole SPL is implemented, can be very costly [4]. Therefore, an approach for detecting earlier the variability inconsistencies is needed, for example, to be able to select a single or a group of *vp*-s and to check them against the specified features in an FM early during the development process. Typically, we could expect that the earlier a variability inconsistency is identified, the cheaper becomes the fix.

To address these challenges, we propose a method for checking the consistency between specification and implementation variabilities, when the variability implemented by several techniques is modeled in terms of variation points and variants. The method supports an early checking, which is made feasible by organizing the implemented variability in a forest-like structure [30]. In this way, we can select some of the implemented variability easily and check its consistency against the specified variability in an FM, by slicing their propositional formulas.

In the following, we give some background on variability modeling at the specification and implementation levels, as well as on traceability between the two levels (Section 2). We then describe our consistency checking method (Section 3), for single and multiple types of trace links. We also report on our prototype implementation, and applications to four case studies (Section 4). We then discuss related work (Section 5), and conclude by evoking obtained properties, limitations and future work (Section 6).

2 MOTIVATIONS

In the following the abstractions and variability models at specification and implementation levels that support our approach are introduced and exemplified, as well as trace links between their abstractions. The end of the section discusses the form of consistency checking that must be provided in our context.

2.1 Variability Modeling at Specification Level

In an SPL, variability at specification level is commonly modeled in a feature model (FM) using the concept of *feature* [11, 15]. For example, Figure 1 shows an excerpt of the FM for the Graph SPL [1], which is one of our analyzed case studies. This SPL is quite well understood and used by the community [21]. The Graph SPL is conceptually represented by the root feature, `GraphProductLine`, and has two compound mandatory features, `Type` and `Weight`, with their alternative variant features, «`Directed`, `Undirected`» and «`Weighted`, `Unweighted`», respectively. It has also one compound optional feature, `Search`, with two alternative variant features, `DFS` and `BFS`. This excerpt of FM has no cross-tree constraints between features. Commonly, they are shown in propositional logic, *e.g.*, when an algorithm feature requires `Directed` and `Weighted` features, this is written at the end of the feature diagram as $Algorithm_x \rightarrow Directed \wedge Weighted$.

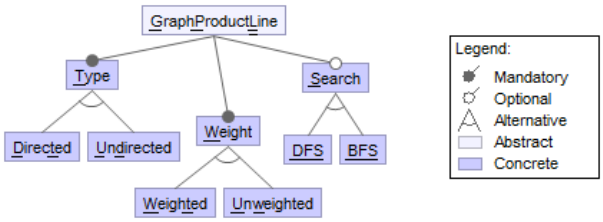

Figure 1: Excerpt of the FM of the Graph SPL

2.2 Variability Modeling at Implementation Level

In realistic SPL settings, variability is implemented by using a combined set of traditional techniques, *e.g.*, inheritance, overloading, design patterns. These techniques offer a form of *imperfectly modular* variability [30] at the implementation level. This *Imperfect* form of modularity comes from the fact that a feature at specification level is a domain concept, while its refinement at core-code assets is a set of variation points (*vp*-s) and *variants* [8, Ch. 3] that may be modular, *i.e.*, it may not have a direct and single mapping. A *vp* is the place in a design or an implementation that identifies where the variation occurs. It represents the used technique to realize the variability, while the way that a *vp* is going to vary is expressed by its *variants* [14, 30]. They can also be understood as the symmetry (*i.e.*, commonality invariance or *vp*-s), and the symmetry breaking places (*variants*) in software [9, 10].

```
1  import dsl._
2  import scala.reflect.runtime.universe._
3  object tvm_search {
4  /* The abstractions of variation points and variants */
5  val Search = OPT_VP(asset(....)) //...
6
7  /* The modeling of the implemented variability */
8  import fragment._
9    fragment("Search.scala") {
10     Search is ALT with_variants (dfs, bfs) use
11     STRATEGY_P with_binding RUN_TIME and_evolution CLOSE
12   }
13 }
```

Listing 1: Usage of the DSL to model the TVM for the Search *vp*

As the *vp*-s are not by-products of the implementation techniques [7], they should be explicitly modeled. Instead of modeling the whole implemented variability at once and in one place, we choose to model it in a fragmented way [30]. A fragment can be any unit, *i.e.*, a package, file, or class, that has inner variability and is worth to be modeled locally and separately in a technical variability model (TVM). A TVM is a concept that contains the abstractions of *vp*-s and *variants*, their strong consistency with the core-code assets, and model the variability of some specific core-code assets [30].

[1] The whole FM is available at https://github.com/ternava/Expressions_SPL/wiki/Feature-Models

Technically, we use a textual Domain Specific Language (DSL) [2], written in Scala, to model the implemented variability in TVMs. The TVM for the `Search` vp, expressed using the textual DSL, is shown in Listing 1. In an illustration, Figure 2 shows two TVMs with two

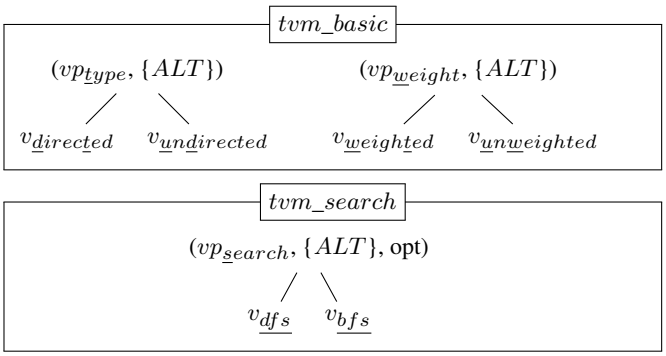

Figure 2: Documentation of the implemented variability in TVMs for the Graph SPL (*cf.* Figure 1)

and one *vp*-s, respectively, which model the implemented variability of the Graph SPL (*cf.* Figure 1).

The implemented variability can be modeled using five available types of *vp*-s: *(i) ordinary*, *e.g.*, a simple *vp*-s as in tvm_basic (*cf.* Figure 2), *(ii) optional*, when the *vp* itself, not its variants, is optional, *e.g.*, the `Search` vp (line 5, Listing 1), *(iii) nested*, when some variable part in a core-code asset becomes the common part for some other variability, *(iv) technical*, a *vp* that is introduced and implemented only for supporting internally the implementation of a specified *vp*, and *(v) unimplemented*, when a *vp* is introduced but has no implemented *variants* yet. Depending on the used variability implementation technique, each of these *vp*-s can be different with regards to the *(i)* relation logic between its variants (alternative, or, optional), *(ii)* their binding time (*e.g.*, compile-time, runtime [8, Ch. 4]), or *(iii)* their evolution (*i.e.*, to be extended with new variants in the future or not).

All TVMs of an SPL constitute the Main TVM (MTVM) at the implementation level. Therefore, the modeled variability in terms of *vp*-s at the implementation level has a forest-like structure (*cf.* Figure 2), unlike the tree structure of features in an FM.

2.3 Trace Links

The specified features in an FM and their implementation as *vp*-s and *variants* in TVMs use different names, and we consider they have n – to – m mapping within the SPL. Therefore, their mapping correspond to trace links. For example, to indicate that the feature `Search` (*cf.* Figure 1) is implemented by the *vp* `vp_Search` (*cf.* Figure 2), their trace link is `Search` ↔ `vp_Search`. Technically, in this work, the trace links are established by using the DSL. They are bidirectional links and are kept in a map structure. It must be noted that our contribution is applicable in a general case where these trace links exist or can be established by any other means.

[2]The source of our DSL,the implementation of Graph SPL, and examples of TVMs, are available at https://github.com/ternava/Variability-CChecking.

Table 1: Translation rules from an FM to propositional logic

Where P is a compound feature and $C_1, C_2, ..., C_n$ its subfeatures.

Relation Logic	Propositional Logic
Mandatory	$P \leftrightarrow C_i$
Optional	$C_i \rightarrow P$
Or-group	$P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i$
Alternative-group	$\left(P \leftrightarrow \bigvee_{1 \leq i \leq n} C_i\right) \wedge \bigwedge_{i<j} \left(\neg C_i \vee \neg C_j\right)$

2.4 On Consistency Checking

For the consistency checking process to scale on large models, we rely on existing automated analysis techniques for feature modeling [6]. We specifically choose solving techniques that rely on propositional logic [5, 6], in which an FM is translated into a propositional formula and SAT-solved. Table 1 shows the well-known translation rules for each relation logic between features in an FM. Cross-tree constraints are considered to be already propositional formulas. Similar to the FM ones, the relation logic between *variants* in a *vp* in a TVM is translated to a propositional formula.

In the literature, different analysis operators for checking FM consistency have been devised [6]. The most important ones concern the detection of the three following anomalies (*i.e.*, inconsistencies):

- *Validity*. The FM is valid if it represents at least a valid configuration, *i.e.*, at least a single software product.
- *Dead features*. A feature is dead if it is not part of any software product, *i.e.*, of any configuration.
- *False optional features*. When a variable feature is part of every configuration, thus becoming a mandatory feature.

These anomalies are common within a single FM, or between FMs at the same abstraction level. In our context, being able to check for variability inconsistencies between two variability models that are supposed to represent the same variability in two different abstraction levels should meet challenge C1. As the TVM model represents the variability implementation with different variability implementation techniques captured through the proposed DSL, the consistency checking between these two models should also meet C2. Finally, to meet challenge C3, we should also be able to deal with partial models, *i.e.*, with partial but semantically correct corresponding propositional formulas.

3 CONSISTENCY CHECKING

3.1 Principles

Following an inconsistency management approach [28], we define a consistency rule, which represents what must be satisfied by the variabilities at specification and implementation levels.

Consistency Rule. *Within an SPL, where the specified domain variability and implemented variability convey the same functionality, they also should represent the same set of software products.*

An inconsistency, *i.e.*, when the consistency rule is not satisfied, concerns a specific feature configuration for which it is impossible to derive a concrete software product from the existing core-code assets, despite the fact that the whole specified variability is implemented.

We thus propose a method, based on propositional logic, for checking and detecting the source of such variability inconsistency.

Conversion to Propositional Logic. To begin with, we convert the modeled variability at the specification and implementation level, *i.e.*, the FM and MTVM, in propositional logic. In this way, the whole issue of consistency checking is translated to merely analysing their propositional formulas ϕ_{FM} and ϕ_{MTVM}, respectively. For example, using the underlined letters for the feature names in Graph SPL (*cf.* Figure 1) and translation rules in Table 1, its propositional formula ϕ_{FM_g} [3] is (*e.g.*, for `GraphProductLine` is used the letter g):

$$\begin{aligned}\phi_{FM_g} = \underline{g} \wedge \underline{(t \leftrightarrow g)} \wedge \underline{(w \leftrightarrow g)} \wedge \underline{(s \rightarrow g)} \wedge \\ (t \leftrightarrow (dt \vee ud)) \wedge (\neg dt \vee \neg ud) \wedge (w \leftrightarrow (wt \vee uw)) \wedge \\ (\neg wt \vee \neg uw) \wedge \underline{(s \leftrightarrow (dfs \vee bfs))} \wedge \underline{(\neg dfs \vee \neg bfs)}\end{aligned} \quad (1)$$

Similarly, for the TVMs in Figure 2, the $\phi_{TVM_{basic}}$ and $\phi_{TVM_{search}}$ are given in Eq. 2 and Eq. 3, respectively. A mandatory *vp* in a TVM is shown as a single positive literal in a propositional formula, *e.g.*, the vp_t or vp_w in Eq. 2, whereas an optional *vp* needs another parent feature to become optional. For this reason, we inserted a mandatory root *vp* that is the common root for all *vp*-s, *e.g.*, see the vp_{root} in Eq. 3 which is used to make optional the $vp_{\underline{s}earch}$ in Figure 2. When the *vp* is mandatory the vp_{root} does not make any difference in the formula, and thus we did not show it in Eq. 2.

$$\begin{aligned}\phi_{TVM_{basic}} = vp_t \wedge vp_w \wedge \\ (vp_t \leftrightarrow (v_{dt} \vee v_{ud})) \wedge (\neg v_{dt} \vee \neg v_{ud}) \wedge \\ (vp_{wt} \leftrightarrow (v_{wt} \vee v_{uw})) \wedge (\neg v_{wt} \vee \neg v_{uw})\end{aligned} \quad (2)$$

$$\begin{aligned}\phi_{TVM_{search}} = vp_{root} \wedge (vp_s \rightarrow vp_{root}) \wedge \\ \left(vp_s \leftrightarrow \left(v_{dfs} \vee v_{bfs}\right)\right) \wedge \left(\neg v_{dfs} \vee \neg v_{bfs}\right)\end{aligned} \quad (3)$$

The consistency rule then corresponds to the fact that ϕ_{FM} and ϕ_{MTVM} must be semantically equivalent, *i.e.*,

$$\begin{aligned}\phi_{FM} \equiv \phi_{MTVM} \\ \text{or, } \phi_{FM} \equiv \left(\phi_{TVM_1} \wedge \phi_{TVM_2} \wedge ... \wedge \phi_{TVM_n}\right)\end{aligned} \quad (4)$$

Two propositional formulas are *semantically equivalent* if and only if they have the same set of models. Let be $[\![\phi_{FM}]\!]$ the set of feature configurations for ϕ_{FM} and $[\![\phi_{MTVM}]\!]$ the set of *vp*-s and *variants* configurations for ϕ_{MTVM}. Every feature configuration in $[\![\phi_{FM}]\!]$ has a mapping to a *vp*-s and *variants* configuration in $[\![\phi_{MTVM}]\!]$. As ϕ_{FM} and ϕ_{MTVM} use different variability abstractions (*i.e.*, different names for features and *vp*-s with *variants*, respectively), which can have an n – to – m mapping, then we could check their semantic equivalence only under the existence of their trace links. Consequently, the accurate formal definition of the consistency rule (*cf.* Eq. 4) becomes:

$$\phi_{FM} \wedge \phi_{TL} \equiv \phi_{MTVM} \wedge \phi_{TL} \quad (5)$$

As the trace links are bidirectional, *i.e.*, features and *vp*-s with *variants* mapped to each other as $f \leftrightarrow vp$, then Eq. 5 is valid thanks to the substitution theorem in propositional logic [17]:

$$\phi_{FM(f)} \wedge (f \leftrightarrow vp) \equiv \phi_{MTVM(vp)} \wedge (f \leftrightarrow vp) \quad (6)$$

i.e., within the context of their trace links ϕ_{TL}, ϕ_{FM} and ϕ_{MTVM} represent the same variability. Moreover, we assume that,

[3] The underlined parts of the formula will be explained in the following.

Assumption 1. *The mapping between features in an FM to the vp-s and variants in TVMs is performed by bidirectional trace links ϕ_{TL}, which are established before the consistency checking and are consistent themselves.*

Generally, the mapping between features and *vp*-s with *variants* is n – to – m. In our approach, we exclusively consider the single links (*i.e.*, 1 – to – 1) and multiple links (*i.e.*, 1 – to – m), as they are the two common forms of mapping in our targeted SPL implementations. In our example, ϕ_{FM_g} has 1 – to – 1 mapping to $\phi_{TVM_{basic}}$ and $\phi_{TVM_{search}}$. Therefore, their single links ϕ_{TL_g} are:

$$\begin{aligned}\phi_{TL_g} = \underline{(g \leftrightarrow vp_{root})} \wedge \\ (t \leftrightarrow vp_t) \wedge (w \leftrightarrow vp_w) \wedge \underline{(s \leftrightarrow vp_s)} \wedge \\ (dt \leftrightarrow v_{dt}) \wedge (ud \leftrightarrow v_{ud}) \wedge (wt \leftrightarrow v_{wt}) \wedge \\ (uw \leftrightarrow v_{uw}) \wedge \underline{\left(dfs \leftrightarrow v_{dfs}\right)} \wedge \underline{\left(bfs \leftrightarrow v_{bfs}\right)}\end{aligned} \quad (7)$$

According to the consistency rule in Eq. 5, ϕ_{FM_g} in Eq. 1 represents the same software products with $\{\phi_{TVM_{basic}}, \phi_{TVM_{search}}\} \in \phi_{MTVM_g}$ in Eq. 2 and Eq. 3 when

$$\phi_{FM_g} \wedge \phi_{TL_g} \equiv \left(\phi_{TVM_{basic}} \wedge \phi_{TVM_{search}}\right) \wedge \phi_{TL_g} \quad (8)$$

i.e., within the context of trace links ϕ_{TL_g}, ϕ_{FM_g} is equivalent to $\left(\phi_{TVM_{basic}} \wedge \phi_{TVM_{search}}\right)$.

3.2 Proposed Method

For checking the consistency of the entire variability between both levels, *i.e.*, using the Eq. 5, it is required that *(i)* the whole specified variability in an FM is implemented and documented in an MTVM, and *(ii)* all their trace links are established. This restricts checking to a complete system, which itself is likely to be represented by large variability models, harder to check, but also harder for tracing and fixing inconsistencies after all of them are shown at once [32]. In addition, it is common that some variability implementation is deferred, e.g., during the application engineering phase, and some partial checking is then highly desirable. Besides, even for illustrative SPLs with a small set of features, the propositional formula to compute Eq. 5 becomes already quite large. Moreover, in realistic SPLs, checking for inconsistencies only within a single FM has still scalability issues [6].

To overcome these problems, we propose a consistency checking method for detecting the variability inconsistencies earlier during the development process. Its main steps are based on slicing, substitution, and assertion properties, which are depicted in Figure 3. First, we will explain the method when single links are used, to extend it to multiple links just after.

Initial Checking. As a prerequisite, we check first if ϕ_{FM} and ϕ_{MTVM} individually are consistent. To do so, we use state of the art methods to check if each of them in isolation is valid, as well as free of dead and false-optional (a.k.a common) features or *vp*-s and *variants*, respectively (*cf.* Section 2.4 and Figure 3). We also check whether the Assumption 1 about trace links hold, *i.e.*, trace links are established, bidirectional, and consistent. A trace link is by default translated into a propositional formula as an equivalence. Their consistency is ensured by the DSL and, when some variability is selected to be checked, it is first checked whether it is traced. If

ϕ_{FM} and ϕ_{MTVM} are free of such individual inconsistencies, we can proceed to the variability consistency checking between them.

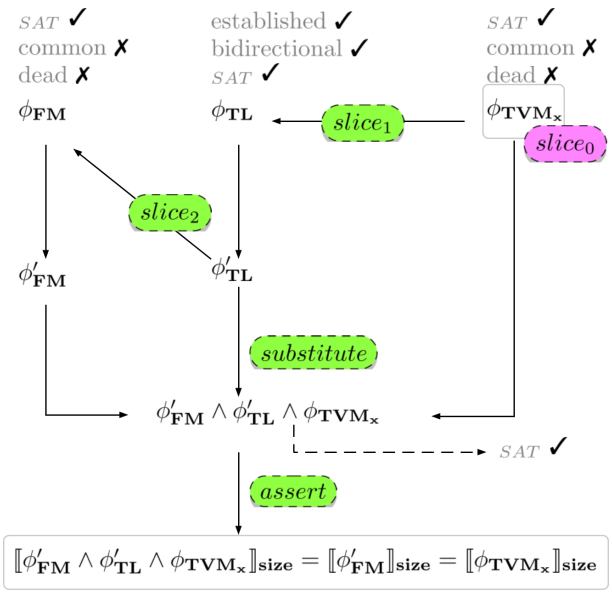

Figure 3: Proposed consistency checking method

Slicing. The originality of our method lies in the fact that we can select a single TVM, *i.e.*, ϕ_{TVM_x} as in Figure 3, or a subset of them from the MTVM, so to check the consistency of their variability against the specified variability in an FM. This selection corresponds to the first slicing step, *i.e.*, $slice_0$ in Figure 3, which is manual in our method. The consequence of selecting a single TVM instead of the whole MTVM is that the *initial checking* has to be done only for the selected ϕ_{TVM_x}, and the Assumption 1 about trace links must be met only for this TVM.

In the second step, *i.e.*, $slice_1$ in Figure 3, we use the ϕ_{TVM_x} to simplify the formula for trace links ϕ_{TL} by selecting only those trace links that are relevant for the ϕ_{TVM_x}. As a result the new formula for trace links ϕ'_{TL} is generated (*cf.* Figure 3). Further, we slice the FM, *i.e.*, $slice_2$, using the ϕ'_{TL} relevant trace links. The result is a new smaller formula ϕ'_{FM}, which contains only the relevant features for the *vp*-s and *variants* in ϕ_{TVM_x}, against which they should be checked.

Slicing an FM is an operation that has recently drawn attention in the SPL community. In the literature, there are already some well defined and validated algorithms [3, 18]. While we have experimented with them, we used a new slicing algorithm based on clause selection in a conjunctive normal form formula because of the trace links, as will be explained in the following. For example, let us suppose that we want to check the consistency of $\phi_{TVM_{search}}$ (*cf.* Eq. 3) against its specification in ϕ_{FM_g} (*cf.* Eq. 1), *i.e.*, the step $slice_0$. By applying $\phi_{TVM_{search}}$ to slice ϕ_{TL_g} (*cf.* Eq. 7) we get the new formula ϕ'_{TL_g}, *i.e.*, during the step $slice_1$, which keeps only those clauses that contain the *vp*-s and *variants* that are in $\phi_{TVM_{search}}$. So, the generated ϕ'_{TL_g} contains only the underlined clauses in Eq. 7. As we can see, by clause selection we keep the links to feature names, *i.e.*, to the features g, s, dfs, and bfs. Thus,

$$\phi'_{TL_g} \subseteq \phi_{TL_g} = (g \leftrightarrow vp_{root}) \wedge (s \leftrightarrow vp_s) \wedge \big(dfs \leftrightarrow v_{dfs}\big) \wedge \big(bfs \leftrightarrow v_{bfs}\big) \quad (9)$$

Similarly, by applying the new formula ϕ'_{TL_g} to ϕ_{FM_g} (Eq. 1), we select only the relevant clauses for the features in these trace links, *i.e.*, only the underlined clauses of ϕ_{FM_g}. Thus, the slice ϕ'_{FM_g} is:

$$\phi'_{FM_g} \subseteq \phi_{FM_g} = g \wedge (t \leftrightarrow g) \wedge (w \leftrightarrow g) \wedge (s \rightarrow g) \wedge (s \leftrightarrow (dfs \vee bfs)) \wedge (\neg dfs \vee \neg bfs) \quad (10)$$

It must be noted that the existing slicing algorithms cannot be applied to slice the trace links. Basically, the existing algorithms consist in eliminating the unselected variables in a propositional formula. But as we need the bidirectional relationship of a selected variable (*i.e.*, a *vp* or variant) to another unselected variable (*i.e.*, a feature), we have to apply clause selection instead of variable elimination. This is the main reason why we came up with the new slicing algorithm. Consequently, except for slicing trace links, slicing the FM itself can be done by previously proposed algorithms, which have show good scalability on larger scale FMs [3], compared to our slicing algorithm.

Substitution. After the slicing steps, the formal consistency rule (*cf.* Eq. 5) becomes:

$$\phi'_{FM} \wedge \phi'_{TL} \equiv \phi_{TVM_x} \wedge \phi'_{TL} \quad (11)$$

In essence, this consistency rule is applicable based on the substitution theorem [17] for propositional formulas. Therefore, checking consistency between ϕ'_{FM} and ϕ_{TVM_x} by using the Eq. 11 is equivalent to checking the following formula (*cf.* Figure 3):

$$\phi'_{FM} \wedge \phi'_{TL} \wedge \phi_{TVM_x} \quad (12)$$

Thus, we apply the substitution directly:

$$\phi'_{FM(f)} \wedge (f \leftrightarrow vp)' \wedge \phi_{TVM_x(vp)} \quad (13)$$

Concretely for our Graph SPL example, this formula becomes:

$$\phi'_{FM_g} \wedge \phi'_{TL_g} \wedge \phi_{TVM_{search}} \quad (14)$$

Assertion. Just checking if the resulting formula after slicing in Eq. 12 is satisfiable is insufficient to determine whether ϕ'_{FM} and ϕ_{TVM_x} are consistent or not. From before, they are consistent when they have the same configurations. But, instead of comparing their configurations after they are generated we achieve this comparison while the formula in Eq. 12 is calculated. Specifically, the Eq. 13 indicates that Eq. 12 will generate only those models (*i.e.*, configurations) which are similar between ϕ'_{FM} and ϕ_{TVM_x}. When ϕ'_{FM} is consistent with ϕ_{TVM_x}, then they will have the same models with $\big(\phi'_{FM} \wedge \phi'_{TL} \wedge \phi_{TVM_x}\big)$. As a result, we can simplify checking to only comparing their number of configurations.

More exactly, we assert whether

$$[\![\phi'_{FM} \wedge \phi'_{TL} \wedge \phi_{TVM_x}]\!]_{size} = [\![\phi'_{FM}]\!]_{size} = [\![\phi_{TVM_x}]\!]_{size} \quad (15)$$

When this assertion is false then ϕ_{TVM_x} and ϕ'_{FM} are inconsistent.

Let us illustrate how this works on our example. Figure 4 shows the sets of configurations for $[\![\phi'_{FM_g}]\!]$, $[\![\phi_{TVM_{search}}]\!]$, and $[\![\phi'_{FM_g} \wedge$

$\phi'_{TL_g} \wedge \phi_{TVM_{search}}]\!]$. In this case, $\phi_{TVM_{search}}$ is consistent

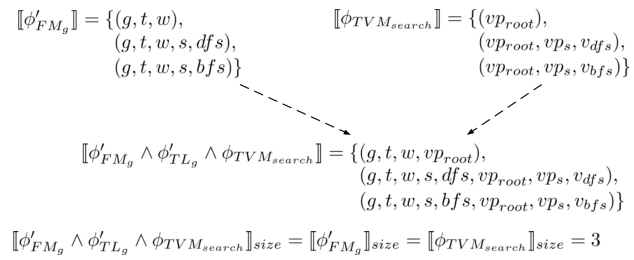

Figure 4: Consistency checking by asserting the number of configurations

against the ϕ'_{FM_g} as the assertion is true. As one can see, the bidirectional trace links ensure to generate from Eq. 12, respectively Eq. 14, only those configurations that are similar between the ϕ_{TVM_x} and ϕ'_{FM}.

As another example, let us suppose that the vp_s in $\phi_{TVM_{search}}$ has an Or relation between its variants (*cf.* Table 1), *i.e.*,

$$\phi_{TVM_{search}} = vp_{root} \wedge (vp_s \rightarrow vp_{root}) \wedge \big(vp_s \leftrightarrow \big(v_{dfs} \vee v_{bfs}\big)\big) \tag{16}$$

Figure 5 shows the assertion step between $[\![\phi'_{FM_g}]\!]$, the new formula $[\![\phi_{TVM_{search}}]\!]$, and $[\![\phi'_{FM_g} \wedge \phi'_{TL_g} \wedge \phi_{TVM_{search}}]\!]$. In this case ϕ'_{FM_g} and $\phi_{TVM_{search}}$ are inconsistent as they have different sets of configurations, with 3 and 4 number of configurations, respectively.

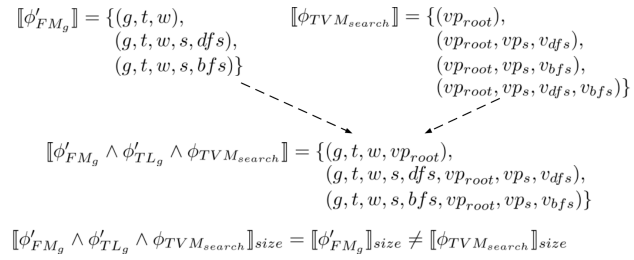

Figure 5: Example of inconsistency detection

For checking the next TVMs, *e.g.*, $\phi_{TVM_{basic}}$ (*cf.* Figure 2), we repeat the same steps in our method except the initial checking for ϕ_{FM}, which is unnecessary. Instead of checking a single TVM, we can select for checking a set of TVMs until a total checking, *i.e.*, in case the whole specified variability is implemented. In this way, the form of variability consistency we defined can be performed as soon as it is addressed, thus meeting challenge *C3* on *early* consistency checking.

Handling 1 – M trace links. So far we considered that the mapping between features in the FM to the *vp*-s and *variants* in MTVM, respectively between their slices, is 1 – to – 1. When their mapping is 1 – to – m, the assertion step becomes insufficient. To illustrate the issues related to this mapping, let us suppose that the vp_{search} (*cf.* Figure 2) in $\phi_{TVM_{search}}$ has an implemented technical *vp* $tvp_{\underline{n}one}$ that we have documented as in Figure 6. Its functionality consists in coloring the graph with a green ($v_{gr\underline{e}en}$) or blue ($v_{\underline{b}lue}$) color instead of performing a search. This technical *vp* is not specified at the specification level, *i.e.*, at the FM in Figure 1, but it is implemented as an alternative variant with v_{dfs} and v_{bfs}.

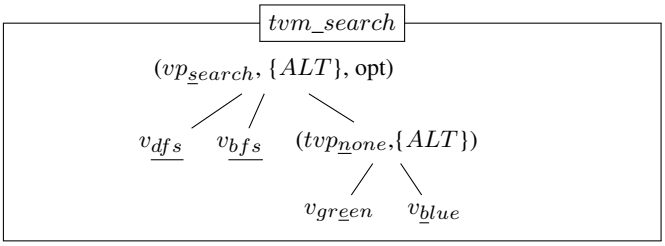

Figure 6: The `tvm_search` with a technical *vp* (*cf.* Figure 1)

The propositional formula for this new TVM is (*cf.* Table 1):

$$\begin{aligned}\phi_{TVM_{search}} = vp_{root} \wedge (vp_s \rightarrow vp_{root}) \wedge \\ \big(vp_s \leftrightarrow \big(v_{dfs} \vee v_{bfs} \vee tvp_n\big)\big) \wedge \big(\neg v_{dfs} \vee \neg v_{bfs}\big) \wedge \\ \big(\neg v_{dfs} \vee \neg tvp_n\big) \wedge \big(\neg v_{bfs} \vee \neg tvp_n\big) \wedge \\ (tvp_n \leftrightarrow (v_e \vee v_b)) \wedge (\neg v_e \vee \neg v_b)\end{aligned} \tag{17}$$

It is common to consider that the technical *vp* and its variants, *i.e.*, tvp_n with v_e and v_b, respectively, should be traced to the feature `Search` in Figure 1. In this case, the slice of trace links ϕ'_{TL_g} corresponds to:

$$\begin{aligned}\phi'_{TL_g} \subseteq \phi_{TL_g} = (g \leftrightarrow vp_{root}) \wedge \\ (s \leftrightarrow vp_s) \wedge \big(dfs \leftrightarrow v_{dfs}\big) \wedge \big(bfs \leftrightarrow v_{bfs}\big) \wedge \\ (s \leftrightarrow tvp_n) \wedge (s \leftrightarrow v_e) \wedge (s \leftrightarrow v_b)\end{aligned} \tag{18}$$

For this new TVM (*cf.* Eq. 17) and its 1 – to – m trace links (*cf.* Eq. 18), the assertion step is given in Figure 7.

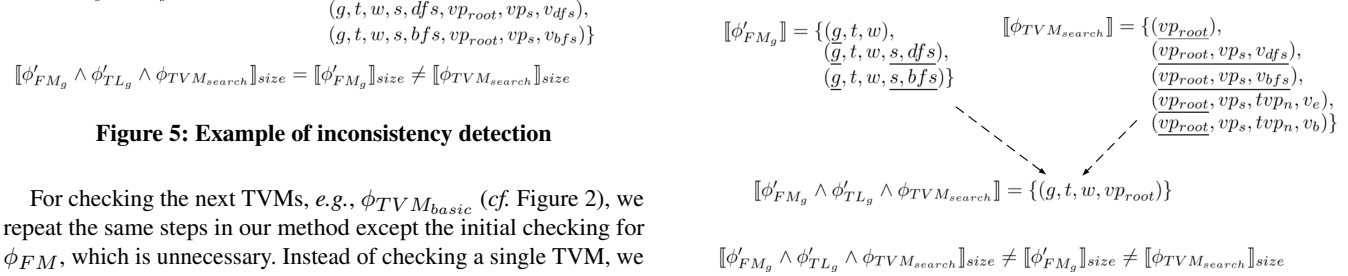

Figure 7: False inconsistency detection for 1 – to – m links

Although the implementation of v_{dfs} and v_{bfs} are consistent to features `DFS` and `BFS` in Figure 1, the assertion step in our method shows that they are inconsistent. The reason is that, when we trace the technical *vp* to the same parent feature, we dismiss the relation logic between its variants, *e.g.*, the alternative relation between v_e and v_b is not considered.

In Figure 7, by comparing the underlined parts between $[\![\phi'_{FM_g}]\!]$ and $[\![\phi_{TVM_{search}}]\!]$ we can see that the intersection between each

feature configuration, and *vp*-s and *variants* configuration is the set of configurations that shows their consistency. This example reveals that, if for each configuration $c_f \in [\![\phi'_{FM}]\!]$ and $c_{vp} \in [\![\phi_{TVM_x}]\!]$ their intersection $c_f \cap c_{vp}$ is not empty, then ϕ'_{FM} and ϕ_{TVM_x} are consistent. But, instead of comparing the configurations we need a solution at the formula level.

Among the possible solutions, the option to not trace the technical *vp*-s, *i.e.*, the unspecified variability, cannot be considered because they can invalidate the other *vp*-s themselves, which have an 1 – to – 1 mapping. A feasible solution, left for future work, seems, first to trace these technical *vp*-s and then to remove them by slicing the TVM_x. From the documentation of variability it is easy to recognize when a *vp* is technical. In this way we bring back the 1 – to – 1 mapping and our method is fully applicable.

In our method we do not include checking for dead or false-optional features, respectively *vp*-s and *variants*, between both levels. To check for them would require a complete implementation of the specified features, and a total consistency checking of variabilities.

4 EVALUATION

4.1 Implementation

We implemented the slice-substitute-assert method using the Scala language. The prototype implementation is publicly available at https://github.com/ternava/Variability-CChecking.

Initially, the FM of an SPL and its respective TVMs at the implementation level are converted to propositional formulas, in conjunctive normal form (CNF). Further, they are encoded in DIMACS CNF format and analysed using SAT techniques with the SAT4j solver [20].

It first consists in checking the validity, dead, and false optional features or *vp*-s and *variants* within a single FM or TVMs, respectively. Then, it performs finding and counting the number of valid configurations for a variability model or a sliced model. The slicing steps are implemented by using the clause selection algorithm on CNF formulas, but other slicing algorithms can be used too. It would be interesting to compare their usage and performance in real SPLs.

Our method only requires to obtain the propositional formula for the FM, the implemented variability that is documented in TVMs, and then establishing their trace links. We used FeatureIDE [31] for converting the whole FM to propositional formula, although it is not integrated in our implementation.

In our implementation, the specified variability in terms of features in an FM is used as a reference model against which is checked the consistency of the implemented variability in terms of *vp*-s and *variants* in TVMs.

4.2 Applications

Case studies. We evaluate our method by using it for check the consistency of variability in four case studies: Graph SPL [21], Arcade Game Maker SPL [1], Microwave Oven SPL [12], and JavaGeom [2]. The domains of the first three case studies are quite well understood and used by the SPL community. We implemented each of them with the Scala language, they are publicly available at https://github.com/ternava/Variability-CChecking. The fourth case study, JavaGeom, is an open source library implemented in Java. It is a feature-rich system for creating geometric shapes, which has been used in our previous work on traceability between specification and implementation levels [30]. The interoperability between Java and Scala enables us to use the DSL in JavaGeom as a Java-based system.

Table 2: Number of implemented features (*F*-s), *TVM*-s, and *vp*-s with *variants* for each case study

Case Study	*F*-s	Impl. *F*-s	*TVM*-s	*vp*-s with *v*-s
Graph	19	16	3	4 with 12
Arcade Game M.	27	26		8 with 17
Microwave Oven	26	23	8	9 with 20
JavaGeom	110	110	11	199 with 269

We used these different case studies to evaluate our method regarding the types of features, respectively *vp*-s, that can be checked, *i.e.*, *mandatory* and *optional* *vp*-s, as well as *alternative* and *or* relation logic between *variants*. Specifically, we evaluated whether the method is capable to detect the inconsistencies successfully in all these cases. For example, in the Microwave Oven SPL and JavaGeom we experimented with the technical and nested *vp*-s of any type, while in Graph SPL with *vp*-s that are implemented as a refactoring form of the specified variability.

The implemented variability was documented in each case study, using our DSL. In Table 2 is shown their respective number of TVMs with the *vp*-s and *variants*. In all case studies, a TVM has at least one *vp* with its *variants*. In the first three SPLs, we did not implement all features. First it is common that some variability is implemented later. Second, the SPLs can be evolved during the time with new features. Despite that some features are unaddressed, we could check the consistency for only that part of the implemented variability, showing the partial checking capability of our method.

Evaluation process. We performed the evaluation process in two stages. First, we checked the consistency of variability by selecting TVMs one by one, and then we were selecting a subset of them. In each stage, we analysed first the TVMs with 1 – to – 1 trace links and then those with 1 – to – m links.

The first inconsistencies that can be reported are about trace links. When we select a TVM, or a subset of them, if the trace links are not well-established, *i.e.*, the Assumption 1 is not met (*cf.* Section 3), an inconsistency meaning that the consistency of the selected variability cannot be checked without its trace links is reported. If trace links are well-established, the FM and TVM are checked about their self-consistencies of variability, then the consistency checking between them is performed using our prototype implementation.

Single trace links. We selected several TVMs in each case study, with the aim to assess our implementation with different types of *vp*-s. When the variability was implemented in the right way and the mapping was 1 – to – 1 to the feature in FM, then we could check successfully their consistency. For example, when an *optional* feature is implemented as an *optional vp*, their checking reports a success. Consistency was also reported when a *vp* was a refactoring form of a group of features in FM, *e.g.*, when a group of *or* related features are implemented as *optional variants* in a *vp*.

Similarly, we selected the TVMs that have different types of *vp*-s, but now when the variability is not implemented properly. For example, when an *alternative* group of features was implemented as an *or* group of *variants* in a *vp*, or a *mandatory* feature was implemented as an *optional vp*. In such case, their inconsistency was reported as a difference between their numbers of configurations.

Then, within a single case study, we selected different subsets of TVMs to check together the consistency of their variability to the features in FM. When each individual TVM in the subset was consistent, we could check that together they are consistent too. Further, we combined a consistent TVM, or a group of them, with at least one inconsistent TVM and we checked their variability together. As was expected, we got an inconsistency, but now it is part of the all selected TVMs, or their respective sliced FM.

In Figure 8 is shown the detection of an inconsistency for two TVMs in Microwave Oven SPL. All kinds of inconsistencies are reported as a difference between the number of *vp*-s and *variants* configurations in TVMs and the number of feature configurations in the sliced FM. Their configurations are also made available for any further comparison.

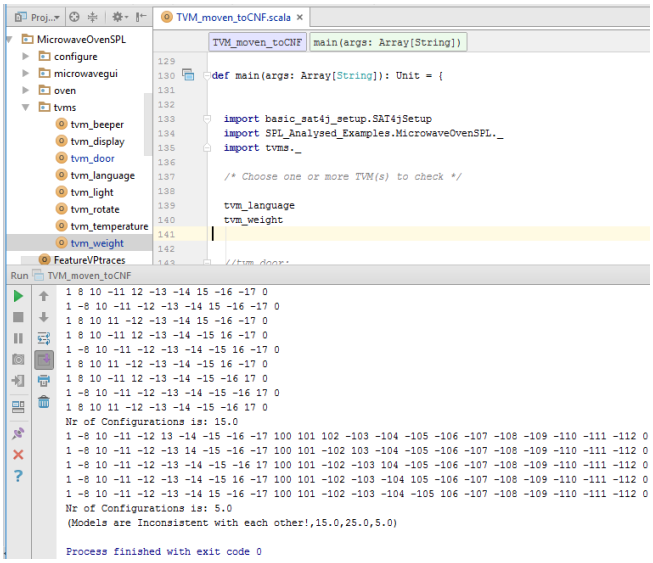

Figure 8: The detected inconsistency for two TVMs

Multiple trace links. Our current implementation supports tracing a feature to more than one *vp* or variant. It means that some *vp*-s are technical or nested, which are not modeled in the specification level but we consider them during the consistency checking of variability. We applied our consistency checking method when a technical *vp* and its variants are traced directly to the same specified feature in the FM (*cf.* Figure 6). Under these 1 – to – m trace links, we did similar evaluations with different types of *vp*-s. However an inconsistency was always reported even when the other *vp*-s where consistent. As a step toward a solution, we propose to slice the selected TVM by removing the technical *vp*-s and keeping only *vp*-s and *variants* that have single links to features in FM (*cf.* Section 3). Currently, this extension is under implementation.

Execution time. By selecting a single or a subset of TVMs at a time, the consistency checking is made possible early in the development process, *i.e.*, as soon as the specified variability is implemented and documented. But, we also expected that the consistency checking time for the resulting sliced formulas should be smaller. As for comparison, we recorded the execution time for some TVMs in Microwave Oven SPL, as it has most of the *vp*'s types. The resulting data are given in Table 3. The measurements are performed on a PC with 2.50 GHz and 4 GB RAM, on Windows 10.

We selected for checking three TVMs from the Microwave Oven SPL, tvm_{lang}, tvm_{temp}, tvm_{weight}, with 7, 7, and 3 *vp*-s and *variants*, respectively. We then checked them together. Execution times in each case are reported in Table 3. For example, for tvm_{lang} we measured the execution time for the sliced FM' (.002), the TVM itself (.007), and the time for checking their consistency (.005). The execution time for checking the validity of the whole FM with 26 features and 720 configurations took around 0.13 seconds (it is not shown in the Table). As we expected, the execution times for checking the consistency of partial variability compared to validity of only the FM is smaller. This may indicate that the number of features and *vp*-s with *variants* tend to be smaller, but we did not measure how much and for what number of *vp*-s and *variants* in TVMs that are checked.

The slicing time of trace links and FM contribute also to the consistency checking time of a TVM. We did not measure it as it is a property of the slicing algorithm, and requires extra validation (see Section 6).

5 RELATED WORK

Consistency checking of variability models and implementations is a topic of prime importance since the emergence of SPLE [26]. In the following we discuss works related to our approach according to different aspects.

Metzger et. al [24] are among the first that propose to check consistency between the variability at specification and implementation levels. In their approach, the specified variability is modeled in an orthogonal variability model (OVM) [25] in terms of *vp*-s and *variants* while the implemented variability is represented in an FM in terms of features. Although not explicitly, the mapping between *vp*-s and features is 1 – to – 1. Basically, they check when cross-tree relations of features in FM or *vp*-s in OVM may cause inconsistencies between each other. Unlike this approach, we advocate that the implemented variability is better captured by a forest-like structure, instead of a hierarchical structure that can be modeled by an FM. Further, in our approach the *vp*-s are not merely abstractions, as they are tags to the existing variability in core-code assets, which can be implemented by different variability implementation techniques.

Le at. al [19] propose to check consistency of variability between features in an FM and preprocessor directives in C, their mapping being 1 – to – 1 and variability being implemented by only a single technique. Differently from us, they check the consistency of variability at once, considering that all features are implemented. Another major difference is that they propose to extract the variability information from the core-code assets, *i.e.*, the preprocessor directives, and to rebuild the feature model which variability should be checked against the existing FM. On our side we use a DSL to

Table 3: Times for consistency checking of *TVM*-s compared to self-consistency checking of the FM, in the Microwave Oven SPL

	tvm_{lang}			tvm_{temp}			tvm_{weight}			$tvm_{\{lang,temp\}}$			$tvm_{\{lang,temp,weight\}}$		
P. Formula	FM'	TVM	CC	FM'	TVM	CC	FM'	TVM	CC	FM'	TVM	CC	FM'	TVM	CC
# configurations	6	6	6	5	5	5	2	2	2	30	30	30	60	60	60
Checking time (avg. in sec.)	.002	.007	.005	.0009	.008	.002	.001	.006	.002	.003	.015	.005	.008	.027	.013

document the implemented variability while it is implemented in a forward engineering process [30].

In some points, the approach by Tartler et. al [29] is similar to the one of Le at. al [19]. The main difference is that they check variability consistency in a specific software system – the Linux kernel. Similarly, they check the total consistency of variability and identify the dead or false optional features modeled in KConfig language and their respective implementation as preprocessor directives. From our part, we target any SPL that use traditional techniques for addressing the variability in core-code assets.

Vierhauser et. al [32] propose a tooled approach for checking the consistency between a variability model at specification level and other realization models, including the core-code assets. Unlike us, they define several consistency checking rules that are more about checking whether the variable units are addressed, and not if their relation logic is consistent with different models across the abstraction levels. As a result, they do not check whether the right mechanism or technique is used to realize the variability. As for their checking at code level, code artifacts are transformed into model elements and then checked. Another difference is that they propose an incremental checking, *i.e.*, whenever a developer makes a change it will be checked for consistency. Similarly, we propose to check the variability as earlier as it is implemented, but not after every single change. We do not check if a feature is simply addressed, but if the relation logic between a set of features are consistent with their implementation.

6 DISCUSSION AND FUTURE WORK

Summary. In this paper, we proposed a method for checking automatically the consistency between specification and implementation variabilities early during the development process of an SPL. We handle the case when variability, *i.e.*, *vp*-s and *variants*, is implemented by different variability implementation techniques, and has n – to – m mapping to the specified domain features in an FM. The application of our prototype implementation on four case studies, with different types of *vp*-s, show that our method can be applied successfully to check the consistency of variabilities between specification and implementation level as early as possible during the development process.

On Challenges. In our context, variability consistency can be checked successfully whenever features and *vp*-s with *variants* have a single mapping, whereas it becomes harder when their mapping is 1 – to – m, *i.e.*, with multiple trace links. The difficulty to check consistency under multiple links lies in the fact that it is ambiguous how to trace the technical *vp*-s and *variants* that do not have a single mapping to some features in the FM. We evaluated our method by tracing them at the same feature as their parent *vp* (*cf.* Section 3). Despite that some *vp*-s and *variants* with single links were consistent, an inconsistency was reported. However, since multiple mapping links between these levels of variabilities can be reduced to single mapping links, it is possible to use the same presented method for detecting the variability inconsistencies.

In this way, by checking the consistency between the specification and implementation variabilities, and considering their multiple trace links, we meet the challenges *C1.* and *C2.* (*cf.* Section 1). Also, instead of doing a complete checking, we select a TVM or a subset of them to detect their inconsistencies as early as possible during the development process, *i.e.*, immediately after some variability is addressed, thus, meeting the challenge *C3.*

On the scope of the contribution. Except for the relation logic between features or *vp*-s with *variants*, the consistency of variability can be also checked with regards to their binding time or evolution. Usually, these properties are modeled only at the realization level and checking for their consistency requires them to be specified at the specification level, too, *e.g.*, only in TVMs are document explicitly these two properties of *vp*-s (*cf.* Listing 1). This is also the main limitation of our approach. First, the variability models in both abstraction levels are required. Then, checking the binding time requires it to be available in each variability model. Besides, for an automatic checking, the binding time should be represented in the propositional formula in some way.

Instead of using our DSL, the implemented variability can be documented using other ways, such as a form of annotations [13]. When annotations are used, they should not only annotate the place where a variable unit is implemented in core-code assets, but also what is the relation logic between those units. This is related to our consistency rule. Specifically, we did not check only whether some variability is merely addressed. We checked whether the same software products that are specified can be derived from the core-code assets, within an SPL. Further, we supposed that the trace links are well-established. Otherwise, when trace links are the source of an inconsistency, we considered that it is not a variability inconsistency anymore. It is then an inconsistency regarding the addressing and mapping of variabilities, *e.g.*, when we try to check the consistency of an unimplemented variability or the mapping is mistaken.

In our work, variability is implemented by using a combined set of traditional variability implementation techniques (*e.g.*, inheritance, design patterns). In these techniques the *vp*-s are not explicit such as by using preprocessors in C. If we take preprocessors, they also can offer all kinds of relation logic between *variants* in a *vp*. Although, they offer a single binding time for *vp*-s compared to the case when several traditional techniques are used. As preprocessors are a form of annotations with variability information between *variants*, it could

be interesting to apply our consistency checking method when only this implementation technique is used.

When an inconsistency is detected, it is supposed to be handled, *i.e.*, finding its location and resolving it. Finding its location is quite trivial as we select a single TVM or a set of them for checking their consistency against their sliced FM. When the location is known, the variability inconsistencies as in Figure 5 can be resolved by changing the implementation technique for the *vp*-s, changing the way how the *variants* are implemented, or refactoring the specified features in the FM. In order to give help for resolving such inconsistencies at the implementation level, we have analysed around 20 variability implementation techniques by 16 characteristic properties, which are organized in form of a catalog. This catalog is available at https://github.com/ternava/Expressions_SPL/wiki while we currently study its validity.

Future Work. In the future, we plan to first raise some limitations of our consistency checking method. The *vp*-s and *variants* may have dependencies among different TVMs and the current prototype of the DSL for modeling implemented variability does not support yet these dependencies. It must be noted, the cross-dependencies of *vp*-s and *variants* in core-code assets are different from the cross tree-constraints of features in a feature model. Extending the DSL accordingly should improve our method and enables us to experiment with other case studies.

As we need both variability models, we have not yet studied precisely the scalability of our method. However the experimental results on our case studies and the short execution time of TVMs in Table 3 are promising indicators. In addition, we plan to use existing slicing algorithms to slice the FM and compare their performances. Choosing the best slicing algorithm will improve the scalability of our implementation.

Some ongoing work is currently tackling the implementation and analysis of the consistency checking method under 1 – to – m trace links when the technical *vp*-s are first removed by slicing the TVM. We expect these advances will complement the current method, so that we can more largely evaluate its practicality and usefulness in order to obtain insights to guide SPL practitioners.

REFERENCES

[1] *Arcade Game Maker Pedagogical Product Line*. http://www.sei.cmu.edu/productlines/ppl/.

[2] *JavaGeom case study*. http://geom-java.sourceforge.net/index.html.

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2011. Slicing feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 424–427.

[4] Felix Bachmann and Paul C Clements. 2005. *Variability in software product lines*. Technical Report. DTIC Document.

[5] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, 7–20.

[6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[7] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J Henk Obbink, and Klaus Pohl. 2001. Variability issues in software product lines. In *International Workshop on Software Product-Family Engineering*. Springer, 13–21.

[8] Rafael Capilla, Jan Bosch, and Kyo-Chul Kang. 2013. *Systems and Software Variability Management*. Springer.

[9] James O Coplien. 1999. *Multi-paradigm design for C++*. Vol. 53. Addison-Wesley Reading, MA.

[10] James O Coplien and Liping Zhao. 2000. Symmetry breaking in software patterns. In *International Symposium on Generative and Component-Based Software Engineering*. Springer, 37–54.

[11] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems*. ACM, 173–182.

[12] Hassan Gomaa. 2005. *Designing software product lines with UML*. IEEE.

[13] Patrick Heymans, Quentin Boucher, Andreas Classen, Arnaud Bourdoux, and Laurent Demonceau. 2012. A code tagging approach to software product line development. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 553–566.

[14] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co.

[15] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report. DTIC Document.

[16] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering* 5, 1 (1998), 143.

[17] Stephen Cole Kleene. 2002. *Mathematical logic*. Courier Corporation.

[18] Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. 2016. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, 60–64.

[19] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating consistency between a feature model and its implementation. In *International Conference on Software Reuse*. Springer, 1–16.

[20] Daniel Le Berre and Anne Parrain. 2010. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), 59–64.

[21] Roberto E Lopez-Herrejon and Don Batory. 2001. A standard problem for evaluating product-line methodologies. In *International Symposium on Generative and Component-Based Software Engineering*. Springer, 10–24.

[22] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the linux kernel variability model. In *International Conference on Software Product Lines*. Springer, 136–150.

[23] Andreas Metzger and Patrick Heymans. 2007. Comparing feature diagram examples found in the research literature. *Technical report, Univ. of Duisburg-Essen* (2007).

[24] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. 2007. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*. IEEE, 243–253.

[25] Klaus Pohl, Günter Böckle, and Frank J van der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

[26] Alcemir Rodrigues Santos, Raphael Pereira de Oliveira, and Eduardo Santana de Almeida. 2015. Strategies for consistency checking on software product lines: a mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 5.

[27] Klaus Schmid and Isabel John. 2004. A customizable approach to full lifecycle variability management. *Science of Computer Programming* 53, 3 (2004), 259–284.

[28] George Spanoudakis and Andrea Zisman. 2001. Inconsistency management in software engineering: Survey and open research issues. *Handbook of software engineering and knowledge engineering* 1 (2001), 329–380.

[29] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.

[30] Xhevahire Tërnava and Philippe Collet. 2017. Tracing imperfectly modular variability in software product line implementation. In *International Conference on Software Reuse*. Springer, In press.

[31] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[32] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. 2010. Flexible and scalable consistency checking on product line variability models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 63–72.