# Managing a Software Ecosystem Using a Multiple Software Product Line: a Case Study on Digital Signage Systems

Simon Urli, Mireille Blay-Fornarino, Philippe Collet, Sébastien Mosser, Michel Riveill

# Managing a Software Ecosystem Using a Multiple Software Product Line: a Case Study on Digital Signage Systems

Simon Urli, Mireille Blay-Fornarino, Philippe Collet, Sébastien Mosser, Michel Riveill
Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
Email: {urli,blay,collet,mosser,riveill}@i3s.unice.fr

***Abstract*—With the advent of Web 2.0, the growth of developer teams and user communities increases the number of software ecosystems: software platforms developed and maintained in a decentralized way by external contributors. As complexity grows, these large software systems become more and more complex to manage and to adapt to specific user needs. In this paper, we report on a case study on the development of a digital signage software system called YourCast. Based on several years experience evolving YourCast from a single system to a medium-scale ecosystem, we show how organizing it as a multiple software product line helps in organizing the software platform, taming some management tasks for a growing community, and giving more capabilities to final users to build their own products.**

## I. Introduction

As software increasingly grows in size and complexity, the software industry is facing new challenges. First, in most cases the number of features needed by potential customers exceeds what can be developed and maintained internally by a company. The developed software, or at least a part of it published as an open platform, should be available to a community of developers. Second, a fully tailored product is almost needed for each user, and mass customization should be facilitated by the possibility to extend the software product with externally developed artifacts [1]. The software ecosystem approach meets these requirements by proposing to build large software systems from a software platform. This platform aggregates components developed in a decentralized way by multiple contributors, being internal or external to the organization that maintains the platform.

The emerging of Web 2.0 few years ago changed a lot the involvement of users in many projects, including software projects [2]. Regardless of well-known software ecosystems such as Eclipse, Google Play, or Apple Store, there exists numbers of smaller ecosystems emerging inside free and open-source software projects. The fast evolution of the platform and its components is also a distinctive property of all these ecosystems.

Besides, *software product lines* (SPLs) support the development of a family of similar software products from a common set of shared assets [3], [4]. SPLs are now commonly adopted in industries in order to optimize the reuse of quality software components. The possible shift from SPLs to software ecosystems has also been identified by Bosch *et al* [1], who characterized it by a move from centralization to decentralization, few to many and multiple contributors and a more complex evolution settings. In the meantime, research works have been conducted on the usage of *Multiple Software Product Lines* (MSPLs), defined by Holl *et al* as "*a set of several self-contained but still interdependent product lines that together represent a large-scale or ultra-large-scale system*" [5]. Consequently considering a large software development as a MSPL brings benefits of SPL techniques, while taking into account issues such as multiple software contributors.

In this paper, we show how organizing a software ecosystem as a MSPL helps in structuring the software platform with separation of concerns, taming some management tasks for a growing community while ensuring consistency between artifacts, and giving more capabilities to final users to build their own products. We overview the following approach by reporting on a detailed case study related to the development and evolution of a medium-scale software ecosystem following a MSPL approach. The project, called YourCast[1], provides digital signage software systems with an innovative architecture enabling to easily integrate external sources of information while providing end-users automation and appropriate customization support.

The intended audience of this paper are *i)* researchers and practitioners on software ecosystems that are interested in the SPL paradigm and its benefits; *ii)* SPL researchers dealing with fast evolving systems ; and *iii)* MSPL researchers taking an interest in applications and lessons learned.

The remainder of the paper is organized as follows. The next section introduces our case study and the associated ecosystem. In section III we explain how we evolve from an ecosystem to a MSPL. In section IV we analyse benefits and drawbacks of the MSPL approach to support an ecosystem, while discussing related work. Section V concludes this paper and discusses future work.

## II. A Software Ecosystem for Digital Signage Systems

We present in this section the background of our case study. First we introduce the underlying concepts of a *Digital Signage System* (DSS). We then discuss the birth of the YourCast system and its evolution towards an ecosystem.

### A. On Digital Signage Systems

A DSS is a software system whose purpose is to display information or advertising on screens. The decrease of electronic

[1] http://www.yourcast.fr

products prices and the democratization of the Internet encouraged the usage of DSS to broadcast dynamic information, mainly from the web, both in public institutions and private companies. Depending on locations, targeted populations and broadcast information types are different, resulting in many variations of these systems [6].


Fig. 1. Screenshot of one of our DSS

Figure 1 shows a concrete example of a DSS deployed in a department of the University of Nice. The chosen design is on purpose very sober in order to emphasize contents. This DSS uses four different zones to display information: one at the top (1), one in the middle (2), one on the right (3) and one at the bottom (4). The first one at the top (1) displays a clock and forecast information. The second one (2) is configured to display plannings, picture albums and so one. On the figure it displays internal information about the department. This zone uses a fade transition system to go from an information to another. The third zone (3) on the right presents RSS feeds in details, using a push system from top to bottom as transition mechanism. Finally the fourth zone (4) at the bottom also displays RSS feeds but only the titles, and it uses a scrolling system from right to left as transition mechanism.

Defining such a DSS needs to take into account three kinds of end-users, the *viewer*, who does not know how the system works but is the main target of displayed contents or advertising, the *manager*, who needs to manage contents of her DSS, the *product owner*, who wants to build a new DSS. To define a given DSS a product owner has at least to answer to some questions related to the different DSS concepts:

- *Sources*: which kind of information do I want to display and where do I get them? *In our example on Figure 1, she chose to display RSS feeds, picture albums and forecast.*
- *Renderers*: how do I want to graphically display the information set? *RSS feeds are presented in details on the right hand side and in short (only titles) at the bottom.*
- *Transitions*: how does each piece of information move from one to another? *A scrolling or fading transition mechanim is typical here.*
- *Zones*: where do I put my information contents on the screen? *For example, she chose to display RSS feeds in the zones 3 and 4.*
- *Layout*: how many information zones do I need? Which layout to organize them? With which design? *She chose a layout containing four zones with a blue sober design.*

In the remainder of the paper, we discuss DSS with the point of view of a product owner who needs to satisfy viewers and managers expectations.

### B. From a Framework to an Ecosystem

We started to create an internal DSS (jSeduite[2]) for our campus in 2005. At that time, our main interest was to be able to display information from our internal services (students planning, general university information, etc.). As much information was accessed through the Internet, we decided not to rely a complex and cumbersome content management system that would duplicate or badly aggregate the needed data. We thus decided to create a decoupled service-based Web infrastructure organized around the following key components. The *player* is the product display seen by the viewer. It is basically an animated web page using JavaScript components for renderers and transitions, as well as HTML and CSS for layout and zones. *Sources* are Web services from where the DSS gets information to display. They can be internally developed to access information from campus specific services, or can be directly provided by third parties such as FlickR or Twitter API. The *customizer* is a web service dedicated to the management of the DSS: it allows a manager to specify parameters of sources and player. It must be noted that this component is only used to set parameters of other components (*e.g.* which Twitter account a source is referring to), not to configure the architecture of the DSS (*e.g.* how many sources are used or in which zones are they displayed).

Quickly after deploying a first DSS, we faced different deployments in other institutions and users' needs started to grow, people requesting for more and more information to be displayed. This encouraged us to open our API, switching from jSeduite to YourCast, in order to allow new developers, essentially students as a start, to contribute to our platform. Then the different parts of the applications (player, sources, and customizer) were almost independently maintained by different teams. We reached here the definition of software ecosystems given by Mens et al: "*software ecosystems consist of a relatively closed core software system that provides the basic functionality and that is developed by a more or less stable core team of developers surrounded by a large collection of contributions provided by peripheral developers or even end-users*" [7]. We more precisely identify the YourCast system to be an application-centric software ecosystem, as defined by Bosch in his taxonomy [1].

However managing the growth of this community became challenging. We identify three kinds of contributors, each leveraging a challenge discussed in the next sections.

- **Passive users** are the end-users of our applications (viewers, managers and product owners). They can report bugs and propose new requirements. Product owners want their DSS to be tailored to their vision without needing a developer to build the DSS. **They thus want to use a tool helping them to master the**

[2]http://jseduite.polytech.unice.fr

**variability, i.e. to configure and get their envisioned products.**
The YourCast system is deployed in three different classes of environment with their own users: universities (places of learning or research), facilities for persons with disabilities (schools and centers) and large events (gatherings, conferences, etc.). For example, in universities, viewers are students, teachers, researchers or visiting people from industry, while managers are project assistants, secretaries or teachers. Product owners are site or community manager. For all deployments already completed, we now have about 50 managers and 10 product owners.

- **External developers** are contributors of the ecosystem. They do not have all the knowledge of the system and contribute by implementing new features and fixing bugs. **They have to be able to contribute separately in different parts of the system with minimal interference.**
Since the inception of the project, about 50 people have contributed (see table I). There are wide variations depending on the time of year. Currently we only have 3 contributors, whereas during the last year we had at some times more than 10 simultaneous contributors. Much of the contributors are students of different levels and therefore address different concerns or parts of the system. Some industrial partners also contribute, in particular to integrate their own sources of information in the ecosystem.

- **Internal developers** are leaders of the ecosystem. They have the knowledge of the global architecture and manage all the contributions of external developers to integrate them in products. **They have to be able to maintain the platform and to ensure the consistency of all products despite the fast evolution of the ecosystem.**
Currently, only 2 people in the team have this knowledge.

The table I gives metrics comparing the jSeduite framework centric application, i.e. our starting point in terms of development, and the YourCast ecosystem supported by a multiple software product line (MSPL). We show in the next section how this MSPL allows to manage more easily the community of external developers and how it enables passive users to make their own DSS, explaining the increase in the number of deployed DSSs.

TABLE I. METRICS ABOUT JSEDUITE AND YOURCAST

| Metrics | jSeduite | YourCast |
|---|---|---|
| Lifetime | 09/01/2007 - 03/31/2011 | 01/01/12 - 03/05/14 |
| # of days | 1260 | 780 |
| # of internal developers | 2 | 2 |
| # of external developers | 14 | 19 + 15[3] |
| # of commits | - | 1410 |
| # of external commits | - | 608 |
| # of internal commits | - | 802 |
| KLOC | 200 | 470 |
| # of deployed DSS | 4 | 18 |

[3]These developers are not involved in the commits count.

## III. SUPPORTING ECOSYSTEM WITH A MSPL

In order to manage the different requirements presented in the previous section, we built a MSPL as a part of the YourCast project. As shown in Figure 2 (write on red numbers on the left), this MSPL is composed of:

1) a domain model describing how the SPLs are interrelated,
2) several feature models describing the variability of each SPL and constraints between them[4],
3) assets used both at the SPL level and at the MSPL level and,
4) generation tools associated to each SPL and to the MSPL.

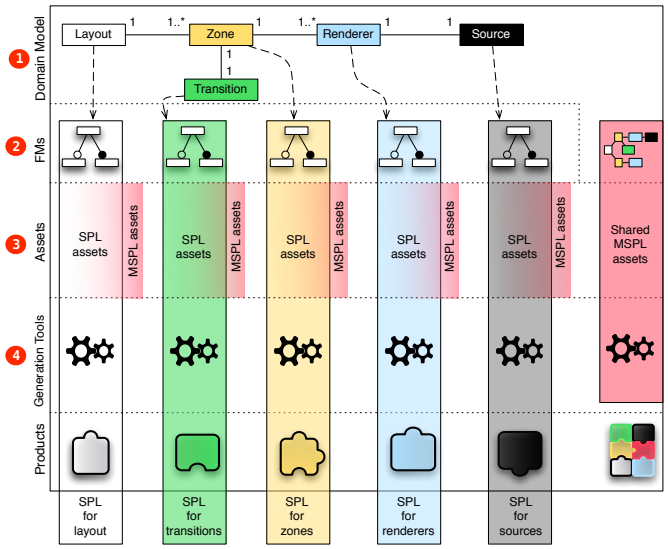


Fig. 2. Overview of the MSPL

In this section we show *i*) how the MSPL helps passive users in order to create a valid DSS product in composing configurations; *ii*) how the external developers can work with less interference following the MSPL separation of concerns and *iii*) how finally the internal developers are able to maintain the platform ensuring consistency properties for the MSPL.

### A. DSS: a Composition of Configurations

From the product owner's point of view a DSS is easily described as a set of linked artifacts. For example, the DSS presented in Figure 1 can be seen as a layout, containing 4 zones, each zone containing a transition system and some renderers to display sources of information like RSS feeds, forecast, picture albums, internal news, etc. An abstract model of this assembly is depicted in Figure 3.

Each artifact in this representation corresponds exactly to one instantiation of the concepts described in section II-A with specific configurations such as *Fade* for a transition or *FlickR* for an information source.

Moreover a well-formed DSS must respect some constraints: for example, a DSS without layout does not make

[4]These constraints are not on the metamodel itself, but between the features of the different SPLs, as it will be exemplified afterwards.

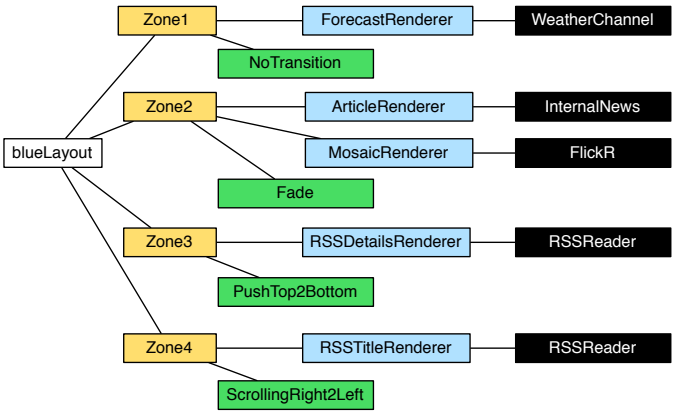


Fig. 3. Composition of artifacts modeling the DSS of Figure 1

any sense, neither does a renderer without a connected source. Figure 4 depicts a metamodel representing a DSS system. We can see that the model given in Figure 3 conforms to this metamodel.

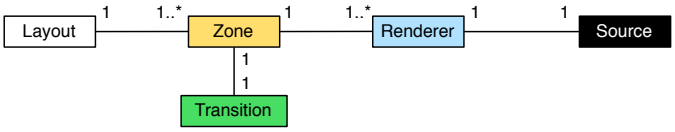


Fig. 4. Domain metamodel for a DSS

Then each instantiated concept presented in Figure 3 is a feature model configuration. As feature models are a form of *de facto* standard to capture variability among family of products [8], [9], we decided to use them to represent the variability of our different concepts.

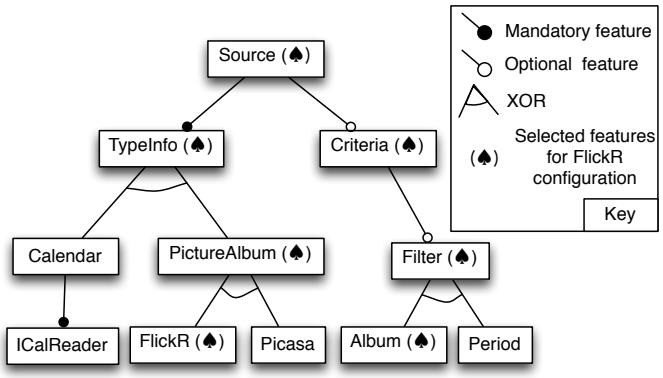


Fig. 5. Excerpt of a feature model for Sources

The Figure 5 shows an excerpt of a feature model to represent sources. The real feature model contains more than 80 features (see table II) and this number is still growing as the ecosystem evolves. We can see that the configuration for *FlickR* represented in Figure 3 is in fact a set of selected features inside this feature model.

In a situation similar to the work of Rosenmuller *et al* [10], we need to deal with multiple variants of one SPL linked by association instances to variants of other SPLs. Then the metamodel presented in figure 4 is not only used to define how configurations should be composed, but also to represent which are the different SPLs and how they are constrained with each other. Actually the associations contained in the metamodel are used to instantiate links between configurations, but they also support constraints expressed between the configurations sets of feature models. For example, a constraint is used to express that a specific renderer must be connected to a source providing an information type it can manipulate. Consequently choices made inside a feature model will impact the other connected feature models following both the associations given in the metamodel and the semantic of the constraints.

Finally, we cannot assume that a product owner will master the global knowledge of the MSPL. Even if the architecture of the MSPL come from a domain engineering work, end-users need to be guided to compose a DSS conforming to the meta-model. Moreover, as stated by Holl *et al*: "*User guidance focuses on easing the modeling and configuration process for users in an MPL setting. People working on different product lines in an MPL need to be made aware of the impacts of their changes on other product lines and vice versa. Such impacts need to be propagated for instance through dependencies between the product lines.*" [5]. Then we must provide an appropriate tool to help users making their choice of configurations, but also creating links between these configurations.

This environment has been implemented using a model-driven approach. A domain metamodel captures the relationships between SPLs and supports the process of a composite configuration derivation. The tool exploits both the metamodel and the constraint mechanism described above in order to only propose valid choices to the end-user regarding her previous ones, and to guide her making links. Moreover the tool itself is completely domain independent, the domain metamodel being a simple input.

This environment also provides a graphical interface dedicated to the product owner. The interface is adapted to the context of DSS exploiting feature models annotated with visual information as icons and explanations, so to improve the guidance during the derivation process.

### B. Separation of Concerns: a Set of Software Product Lines

As stated in section II-B, we have to deal with external contributors who develop parts of the software "semi-independently" [11].

They must be able to add a new contribution without mastering all the knowledge of the final product. For example, a contributor who wants to add a connection to a specific picture sharing service should not need to know how to effectively display pictures in the player, but only how picture data must be formatted to be used in the DSS.

We reach this independence using a dedicated SPL with its own feature model for each concept of the DSS domain as shown in figure 2. Then the usual SPL tools and practices are used independently by different teams of external developers, each of them dealing with a specific part of the domain knowledge. More precisely, each SPL contains *(i)* a feature model, *(ii)* a repository of assets and *(iii)* a generation tool in order to build or retrieve the SPL product.

To ease the growth of feature models by external developers, we decided to use feature models as libraries of available

products. When developers build new products in one of the SPLs, they give the assets and a representation of this new product as a feature model containing all its features as mandatory: this feature model then contains a unique configuration. A feature model representing the available products of the SPL is then built automatically by merging all feature models of each product [12], [13]. This feature model building operation also allows to create a mapping file linking each feature model configuration to the right asset code.

We used this tooled method to develop our software ecosystem, keeping trace of changes by putting codes, feature models and mapping file in a version control system. We get more than 500 commits in six months between May 2013 and December 2013, all SPLs combined. We only had two active internal developers from the beginning of this period until September. Since September external developers contribute to the ecosystem on their own. The number of active external developers evolved a lot during this period: we started with 4 active external developers, had a pick with 10 contributors during the summer, and finished the year with only two. Almost half of the commits (44%) have been made by external developers during this period, and we can see in Figure 6 that they are almost as much involved as internal developers to make changes in different parts of the SPLs.

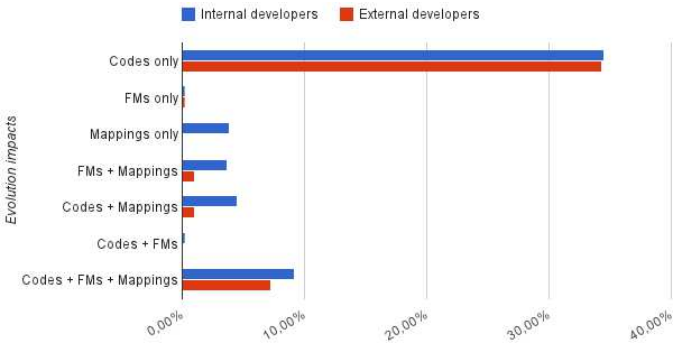


Fig. 6. Evolution impacts by actors

This figure shows the number of commits for each impacted parts of the SPLs, in percentage of the total number of commits. We observe that internal and external developers have done almost the same number of commits impacting only the code and impacting the triplet composed by codes, feature models and mappings between code and feature models. These evolutions are indeed supported by our tools, allowing external developers to do them quite easily. In particular, evolutions impacting the whole SPL (i.e., the triplet code, mapping, feature model) are completely transparent for the contributors who only commit pieces of code and forms. However, some maintenance changes impacting only the mappings (for example) are exclusively done by internal developers as they need to master the knowledge of the whole MSPL.

This separation of concerns driven by the domain allows external contributors to develop new specialized products with less coordination with developments in other SPLs. As shown on Figure 7, more than half of the commits impacted a single domain element and only 10% of the commits impacted the whole MSPL.

To achieve that, developers also need to be able to test their products. In our MSPL organization, automated tests can be

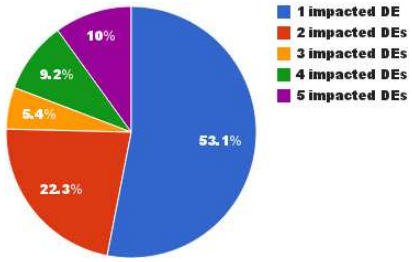


Fig. 7. Number of impacted DEs by commits

provided for each SPL, using sometimes knowledge of other SPLs to ensure that the new products meet the integration requirements. For example, a new renderer displaying picture albums must consume information following a provided schema that describes how these albums are represented inside the system. However, these tests do not necessarily ensure that there is a global DSS product that can integrate this new product: some other constraints could forbid this product.

TABLE II. CONFIGURATION NUMBERS FOR EACH CONCEPTS

| Concept | # Configurations | # Features |
|---|---|---|
| Sources | 68 | 81 |
| Renderers | 74 | 76 |
| Transitions | 15 | 33 |
| Zones | 27 | 49 |
| Layouts | 13 | 51 |

We show in Table II the number of configurations and features for the feature models of each concept of the MSPL. If the number of configurations in each SPL is not very high (but still growing), the number of combinations between products is really important considering that each DSS can use several sources, renderers, transitions and zones. As an example, if we only consider a DSS with three zones, the total number of possible combinations is the order of $10^{19}$. However, this number does not take into account all the constraints between feature models which greatly decreases it in practice.

Next internal developers must be able to manage the MSPL as a whole despite this fast evolution and the large amount of possible combinations.

### C. MSPL: a SPL Composed of Interrelated SPLs

As they master the knowledge of the whole MSPL, internal developers have to maintain it and to ensure its consistency. However the fast evolution of the ecosystem encouraged us to develop automated tools for this purpose.

First we consider that the generation process to obtain a final DSS product must be completely automatized. A DSS is not only an assembly of all products realized in the different SPLs, and the work to integrate all these elements in a final and packaged product is not trivial. This involves to master the knowledge of the MSPL as a whole and to use assets coming from different SPLs. For example the customizer (cf. section II-B) is built using assets coming from all the SPLs to create a web interface reflecting the product owner choices: if a source is used twice or more in the DSS, in different zones, or if it uses different renderers, the manager must be able to distinguish them. That is why this component can only be

created from the different assets coming from all independent SPLs, as shown on Figure 2. For example, the signature of a specific source, specifying what are the parameters of its web service, is retrieved from specific assets of the dedicated SPL, but only used in the transformation process of the MSPL: the SPL itself does not need these assets to create its own products.

The generation process does not only consist in assembling SPL product from chose associations, it also implies combining SPL assets to make complex components, like the customizer, or to use them as glue code, for example inside the player. In that sense, the generation process of our MSPL is *more than the sum of the products of each SPL.*

To facilitate the addition of new products by external developers, we also consider that adding a new product in one of the SPL must be completely transparent to them. To make directly available the new functionality to product owners, we thus have to ensure, in an automated way, that the whole MSPL remains consistent through its evolution.

The first step consists in ensuring the consistency of each SPL independently. As we explained before, this step is realized using some automated tests on each SPL and by building automatically the feature model. Then the consistency of the MSPL as a whole is obtained by ensuring that for every possible SPL product, a final DSS including this product is reachable. In other words, we ensure that our MSPL does not contain any dead product. The very high number of possible combinations including a specific product does not allow to build each DSS individually. We thus decided to check whether composite configurations including a specific configuration conform to the metamodel and respect the feature models constraints. This checking process is automatized by using an incremental algorithm. We assume that a MSPL containing only one configuration for each SPL is consistent, checking it manually. Then for each new product added, we are able to compute with which other SPL products it can be associated and if it exists a final combination without inconsistencies regarding the expressed constraints. Thus if one valid combination can be found, the product is reachable. Assuming that tests conducted in each SPL are sufficient to ensure the integration of a product, we can then ensure the consistency of the derivation process

Consequently, using the MSPL infrastructure, internal developers have only been involved in the MSPL evolution for minor maintenance tasks, like fixing platform bugs.

## IV. Related Work and Discussions

We discuss in this section the different choices and observations made in our case study, relating them to the literature.

### A. Centralized vs. Decentralized Organizations

In our view, a key idea when realizing a MSPL to support an ecosystem is to maintain the organization of the ecosystem. The term "*ecosystem*" comes from the biology field and as stated by Mens *et al*, analogies can be made between software and natural ecosystems. One of them concerns the evolution capabilities of such ecosystems: "*the resilience of a software ecosystem then refers to its ability to return to a stable equilibrium after minor or major disturbances*" [7]. The resilience is then one of the great strength of the ecosystem and we believe it comes, in part, from its decentralized organization.

Nakakoji *et al* define a taxonomy of open-source projects that can also be applied to software ecosystems [14]. Applying it in our context, it appears that the project has evolved from an *exploration-oriented* project to a *utility-oriented* project two years ago and a *service-oriented* project at present. At the beginning of the project, its organization was completely centralized: only one leader controlled the project which was pretty small and subject to many changes to respond to specific needs. We tried to set up appropriate process and architecture to meet them. This exploration-oriented project mutated successfully into a utility-oriented project when we stabilized the architecture and involved multiple external developers to satisfy user needs. It was the birth of the ecosystem, leveraging a decentralized organization involving many different users with various roles. Finally, in order to involve more the passive users in the ecosystem and to let them build their own product, the project moved to a service-oriented project supported by the MSPL approach.

This move to a MSPL allows us to shift from a traditional value chain to a networked organization, as expressed by Hanssen [2]. Through the domain modeling and dedicated tools that simplify line evolutions, we observer that the MSPL reinforces the involvement of the external developer community inside the development process. End-users participate indirectly to the development, for example by suggesting to add new web sources. This network is currently expanding with the integration of user interface ergonomists. The project organisation itself is architecture-centric, and based on a domain engineering unit model (one *product engineering unit* per SPL) [15]. Contributors being involved at different times and often geographically distributed, this organization has several advantages such as centering communication around the MSPL internal developers. When conflicting requirements arise from different SPLs (*e.g.* adding a new behavior to control translations has impacted the way to define renderers), the internal developers allow product engineering units to, temporarily, create their own versions of shared assets, in line with Bosch vision [15]. The propagation of changes is then reduced to the few impacted SPLs.

Finally, Bosch also emphasized the need to use a compositional approach in SPL to improve their management [16]. He explicitly encouraged a decentralized approach to manage large-scale SPLs and proposed "*to move any remaining coordination needs from the process level to the architecture*". From our experience in Yourcast, this is largely facilitated by our MSPL approach, with the MSPL domain metamodel and especially its associations. These associations are the keystones of our architecture as they are used both to guide users and to generate the systems [17]. They are also really important to understand how the teams communicate with each other.

### B. Domain-driven MSPL

The domain metamodel together with the associations and constraints are the backbone of our approach. Considering associations as first class entities, the traceability among domain design and code is also guaranteed. At the feature model level, the associations model interactions among feature models; at configuration level, they model links between configurations; at code level, they drive the code generation. To some extent, we are close to the definition of connectors used by Dial *et al* in their work on transforming an SPL architecture to

Aspect-J code [18]. Nevertheless we do not use an approach based on aspects for the generation, especially as we do not only target a composition of components. Contrary to this work, a generalization of the code generation process is not an objective in our work.

In the DSS application domain, model elements must be configured individually [19]. Modeling and configuration steps are not strictly separated: deciding how many information sources will be involved in a DSS before configuring them is not a natural approach. Rosenmuller *et al* use a model in order to compose instances coming from MSPL [10], [20]. Their model supports constraints between SPLs and can express associations between software artifacts, but it does not allow an end-user to decide by herself how to compose the chosen software artifacts coming from the different SPLs. In that sense, our approach is more flexible and can be viewed as an extension of this work.

In the current implementation of our approach only constraints between feature models are supported. Thus inlike work of Arboleda *et al* [19], we are not able to express constraints inside the metamodel, e.g, stating that only three picture album sources are allowed in a DSS. Nevertheless, we see no technical empdiments to integrating similar functionalities.

Besides our approach is driven by a case study in which only one user at a time is involved in the derivation process. The generation process uses model transformations from a final consistent composite configuration to obtain a product. It does not address multiple technological platforms. As a result we do not face issues related to code weaving or multi-staged configurations [19] or [21].

Finally one of our main requirements is to provide a configuration tool to an end-user to let her create her own DSS. As stated by Holl *et al* the tool support for product derivation in the context of a MSPL is still an emerging field to investigate [5]. For this purpose we partially reuse some works done in the domain of configuration interface generation from feature models [22].

### C. Ecosystem-driven Evolution

To support the evolution of feature models and software assets in the MSPL, we need to provide tools that ensure their consistency. However, many changes may occur according to user roles and development steps. To identify recurrent evolution schemes, we used an empirical study done by Lakhal *et al* about evolutions of UML Profiles [23]. We carried out this study analysing commits and state of the art. Lot of works have already been done in the field of SPL evolution. Here we try to tame the ecosystem evolution using our MSPL organization.

Botterweck and Pleuss propose a taxonomy for SPL evolution strategies, which is based on the impacted level and the triggers of evolutions [24]. Using this taxonomy, we have faced, as of now, the four following situations in our case study:

- *Proactive evolution* (rare): this concerns all planned evolutions of the SPL. In our case, this kind of evolution is only realized by internal developers in order to prepare the MSPL at all levels, when features impacting the whole product are needed. Until now, this evolution occurs only once at the beginning of the MSPL development, in order to manage a technological change. No dedicated tool were thus defined to support this kind of evolution.
- *Reactive evolution* (common): this concerns evolutions conducted by the external developers answering a passive user need. In our case, most of them are anticipated and automated, allowing us to check the consistency of the MSPL when a new product is added to one of the SPLs. Tools such as testing environments were developed to support them.
- *branch-and-unite* (rare): it happens when a user need impacts many SPLs at the same time, as well as the generation process. Then internal developers can make a specific branch using existing products to make an experimental prototype before merging the changes inside the MSPL. In the YourCast case, this happened at least twice, when we moved from a technology to another in the generation process. It is also currently happening as we are extending our DSS to add interactions. No tool has been created to support them.
- *Maintenance evolution* (frequent): in our case study, it exists different kinds of maintenance operations *e.g.*, on the SPL assets, on the MSPL constraints or on the feature models. They directly depend on the user role. SPL asset maintenance evolutions correspond to evolutions for fixing bugs, for example, inside existing assets. Specific tools have been created in order to allow this kind of maintenance for external developers. Modifications of MSPL constraints are critical for the MSPL organization as it impacts its consistency. Only the internal developers are allowed to do them and we are currently developing tools to support them. Finally, vocabulary alignment on feature models are authorized both for internal and external developers, and dedicated tools have been developed to support it.

Another point of view concerns the responsibility levels described by Bosch [15]. They are also driven by the MSPL architecture. The internal developers are responsible for shared architecture elements such as metamodel evolutions, mapping or global process transformations. On their side, external developers implement the reusable assets for each SPL. For the time being we consider that each developer is responsible for products that she added into a SPL. Thus a person cannot modify or delete a product created by another (this includes adding new features). However, we note that leaders emerge and that other responsibility levels could be needed, as distributing responsibilities in an hierarchical manner.

The main issue here is to be able to ensure the MSPL consistency at any time, even if the ecosystem makes it evolve very rapidly. We manage our MSPL evolution using a framework which takes into account both the user roles and some defined types of evolution [25]. However as Botterweck states about evolution in systems-of-systems: "*it is impossible to introduce changes without causing inconsistencies at least temporarily. Hence many changes can only happen incrementally and changes are propagated through an introduction and subsequent resolution of inconsistencies*" [26]. Better handling different kinds of evolution while balancing inconsistent

changes and consistency checking is part of our future research plan.

## V. Conclusion

Software product lines and software ecosystems are infrastructures that are targeting reduction of complexity in development and maintenance of very large software systems. In this paper, we have reported on the usage of a multiple software product line (MSPL), defined as a set of interrelated SPLs organized around a domain model, to manage the YourCast digital signage software (DSS) ecosystem. Promoting separation of concerns, it enables us to leverage the still needed customization capabilities of SPL techniques so that each DSS can be easily configured by final users in a consistent way.

The YourCast system evolved from a framework applying decoupling principles to a medium-scale ecosystem organized around a MSPL. We discussed the different stages of the system evolution that naturally led to a common software platform developed and maintained by internal and external contributors, *i.e.*, a software ecosystem. The MSPL thus enabled us to automatically build valid composite configurations for a YourCast deployment, but also to drive evolution of the ecosystem while preserving its decentralized organization.

We expect this case study report to be valuable for researchers handling either software product lines or software ecosystems. As future work, we plan to extend the YourCast MSPL with user interaction features and ergonomic properties, to evaluate further the practicality of the underlying MSPL approach and to improve support for evolution management. We hope these insights can contribute to a methodology that guide practitioners in managing similar software ecosystems.

## Acknowledgment

The work reported in this paper is partly funded by the ANR YourCast project under contract ANR-2011-EMMA-013-01.

## References

[1] J. Bosch, "From software product lines to software ecosystems," *Proceedings of the 13th International Software Product Line Conference*, pp. 111–119, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1753235.1753251

[2] G. K. Hanssen, "Opening up software product line engineering," in *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*. ACM, 2010, pp. 1–7.

[3] P. Clements and L. M. Northrop, *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0201703327

[4] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[5] G. Holl, P. Grünbacher, and R. Rabiser, "A systematic review and an expert survey on capabilities supporting multi product lines," *Information and Software Technology*, vol. 54, no. 8, pp. 828–852, Aug. 2012. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S095058491200033X

[6] K. Kelsen, *Unleashing the power of digital signage: content strategies for the 5th screen*. Taylor & Francis, 2010.

[7] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, "Studying evolving software ecosystems based on ecological models," in *Evolving Software Systems*. Springer, 2014, pp. 297–326.

[8] S. Apel and C. Kästner, "An overview of feature-oriented software development," *Journal of Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, July/August 2009. [Online]. Available: http://www.jot.fm/issues/issue_2009_07/column5/index.html

[9] D. Benavides, S. Segura, and A. Ruiz-Corts, "Automated Analysis of Feature Models 20 years Later: a Literature Review," *Information Systems, Elsevier*, 2010.

[10] M. Rosenmüller, N. Siegmund, C. Kästner, and S. S. Ur Rahman, "Modeling dependent software product lines," in *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, 2008, pp. 13–18. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.147.2830\&rep=rep1\&type=pdf

[11] H. Brummermann, M. Platz, and K. Schmid, "Formalizing Distributed Evolution of Variability in Information System Ecosystems," in *Vamos*, ser. VaMoS '12. ACM, 2012, pp. 11–19. [Online]. Available: http://doi.acm.org/10.1145/2110147.2110149

[12] M. Acher, P. Collet, P. Lahire, and R. France, "Composing feature models," in *Software Language Engineering*. Springer, 2010, pp. 62–81.

[13] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire, "On extracting feature models from product descriptions," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, ser. VaMoS '12. New York, NY, USA: ACM, 2012, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/2110147.2110153

[14] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Proceedings of the international workshop on Principles of software evolution*. ACM, 2002, pp. 76–85.

[15] J. Bosch, "Software product lines: organizational alternatives," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 91–100.

[16] ——, "Toward compositional software product lines," *Software, IEEE*, vol. 27, no. 3, pp. 29–34, 2010.

[17] S. Urli, S. Mosser, M. Blay-Fornarino, and P. Collet, "How to exploit domain knowledge in multiple software product lines?" in *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*. IEEE, 2013, pp. 13–16.

[18] J. Diaz, J. Perez, C. Fernandez-Sanchez, and J. Garbajosa, "Model-to-code transformation from product-line architecture models to aspectj," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, Sept 2013, pp. 98–105.

[19] H. Arboleda, A. Kastler, and F.-N. Cedex, "Dealing with Fine-Grained Configurations in Model-Driven SPLs," *Transformation*, pp. 1–10, 2009.

[20] M. Rosenmüller and N. Siegmund, "Automating the configuration of multi software product lines." in *VaMoS*, 2010, pp. 123–130.

[21] L. Tizzei, C. Rubira, and J. Lee, "An aspect-based feature model for architecting component product lines," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, Sept 2012, pp. 85–92.

[22] Q. Boucher, G. Perrouin, and P. Heymans, "Deriving configuration interfaces from feature models: A vision paper," in *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. ACM, 2012, pp. 37–44.

[23] F. Lakhal, H. Dubois, and D. Rieu, "Pattern based methodology for uml profiles evolution management," in *RCIS*, R. Wieringa, S. Nurcan, C. Rolland, and J.-L. Cavarero, Eds. IEEE, 2013, pp. 1–12.

[24] G. Botterweck and A. Pleuss, "Evolution of software product lines," in *Evolving Software Systems*. Springer, 2014, pp. 265–295.

[25] D. Romero, S. Urli, C. Quinton, M. Blay-Fornarino, P. Collet, L. Duchien, and S. Mosser, "Splemma: a generic framework for controlled-evolution of software product lines," in *Proceedings of the 17th International Software Product Line Conference co-located workshops*. ACM, 2013, pp. 59–66.

[26] G. Botterweck, "Variability and evolution in systems of systems," *EPTCS*, vol. 133, pp. 8–23.