Formalizing Agile Software Product Lines with a RE Metamodel

Hassan Haidar[1], Manuel Kolp[1] and Yves Wautelet[2]
[1]*LouRIM-CEMIS, Université Catholique de Louvain, Belgium*
[2]*KULeuven, Faculty of Economics and Business, Belgium*

Keywords: Agile Product Line Engineering, Requirements Engineering, Goal Model, Feature, Feature Model, AgiFPL.

Abstract: Requirements engineering (RE) techniques can play a determinant role when making the strategic decision to adopt an Agile Product Line approach to the production of software-intensive systems. This paper proposes an integrated goal and feature-based metamodel for agile software product lines. The aim is to allow analysts and developers to produce specifications that precisely capture the stakeholder's needs and intentions as well as to manage product line variabilities. Adopting practices from requirements engineering, especially goal and feature models, helps designing the domain and application engineering tiers of an agile product line. Such an approach allows a holistic perspective integrating human, organizational and agile aspects to better understand product lines dynamic business environments. It helps bridging the gap between product lines structures and requirements models, and proposes an integrated framework to all actors involved in the product line architecture.

1 INTRODUCTION

"Agile Product Line Engineering" is considered as a pioneer approach that deals with the growing complexity of information systems and the handling of competitive and changing needs of the IT production industry (da Silva et al., 2011). This approach offers better support for reusable and evolving software artefacts and helps managing changes in requirements, promoting product quality, decreasing development costs and reducing time to market.

Requirements engineering (RE) – more precisely in this research GORE (Goal-Oriented Requirements Engineering) and Feature Modeling − including elicitation, analysis, specification, verification, and management (Pohl et al., 2010), plays a determinant role in making the strategic decision to adopt a Software Agile Product Line.

Considering this role of requirements engineering, we formulate the following research question: *Which requirements engineering techniques allow analysts and developers of an agile product line to represent efficiently, stakeholders' intentions and goals on the one hand and product line variabilities and communalities on the other hand?*

Intentions, goals and variability play an important role in the development life tiers of product lines i.e., domain and application engineering. In domain engineering, intentions and goals guide the variability development of the product line, while they are used for the configuration of products in the application engineering.

This paper focuses on defining a Goal and Feature-based Metamodel for engineering agile product lines. We apply it on a concrete example taken from the literature for illustration purpose. The metamodel is defined mainly for feature-oriented agile product lines such as our own methodology called AgiFPL (Haidar et al., 2017a) considered in the context of this research. Usually these agile approaches involve two classical tiers of product line engineering: Domain Engineering and Application Engineering.

The domain engineering deals with all the aspects of managing reusable assets (artifacts), while the application engineering aims at developing a specific product for a particular stakeholder. Therefore, requirements engineering approaches have to cope with the different organizational levels and architectural complexity. Specifically, for product lines, requirements engineering, captures

Haidar, H., Kolp, M. and Wautelet, Y.
Formalizing Agile Software Product Lines with a RE Metamodel.
DOI: 10.5220/0006849000900101
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 90-101
ISBN: 978-989-758-320-9
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

both commonality and variability among product line members (Borba and Silva, 2009).

Our proposed metamodel follows a holistic approach that allows the modeling of the organizational and operational context of a product line within flexible and rapid environment. It offers thus a better understanding of the representation of product lines requirements and their stakeholders' requirements. In addition, our model takes inspiration from research in GORE frameworks such as iStar 2.0 (Dalpiaz et al., 2016), and from feature-oriented modeling (Acher et al., 2012), related to agile requirements practices like user stories (Leffingwell, 2011; Wautelet et al., 2014).

The remainder of this paper is organized as follows. Section 2 describes the main concepts of our metamodel and details the main representative elements using the Z specification. Section 3 highlights an example of application of our proposal. Section 4 presents briefly the integration of our proposed metamodel to our AgiFPL agile methodology. Section 5 concludes the paper.

2 A METAMODEL FOR AGILE PRODUCT LINES

Our motivation is to understand and build an efficient structure of the requirements engineering of a feature-oriented product line in an agile context. This leads us to define a goal and feature-oriented specification to provide modeling constructs that permit:

- Representations of stakeholder's intentions and goals;
- Variability, commonality and technical elements of the agile product line;
- Requirements artifacts and their relationships used by agile teams.

The proposed metamodel defines two main perspectives. The first one is the product line engineering perspective itself, in which goal (i.e. Family goal model) and feature models provide different variability perspectives and the rationale of the variability. The second one is the agile development perspective, in which the agile requirements artifacts and goal models provide an exhaustive structure for the implementation of product line's features and products derivation.

Standard goal models languages like i* (Yu et al., 2011) can represent intentional variability, but lack mechanisms for representing differences between intentional spaces of various systems (i.e., product line variability in the intentional space). Therefore, Asadi et al. (2016) have introduced the notion of *family goal model* to extend standard goal modeling techniques, which we apply in this paper to *iStar 2.0* (Dalpiaz et al., 2016).

Our metamodel connects family goal models and features models through mappings. They provide bidirectional relationships and traceability links between high-level stakeholders' business objectives, which are described by goal models and implementation units enclosed within features in feature models. In addition, we seek to support the stakeholders of a product line, especially in the application engineering tier, through iStar 2.0 models, which provide a graphical and comprehensive vision of the stories and their relationships. In fact, in our proposed model, Backlog items (i.e., user stories,…), and family goal models are connected by mappings performed through heuristics rules proposed in (Jaqueira et al., 2013) and (Apel et al., 2010).

Figure 1 introduces the main entities and relationships of our metamodel. We subdivide it into four sub-models:

- The *Organizational sub-model*, describing the members (i.e. actors, teams, …) of the product line, their organizational roles, responsibilities, capabilities and relationships;
- The *Goal-oriented sub-model*, describing the intentions of the product line stakeholders and generating a stakeholder's view of feature models;
- The *Feature-oriented sub-model*, illustrating the product line variability;
- The *Agile requirements artifacts sub-model* defining the requirements artifacts used by agile teams, as well as the relationships among these artifacts.

The primitives of our framework are also of different types. We classify them as:

- Meta-concepts: Goal, Feature, Actor, User Story …
- Meta-relationships: Qualifies, Refines, Composition, Aggregation, Generalization …
- Meta-attributes: Power, Motivation ...
- Meta-constraints: implications between features located in different parts of the feature hierarchy.

All meta-concepts, meta-relationships and meta-constraints have the following mandatory meta-attributes: *Name*, and *Description*.

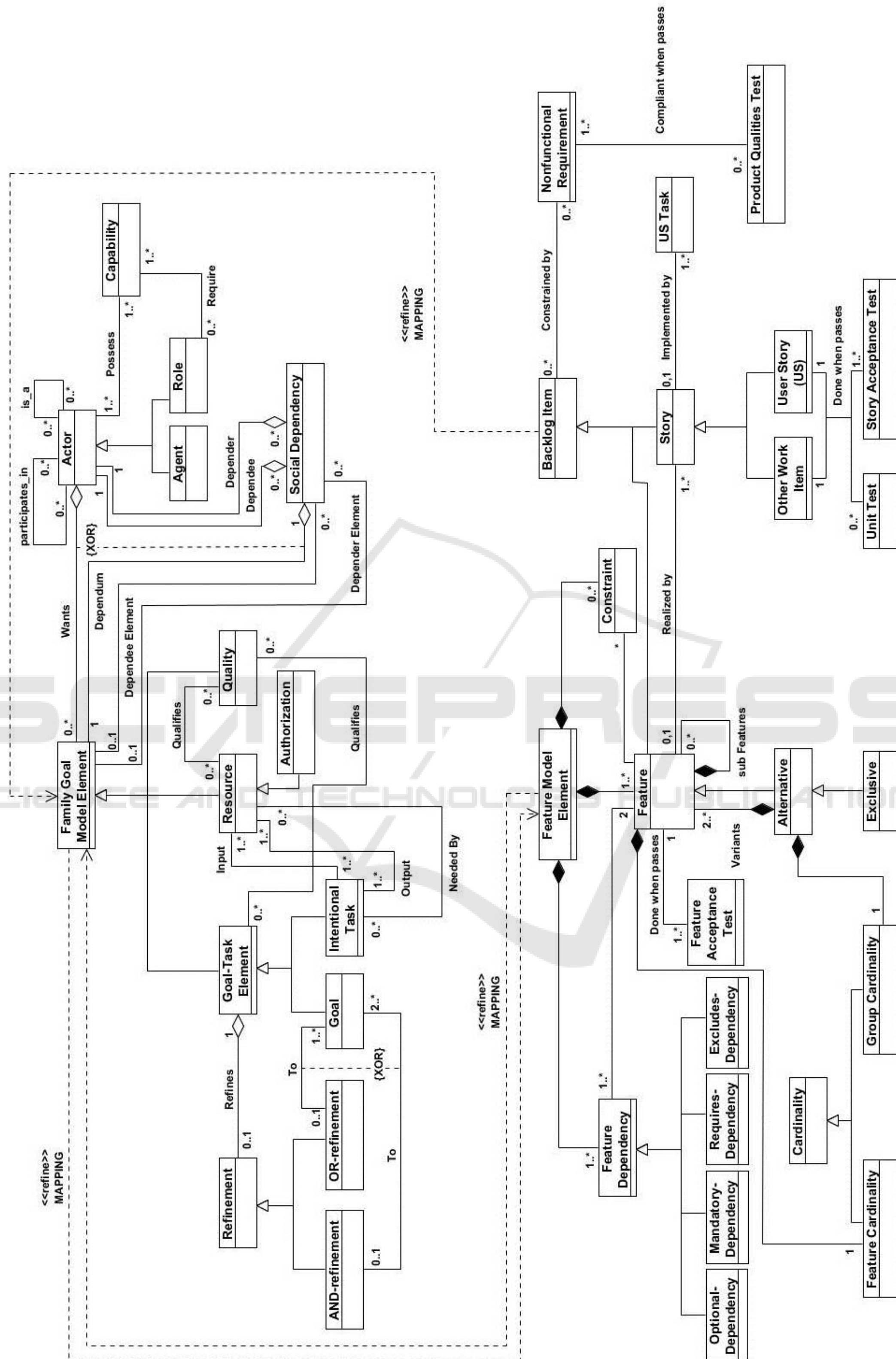

Figure 1: Requirements-oriented Meta-model for Agile Product Lines.

Name allows unambiguous reference to the instance of the meta-concept; and *Description* provides a precise and unambiguous description of the corresponding instance of the meta-concept. The description should contain sufficient information for a formal specification to be derived for use in requirements specifications for a future product or application of the product line.

Figure 1 insists on meta-concepts and meta-relationships. Meta-attributes and meta-constraints are formalized with the Z state-based specification language (O'Regan, 2013). We use Z since it provides sufficient modularity, abstraction and expressiveness to describe the requirements engineering aspects of agile product line and the wider context in which they are used in a consistent and structured way. In addition, Z offers a pragmatic approach to specifications by allowing a clear transition between specification and implementation of product line's applications. Moreover, it is widely accepted in the software development industry and Academia.

Due to lack of space, this paper only details the organizational, goal-oriented, feature-oriented sub-models and their integration, and user story concept. It also discusses their relevance for agile product lines requirements engineering.

2.1 Organizational Sub-model

This sub-model identifies the relevant ***Actors*** of the product line, the ***Roles*** they occupy, and the ***Dependum*** for which Actors depend on one another.

2.1.1 Actor

Most of stakeholders are represented as actors. Actors can be human, organizations, technical systems (i.e. hardware, software), or any combination thereof. Actors are active, autonomous entities that aim at achieving their goals by exercising their know-how, in collaboration with other actors. According to the iStar 2.0 language, two types of actors can be distinguished (Dalpiaz et al., 2016):

- **Role:** *an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor.*
- **Agent:** *an actor with concrete, physical manifestations, such as a human individual, an organization, or a department.*

Actor's intentionality is made explicit through the *actor boundary*, which is a graphical container for their intentional elements.

```
[Name]
[Informal_Defintion]
[Actor_Type] := Role | Agent
[Goal]

___Actor________________________________
| name : Name
| description : Informal_Definition
| is_a : Actor_Type
| want : set Goal
| own : set Resource
| possess : set Capability
|_______________
| (want ≠ ø) ∧ (possess ≠ ø)                          (c1)
| (∀ act : Actor) act.is_a = Agent ⇒ act.own ≠ ø       (c2)
|________________________________________
```

The ***Actor*** schema above shows the Z formal specification of the Actor concept. The first part of the specification represents the definition of types. The Actor specification first defines the type Name (which represents the Name attribute) by writing [Name]. This declaration introduces the set of all names, without making assumptions about the type (i.e. whether the name is a string of characters and numbers, or only characters,…). The type [Actor_Type] is defined as being either a *Role* or an *Agent* or even just an *Actor*.

More complex and structured types are defined with specific schemata. For instance, the ***Actor*** schema is partitioned horizontally into two sections:

- The *declaration section* introduces a set of named, typed variable declarations.
- The *predicate section* provides predicates that constrain values of the variables. We use identifiers e.g. "(c1)" to refer to predicate, i.e. constraint (c1) of the schema.

In essence an *Actor* of an agile product line *wants* to fulfil the product line *Goals* as well his/her own *Goals*. In fact, an Actor possesses his/her specific *Capabilities* and owns a set of *Resources*. Each Actor applies plans that are part of his/her *Capabilities* and uses *Resources* in order to achieve the *Goal* that he/she *wants*. As the *Actor* is present in a rapid and flexible environment, he/she to take into account the changing *Intentional Elements* related to the product line as well as the ones related to specific customer's needs, in order to adapt its behavior to environmental circumstances. Considering these changes is crucial when eliciting

product line requirements as well as Stakeholders (i.e. product owner, etc.) requirements.

Since an *Actor* can be also a *Role* or an *Agent*, two different types of actor links exist:

- *is-a*: represents the concept of generalization or specification. Only *Roles* can be specialized into *Roles*, or general *Actors* into general *Actors*. However, *Agents* cannot be specialized via is-a, as they are concrete instantiations.
- *participates-in*: represents any kind of association, other than generalization or specialization, between two *Actors*. No restriction exists on the type of actors linked by this association. Note that every *Actor* can *participates-in* multiple other *Actors*.

Thus, a *is-a* relationship applies only between pairs of *Roles* or pairs of *Actors*. There should be no *is-a* cycles. In addition, there should be no *participate-in* cycles. A pair of *Actors* can be linked by at most one actor link. It is not possible to connect two actors via both *is-a* and *participates-in*. An *Actor* can (sometimes, has to) cooperate with another *Actor* to fulfil common *Goals* to the *Roles* that each of these *Actors* occupies.

2.1.2 Role

As stated above, an organizational *Role* of the product line is an abstract characterization of expected behavior of an *Actor* within some specified context of the product line. An *Actor* can occupy multiple *Roles* and multiple *Actors* can occupy a *Role*.

The following ***Role*** schema shows the Z formal specification of Role concept within a product line. Each Role requires a set of Capabilities to fulfil or contribute to Goals for which it is responsible. An Actor can occupy the Role only if it possesses the required Capabilities **(c4)**. Moreover, to entering Roles, Actors should be able to leave roles at runtime **(c5)**.

Roles are responsible for *Goals* **(c6)** and can control their fulfilment. This control procedure requires that a single *Actor* can never occupy distinct *Roles* that are responsible of and control the fulfilment of the *Goal* **(c7)**. In addition, *Roles* can have different levels of authority. Consequently, a Role can have authority on other Roles. The authority on relationship specifies the hierarchical structure of the product line.

[Goal_control_Status]

Role

name : *Name*

description : *Informal_Definition*

require : *set Capability*

responsible : set *Goal*

control : *set* (*Goal*, *Goal_control_status*)

authoroty_on : *set Role*

$(\text{require} \neq \emptyset) \wedge (\text{responsible} \neq \emptyset)$ **(c3)**

$(\forall\ act : Actor\ ;\ r : Role)$ **(c4)**

$r \in act.occupy \Rightarrow r.require \subset act.possess$

$(\forall\ act : Actor\ ;\ r : Role)$ **(c5)**

$act.leave \Rightarrow r \notin act.occupy$

$(\forall\ r : Role\ ;\ g : Goal)$ **(c6)**

$g \in r.responsible \Rightarrow g.sec_is_a = Goal$

$(\forall\ r_1 , r_2 : Role\ ;\ g : Goal\ ;\ a_1 , a_2 : Actor)$ **(c7)**

$(g.sec_is_a = Goal \wedge g \in r_1.responsible \wedge$

$g \in r_2.control \wedge r_1 \neq r_2 \wedge r_1 \in act.occupy \wedge$

$r_2 \in act.occupy) \Rightarrow a_1 \neq a_2$

2.1.3 Dependum

In social models such as iStar 2.0, dependencies represent social relationships. A dependency is defined as a relationship with five arguments:

- *Depender* is the actor that depends for something (the dependum) to be provided;
- *DependerElmt* is the intentional element within the depender's actor boundary where the dependency starts from, which explains why the dependency exists;
- *Dependum* is an intentional element that is the object of the dependency;
- *Dependee* is the actor that should provide the dependum;
- *DependeeElmt* is the intentional element that explains how the dependee intends to provide the dependum.

Dependencies link the *dependerElmt* within the *depender* actor to the dependum, outside actor boundaries, to the *dependeeElmt* within the *dependee* actor.

The type of the dependum specializes the semantics of the relationship:

- *Goal*: the dependee is expected to achieve the goal, and is free to choose how;

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

- *Quality:* the dependee is expected to sufficiently satisfy the quality, and is free to choose how;
- *Task:* the dependee is expected to execute the task in a prescribed way;
- *Resource:* the dependee is expected to make the resource available to the depender.

[Dependum_Type] := Goal | Quality | Task | Resource

Dependum

name : *Name*

description : *Informal_Definition*

type : *Dependum_Type*

depender : set *Role*

dependee : set *Role*

(type ≠ ø) ∧ (depender ≠ ø) ∧ (dependee ≠ ø) **(c8)**

(∀ d : *Dependency* ; dpd : *Dependum* ; r_1, r_2 : *Role*) **(c9)**

$r_1 \neq r_2 \wedge (d \equiv r_1 \times dpd \times r_2) \Rightarrow (depender = r_2 \wedge dependee = r_1)$

(∀ d : *Dependency* ; dpd : *Dependum* ; r_1, r_2 : *Role*) **(c10)**

$r_1 \neq r_2 \wedge (d \equiv r_1 \times dpd \times r_2) \wedge$ (dpd.type = authorization)

⇒ $r_1 \in r_2$.authoroty_on

(∀ res : *Resource* ; a_1, a_2 : *Actor* ; cap_1, cap_2 : *Capability* ;

t_1, t_2 : *Task* ; r_1, r_2 : *Role*) **(c11)**

($a_1 \neq a_2 \wedge cap_1 \neq cap_2 \wedge t_1 \neq t_2 \wedge$ ($t_1 \in cap_1$.composed_of ∧

$cap_1 \in a_1$.possess) ∧ ($t_2 \in cap_2$.composed_of ∧

$cap_2 \in a_2$.possess) ∧ res ∈ t_1.postcondition ∧

res ∈ t_2.input ∧ $r_1 \in a_1$.occupy

∧ $r_2 \in a_2$.occupy ∧ { r_1, r_2} ∉ { a_1.occupy ∩ a_2.occupy})

⇔ (∃ dm: Dependum ∧ dm.type = Resource ∧

dm.name = res.name ∧ dm.depender = r_2 ∧

dm.dependee = r_1)

The ***Dependum*** schema above shows the formal specification of the *Dependum*. *Resource* dependency allows us to represent any specialization of the *Resource* concept as a *Dependum*. For example, a *Role* (r_1) might depend on another *Role* (r_2) for an *Authorization*. This has implication on the authority on relationship, as this dependency means that r_2 must have authority on r_1 (i.e. **c11**). In addition, the constraint **(c11)** demonstrates that the existence of a *Resource Dependum* among *Roles* has implications on the *Input* and *Postcondition* of *Tasks* accomplished by *Actors* that occupy these *Roles*.

2.2 Goal Sub-Model

Intentional elements are the actors' needs. As such, they model different kinds of requirements and are central to our proposal. The following elements are considered as *Intentional Elements (Family Goal Model Elements)* in this work:

- *Goal:* a state of affairs that the actor wants to achieve and that has clearly cut criteria of achievement;
- *Quality:* an attribute for which an actor desires some level of achievement;
- *Task:* an action that an actor wants to be executed, usually with the purpose of achieving some goal;
- *Resource:* a physical or informational entity that the actor requires in order to perform a task.

[Family_Goal_Element] := Goal | Quality | Task | Resource
[Goal_Type] := Requirement | Expectation
[Goal_Pattern] := Achieve | Cease | Maintain | Avoid
[Status] := Fulfilled | Unfulfilled
[Refinement_Alternative]

Family Goal Model

name : *Name*

description : *Informal_Definition*

intentional_elmt_is_a : *Family_Goal_Element*

goal_is_a : *Goal_Type*

pattern : *Goal_Pattern*

status : *Status*

refined_by : set *Refinement_Alternative*

(∀ g: *Goal* ; t: *Task*) g.intentional_elmt_is_a = Goal ∧

t.intentional_elmt_is_a = *Task* ⇒ (g.status ≠ ∅)

∧ (t.status ≠ ∅) **(c12)**

(∀ g: *Goal*) g.intentional_elmt_is_a = *Goal*

∧ ∃ tset = {t_1 , ... , t_2} ⇒ g.status = Fulfilled **(c13)**

(∀ g : *Goal* ; r : *Role* ; act : *Actor*) **(c14)**

(g.intentional_elmt_is_a = *Goal* ∧ r ∈ act.occupy

∧ g ∈ r.responsible ∧ act.isa = *Agent*) ⇒

g.goal_is_a = *Requirement*

(∀ g : *Goal* ; r : *Role* ; ac t: *Actor*) **(c15)**

(g.intentional_elmt_is_a = Goal ∧ r ∈ act.occupy

∧ g ∈ r.responsible ∧ act.isa = Role) ⇒

g.goal_is_a = Expectation

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

The ***Family Goal Model*** schema above highlights the formal specification of the *Family Goal Model* adopted in our proposal. Constraint **(c12)** states that Goals and Tasks must have a non-empty status. In addition, if there is a set of Tasks *(tset)*, such that the *Goal* is a subset of *tset*, then the *Goal is fulfilled* **(c13)**. Moreover, a Goal is a *Requirement* if there is some *Agent Actor act* which occupies a *Role* which in turn is responsible for the Goal **(c14)**. A Goal is an *Expectation*, if there is some specific *Role* that is responsible for the *Goal* **(c15)**.

Several types of link exist in order to connect intentional elements. These links are: *refinement*, *needed-by*, *contribution* and *qualification*.

Refinement is an n-ary relationship relating one parent to one or more children. An intentional element can be the parent in at most one refinement relationship. There are two types of refinement – applied to any kind of parent (i.e. Goal or Task) – that define the logical operator relating the parent with the children:

- *AND-refinement*: the fulfillment of all the n children ($n \geq 2$) makes the parent fulfilled.
- *Inclusive OR*: the fulfillment of at least one child makes the parent fulfilled.

The *Needed-By* relationship links a task with a resource and indicates that the actor needs the resource in order to execute the task. The *Contribution links* represent the effects of intentional elements on qualities, and are essential to assist analysts in the decision-making process among alternative goals or tasks. Contribution links lead to the accumulation of evidence for qualities. The *Qualification* relationship relates a quality to its subject (i.e. a task, goal, or resource).

In our proposal the goal model called Family Goal Model, represent the intentional space of a domain for which the product line is developed. Basically, the adopted goal-oriented approach helps to build artifacts that represent stakeholders' objectives and strategies.

2.3 Feature Sub-Model

As stated above, our proposal offers feature-oriented design and implementation for which Feature Models are a standard visual representation. Feature models support a natural description of a wide range of variability schemata.

Several definitions to what domain experts call *"feature"* exist in the literature (See Haidar et al., 2017b). Due to the lack of space, we will not list them here and adopt the following definition of the term "feature" based on (Haidar et al., 2017b): *A feature is a characteristic or end-user-visible behavior of a software system. Features are used in product line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle* (Apel et al., 2013).

A feature model is a tree whose nodes are labelled with feature names. It also proposes various parent-child relationships between features and their constraints. In fact, if a feature ***f*** is a child of another feature ***p***, ***f*** can be selected only when ***p*** is also selected. Typically, a feature model includes mutual relations between features. In addition, *Mandatory* and *Optional* features are distinguished within the feature model. Note that in our proposal we focus on *Boolean features* identified by a name. In principle, *non-Boolean features* or *attributes of features* may also be of interest in distinguishing applications of the product line. In this paper, we cover essentially Boolean features; non-Boolean features will be studied in future work.

[Feature_Type] := Parent | Child | Abstract | Concrete
[Feature_Availability] := Available | Unavailable
[Feature_Constraint_Type] := Mandatory | Optional | Alternative | Or

Feature Model

name : Name

description : Informal_Definition

is_a : Feature_Type

availability : Feature_Availability

constraint_type : Feature_Constraint_Type

---

root (f) ≡ f **(c16)**

mandatory (p, f) ≡ f ⇔ p **(c17)**

optional (p, f) ≡ f ⇒ p **(c18)**

alternative (p, {f_1 , … , f_n}) ≡ (($f_1 \vee \ldots \vee f_n$) ⇔ p) ∧ ($\bigwedge_{i<j} \neg (f_i \wedge f_j)$) **(c19)**

Or (p, {f_1 , … , f_n}) ≡ ($f_1 \vee \ldots \vee f_n$) ⇔ p **(c20)**

The ***Feature Model*** schema above formalizes the *Feature Model* concepts. All feature names from

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

the set ***F*** of feature names are interpreted as propositional variables, ***p***, ***f*** and f_i represents members of ***F***. Each edge in the tree is defined by exactly one feature constraint, that is, by a declaration of one of the feature constraint types *mandatory*, *optional*, *alternative*, or "*or"*.

A *mandatory* feature definition between a parent feature and a child feature corresponds to a logical equivalence. That is, whenever the parent feature is selected, so must the child and vice-versa **(c17)**.

An *optional* feature corresponds to implication. The implication states that the parent feature may be chosen independently from the child feature, but the child feature can only be chosen if the parent feature is selected **(c18)**.

The *alternative* constraint defines a one-out-of-many choice. The definition of the constraint **(c19)** has the parent feature as first parameter and a non-empty set of child features as second parameter. This constraint is a disjunction, in which, at least, one child feature is selected when the parent is chosen. In addition, we ensure for each pair of child features that no two child features are selected together.

An unrestricted *choice* or "*or"* defines a some-out-of-many choice. Again, the constraint **(c20)** has a non-empty set of child features as second parameter. The selection of parent feature is equivalent to a disjunction of the child features.

Additionally, a set of cross-tree constraints may be defined in the Feature Model. The corresponding propositional formula of the feature constraints and the cross-tree constraints are conjoined resulting in one logic formula that represents the semantics of the whole Feature Model.

2.4 User Story Concept

Our proposed metamodel focuses on agile perspectives. Relevant agile requirements artifacts play, thus a core role within the proposal. This section details the user story concept, which the proposed metamodel integrates. User stories are considered here due to their wide use and to take profit from their effectiveness.

Leffingwell (2011) and Chon (2004), consider them as an increasingly popular textual notation to capture requirements in agile software development. User stories are statements that use a simple template such as "*As a ⟨role⟩, I want ⟨goal⟩, [so that ⟨benefit⟩]*".

[User_Story_Element] := Format | Role | Means | Ends
[Mean] := Subject | Action_Verb | Direct_Object | Indirect_Object | Adjective
[End] := Clarification | Dependency | Quality
[Status] := To_Do | In_Progress | Testing | Done

User Story

identifier : *Identifier*

user_story_elmt_is_a : *User_Story_Element*

mean_is_a : *Mean*

end_is_a : *End*

status : *Status*

$(\forall \mu_1, \mu_2 : User_Story)$ **(c21)**

$is_Full_Duplicate(\mu_1, \mu_2) \leftrightarrow \mu_1 =_{syn} \mu_2$

$(\forall \mu_1, \mu_2 : User_Story)$ **(c22)**

$is_Sem_Duplicate(\mu_1, \mu_2) \leftrightarrow \mu_1 = \mu_2 \land \mu_1 \neq_{syn} \mu_2$

$(\forall \mu_1, \mu_2 : User_Story ; m_1, m_2 : Means ; E_1, E_2 : Ends)$ **(c23)**

$diff_Means_same_Ends(\mu_1, \mu_2) \leftrightarrow m_1 \neq m_2$

$\land E_1 \cap E_2 \neq \emptyset$

$(\forall \mu_1, \mu_2 : User_Story ; m_1, m_2 : Means ; E_1, E_2 : Ends)$ **(c24)**

$same_Means_diff_Ends(\mu_1, \mu_2) \leftrightarrow m_1 = m_2$

$\land (E_1 \setminus E_2 \neq \emptyset \lor E_2 \setminus E_1 \neq \emptyset)$

$(\forall \mu_1, \mu_2 : User_Story ; m_1, m_2 : Means ; E_1, E_2 : Ends ;$

$r_1, r_2 : Role)$ **(c25)**

$same_Role_diff_Story(\mu_1, \mu_2) \leftrightarrow r_1 \neq r_2 \land$

$(m_1 = m_2 \lor E_1 \cap E_2 \neq \emptyset)$

$(\forall \mu_1, \mu_2 : User_Story ; m_1, m_2 : Means ; E_1, E_2 : Ends)$ **(c26)**

$purpose_is_Means(\mu_1, \mu_2) \leftrightarrow E_1 = \{m_2\}$

$(\forall \mu_1 : User_Story ; f_1, f_{std} : Format)$ **(c27)**

$is_not_Uniform(\mu_1, f_{std}) \leftrightarrow f_1 \neq_{syn} f_{std}$

$(\forall \mu_1 : User_Story ; av_1, av_2 : Action_Verb ;$

$do_1, do_2 : Direct_Object)$ **(c28)**

$has_Dep(\mu_1, \mu_2) \leftrightarrow depends(av_1, av_2) \land do_1 = do_2$

$(\forall \mu_1, \mu_2 \in U : User_Story ; do_1, do_2 : Direct_Object)$ **(c29)**

$has_is_a_Dep(\mu_1, \mu_2) \leftrightarrow \exists \mu_2 \in U . is_a(do_1, do_2)$

$(\forall \mu_1, \mu_2 \in U : User_Story ; av_1, av_2 : Action_Verb ;$

$do_1, do_2 : Direct_Object)$ **(c30)**

$void_Dep(\mu_1) \leftrightarrow depends(av_1, av_2) \land \nexists \mu_2 \in U . do_1 = do_2$

The ***User Story*** schema above formalizes the *User Story* (μ_i) concept. Let $U = \{\mu_1, \mu_2, \ldots\}$ a set of user stories in a project. A user story μ is a 4-tuple $\mu_i = \langle r_i, m_i, E_i, f_i \rangle$ where r is the role, m is the means, E

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

$= \{e_1, e_2, \ldots\}$ is a set of ends, and f is the format. In addition, a means m is a 5-tuple $m = \langle s, av, do, io, adj \rangle$ where s is a *"subject"*, av is an *"action verb"*, do is a *"direct object"*, io is an *"indirect object"*, and adj is an *"adjective"* (io and adj may be null).

A user story μ_1 is an exact duplicate of another user story μ_2 when they are identical **(c21)**. The constraint **(c22)** indicates that a user story μ_1 duplicates the request of μ_2, while using a different text (i.e. Semantic Duplicate). **(c23)** denotes two or more user stories that have the same end, but achieve this using different means. **(c24)** represents the case in which two or more user stories use the same means to reach different ends. For the case where two or more user stories with different roles, but same means and/or ends we formalize the constraint **(c25)**.

When there is a strong semantic relationship between two user stories, it is important to add *explicit dependencies* to the user stories, although this breaks the *independent* criterion **(c26)**.

Uniformity in the context of user stories means that a user story format is consistent with the one of the majority of user stories in the same set. Therefore, the format f_1 of an individual user story μ_1 is syntactically compared to the most common format f_{std} to determine whether it adheres to the uniformity criterion **(c27)**.

In some cases, it is necessary that one user story μ_1 be completed before the developer can start on another story μ_2. Formally, the predicate $has\text{-}Dep(\mu_1, \mu_2)$ holds when μ_1 causally depends on μ_2 **(c28)**. Moreover, an object of one user story μ_1 can refer to multiple other objects of stories in U, indicating that the object of μ_1 is a parent or superclass of the other objects. Formally, predicate $has\text{-}is\text{-}a\text{-}Dep(\mu_1, \mu_2)$ is true when μ_1 has a direct object superclass dependency based on the sub-class $do_2 \circ do_1$ **(c29)**.

Implementing a set of user stories U should lead to a feature-complete application. While user stories should not thrive to cover 100% of the application's functionality preemptively, crucial user stories should not be missed, for this may cause a show stopping feature-gap. The predicate $void\text{-}Dep(\mu_1)$ holds when no story μ_2 satisfies a dependency for μ_1's direct object **(c30)**.

3 APPLYING THE METAMODEL

A simple and short example related to an e-commerce product line is outlined below to describe and show the applicability of our proposed metamodel. The e-commerce case study is available in the SPLOT repository (Software Product Lines Online Tools – http://www.splot-research.org/).

We first design the family goal model related to the case study and then follow the practices of our proposed metamodel to generate the correspondent feature model. Due to the lack of space, we only present the application of Goals and Features sub-models, the mapping from the goal model to its correspondent feature model and an example of user story.

Figure 2 shows a concrete "Family Goal Model" of the "Order Process" related to e-commerce case study (modeled using iStar 2.0). It represents the intentional elements and relations. For example, the goal <Item_Available> can be achieved by <Prepare_and_Package_Item>, by <Obtain_From_Stock>, and by <Aquire_From_Supplier>. In addition, satisfactions of the tasks <Obtain_From_Stock>, and <Aquire_From_Supplier> lead to satisfaction and dissatisfaction of the quality <Avoid_Unsold_Stock>. In fact, if the "sales department" adopts a "Make to Stock" strategy, it could could lead to unsold items. However, adopting "Make to Order" strategy will help to avoid a stock of unsold items.

According to our proposed framework, to represent a mapping we should develop a mapping relation *(Φ)* for each mapped task. For example, the *Approve Order (AO)* task is mapped to the *Automatic Approval*, and *Manual Approval* features. Therefore, the mapping relation created is the following:

ΦAO (Approve Order, {Automatic Approval, Manual Approval}).

Moreover, the *Receive e-Payment (REP)* task is mapped to *Debit Card Payment*, *Credit Card Payment*, and *Payment Gateway*. Thus, the mapping relation created is the following:

ΦREP (Receive e-Payment, { Debit Card Payment, Credit Card Payment, Payment Gateway}).

Once the "explicit mapping" between tasks in family goal model and features in feature model is executed, we can start an "implicit mapping" between intermediate tasks, goals, and features. The implicit mapping is performed between "intentional relations" in family goal models and "feature relations" in feature models. For example, following our proposed metamodel, we can infer that the goal

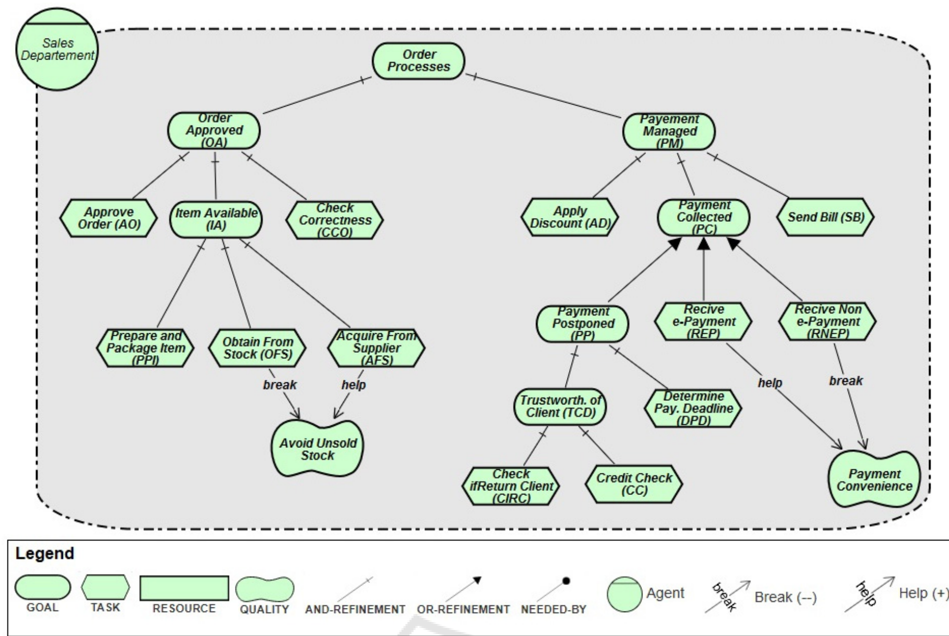

Figure 2: A family goal model for "Order Processes" modeled from the e-commerce case study.

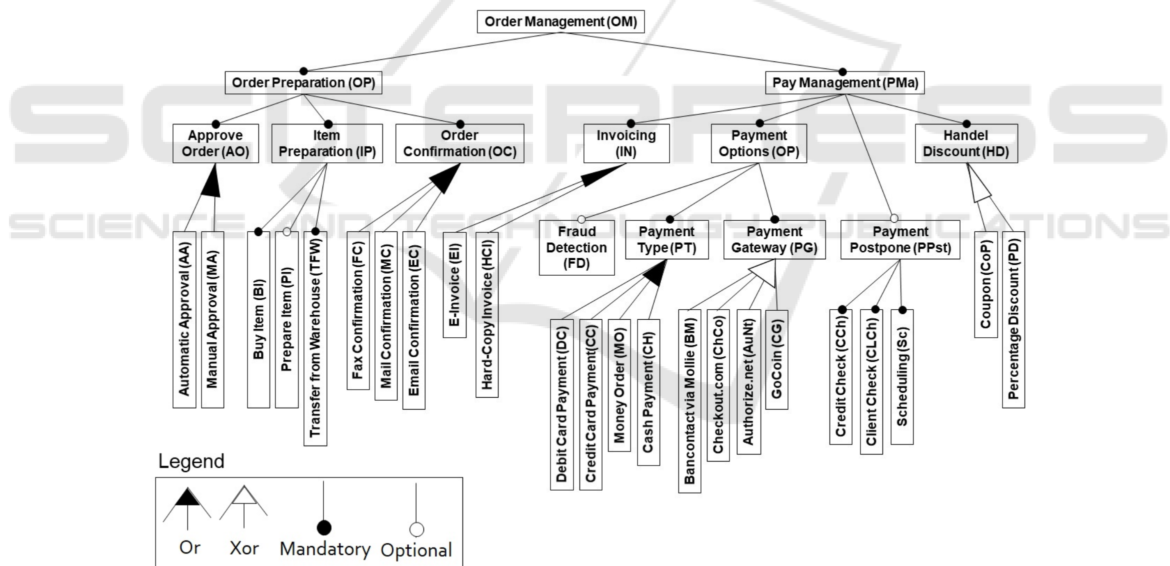

Figure 3: Correspondent Feature Model.

Payment Managed (PM) in the family goal model (see Figure 2) is implicitly mapped to the feature *Payment Management (PMa)* (see Figure 3).

Note that, if a feature is mapped to more than one goal or/and task, then the corresponding feature appears in the mapping relations of all those goals or/and tasks.

Figure 3 shows the corresponding feature model of the family goal model presented in Figure 3. The obtained feature model is represented using a tree graphical notation that could be translated into propositional logic. In addition, the feature model of Figure 3 is generated according to the rules and practices of our proposed metamodel.

Based on the illustrated example, it was shown that the modeled family goal model of "Order Processes" (i.e. Figure 2) captures the intentional variability and describes the intentions behind existing features in the product line of e-commerce. Hereafter we present some mapping as follow:

Order Processes (G-OP) = Order Management
Order Approved (G-OA) = Order Preparation
Payment Managed (G-PM) = Payment Management
Item Available (G-IA) = Item Preparation
Check Correctness of Order (T-CCO) = Order Confirmation (FC ∨ MC ∨ EC)
Obtain From Stock (T-OFS) = TFW
Acquire From Supplier (T-AFS) = BI
Apply Discount (T-AS) = (CoP ∨ PD)

Finally, as an illustration of user stories generated according to our proposed metamodel from the correspondent features, <**Invoicing**> could be realized by several user stories, such as:

As ⟨Accountant⟩, I want to ⟨Generate and Send Invoices⟩, so that ⟨the Invoice can be paid⟩.

4 THE AgiFPL METHODOLOGY

This section illustrates how our proposed metamodel could be applied for the requirements engineering processes of agile product lines, precisely our very own methodology AgiFPL.

Like classical agile product lines methodologies, AgiFPL is a feature-oriented approach involving two classical tiers of product line engineering: Domain Engineering and Application Engineering.

AgiFPL also considers two spaces: the Problem and Solution ones. The problem space calls attention to the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. The solution space represents the developer's and vendor's perspectives (Apel et al., 2013). The Solution Space is not targeted in this work since our proposed metamodel is designed essentially for the requirements engineering concerned by the Problem Space.

Integrating our proposed metamodel to AgiFPL allows modelling and managing intentions, goals, variabilities and commonalities of the product lines. For example, the "Family Goal Models" of our proposed metamodel will guide the development of variability of the product line in the domain engineering, while they are used for the configuration of products in the application engineering.

Figure 4 illustrates the problem space of the Domain Engineering tier. The figure depicts the main steps of the RE process followed in the domain engineering. Based on the strategy of a software vendor who decides to adopt AgiFPL, *Domain Experts* and *concerned teams* apply our proposed metamodel as follows:

1. Execute a sub-process for modeling the family goal models. (i.e. Goal-oriented requirements engineering);
2. Apply the stated practices and rules of our proposal in order to generate the correspondent feature models. (Specifies and design the desired domain – i.e. Domain Design & Feature Backlog);
3. Prioritize the identified features of the designed domain and then document the required user stories. (Apply the correspondent agile requirements practices – i.e. Stories Backlog, tests, …).

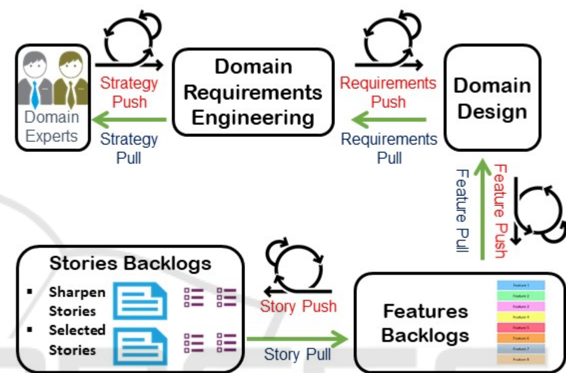

Figure 4: Problem space of Domain Engineering in AgiFPL.

Figure 5 presents the problem space of Application Engineering tier. The concerned requirements engineering process of this tier starts with the goals and the intentions of a specific product owner. These personal goals and intentions are studied, modelled and realized according to our proposed metamodel. For this stage, we propose two optional ways.

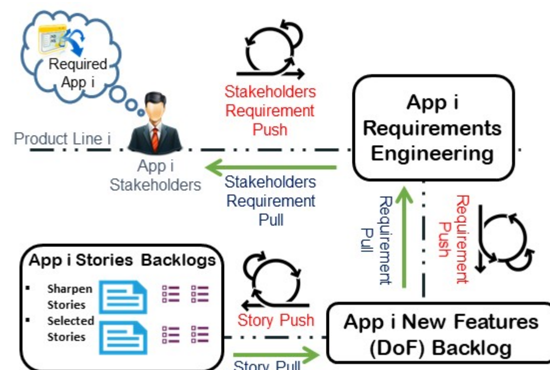

Figure 5: Problem space of Application Engineering in AgiFPL.

Based on the goals and intentions of the “App i Owner” and the context of the “Line i”, the “Line i Team” has to choose the way that best fits the context: ***First,*** in the case where the “line team” has to develop new reusable artefacts that do not exist within the common assets, the team applies the same process used for the domain engineering phase. ***Second*****,** in the case where some stakeholders’ goals do not affect the product line, have not equivalent features in the common assets, and concern a specific product, the “Line Team” produces directly the User Stories and their Backlogs.

5 CONCLUSION

The aim of this paper was to propose an integrated and consistent metamodel for software analysts and developers who adopt agile product line approaches. The research was conducted in the context of defining our own agile software product line called AgiFPL. The main contribution of the metamodel is to allow capturing intentional variability and describing the intentions behind existing features in the agile product line. As a consequence, by using family goal models we can ensure that existing features and variability relations in feature models are aligned with intentional variability in the family goal models. In addition, we can trace back differences in products to differences in the intentions of stakeholders. Moreover, applying intentional elements within agile product lines not only facilitates identifying features in domain engineering lifecycle, but also eases the selection of features based on stakeholder’s intentions and needs in the application engineering lifecycle.

Modeling the organizational and operational context of the domain and application engineering tiers within the flexible and rapid environment of a product line is usually founded on primitive concepts such as those of Goal, Role, Feature, Actor, and User Story. Our paper proposed an integrated metamodel, described its main concepts, illustrated it with an example and related it to our AgiFPL methodology. Our approach differs from others primarily in the fact that it is based on ideas from goal-oriented requirements engineering frameworks, feature-oriented approaches, and agile requirements practices found to be relevant for the solicited requirements engineering approach.

Future work will develop a procedure to discover inconsistencies in mapping results (i.e. generated goal models and/or generated feature models). Finally, we will lead a formal and empirical evaluation of our proposed framework.

REFERENCES

Acher M., Collet P., Lahire P., and France R. B. 2012. Separation of concerns in feature modeling. In *Proceedings of the 11th international conference on Aspect-oriented Software Development, New York, NY, USA, 1-12.*

Apel S., Batory D., Kästner C., and Saake G. 2013. Feature-oriented Software Product Lines. *Springer-Verlag, Berlin Heidelberg.*

Asadi M., Gröner G., Mohabbati B., and Gasevic D. 2016. Goal-oriented modeling and verification of feature-oriented product liens. *Softw Syst Model 15: 257.*

Borba, C. Silva C. 2009. A comparison of goal-oriented approaches to model software product lines variability. In: *LNCS, vol. 5833, pp. 184-253, Springer-Verlag.*

Cohn, M. (2004). User Stories Applied for Agile Software Development. *Boston: Pearson Education Inc*.

da Silva, I. F., Neto, P., O'Leary, P., de Almeida, E., and de Lemos, S. R.. 2011. Agile Software product lines: a systematic mapping study. 41(8) 2011, pp. 899–920.

Dalpiaz, F., Franch, X., and Horkoff, J. 2016. iStar 2.0 Language Guide, cs.SE 2016, arXiv: 1605.07767v3, https://arxiv.org/pdf/1605.07767v3.pdf (accessed on 17-09-2017).

Ernst, N. A., Borgida, A., Mylopoulos, J., Jureta, I. J. 2012. Agile requirements evolution via paraconsistent reasoning. In: *Proceedings of CAiSE’12, pp. 382–397. Springer, Berlin.*

Haidar, H., Kolp, M., and Wautelet. Y. 2017a. Agile Product Line Engineering: The AgiFPL Method. In *Proceedings of the 12th International Conference on Software Technologies – Vol. 1: ICSOFT, 275-285, Madrid, Spain.*

Haidar, H., Kolp, M., Wautelet, Y. 2017b. Goal-oriented requirement engineering for agile software product lines: an overview. LouRIM Working Paper Series, 2017/02, http://hdl.handle.net/2078.1/185846.

Jaqueira A., Lucena M., Alencar F. M. R., Castro J., and Aranha E. 2013. Using i* Models to Enrich User Stories. *In the proceedings of the 6th International i* workshop, pp. 55-60.*

Leffingwell, D. 2011. Agile Software Requirements. Addison-Wesley Professional.

O’Regan G. 2013. Z Formal Specification Language. In: Mathematics in Computing. *Springer, London*.

Pohl, K., Böckle, G., and van der Linden, F. J. 2010. Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Publishing Company, Inc.*

Wautelet Y., Heng S., Kolp M., Mirbel I. 2014. Unifying and Extending User Story Models. In*: CAiSE 2014. vol 8484. Springer, Cham.*

Yu E., Giorgini P., Maiden N., Mylopoulos J. (eds.). 2011. Social Modeling for Requirements Engineering. *MIT, Cambridge, MA.*