Approach for Variability Management of Legal Rights in Human Resources Software Product Lines

M. Derras[1], L. Deruelle[1], J.-M. Douin[2], N. Levy[2], F. Losavio[3], R. Oumarou Mahamane[2] and V. Reiner[1]

[1]*Berger-Levrault, Boulogne Billancourt, France*
[2]*CNAM – CEDRIC, Paris, France*
[3]*Universidad Central de Venezuela, Caracas, Venezuela*

Keywords: Software Product Lines (SPL), Variability Management, Human Resources, Legal Rights.

Abstract: This work concerns software product lines (SPL); it comes from the experience gained collaborating with Berger-Levrault, a French society leader in Human Resources systems. This enterprise serves many French and European territorial communities. They had a variability problem associated to the differences of applicable legal rights in different countries or territories, and this activity was performed manually at a high cost. On the other hand, functionalities were common and mandatory and did not very much. The crucial issue in SPL development and practice is to manage the correct selection of variants. However, no standard methods have been developed yet, and industry builds SPL using on-the-market or in-house techniques and methods, aware of the benefits a product line can provide; nevertheless, this development must return the investment, and this is not always the case. In this work an approach to variability management in case of legal rights applicability to different entities is proposed. This architecture-centric and quality-based approach uses a reference architecture that has been built with a bottom-up strategy. Variability is incorporated to the reference architecture at abstract level considering non-functional properties. A "production plan" to reduce the gap between abstraction and implementation levels is defined.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

This work concerns *software product lines (SPL)*; it comes from the experience gained after 2-years of collaboration with Berger-Levrault (BL) a French society leader in Human Resources (HR) services. The enterprise has a main HR system, called *S-SEDIT*, which serves French and European territorial communities. We realized that the variability problem they had was associated to the differences of applicable law(s) and regulations in different countries or territories and also with occasional local laws changes. On the other hand, functionalities were common and mandatory and did not vary much. The reference architecture (RA) construction and the variability management were identified as main challenges (Derras et al., 2018).

1. *The RA Construction*. The migration to a SPL (Northup and Clements, 2012) was first proposed by BL to exploit variability and avoid configuring almost manually their system for local entities at a high cost, due to different applicable legal rights (laws and regulations). It was decided to use a bottom up strategy, starting from one in-house product, the *Vacation Request System (VRS)*, part of S-SEDIT. On the basis of available documentation, the SPL RA was incrementally built for VRS, using a practical approach (Derras et al., 2018) that considers functional and non-functional components. Relations between components are of type "provide/require". The variation points (Pohl et al., 2005) were mostly non-functional components representing domain and system quality properties that were specified by a quality model (ISO/IEC 25010, 2011). But relevant variation points concerned the different applicable legal rights for each entity and their modifications. In-house systems or technical tools realized the other variation points.

A general research question arise from the SPL engineering practices:

- Is there a mature process to convert the RA high-level abstract reusable components into lower level components to ease the derivation of concrete products of the SPL family?

Derras, M., Deruelle, L., Douin, J., Levy, N., Losavio, F., Mahamane, R. and Reiner, V.
Approach for Variability Management of Legal Rights in Human Resources Software Product Lines.
DOI: 10.5220/0007955905140521
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 514-521
ISBN: 978-989-758-379-7
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

An answer to the above question is to handle the SPL variability.

2. *Variability Management.* A process should be defined to map the RA variation points with lower level modules realizing solutions for variants management, establishing a plan or strategy to be able, later, to derive concrete SPL products. Our present approach follows the guidelines for the Domain Realization phase proposed by the reference model framework for SPL engineering (ISO/IEC 26550, 2015), and the SEI framework (Northup and Clements, 2012). In case of our HR SPL, variability is reduced to the applicability of different laws and regulations that can vary depending on each country or region. Thus, all laws could be defined by a centrally set of regulations and the configuration of a client would consider only one extraction of this set. One advantage would be the ease of updating when for example, laws evolve.

Concerning our previous work on the first challenge (Derras et al., 2018), that is the construction of the RA, the quality properties required by each functional component are considered at an early stage. They are obtained from the domain architectural style(s) and the in-house product properties. This quality-driven approach begins with the domain requirements analysis and has facilitated the identification of variation points and the decision process to select variants.

The goal of this work is to present our advances to respond to the second challenge of legal rights variability management in the HR domain. Guidelines for a variability management process will be presented.

This paper is structured as follows, besides this introduction and the conclusions: Section 2 discusses the context of the architecture-centric quality-based approach to build the SPL RA. Section 3 presents some guidelines for variability management at the domain engineering stage, on the threshold of the application engineering stage; a configuration process is outlined. Finally, Section 4 presents related works.

2 CONTEXT

According to the SEI[1] (Northup and Clements, 2012), a SPL is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission, developed from a common set of core assets in a prescribed way. This definition is consistent with those traditionally given for any SPL, but it focuses on the fact that the SPL systems or products are obtained in a "prescribed way", meaning that there is a process to be followed to derive a specific product. Building a SPL and bringing it to market requires skilful engineering as well as technical management. Many organizations have followed their own activities to achieve a SPL, and there is no standard practice but many different ones (ISO/IEC 26550, 2015), (Käköla, 2010), (Northup and Clements, 2012), (Ouali et al., 2011). The *SPL Engineering (SPLE*) model of (Pohl et al., 2005), the *SEI framework*, and the *standard reference model for SPLE* (ISO/IEC 26550, 2015) coincide on the fact that the SPL development focuses an architecture-centric approach. SPL common and non common functionalities can be identified by using different practices, such as use case models, feature models (Lee et al., 2002) or business processes (BPMN[2]); however, these functionalities must be organized into an evolutionary structure to represent related components of an architecture, which is the SPL backbone.

The capture of the SPL domain knowledge is crucial for SPLE. Usually, the functionality of the domain is reflected into the main available market or in-house products; it can be obtained using different practices as it has been pointed out. The quality goals or non-functional properties for a system, such as performance, reliability, modifiability, availability, etc., are largely determined by the study of the domain architectural style(s), also present in the existing market or in-house products. A software product quality model (ISO/IEC 25010, 2011) can be used to specify quality characteristics. Quality properties have been found to be major responsible of the SPL variability (Siegmund et al., 2012).

Variability is defined as the ability of a system to be efficiently extended, changed, customized or configured for use in a particular context (Van Gurp 2000). The realization or choice of a placeholder or *variation point* (set of variant solutions) of a component or asset is called a *variant* (Pohl et al., 2005). The SPL that is a family of similar systems must be designed to support the variation needed by the concrete products. The RA should be reconfigurable and remain compliant with the architectural style and the quality properties of the SPL domain. The design and selection of variants to conform concrete products concern a *production plan* (Northup and Clements, 2012). Some of the products' constraints may be extracted from a set of existing products, by a *reactive*

[1] Software Engineering Institute, Carnegie Mellon University, USA

[2] Business Process Modeling Notation

or *bottom-up* strategy; which is widely spread in industrial practice and it is the approach we used. However, precise guidelines for a general production plan are not provided in known SPLE frameworks (Northup and Clements, 2012) (ISO/IEC 26550, 2015), (Pohl et al., 2005).
A production plan fills two roles:
1. The *production process* to be used for building the products. It consists mainly in defining the components interfaces; the relations or links between components are already part of the RA. The Domain Realization phase (ISO/IEC 26550, 2015) is focused on building this process, called “architectural texture”. Hence, defining a “production process” or realizing the “architectural texture” are roughly similar activities.
2. The *production method* to specify the models, processes, and tools to be used in the production processes attached to components.
In our case, the variation points considered are non-functional properties; for example the *Hibernate* toolkit is selected among the available technological tools, to provide data availability/persistency and portability; to achieve this, new components could be added or deleted. Non-executable loosely coupled components are produced at this stage.

In conclusion, the management of variability is still a problem. In the literature the challenge of mapping abstract RA components into concrete components to retrieve the right code module of a variant is always mentioned but not detailed. However no mature method, process or approach to bridge or reduce the gap between the abstract RA design level and the more concrete application level has been adopted in industrial practice. We propose an approach for variability management by adapting the guidelines of the Domain Realization phase of (ISO/IEC 26550, 2015) to the Production Plan of (Northup and Clements, 2012).

3 AN ARCHITECTURE-CENTRIC AND QUALITY-DRIVEN APPROACH FOR VARIABILITY MANAGEMENT

We have covered the *Domain Engineering* lifecycle of SPLE until the *Design* phase, where the RA for the VRS of the S-SEDIT HR system was built (Derras et al., 2018). We recall that a bottom-up strategy was followed with a single in-house product. The RA was incrementally constructed considering first functional components extracted from the enterprise business processes; secondly, the quality required by functional components was incorporated as non-functional components, and in the 3rd place, the components were distributed according to the domain styles for HR, that is layers, event-based, and client/server model for communication. Finally, the variation points were determined by using the quality properties required by the functionalities, which were mostly common mandatory components; the majority of the variation points were non-functional components. We faced the problem of the compliance with laws and regulations. So, an automatic configuration system was needed to overcome the different applicability and changes of laws. The laws variability will be treated considering the production plan (process and method) proposed in the SEI framework to provide the component interface specification, which is the main activity of the Domain Realization stage.

3.1 Guidelines for a Production Plan

1. Establish a Mapping between the RA Components and their Solutions.
Input: the RA (not shown here to abridge the presentation) structured according to the HR domain style; the enterprise technological platform (list of reusable in-house or to be developed software artefacts).
- For each component and sub-component in a layer, the relation with the corresponding technical tool or with an architectural component present in the subsequent layer, is analysed.
- Each variation point component is annotated with a list of variant technological tool(s)/in-house solution(s).
Output: components annotated with possible solutions.
2. Establish the Production Process, Adapted from (Northup and Clements, 2012).
(New components can be added, and/or existing components can be deleted).
Input: RA components annotated with reusable technological tool(s)/in-house solutions.

- e.g. in *Presentation Layer*, the functional component *Supervisor* requires to *Check Signature Right* in *Process Layer,* connecting to the variation point <<*Signature Hierarchy*>> to provide *Security* and *Authenticity,* which is solved by the in-house system *X.Net*, developed with *Spring*; the services provided by *Log4j, SLF4j, Spring Security,* and *Kerberos Security* are used to satisfy the security and authenticity quality properties; this solution requires to use the *Signature Rights DB* from *Data Layer;*
  - For each component:
- Specify the *provided/required parameters*, which were extracted from the RA functional and non-

functional requirements during phases *PL Scoping* and *Domain Requirements Engineering*; techniques/ methods used: *ISO/IEC 26550 SPLE, BPMN, ISO/IEC 25010 Quality Model (*e.g. *Supervisor* requires the parameters *entity, staff-id, sign-approval, notify, security* and *authenticity*; the method *Check Signature Rights* calls *<<Signature Hierarchy>>* to receive the *sign-approval* parameter; the method *Take Decision* receives *notify* from the *Send Notification* component; notice that the two sub-components *Check Signature Rights* and *Take Decision* of the original RA *Supervisor* component are now expressed as methods calls in Figure 1);

- Specify techniques/methods used: BPMN, criteria to map BPMN activities to architectural components (e.g. *Supervisor (entity, staff-id, sign-approval*);
- Define the *component main methods*; they can correspond to the RA abstract sub-components of the component (e.g. *Supervisor* calls methods *Check Signature Rights* and *Take Decision)*;

Output: RA structure expressed as a UML 2.0 logic view, showing the components interfaces (see Figure 1).

The parameters of the interfaces of components are defined as follows: *entity = {country, region, city, community, … }; staff-id = {id, password, status, … }; access, notify, sign-approval, rights-check = {yes, no}; period = {dates for the vacation request or number of days required and period of the year}; eval-required = {yes, no}* (push button to activate the evaluation in *Process Layer)*; *updated-data*: results of administrative tasks related with the vacation request of an employee; *law*: data structure representing the law text that will be expressed as a set of rules; *config-file:* configuration file expressing the laws possible changes and/or the injection of new components. The quality properties required by each functional component are shown as parameters; they are actually the links to the components providing the appropriate solution.

All the connections between layers follow the *REST* system architectural style, where resources are directed only through their URLs, via the http/https protocol. The *JAMon* system monitors all connections between architectural components, including message passing and *RMI*[3].

In the *Context* external system shown in green in Figure 1, the *<<Implementation>>* component is considered as the *Java* programming environment of the *Configuration* component. It is the system used to manage the RA variability. This component will actually implement the laws variants selection. The *Configuration* component should basically contain the tools/mechanisms to configure the system required by a client, with the selected variants. Variation points *<<Login>>, <<Signature Hierarchy>> and <<Access Rights>>* are realized by in-house modules or mechanisms, so they are already implemented solutions for a variant.

Most of the functional components of the RA, such as *Supervisor*, *Employee* and *Administrator* in *Presentation Layer*, and *Administrative Tasks*, *Evaluate Case*, *Send Notification* and *Send Response* in *Process Layer* are common mandatory components that are supposed to have their coded modules available, such as *Administrative Tasks* that is solved by the in-house *e-Sedit* system. In this case, required quality properties are taken into account by these in-house modules. MVC[4] concerns the independence between user interface and process layer, ensuring maintainability; *<<MVC Client-side>>* and *<<MVC Server-side>>* are solved by *Angular Js*; security/authenticity required by *<<Login>>* and *<<Signature Hierarchy>>* is solved by protocol *LDAP* and by the in-house *X.Net* system with *Spring Security* and *Kerberos Security* and using *Access Rights DB*; *<<Data Access>>* solves *persistency/availability* with *Hibernate*, and finally *<<Data Base>>* is actually settled to *Oracle* by enterprise decision. The quality required here is *suitability/correctness*, which is realized by *Data Schemas*.

However, it is not enough to have an annotation (for example the @*Variability* tag shown in section 3.2) at the RA abstract level to force the choice of a concrete solution. At the abstract level, a feature is represented by a component that is potentially implemented by different concrete components (mechanisms/modules/tools). Thus, there are at least three levels:

1. The RA abstract level describing the functionalities (not shown here to abridge the presentation);
2. The RA concrete level including components variation points and business patterns (domain styles, layers, etc.), and
3. A more concrete level, where components interfaces show parameters representing provided/ required resources and able to make calls; this level is closer to the implementation level, with the code supposed to be distributed in different files. Figure 1 integrates these two architectural views. The @*Variability* label would then appear at the more concrete level and to generate the code that corresponds to the choice in the *config-file.*

[3] Remote Method Invocation

[4] Model View Controller

SCIENCE AND TECHNOLOGY PUBLICATIONS

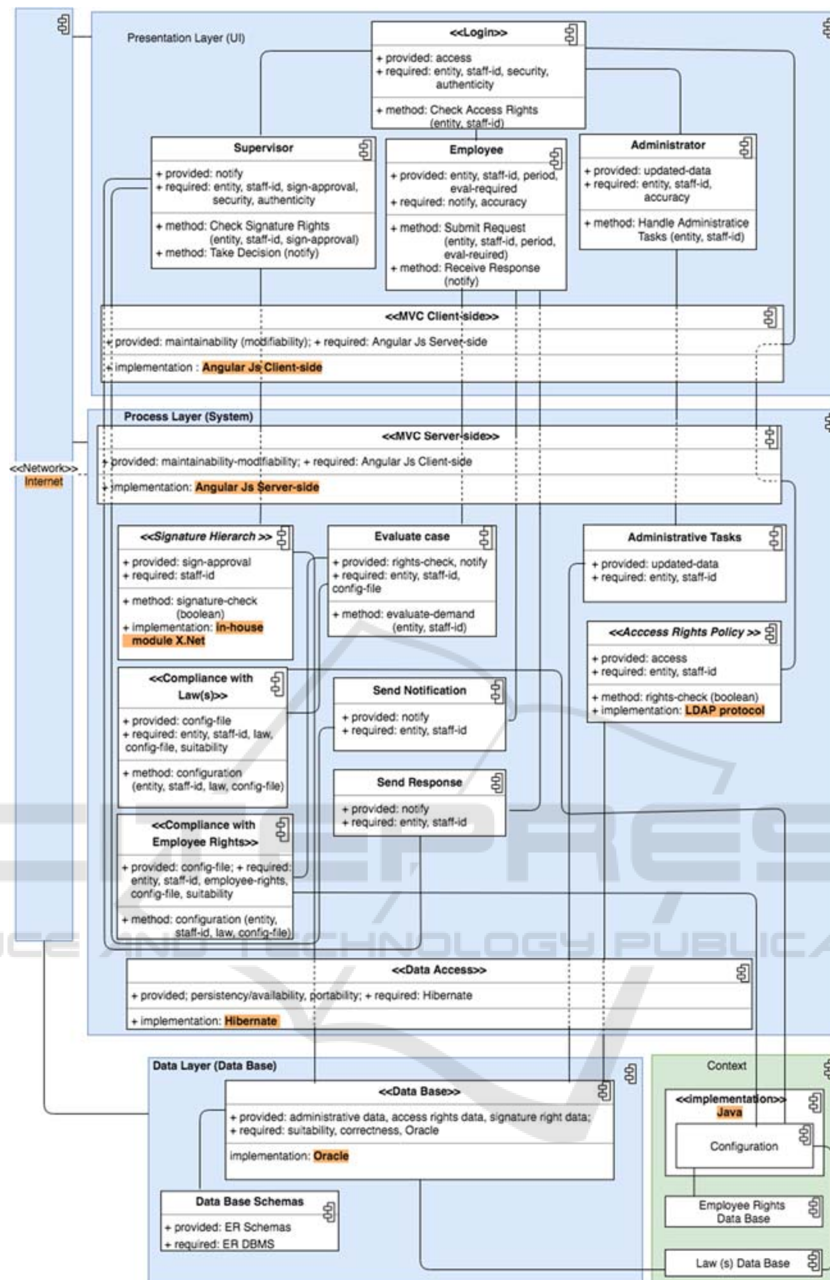

Figure 1: The VRS RA of the S-SEDIT HR system with components' interfaces; in orange the variants' choices.

3.2 Handling Variability - Compliance with Legal Rights

The RA variation point *<<Compliance with Law(s)>>* responsible of reflecting the changes of the law (see Figure 1), will be taken as an example. At this abstraction level, the method *configuration* in *<<Compliance with Law(s)>>* requires the *Configuration* component of the *Context* external system to update/build a *config-file* with the Law(s) changes or adaptations that will be stored in the *Law(s) Data Base* (see Figure 2), and the RA Data Base will be also updated.

The *Configuration* component conformed by an *Eclipse plug-in* (Vélitchkoff, 2019), will be used by *Maven Build* to Tag the variation point with *@Variabiliy* annotation, indicating also the injection mechanism, for example *AspectJ, Javassist* or *Java.lang.instrument*.

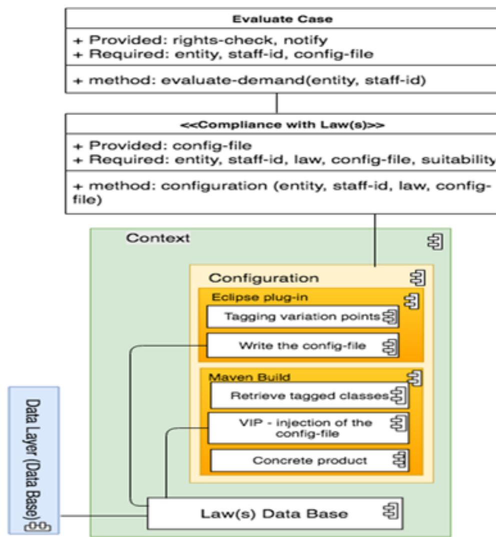

Figure 2: *<<Compliance with Law(s)>>* - details.

3.2.1 The Vip (Variability and Injection Pattern) Framework

The goal is to add, remove or modify functionalities using a simple structure and without Java code updating; software development is reduced to business-related rules of the form *if condition (entity) then command (entity, result)*. The condition as the command are java beans, a beans container assumes their initialization. The rules are defined as a sequence of conditions to be satisfied and commands to be executed; in Java they are classes that must have been defined to be injected into the configuration file. The rules answer some main questions:

- *Who are the Entities?* They are the business objects, e.g. in case of VRS, they are the parameters of the interfaces of the RA components such as *entity, staff-id*.

- *For which Type of Result?* They are values, e.g. Boolean, integer, table, any java classes with eventual side effects such as a display command, a request to a database, a REST service invocation or an http invocation, … etc. In this context, the definition of a configuration is a sequence of rules that is specified iteratively by a given configuration file. The configuration method will provide the updated-config-file (text or XML file) from the Configuration component of the Context system. An example of a config-file in Java is the following:

```
bean.id.1=calcul
calcul.class=commands.Command
# if(condition) then command
calcul.property.1=condition
calcul.property.1.param.1=condition
calcul.property.2=command
calcul.property.2.param.1=command
```

Let us consider the computation of the authorized leave for an employee, according to his status and entity. The rules could be of the form:

– If the employee holds the position (status), …
– If the employee is native of the entity, …
– If the employee seniority is … …
– 1 additional day each 5 years of holding the position
– If the community is in region, …

An example of the configuration file could be:

```
bean.id.1=conditionSeniority
conditionSeniority.class=ConditionSen
iority
conditionSeniority.property.1=numberO
fYearsOfSeniorityRequired
conditionSeniority.property.1.param.1
=5
bean.id.2=operationSeniority
operationSeniority.class=OperationAdd
ition
operationSeniority.property.1=operand
operationSeniority.property.1.param.1
=5
# Rule : if conditionSeniority then
operationSeniority
bean.id.3=commandSeniority
commandSeniority.class=commands.Comma
nd
commandSeniority.property.1=operation
commandSeniority.property.1.param.1=o
perationSeniority
commandSeniority.property.2=condition
commandSeniority.property.2.param.1=c
onditionSeniority
bean.id.4=invoker
Invoker.class=commands.Invoker
Invoker.property.1=command
Invoker.property.1.param.1=commandSen
iority
```

In the complete Java example provided in (Wu, 2018), variability at *entity* level is satisfied; only one configuration file is used for each *entity* (French communities in this case). Once the constraints are known, it is possible to define all the rules for each French community and implement them using the VIP framework. In the example taken in (Wu, 2018), 60 rules were required to model vacation laws. The size of the configuration file could be a problem that may require the use of the *service locator*[5] pattern.

[5] https://en.wikipedia.org/wiki/Service_locator_pattern

4 RELATED WORKS

On the design of a configuration process for SPL, extensive literature reviews have been written; many works from almost two decades on the configuration management in SPL are found showing the interest of the SPL community on this subject that it is still an open problem. *Software Configuration Management (SCM)* is the discipline of managing the evolution of complex software systems, but this is not always the case for SPL (ISO/IEC, 2015), (Thao, 2012).

The problem that conventional SCM tools are not suited for configuration management in SPL is faced in (Thao, 2012). He presents an interesting work on a configuration management prototype called Molhado SPL that is designed specifically to support the evolution of SPL. We are interested in this work because it addresses the evolution problem at domain engineering level instead of at code level, which is also our case, and because it presents an extensive literature review on the subject.

The work of (Soujanya and Rao, 2015) affirms that SPL achieve significant cost and effort reduction through large-scale reuse of software product assets. SPL consists of core assets and custom assets, which are shared among multiple products, and they evolve independently. The evolution of products in SPL is in time and space dimensions. Available SCM systems are suitable for the product evolution but inadequate for SPL systems. A software version management system is proposed for SPLE. It supports cases of assets changes from core and custom assets to concrete products. Component sharing is also proposed.

In (Van Gurp and Prehofer, 2006) a combination of traditional variability tools and files subversion is proposed to support product derivation and configuration management. Tools like Apache Subversion (SVN) are used to maintain current and historical versions of files such as source code, web pages, and documentation. Its goal is to be a compatible successor to the widely used Concurrent Versions System (CVS). The relevant problem found here is still the link between the variability model and the concrete artefact.

In (Krueger, 2002), the products are considered simply transient outputs. All changes are made to the common and variant artefacts. There is only the SPL to be managed as a unique product. Component composition is the process of composing different components to form a product. Software customization is the process of specializing a product. The component composition and software customization layers use the configuration management layer to supply the correct version at a fixed point in time. Krueger suggests using the 'context' approach that represents a possible composition of component versions. Customized products are instantiated from customized components, which are instantiated by selecting the appropriate variant of the variation points in the domain space. The variation points are mapped to common variant files and components.

The work of (Uk and Lee, 2015) presents a methodology based on a decision model with associated tool supports, to design a SPL model, analyse features, and configure a valid product. XML is used to model the SPL, where a schema is defined to specify core assets. The decision model is represented as a UML activity diagram, and the SPL has to deal with the combinatorial problem of variability. It extracts all the properties of required features from the SPL model. It uses the Alloy formal language, and the Alloy Analyser. The SPL is supposed already designed and how the components are obtained is not discussed. The SPL RA is not mentioned, the SPL model is supposed to contain all the information concerning the components. The work is more oriented towards product configuration at application engineering.

The recent works of (Mazo, 2018) present the VariaMOS SPLE framework focused on a complete process to develop dynamic product lines, in the sense that an intermediate automatic adaptation layer is placed between domain and application engineering layers of the standard SPLE lifecycle. A run-time self-adaptation of the system is simulated at application engineering stage, once the product is derived. The importance of the RA is mentioned but the emphasis is placed on the bottom-up strategy, where market, third party or in-house components are specified with the High-Level Constraint Language (HLCL) for SPLE used through the whole process. The code can be modified using a "fragmentation" technique that is similar to the Java injection pattern discussed in section 3. This approach that handles variability in abstract components is similar to our approach, however it is not architecture-centric

The work of (El-Sharkaway et al., 2018) states that ideally a variability model is a correct and complete representation of SPL features and constraints among them. Together with a mapping between features and code, only valid products can be configured and derived. However, in practice the modelled constraints might be neither complete nor correct, which causes problems in the configuration and product derivation phases. The work presents an approach to reverse engineer variability constraints from the implementation, and thus improve the correctness and completeness of variability models.

To conclude, the SPL configuration process roughly consists in setting the convenient variants. In

this work we are concerned with SPL in the HR domain, where the compliance with legal rights greatly affects the system evolution and it is almost manually achieved (Dai, He, Xing, 2015), (Mazo et al. 2014). In the SPL HR context processes that could translate automatically or semi automatically the evolution of the law to a particular concrete product, and how the law representation can be included into a configuration mechanism, were not found in the works studied.

5 CONCLUSIONS

This paper considers the variability management of laws and regulations for a SPL in the domain of human resources. Guidelines are proposed for a process driven by the RA; the variability model considered is extracted from the domain and product quality model, since most of the variants are qualities required by the HR functionalities, which are common and mandatory. This process is on the threshold between the abstract RA level and the concrete product derivation level. It establishes a production plan at a more concrete level than the abstract level imposed by the RA that is inspired from (Northup and Clements, 2012). At this level the variants choices are facilitated by the definition of components interfaces. A configuration process, to inject into the system the application of territorial laws changes expressed as rules, is outlined. The design of an automatic configuration system to deploy on a SaaS multi-tenant cloud a concrete S-SEDIT product, compliant with the client demand, is an on-going work. It should be noticed that in this case, the configuration system is what is important; however, the SPL RA should evolve into concrete micro-services architecture.

REFERENCES

ANSI/IEEE, 2005. *Standard for Software Configuration Management Plans*, ANSI/IEEE Std 828-2005.

Dai, L., He, Y. and Xing, G., 2015. *Intelligent Information Management*, SciRes Online 7, 1-6, http://www.scirp.org/journal/iim, http://dx.doi.org/10.4236/iim.2015.71001.

Derras, M., Deruelle, L., Douin, J. M., Levy, N., Losavio, F., Pollet, Y. and Reiner, V., 2018. Reference Architecture Design: a practical approach, *ICSOFT 2018, Porto, Portugal, 599-606.*

El-Sharkawy, S., Dhar S.J., Krafczyk, A., Duszynski, S., Beichter, T. and Schmid, K., 2018. Reverse Engineering Variability in an Industrial SPL: Observations, Lessons Learned, *SPLC18 1, 215-225.*

ISO/IEC 25010, 2011. *Systems and software engineering-Systems and SQuaRE - System and software quality models*. ISO/IEC JTC1/ SC7/ WG6, Draft.

ISO/IEC 26550, 2015. *Software and Systems Engineering– Reference Model for Software and Systems Product Lines*, ISO/IEC JTC1/SC7 WG4.

Käkölä, T., 2010. Standards initiatives for SPLE and management within the international organization for standardization, *System Sciences (HICSS), 43rd Hawaii International Conference, IEEE, 1-10.*

Krueger, C.W., 2002. Variation management for SPL, *SPLC2: 2nd International Conference on SPL, London, UK, 37--48, Springer-Verlag*.

Mazo, R., Assar, S., Salinesi, C. and Hassen, N.B., 2014. Using SPL to improve ERP Engineering: Literature Review and Analysis, *LAJC, Vol. 1(1*).

Mazo, R., 2018. *Software Product Lines, from Reuse to Self Adaptive Systems*, HDR, Paris 1, France.

Northup, L. and Clements, P., 2012, *A framework for SPL practice*, 5.0, SEI, Carnegie Mellon University.

Ouali, S., Kraiem, N. and Ben Ghezala, H., 2011. Framework for Evolving SPL, International *Journal of Software Engineering & Applications (IJSEA), 2 (2).*

Pohl, K., Bockle, G. and Van Der Linden, F., 2005. SPLE: Foundations, Principles, and Techniques, *Springer*.

Siegmund, N., Rosenmuller, M., Kuhlemann, M., Kastner, C., Apel, S. and Saake, G., 2012. SPL Conqueror: Towards Optimization of Non-functional Properties in SPL, *Soft. Qual. Journal., 20, 3-4, Sept, 487-517(31).*

Soujanya K.L.S. and Rao A., 2015. A Systematic Approach for Configuration Management in SPL, *Inter. Multi Conference of Engineers and Computer Scientists 2015 (IMECS), Vol. I, March, Hong Kong*.

Thao, C., 2012. *A Configuration Management System for SPL,* Phd thesis, University of Wisconsin-Milwaukee, USA, August.

Uk, S. and Lee, J., 2015. An Effective Methodology with Automated Product Configuration for SPL Development, *Hindawi Publ. Corp, Mathematical Problems in Engineering, Vol. 2015, ID 435316, 11 pages, http://dx.doi.org/10.1155/2015/435316.*

Vélitchkoff, V., 2019. *Développement de plugins pour la gestion de la variabilité dans les lignes de produits*, Rapport de stage, CNAM, Paris, France.

Van Gurp, J., 2000. *Variability in Software Systems, the key to software reuse*, Univ. of Groningem, Sweden.

Van Gurp, J. and Prehofer, C., 2006. Version management tools as a basis for integrating product derivation and software product families, *Workshop on Variability Management, SPLC 2006, 152.06/E, 48--58.*

Wu, M., *2018. Variability&Injection Pattern*, Rapport de stage 2e année, ENSIIE-CNAM, Paris, France.