Tool Interview – ECCO

1 INITIAL CONCEPTUAL MODEL

The initial conceptual model describes essential concepts (or a superset of it) for modeling variability of a software system in space and time and shall subsume functionality related it. Additionally, the model unifies those concepts to represent revisions of variable system parts. The conceptual model follows an open-world assumption (descriptive) instead of a closed-world assumption (prescriptive) as metamodels commonly do. In Table 1 we provide a definition of the involved concepts.

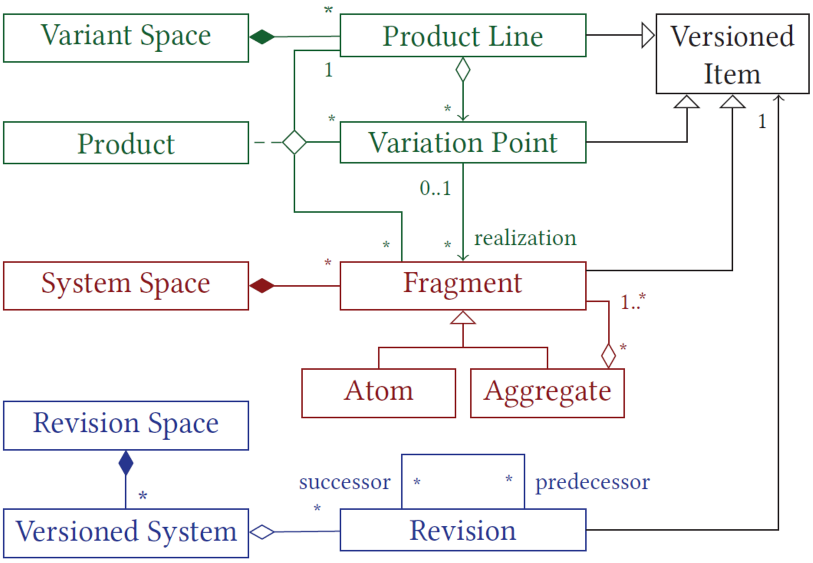

Figure 1: The Initial Conceptual Model with essential and combining Concepts for Variability in Space and Time.

Table 1: Definition of concepts in the Conceptual Model.

Concept	**Direct relation to other Concepts**	**Definition**
Fragment	*Variation Point, Product*	*Fragments* are the essential concept to describe a system on realization level. A *Fragment* can either be an atom or an aggregate, e.g. a single file, character or the node of an AST. A hierarchical structure of containments is not enforced but instead *Fragments* can be composed to various combinations.
Product Line	*Variation Point, Versioned Item*	A *Product Line* represents the configurable space regarding spatial variability and is composed of a system's *Variation Points*.
Variation Point	*Product Line, Fragment,*	A *Variation Point* expresses the variability of a system by representing an option set for variation of the *Product Line*.

	Product, *Versioned Item*	A *Variation Point* can either be explicit (e.g., if-defs or a plug-in system with a compositional variability realization mechanism) or implicit (a reference between a feature module and fragment represents the implicit variation points, therefore the fragment is not aware of its variation e.g., FOP, AOP, delta modeling).
Product	*Product Line,* *Variation point,* *Fragment*	A *Product* is fully specified if all existing *Variation Points* in the *Product Line* are bound to *Fragments* or *Variation Points* are not bound explicitly, e.g., if a feature is optional and not selected for product (hence, all to a configuration relevant *Variation Points* are bound to fragments). A partial *Product* does not require the binding of every *Variation Point*.
Revision	*Versioned Item*	A *Revision* of the *Fragment* evolves along the time dimension and is intended to supersede its predecessor by an increment, e.g., due to a bug fix or refactoring.
Versioned System	*Revision*	A *Versioned System* represents the configurable space regarding temporal variability. It is composed of a system‘s revisions.
Versioned Item	*Revision*	The *Versioned Item* represents versioning of the introduced concepts for *Fragment*, *Variation Point* and *Product Line* by putting them under revision control.

Table 2: Particular Relations of the Conceptual Model.

Relation	Direct relation to Concepts	Definition
Realization	*Variation Point,* *Fragment*	Each *Variation Point* has a set of possible options for variation whereby each option is realized by *Fragments*.
Configuration	*Product Line,* *Variation Point,* *Fragment*	A *Configuration* defines one particular *Product* of a *Product Line* by resolving the variability of a *Product Line*, i.e., binding all relevant *Variation Points* of a *Product Line* to *Fragments*.
Branching / Merging	*Revision*	To represent *branching* (which is considered a temporary divergence for concurrent development) along with *merging*, multiple (direct) successors and predecessors relate to a revision. This relation gives rise to a revision graph, which is a directed acyclic graph where each node represents a unique revision.

2 INTERVIEWS

Please inspect

1. If
2. and if yes, how

concepts of the conceptual model are represented by constructs used in your tool. Therefore, the representation of each concept in the tool and their (direct) relation to other constructs is considered separately.

Table 3: Concept Mapping between Conceptual Model and Tool.

Concept	Representation of Concept in Tool	Relation to other Constructs
Fragment	Equivalent notion: *Artifact* Artifacts represent the implementation of a single feature or a feature interaction and can be anything (whole file, source code (text), models (AST Nodes), test cases or requirements, etc.). Artifacts are composed in a generic tree structure representing their hierarchy and order. However, cross-references can be used for graph representations (like in Ecore). An artifact in ECCO is immutable (equivalent to fragment of the Conceptual Model).	An artifact may reference another fragment, e.g., an import of a class.
Product Line	There is no explicit representation of a product line (e.g., feature model) in ECCO. Instead, an unconstrained feature set is used. All features are implicitly considered to be optional (it is up to tjhe developer how to compose & checkout).	Variation Point
Variation Point	Equivalent notion: *Feature*. No explicit representation of variation point in problem space or solution space. However, in solution space, a variation point could possibly be mapped to associations that contain Boolean formulas (presence conditions) that map feature revisions and their	Feature

	interaction (e.g., inclusions or exclusions between features) to relevant artifacts (aka *fragments* in Conceptual Model). *association1 = (A.1 && B.2, { …, fragment1, fragment2, … })* *association2 = (A.2 && !C.5, {fragment 3})*	
Product	Equivalent notion: *Variant* A variant exists outside of ECCO scope and is the result of a configuration and the composition process. A Product is represented on two levels: A configuration is the conceptual level of a product of a product line, i.e., a selection of features in specific versions. A product on the realization level is referenced as a product, i.e., the fragments in the variation representing the selection of the configuration (in other words, the software system resulting from the configuration). A product on the realization level is represented by artifacts Folder, File (C++, Java,...).	-
Revision	Currently, there are no revision predecessors and successors (but could be made possible with order) → no DAG Fragments are mapped to revisions of features (The Conceptual Model currently supports the mapping of revisions to fragments).	Feature
Versioned System	Equivalent notion: *Repository* The repository represents a container for both variability in space (*product line* in CM) and in time (*versioned system* in CM). The Repository encompasses a collection of associations (presence condition (Boolean formula) + set of artifacts), and a list of revisions per features.	Fragment, Feature, Association
Versioned Item	Features	
Realization	Equivalent concept: *Associations* *Associations* (traces) represent the realization link between revisions of features and artifacts.	Artifacts, Revisions of Features

<table>
<tr><td></td><td>No Derivation Mechanism, internal derivation similar to FeatureHouse (superimposition of trees)
User-facing: Variants in, Variants out</td><td></td></tr>
<tr><td>Configuration</td><td>A configuration on conceptual level is a selection of features with exactly one version elected for each feature.</td><td>Revision, Feature</td></tr>
<tr><td>Branching / Merging</td><td>No DAG structure possible</td><td></td></tr>
<tr><td>Remarks</td><td colspan="2">Core Concepts of ECCO: Features, Commits, Associations, Artifacts, Artifact graph, Dependency Graph (aka Sequence Graph?), Commit Graph, Charts, Presence Table. View for Traces, View for Artifact Tree, View for Viewer

Questions / Consider in Conceptual Model:
How to handle revision scope? Globally or locally?
Add Fragment Space (for consistency)?
Is the association from product to fragment redundant (given by transitive relationship via variation point)?
Consider VTS for evaluation (Thorsten Berger)?</td></tr>
</table>

3 USE CASES

Please provide an overview of use cases that your tool addresses.

- Supports the practice of clone-and-own in software engineering
- → Combination of Clone-and-Own (variant-centric) and SPL (automated reuse)
- → Automatic localization and extraction of reusable artifacts from existing clone-and-own variants
- Manages different types of implementation artifacts
- Provides the check-out of one particular configuration → results in a simple view
- Operations: commit<conf>, checkout<conf>, fork<url>, push <features>, pull<features>

4 PREVIEW: SEMANTICS

The semantics of several concepts is only defined through the mechanisms that operate on them. For example, the configuration of a product from a product line, variation points and fragments is expressed in the conceptual model, but constraints that define which variation points and fragments may be selected have to be ensured by a configuration mechanism. The same applies to the generic concept of the *Versioned Item*. A mechanism that defines how the relation between revisions of product lines, variation points and fragments can be combined has to be defined. Designing such mechanisms, based on the conceptual model, is the next step towards a unifying concept for variability in space and time.

We consider semantics represented by the following mechanisms of a system that deal with variability in space and / or time:

1) *Analyses mechanisms* support the validity of:
   a. the variability model
   b. the configuration
   c. the fragment

2) The *mapping mechanism* that is used to resolve a configuration from a variability model to a set of realization artifacts

3) A *variability realization mechanism* assembles realization artifacts for a configuration in a particular manner (*annotative* variability, e.g. #ifdefs; *compositional* variability, e.g., feature-oriented programming; *transformational* variability, e.g., delta modeling).

In the following, please describe the semantics of your tool regarding the described mechanisms.

Analyses mechanisms **Variability model /** **Configuration:** To ensure the validity of a configuration (prior to product derivation based on the configuration), all associations that contain the selected features need to be satisfied. **Fragment:** Syntactical validity is ensured (e.g., method calls and fields, containments,…), otherwise a warning message occurs.

Mapping mechanism

As mapping mechanism, ECCO uses associations (aka traces) that contain Boolean formulas (conditions) that map feature revisions and their interaction to relevant modules (artifacts).

"For ordered nodes, a trace is more than just the information of whether an artifact is required for the implementation of a module or not. In Addition, the ordering of the artifacts must be considered. Therefore, for every set of ordered nodes, a sequence graph is maintained which is a partial order relation describing the order of the nodes direct children among all traces"

association1 = (A.1 && B.2, { …, fragment1, fragment2, … })
association2 = (A.2 && !C.5, {fragment 3})

Variability realization mechanism:

- No Derivation Mechanism. The internal derivation is similar to FeatureHouse (superimposition of trees).
- A product may be complete or incomplete. If it is incomplete: manual but tool-supported completion of the new checked-out variant.
- For unresolved dependencies between artifacts during composition, four options are offered (insert referenced artifact or association, remove referencing artifacts, leave the reference unresolved)
- Hints (Surplus artifacts that need to be manually removed; manual reordering of artifacts)

A. TABLE OF TABLES

B. REFERENCES

[1] S. Ananieva, T. Kehrer, H. Klare, A. Koziolek, H. Lönn, S. Ramesh, A. Burger, G. Taentzer and B. Westfechtel, "Towards a conceptual model for unifying variability in space and time," *Proceedings of the 2nd International Workshop on Variability and Evolution of Software-Intensive Systems,* 2019.

[2] G. Guizzardi, L. F. Pires and M. van Sinderen, "An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages," *Proceedings of the International Conference on Model Driven Engineering Languages and Systems,* 2005.