ISSELAB TECHNICAL REPORT

FM-PRO Feature Modeling Process, Technical Documentation

Thorsten Berger, Wardah Mahmood,
Johan Martinson and Jude Gyimah

Version 2, November 2024

Intelligent Software Systems Engineering (ISSE) Lab
Chair of Software Engineering
Ruhr University of Bochum, 44801, Bochum, Germany

se.rub.de | **isselab.org**

FM-PRO Feature-Modeling Process, Technical Documentation

Overview and Legend

In the following, we assume that product-line scoping has already been conducted (or is underway), as it is not a part of the feature-modeling process. Additionally, we assume that the reader is familiar with feature modeling, including syntax and semantics of feature models, as well as its usages. The appendix at the end briefly illustrates those and refers to the relevant literature for further details.

Adopting Product Lines or Highly Configurable Systems. Product lines platforms or highly configurable systems can be adopted in different scenarios [12, 6]. Since they significantly influence the creation of feature models, we briefly discuss them.

- When building a product line platform from scratch (*pro-active adoption*), you predominantly create the feature model in a top-down fashion. You analyze the domain and model it in a reasonable scope. For instance, you model the features that you think you can develop and sell to customers. In other words, you start by creating the top-level features and then refine them.
- When building a product line platform from one existing product (*re-active adoption*) or from multiple existing products (*extractive adoption*) you predominantly build the feature model in a bottom-up fashion. You analyze the existing product(s), specifically their codebase, their documentation, and other relevant artifacts (e.g., models or configuration options), and start by creating leaf features, which are then generalized and abstracted.

However, while we say “predominantly” top-down or bottom-up, in all three adoption scenarios one does both. Notably, besides explicitly recording features, their relationships, and their constraints, feature models are more than just descriptive assets (see uses in Fig. 3). They serve to provide an overview understanding of the system and prescribe the valid configurations for derivation in product lines.

Process Phases. In our process, we classify the different activities into four phases: Pre-Modeling, Domain Analysis and Scoping, Modeling, and Maintenance and Evolution. Figure 1 depicts these phases, together with typical iterations among the last three phases.

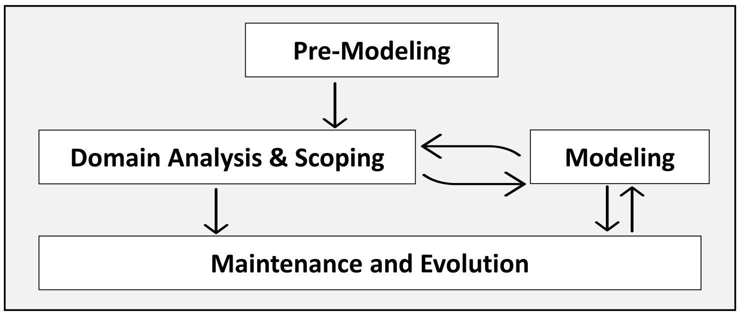

Figure 1: *The four main phases of the feature-modeling process*

Process Legend. Table 1 shows a legend for the symbols we use to describe our process.

P: Pre-Modeling Activities

Before you start the actual modeling, you plan the feature modeling and train the relevant stakeholders. The result is a clarification of the stakeholders involved and their roles, a description of the model purpose, and a change and expectation management plan. We recommend defining the model purpose and providing training in iteration, which allows the purpose to be clarified and refined.

Table 1: *Process legend*

symbol	description
❓	Decision affecting the following activities
⚙	Activity
(⚙)	Optional activity
⚙⚙	Composite activity
(⚙⚙)	Optional composite activity
⚙	Sub-Activity of a composite activity

⚙ **P1: Identify stakeholders.** This activity requires you to identify the key stakeholders that will participate in the feature modeling process. It is a mandatory activity.

Description: You need to identify the relevant stakeholders, who can have diverse roles in the organization. We distinguish between four kinds, which are not necessarily disjoint:

- (i) *experts* are those who will provide input about features and their constraints, as domain- or implementation-oriented experts;
- (ii) *modelers* are those who will perform the modeling;
- (iii) *method experts* are those who are knowledgeable in variant management and can be consulted in case of doubts (explained below); and
- (iv) *model users* are those who will use and benefit from the feature model.

The *experts* (i) should have sufficient knowledge about the domain or about the implementation. While the former understand what features need to be developed for economic benefit, the latter know the technical details about the software in depth (e.g., developers). Depending on the purpose of the feature model, one can have one representative of each kind, multiple representatives of either kind, or one who has knowledge of both. In our experience, we even observed companies where the developers traditionally had very good insights into the business and sales aspects, especially when there used to be a close relationship due to frequent meetings. In many cases, however, developers have never learned to think in terms of the domain or business, and require training and a pilot project to obtain such a perspective.

Modelers (ii) are often system or software architects, project managers, or requirements engineers, since they usually build abstract system models. The number of stakeholders performing the modeling in an organization should be low, ideally as low as a single person.

Method experts (iii) are often developers who have technical knowledge of the variants, who can answer questions related to commonality and variability of the variants. Specifically, while the modelers are more familiar with the tooling and notation, method experts are more familiar with the functionality of the system, and more importantly, the implemented features. They become relevant when determining if a certain functionality qualifies as a feature, and if so, where should it be placed in the feature model.

❓ Finally, it is important to decide who are the *model users* (iii). If they are end-users or even customers, then the feature model needs to be intuitive and easily understandable.

Outcomes: You should have a list of all participants taking part in the feature modeling creation process.

❓ A core question is *whether the feature modeling is initiated from inside or outside the organization.* If the *modeler* (i.e., the stakeholder or group leading the modeling) is external to the organization, we recommend conducting the introductory meeting explained shortly below. This develops mutual trust between the stakeholders and helps understanding the organization's structure, the software product(s), and the potential stakeholders.

❓ Another core question is *whether the potential participants are aware of the benefits of feature models and whether they are motivated to construct one.* If the stakeholders are mainly developers, who often are not used to think in terms of features, they are probably less aware of the benefits, such as automated product derivation and faster product delivery. Then, we recommend the following introductory meeting before commencing the modeling.

( ) **P2: Conduct an introductory meeting.** In this activity, an introductory meeting is conducted to ensure that all participants are on the same page regarding the intent behind the initiative of feature model construction as well as its potential benefits. It is an optional activity.
Roles: The experts and modelers should participate in this activity. Method experts and model users can also participate in this activity optionally.
Description: You focus on motivating the stakeholders based on the feature model benefits, which can involve demonstrating product derivation by configuration in an existing product line. It is important to lay a solid foundation here, since it creates the momentum for the later phases. You can emphasize *incremental benefits for incremental investment* [16, 17], since the more features that are modeled, the more developers can benefit from the it (e.g., easier code maintenance, automated product derivation). Lastly, you can also use this meeting to appoint an authority figure—an internal or external representative for the organization who might or might not participate in the modeling. It is important that the authority figure is trusted and followed by the stakeholders, to effectively direct them to manage the time and resources for effectively performing the feature-modeling process. An example of an authority figure could be a business champion; an individual or group from the product management team that ensures that the practices in the company are well-aligned with the established business goals.
Outcomes: The participants should be informed on the rationale behind feature model creation and its resulting benefits. You should have an appointed authority figure that will motivate and enables the feature modeling process.

P3: Define model purpose. Here, all the various purposes the model could serve are elaborated on. This is a mandatory activity.
Roles: The experts and modelers should participate in this activity. It is recommended that the model users also participate in this activity in order to obtain a better understanding of the possible ways the feature model can be used.
Description: You clarify what to use the model for. Some examples are shown in Fig. 3. This activity is important to focus the modeling on the relevant features and modeling concepts (e.g., constraints), and avoid wasting time on irrelevant ones. Note that when the feature model should serve both management & design and development & QA purposes, there is often a tension between designing the model more towards capturing domain- and business-oriented features or towards implementation-oriented features. In other words, the feature model is often seen as a pivotal model artifact, used as a communication platform to support business goals, while at the same time it should be able to derive individual products from the platform in an automated process supported by a configurator tool.
Outcomes: The participants should be informed of the different purposes the feature model could serve.

A core question to decide when to train stakeholders is *whether they possess product-line education*. If they are already knowledgeable on the relevant concepts (e.g., features, feature models, constraints) and their potential benefits, then the following activity can be kept at the same position and used to re-iterate through the concepts briefly. Otherwise, it should be done earlier in the process, specifically before defining model purpose, and used to reconcile the education level.

P4: Provide training. Next, the participants are trained on the required tools, notation, and the relevant concepts of software product-line engineering. A pilot project is also employed to make the participants accustomed to the process using a system of smaller scale. This is a mandatory, composite activity.
Roles: All roles should participate in the training activity.
Description: Training includes becoming familiar with the feature-modeling notation and the tool used, as well as with the Modeling phases and principles, called **P4.1: Tool and Notation**

Training. Share an agenda of the training beforehand to facilitate a structured meeting. Next, training also involves familiarization with product line engineering (e.g., platform architectures, software configuration, and product derivation) in a sub-activity we call **P4.2: Product-Line Education**. Additionally, to support learning the feature-modeling notation and its semantics, relate to known concepts. For developers, for instance, feature types and their graphical representation can be related to classes or data types. In practice, the training is often done together with the vendor of a feature-modeling tool.

We recommend involving a **P4.3: Pilot Project** of around three days. This should be done with a small (sub-)system of the company that exists in multiple variants, which have sufficient commonality and do not come with strict deadlines regarding the release to production. This allows very fast feedback loops and facilitates training. If your organization does not have an existing system and rather wants to adopt a product-line or highly configurable system from scratch, then you can refer to existing data sets of clone&own-based systems [13, 24, 14].

The pilot should comprise all the activities of the feature-Modeling phases below. We recommend creating a platform with around 10–20 variation points that represent the differences in the individual variants. So, identify the differences in the implementations, abstract them into features, and model them in a feature model.

As a guided exercise, a benefit of a pilot is to "walk" those having technical knowledge up to the domain. Those stakeholders usually understand the differences in detail, that is, in terms of implementation concepts. When asked about the details, they usually provide those implementation-level details. The idea is to ask them various times why the differences exists, leading to increasingly domain-oriented explanations, until the difference can be described by the presence or absence of a specific feature. The pilot project will also give experience in product derivation

The pilot also helps to connect the business and development worlds. Connecting features to assets and business aspects is important, since doing that later is difficult. This will also improve acceptance of the feature model, since product derivation before was usually a manual and error-prone activity, requiring copying and pasting software assets and packaging them properly. Selecting a reasonably small sub-system for the pilot can substantially improve the training and acceptance.
Outcomes: Participants have acquired the required training for the tools and notations, as well as the relevant concepts in software product-line engineering. They should also be familiar with the flow between the phases and activities of the process.

P5: Create change and expectation management. The next activity involves creating a communication plan that is clear and well-understood. This is a mandatory activity.
Roles: The experts, modelers, and method experts should participate in this activity.
Description: Defining and executing a communication plan is crucial. It should explain the benefits, especially the reuse potential and the respective business-related benefits, such as shorter time to market. We recommend describing the benefits tailored to the different stakeholders. The communication plan should also explain the necessary changes in overall processes and organizational structures, as well as in the architecture of the platform and the individual products. Explaining the notion of feature, and why needed, is also important.
Outcomes: You should have a communication plan entailing the potential benefits of feature models tailored to different stakeholders. It should also entail the required changes in the process, organization, and architecture.

P6: Establish a forum and a workshop format. In this activity, a workshop format is adopted for executing the activities of the Pre-Modeling phase and validating the feature model towards the end of the process execution. This is a mandatory activity.
Roles: All roles should participate in this activity.
Description: It is advisable to establish a forum with regular meetings to execute the Pre-Modeling activities and discuss maintenance and evolution. Since many Pre-Modeling activities focus on shared goals (e.g., define model purpose, identify stakeholders, unify domain terminology, it is important that the stakeholders share a consensus on important decisions. Additionally, since a feature model is brittle, one or a few stakeholders in the organization should maintain and evolve it in a workshop

setting. Lastly, when modeling, the workshop format helps validate the feature model (cf. page 4), as well as to evolve and maintain it. It is also advisable to define an approval process for new features, ideally as part of the workshop.
Outcomes: You should have a workshop for executing the Pre-Modeling phase and validating the feature model at the end.

P7: Define decomposition criteria. Here, you define a criteria that helps modelers decide how to decompose features in the model. It is an optional activity.
Roles: The experts, modelers, and method experts should participate in this activity.
Description: Defining a decomposition criteria will help modelers create the feature hierarchy when modeling the features (cf activity Define Coarse Feature Hierarchy). Notably, the meaning of the hierarchy edges in a model is not well defined. Modelers are relatively free to stick with Part-Of or Is-A relationships between features and model the hierarchy freely to be as intuitive as possible, or to conceive and document domain-specific decomposition criteria for the model. These could reflect existing hierarchies (e.g., of physical parts of the product) in the organization or even parts of the architecture decomposition, or other hierarchies that your stakeholders are familiar with in customer-facing catalogs.
Outcomes: You should have a well-defined decomposition criteria for decomposing features into sub-features.

P8: Unify domain terminology. Finally, the domain terminology is unified to prevent ambiguities in the process execution. This is an optional activity.
Roles: All roles should participate in this activity (in a workshop format).
Description: This optional activity can be necessary when the domain terminology is too diverse and ambiguous in the organization. The risk is that different perceptions of domain concepts might cause confusion and lengthy discussions. We suggest a dictionary with descriptive terms for feature names. If several feature models will be created, you could also define a hierarchical naming schema.
Outcomes: You should have a catalog of unified terminology that can be referred to at any point in the process.

A final and important question is *whether the process is applicable for the specific use case.* This is a complex decision which needs to be taken by all stakeholders involved.

D: Domain Analysis and Scoping Activities

After the Pre-Modeling, there are two main phases carried out iteratively: Domain Analysis and Scoping, and Modeling. In the first one, described in this subsection, information about features and their relationships relevant to the subsequent Modeling phase is extracted. Iterating between both phases allows you to gradually increase your modeling expertise, as well as to safely and incrementally evolve the feature model. The idea is that you start with an initial domain analysis and scoping, to gather and document information that is sufficient to proceed with the modeling activities. Then you iterate—increasingly more closely—where you obtain features and immediately model them. You iterate until all the identified features have been modeled. Usually, you even develop the system in parallel. Once you have an initial software system controlled by the feature model, this will also help with the iteration.

D1: Identify features. This activity initiates the Domain Analysis and Scoping phase. In this activity, you identify the features to be modeled, and model them, either right-away or until you have a sizable number of features that can be modeled. This is a mandatory, composite activity.
Roles: The experts, modelers, and method experts should participate in this activity.
Description: Before modeling features we need to identify them. We distinguish between the bottom-up and the top-down strategy as explained above. Recall that the former you mainly apply for the extractive and the re-active adoption of product lines, so when you already have a system or a set of cloned system variants. In practice, you apply both the bottom-up and the top-down

strategies, but put more emphasis on either one based on the adoption strategy. When identifying features, you should look for units that represent a distinct, well-understood, and graspable aspect of the software system [4]. Considering the flexible nature of features, modelers can define the granularity of the features as they see fit. We intentionally leave it open so the modelers can decide on a granularity based on factors such as defined model purpose and size of the codebase.

When identifying features, first focus on those that distinguish variants. You should also prefer features of type Boolean for easy comprehension of the resulting model. The following two main identification strategies exist:

- ⚙ **D1.1: Bottom-up feature identification** In this sub-activity, features are extracted from one or multiple systems using different sources. This activity is mandatory in situations where there is any software system in place.
  Description: For *re-active product-line adoption*, if you have one existing system, you start by considering the existing and demanded configuration options, which give you a list of features to start with. Then, you can analyze the user interface to gain an overview understanding of the variant as well as interact with the system to identify the functionality meaningful to end-users. At this stage, you can already identify if a feature is mandatory or optional depending on its relevance for different stakeholders. You should look for domain-oriented features (i.e., what functionality is provided) instead of solution-oriented features (i.e., how is the functionality implemented) at this stage. Next, you look into the codebase to identify functionality corresponding to the identified features. It is likely that you find distinct features from the codebase that were not apparent from the user interface analysis (i.e., solution-oriented features). You can also consider other sources to identify features and their dependencies, including commit messages, pull requests, user stories, and product documentation. For identifying feature groups (i.e., *OR*, *AND*, and *XOR*), you can look into the conditional rendering at the code level. Once all the features are modeled, you can consult the product documentation (e.g., product wiki or requirements specification) to standardize the terminology of the identified features if needed.

  For *extractive adoption* (when you have existing system variants often arising from clone&own), you perform pairwise diffing. You can use a standard diffing tool or one that targets merging (e.g., Meld[1]), which is more extensive. You observe the differences, then try to understand why these differences are there in order to identify features. A typical technique is to ask those with detailed variant implementation knowledge various times why the difference exists. This leads to increasingly domain-oriented explanations, until the difference can be described by the presence or absence of a specific feature. In other words, you lift the implementation-level differences to the domain. You can refine the domain-oriented features later by following the same procedure as prescribed in reactive adoption, that is, by looking into the codebase of the different variants, and consulting other sources such as commit messages, pull requests, user stories, and product documentation.

- ⚙ **D1.2: Top-down feature identification** Here, you extract features by interviewing experts of different kinds. This activity is mandatory if there is no software system in place.
  Description: This sub-activity is usually the responsibility of dedicated domain analysis [11] and product-line-scoping methods [22, 9, 10]. Product-line-scoping methods, such as PuLSE-Eco [2], systematically select and prioritize the features that an organization wants to realize. These should bring an economic benefit for the organization and be in line with its business strategy (e.g., considering vision, strategy, finance, and commercial aspects). To find suitable feature abstractions, you can look at the *capabilities* the feature should provide. Capabilities are high-level abstractions over features that represent the functionality the feature will provide. Use a one-to-one interview format to elicit features from experts, which not only bypasses group meeting setup and coordination challenges, but also incorporates varying perspectives. Instead of starting from scratch and evolving the feature model incrementally after each interview, we suggest creating independent feature models, one per expert, and merging them. This eliminates the effect of different experts' perspectives on each other and allows diverse inputs. You should keep the number of consulted experts low in order to scale the interviews and prevent the challenges when merging multiple feature

[1]`https://meldmerge.org`

models. Alternatively. you can conduct interviews by grouping the same types of experts into one interview (e.g., one interview with developers and another with product managers). This will help eliminate conflicting viewpoints and lead to a lower number of feature models to be merged.

Outcomes: At the end of feature identification, you should have a tentative list of features as well as the relationships between features. You should also have the information about which features are mandatory and which are optional. After identification, the feature needs to be approved in some way by your organization. This approval process can be part of the established forum and workshop format (cf. page 4). Once approved, you can add it to the feature model (see activity Add Features below). It should also be documented.

The next question is *whether you need to identify and model cross-tree constraints between features*. Many constraints will already be reflected in the feature hierarchy and in feature groups, or as mandatory features. In any case, these constraints need to reflect the semantics of how you can combine features via the assets they map to. For instance, when you combine features into an OR group, but the system does not build or crashes when you select more than one of these features, then an XOR group would properly constrain the features. Beyond these constraints, which are easily visible in a feature model, you need to decide whether you need to model cross-tree constraints, which are often more intricate and challenge comprehension of the feature model.

Two principles help deciding. If the model is configured by experts in the organization, you can chose to avoid modeling those constraints. Since it is very expensive to accurately model them, and since the experts will likely know all the constraints, it usually will not pay off to model them. First, you often need a consultant to help the customer to decide which features are needed, so you can often save the effort of modeling constraints. Another strategy seen in practice is to maintain sets of tested configurations, which are evolved and maintained together with the model. Still, modeling constraints significantly enhances the value of the feature model, as it enables automated product derivation. Additionally, experts might eventually leave the company, in which case, the constraints that were not recorded might never be recovered. We therefore suggest companies to consider the trade-offs we discuss above while making the decision of whether they should identify and model the constraint or not.

Alternatively, if the main users of the feature model are end-users, then you should model the cross-tree constraints.[2]

( ) **D2: Identify constraints.** In this activity, you identify the constraints between the features identified in the previous activity. This is an optional activity.

Roles: The experts, modelers, and method experts should participate in this activity. Marketing experts can also optionally participate in this activity.

Description: All systems composed of parts have constraints over those parts, arising from domain, marketing, or technical restrictions. Since we abstract the selection of those parts to the selection of features (i.e., we mapped the parts to features), we lift those constraints over parts to constraints over features, which is not always trivial.

- **Code constraints:** Empirical studies show that in systems software, up to half of the constraints in a feature model can be found in the codebase and extracted using various program analysis techniques [20, 21]. Since such analysis techniques are difficult to set up and use, the developers should rather inform the modelers about such constraints or declare them directly in the model. We distinguish between two major kinds of sources: the so-called *feature effect* and the prevention of *build- or runtime errors.* The former refers to the effect of enabling a feature in the feature model on the resulting variants. The latter are errors that can occur early when the system fails to preprocess, parse, compile, run, type-check, or link.
- **Domain constraints:** Such constraints arise from domain knowledge and are usually not contained in the codebase. Examples are dependencies among hardware devices, which are instead contained in documentation or in the experience and knowledge of domain experts or developers.

[2]This can easily be seen in the Linux kernel [23] and many other systems software projects [7]. The complexity of these models and the sheer number of their configurations demand that all constraints are modeled.

To some extent, these constraints can be found through testing the different combinations of hardware and then adding them. However, mostly they need to be provided by the domain experts.

- **Other constraints:** Further sources are marketing experts, who might want to limit feature combinations for business reasons, or to simplify feature selection for the customer. Constraints can also be used to partially configure a feature model, which is called staged configuration [8]. Finally, some feature-modeling tools allow specification of soft constraints, such as "recommends" [5].

From these sources of constraints, observe that, while code constraints are reflected in the codebase and could in principle be recovered, the other sources illustrate that feature models contain unique knowledge. Finally, when identifying constraints, it is normal that initially you are not aware of all the dependencies. In fact, it is often difficult to see them early on.[3] Finally, after identifying the constraints, document them in the model, ideally together with their rationales.

Outcomes: You should have a set of feature constraints along with the rationale behind each constraint at the end of this activity.

M: Modeling Activities

In the Modeling phase, the goal is to obtain a feature model based on the documented information about features and relationships in the previous phase (Domain Analysis and Scoping Activities). Another aim is to validate that the modeled features and their constraints are correct and lead to valid configurations.

A first question is *whether you want to physically separate the partitions of the envisioned model into different feature-model files or not.* If so, perform the following two activities, otherwise continue with Define Coarse Feature Hierarchy below. Still, even if you do not want to decompose and rather want to create one model, it can be beneficial to temporarily decompose into models representing different stakeholder-related features, to model them in isolation and later integrate them.

( ) **M1: Model modularization.** In this activity, it is expected that created feature models are decomposed into smaller, easier-to-manage models. It's worth noting this is an optional activity.
Roles: The method experts and modelers should participate in this activity.
Description: Decomposing a feature model into smaller ones has pros and cons. It facilitates distributed, independent model evolution and maintenance, eases version management, and discourages (or limits) constraints across the models. However, it also raises consistency issues. In contrast, not decomposing avoids the overhead of maintaining multiple model files and their inclusion in a central one, but large models quickly become unmanageable.

Whether you should decompose depends on multiple factors. First, it requires finding an easy decomposition of the feature-model hierarchy into coherent sub-trees. For instance, a sub-tree could contain features that are more implementation-oriented, and another one those representing user-visible characteristics. Other factors are the estimated software size and number of features.[4] The hierarchy of feature models sets up the first framework for the platform—it is an initial structure that helps with the modeling. This hierarchy can be distributed along the codebase (i.e., as in the Linux kernel) or organized in a dedicated folder structure. Model modularization has two sub-activities:

- **M1.1:Define structure of model files.** To decompose, you define a hierarchy of feature models, beginning with a root model. This model's top-level features then become root features in the decomposed model files. You carry out this sub-activity at the beginning.

- **M1.2Maintain consistency between model files.** To maintain consistency, you find features that participate in dependencies across the models, and then move them into a separate "interface" feature model. This practice isolates the inter-model dependencies and eases their

[3]This can also be seen in the Linux kernel [15]. There, when developers add new features, it is sometimes observable that they fix the dependencies in several subsequent commits.

[4]From our experience, large models with several hundreds of features are all modularized into multiple files (e.g., the Linux Kernel [23]). All commercial models we have seen with several hundred features were all split into multiple ones.

maintenance. You carry out this sub-activity during the actual modeling once you feel that the cross-model dependencies are getting out of hand.

Outcomes: You should have a hierarchy of feature models with a root feature model, and the cross-model dependencies in a separate feature model.

M2:Define coarse feature hierarchy. Here, an initial, high-level hierarchy of features is created. This is a mandatory activity.

Roles: Modelers should conduct this activity. If required, they can ask for clarifications from the experts and method experts.

Description: You start by creating an initial, coarse hierarchy of features. If you created multiple feature models, select the one whose features you think are most well-understood.

Start by defining feature groups, where you model features that belong to a horizontal domain or have a close relationship. Think how to navigate those groups and existing features in a better way. You maximize cohesion and minimize coupling with feature groups. Specifically, feature groups should represent related functionalities—these are within a group, while there is low coupling to other groups (so, no cross-tree constraints). In contrast, you use abstract or mandatory features for structuring the overall model.

Another strategy is to organize features into sub-trees that logically partition the domain. Thereby, you try to reduce the need for cross-tree constraints across those partitions (sub-trees), but rather keep constraints within them. In other words, you try to increase cohesion and reduce coupling.

When forming the hierarchy, it is useful to recall that the top-level features are more abstract and business-oriented, so that they can be communicated to customers. Intermediate features represent functional aspects. Towards the leaves, the features are more technical—often, you create a domain- and business-oriented feature and then, when actually implementing it, need to add more specific and perhaps technical sub-features. You try to avoid having many intermediate features, which are usually vague and difficult to understand for your stakeholders. After defining a coarse hierarchy, it will be iteratively refined in the next activity (Add Features).

Outcomes: You should have a coarse hierarchy of domain-oriented features that are grouped in a way to maximize cohesion and minimize coupling in terms of cross-model constraints.

M3: Add features. Here, the feature model obtained as a result of the previous activity is refined and extended. This is a mandatory activity.

Roles: Modelers should conduct this activity. If required, they can ask for clarifications from the experts and method experts.

Description: While identifying features, you extend and refine your model. The new features will either already exist in the model, or you need to add them at relevant places in the model.

Since you always want to limit the number of features, you should first look for features that are similar and ask yourself whether an existing feature can be adjusted. You also do that because there is always the cost of a new feature to consider, and you want to avoid a growing pool of features. When placing the feature in the hierarchy, its location should "feel right" to the involved stakeholders, and as such, a discussion among them might be necessary.

Finally, define the relevant meta-data (e.g., feature title and short description); especially define default feature values, which substantially eases creating a feature-model configuration (making deriving a product a *reconfiguration* problem). Alternatively, you can define default configurations of the feature model once a sizable number of features have been modeled. However, since configuration is typically *reconfiguration* [7] (where an initial configuration is created based on default feature values with the the help of a non-trivial algorithm and then modified to reach a desired configuration), defining default feature values is likely more beneficial. Further meta-data that might be relevant in your organization could be the rationale why the feature was added, the feature owner (if this role exists) or party responsible for the feature, or so-called visibility conditions [7].

Outcomes: You should have a refined feature model with no duplicate features modeling the same functionality. Their meta-data should be documented.

(⚙) **M4: Model constraints.** In this activity, if you decided to identify and model constraints (recall the question and discussion on Page 7), then, conduct this optional activity.
Roles: Modelers should conduct this activity. If required, they can ask for clarifications from the experts and method experts.
Description: Declaring dependencies between features might require regrouping of features, removing the dependency, or extracting the dependencies into an interface feature model. So, you should always evaluate whether you really need to define those dependencies.

You should avoid complex constraints, which typically come in the form of Boolean expressions. Such constraints challenge model comprehension, maintenance, and evolution. You first try to model constraints using the feature hierarchy and other graphical elements from feature models (e.g., mandatory features or feature groups). In fact, an indicator of a good feature hierarchy is a low ratio of cross-tree constraints. If you still cannot restrict the remaining cross-tree constraints to simple binary dependencies (e.g., *required* or *excludes*), you can also put some constraints into the presence conditions of the variation points in your codebase, which keeps the model clean at the cost of a slightly more complex mapping between features and software assets. You can already see if the constraints work properly by experimenting with product derivation (cf activity Perform Product Derivation) as a quick feedback loop.

The source of the constraint (cf. activity Identify Constraints) gives you an indication on how to model it. Interestingly, constraints arising from the source we called *feature effect* are mostly reflected in the feature hierarchy. This makes a lot of sense when you remember that a feature always implies its parent in a feature model, enforcing that the sub-feature has an effect. Constraints preventing build and runtime errors are rather seen in cross-tree constraints or feature groups.
Outcomes: You should have a set of constraints that are ideally minimal and simplistic.

⚙ **M5: Merging multiple feature models.** Here, you merge the multiple feature models created through multiple interviews as well as different strategies for feature identification (cf. activity Identify Features). This is a mandatory activity.
Roles: Modelers should conduct this activity. If required, they can ask for clarifications from the experts and method experts.
Description: As mentioned above (page 5), conducting one-to-one interviews leads to multiple feature models. If that is the case, you need to merge the feature models. The following conflicts can occur. Conflicts regarding terminology can be resolved using the product documentation (cf. activity Unify Domain Terminology). For conflicts in feature relationships and constraints, you prioritize the stakeholders who will likely use the feature model. Thus, in a top-down modeling scenario, the onus to select the best model candidates to be merged from the set of valid feature models available, lies on the stakeholders identified in activity **P1**. Whereas in a bottom-up modeling scenario, valid model selections are delegated specifically to the developers. In the event of any contention relating to the validity of models produced thereafter, the final selection authority lies in the hands of the feature modeling domain expert in charge of the process. Especially when merging models from the top-down and bottom-up analyses, there are likely features in the latter that refine features from the former. Of course, it is natural to defer some decisions until the validation activity (page 11) and factor in experts' opinions. We recommend conducting the merging iteratively and tracking it with a version-control system (page 12).
Outcomes: You should have a consolidated feature model that merges feature models resulting from different sources.

(⚙) **M6: Define views.** Here, views are defined over the feature model for different purposes, such as partial configurations. This is an optional activity.
Roles: Experts and modelers should participate in this activity. It is recommended that model users also participate in this activity to define views that would benefit them.
Description: In addition to model modularization, some feature-modeling tools allow creating views, e.g., through filters or partial configurations, sometimes also called profiles. Views represent a subset of the feature model, and employed to facilitate the configuration of systems modeled in large feature models. For multiple feature models (e.g., created as a result of decomposition),

you can create multiple views and combine the partial configurations from each one to get a full configuration.
Outcomes: You should have a definition of views representing a smaller subset of the features in the feature model.

M7: Validation. Next, the feature model created is validated by the modellers in different ways. This is a mandatory, composite activity.
Roles: All roles should participate in this activity.
Description: After the modeling activities, it is time to check that the modeling was correct in the eyes of the stakeholders. After changes during evolution and maintenance, you should use the following ways of validation, especially the last one, regression testing.

- **M7.1: Stakeholder Reviewing.** In the workshop format established during the planning phase, various stakeholders should be invited to validate the model. We advise that different domain experts participate, given their individual area of expertise. They can validate that the right features and constraints were identified and modeled correctly, and they can advise on feature names and whether the hierarchy is intuitive. It is also beneficial when experts who did not participate in the modeling take part—among others, to comment on the intuitiveness of the model.
- ( ) **M7.2: Perform Product Derivations.** When one of the model purposes is to support product derivation, you should let the relevant stakeholders perform it for some variants. This can be done in the workshop format established. Obviously, the experience will be different than before, which was mostly manual. So, the stakeholders will select features in a certain order, and by doing so, they will be able to tell the modeler whether it feels right and whether it will be effective. As for which variants to derive, you should do that for existing ones, but also derive at least one that never existed before, reinforcing the benefit of a platform with automated product derivation through configuration.
- **M7.3:Regression Testing.** When iteratively creating the model, as well as maintaining and evolving it, you can easily break existing configurations. Many of the established feature-modeling tools provide some automated analysis that tells whether a change to the model has an effect on existing configurations. These analyses are confined to the feature model, but it is often desired to analyze the effect on the actual variants [19]. This requires creating regression tests using typical testing methods (e.g., unit tests). These should be given different configurations, ensuring the coverage of feature configurations that cover variants that are in use, ideally on the customer side. Knowing those requires either tracking such configurations or obtaining expert knowledge from the developers implementing the software. For instance, a developer usually knows from experience which features might interact and should be tested for certain modules.

Outcomes: You should have validated the constructed feature model and identified potential problems in the intuitiveness and correctness (with respect to configurability). You can solve the problems right-away or later in the Maintenance and Evolution phase (explained below).

ME: Maintenance and Evolution Activities

To evolve the model, you still apply the activities from the previous two phases (Domain Analysis and Scoping, and Modeling Activities). Especially the established workshop and forum come in handy here. Still, while many stakeholders are involved, one or only a few of them should ultimately control the model and make changes. Feature models are brittle assets and need to be evolved with care, to avoid inconsistencies that would have an impact on many different variants. In this light, it is also important to regularly perform the validation activities (cf. page 11). The following activities additionally support evolving the model, as well as maintaining it. Notably, since maintenance and evolution are both long-term and continuous activities, we do not specify any fixed outcomes here. Additionally, all three activities presented below can be performed by any role. The latter two activities do not follow an order and can be performed in any order. Lastly, all three sub-activities in this phase are mandatory.

ME1: Model version control. Tracking the evolution of the feature model is core. You should version the feature model in its entirety. While keeping an overview with a more fine-grained way of versioning is already difficult, the main reason is that individual features are not units of deployment or packaging, but whole system variants are. As such, it is more relevant to go back to such whole snapshots instead of individual feature versions.

ME2: Remove features. Performing this activity is necessary from time to time, but surprisingly difficult. Many companies therefore avoid removing features. However, for long-living platforms, removal is necessary to reduce the maintenance overhead and system complexity. Feature removal should be discussed in the established workshop or forum format. Once decided, a strategy is to remove the feature step-wise. If supported by the modeling tool, the feature is first flagged as deprecated and its default value changed to false. The next step is to make the feature a dead feature via constraints, so that it cannot be selected anymore. The final step is to remove the feature from model and software.

ME3: Optimizations. Of course, over time, the constraints become more intricate, and the hierarchy might not be as intuitive as necessary. So, an important activity is to optimize the hierarchy and the constraints. However, without proper tool support for refactoring, it is relatively easy to invalidate existing variants, which should be avoided. Performing the validation activities is crucial (page 11).

Appendix

We briefly illustrate a feature model in Fig. 2 and the different usages of feature models in Fig. 3. For details, we refer to the respective literature [11, 1, 18, 25].

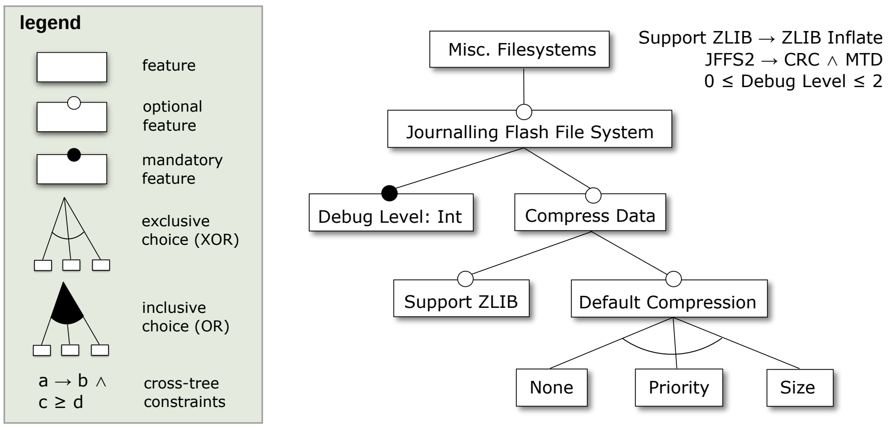

Figure 2: *Feature model example*

Tool Suggestions. Overall, we believe that textual modeling languages are the most essential tool that stakeholders would need to apply FM-PRO effectively. Textual modeling languages offer several advantages for feature modeling, particularly in industrial settings. They are easier to edit and manage, especially when integrated with version-control systems [3]. Clafer[5] and UVL[6] are two notable languages, both supporting basic and advanced feature modeling concepts. UVL, being more recent, has better tool support, while Clafer's concise syntax is easier to adopt in industry. Clafer also has an IDE plugin for creating and maintaining feature models and traceability links, though it currently lacks constraint support [17]. Our experience shows that UVL's more complex syntax can be harder to understand for developers. Therefore, we suggest to use Clafer when applying FM-PRO, however UVL is also a suitable option.

[5] `https://www.clafer.org/`
[6] `https://universal-variability-language.github.io/`

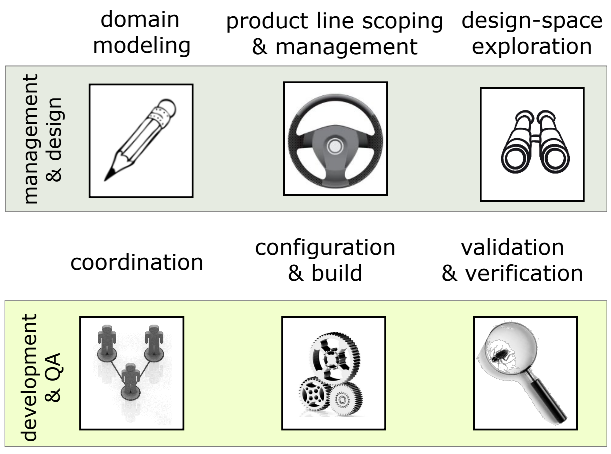

Figure 3: *Range of feature-model usages*

In addition to this, we recommend the following tools, which can be used in support of clafer at each phase of our feature modeling process. For the premodeling phase, **Confluence**[7] can be used to collaboratively create and organise knowledge-based ideas on projects. Next, on the domain analysis and scoping side of things, **Miro**[8]; which can also be used collaboratively to perform visually intuitive mind-mapping activities may suffice. In addition to this, **FeatureIDE**[9]; an extensible framework for feature-oriented software development, can provide modeling support for features, their relationships and constraints. Finally, for the maintenance and evolution phase of FM-PRO, **git** provides worthwhile support for model versioning, refactoring and general change management.

References

[1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.*

[2] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. 1999. PuLSE: A methodology to develop software product lines. In *Proceedings of the 1999 Symposium on Software Reusability.*

[3] Maurice H ter Beek, Klaus Schmid, and Holger Eichelberger. 2019. Textual variability modeling languages: an overview and considerations. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B.* 151–157.

[4] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a feature? a qualitative study of features in industrial software product lines. In *Proceedings of the 19th international conference on software product line.* 16–25.

[5] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three cases of feature-based variability modeling in industry. In *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28–October 3, 2014. Proceedings 17.* Springer, 302–319.

[6] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A survey of variability modeling in industrial practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems.* 1–8.

[7] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. [n. d.]. A study of variability models and languages in the systems software domain. 39, 12 ([n. d.]), 1611–1640.

[8] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. [n. d.]. Staged configuration through specialization and multilevel configuration of feature models. 10, 2 ([n. d.]), 143–169.

[9] Isabel John and Michael Eisenbarth. 2009. A decade of scoping: A survey. In *Proceedings of the 13th International Software Product Line Conference.* 31–40.

[10] Isabel John, Jens Knodel, Theresa Lehner, and Dirk Muthig. 2006. A practical guide to product line scoping. In *10th International Software Product Line Conference (SPLC'06).* IEEE, 3–12.

[11] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. Software Engineering Institute, Carnegie Mellon University.

[12] Charles Krueger. 2002. Variation Management for Software Production Lines. In *Proceedings of the Second International Conference on Software Product Lines (SPLC 2).*

[13] Jacob Krueger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an

[7]`https://www.atlassian.com/software/confluence`

[8]`https://miro.com/`

[9]`https://featureide.github.io/`

Integrated Platform. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*.

[14] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting rid of clone-and-own: Moving to a software product line for temperature monitoring. In *SPLC*.

[15] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. [n. d.]. Evolution of the Linux Kernel Variability Model. In *SPLC* (2010) *(Lecture Notes in Computer Science, Vol. 6287)*, Jan Bosch and Jaejoon Lee (Eds.). Springer, 136–150.

[16] Wardah Mahmood, Daniel Strueber, Thorsten Berger, Ralf Laemmel, and Mukelabai Mukelabai. 2021. Seamless variability management with the virtual platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1658–1670.

[17] Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. 2021. Hans: Ide-based editing support for embedded feature annotations. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B*. 28–31.

[18] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. [n. d.]. *Mastering Software Variability with FeatureIDE.* Springer.

[19] Mukelabai Mukelabai, Damir Nesic, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[20] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th international conference on software engineering*. 140–151.

[21] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[22] Klaus Schmid. 2000. Scoping software product lines. In *Software Product Lines*. Springer, 513–532.

[23] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. *VaMoS* 10, 10 (2010), 45–51.

[24] Daniel Strueber, Mukelabai Mukelabai, Jacob Krueger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *23rd International Systems and Software Product Line Conference (SPLC)*.

[25] Andrzej Wasowski and Thorsten Berger. 2023. *Domain-specific Languages: Effective Modeling, Automation, and Reuse*. Springer.