# Modeling Product-Line Legacy Assets using Multi-Level Theory

Damir Nešić
Royal Institute of Technology
Brinellvägen 85
Stockholm, Sweden 100-44
damirn@kth.se

Mattias Nyberg
Royal Institute of Technology
Brinellvägen 85
Stockholm, Sweden 100-44
matny@kth.se

Barbara Gallina
Mälardalen University
Högskoleplan 1
Västerås, Sweden 721 23
barbara.gallina@mdh.se

## ABSTRACT

The use of non-systematic reuse techniques in Systems Engineering (SE) leads to the creation of legacy products comprised of legacy assets like software, hardware, and mechanical parts coupled with associated traceability links to requirements, testing artifacts, architectural fragments etc. The sheer number of different legacy assets and different technologies used to engineer such legacy products makes reverse engineering of PLs in this context a daunting task. One of the prerequisites for reverse engineering of PLs is to create a *family model* that captures implementation aspects of all the legacy products. In this paper, we evaluate the applicability of a modeling paradigm called *Multi-Level Modeling*, which is based on the *class-instance* relation, for the creation of a family model that captures all the implementation concerns in an SE PL. More specifically, we evaluate an approach called *Multi-Level conceptual Theory* (MLT) for capturing different legacy assets, their mutual relations and related variability information. Moreover, we map PL concepts like variants, presence conditions and product configurations to MLT concepts and provide formal interpretation of their semantics in the MLT framework. The illustrative example used throughout the paper comes from a real case from the automotive domain.

## CCS CONCEPTS

• **Theory of computation → Data modeling**; • **Software and its engineering → Software product lines**;

## KEYWORDS

Multi-Level Modeling, Legacy systems, Reverse engineering

**ACM Reference format:**
Damir Nešić, Mattias Nyberg, and Barbara Gallina. 2017. Modeling Product-Line Legacy Assets using Multi-Level Theory. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 8 pages.
DOI: 10.1145/3109729.3109738

## 1 INTRODUCTION

*Product Line Engineering* (PLE) has been praised as the development approach ensuring systematic reuse of engineered assets, increase in product quality, and reduction of development costs and time to market [37]. However, proactively adopting PLE in the area of *Systems Engineering* (SE) is challenging because of organizational issues, high investments in *legacy products*, and ultimately because of product complexity. Each legacy product in an SE Product Line (PL) is a composition of several types of different assets, e.g. software, hardware, and mechanical parts, with the accompanying documentation, e.g. requirements, architecture models, test artifacts, and corresponding traceability links. Consequently, *reverse engineering* [14] a model that captures all of the above information is a challenging task [5, 28].

Reverse engineering PLs has been the goal of many contributions in recent years, see surveys in [1, 38]. As noted in [1], *"reverse engineering is concerned with understanding the systems"*, i.e. obtaining a different or a more abstract representation of the system compared to the existing one. Understanding legacy products usually entails the creation of a *family model* [37], which is the superimposition of the implementation of all product variants. Unlike pure software PLs where the family model is a code-base, in SE PLs such a model should contain different types of legacy assets and different relations among them, e.g. one part *contains* another part or an embedded system *implements* a function. Moreover, the model should contain traceability links to the accompanying documentation assets and if available, variability information. Such a model could then be used to reengineer the legacy assets but it can also fulfill numerous other use cases across the PL. Some examples of the use cases are: change impact analysis, integration testing design and configuration [36], generation of assurance cases [26], and in general better decision making and safer PL evolution.

There are two main challenges for the creation of such a family model in an SE PL. Firstly, different assets originate from different engineering domains that refer to different concepts and use different data formats, all of which makes the modeling process a manual, human-intensive task. Secondly, although there are some recommendations for the appropriate process of building such a model [5, 28] there is no standard modeling framework that supports the modeling of different concepts induced by the heterogeneity of legacy assets.

The present paper approaches the latter challenge and proposes the emerging *Multi-Level Modeling* (MLM) [2, 24] paradigm as a framework for the creation of a family model in an SE PL. The MLM paradigm emerged in order to threat the well-known deficiencies [3, 34] of *MOF-compliant frameworks* [35]. Specifically, MLM approaches allow an arbitrary number of abstraction levels and several different types of relations between adjacent abstraction levels compared to the MOF framework with fixed four levels of abstraction with only *instanceOf* relation between them. Reported benefits [15, 19, 29] of using an approach belonging to the MLM paradigm are: simpler models when capturing complex domains, greater separation of concerns, more maintainable models etc.


*SPLC '17, Sevilla, Spain*

DOI: 10.1145/3109729.3109738

More specifically, the present paper evaluates a particular MLM approach called *Multi-Level conceptual Theory* (MLT) [11] which is chosen for several reasons. Firstly, the foundation of MLT is formal logic which guarantees the creation of unambiguous models. Secondly, MLT defines several different relations between types on the adjacent levels of abstraction. These relations capture the refinement of more abstract concepts into more specific ones, similar to the relations between *domain* and *application* engineering in PLE [41]. Finally, MLT uses similar concepts as the widely-known UML class-diagram [42], which should facilitate industrial acceptance.

The evaluation of the MLT framework is driven by the question whether the concepts of the MLM paradigm are applicable for legacy asset modeling in the SE PL context, and if there is a benefit in doing so. The contributions of the present paper are: (i) embedding PL concepts into MLT framework and showing how legacy assets from a SE PL can be modeled in MLT (ii) a formal semantic interpretation of PL concepts inside the MLT framework (iii) an initial evaluation of MLT applicability for modeling of legacy assets and their mutual relations. The evaluation shows that main benefits of using the MLT framework is its ability to capture all types of legacy assets and their relations in the same model with a unified formal semantics, thus acting as a unified target modeling framework for different extraction tasks.

Applicability of MLM-based approaches for modeling PL related issues has not been explored in previous literature. However, there is a single contribution [25] discussing the similarities between PL concepts and MLM concepts. The need to model different types of legacy assets has been recognized by the PLE community [16, 33]. Some of the existing approaches aim at capturing the asset data together with corresponding variability information, e.g. Clafer language [6, 30, 32], UML-based approaches [8, 20] and other approaches [17, 18]. However, the majority of these approaches focuses on a single type of assets and corresponding variability information with the goal of generating possible configurations of the modeled assets. In contrast, we employ MLT for simultaneous modeling of different types of assets at different abstraction levels with the goal of creating a family model and enabling much broader set of use cases. Solutions presented in [17, 18] follow a similar idea like the present paper but the used modeling language is not publicly available. A more detailed discussion on the comparison between the MLT framework and other used approaches can be found in Section 5, which surveys related work.

The rest of the paper is organized as follows. Section 2 introduces the MLM modeling paradigm and the MLT framework, which is then used to model the legacy assets from an illustrative industrial example. In Section 3, PL concepts of variants, presence conditions, and product configurations are mapped to MLT concepts and configuration dependent relations receive a formal treatment. Section 4 evaluates the properties of the MLT framework and its applicability as a target framework for reverse engineering family models in SE PLs. Section 5 surveys the related work and is followed by Section 6 where we summarize the present paper and outline future work.

## 2 MULTI-LEVEL MODELING

In the recent years, a new modeling paradigm known as *Multi-Level Modeling* (MLM) has been emerging [2, 24]. MLM approaches are based on the notion of *classes* and *instances* but contrary to well-established MOF framework [35] with four abstraction levels, MLM approaches allow an arbitrary number of abstraction levels. The consequence is that an entity in a single model can simultaneously be a class and an instance which softens the line between models and meta-models. Some of the reported advantages [15, 19, 29] of using MLM approaches over the standard MOF-like approaches are: capturing complex domains with simpler models than MOF-compliant models, separation of concerns, flexible and more-maintainable models etc. In [13], the *Multi-Level conceptual Theory* (MLT) framework is introduced, which is a particular instance of the MLM paradigm. The following section presents the characteristics of MLT on an illustrating example from a real PL in the automotive domain, Scania CV AB.

### 2.1 MLT: A Theory for Multi-Level Conceptual Modeling

An example MLT model is shown in Figure 1. Because MLT does not have a defined graphical syntax, an adaptation of the graphical representation from [12] is used and a proposal for an MLT-UML profile can be found in [13].

Two central notions in MLT are the notions of *type* and *individual* and they are jointly referred to as *entities*. A type represents a set of entities and therefore it corresponds to the notion of *class* in UML class diagram [42]. Entities that cannot be instantiated are individuals. Types and individuals are related with the *instanceOf* relation, denoted as `iof`. For example, `iof(:ems2,EMS2)` states that the legacy asset `:ems2` is an instance of type `EMS2` which stands for the second generation of the *Engine Management System*. Being an MLM approach, MLT allows a type to simultaneously be a type and an instance of another type. This leads to multiple abstraction levels. For example, besides stating `iof(:ems2,EMS2)`, MLT model in Figure 1 also states `iof(EMS2,ECUgeneration)`, i.e. `EMS2` is an instance of type `ECUgeneration` which is the type of all *Electronic Control Unit* (ECU) generations. On an even higher level of abstraction, there is a type `ArchitecturalElement` whose instances are types `ECUfamily` and `ECUgeneration`. In order to reduce clutter, entities which are instances of the same type are encompassed by a dashed line rectangle and only one *instanceOf* relation is drawn, e.g. both `EMS1` and `EMS2` are instances of the type `ECUgeneration`.

In the interest of distinguishing between multiple levels of abstraction, MLT introduces *type orders* between which instantiation relation holds. There can be an arbitrary number of type orders but for the purpose of the present paper only *3rd Order Type* (3rdOT) *2nd Order Type* (2ndOT), *1st Order Type* (1stOT), and *Individual Order Type* (IndividualOT) are considered. It should be noted that the IndividualOT represents types whose instances cannot be further instantiated. Types defined on one order type, e.g. IndividualOT, are always instances of a type on the adjacent higher order type, e.g. 1stOT, except the highest order type, i.e. 3rdOT, which is an instance of itself. On a single order level, the order type is the *super-type* of all other entities on this level.

Types in MLT can have attributes, which are denoted using a ternary predicate *typeHasAttribute(A, at, B)* where *A* is the type that has the attribute, *at* is the attribute name, and *B* is the type of the attribute value, typically an *integer, string* etc. If an entity

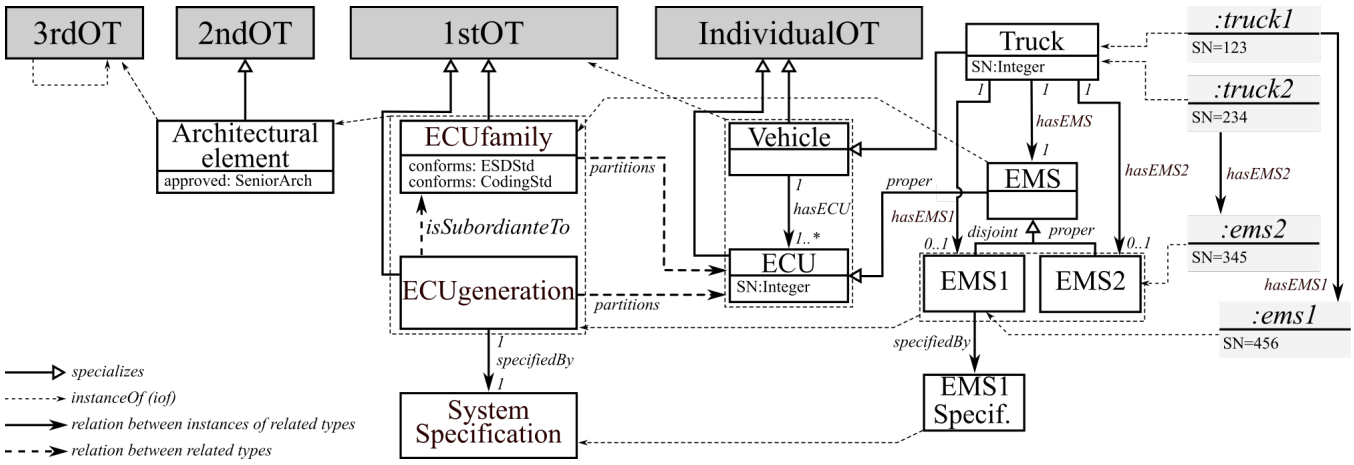


**Figure 1: MLT model of the illustrating example**

$e$ assigns a specific value $v$ to an attribute $at$, this is denoted by the predicate *hasValue(e, at, v)*. In Figure 1, the attribute `SN` is denoted by `typeHasAttribute(Truck,SN,Integer)` and the individual `:truck` assigns it a specific value expressed by the predicate formula `hasValue(:truck,SN,123)`. Attributes can have other value types besides *integers, strings* etc. For example, `ECUfamily` has an attribute `conforms` of the type `CodingStd` which means that each instance of the type `ECUfamily` can have a specific value for this attribute, for example *MISRA C*.

Relations between entities in MLT are graphically represented as a directed line between types and we say that a relation has a *source type* and an *end type*. Relations in MLT can be divided in two groups, *structural relations* and *basic relations*. Structural relations are the relations that hold between types. Definitions of structural relations in natural language and the corresponding MLT predicates can be found in Table 1 while the formalization in first-order logic can be found in [11]. Because MLT is formalized in *many-sorted first-order logic* [31], the variables $e$, $at$, and $v$ appearing in the definition of predicates in Table 1 are quantified over three disjoint sets: $\mathcal{E}$ is the set of all declared entities in an MLT model, $\mathcal{A}$ is the set of all declared attributes, and $\mathcal{V}$ is the set of all attribute values where $\mathcal{V} = \mathcal{P}(\mathcal{E})$. The original MLT work in [11] contains the fourth set $\mathcal{W}$ which is the set of all possible worlds but in the present paper, we assume that there exists only one world.

MLT defines the following structural relations:

(1) *specialization, proper specialization*, and *subordination*. These structural relations are called *intra-level* structural relations because they can exist only between types of the same order level.
(2) *instantiation, power type, categorization, complete categorization, disjoint categorization*, and *partitioning*. These structural relations are called *cross-level* structural relations because they can exist only between types on different but adjacent order levels.

*2.1.1 Intra-level structural relations.* The model in Figure 1 states that `properSpecializes(EMS,ECU)` which means that there are other instances of type `ECU` which are not instances of type `EMS`, i.e. other ECUs exist. Contrary to *proper specialization*, the *specialization* relation, e.g. `specializes(Truck,Vehicle)`, means that instances of two types can be the same. Both *specialization* relations imply attribute inheritance between types and therefore any specialization of type *ECU* has an attribute `SN` of type `Integer`.

The *subordination* relation can hold only between two types on two distinct but adjacent order levels. The meaning of relation `isSubordinateTo(ECUgeneration,ECUfamily)` is that every instance of type `ECUgeneration` is a proper specialization of an instance of type `ECUfamily`. The relevance of the *subordination* relation is that it can be used as a modeling construct that enforces certain structure of types on adjacent lower order level without explicitly stating these types.

*2.1.2 Cross-level relations.* MLT defines five different cross-level relations that are based on the pattern of *powertypes* [21]. An example of a *cross-level* relations is the *partitions* relation and the statement `partitions(ECUfamily,ECU)` means that every instance of type `ECUfamily` is a *proper specialization* of the type `ECU`. Furthermore, all instances of type `ECUfamily` are types that are pair-wise disjoint, i.e. they do not have common instances. The *partitions* relation exists also between types `ECUgeneration` and type `ECU` but because of the previously mentioned *subordination* relation between `ECUgeneration` and `ECUfamily`, each instance of type `ECUgeneration` is a *proper specialization* of an instance of the type `ECUfamily`.

*2.1.3 Basic relations.* *Basic relations* capture relations between instances of types, e.g. *hasPart* or *createdBy*, that represent containment, traceability, and other relations between different assets. As can be seen in Figure 1, basic relations can appear on any abstraction level, e.g. `hasECU` on IndividualOT level and `specifiedBy` on the 1stOT level. MLT treats all basic relations in the same way as attributes and it uses the same predicates to denote basic relations. The only difference is that the type of a basic relation value is usually not a type like *string* but instead one of the types declared in the MLT model. The example in Figure 1 shows the basic relation `hasEMS2` which is expressed by a predicate formula `typeHasAttribute(Truck,hasEMS2,EMS2)` where the arrow at

**Table 1: MLT definitions related to structural relations and attributes**

| Relation | Predicate | Semantics |
|---|---|---|
| | | **Intra-level structural relations** |
| *specialization* | `specializes(A,B)` | Type A *specializes* type B iff all instances of type A are also instances of type B. |
| *proper specialization* | `properSpecializes(A,B)` | Type A *proper specializes* type B iff A *specializes* B and A is different from B. |
| *subordination* | `isSubordinateTo(A,B)` | Type A is *subordinate* to type B iff every instance of A *proper specializes* an instance of B. |
| | | **Cross-level structural relations** |
| *powertype* | `isPowertypeOf(A,B)` | Type A is *powertype* of type B iff all instances of A are specializations of B and every possible specialization of B is an instance of A. |
| *categorization* | `categorizes(A,B)` | Type A *categorizes* type B iff all instances of A are specializations of B. |
| *complete categorization* | `completelyCategorizes(A,B)` | Type A *completely categorizes* type B iff A categorizes B and every instance of B is instance of at least one instance of A. |
| *disjoint categorization* | `disjointlyCategorizes(A,B)` | Type A *disjointly categorizes* type B iff A categorizes B and every instance of B is instance of at most one instance of A. |
| *partitioning* | `partitions(A,B)` | Type A *partitions* type B iff each instance of B is an instance of exactly one instance of A. |
| | | **Attribute related definitions** |
| *attribute definition* | `typeHasAttribute(A,`$at$`,B)` | $at$ is defined as an attribute of type A and the value of the attribute is an instance of type B |
| *attribute value assignment* | `hasValue(`$e,at,v$`)` | The predicate holds if an entity $e$ assigns a value $v$ to an attribute $at$. |
| *mandatory attribute* | `isMandatoryAttribute(`$at$`)` | An attribute $at$ is *mandatory* iff each instance of the type that has the attribute $at$ assigns a value to $at$ |
| *mono-valued attribute* | `isMonoValuedAttribute(`$at$`)` | An attribute $at$ is *mono-valued* iff each instance of the type that has the attribute $at$ assigns the same type of value to $at$ |

the end of the line connecting type `Truck` and type `EMS2` represents the relation end type. The meaning of relation `hasEMS2` is that it is a more specific relation than the `hasEMS` but MLT does not provide the concept of relation specialization. Basic relations have a dedicated graphical representation in order to visually indicate that basic relations exist between instances of declared types but in general, basic relations can be visually represented as attributes and vice versa, e.g. the type of the `conforms` attribute can be represented as an MLT type. MLT allows expressing that a basic relation, i.e. an attribute, is a *mandatory* relation. This is denoted using the predicate *isMandatoryAttribute(at)* or graphically by the relation cardinality that is at least one, e.g. [1..*], at the relation end type. The meaning of a mandatory relation is that each instance of a type that has a mandatory relation must have a specific value for the mandatory relation. A basic relation can also be *mono-valued*, which is denoted by the predicate *isMonoValuedAttribute(at)*. The meaning of a mono-valued relation is that the instances that assign values to the mono-valued attribute are always of the same type. For example, relation `hasEMS` is both mandatory and mono-valued, which is captured by `isMandatoryAttribute(hasEMS)` and `isMonoValuedAttribute(hasEMS)`. Other cardinalities of basic relations in MLT are possible, e.g. [0..*] or [2..5], but there are no predicates assigning them a specific meaning, and they are interpreted in the standard way. It should be noted that despite the graphical representation, an MLT model is just a set of predicate formulas.

## 2.2 Summary of MLT applicability

Belonging to the MLM paradigm, MLT framework offers general modeling constructs and expressiveness needed to create models capturing heterogeneous assets on different abstraction levels with corresponding relations. In a full scale model with hundreds of types, Figure 1 would have to include types like the SW and HW of an ECU that have further sub-components and that are specified by different specification documents. A particular strength of the MLT framework are the various structural relations. For example, expressing the structural relations `isSubordinateTo` and two `partitions` relations in Figure 1, implies the structuring of all instances of types `ECUfamily` and `ECUgeneration` into specialization hierarchies and in the case of our industrial partner the number of such assets exceeds three hundred.

Nonetheless, applying MLT for modeling in PL context requires establishing a mapping between the general MLT concepts and more specific PL concepts, e.g. variants, versions, product configurations etc. Additionally, in order to support the previosly outlined use cases, the semantics of the models should be unambigous and preferably formaly defined. Furthermore, depending on the assets and their relations, there will be different cardinalities of relations in the MLT model. For example, Figure 1 shows two individuals called `:truck1` and `:truck2` and each of them is in a relation with a distinct individual of EMS-ECUs, i.e. `:ems1` is an instance of type `EMS1` and `:ems2` is an instance of the type `EMS2`. It can be concluded that the cardinality of basic relations `hasEMS1` and `hasEMS2` is [0..1] and that the end types of these basic relations are optional legacy assets. In MLT terminology, basic relations `hasEMS1` and `hasEMS2` are not mandatory basic relations. Because basic relations with cardinality [0..1] imply that these basic relations exist only in some product configurations, from hereon they will be referred to as *Product-Configuration Dependent* (PCD) relations. Often in PLE, *presence conditions* [43] specify when a certain asset belongs to a product configuration, i.e. presence conditions are the information that complements PCD relations.

# 3 MAPPING PL CONCEPTS TO MLT CONCEPTS

Because MLT framework support only concepts of types, relations, and attributes, mapping PL concepts like variants, presence condition, product configuration to MLT concepts is not straightforward. Mapping concepts of presence conditions, which are propositional formulas, and product configurations, which are sets of product features, require some interpretation of the MLT framework semantics because the MLT definitions and axioms primarily capture the syntactical constraints of MLT models.

## 3.1 Mapping variants and versions

The first two concepts that we map to the MLT framework are the concepts of *variant* and *version*. Whenever a type is specialized by another type, we say that the latter is a *variant type*, hereinafter only *variant*. For example, type `EMS` is a variant of type `ECU`. As

can be seen in Figure 1, variants are organized into a specialization hierarchy and in general specialization hierarchies can be arbitrarily deep, i.e. an asset can have an arbitrary number of variants. The leaf variants represent versions of assets and are referred to as *version types*, hereinafter only *versions*. For example, types EMS1 and EMS2 are versions of the type ECU.

## 3.2 Model-Theoretic Semantics of MLT

In the original MLT framework, the set $\mathcal{V}$ which contains all value types that can be assigned to attributes is defined as the powerset of the set $\mathcal{E}$, $\mathcal{V} = \mathcal{P}(\mathcal{E})$, in order to accommodate attributes that are assigned with several different values simultaneously, i.e. composite types. For the purpose of the present paper, this is not required and from hereon all attributes are considered to be *mono-valued*.

We interpret the semantics of MLT concepts in a model-theoretic fashion based on concepts of sets and relation similar to *Description Logic* (DL) [4]. We define the semantics of types, individuals, and attributes through an *interpretation* $\mathcal{I}$ which is a tuple $\{[\![\cdot]\!]^{\mathcal{I}}, \mathcal{E}, \mathcal{A}, \mathcal{V}\}$ that consists of an *interpretation function* denoted as $[\![\cdot]\!]^{\mathcal{I}}$, and the already introduced sets $\mathcal{E}, \mathcal{A}, \mathcal{V}$. The interpretation of a type $T$, denoted as $[\![T]\!]^{\mathcal{I}}$ assigns a subset of the set of all entities in $\mathcal{E}$ to the type $T$, $[\![T]\!]^{\mathcal{I}} \subseteq \mathcal{E}$, i.e. a type corresponds to the set of its instances. The interpretation of an individual $i$, denoted as $[\![i]\!]$, assigns an element of the set of all entities to the individual $i$, $[\![i]\!]^{\mathcal{I}} \in \mathcal{E}$. The interpretation of an attribute $at$, denoted as $[\![at]\!]^{\mathcal{I}}$, assigns a binary relation to the attribute $at$, $[\![at]\!]^{\mathcal{I}} \subseteq \mathcal{E} \times \mathcal{V}$, where the relation is possibly an empty relation. When an attribute $at$ represents a basic relation it will be denoted as $R$ instead of $at$.

The specialization relation, denoted as specializes($T_1$, $T_2$), semantically corresponds to the subset relation $[\![T_1]\!]^{\mathcal{I}} \subseteq [\![T_2]\!]^{\mathcal{I}}$. The proper specialization relation, denoted as properSpecializes($T_1$, $T_2$) semantically corresponds to the proper subset relation $[\![T_1]\!]^{\mathcal{I}} \subset [\![T_2]\!]^{\mathcal{I}}$. If types $T_2$ and $T_3$ specialize type $T_1$, and if types $T_2$ and $T_3$ are declared as disjoint and complete with respect to type $T_3$, then the disjointness constraint corresponds to $[\![T_2]\!]^{\mathcal{I}} \cap [\![T_3]\!]^{\mathcal{I}} = \emptyset$, and the completeness constraint corresponds to $[\![T_1]\!]^{\mathcal{I}} = [\![T_2]\!]^{\mathcal{I}} \cup [\![T_3]\!]^{\mathcal{I}}$.

Besides model-theoretic semantics, in Section 2 the lack of a construct expressing relation specialization was identified. In order to overcome this issue, a predicate *specializesBasicRelation($R_1, R_2$)* is introduced and semantically it corresponds to $[\![R_1]\!]^{\mathcal{I}} \subseteq [\![R_2]\!]^{\mathcal{I}}$. For example, in Figure 1, basic relation specialization captures the fact that the hasEMS2 basic relation is a specialization of the hasECU basic relation, i.e specializesBasicRelation(hasEMS2,hasECU).

## 3.3 Mapping presence conditions

Issues regarding PCD relations were identified and discussed in Section 2.2. The example of the PCD relations in Figure 1, is indicated by the [0..1] cardinality of basic relations. However, in order to formally relate PCD relations with other PL concepts, we explicitly define the PCD relation.

*Definition 3.1 (PCD Relation).* : A relation $R$ from the source type $T_1$ to the end type $T_2$ is *Product-Configuration Dependent* if $[\![R]\!]^{\mathcal{I}} \subseteq [\![T_1]\!]^{\mathcal{I}} \times [\![T_2]\!]^{\mathcal{I}}$ and $\exists t_1 \in [\![T_1]\!]^{\mathcal{I}}\ \forall t_2 \in [\![T_2]\!]^{\mathcal{I}} . \Big((t_1, t_2) \notin [\![R]\!]^{\mathcal{I}}\Big)$.

The definition of the PCD relation includes empty relations, i.e. $[\![R]\!]^{\mathcal{I}} = \emptyset$, but in the remainder of the paper it is considered that no relation is empty. As mentioned in Section 2.2, in PLE a PCD relation is usually complemented with a presence condition that determines in which product configurations does this relation hold.

Based on the Definition 3.1, a PCD relation $R$ from type $T_1$ to $T_2$, can be specialized by two types $T_1'$ and $T_1''$ that are disjoint and complete. Type $T_1'$ is such that each instance of it is in a relation $R$ with an instance of the type $T_2$ and type $T_1''$ is such that there is no instance of it which is in the relation $R$ with an instance of the type $T_2$. For example, type Truck from Figure 1 can be disjointly and completely specialized by two types, Truck_with_EMS2 and Truck_without_EMS2. Then, the PCD relation hasEMS2 can be expressed as a mandatory basic relation that always exists for the set of product individuals Truck_with_EMS2, i.e typeHasAttribute(Truck_with_EMS2,hasEMS2,EMS2) and also isMandatoryAttribute(hasEMS2). Furthermore, the type called Truck_with_EMS2 can now be annotated with the presence condition of the PCD relation hasEMS2 because type Truck_with_EMS2 represents all the instances that satisfy the presence condition.

In order to avoid having presence conditions only as syntactical annotations in the model, in the next section we interpret the semantics of the presence conditions.

*3.3.1 Semantics of Presence Conditions.* In order to characterize each product individual in the MLT framework, the notion of a *configuration* must be defined. Let $S$ be a non-empty and finite set of unique *product-describing symbols* $S = \{s_1, s_2, ..., s_n\}$, hereinafter referred to as *symbols*, such that each symbol represents some characteristic of a product individual. Symbols in $S$ can represent characteristics like features, time of production etc. A *product configuration* $C$, hereinafter referred to as *configuration*, is a subset of $S$, $C \subseteq S$, and it characterizes a set of product individuals. The set of all configurations is the powerset of the set of symbols, $\mathcal{C} = \mathcal{P}(\mathcal{S})$.

Because each product individual is realized or described by a subset of all assets, we assume that each asset, represented by a type being the end type of a PCD relation, has a *presence condition*. A presence condition is a logical expression $\phi$ over the set of symbols $S$ and it is formed using the logical operators $\wedge, \vee, \neg$. A product individual characterized by a configuration $C$ is in a relation with an asset if and only if the *presence condition* $\phi$ of the asset is entailed by the symbols in configuration $C$, denoted as $C \models \phi$. For example, if the asset EMS2 has the presence condition $\phi = s1 \wedge s2$ then each instance of type Truck that is in a relation with an instance of EMS2 must be characterized by a configuration $C$ such that $s1 \in C$ and $s2 \in C$.

Following the idea from the end of Section 3.3, we proceed to characterize sets of product individuals in which a particular PCD relation always exists. Let the type $\hat{P}$ represent all possible product individuals. Any type $P_j$ representing a subset of product individuals, $[\![P_j]\!]^{\mathcal{I}} \subseteq [\![\hat{P}]\!]^{\mathcal{I}}$, is a specialization of $\hat{P}$, i.e. specializes($P_j, \hat{P}$). Because type $P_j$ represents a subset of all product individuals, it will be refer to as *product group type*, for short only *product group*.

Let a function Config : $\hat{P} \rightarrow \mathcal{C}$ be defined such that it returns a configuration $C$ for each product individual in $\hat{P}$. For each asset with a presence condition $\phi$, modeled as type $T$ that is the end type of a PCD relation $R$, a type $P_j$ can be defined as the set of all product individuals whose configurations entail the presence

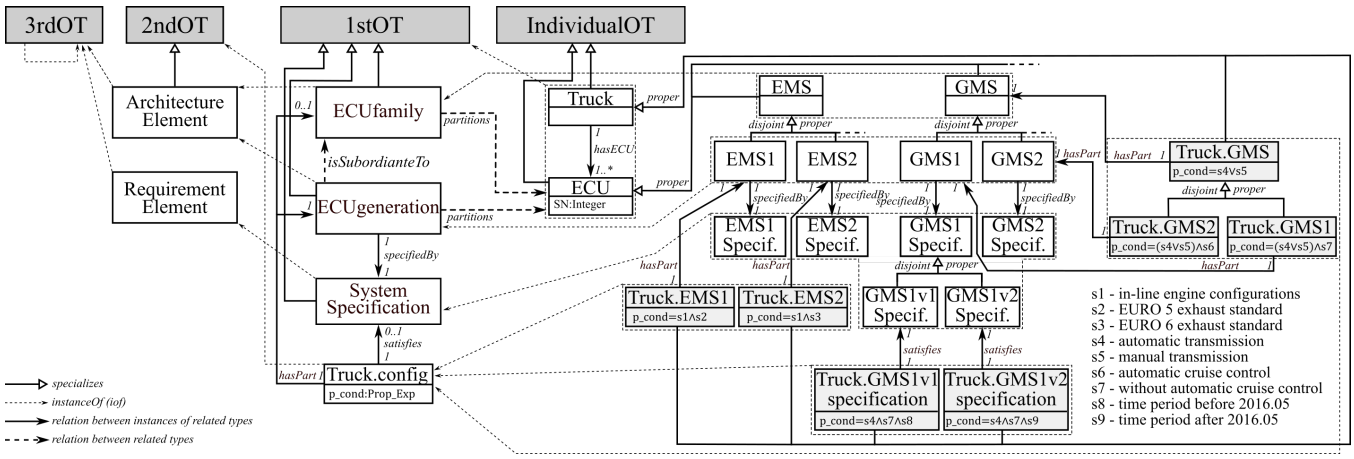


**Figure 2: Product groups and PCS relations applied to the extended illustrating example**

condition $\phi$, i.e. $[\![P_j]\!] = \{x | \text{Config}(x) \models \phi\}$. Then, let $R'$ be a relation from the type $P_j$ to type $T$, i.e. `typeHasAttribute(`$P_j$`,`$R'$`,`$T$`)`. Because the configuration of each product individual in the product group $P_j$ entails the presence condition of type $T$, it follows that each product individual in $P_j$ is in the $R'$ relation with an instance of the type $T$, i.e. relation $R'$ is a mandatory relation denoted as `isMandatoryAttribute(`$R'$`)`. Due to the fact that relation $R'$ is specific for the product individuals in $P_j$, we refer to such relations as *Product-Configuration Specific* (PCS) relation.

*Definition 3.2 (PCS Relation).* A relation $R'$, from product group $P$ to type $T$, interpreted as $[\![R']\!]^{\mathcal{I}} \subseteq [\![P]\!]^{\mathcal{I}} \times [\![T]\!]^{\mathcal{I}}$, is *Product-Configuration Specific* if it holds that $[\![R']\!]^{\mathcal{I}} = [\![R]\!]^{\mathcal{I}}$ where $R$ is a PCD relation from type $\hat{P}$ to type $T$.

It follows from Definition 3.2 that each PCD relation can be replaced by a PCS relation which is a stronger construct, i.e. it is a mandatory basic relation. Moreover each presence condition of each asset induces the creation of a product group, i.e. presence conditions are represented using concepts that are *first-class citizens* in the MLT framework and they have a semantic interpretation in contrast to being purely syntactic constructs.

Next section introduces an extension of the illustrating example and shows how products groups, presence conditions, and PCS relations can be modeled in MLT framework.

## 3.4 Expressing PL concepts in MLT

Figure 2 shows the extended illustrating example from Figure 1 with added product groups and *product-configuration specific* relations instead of *product-configuration dependent* relations.

Each presence condition is represented as an attribute of the product group that represents product individuals whose configurations entail the presence condition. Presence conditions are mandatory attributes with the value type being a singleton set whose only member is the logical expression capturing the presence condition. For example, the presence condition of the asset `EMS2` is $\phi = s1 \wedge s3$, and in MLT terms it is represented as `typeHasAttribute(Truck.EMS2,p_cond,{`$s1 \wedge s3$`})` and additionally `isMandatoryAttribute(p_cond)`. The existence of product groups on `IndividualOT` type order, implies that there is a type on the `1stOT` type order, here called `Truck.config` that represents all product configurations, i.e. each product group is an instance of the type `Truck.config`. Furthermore, the *hasPart* relation between type `Truck.config` and types `ECUfamily` and `ECUgeneration` models all PCS relations that exist between assets on the IndividualOT order level.

Each presence condition associated with an asset is interpreted as a product group that are shaded in light gray. The meaning of *product-describing symbols* that form the presence conditions of assets are shown in the bottom right of Figure 2. The extended example omits individuals in order to reduce clutter but introduces some additional assets in order to highlight a consequence of introducing product groups. The assets on the right side of Figure 2 represent another ECU, called *Gearbox Management system* (GMS), besides the previously mentioned *engine management system*.

A consequence of introducing product groups in MLT is that depending on the presence conditions entailed by the configurations of product group instances, specialization relations appear between product groups. As can be seen in Figure 2, product groups `Truck.GMS1` and `Truck.GMS2` are specializing the product group `Truck.GMS`. These particular specialization relations reflect the fact that the presence conditions of types `GMS1` and `GMS2` entail the presence condition of the ECU variant `GMS`. For a large number of assets, specialization relations might not be as obvious as in Figure 2 and because of that, Proposition 3.3 formalizes the relations between the presence condition attributes of product groups that are in a specialization relation.

Proposition 3.3. *Let $P$ and $P_1, ..., P_n$ be product groups with presence-condition attributes $\phi$ and $\phi_1, ..., \phi_n$, respectively. Then:*

*a) product group $P$ specializes product groups $P_1, ..., P_n$, i.e. it holds that $[\![P]\!]^{\mathcal{I}} \subseteq \bigcap_{j=1}^{n} [\![P_j]\!]^{\mathcal{I}}$, if and only if $\phi \models \phi_1, ..., \phi_n$.*

*b) product group $P_1, ..., P_n$ specialize product group $P$, i.e. for each $P_j$ it holds that $[\![P_j]\!]^{\mathcal{I}} \subseteq [\![P]\!]^{\mathcal{I}}$, if and only if $\phi_1 \models \phi \wedge \cdots \wedge \phi_n \models \phi$.*

Making the relations between presence conditions explicit in the MLT model allows their further mining or just visualization, which can be particularly useful in large PL in SE [10].

# 4 EVALUATION OF THE MLT FRAMEWORK

Evaluation of modeling frameworks can be conducted using different methodologies, e.g. case studies, experiments, analysis of syntax and semantics etc. In this paper, we discuss different aspects of the MLT framework followed by a discussion on potential applications of the MLT framework for the creation of a family model during the process of reverse engineering a PL in SE context.

## 4.1 Syntax and Semantics

MLT is conceived as a framework with textual syntax, i.e. predicate statements in *First-Order Logic*, but both the original paper and the examples in the present paper use a graphical representation in order to facilitate comprehension. Because MLT uses similar concepts as the well-known UML class-diagram language, graphical MLT models should be intuitive for a large audience. Creation of a MLT-UML profile, according to the proposal in [13], would facilitate the use of MLT framework while the generation of the textual representation, suitable for the analysis of MLT models, can be automated.

The semantics of the MLT framework is not fully captured in the original paper and the present paper complements it with additional definitions. Additional definitions have contributed to the mapping of all considered PL concepts, i.e. variants, presence conditions, and product configurations, onto concepts that are first-class citizens in the MLT framework. The semantics itself is simple, i.e. expressed in terms of sets and relations, and is compatible with semantics of other traditional *class-instance* based languages [6, 35] but also with *knowledge representation* languages [7].

Since both the syntactical rules and the semantics of the MLT framework are formally defined, ensuring well-formedness of MLT models can be automated, for example using Alloy analyzer [27].

## 4.2 Applicability for family model creation in SE PL

At the conceptual level, where each legacy asset is represented as a class, i.e. a type in MLT, commonly used modeling frameworks, e.g. UML class-diagram, can be used to capture the implementation details of legacy products. The advantage of using the MLT framework becomes evident if besides implementation details, other assets like specification documents or architectural components should also be captured by the same model.

The multiple abstraction levels in the model in Figure 2 can be viewed as several pairs of models and meta-models from tools that manage different legacy assets. For example, types on levels 2ndOT and 1stOT could be the meta-model and models from a tool that manages architectural assets. Types on levels 1stOT and IndividualOT could be the meta-model and models from a tool that manages implementation assets and specification documents. Finally, types on IndividualOT and real world individuals, shown in Figure 1, could be the objects from a production tool. Reverse engineering a family model that captures all the assets of legacy products in an SE PL then becomes the problem of merging existing models of legacy assets in a single MLT model on appropriate abstraction levels. In such scenario, reverse engineering a family model turns into a data integration problem that can be approached by reusing existing data integration methods [9].

A further advantage of reverse engineering an MLT family model is that all the use cases outlined in the introduction, e.g. change impact analysis, could be performed over a single model which would increase the reliability of further extraction tasks and in general to safer PL evolution.

# 5 RELATED WORK

Work in [25] also explores the relations between a MLM approaches and PL concepts. This paper introduces a meta-language that captures PL concepts of *configuration*, *parametrization*, and *template instantiation* with the purpose of allowing model-based language engineering and further disambiguating the relations between abstraction levels in MLM languages.

Clafer [6] is a modeling language designed with the goal to unify notions of feature and a class thus allowing expressing both feature-like and class-diagram-like models. Clafer is used for PL asset modeling but in some aspects, Clafer is less expressive than MLT. Clafer does not support multiple levels of specialization, e.g. if type `EMS` specializes type `ECU`, it can't be stated that type `EMS1` specializes type `EMS`. Meta-modeling in Clafer is to a certain extent supported through refinement of abstract classes that cannot have instances while in MLT meta-modeling is achieved by defining types and relation on higher abstraction levels where each type in MLT model can have instances. Instances in Clafer are semantically still classes with a single individual while MLT can represent actual real world individuals.

CVL [23] and BVR [22] languages are intended for derivation of MOF-compliant [35] models by connecting the PL assets models and the variability models. The variability of PL assets is captured in a feature-like variability model called *VSpec* that is then resolved through the process of *materialization* and reflected in the PL assets model. The approach presented in the present paper is conceptually different because the use of the MLT framework keeps PL asset data and PL asset relations in the same model with variability information, i.e. presence conditions and product groups.

ADOM [40] is a domain and application engineering modeling framework that is implemented using *UML stereotypes* and *multiplicities* in order to capture the variability in the domain and check the conformance of product variants to the domain model. However, ADOM does not explicitly model presence conditions and product configurations. Work in [39] introduces an approach that models phisycal product structure and multiplicities of product parts together with corresponding presence conditions. The relations between the parts are limited to containment. Work in [17] introduces an approach for integration of different PL assets models and their configuration into specific product models while conforming to the dependencies from the original domain models. However, finding a framework in which the data about PL assets can be integrated systematically is left as future work.

# 6 CONCLUSIONS

Because of the scale of SE PLs and the heterogeneity of legacy assets, reverse engineering of SE PLs is a challenging task. One of the challenges is the absence of a common modeling framework in which the *family model* including all legacy assets can be captured

together with their relations, traceability links, and variability information. In the present paper we have explored the applicability of a *Multi-Level Modeling* (MLM) approach called *Multi-Level conceptual Theory* (MLT) framework for capturing such a *family model*. First, the default MLT framework was used to model different types of legacy assets on different abstraction levels. After that, PL specific concepts of variants, versions, presence conditions, and product configurations were mapped to MLT concepts. Furthermore, formal semantic interpretation of relations that are specific for specific product configuration and their corresponding presence conditions has been provided. Finally, an initial evaluation and potential application of the MLT framework for reverse engineering PLs was discussed.

As can be seen in the provided examples, the consequence of being an instance of the MLM paradigm allowed modeling of concepts on different abstraction levels, from real world individuals to abstract entities like `ArchitectureElement`. This outcome was expected due to the expressiveness of the MLT framework. Mapping PL concepts of variants and versions was straightforward but mapping presence conditions and product configurations required careful interpretation of MLT semantics. The main benefit of using the MLT framework is that all legacy assets and their relations can be expressed in the same model with a unified formal semantics, thus acting as a unified target modeling framework for different extraction tasks. Moreover, having all legacy assets with associated relations and variability information expressed in a single model facilitates the fulfillment of the use cases outlined in the introduction.

Although the findings of this paper are promising, future work will include more extensive evaluation of the MLT framework on large industrial examples. Additionally, one of the main goals of future work is to automate reverse engineering of MLT models by leveraging Linked Data technologies [44] in order to create a generic data extraction framework.

## 7 ACKNOWLEDGMENTS

This work was funded by the ITEA 14014 ASSUME project with the support from Scania CV AB. B. Gallina is partially financially supported by the Swedish Foundation for Strategic Research via the SSF SM140013 Gen&ReuseSafetyCases project.

## REFERENCES

[1] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* (2017).
[2] C. Atkinson, M. Gutheil, and B. Kennel. 2009. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering* (2009).
[3] C. Atkinson and T. Kühne. 2001. The Essence of Multilevel Metamodeling. In *UML '01*.
[4] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider (Eds.). 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.
[5] E. Bagheri, F. Ensan, and D. Gasevic. 2012. Decision support for the software product line domain engineering lifecycle. *Automated Software Engineering* (2012).
[6] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski. 2016. Clafer: unifying class and feature modeling. *Software & Systems Modeling* (2016).
[7] S. Bechhofer. 2009. *OWL: Web Ontology Language*. Springer.
[8] R. Behjati, T. Yue, L. Briand, and B. Selic. 2013. SimPL: A product-line modeling methodology for families of integrated control systems. *Information and Software Technology* (2013).
[9] Z. Bellahsene, A. Bonifati, and E. Rahm (Eds.). 2011. *Schema matching and mapping*. Springer.
[10] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS '14*.
[11] V. A. Carvalho and J. P. A. Almeida. 2016. Toward a well-founded theory for multi-level conceptual modeling. *Software & Systems Modeling* (2016).
[12] V. A. Carvalho, J. P. A. Almeida, C. M. Fonseca, and G. Guizzardi. 2015. Extending the Foundations of Ontology-Based Conceptual Modeling with a Multi-level Theory. In *ER '16*. Springer.
[13] V. A. Carvalho, J. P. A. Almeida, and G. Guizzardi. 2016. Using a Well-Founded Multi-level Theory to Support the Analysis and Representation of the Powertype Pattern in Conceptual Modeling. In *CAISE '16*.
[14] Elliot J. Chikofsky and James H. Cross II. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* (1990).
[15] T. Clark, C. Gonzalez-Perez, and B. Henderson-Sellers. 2014. A foundation for multi-level modelling. In *MODELS '14: Workshop on Multi-Level Modelling*.
[16] P. C. Clements. 2015. Product Line Engineering Comes to the Industrial Mainstream. *INCOSE International Symposium* (2015).
[17] D. Dhungana, A. Falkner, and A. Haselböck. 2013. Generation of Conjoint Domain Models for System-of-systems. *SIGPLAN Notices* (2013).
[18] D. Dhungana, P.l GrÃijnbacher, R. Rabiser, and T. Neumayer. 2010. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software* (2010).
[19] U. Frank. 2014. Multilevel Modeling: Toward a New Paradigm of Conceptual Modeling and Information Systems Design. *Business & Information Systems Engineering* (2014).
[20] H. Gomaa and M. E. Shin. 2004. A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. In *ICSR '04*.
[21] C. Gonzalez-Perez and B. Henderson-Sellers. 2006. A powertype-based metamodelling framework. *Software & Systems Modeling* (2006).
[22] Ø. Haugen and O. Øgård. 2014. BVR – Better Variability Results. In *SAM '14*.
[23] Ø. Haugen, A. Wąsowski, and K. Czarnecki. 2013. CVL: Common Variability Language. In *SPLC '13*.
[24] B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez. 2013. On the Search for a Level-Agnostic Modelling Language. In *CAISE '13*.
[25] Reinhartz-Berger I., A. Sturm, and Clark T. 2015. Exploring Multi-Level Modeling Relations Using Variability Mechanisms. In *MODELS '15: Workshop on Multi-Level Modelling*.
[26] ISO/IEC 15026 2011. *ISO/IEC 15026-2:2011 - Systems and Software assurance: Assurance cases*. standard. International Organization for Standardization, Geneva.
[27] D. Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering Methodology* (2002).
[28] J. Knodel, I. John, D. Ganesan, M. Pinzger, F. Usero, J. L. Arciniegas, and C. Riva. 2005. Asset recovery and their incorporation into product lines. In *WCRE '05)*.
[29] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and How to Use Multilevel Modelling. *ACM Transactions on Software Engineering Methodology* (2014).
[30] Antkiewicz M., Czarnecki K., and Diskin Z. 2013. Example-Driven Modeling Using Clafer. In *MDEBE '13*.
[31] K. Meinke and J. V. Tucker (Eds.). 1993. *Many-sorted Logic and Its Applications*. John Wiley & Sons.
[32] Murashkin, A. 2014. Automotive Electronic/Electric Architecture Modeling, Design Exploration and Optimization using Clafer. (2014).
[33] D. Nesic and M. Nyberg. 2016. Multi-view modeling and automated analysis of product line variability in systems engineering. In *SPLC '16*.
[34] B. Neumayr, M. A. Jeusfeld, M. Schrefl, and C. Schütz. 2014. Dual Deep Instantiation and Its ConceptBase Implementation. In *CAiSE '14*.
[35] OMG. 2017. Meta-Object Facility. (2017). http://www.omg.org/mof/
[36] Akira K Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression testing in an industrial environment. *Commun. ACM* (1998).
[37] K. Pohl, G. Böckle, and F. J. van der Linden. 2005. *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer.
[38] Arshad R. and Lau K. 2016. A Concise Classification of Reverse Engineering Approaches for Software Product Lines. In *ICSEA '16*.
[39] R. Rabiser, M. Vierhauser, P. Grünbacher, D. Dhungana, H. Schreiner, and M. Lehofer. 2014. Supporting Multiplicity and Hierarchy in Model-Based Configuration: Experiences and Lessons Learned. In *MODELS '14*.
[40] I. Reinhartz-Berger and A. Sturm. 2009. Utilizing Domain Models for Application Design and Validation. *Information and Software Technology* (2009).
[41] I. Reinhartz-Berger, A. Zamansky, and Y. Wand. 2016. An Ontological Approach for Identifying Software Variants: Specialization and Template Instantiation. In *ER '16*.
[42] J. Rumbaugh, I. Jacobson, and G. Booch. 2004. *Unified Modeling Language Reference Manual, The 2nd Edition*. Pearson.
[43] A. v. Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *ICSE '15*.
[44] World Wide Web Consortium. 2017. Linked Data. (2017). http://linkeddata.org/