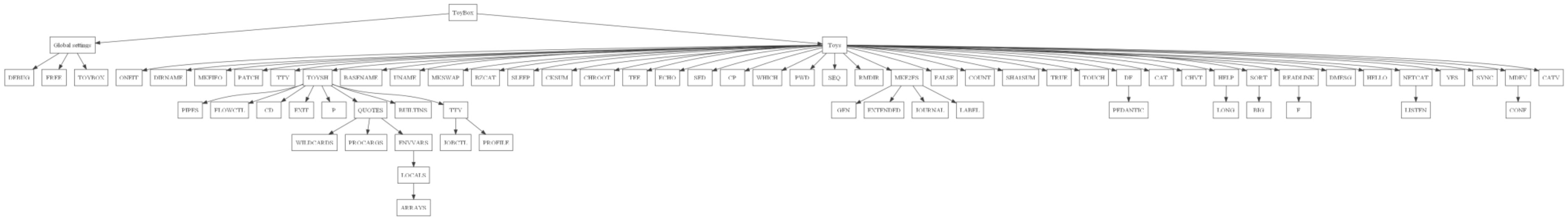


Usage Scenarios for a Common Feature Modeling Language

Thorsten Berger and Philippe Collet

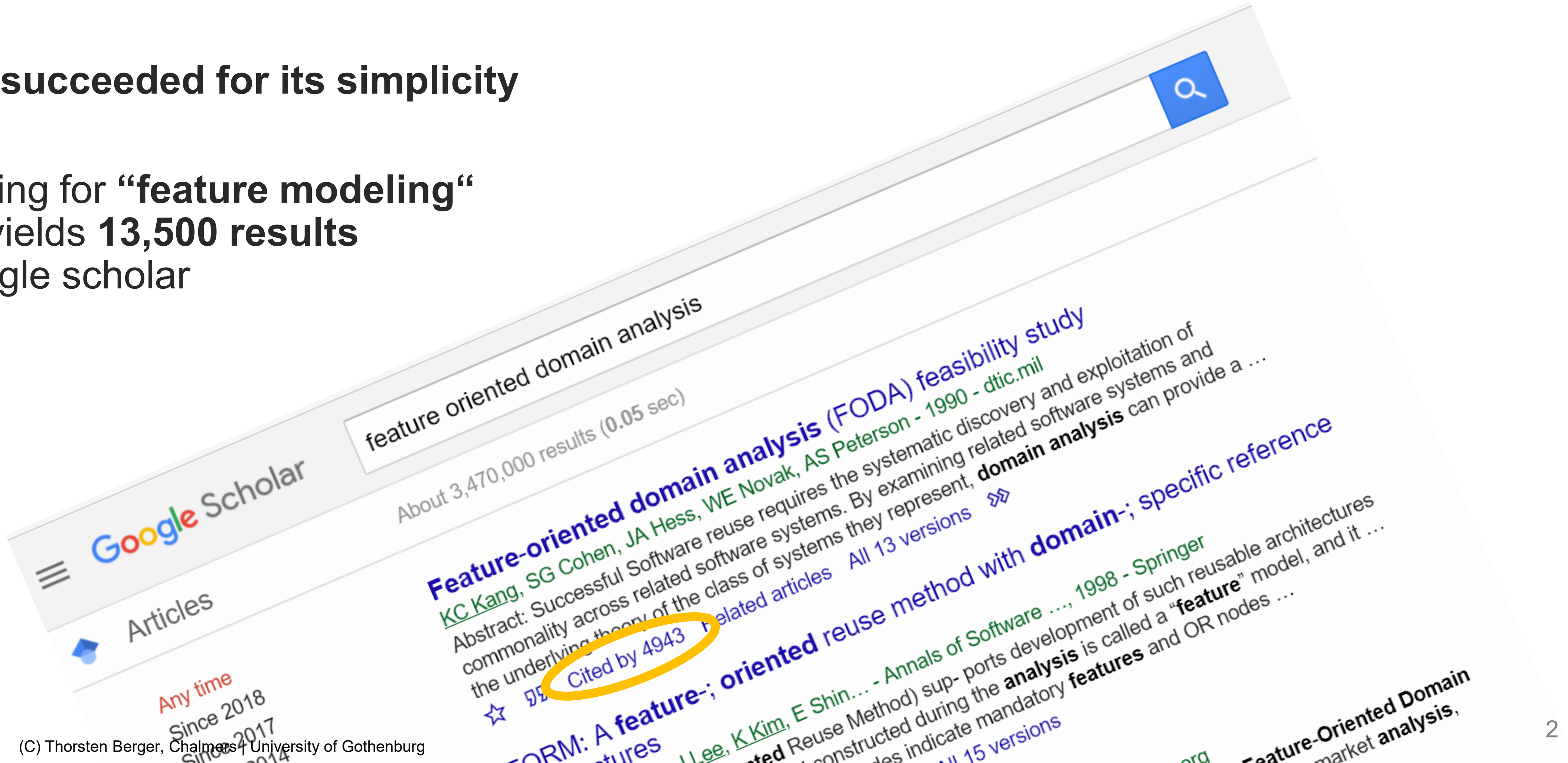
Membre de UNIVERSITÉ CÔTE D'AZUR

feature modeling

Feature Oriented Domain Analysis (FODA) by Kang et al. 1990

FODA succeeded for its simplicity

searching for “**feature modeling**“
alone yields **13,500 results**
on google scholar



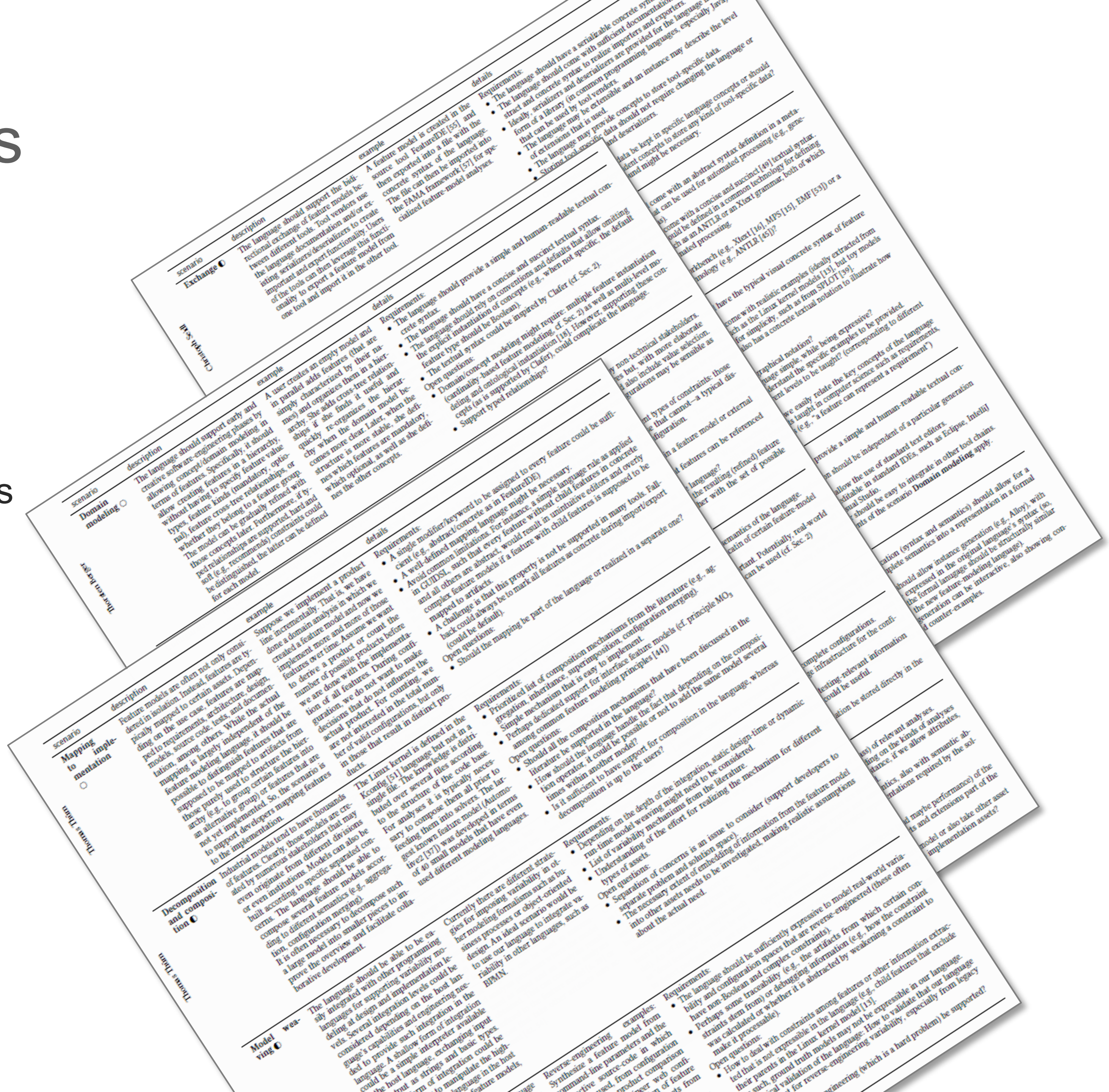
Methodology

- initial meeting at SPLC’18 in Gothenburg
  - agreement on scenario-driven methodology, brainstorming
  - first set of 15 usage scenarios (voted)
  - two researchers assigned (typically, one writing, another proofreading)
- scenarios described mid September to mid October 2018
- survey to evaluate scenario clarity and usefulness
  - created by David
  - distributed via the initiative’s mailing list
  - 15 responses
- **analysis and refinement**
  - **upon results, refined and extended the scenario**
  - **also removed and added (very) few**

14 refined scenarios

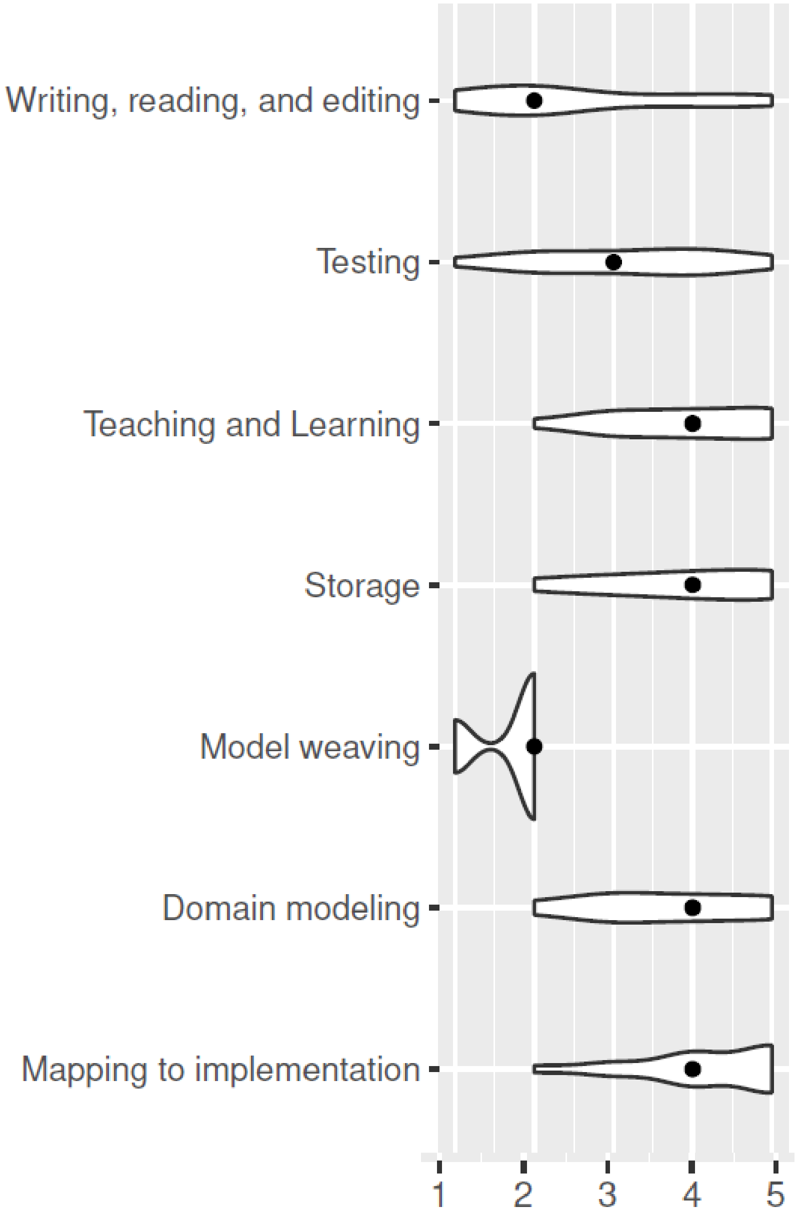
scenario:

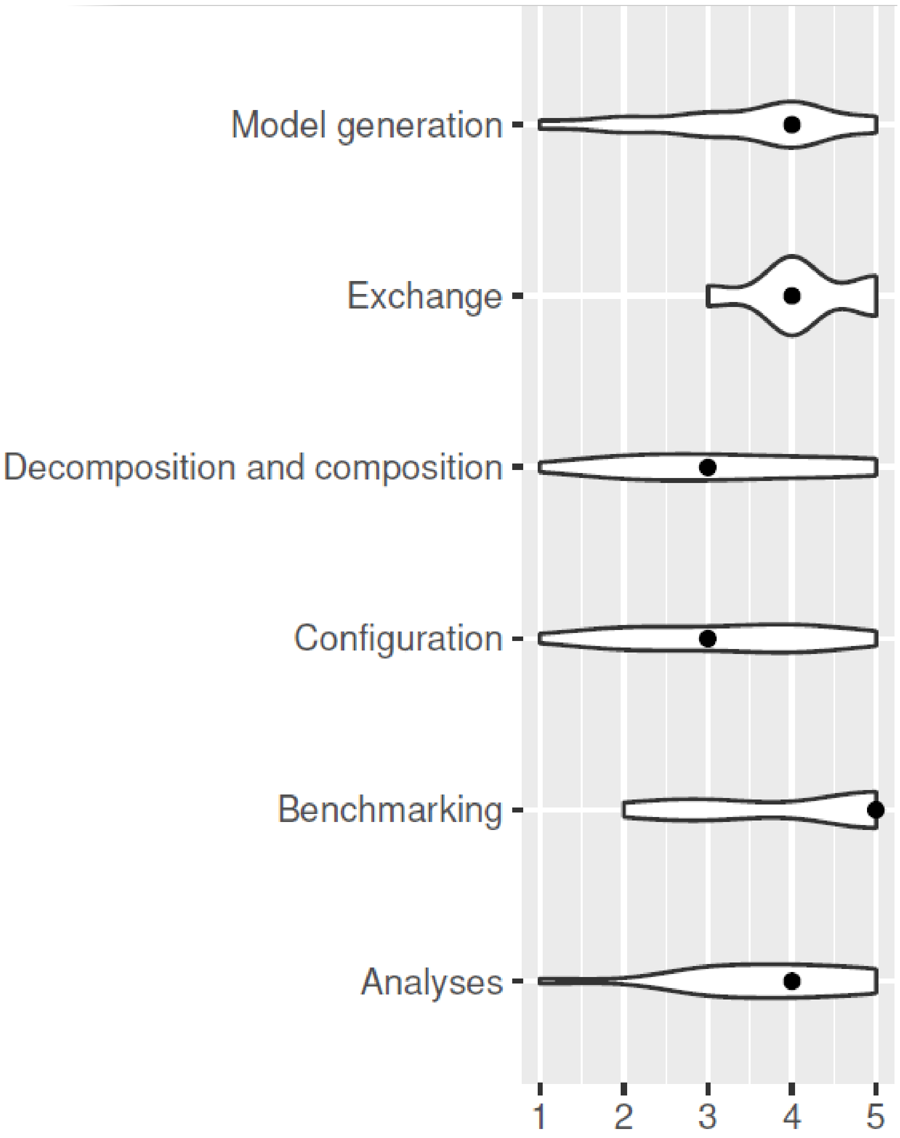
- name
- short description
- example
- notes (e.g., specific requirements or open questions)



usefulness/priority

What is the usefulness/priority of the scenario?
1 (not useful at all), 2 (not useful), 3 (more or less), 4 (useful), 5 (very useful).





a preliminary roadmap

idea: incrementally build the language to make progress

second evaluation, of the refined and extended usage scenarios?

re-open the discussion about further scenarios that need to be realized

- e.g., collaborative creation of feature models discussed at workshop, but not formulated

devise first set of features from scenarios perceived most useful:

- **Exchange, Storage,**
- **Domain Modeling, Teaching and Learning,**
- **Mapping to implementation, Model generation,**
- **Benchmarking, Analyses**

scenario	description	example	details
Exchange ◐	The language should support the bidirectional exchange of feature models between different tools. Tool vendors use the language documentation and/or existing serializers/deserializers to create important and expert functionality. Users of the tools can then leverage this functionality to export a feature model from one tool and import it in the other tool.	A feature model is created in the source tool FeatureIDE [55] and then exported into a file with the concrete syntax of the language. The file can then be imported into the FAMA framework [57] for specialized feature-model analyses.	Requirements: • The language should have a serializable concrete syntax. • The language should come with sufficient documentation about its abstract and concrete syntax to realize importers and exporters. • Ideally, serializers and deserializers are provided for the language in the form of a library (in common programming languages, especially Java) that can be used by tool vendors. • The language may be extensible and an instance may describe the level of extensions that is used. • The language may provide concepts to store tool-specific data. • Storing tool-specific data should not require changing the language or provided serializers and deserializers. Open questions: • Should tool-specific data be kept in specific language concepts or should there be tool-independent concepts to store any kind of tool-specific data? Finding a middle ground might be necessary.
Storage ●	The language should allow tools to efficiently store and load feature models. Tools can use the language and its concrete syntax as the primary means to store models. Tool vendors leverage the language specification to realize fast storage and loading of models. Two subscenarios are possible: (i) the model is stored in a database, and (ii) the model is stored in a textual representation.	Consider a new product line tool that needs to store feature models. The tool vendor can develop its persistence layer by creating leveraging the language specification (i.e., the abstract syntax definition) to derive a database schema and generate CRUD functionality as well as initialize the database.	Requirements: • The language should come with an abstract syntax definition in a meta-modeling notation that can be used for automated processing (e.g., generate database schemas). • The language should come with a concise and succinct [49] textual syntax. • The textual syntax should be defined in a common technology for defining concrete syntaxes, such as an ANTLR or an Xtext grammar, both of which can be used for automated processing. Open questions: • Select a language workbench (e.g., Xtext [16], MPS [15], EMF [53]) or a parser-generator technology (e.g., ANTLR [45])?
Teaching and learning ◐	The language should be easily usable for teaching. Specifically, it should be possible to describe the language within a few slides, using concepts typically taught in computer science education (e.g., types, grammars, meta-modeling). Furthermore, the language's concepts should align well with the typical and established concepts (cf. Sec. 2) that have been introduced in the product-line community and are typically taught in SPL courses (features, attributes, constraints).	The teacher describes the language with fewer than a dozen slides, and the students are able to read and write simple examples afterwards.	Requirements: • The language should have the typical visual concrete syntax of feature models. • The language should come with realistic examples (ideally extracted from real-world models, such as the Linux kernel models [13], but toy models can also be provided for simplicity, such as from SPLOT [39]. • Ideally, the language also has a concrete textual notation to illustrate how to scale models. Open questions: • Teach the textual or graphical notation? • How to keep the language simple, while being expressive? • There is a need to understand the specific examples to be provided. • Should there be different levels to be taught? (corresponding to different levels in teaching) • When teaching, can we easily relate the key concepts of the language with standard concepts taught in computer science such as requirements, components, modules (e.g., "a feature can represent a requirement")

Model generation ◐	Model generation (a.k.a., instance generation) automatically creates instances (models) of the language, typically aiming at instances with certain properties, such as size, coverage of language concepts, or other structural characteristics (e.g., cross-tree constraints ratio [8, 40, 50]). Tool developers can use it to generate a set of models, useful for functional testing and performance testing of the different tools supporting the language.	A tool developer launches the instance generation tool, inputs the desired properties of the model to be generated, and obtains the desired model(s).	Requirements: • The language specification (syntax and semantics) should allow for a translation of the complete semantics into a representation in a formal language. • The formal language should allow instance generation (e.g., Alloy), with instances that can be expressed in the original language's syntax (so, instantiated model in the formal lanugage should be structurally similar to the target model in the new feature-modeling language). • Ideally, the instance generation can be interactive, also showing conflicting constraints and counter-examples.
Domain modeling ○	The language should support early and creative software-engineering phases by allowing concept/domain modeling in terms of features. Specifically, it should allow creating features in a hierarchy, without having to specify feature value types, feature kinds (mandatory, optional), feature cross-tree relationships, or whether they belong to a feature group. The model can be gradually refined with those concepts later. Furthermore, if typed relationships are supported, hard and soft (e.g., recommends) constraints could be distinguished, the latter can be defined for each model.	A user creates an empty model and in parallel adds features (that are simply characterized by their names) and organizes them in a hierarchy. She adds cross-tree relationships if she finds it useful and quickly re-organizes the hierarchy when the domain model becomes more clear. Later, when the structure is more stable, she defines which features are mandatory, which optional, as well as she defines the other concepts.	Requirements: • The language should provide a simple and human-readable textual concrete syntax. • The language should have a concise and succinct textual syntax. • The language should rely on conventions and defaults that allow omitting the explicit instantiation of concepts (e.g., when not specific, the default feature type should be Boolean). • The textual syntax could be inspired by Clafer (cf. Sec. 2). Open questions: • Domain/concept modeling might require: multiple feature instantiation (cardinality-based feature modeling, cf. Sec. 2) as well as multi-level modeling and ontological instantiation [18]. However, supporting these concepts (as is supported by Clafer), could complicate the language. • Support typed relationships?
Analyses ◐	The language can be used in automated analysis processes where the model is used as input and an analysis result is obtained. This can comprise analyses confined to the feature model [6, 29] or those that take other artifacts into account [42, 54].	Consider a Linux distribution, such as Debian. Let us assume the packages (each representing a feature) and their dependencies are described using our language (or, more realistically, are transformed from Debian's manifests into a feature model). An off-the shelf analysis, such as "dead features" can then be used to detect packages that are not selectable.	Requirements: • Community agreement on a core set (or class) of relevant analyses. • Consider different solver strategies depending on the kinds of analyses and the constructs of the language. For instance, if we allow attributes, then, specific solver capabilities are needed. • Well-specified language syntax and semantics, also with semantic abstractions into the different logical representations required by the solvers. Open questions: • Is the representation of correspondence (and maybe performance) of the solving strategies to the different constructs and extensions part of the

Benchmarking ◐	The language should be designed for tool support, and several implementations are expected to be available. There should be a well-defined set of indicators to measure the performance of the most relevant operations (e.g., analysis, refactoring, configuration completion), so to be able to compare them. The benchmarking setup would allow to compare tool support execution times of these operations in isolation (e.g., without taking into account file loading or feature model parsing times when focusing on a reasoning operation).	The user loads the model with FAMA [57], Familiar [4] or Feature IDE [55] and executes the operation 'dead features,' also measuring the completion times. Then she knows which is the best tool for that operation and model. Each tool built upon the language can run the common benchmark and automatically produce an exploitable performance result.	Requirements: • Well-engineered and specified syntax and semantics of the language. • There should be an agreement on the specificatin of certain feature-model operations. • The availability of realistic models is important. Potentially, real-world models from the systems software domain can be used (cf. Sec. 2)
Mapping to implementation ○	Feature models are often not only considered in isolation. Instead, features are typically mapped to certain assets. Depending on the use case, features are mapped to requirements, architecture, design, models, source code, tests, and documentation, among others. While the actual mapping is largely independent of the feature modeling language, it should be possible to distinguish features that are supposed to be mapped to artifacts from those purely used to structure the hierarchy (e.g., to group certain features into an alternative group) or features that are not yet implemented. So, the scenario is to support developers mapping features to the implementation.	Suppose we implement a product line incrementally. That is, we have done a domain analysis in which we created a feature model and now we implement more and more of those features over time. Assume we want to derive a product or count the number of possible products before we are done with the implementation of all features. During configuration, we do not want to make decisions that do not influence the actual product. For counting, we are not interested in the total number of valid configurations, but only in those that result in distinct products.	Requirements: • A single modifier/keyword to be assigned to every feature could be sufficient (e.g., abstract/concrete as in FeatureIDE) • A well-defined mapping language might be necessary. • Avoid common limitations. For instance, a simple language rule as applied in GUIDSL, such that every feature without child features in concrete and all others are abstract, would result in unintuitive editors and overly complex feature models if a feature with child features is supposed to be mapped to artifacts. • A challenge is that this property is not be supported in many tools. Fallback could always be to mark all features as concrete during import/export (could be default). Open questions: • Should the mapping be part of the language or realized in a separate one?

a preliminary roadmap

Exchange:

a simple textual language seems to meet the scenario’s challenges

Storage:

realize using a common language workbench (e.g., Eclipse EMF with Xtext) or YAML/JSON technology

Domain Modeling:

capability to incrementally and partially create a feature model is needed

Teaching and Learning:

simplicity of the language for writing, editing, and configuring should be kept in mind.

Model generation, Benchmarking, and Analyses

could be easy to meet if propositional feature models chosen as first level of expressiveness

Mapping to implementation

not easy scenario to meet

still open problem, depending on types of artifacts and variability realization techniques

discussion

design and implement first kernel of functionalities at same time?

- Implementation enables scenario validation automatically (continuous integration)

for implementation, important design decisions:

- fluent API
- external or internal DSL, or
- clever combination

validation

- use scenario *Analyses* with its first example (dead-feature detection) as first validation
- discuss other useful analysis scenarios
- similarly, use the *Benchmarking* scenario? (first example is a benchmark for dead-feature computation)

initial kernel of a language

- strip down Clafer into language levels?

paper with detailed scenario descriptions:

http://www.cse.chalmers.se/~bergert/paper/2019-modevar-fml-scenarios.pdf

survey about refined scenarios (only 5 answers so far)

https://forms.gle/HaG2reNZwWKCMQzm7

also thanks to:
Mathieu Acher, Maurice Ter Beek, David Benavides, José A. Galindo, Rick Rabiser, Klaus Schmid, Thomas Thüm, and Tewfik Ziadi

Usage Scenarios for a Common Feature Modeling Language

Thorsten Berger and Philippe Collet
<thorsten.berger@chalmers.se>,
<philippe.collet@univ-cotedazur.fr>

Université Nice Sophia Antipolis
Membre de UNIVERSITÉ CÔTE D'AZUR
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG