Tool Interview – SiPL

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.

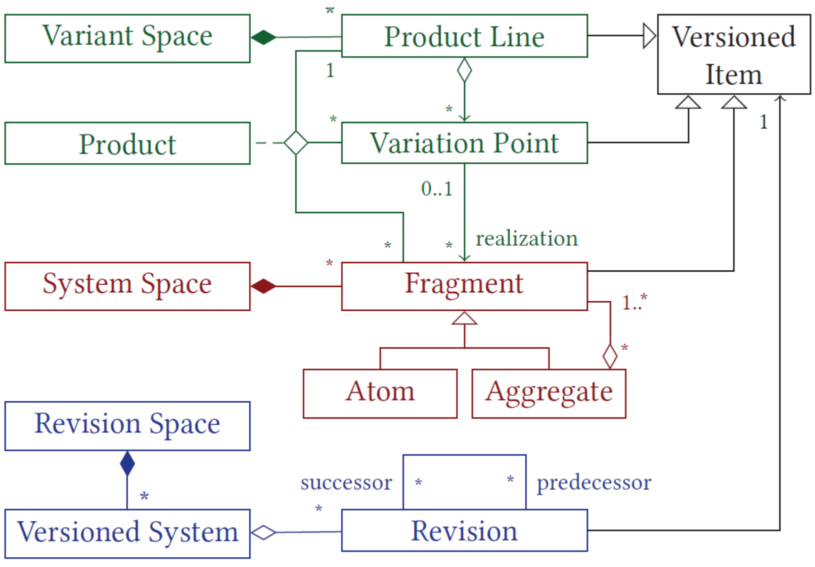

Figure 1: The Initial Conceptual Model with essential and combining Concepts for Variability in Space and Time.

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	A fragment can be any type of realization artifact and may span code, models, documentation etc. Due to technical reasons, a prerequisite is that these fragments have a representation based on EMF Ecore, i.e., a meta model that is suitable to represent concrete fragments as models of that meta model. There are no further requirements on, e.g., the structure of the meta model or regarding marking of variation points. A fragment composes a system, e.g., a product in SPL while delta modules modify fragments.	A fragment may reference another fragment, e.g., an import of a class.
Product Line	A product line is represented by a variability model. The variability model is assumed to be a propositional formula over the set of features. SiPL can be extended by arbitrary variability adapters which convert a proprietary variability model into a propositional formula. An adapter for concering a feature diagram expressed in FeatureIDE is already available. The problem space comprises the variability model while the solution space encompasses the delta module set.	Feature, boolean operator
Variation Point	Problem Space: Encapsulated within the variability model (the set of implicit variation points is constrained by configuration via feature model + application conditions of delta modules)	Feature

Product	A product is based on a valid configuration and is composed by fragments of any type of realization, e.g., state machine + java code. A product can either be created from scratch (pure delta modeling) or based on another existing product (a core model for the SPL needs to be defined always – may be empty or not).	
Revision	/	
Versioned System	Versioning could be made possible via a version control system (e.g., Git or SVN).	
Versioned Item	/	
Realization	All application conditions together represent the mapping of the problem space onto the solution space. This mapping does not need to be a 1 : 1 relation, i.e. a delta-module can contribute to the implementation of several features, and a feature can be implemented by several delta-modules.	Feature, Delta Module
Configuration	A configuration is performed interactively by selecting desired features or represented by a configuration file which is a simple list of features.	Feature
Branching / Merging	/	
Remarks	Consider following in the Conceptual Model? • Definition of fragment structure may be superfluous in the Conceptual Model (structure depends on tool realization). Otherwise: add, for instance, self-reference for fragment to support graph structure ? • Incorporate problem and solution space in the Conceptual Model? About SiPL: - SiPL derives delta modules from a model difference. Therefore. SiPL is integrated with the model differencing framework SiLift which provides advanced differencing and patching facilities based on graph transformation concepts. This way, the difference between an origin model and a changed version may be described as an asymmetric difference (aka edit script). Each operation invocation calls a parameterized graph transformation rule.	

	- A delta module application condition is a propositional formula in which the features are used as propositional variables. - Edit operations are implemented as declarative rules based on graph transformation concepts. - A delta language actually consists of a set of graph transfomratoin rules which must be specified for the respective source language. A basic set of operations can be automatically derived from the metamodell of the modeling language. Only application conditions need to be specified manually. - Difference 2 DeltaEcore: delta operation semantics specified in java (imperative), while in SiPL it is specified as graph transformation rules (declarative) - Same with DeltaEcore: Automated Delta Language Creation

3 USE CASES

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

- SiPL is a delta-based modeling framework for SPLs
- differencing for creation of delta modules instead of manual programming
- → derivation of delta module with existing visual editors instead of textual delta languages
- support evolution by a variety of analyses and restructuring functions
- → based on graph transformation rules, dependencies, conflicts etc. can be easily calculated (based on declarative nature)
- Recommended refactorings to restructure delta module sets (feature model is constant, but design problems may occur, e.g., redundancy, or derivation of particular configuration is not possible (improvement to solution space))

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
    d. family-based analyses

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:
Utilization of analyses mechanisms of other tools, e.g., FeatureIDE.

Configuration:
The validity of a configuration is analyzed externally based on the feature model, e.g., FeatureIDE.

Fragment:
Syntactical validity of fragment notation (e.g., conformity to metamodel) is ensured.

Family-based analyses

for pairs of delta-modules:

- The detection of *conflicts* between two delta-modules, i.e. both delta-modules cannot be applied together. This function returns a set of pairs of edit steps which are in conflict.
- The detection of *dependencies* between two delta-modules, i.e. both delta-modules can only be applied in a certain order. This function returns a set of pairs of edit steps which depend on one another.
- The detection of *duplicate edit steps* in both delta-modules. This function returns a set of pairs of edit steps which appear identically in both delta-modules.
- The detection of *transient effects*. This function returns a set of pairs of edit steps where one of them removes the effect of the other.
- Refactoring operations based on restructuring operations (automatic conflict resolution strategies)
- Some conflicts can only be resolved by extracting the conflicting delta actions from one of both delta modules into a new delta module which is equipped with an application condition that prevents the application of the conflicting delta actions.
- Family based analyses (optimization & improvement of solution space prior to product derivation but with regard to all configurations)

Evolution support:

- Metrics for quality assurance
- Refactoring recommendation (relations between delta actions can be aggregated to relations between delta modules and can be validated against the feature model -> hint for mismatch between problem and solution space, e.g, two features are compatible in the solution space but their delta modules are in conflict, hence the variant cannot be derived as expected.)
- Versioning via version control system

Mapping mechanism

All application conditions together represent the mapping of the problem space onto the solution space. This mapping does not need to be a 1 : 1 relation, i.e. a delta-module can contribute to the implementation of several features, and a feature can be implemented by several delta-modules.

Variability realization mechanism:

As variability realization mechanism, delta modeling as transformational mechanism is applied + a delta language creation infrastructure for different source languages.

A variant is derived by selecting features or creating a configuration file. For all delta modules whose application condition is evaluated to true, an application order is determined acording to dependency relations (the dependencies between edit scripts of the selected delta-modules lead to a partial order in which the edit scripts can be applied). Next, the execution of delta operations of a delta module is performed.

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.