Automated analysis of two-layered feature models with feature attributes☆

Michael Lettner[a], Jorge Rodas[b], José A. Galindo[*,c], David Benavides[c]

[a] Tractive inc and University of Applied Sciences Upper Austria, Softwarepark 11, Hagenberg 4232, Austria
[b] Facultad de Ciencias de la Ingeniería, University of Milagro, Cdla. Universitaria Km 1 1/2 vía Km 26, Milaro, Ecuador
[c] Dept. Lenguajes y Sistemas Informáticos, University of Seville, Avda, Reina Mercedes s/n, Seville 4102, Spain

HIGHLIGHTS

- There is a lack of approaches to model application capabilities and platform features with support of complex constraints.
- We propose MAYA, an extension of the FaMa Feature model analyzer to cope with such requirements.
- We evaluate in a real scenario demonstrating that this approach is valid and underline its limitations.

ABSTRACT

Keywords:

Variability intensive systems

Feature models

Android

The proliferation of features and platforms in variability intensive systems, coupled with substantial technolo-gical progress, imposes several challenges for software developers and equipment manufacturers—in some cases referred as technical sustainability. For instance, in the mobile application domain, developers often need to know the requirements and limitations of their applications to be supported on a specific platform. Conversely, an equipment manufacturer is interested in knowing what additional features become accessible on the appli-cation layer when the or platform is being upgraded. To date, analyzing such interdependencies between specific feature and platform combinations is a tough problem, but important to solve. There are well-established ap-proaches in the literature to analyze variability–intensive systems using feature models. However, there is a lack of approaches to analyze application and platform features in multiple layers. In this paper we present a fra-mework towards the analysis of multi-layered feature models. First, modeling the two layers including their respective interdependencies. Second, a definition of operations that can be imposed on such models. We also provide a reference implementation for analysis of multiple layers. Finally, we present two empirical evaluations demonstrating the feasibility of the approach in practice.

1. Introduction

Software product line engineering is about handling multiple variants of a software system—often referred to as a family of products—by clearly defining what is common and what is different between them [1]. A well known approach to describe the common and variant parts of a software product line in terms of an hierarchical structure of features and relationship among them are *feature models* [2].

Mobile phones, with their hundreds of variations in features, are a perfect example for a software product line [3]. The frequent application feature changes coupled with the progress of technology make it necessary to provide automated mechanisms to handle them. Using state of the art product line techniques, features and their interactions can be analyzed using the so-called *automated analysis of feature models* [4].

In today's world, mobile phones are capable of things that were hardly imaginable a few years ago. Looking a few years ahead, it is not unlikely that people might consider today's cutting edge technologies as legacy technologies. Technical progress will continue to enable new type of applications. While this situation is usually a very pleasant one for the end user, not all parties experience advantages. From an application developer point of view, it is challenging to keep up with the latest developments [5]. The close relationship between application features and platform components that realize these features is hard to track. Considering near field communication (NFC) as an example, a developer who wants to incorporate this feature into an application must be aware that the required platform version must be 2.3 or higher

* Corresponding author.
E-mail addresses: michael.lettner@tractive.com (M. Lettner), jrodass@unemi.edu.ec (J. Rodas), jagalindo@us.es (J.A. Galindo), benavides@us.es (D. Benavides).

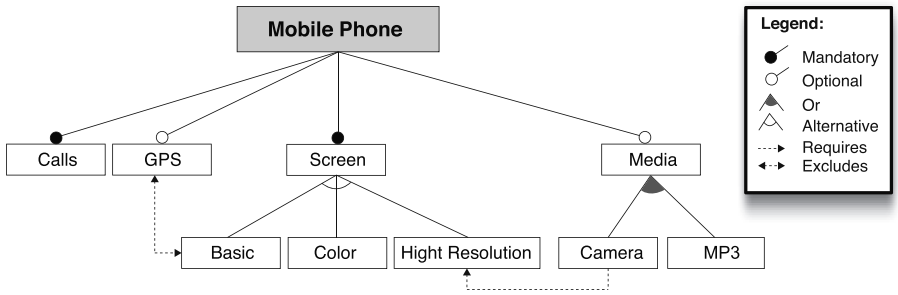

Fig. 1. A sample feature model of a mobile phone (from [4]).

on Android, or at least as high as iOS 11.x for third-party app support on iOS devices.

The proliferation of tablet computers illustrates another useful use case: Developers need to be aware of the consequences on existing applications when the lower layer changes, e.g., when an application shall be ported from a smartphone to a tablet computer or more generally, when a platform shall be upgraded to a new version. The different screen sizes and resolutions, the possible absence of a cellular radio or the increased amount of memory may all have positive or negative impacts on an application.

In the former scenarios, two elements are distinguished: the features of the application and the features of the platform. An examples of application feature is the text–to–speech capability while an example of platform feature is the physical audio output. As previously indicated, these features often are represented using features models. Since application and platform features are conceptually separated, their features can be modeled in two separated models organized in *layers*.

The problem at hand is the difficulty of tracing application features to platform features. This calls for a means to model application and platform features separately to reflect the possibly independent evolution of each layer. While that general idea is not new [6,7], little previous work has been done that focuses on the analysis of multiple-layer feature models. In the literature, there are proposals working with multiple layers of variability. For example Acher et al. [8], proposed the merging and slicing of feature models to scale feature model analysis. Also, there are works that rely on automated analysis of feature models to deal with the increasing number of features in the context of smart phones [3]. However, there is a lack of proposals to support the analysis of dependencies between different variability layers.

The proposal of this work is to provide a framework, where by specifying the desired features from a user's point of view, and specifying the provided features from a platform point of view, one can easily track the required platform features and get notifications about potential conflicts (such as the unsupported NFC feature on platform X in version Y). Conversely, by selecting a certain platform, the framework would be able to tell the user what application features are available for a particular platform configuration. The basic idea behind this proposal has been introduced in [9], but it did not go into detail on how such a system could be realized.

The problems mentioned above of easing the migration of an application from a platform to another is known in the literature as *technical sustainability* [10,11].

This paper's main contribution is a two-layered feature model framework that includes application features of a system on a top layer and platform features on a bottom layer. A detailed dependency mapping between these two to enable automated analysis is provided. We also offer a reference implementation. Finally, two real-world empirical evaluations are presented to show the applicability and need for the proposal.

The remainder of this article is organized as follows: Section 2 provides background information on terms and concepts used throughout the work. In Section 3 we present some typical challenges this work is addressing. Section 4 introduces the proposed solution and specifies the required model assets and type of relations. Operations that define what questions the system will be able to answer are defined in Section 5, and implementation-specific details are covered in Section 6. Case studies of two real world examples in Section 7 demonstrate the applicability and benefits of the approach, before Section 8 compares our approach to related work. Sections 9 and 10 conclude with a summary and outlook.

2. Preliminaries

In this section we present the background information:

2.1. Feature modeling

According to Clements and Northrop [1]: A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment and that are developed from a common set of core assets in a prescribed way. Back in 1990, the FODA (feature-oriented domain analysis) feasibility study by Kang et al. [2] introduced the concept of *feature modeling*, which remained to be one of the major research areas in product line engineering. They also defined a feature to be "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system".

A *feature model* itself allows to represent all products of a product line in terms of features and relationships in a compact way. A graphical illustration, often referred to as *feature diagram*, is depicted in Fig. 1.

Interestingly, it shows the features of a mobile phone example. We investigate the same domain in subsequent sections and will argue how we build upon and extend these classical examples.

Relationships. A feature model has a hierarchical structure, and one *root* element. A child feature can only be included if the parent feature is also included. Following types of relationships are typically distinguished [12]: *Mandatory, optional, alternative, or-relation*. In addition to that, so-called cross-tree relations can be used to express dependencies between features that are not in a parent-child relationship. The two most common types are *requires* and *excludes* relations.

In the example above, the features *Calls* and *Screen* are *mandatory* (i.e., these features must be included), while *GPS* and *Media* are *optional* (i.e., none of these features are required). When the parent feature *Screen* is selected, exactly one of its child elements (*Basic, Color, High resolution*) must be selected due to the *alternative* relationship. When *Media* is selected, either *Camera* or *MP3* (or both) must be selected (*or-relationship*). The cross-tree constraints specify that selecting the *Camera* feature requires to have a *High resolution* screen feature. Likewise, *GPS* and *Basic* screen features exclude each other (i.e., they cannot be selected together for one product).

Configurations. A *configuration* specifies one particular instantiation of the product line. It is characterized by specifying a set of *selected* and *removed* features. Configurations can be classified in to different categories, e.g. a *valid* configuration adheres to the defined relations and constraints. A *full* configuration contains every feature in either the selected or the removed lists. When not all features are contained in the selected/removed sets, it is referred to as *partial* configuration. A full configuration that only contains the set of selected features, while all remaining features are implicitly removed, is known as *product* [4]. In the context of extended feature models, configurations may also include the configuration of attributes [13] (i.e., determining a value for each attribute).

Extended feature model types. Next to the previously described *basic* feature models, a couple of extensions have been proposed. Schobbens [14] summarizes general semantics of feature diagrams common in literature. For instance, an *extended* feature model is characterized by features that can contain *attributes*. This allows for modeling specific properties of a feature such as costs or version information. At the same time, it enables complex cross-tree constraints that are dependent on the value of a certain attribute, e.g. *if Camera.resolution is bigger than a certain threshold X, then we require to have a High resolution screen*. Alternative terms for extended feature models are *attributed* or *advanced* feature models [15].

2.2. Automated analysis of feature models

Analysis is concerned with investigating feature models to extract valuable information, which could be of assistance for marketing or technical decisions. Using *operations*, different aspects of the models are analyzed. More than thirty different operations can be found in literature to date [4]. For instance, a *Valid product* operation will check whether a specified product represents a valid combination of features pertaining to a certain feature model. Likewise, the *Number of products* operation determines the number of valid products. Other operations aim at comparing feature models, e.g., regarding their resemblance.

The analysis of feature models is a tedious and error prone task. In order to help practitioners of feature modeling to extract information, different computer-aided mechanisms have been proposed. There are proposals based on specific algorithms, binary decision diagrams (BDD), SAT and CSP. Some of the most known tools can be found in [3,4,8,16]. In this paper we extended the open-source framework FAMA [17], to enable the reasoning of two layered feature models.

3. Challenges

In order to elaborate on what kind of problems this work is addressing, some typical challenges are pointed out below.

Challenge 1: Modeling application and platform features in software development to support technology evolution and introduction of new features. Application developers are constantly striving for improving their applications, possibly integrating new features to account for the best possible user experience. Adding features to an application brings up various questions: What are the platform features that are required in order to support that invention? Will the application still be compatible with currently supported target devices? Different operating systems and versions[1] and the sheer number of devices make it difficult to understand the consequences of such introductions.

Besides *integration of new features*, the fast pace of *technical innovation* is the primary motivation that calls for a way to easily track application features and their implementing platform components, as otherwise it is easy to lose track of new developments and platform evolutions.

We state *challenge one* as the need for a way to model application and platform features and their interdependencies among each other. Such a detailed model is a prerequisite to keep track of technology evolution and introduction of new features into applications.

Challenge 2: Configuration and automated analysis assistance for supporting technology evolution and introduction of new features. As stated in the previous challenge, introducing features and technology evolution raises questions about the consequences of such changes. For instance, what features does a platform must have to in order to support a new application feature? Is that new application feature compatible with existing platform? What is the impact of porting an existing application to a new platform? Finding answers to questions like this is key to understand the impact of adding features to applications, porting applications across different platforms, and many others.

Feature model analysis can help to extract valuable information from feature models [4]. Existing analysis operations usually focus on single layer feature models. Applied to models of application and platform features, *challenge two* is about defining operations that are capable of analyzing questions arising from changing these feature models. As a result, this will enable drawing conclusions about the impact of changes due to technological evolution and introduction of new features.

Challenge 3: Automated analysis / tool support. While analysis operations as outlined above enable drawing conclusions in general, the real benefit is in automating the analysis procedure. Quicker, repeatable results, applying the operations to models of larger scale etc, are just some of these advantages.

State of the art analysis tools are capable of running operations as defined in today's literature. By default, they would not be able to analyze models that include application features on one, and platform features on another layer (since they focus on a single layer instead).

Challenge three therefore is about seeking a way to adapt existing configuration and analysis tools to account for the multi layer nature (application features, platform features) of the models discussed in the previous challenge. This enables the use of these tools for automating the execution of the operations as mentioned in challenge two.

The challenges mentioned above are related to what is known in the literature as technical sustainability. Concretely, these challenges can affect the three dimensions of software sustainability mentioned in [10,11]: the human, the economic and the environmental dimensions.

4. Two-layered feature models

To cope with the previously highlighted challenge in Section 3, MAYA proposes to use a two-layered feature model approach to separate application and platform features. For that, we need to have detailed models in the two layers and relationships to connect them.

Fig. 2 shows an overview of our solution. The feature model on top (a.k.a. *top layer*) illustrates the variability inherent in an application, while the feature model at the bottom (a.k.a. *bottom layer*) depicts the variability present in the underlying device/platform, where the application is going to be executed. Given these two models, one can reason over different questions, such as finding the set of devices that can host a specified application. These reasoning operations are presented in Section 5.

[1] Android version market share distribution among smartphone owners as of September 2017 http://bit.ly/2iIitzz,https://developer.android.com/about/dashboards/index.html.

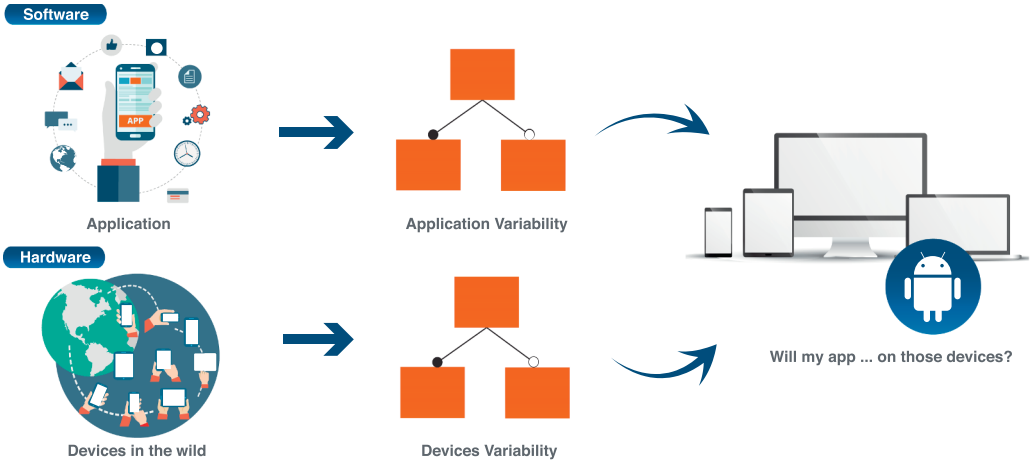

Fig. 2. Overview of the MAYA solution.

4.1. Top layer feature model

The top layer comprises application functionality such as *SMS, Call* or *Text Input* from a *user's point of view*. Note that a user here could mean an end user, a developer, or a marketing representative, depending on the context of use. Fig. 3 illustrates an example of such application functionality. This small subset of a feature phone (depicting its communication features) will be used as example for illustrative purposes throughout the remainder of the article. More complete scenarios are discussed in Section 7.

Features are organized in a tree-like hierarchical structure, and can be characterized by attributes (an *InternetAccess* feature might have a minimally required *datarate* specified). There are also cross-tree relations, i.e., dependencies from a feature in one category to a feature in a different category (e.g., *SMS requires Text Input*). Disregarding the mapping between different layers, features and relations can be modeled by means of state of the art product line engineering tools.

4.2. Bottom layer feature model

We refer to elements from the bottom layer as *platform features*. This rather general term reflects the combined hardware and platform aspect, i.e., it is not meant to depict hardware only. For instance, a *camera* feature in this model does not only consist of the actual hardware (lens, etc.), but has a software aspect as there need to be drivers and/or APIs to access the camera functionality. This combination allows to model additional information as attributes that could be either hardware-specific (a given maximum data rate of a communication component would be a good example), or platform-specific (the provided platform version (that could be minimally required for a certain feature) is an example for that). Fig. 4 depicts example elements for this layer.

Regarding granularity level, it is noted that the actual level used depends on how precise the analysis should be to fit one's needs. Instead of limiting somebody to a specific level of granularity, we would rather propose the workflow and means, which can then be adapted to meet one's requirements. The same holds true for application functionality (e.g., instead of *Internet Access* one may have actual *services, protocols, etc.*).

4.3. Attribute specification

Each feature pertaining to either of the two layers can be further described using attributes such as costs or version numbers. This additional information can be used to model more complex constraints, as mentioned in the next subsection. The following way of specifying attributes is inspired by [18]. An attribute specification usually contains a *name*, a *domain*, and a *value* [4]. We will also add a *nullValue*, a *defaultValue*, an *isInheritable* indicator and a *unit*. The combination of feature name and *attribute name* must be unique, e.g. WiFi.datarate. The *domain* declares the valid data range, e.g. Boolean, Integer, and possibly a range (e.g., [0.1024]). The *null value* specifies the value if the attribute's feature is not selected, and the *default value* declares the value if no separate value has been configured. If *isInheritable* evaluates to true, all child features inherit the same attribute definition (i.e., same name, domain, null and default value, unit), however, it does not affect the attribute's values. This mechanism allows convenient attribute specification for huge feature models. A *unit* will elaborate on the interpretation of the value, i.e. whether the number is quantified in meters, bytes, seconds, or others.

Regarding the computation of attribute values, each attribute can be of one of two types:

- *Basic attribute:* The value of this type of attribute is a simple value, i.e. it is directly assigned and does not depend on any other attributes.
- *Composite attribute:* A composite attribute is one whose value is composed of other attribute values, e.g. by calculating the sum or determining the minimum of a group of specified attributes. If the attribute is inheritable, it is sufficient to declare the desired group function (e.g., sum, max, min) and then the function will be applied to the attribute values of its selected child features. An example is an inheritable attribute *minApiLevel* for the root feature, which is calculated by determining the maximum value of all attributes named *minApiLevel* in any of the selected child features.

4.4. Inter-tree relationships

To model relationships across layers *inter–tree relationships* are used. Due to the nature of multiple layer feature models, common notations used in feature models must be extended to connect them. All defined inter–tree relationships are top-down, i.e. they connect a feature from the top layer with features from the bottom layer. In our proposal, we do not use a bottom-up relationship because we assume that there is an abstraction gap between the application and platform features. This is, the layers are decoupled enough that there are no software requirements for a hardware component (e.g. drivers). Figure 5 depicts example elements to model relationships across layers. A summary of the textual language used in presented in Table 1 and described as follows:

Table 1
A summary of currently supported language constructs that can be used to create inter–tree relationships in MAYA. This list may be extended as required to build more complex relationships between layers.

Construct	Syntax and example	Description
1:1	`top1` ***requires*** `bottomA` **example**: `SMS` ***requires*** `Cellular`.	The feature of the top layer SMS connection in the mobile example requires a feature of the bottom layer (Cellular in the example).
1:*n*	`top1` ***requires*** `bottomA` ***and/or*** `bottomB`. **example**: `Call` ***requires*** `GSM` ***or*** `UMTS`	An upper layer feature does not always require a single specific lower layer feature, but rather one or multiple features. The syntax to establish this connection defines the upper level feature name followed by the required features of the lower level model.
Constraint	`top1` ***requires*** `bottomA` ***and/or*** `bottomB` ***and*** [constraint]. **example**: `InternetAccess` ***requires*** `GPRS` ***and*** [`GPRS.datarate` ≥ `InternetAccess.datarate`].	These restrictions can be combined both with 1:1 and 1:n relationships. The syntax to establish this kind of restrictions between layers defines the name and the upper layer feature together with the lower level features required as well as constraints on attributes

- *1:1 inter-tree relationships:* The simplest type of inter-tree relationships connect one feature on the top layer with one on the bottom layer. They resemble common cross-tree relations (such as *requires, excludes*), except that they connect features across trees (as opposed to connecting features within the same tree).
  Format: top1 *requires* bottomA (feature1 of top layer requires featureA of bottom layer).
  Example: AudioInput *requires* Mic.
- *1:n inter-tree relationships:* To account for more complex feature interaction patterns, a top-level feature does not always necessarily require a single specific bottom layer feature, but one (or more) out of a group of features. Therefore, the well known 1:1 relations are no longer sufficient to model such circumstances [19,20] and 1:*n* relations must be introduced. Similarly, the available parent-child relations within a feature tree, group relations such as *and, or, xor* shall be available to connect an application feature to multiple platform features.
  Format: top1 *requires* bottomA *and/or* bottomB.
  Example: The top-level *Call* feature requires either a *GSM* or *UMTS* bottom layer component (having both components is also possible).
- *Constraint-based inter-tree relationships:* While constraints are common in extended feature models, their application to inter-tree relationships enables modeling new aspects of a system. Note that constraints can be combined both with 1:1 and 1:n relations—effectively resulting in four different kinds of relations.
  Format: top1 *requires* bottomA (and/or bottomB) *and* [constraint].
  Example: The top-level feature *Internet Access* could be mapped to either *GPRS, UMTS, HSPA, WiFi* (among others). While modeling a VoIP application, it would be beneficial to specify a minimally required data rate to meet a certain quality of service. Therefore, when the building block *Internet Access* is selected for such a VoIP application, an inter-tree relationship constraint can be used to connect bottom and top feature attributes. The constraint (e.g., bottomA.datarate > = top1.datarate) can then be evaluated to see whether the requirements regarding datarate are met. In practice, requiring a minimum data rate of e.g. 300 kbit/s for the Internet Access feature can result in certain platform features not being able to satisfy the condition. For instance, the maximum data rate of GPRS is lower than the required 300 kbit/s, thus GPRS cannot be used for VoIP applications.

These type of inter-tree relationships are the ones we defined for MAYA—but are subject to extension for different applications.

5. Analysis operations

While the previous sections discussed problems regarding frequent technological innovations and how to approach them using multi-layer feature models, this section focuses on how MAYA defines operations that can be executed on these models. The derived results provide guidance for technical decisions or marketing strategies [21] and allow to study the consequences of different input configurations. Actual use cases will be demonstrated in Section 7.

Thirty different operations for the analysis of feature models have been identified in the literature [4]. Below we will build and extend on them by defining four major operations that operate on two-layered feature models. For the sake of simplicity, we will refer to them as O1 - O4 throughout this article.

Two *high-level top-down* (O1, O2) and two *bottom-up* operations (O3, O4) are defined so far. *Top-down/bottom-up* refers to its main application of use i.e., a top-down operation generally takes a top layer configuration as input, and derives results pertaining to the bottom layer as output (and vice versa for bottom-up operations).

High-level refers to the fact that each high-level operation might be comprised of multiple low-level operations.

5.1. Operations: Common input

All subsequent operations take a common input:

- *Top Layer Feature Model (TopFM):* Contains the features and cross-tree constraints of the application functionality layer.
- *Bottom Layer Feature Model (BottomFM):* Defines the features and cross-tree constraints of the platform layer.
- *Inter-Tree Relationships:* The highly relevant specification of inter-tree relationships (often referred to as *mapping* throughout the remainder of this article), which enables analysis spanning multiple layers in the first place.

Selected configurations must adhere to these model definitions. Any additional input which is specific to an operation is defined at the respective operation description below.

To illustrate the implications of our operations, we apply them on a simple example throughout the remainder of this section. The common input for this example is defined in Fig. 6.

Prerequisites. We require both top and bottom feature models to be error free, i.e. they do not contain any dead features. A feature is said to be dead if it cannot appear in any of the products. Such analysis can be done using existing single layer operations respectively [4].

5.2. O1: Platform capability analysis

Given a set of selected features for a concrete application, the *Platform Capability Analysis (O1)* identifies the minimally required platform features. This information can be used to understand the platform requirements to support the application. It addresses the

following main question: To enable a certain application, what capabilities are required by the platform in use?

By passing a full configuration of the top layer (the application), this operation determines a partial configuration of the bottom layer which identifies t he m inimally r equired p latform f eatures f or t he specified application. This partial configuration can be used as starting point for further configuration on the bottom layer which can be accomplished using state of the art techniques for single layer feature model configuration, e.g. like the ones supported by FaMa [22].

5.2.1. Input/output

Next to the input common to all operations (top feature model, bottom feature model, inter-tree relationships), O1 takes following input:

- *TopFM_Configuration*: The full configuration of selected application functionality.

Notice that all configurations used in MAYA are specified as *full configurations*, i.e. each feature of the feature model is either explicitly selected, implicitly selected (parent features of explicitly selected features, and connected features due to *required* relations), or implicitly removed (all remaining features). The configuration of attributes is also included in a full configuration. Together with the mapping model this top-level configuration is used as input. Then, the system responds with a configuration detailing the minimally required platform features.

The output of O1 is a *partial configuration* of the bottom layer, as well as constraints that the selected bottom layer features must meet. As the different types of inter-tree relationships need to be interpreted differently, the output is further separated into three categories. The categories reflect to which extent a feature is required on the bottom layer - i.e., whether it *must* be present, *must not* be present, or its attributes must be within a limited *attribute domain*. Features that are in neither of these categories *can* be freely selected.

- *Must-have:* A list of platform features that must be present on the platform layer in order to meet the specified application requirements.
- *Removed:* The features in these categories must not be selected on the bottom layer.
- *Attribute domains:* Constraint-based inter-tree relationships may affect the range of attribute values in order to meet the specified requirements from the top layer. If an attribute's range is limited due to the selection on the top layer, the attribute and its limited range is listed in this category.

In addition, the output states some *constraints* that must be met when further configuring the bottom layer. These constraints arise from inter-tree relationships and can be easily calculated from their syntax.

5.2.2. Example

Next, we apply O1 on our running example. The configuration of the input is:

TopFM_Configuration: $S = \{SMS, VoIP\}$

Therefore, two features are explicitly selected. Their parents (AppFeatures, Communication, Text, Voice) will be implicitly selected—so would any of the required features due to cross-tree constraints (InternetAccess, and its parent Data). As it is a full configuration, all other features from the top layer are implicitly removed. The full configuration therefore is shown in Fig. 7 where selected features are presented in gray and deselected in white:

The full configuration therefore is shown in Fig. 7, where *selected* features are illustrated in *gray* and the *deselected* features are illustrated in *white*:

In addition to the configuration of features, their attributes must be configured as well. Only one feature has an attribute defined in our example, and its value is assigned to be 300.

The corresponding output of O1 is:

```
Must-Have:
  PlatformFeatures, Radios, Cellular, Input

Removed:
  -

Attribute Domain Limitations:
  GPRS.datarate:  Integer[300 to MaxInt]
  WiFi.datarate:  Integer[300 to MaxInt]

Constraints:
  PlatformFeatures requires (GPRS or WiFi)
```

PlatformFeatures, Radios, Cellular and *Input* are components that are mandatory, thus not to be sacrificed. The set of removed features is empty, as there are no constraints that cannot be met (e.g., an alternative relationship on the bottom layer could result in one feature being removed, if the other is included in the set of must-have features). The bottom layer constraint requires that either *GPRS or WiFi* (or both) have to be selected.

The ranges of two attributes have been limited. Remember the inter-tree relationship that specified that the bottom layer features' datarate had to be higher than the *Internet Access* top feature's datarate: `InternetAccess requires (GPRS or WiFi) and [bottom.datarate >= top.datarate]`. Since the required datarate of InternetAccess has been configured to be 300 as input to O1, both GPRS and WiFi need to provide a minimum datarate of at least 300 which is expressed as attribute limitations in above example.

5.3. O2: Platform compatibility analysis

This high-level analysis operation can be used to compare the required platform features for a specific application (as retrieved in O1) with concrete platforms/devices, and determine their (in-)compatibility level. Such a categorization reveals features that are potentially incompatible or are missing to support the selected application. O2 addresses following main question: To what extent is the selected platform combination capable of supporting the desired application on the top layer?

By passing a full configuration of the top layer (the application) and a full configuration of the bottom layer (the platform), this operation determines a categorization of the bottom layer configuration which identifies the components and their level of (in-)compatibility with

regard to the specified application.

5.3.1. Input/output

In addition to the common input, following specific input for O2 is required.

- *TopFM_Config1*: The configuration of selected application functionality.
- *BottomFM_Config1*: The configuration of the components for the selected platform combination.

By executing O2 multiple times for different platforms (and/or versions), the question of compatibility among a wide variety of platforms can be answered.

The output of O2 categorizes the features of the provided bottom layer configuration into three distinct categories:

- *Compatible:* A list of platform features that are fully compatible with the specified application features. A feature is said to be compatible when it is required by the top layer and is present on the bottom layer, and meets a given constraint, if any. *Example:* Consider the inter-tree relationship *A requires X*, where A is part of the top layer configuration, and X is part of the bottom layer configuration. Then X is said to be *compatible*, since it is required by feature A, and provided on the bottom layer.
- *Missing:* The features in these categories are missing in the respective bottom layer configuration, although their existence would have been required to realize the desired top level functionality. *Example:* Consider the same inter-tree relationship *A requires X* from above. If X is not part of the bottom layer configuration, then X is said to be *missing*, since it is required by feature A, but not provided on the bottom layer.
- *Incompatible:* Features declared in this list are present on the bottom layer, but do not meet a given constraint, and therefore are said to be *incompatible. Example:* Above inter-tree relationship is extended by a constraint, such as *A requires X and [constraint]*. When A is a selected top level feature, and X a selected bottom level feature, the feature X is said to be *incompatible* if the constraint is not met.

5.3.2. Example

In the example below, we use the same input configuration as in O1 for the top layer, and provide a full configuration of the bottom layer in addition to that:

$TopFM\_Configuration$: $Selected = \{SMS, VoIP\}$

$BottomFM\_Configuration$: $Selected = \{Input, GPRS\}$

Two features are explicitly selected on each layer. Remember that their parents and any other required features (e.g. due to cross-tree constraints) are implicitly selected, while all remaining features are implicitly removed. The full configuration therefore is presented in Fig. 8.

The resulting output is as follows:

```
Compatible (= required and provided):
  Radios, PlatformFeatures, Input, Cellular

Missing:
  WiFi

Incompatible (= provided, but constraint not met):
  GPRS
```

Most of the features are *compatible* as is. WiFi on the other hand is listed as *missing*, meaning that the component would be required to realize all application functionality. GPRS is *incompatible*, as its datarate is declared as 128, while the corresponding inter-tree relationship requires a datarate higher than that of the InternetAccess feature, which is configured to be 300.

5.4. O3: Application functionality potential analysis

Similar to O1 (though bottom-up), this operation will find the corresponding application features that are enabled by a certain platform. However, it detects (a maximum number of) features that could theoretically be realized on the given platform. One use case would be to recognize functionality that could be tapped into by an application (such as a new text-to-speech feature), or early identify application features that are not supported on a particular device/platform. O3 addresses following main question: Given a particular platform, which features can be utilized in an application running on this platform?

By passing a full configuration of the bottom layer (the platform), this operation determines a partial configuration of the top layer, categorized into features which are *enabled* by the given platform, and features that are *removed* by the same and thus cannot be used for an application.

5.4.1. Input/output

Next to the feature models of both layers and the inter-tree relationships (as specified earlier), this operation takes following input:

- *BottomFM_Configuration:* The full configuration of features for a specific platform combination.

As an output, O3 delivers following information:

- *TopFM_Configuration_Max:* A top-level configuration containing the set of features that are potentially enabled by the given platform.
- *TopFM_Configuration_Removed:* A top-level configuration containing all the removed features, i.e. features that can definitely not be realized on the given platform.

Therefore, the output distinguishes between features which can be used by an application, and features which cannot be realized on the particular platform (e.g. due to missing or limited platform features).

5.4.2. Example

Applied to our running example, the full configuration of the bottom layer is shown in Fig. 9.

As a result, O3 retrieves:

```
Enabled:
  SMS, Text, Communication, ApplicationFeatures, Call, Voice

Removed:
  VoIP, InternetAccess, Data
```

The list of enabled features represents functions that can be tapped into by applications. On the other hand, the removed features cannot be realized in an application due to limitations of the platform. For instance, the VoIP feature would require InternetAccess top level feature—which has inter-tree dependencies on either GPRS or WiFi, but none of which is provided on the bottom layer.

5.5. O4: Platform migration analysis

Investigating potential conflicts of a platform migration is a prevailing problem at the core of O4. It addresses the following main question: How are existing application features affected by exchanging the underlying platform?

Given a particular application and the source and target platforms, the operation will highlight potential conflicts caused by such migrations.

5.5.1. Input/output

The platform migration analysis operation takes a full configuration of the top layer, and two different full configurations of the bottom layer as input.

- *TopFM_Config1:* The configuration of selected application functionality.
- *BottomFM_Config1:* The provided modules of the current (= source) platform in use.
- *BottomFM_Config2:* The provided modules of the future (= target) platform. Both configurations must adhere to the same feature model.

The result of the analysis classifies application features by different categories:

- *TopFM_ConfigUnaffected:* A list of compatible application features that are not affected by the platform migration.
- *TopFM_ConfigEnabled:* A list of application features that are enabled by the new platform. Enabled means that this feature can be used now on the new target platform (but was not available on the previous source platform).
- *TopFM_ConfigIncompatible:* A list of incompatible application features that are somehow conflicted due to the platform change. Conflicted in that sense means that the application feature can no longer be sufficiently implemented on the new platform, e.g. due to missing or limited platform features.

5.5.2. Example

The example below helps to illustrate this. The input configurations are:

TopFM_Config: S = *SMS*, *VoIP*

*BottomFM_Config*1

: S = *PlatformFeatures*, *Radios*, *Input*, *Cellular*, *GPRS*, *WiFi*

*BottomFM_Config*2

: S = *PlatformFeatures*, *Radios*, *Input*, *Cellular*, *GPRS*, *GSM*

In order to retrieve the full configuration, all implicitly selected features (i.e., parent features and required ones due to cross-tree constraints) are *added*, and all remaining (i.e. not selected) features are *removed*. These full configurations are presented in Fig. 10.

The outcome of applying O4 is:

```
Unaffected:
  SMS, Text, Communication, ApplicationFeatures, Voice, Data

Enabled:
  Call

Conflicted:
  InternetAccess, VoIP
```

Most application features would be unaffected by the platform change, meaning they could still be utilized after the change. Due to the added *GSM* capabilities in the target platform, the *Call* feature has been enabled on the top layer. Opposed to that, *InternetAccess* is now incompatible, since *WiFi* feature has been removed (and *GPRS* doesn't meet the attribute constraint of a minimum datarate of 300 kbps). As *VoIP requires InternetAccess*, it is added to the incompatible features list.

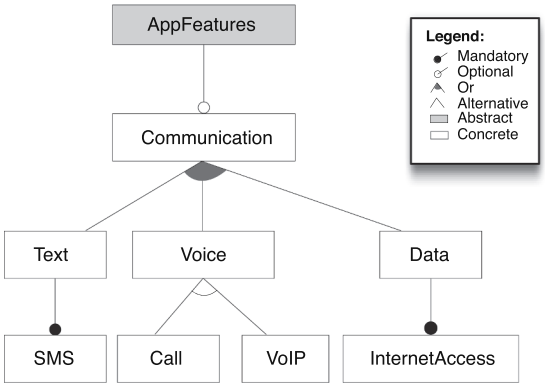

Fig. 3. A sample top layer, comprising application functionality.

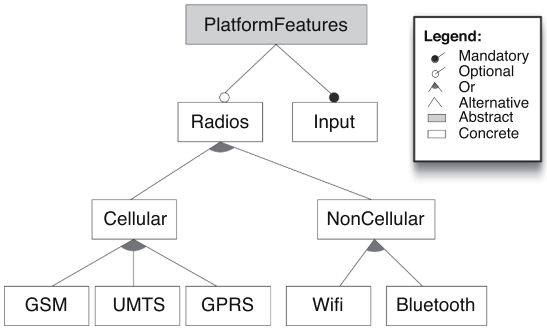

Fig. 4. A sample bottom layer, comprising platform features.

```
%Mapping
SMS requires GSM or GPRS or UMTS;
Call requires GSM or UMTS;
InternetAccess requires (GPRS or UMTS or HSPA or WiFi)
   and [bottom.datarate >= top.datarate];
AudioInput requires Mic;
TextInput requires (Keyboard or Touch);
TextOutput requires Display
   and [bottom.colorDepth >= top.colorDepth];
AudioOutput requires Speakers;
```

Fig. 5. Inter-tree relationships: illustrative example.

6. Implementation

To proof the validity of the MAYA approach, this section discusses the prototypic implementation which laid the foundation to evaluate the approach.

6.1. Tool selection

Tool support to automate some of the tedious and error-prone steps is essential for the approach to be beneficial and efficient. First, it was evaluated whether there are existing tools that could be adapted to our needs, or a new one had to be developed from scratch. Ideally, such a tool would be *highly customizable* to integrate *support for multiple layers*, which, to the best of our knowledge, is not provided by any existing automated analysis tool.

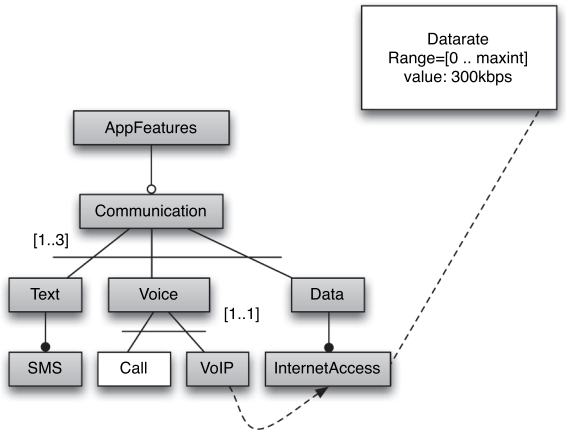

Fig. 7. Top full configuration used for O1.

While there are numerous feature modeling tools available, when it comes to automated analysis of feature models, tool support is more limited. The ones that are available focus on traditional reasoning operations, thus they operate on single layer feature models.

For MAYA, the main tool requirement was *extensibility*. The tool must be adapted in order to

- take multiple input models, pertaining to the proposed top and bottom layer,
- read and interpret the defined inter-tree relationships,
- extend and define reasoning operations that follow the multi layer operation definitions from Section 5,
- extend the solver in order to operate on the defined operations.

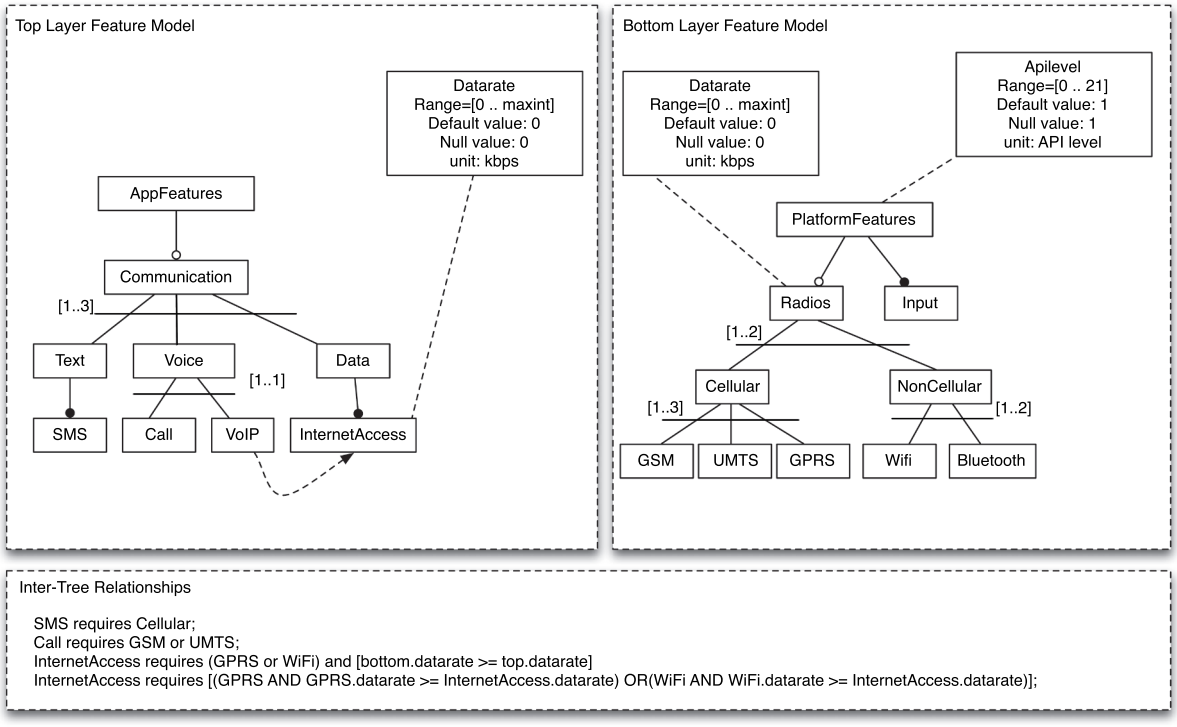

Fig. 6. Example used to illustrate the proposed operations.

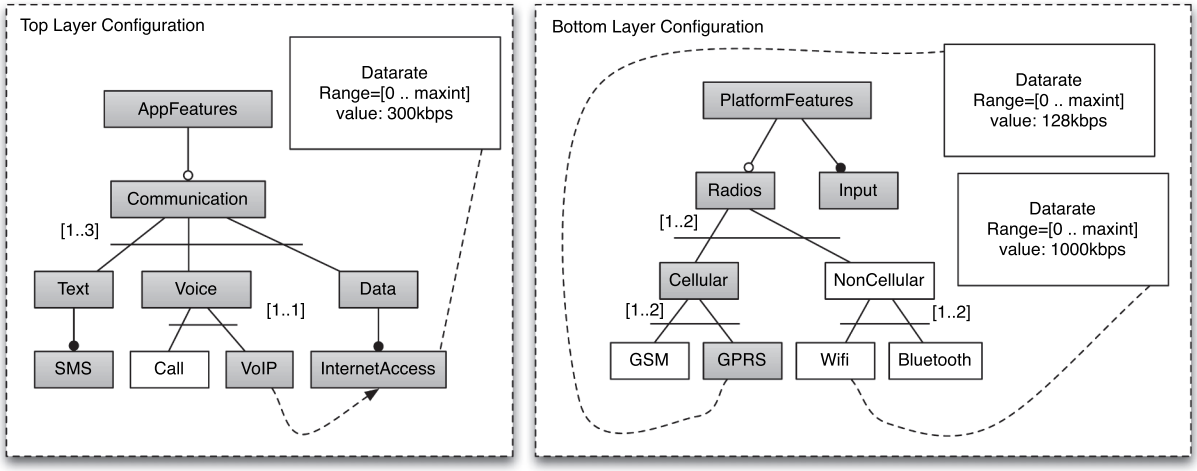

Fig. 8. Top and bottom full configurations used for O2.

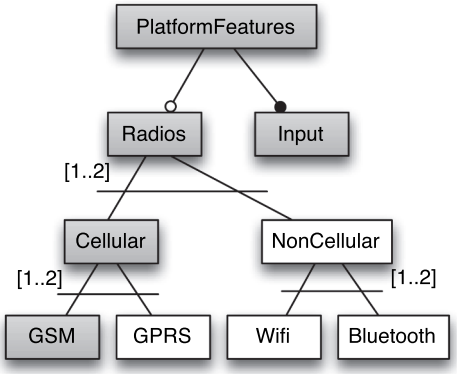

Fig. 9. Bottom full configuration configuration used for O3.

While most of the available reasoning tools offered no or very limited extensibility options, the extensible nature of FaMa [23] and its open source approach made it the ideal choice for MAYA.

6.2. Automated analysis

The MAYA solution performs the reasoning process in three steps. First, it loads the set of models (as text files) describing the problem. Second, it parses the text files and translates them into a concrete logic paradigm. Later, it applies an operation and determines the information requested by the user. The subsections below discuss the process taken by MAYA in more detail. The MAYA solution is shown in Fig. 11

MAYA relies on a concrete serialization for feature models based on the FaMa feature model language. This extension add attribute inheritance between child and parent features support among other features such as support for undetermined ranges when defining attributes.

Fig. 12 shows the syntax for specifying the relationships and cardinalities of the model represented in the feature diagram from Fig. 3. Next to the feature model, *constraints* and *attributes* are specified in a similar text-based format. This format starts by describing the feature model tree. Every line starts with the name of a parent feature in a relation and, after the colon, the children feature names. Note that, depending on the kind of relationship to represent different text structures can be used. For example, if the child is an optional feature it will be presented between square bracelets. However, if is a set relation, the square bracelets are used for the cardinality and the curly bracelets for specifying the features in the group. Finally, if no symbol is used it represents a mandatory feature.

After specifying the feature tree, there is a section containing the set of cross-tree constraints. in this section, more complex constraints can be described. For example, constraints containing relational operators such as *"VoIP REQUIRES InternetAccess OR Data"*. The way an attributed is defined is {feature name}.{attribute name}:{attribute type},{default value},{null value}, Inheritable: {true or false depending is the attribute is inheritable}.

Additionally, the mapping between the two layers is provided in machine-readable text representation—cf. Fig. 13. This representation can contain simple and complex constraints, as described in Section 4.4.

6.2.1. Conversion of feature models into CSPs

In order to answer the operations defined in the previous section, the input models need to be transformed into so-called *constraint satisfaction problems (CSPs)* before they can be solved. As a reasoning mechanism, MAYA relies on *ChocoSolver*, which is a Java implementation of a CSP. The CSP generated is defined as the tuple

$< F, A, C, IMC >$

where:

- F is the set of variables comprising of all boolean features—either from the top layer feature model, or the bottom model. Being F_i the variable representing feature *i*.
- A is the set of variables representing the attributes. Those variables can be defined as real or integer variables. Thus, the A_j variable represents the attribute *j*.
- C is a set of constraints containing *i*) the constraints between attributes defined in the feature models and; (ii) a set of "virtual constraints" between each feature and their attributes. Those constraints will set the value of an attribute depending on the value of their associated features. For example, if feature F_i is associated with attribute A_j, then the constraint to be added will be *if* F_i *equals 1, then* A_j *equals default value, else* A_j *equals null value*.
- IMC (inter model constraints) is a set of constraints representing the *requires* and *excludes relationships* between the different layers.

Implementation-specific details and limitations. The MAYA architecture is shown in Fig. 14. The core artefact provides a set of interfaces to communicate with the different reasoners and metamodels. We highlight the (i) readers and writers that allow MAYA to use different set of textual and graphical representations as input or output. For example, MAYA is able to read the *splx* format, which allows us to load any of the models in the splot repository [16] as input; (ii) different metamodels, by default MAYA takes as input two different types of

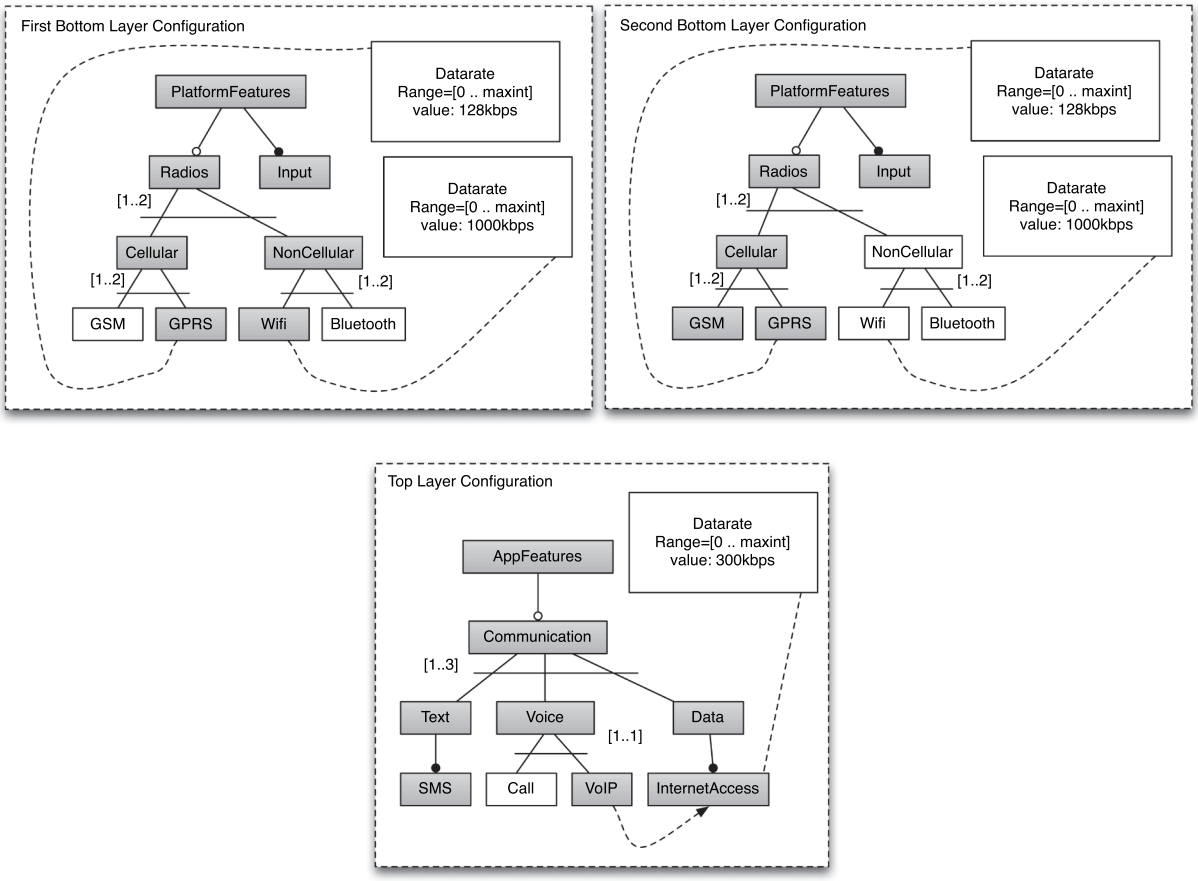

Fig. 10. Configurations used for O4.

metamodels. An attributed feature model representation and a representation of the inter-model dependencies. However, other metamodels such as the standard feature models or orthogonal variability models [24] can be used; (iii) a set of reasoners, currently the implementation offered by MAYA only supports CSP based solvers because of the expressibility they offer. However, depending on the metamodels taken as input we can easily extend this set of interfaces to map the models into different artificial intelligence paradigms. Finally (iv) questions, which represent the common interface for extracting information from a problem.

7. Real world case study

The concepts introduced and defined in the previous sections are demonstrated using case studies on two real world subjects. The *overall objective* is to investigate whether or how MAYA could improve the state of practice regarding handling of application and platform evolution in the domain of application development for mobile phones.

7.1. Research questions

First, we define a set of research questions which shall be elaborated throughout the study. In accordance with the two major tasks to apply MAYA—modeling and automated analysis—we define a different high-level research question for each category.

RQ1 Does our approach allow capturing the specifics of the mobile phone domain?

RQ2 Given a model according to MAYA, does our approach provide useful analysis results for both developers and handset manufacturers?

In order to prove that, these high-level questions are refined into some more concrete questions:

RQ1.1 Can we provide MAYA models (application features,

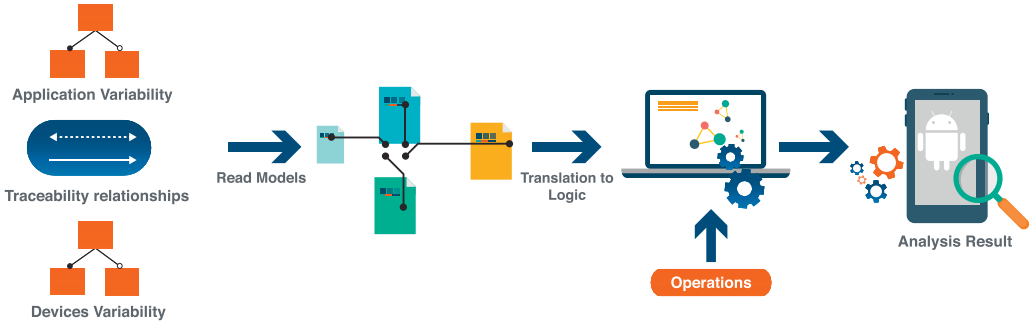

Fig. 11. The MAYA solution analysis process.

```
AppFeatures: [Communication];
Communication: [1, 3] {Text Voice Data};
Text: SMS;
Voice: [1, 1] {Call VoIP};
Data: InternetAccess;

%Constraints
VoIP REQUIRES InternetAccess;

% Attributes
InternetAccess.datarate:   Integer[0 to MaxInt], 0, 0, Inheritable: false, [kbps]
```

Fig. 12. Textual representation of the FM presented in Fig. 3.

```
% Inter-tree relationships
SMS requires Cellular;
Call requires GSM or UMTS;
InternetAccess requires [(GPRS AND GPRS.datarate >= InternetAccess.datarate) OR
    (WiFi AND WiFi.datarate >= InternetAccess.datarate)];
```

Fig. 13. Textual representation of the mapping between the two-layers presented in Fig. 3.

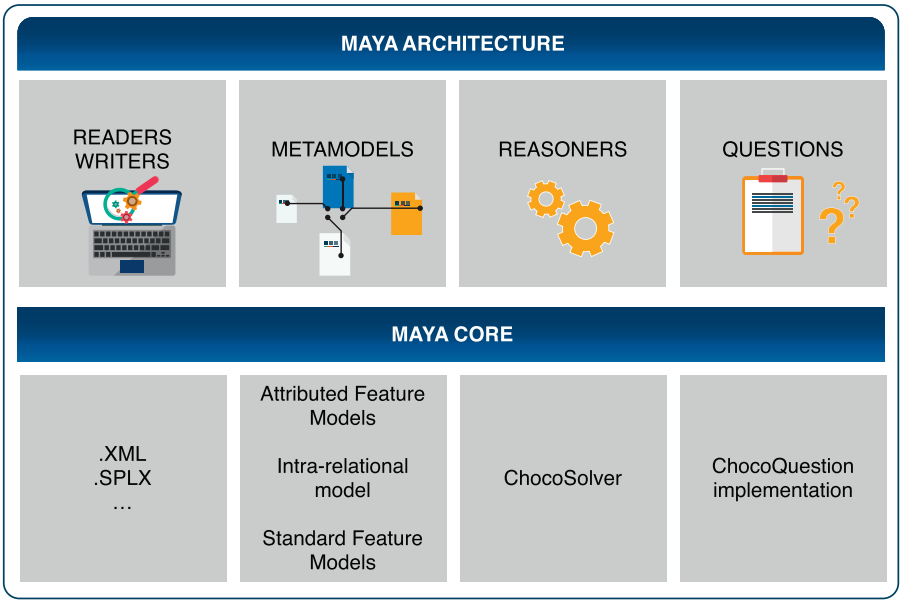

Fig. 14. MAYA solution architecture.

platform features, and inter-tree relationships) for the cases under study, depicting realistic specifics of their domain?

RQ1.2 Is MAYA flexible enough to be applied to mobile development projects of arbitrary scale? (from basic feature phones to sophisticated smartphones)?

RQ1.3 Is it realistic to let top-level features be specified by developers themselves?

RQ1.4 Is it realistic to retrieve bottom-level models in practice (e.g. by manufacturers, specifications)?

To enable a thorough investigation of RQ2 further questions are defined to refine on it:

RQ2.1 Given an application's specification, can we find out the minimally required platform features?

RQ2.2 Given an application's requirements and a concrete phone/platform combination (e.g., Samsung Galaxy S7 running on Android 6.0.1), can we determine a list of compatible and incompatible platform features for that application?

RQ2.3 Given a specific bottom layer configuration, what are the features that are enabled by the platform to be potentially used in an application?

RQ2.4 Given the specifications of two platforms (source, target), and the application to be checked for compatibility issues after porting, which application features are affected by the platform change (i.e., are no longer supported, are somehow limited, or are positively affected (e.g., due to a better display))?

7.2. Hypothesis

Based on the questions posed above, we define our supposed hypothesis:

> We can model the mobile phone domain with our modeling approach, and it is feasible to execute analysis operations as defined in MAYA in a reasonable time-frame, which we define as within one second (near-instantaneous).

7.3. Case selection

Two subjects have been selected for this case study. The rationale behind their selection was their different role they play in the

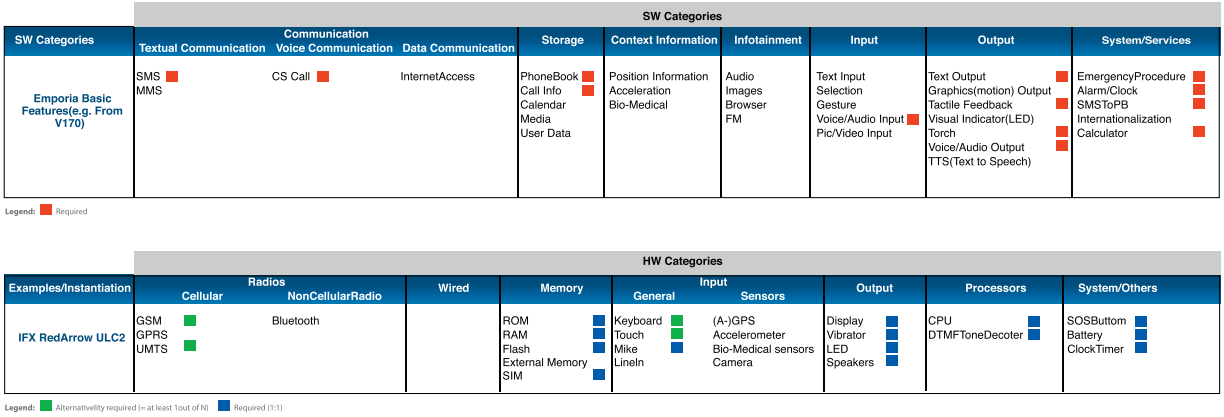

Fig. 15. O1, case study 1: platform capability analysis.

investigation:

Subject one, *Emporia Telecom*,[2] is a manufacturer of mobile phones for the target group of elderly people. The main purpose of these phones is to provide basic communication functionality such as texting, calling, and phonebook management. Due to their focus on simplicity, no advanced features such as video playback or Internet browsing are required. On the other hand, other specific requirements arise, e.g. providing an emergency button which triggers an emergency procedure to establish contact to caring family members or emergency organizations.

One of the authors has had a long-term cooperation with Emporia with respect to software development for their handsets. Therefore, particular insights and personal experience were gained which helped in the data collection and analysis of this case.

Subject two, *runtastic GmbH*,[3] is an Austrian mobile fitness company that foremost focuses on providing smartphone applications targeted at tracking and managing sports activities. The portfolio covers all major smartphone platforms, including iOS, Android and Windows Phone. This makes runtastic a viable candidate for a thorough investigation using MAYA, as issues of compatibility and evolution among different platforms arise on a daily base.

One of the authors was involved in an early project for tracking sports sessions that later led to the foundation of the company. Experiences gained there coupled with extensive use of runtastic's applications over many years, brings a lot of insights that enables the author to create realistic models of sports tracking applications for this case study.

The main features of the application under study are the ability to record accurate position data during a sports session, display that data on a map, compute various statistics, and—after finishing an activity—upload all this information to runtastic's web portal and/or share it on social networks. A so-called *live-tracking* feature allows to send current position data updates on the fly during a session, so that other users can track the sportsman's progress in real time on the web site.

Therefore, major differences exist in how each company could profit from MAYA—the different angle of views (OEM vs. app developer view), and the different classes of devices (feature phones vs. smartphones) being the main characteristics.

7.4. Data collection procedures

Data for these case studies come from various sources, although it is clearly pointed out that the approach has not been directly applied on the subjects in question over a longer period. Instead, the scenarios rather are *thought experiments* that are *grounded in practice*, aiming at exploring the potential consequences for each scenario.

As previously pointed out, one of the authors gained extensive experience in a cooperation agreement with subject one, Emporia, over a period of five years. Access to technical data sheets provided additional foundational data to specify the model assets in a realistic manner.

Subject two, runtastic, develops applications on a variety of smartphone platforms. As we look at this case foremost from a developer's point of view, the extensive use of runtastic applications on different platforms over a long period, coupled with experience in smartphone development, gives us detailed knowledge about the application requirements and corresponding platform components, which are transformed into models during the data collection process. As the application in focus is their solution for Android, another useful source of information was the specification of *permissions* as found on Google Play store, as these permissions could be mapped to our top layer in a rather straightforward way. Regarding specification of the bottom layer, the information used to construct the models comes from various sources. The capabilities of Android and its version history is publicly available and was taken from the official documentation.[4] Regarding detailed hardware specifications, the information was collected from sources such as the respective manufacturers websites, or websites with huge accessible device databases such as GSMArena.[5]

To get feedback and possibly improve on the constructed models, feedback sessions have been held with the respective companies. They were shown the separate layers, and had the opportunity to comment on it, raise concerns, or acknowledge the models as they are.

7.5. Analysis procedures

Analysis is supported using our tool implementation of MAYA, which is based on FaMa's extensible feature model analyzer. The models we derived, as explained in the previous paragraphs, served as input to MAYA. The detailed input requirements and specifications for each analysis operation have been laid out in Section 5.

The output of MAYA is partly included in the results below. It is up to the developers and/or manufacturers to interpret the results and possibly draw further conclusions.

To increase the validity of the results, we held feedback sessions with the two subjects, and inquired about comments or concerns of our analysis. This reflection helped to interpret the output, or fine-tune the models to rerun the operation for more realistic results. Relevant statements are included throughout the results section.

[2] Emporia Telecom Website: http://www.emporia.at.
[3] Runtastic website: http://www.runtastic.com.
[4] Android Developer Documentation: developer.android.com.
[5] GSMArena website: www.gsmarena.com.

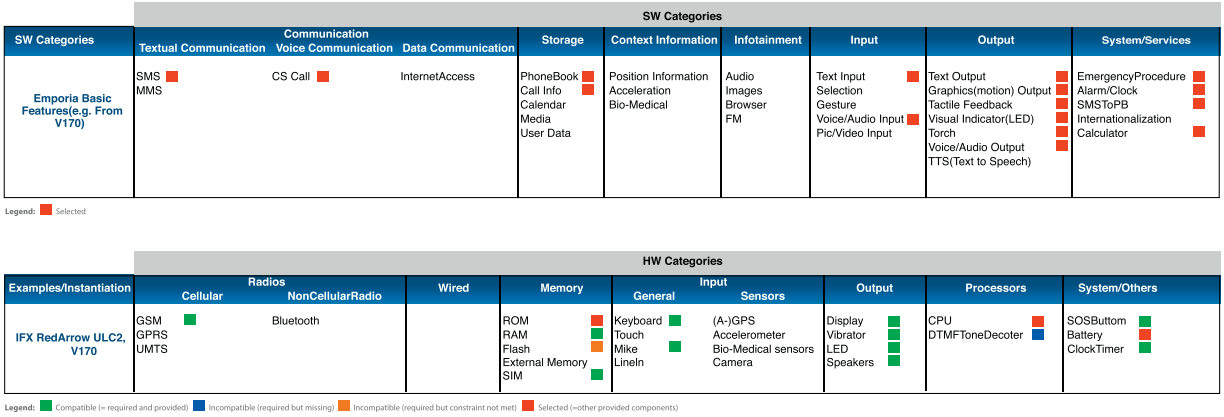

Fig. 16. O2, case study 1: platform compatibility analysis.

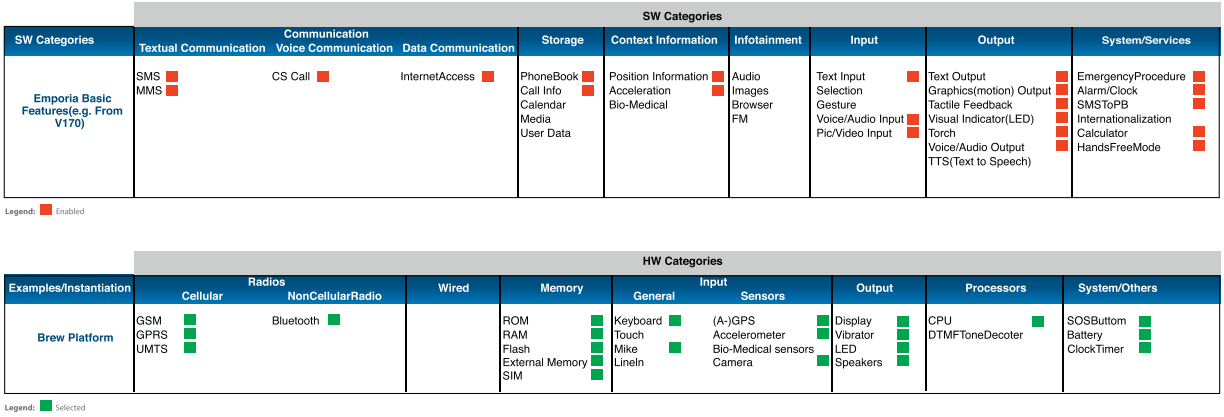

Fig. 17. O3, case study 1: application functionality analysis.

7.6. Results

The following subsection will summarize results retrieved from the study. According to its units of analysis, we will present the results in corresponding order to each research question.

7.6.1. RQ1: platform capability analysis

It was rather straightforward to specify the application features, which served as input for this analysis operation, for both cases. As a developer, one knows what an application's requirements are and can thus easily capture them in a top-level feature model. Likewise, in the case of our OEM of feature phones, the application features were known in detail, too.

The output of the analysis were different sets of platform features. From a developer's point of view, it was useful to retrieve this information, as it helped to be aware of what the exact requirements for the platform and hardware were. Likewise, the OEM could utilize the information to get a clear list of minimally required platform features—precious information that helped the OEM to be clear on the minimum requirements in their process of selecting a future target platform. Fig. 15 represents the output of this operation.

7.6.2. RQ2: platform compatibility analysis

In the case of Emporia, operation two was valuable to evaluate whether the existing application portfolio was compatible with prospective future platforms. Compared to this particular example of Infineon's ULC2 platform, the DTMF component was pointed out as possibly incompatible with the application's requirements. Therefore, the OEM has to watch out for alternative solutions (e.g. removing the corresponding application feature, buying an external DTMF chip, or implementing the functionality in software if the performance allows). Also, the Flash has been marked incompatible, as the memory size doesn't match the required amount by the applications. Therefore, the OEM has to compromise either on the top layer (e.g., support fewer contacts in the phonebook), or upgrade the memory size (which has a relevant cost implication).

Our fitness application company strives to offer their services on a variety of platforms and versions, to further raise its distribution level. It is crucial for them to know whether a specific platform/handset combination meets the requirements of their application. By executing operation two, runtastic was able to test the compatibility of their applications with a specific target device early on. For instance, when compared to one of the flagship models in 2016, the Samsung Galaxy S7, running on Android 6.0.1, all required bottom-layer features were provided and thus compatible. Running the same compatibility check on an old version of the platform, Android 2.3, immediately reveals incompatible features, such as the inability to outsource the application to external storage. Using this information, it is up to the developers to decide how to deal with such incompatibilities—in this particular case, the internal storage can/will be used instead, but the developer must be aware that this could significantly affect the compatibility with old devices that have limited amount of internal memory. Fig. 16 presents the results of the MAYA application.

7.6.3. RQ3: application functionality analysis

Operation three was perceived to be rather easily implemented. All the more interesting was it to find out whether the subjects could retrieve useful information from the operation's output. For the smartphone case, we analyzed the scenario where a brand new Android device has been announced by Google, and provided its specifications as

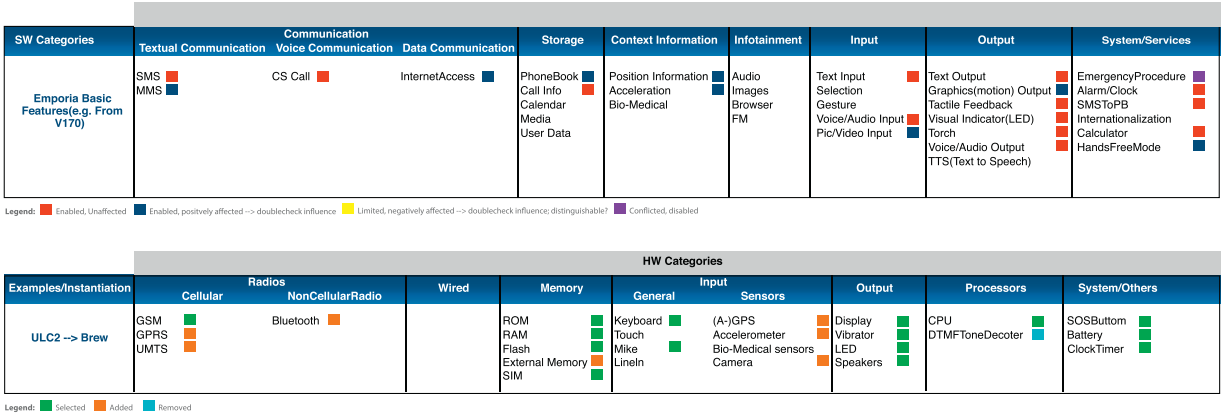

Fig. 18. O4, case study 1: platform migration analysis.

bottom layer input. Specifically, *Google's Nexus 5X*, the initial flagship device for running Android version 6 (Marshmallow), has been selected. It was interesting to see the list of features that an application could tap into.

We observed that the additional advantage of such information could be limited in practice, as information about a device's new features is often publicly released on the Internet. However, the information becomes relevant when it establishes the relation to the application feature layer, as the manufacturer's specifications most often only reveal technical information about the platform layer, not the implications on potential applications.

On subject two, Emporia, we used operation three to get a quick overview of the application features of a prospective new platform running Brew OS.[6] The output of O3 often depicts an abundance of application functionality, and in Emporia's case potentially highlights new features such as MMS or making use of location services.

Another use case was to provide the specification of a rather limited platform—a scenario which is common to increase an application's availability to a larger number of devices. When the output of MAYA's operation three does not include a desired feature, say emergency procedure, it becomes obvious that the selected platform does not sufficiently cover the desired functionality. Fig. 17 shows the results of the MAYA application for this operation.

7.6.4. RQ4: platform migration analysis

The most complex of the four operations, the platform migration analysis was at the same time expected to reveal the most useful information for adopters of MAYA.

In the first scenario, we investigated the case of Emporia's need to upgrade its platform. There were a variety of reasons for that step. From a technical perspective, it was required to add a camera functionality with an easy-to-use interface to its current portfolio. An economical reason was the intended entrance in the Scandinavian market, where GSM technology has been fully replaced by third generation technology, which is why UMTS communication chips had to be included. The specification of this upgraded platform, along with the preceding platform, was provided as input to MAYA.

Using O4 we were able to detect prospective (in-)compatibilities with regard to the specified application features. For instance, the new camera components can be utilized to take and send pictures, the GPS sensor can be used to attach position information when conducting an emergency call, and the accelerometer could be utilized to detect falls of elderly people, which would be an additional benefit in combination with the emergency procedure. While some of these findings seem obvious, the section of the output highlighting possible incompatibilities was all the more valuable. The missing *DTMF tone decoder* negatively impacts the *emergency procedure*.[7] Since the CPU of the new platform is much more powerful, the decoding can be realized in software though, and the limitation can be circumvented by the OEM.

Regarding subject two, the described fitness application shall be ported from a smartphone to a tablet. A developer is interested in analyzing whether the application will be fully compatible on the new target, or what the shortcomings are that may occur.

The tablet is characterized by components such as a bigger screen and a newer software version, but on the other hand lacks cellular connectivity. O4 shows that the specified application is positively impacted with regard to the camera feature (the tablet provides a secondary, front-facing camera) or the graphics output (due to the improved display resolution). These improvements can be used for instance to present the map in more details, and/or display additional content on the screen (such as additional statistics or fitness metrics).

On the other hand, the compromised cellular radios limits the data transmission to runtastic's web services. While data transmission is still possible, the reduced mobility must be taken into account, as the application will only work seamlessly when WiFi coverage is available. Therefore, features such as *live tracking* are severely limited, as it is unlikely to have complete WiFi coverage during a sports session.

Both subjects have pointed out that while some of the results were rather obvious, it was interesting to receive a compact list of features that potentially have to be compromised. Applying this operation in an early analysis phase allows to think about the consequences or possible workarounds to the outlined limitations. Fig. 18 shows the results of the MAYA application for this operation.

7.7. Case study conclusions

We checked the feasibility of our approach on a realistic scenario and showed the strong and weak points of our approach. We think that it would be straightforward to apply these techniques to other apps and OEMs. This certainly would improve the time to market of apps.

Currently we think that the main thing that is hindering a more widespread application of our technique is that we need to convince and motivate both OEMS and developers to release their specifications. Moreover, we would need to encode them in the form of a feature model or to adapt the technique to other variability descriptions.

As a future work we plan to extend the number of operations available as well as finding industrial cases where they apply. Also, we

[6] Brew website: developer.brewmp.com.

[7] DTMF (dual-tone multi-frequency) detection is required to recognize if the called emergency contact has acknowledged that the call was received (by pressing a certain button). If the contact did not confirm the call, the next emergency contact is being called.

Table 2
MAYA vs. other proposals.

Papers	Models	Layers	Attributes	O1	O2	O3	O4	Other operations	Tool	Evaluation
VFD+ [32]	FM/OVM	✓						✓	✓	✓
Feature Trees [33]	FM	✓	✓	✓				✓	✓	✓
CVM [34]	FM	✓						✓	✓	✓
VELVET [35]	FM	✓		✓				✓	✓	
SPL *Workflow*[36]	FM	✓						✓	✓	✓
FAMILIAR [8]	FM	✓	✓	✓				✓	✓	✓
VeAnalyzer [37]	FM	✓		✓				✓	✓	✓
Invar [38]	FM	✓		✓	✓			✓	✓	✓
Clafer [39]	FM/CM	✓	✓					✓	✓	
SPLAnE [40]	FM	✓		✓	✓			✓	✓	✓
MAYA	FM	✓	✓	✓	✓	✓	✓		✓	✓

plan to provide a graphical interface for the tooling as well as an integration for the TESALIA framework [3].

8. Related work

Product line engineering. Back in 1990, the FODA (feature-oriented domain analysis) feasibility study by Kang et al. [2] introduced the concept of *feature modeling*, which remained one of the major research areas in product line engineering since then.

Feature models were used in a variety of scenarios [4], including model-driven development, feature-oriented programming, software factories or generative programming [25]. The focus of this work will be to explicitly model the interrelations between features on different layers, and draw top-down and bottom-up conclusions (e.g., what is the impact of a platform change on an existing application).

An extended work by Kang et al. was the feature-oriented reuse method (FORM) [6], which already recognized the importance of different granularity levels. Each feature therein pertains to one of four layers (capability layer, operating environment layer, domain technology layer, implementation technique layer), and features across layers could be connected using *implemented by* relations. Our work with its different feature model layers resembles this categorization, and adds on it as it focuses foremost on the relations between the capability and implementing layers—to an extent where useful analysis of consequences from application feature or platform feature selection (in both directions) becomes possible.

Dhungana et al. [7] propose a framework to configure multi product lines (i.e., multiple product lines from possibly different suppliers and potentially different notations). To connect these models, they define similar inter-model dependencies. While this highlights the importance of dependencies across models, their focus is quite different as the goal is to support the end-user product configuration (spanning multiple product lines), while our approach aims at analyzing the feature interaction across layers of different granularity (which for instance requires sophisticated relations for meaningful results).

Feature interaction. In the early 1990s, *feature interaction* was acknowledged as a problem in the telecommunication domain, characterizing positive and, more importantly, negative side effects when introducing new features into an existing base system. It was then discovered that the basic problem of feature interaction spans across a lot more disciplines, including software engineering [26].

Previous work has studied this problem in mobile phones. A case study on Nokia phones [27] tried to model feature interaction using Colored Petri Nets, but focused on interactions with the *user interface* of the phones. Opposed to that, including the platform layer is central in our work.

A simple feature model of a mobile phone was depicted in [4]—although it demonstrated an example of feature interaction (a *camera requiring a high resolution screen*), its sole purpose (unlike ours) was to illustrate a sample feature model, but not to investigate the dependencies of the mobile phone's features.

MDD. Lack of portability due to technical revolution and changing requirements is one of the main problems of software development for consumer devices. Model-driven architecture (MDA) [28] has been proposed by OMG as a means to tackle these issues. MDA relates to our work in a way that our proposed two-layered models from Section 4 show a certain similarity to MDA's separation of concerns—note the resemblance of our top layer for application functionality to the concept of Platform-independent models (PIM) in MDA. Analogously, the bottom layer for platform components resembles a Platform-specific model (PSM). While the aim of applying MDA is often to generate code, we want to reason about software and platform evolution impacts on both layers—which could be beneficial knowledge when it comes to targeting platforms in MDA.

To analyze problems from a bottom-up perspective (e.g., what is the impact on application features due to a platform change), Architecture-Driven Modernization (ADM) is a related field [29], as it is concerned with modernization of existing software solutions.

DSL and variability. There are proposals that use different variability dimensions (layers) in language product lines [30,31] considering for instance the abstract syntax, the concrete syntax and the semantics dimensions. In this context, each feature of the corresponding feature model represents a language module and the feature in the given dimension.

These features can be shared and reused by different models, thus establishing a common denominator that facilitates the inter-tree between them and allows the language designer to define a language in an easy way according to the needs of the end users.

Although the applicability of multilevel variability has been addressed in this specific case to build a DSL using a line of language products in the same application domain, in our proposal we address the issue of variability in software product lines with a different approach.

On the one hand, we have considered the use of two layers (application and platform) to track the application features to the platform features and, on the other hand, to allow the reuse of components and the easy migration of applications to different platforms or vice versa. Some related works that use different layers are discussed in more detail at the end of this section in a comparison made with our proposal.

Comparison of MAYA with other proposals. To complement this section, we present some related proposals. Table 2 summarizes some of the main characteristics of MAYA and compare them with other proposals. We remark that the characteristics presented here are the ones that

show the main aspects of MAYA. However, there are other benefits of the related proposals that are not included in the comparison. For example, FAMILIAR [8] supports the merging of several models something that we do not handle in this work. All these other aspects are out of the scope of this comparison and therefore this comparison can be biased.

The *Papers* column in Table 2 shows some of the works that are directly related to our proposal.

The *Models* column in Table 2 indicates the type of model used by the proposals, unlike [32] that uses feature models together with orthogonal variability models, and [39] that unifies feature models with class models; all of the proposals use feature models as the main basis of the work.

The *Layers* column in Table 2 indicates the if the proposal support layers or not. Only [8] and [36] propose an architecture that, although it is not formally defined in the work, we have considered as two layers as it clearly distinguish two levels: the upper level destined to the requirements of the users and the lower level corresponding to the logic of operations. In any case, only MAYA uses a well-defined architecture based on two layers (see Section 4).

The *Attributes* column in Table 2 indicates whether the proposal uses quality attributes. We note that in addition to MAYA, [8,33,39] fulfill this characteristic. Although it is true that the aforementioned proposals use attributes at a certain moment, only MAYA takes full advantage of the attributes to perform the integration of feature models between the layers.

The column *O1* in Table 2 refers to the *"Platform Capability Analysis"* operation included in MAYA. Analyzing each of the proposals, we can say that all of them make use of analysis operations that, in addition to allowing the integration of large-scale models. This enables new variability management operations not previously defined.

The column *O2* in Table 2 refers to the operation of MAYA *"Platform Compatibility Analysis"*. Here, we find that the proposal presented in [38] allows easy adaptation with modeling techniques and notations of other existing tools. On the other hand, in the proposal [40], an analysis is done on the models to ensure that they are compatible with all the specifications given by the user, with which we would say that both proposals include analysis operations for platform compatibility. The other proposals do not include operations that allow this analysis process.

The column *O3* and the column *O4* in Table 2 refer to the operations of MAYA *"Application Functionality Potential Analysis"* and *"Platform Migration Analysis"* respectively. In this group, none of the proposals include analysis operations that allow easy integration of models or the ability to cope with the changes that a feature may have without affecting the related models. On the other hand, it is also not evident that the proposals presented include operations that allow the handling of conflicts that arise due to changes in the configurations of the models, and how to control that the change of one does not affect the other or vice versa. In this sense, MAYA is a pioneer in including this type of analysis operations.

The *Tool* column in Table 2 indicates whether the proposal implements a tool as part of the contribution. All the proposals present a tool for the analysis of feature models in different layers. However, MAYA is the only tool that was implemented using a well-defined and structured scheme based on layers. It is also the only one that incorporates a set of operations that validate the platform's capacity in terms of operationalization, and considers aspects such as migration, support, and the potential to solve problems if there are changes in the definition of the models.

Finally, the *Evaluation* column in Table 2 indicates whether the proposal has been validated using case studies. Almost all the proposals include an evaluation, some of them carried out in the business sector such as [8,33,34,36,41]. Although it is true that the proposals mentioned took real data provided by companies, none had direct participation during the evaluation process. MAYA, on the other hand, was evaluated in a real environment as one of the authors worked directly on projects with the companies where the case studies were carried out.

9. Future work

Extending the operation catalogue. While we propose four operations in this paper, they are are open for extension. For instance, a possible new operation would be to ask for the minimally required platform version to support a particular application. However, this would require to take into account model versioning that is a feature not supported in our current proposal. Also, we can imagine operations that instead of retrieving a configuration work over the inter-tree relationships and analyse them looking for inconstancies or errors.

Graphical frontend for analysis input/output. As mentioned in the previous section, for our prototypical implementation we do not employ tool support for product configuration of the input, nor is there a graphical representation of the analysis output. There is ongoing work[8] to integrate FaMa toolsuite with various modeling tools (e.g., MOSKitt, pure::variants). While these tools help to model each layer individually, we plan an extension that supports configuration on both layers concurrently, e.g. by combining them in a graphical frontend. Providing support for the complex task of defining mappings is very important [19]. At the same time, such assistance also ensures the integrity of the inter-tree relations, as only valid features and relation types can be selected.

Presenting the output in a visual format, similar to the figures in Section 5, will be another beneficial extension. The different colors to reflect the meanings of the different output categories (e.g., a conflicting feature), will help the user to easily understand the impact of his operation. It is envisioned to be implemented in a way that by just hovering over single features, the respective feature(s) on the connected layer can be highlighted, giving the user a real-time feedback upon (de-)selection of a feature.

Runtime analysis. From what has been proposed so far, the mappings between the two layers were of rather statical nature. An interesting aspect would be to extend the analysis with runtime information. Introducing runtime dependencies would allow for modeling dynamic situations such as: *If the battery drops < 15%, certain features get deactivated*. Another example would be: *If we move to an area where there is only GPRS coverage (and therefore the data rate drops), certain applications would no longer work (e.g., a Video chat)*.

In terms of modeling these runtime dependencies, they could be annotated as such, very much like the concept of binding times detailed in [42]. In addition to that, a runtime environment, where one can specify and simulate the environment events (battery drop, out of coverage, etc.), will be required—which is all subject to future work.

Analysis with a unified feature model. As mentioned in Section 4, our proposal was designed to work exclusively with two-layer models since in a real environment there are different practitioners that make platform and software, where each layer evolves in a different way [38]. Having the platform and application layers separate enables us to control the changes that may occur in the applications when they update or when they migrate to different platforms without affecting their performance.

However, in the future we could evaluate the integration of the two layers in a single feature model where the platform and software layers would be represented by two mandatory sub-features. In this way, we

[8] http://www.isa.us.es/fama/?FaMa_Current_Projects.

would show how a change in one feature in the software can affect the platform or vice versa and how these variations could be managed in both scenarios to determine which responds better to the transitions that feature models can experience.

Inter-tree relationships. In this work we defined a textual format to define inter-tree relationships that has room for improvement. First, it would be interesting to define its formal syntax and semantics. Second, it can be enriched to support bottom–up relationships and therefore increase the analysis capabilities. Third, there can be automated mechanisms to reverse engineer this relationships from source code, log files or other kind of documents. Finally, although we have shown the feasibility of the approach by an experience in industry, there are practical questions that can be better validated in future work such as assessing the difficulty of defining these kind of constraints, validation the textual notation used or experimenting with error handling when coping with complex models.

The contribution of this work and the future work that we propose deal with so-called technical sustainability [10,11] and the study of how these works affect the dimensions of sustainability is a challenge that can open a new line of works from theoretical to practical.

More-than-two layers. In this work we have proposed a framework relying on two levels obs abstraction. However, we envision that there might be scenarios where having more than two levels is interesting. In future work, we plan to extend this framework to support as many levels of feature models required in the same spirit as in other domains such as in DSLs [43].

SMT reasoning. In future work we plan to investigate other mechanisms for implementing our tooling. Concretely, we plan to explore the use of SMT solvers that promises an scalable solution while coping with quality attributes and boolean variables.

Operation formalization. In the past, we have already formalized several FM operations [44] for single model scenarios. We plan to perform the same formalization for the operations considering more than one model in future work.

10. Conclusions

In a world where hardware frequently outpaces software in terms of innovation and speed up (which holds especially true for mobile phones and consumer electronics in general), a mechanism to understand the impact of these frequent technology changes on existing software is desirable. At the same time, new types of applications are enabled due to new or improved hardware components. It is equally interesting to see whether certain types of applications can be implemented on existing hardware, and to understand the limitations why some applications may not be realizable (e.g., the platform version in use doesn't support a feature yet).

This work proposed to use two-layered feature models for the top layer (comprising application functionality) and the bottom layer (including platform components) of a mobile phone. The key part was to connect the layers using inter-tree relationships, which enables studying the consequences of changes on one layer onto the respective other layer.

When it comes to platform and design decisions, both developers and handset manufacturers could profit from such a model, because potential problems and incompatibilities can be identified at a very early development stage. Furthermore, such an instrument can be used to discuss design decisions with marketing and other involved stakeholders who might not have detailed technical knowledge.

The contribution of this work was to provide two fairly complex feature models for the respective layers, as a prerequisite for further research. We identified the required type of inter-tree relationships and how they compare to well-known cross-tree relationships. Specifying the operations O1–O4 set the stage for further analysis. A prototypic implementation in Section 6 was used to demonstrate the validity of the proposed approach. Building upon this implementation, future work described planned enhancements. An additional extension could be a runtime simulation environment that allows for studying the consequences of runtime information updates, such as a drain in battery capacity.

This work is flexible enough to cover anything from simple feature phones, to top notch smartphones, to tablets. Finally, this method could very well be applied to other domains facing similar problems, for instance consumer electronics in general.

Accessing source code

The source code of MAYA and the dataset used in the operations presented in this paper can be downloaded from the FaMa Github project website: https://github.com/FaMaFW/FaMA/tree/branches/fama-two-layers in the fama-two-layers branch.

It is not possible to publish the real data extracted from the feature models used in the experimentation due to the privacy policies of the companies that provided us with the information.

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, The SEI series in software engineering, Addison-Wesley, Boston and Mass. and London, 2001.

[2] K.C. Kang, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Carnegie Mellon University, Software Engineering Institute, Pittsburgh and Pa, 1990. URL: http://www.worldcat.org/oclc/25525947 .

[3] J.A. Galindo, H. Turner, D. Benavides, J. White, Testing variability-intensive systems using automated analysis: an application to android, Softw. Qual. J. 24 (2) (2016) 365–405.

[4] J.A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, A. Ruiz-Cortés, Automated analysis of feature models: Quo vadis? Computing (2018), https://doi.org/10.1007/s00607-018-0646-1.

[5] M. Lettner, M. Tschernuth, R. Mayrhofer, Mobile platform architecture review: android, iphone, qt, Proc. EUROCAST 2011: 13th International Conference on Computer Aided Systems Theory, Springer-Verlag, 2011.

[6] K.C. Kang, S. Kim, J. Lee, K. Kim, G.J. Kim, E. Shin, Form: a feature-oriented reuse method with domain-specific reference architectures, Ann. Softw. Eng. 5 (1998) 143–168.

[7] D. Dhungana, D. Seichter, G. Botterweck, R. Rabiser, P. Grünbacher, D. Benavides, J.A. Galindo, Configuration of multi product lines by bridging heterogeneous variability modeling approaches, 15th International Software Product Line Conference, IEEE, 2011, pp. 120–129, https://doi.org/10.1109/SPLC.2011.22.

[8] M. Acher, P. Collet, P. Lahire, R.B. France, Familiar: a domain-specific language for large scale management of feature models, Sci. Comput. Program. 78 (6) (2013) 657–681.

[9] M. Lettner, M. Tschernuth, R. Mayrhofer, Feature Interaction Analysis in Mobile Phones: On the Borderline between Application Functionalities and Platform Components, MoMM 2011, ERPAS Symposium, (2011).

[10] C. Becker, R. Chitchyan, L. Duboc, S. Easterbrook, M. Mahaux, B. Penzenstadler, G. Rodriguez-Navas, C. Salinesi, N. Seyff, C. Venters, et al., The Karlskrona manifesto for sustainability design, arXiv:/1410.6968 (2014).

[11] C. Calero, M. Piattini, Puzzling out software sustainability, Sustain. Comput. 16 (2017) 117–124.

[12] D. Batory, Feature models, grammars, and propositional formulas, Computer Science Dept., Univ. of Texas at Austin, Austin and Tx, 2005. URL: http://www.worldcat.org/oclc/424510430 .

[13] A.S. Karataş, H. Oğuztüzün, Attribute-based variability in feature models, Requirements Eng. 21 (2) (2016) 185–208, https://doi.org/10.1007/s00766-014-0216-9.

[14] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Comput. Netw. 51 (2) (2007) 456–479, https://doi.org/10.1016/j.comnet.2006.08.008.

[15] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, LNCS, Advanced Information Systems Engineering: 17th International Conference, CAISE 2005, Springer, 2005.

[16] M. Mendonca, M. Branco, D. Cowan, Splot: software product lines online tools, Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, ACM, 2009, pp. 761–762.

[17] D. Benavides, P. Trinidad, A. Ruiz-Cortés, S. Segura, FaMa, Springer, Berlin, Heidelberg, pp. 163–171. doi:10.1007/978-3-642-36583-6_11.

[18] F. Roos-Frantz, D. Benavides, A. Ruiz-Cortés, A. Heuer, K. Lauenroth, Quality-aware analysis in product line engineering with the orthogonal variability model, Softw. Qual. J. 20 (3–4) (2012) 519–565, https://doi.org/10.1007/s11219-011-9156-5.

[19] F. Heidenreich, J. Kopcsek, C. Wende, Featuremapper: mapping features to models, Companion of the 30th International Conference on Software Engineering, ICSE Companion '08, New York, NY, USA, 2008, pp. 943–944, https://doi.org/10.1145/1370175.1370199.

[20] Formal Approach to Integrating Feature and Architecture Models, in: M. Janota, G. Botterweck (Eds.), Budapest and Hungary, 2008, , https://doi.org/10.1007/978-3-540-78743-3_3.

[21] D. Benavides, On the Automated Analysis of Software Product Lines using Feature Models: A Framework for Developing Automated Tool Support, Ph.D. thesis, University of Seville, 2007.

[22] D. Benavides, Fama tool suite, 2011. URL: http://www.isa.us.es/fama/.

[23] P. Trinidad, D. Benavides, A. Ruiz-Cortés, S. Segura, A. Jimenez, Fama framework, 2008 12th International Software Product Line Conference, IEEE, 2008, p. 359, https://doi.org/10.1109/SPLC.2008.50.

[24] K. Pohl, G. Böckle, F.J. van Der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer Science & Business Media, 2005.

[25] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison Wesley, Boston, 2000.

[26] L. Blair, G.S. Blair, J. Pang, C. Efstratiou, Feature interactions outside a telecom domain, FICS, (2001), pp. 15–20.

[27] L. Lorentsen, A.-P. Tuovinen, J. Xu, Modelling feature interactions in mobile phones, FICS, (2001), pp. 7–13.

[28] OMG, Mda Guide Version 1.0.1, (2003).

[29] OMG, Adm whitepaper: transforming the enterprise, (2007). URL: http://adm.omg.org/ .

[30] D. Méndez-Acuña, J.A. Galindo, B. Combemale, A. Blouin, B. Baudry, Reverse engineering language product lines from existing DSL variants, J. Syst. Softw. 133 (2017) 145–158.

[31] E. Vacchi, W. Cazzola, S. Pillay, B. Combemale, Variability support in domain-specific language development, International Conference on Software Language Engineering, Springer, 2013, pp. 76–95.

[32] A. Metzger, K. Pohl, Variability management in software product line engineering, Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on, (2007), pp. 186–187, https://doi.org/10.1109/ICSECOMPANION.2007.83.

[33] M.-O. Reiser, M. Weber, Multi-level feature trees, Requirements Eng. 12 (2) (2007) 57–75.

[34] A. Abele, Y. Papadopoulos, D. Servat, M. Törngren, M. Weber, The CVM framework-a prototype tool for compositional variability management. VaMoS 10 (2010) 101–105.

[35] M. Rosenmüller, N. Siegmund, T. Thüm, G. Saake, Multi-dimensional variability modeling, Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, ACM, 2011, pp. 11–20.

[36] M. Acher, P. Collet, A. Gaignard, P. Lahire, J. Montagnat, R.B. France, Composing multiple variability artifacts to assemble coherent workflows, Softw. Qual. J. 20 (3–4) (2012) 689–734.

[37] R. Schröter, T. Thüm, N. Siegmund, G. Saake, Automated analysis of dependent feature models, Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, ACM, 2013, p. 9.

[38] J.A. Galindo, D. Dhungana, R. Rabiser, D. Benavides, G. Botterweck, P. Grünbacher, Supporting distributed product configuration by integrating heterogeneous variability modeling approaches, Inf. Softw. Technol. 62 (2015) 78–100.

[39] K. Bak, Z. Diskin, M. Antkiewicz, K. Czarnecki, A. Wkasowski, Clafer: unifying class and feature modeling, Software & Systems Modeling 15 (3) (2016) 811–845.

[40] G.K. Narwane, J.A. Galindo, S.N. Krishna, D. Benavides, J.-V. Millo, S. Ramesh, Traceability analyses between features and assets in software product lines, Entropy 18 (8) (2016) 269.

[41] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, G. Saval, Disambiguating the documentation of variability in software product lines: a separation of concerns, formalization and automated analysis, Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, IEEE, 2007, pp. 243–253.

[42] M. Svahnberg, J. van Gurp, J. Bosch, On the notion of variability in software product lines, Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on, (2001), pp. 45–54, https://doi.org/10.1109/WICSA.2001.948406.

[43] A. Rossini, J. de Lara, E. Guerra, A. Rutle, U. Wolter, A formalisation of deep metamodelling, Formal Aspects Comput. 26 (6) (2014) 1115–1152, https://doi.org/10.1007/s00165-014-0307-x.

[44] A. Durán, D. Benavides, S. Segura, P. Trinidad, A.R. Cortés, FLAME: a formal framework for the automated analysis of software product lines validated by automated specification testing, Softw. Syst. Model. 16 (4) (2017) 1049–1082, https://doi.org/10.1007/s10270-015-0503-z.