Software Product Line Evolution: The Selecta System

Jacky Estublier
Grenoble University. LIG.
220 rue de la Chimie
F-38041 Grenoble, France
+33 476 63 55 64
Jacky.Estublier@imag.fr

Idrissa A. Dieng
Grenoble University. LIG.
220 rue de la Chimie
F-38041 Grenoble, France
+33 476 63 55 63
Idrissa.Dieng@imag.fr

Thomas Leveque
Grenoble University. LIG.
220 rue de la Chimie
F-38041 Grenoble, France
+33 476 63 55 64
Thomas.Leveque@imag.fr

ABSTRACT
The current technology gives little room for the different kinds of evolution needed for any software product line (SPL): evolution of the associated engineering environment, evolution of the market and SPL scope, evolution of the products and variability. The paper describes how these different evolution needs are addressed in the CADSE and Selecta systems. The solution we propose uses metamodeling and generation for the engineering environment evolution, composition for scope and market evolution, a component database and a selection language for the product and variability evolution. The paper presents the Selecta system and shortly discusses the experience.

Categories and Subject Descriptors
D.2.6 Programming Environments, D.2.11 Software Architectures, D.2.13 Reusable Software, H.1 MODELS AND PRINCIPLES, J.6 COMPUTER-AIDED ENGINEERING.

General Terms
Experimentation, Design, Languages.

Keywords
Product Lines, Product families, Evolution, IDE, Software Environments.

1. INTRODUCTION
Product Lines (PL) are in daily use and have shown their potential [1]. Product lines, despite, or because, they have originally been designed and developed to solve very pragmatic and engineering issues, most often do not use systematic engineering principles like encapsulation, abstraction levels, and evolution control. Many issues are hampering the wide adoption of the PL approach in all software engineering fields. Among these issues, our work focuses on reducing the cost of a SPL engineering environment and on the support of the different kinds of SPL evolution.

Indeed, evolution is a fundamental and universal principle that has impacts on all aspects of a SPL: The market, competition and needs evolve, the techniques, libraries, architectures evolve, the PL scope and family evolve, the existing family members evolve, the PL engineering environment evolves and so on.

These evolution needs are widely acknowledged [11][12] but they are still poorly supported in the current SPL systems. A major reason is that in the current approaches, the associated engineering environment is developed in an ad-hoc way, which makes it very expensive. Most changes require changing the associated environment which explains why evolution is so problematic to control.

Other reasons explain the evolution control difficulties: in the current approaches, the domain engineering work produces a number of artifacts like a generic architecture, a feature model, generators and libraries. Surprisingly, most of these artifacts are monolithic: they are not hierarchically organized, they cannot be composed or extended (at least easily enough), and they do not use abstraction levels, encapsulation and so on.

Based on our original expertise and long experience with an automatic selection system over a component repository and a rich system model [6], we have identified the following shortcomings. Using a component repository and a rich system model is a very powerful approach but it requires a reliable and always up to date repository and system model, which is not the case in general because the information has to be manually provided, which is too expensive and error prone in practice. The issue is therefore to automatically gather this information. To do so we have developed the CADSE (Computer Aided Domain Specific Environment) technology [8]. In a CADSE, the system model is the interface to perform most software engineering activities. For example, creating a component or a dependency relationship between the components is performed by creating the corresponding model element, or creating a link between the model elements. The system model is therefore a causally related abstraction of the reality. In practice, it means that the solution comes from developing a dedicated PLE (Product Line Environment) from which only PL members can be developed.

It is more or less accepted that the technology evolves pretty fast, it is less accepted that the domain evolves too [11]. The next issue is to be capable to build such a PLE (that we call a CADSE) at a low cost, and to be able to make it evolve easily enough. To do so, we have reused a PL approach for the environment itself: a PLE (a CADSE) is a family which members are the various (successive or simultaneous) PLEs each one being the PLE in use at a given point in time.

From that point, we have developed a technology in which it is possible to compose existing CADSE, to consider a PLE as a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLEASE'10, May 2, 2010, Cape Town, South Africa.
Copyright © 2010 ACM 978-1-60558-968-8/10/05 … $10.00.

family in which each member is either specialized for an activity in the PLE, or specialized to support a new feature or a new part of the domain. This technology is used to support scope and market evolution.

The central part of the approach however is related to the selection mechanism: the Selecta system. The approach borrows from the Adele selection mechanism the idea that, with a rich system model, it is possible to automatically build a product based on its desired characteristics (features and others). The approach has been deeply revised. Different kinds of constraints and a selection language have been defined, and an encapsulation mechanism allows defining nested sub-families (or PL fragments).

In Selecta, the metamodel (the domain model) is not predefined; instead we have developed the group concept that identifies the points where a selection should occur, with the explicit definition of the attributes on which the selection should be performed. Groups are types and as such they are part of the domain model. Group can be created at any time and their definition can be changed at any time, making very easy to change and extend the PL domain and scope.

The paper only sketches, in chapter 2 the two first points: the CADSE technology (how to build easily a PLE), and the composition mechanism (how to extend a PLE scope) since these technology have been previously published. We detail a bit more the Selecta system in chapter 3. The way features, scope and hierarchies are managed in Selecta is presented in chapter 4, and versioning in chapter 5. Chapter 6 gives a few hints on the implementation and experience. Chapter 7 concludes the paper.

2. PRODUCT LINE EVOLUTION

A SPL evolves along different lines: evolution of the software engineering environment, evolution of SPL scope and evolution of the products and their variability. In this section, we briefly give an overview and a description of these kinds of evolution.

2.1 Engineering Environment Evolution

What makes product lines effective is the integration, in the engineering environment, of business and engineering knowledge. A major limitation in the adoption of product lines is the very high cost of the associated engineering environment. This cost comes from identifying and making explicit the relevant knowledge, and in programming the environment (automation, guidance, wizards and so on). Today both activities are to a large extent performed in an ad-hoc way, which makes them expensive and difficult to change the environment after it is in production..

To address these difficulties, we have developed the CADSE technology [8] in which both the business and engineering knowledge are formalized through models or metamodels. The business knowledge is formalized through a metamodel that describes the concepts, their relationships and behaviors in very much the same way as many metamodeling environments (MOF, Ecore, Kermeta etc). The major part of the knowledge is related to the technology, environment, process and conventions to be used when developing a product family member. For each one of these aspects, we have defined a metamodel, so that it is enough to write a model (conform to the associated metamodel) to formalize the aspect (see Figure 1).

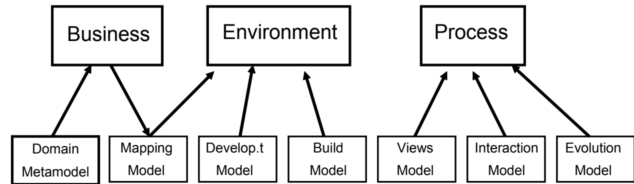

Figure 1. Models and metamodel defining a PL environment.

From these models and metamodel, a CADSE Generator generates the associated engineering environment dedicated to the production of family members. The approach simplifies the definition of the product line and its environment because most of the relevant metamodels (environment, mapping and process) have been already identified and instrumented (editors, libraries), and simplifies significantly the development process since most of the code is generated. It also dramatically simplifies the identification and the realization of the evolution control, since only a few models are to be changed, the new extended environment is generated, and the existing objects are automatically updated or migrated.

This work has been published in [6] [8] and will not be discussed further here.

2.2 Product Line Scope Evolution

The more knowledge is integrated in the PLE, the more effective is the PL, but the PL becomes very narrow. Indeed a standard environment or a platform (e.g. Eclipse) can be used for almost any PL, but not having any business knowledge, the help provided is limited (see Figure 3). Conversely, in a very narrow domain it is possible to statically define the various possibilities and to generate the code (e.g. a wizard). The approach we propose is therefore to define narrow and powerful software environments (CADSEs), and to compose them to define wide scope environments. To that purpose we have developed a number of technologies allowing combining a number of CADSEs:

Horizontal composition, which takes two CADSEs in input, and compose their metamodels (defining metalinks between two concepts pertaining to the two metamodels (See Figure 2), and to compose models (defining links between model elements).

Vertical composition, which relates and synchronizes instances pertaining to the two CADSEs in an abstraction/specialization relationship.

Extension, which adds attributes, links and/or behavior to the concepts already defined in a given CADSE (in an Aspect oriented way).

Group, which relates two concept instances in a materialization or versioning relationship (see Figure 2).

The middle part of Figure 2, contains a SOA (Service-Oriented Approach) metamodel that defines the concepts and relationships of SOA. The core SOA metamodel explicitly defines the concept of service. A service can be a *specification* (specifying the provided functionalities and its properties), an *implementation* (represents a code snippet which is said to provide one or more specification(s)) or an *instance* (a run time entity corresponding to the execution of an implementation). The upper part of the Figure 2 describes an extension of our SOA metamodel which explicitly defines the concept of software application as a composite entity

which can contain (or use) services (specifications, implementations or instances) or which can be defined by its goal i.e. by its characteristics (properties and constraints) [5]. The domain model is defined in a SOA/Business CADSE.

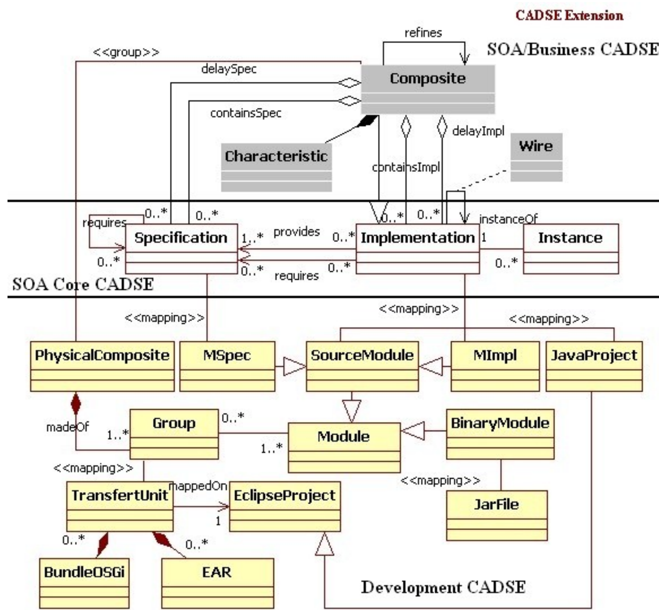

Figure 2. An example of CADSEs composition.

The lower part of the Figure 2 defines the concepts (like *Module*) which correspond to services during the development and packaging phases. A Module can be source module (e.g. a Java project) or binary module (e.g. a library or a Jar File). This part corresponds to the Eclipse concepts relevant when developing and packaging software in the Eclipse platform. These concepts are defined in an Eclipse CADSE. Services are mapped to generic development concepts like *MImpl* or *MSpec* which are *Modules*. A specific model, the Eclipse mapping, relates the domain metamodel with the Eclipse metamodel, in order to establish association, when convenient, between a business concept and one or more Eclipse concepts (project, directory, source file, and so on). The mapping is used to synchronize, both ways, the creation/evolution of a business concept instance with the creation/ evolution of its associated Eclipse artifact(s). We use the CADSE composition and extension mechanisms for providing a sophisticated environment which is specialized to build software applications.

2.3 Product and Variability evolution

In [2] the different approaches for product lines with respect to scope and reuse are the following: ***Standard infrastructures***, which are generic and without business knowledge (e.g. Eclipse); ***Platforms***, with a library of common elements, ***Product lines***, based on generic architecture and variability, and ***Configurable product family***, based on a library of components and business knowledge.

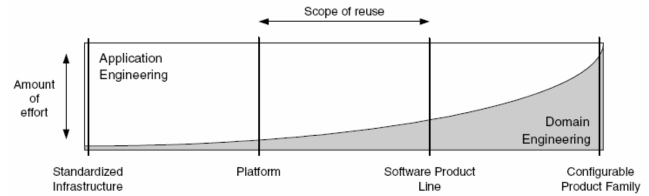

Figure 3. Scope of reuse [2].

Our work falls in the configurable product family category. The issues, in this category, are (1) the definition of the database which contains elements (components or services), (2) the way business and technical knowledge is defined, and (3) the techniques and tools for building a product, selecting from the database the components that together will constitute the desired family member.

The following sections present the product and variability management in the Selecta System.

3. THE SELECTA SYSTEM

The objective of the Selecta system is to take as input an abstract family member description, and to find out, in the component database, the components that together will constitute an implementation of that family member. To do so Selecta relies on a rich data model and on the group concept.

3.1 Data model and groups

In the Selecta approach [5], the database contains products designed and developed following the PLE (the generated CADSE). The entities found in the database are therefore instances of the business model (instance of the concepts defined in the business metamodel), and their associated Eclipse mappings (most often, in our case, the source code and associated packaged modules).

The Selecta approach heavily relies on the group concept. For the sake of this paper, let us assume a generic metamodel made of the *specification* and *implementation* concepts (see Figure 2). Following this metamodel, let us take the example of two implementations of a *SpellChecker* service: the *EnglishSC* and *FrenchSC* implementations. They are both implementations of the *SpellChecker* service because they both provide the same *SpellChecker* interface; they both need a dictionary and have a "language" attribute, but they differ in some aspects (see Figure 4).

An equivalence group (group for short) is made of an object (called the head) and a number of group members. A solved member (member for short) is the union of a group member and its head. Said differently, the head holds the common properties of the group, and the group members hold only the specific properties. Therefore our example can be represented by the *SpellChecker* group. The *SpellChecker* group has as head the object *SpellChecker* (the head is named after the group) which holds the common values. *SpellChecker* is a real object and as such it has a type. In our example, *SpellChecker* is a Specification and the members are Implementations. A major contribution of the group concept is that the group head not only holds the common properties but also defines the properties by which the members can be distinguished (and selected). In our example, the object *SpellChecker* defines the attribute "language: String" which

means that each *SpellChecker* member must define a value for that attribute: the group head is both an object and the type of its members.

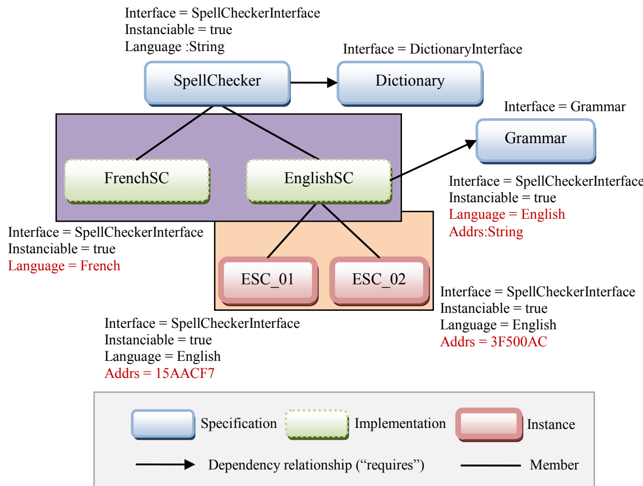

Figure 4. An example of groups

In Figure 4, *Dictionary* is also a group; and the dependency between *SpellChecker* and *Dictionary* being defined between two types is a relation type (of cardinality 1) which means that each *SpellChecker* member must have a dependency toward a *Dictionary* member. We define a group type as the tuple <HeadType, MemberType>; in our SOC system we have defined two such group types <Specification, Implementation> and <Implementation, Instance>. In the example *EnglishSC* is both a member of the *SpellChecker* group and the head of group *EnglishSC* with *ESC*_01 and *ESC_02* members. As a group head, *EnglishSC* defines the attribute "addrs:int" which explains why its members have values for that attribute.

A group is *instantiable* if it is associated with a factory. Whenever a resolution is required for an instantiable group, and no member fits the needs, the associated factory is called with the selection expression as parameter, and the factory creates (instantiates) the missing member, if possible. By default, *implementation* groups are instanciables (it is the usual class instantiation), but specification groups can also be instanciables, if the factory knows how to build an implementation that fits the selection constraint.

3.2 An example scenario

Suppose we are developing the *MyTextEditor* member of a *TextEditor* group. To test it we write the following application model:

```
Config MyEditorTest {
  Init TextEditor/MyTextEditor;
 Optional Printing(&(type=laser)(speed>=8ppm));
 Select SpellChecker(language=English);
 Delay SpellChecker;
}
```

In this example, the first line expresses that the *MyEditorTest* application starts by executing (an instance of) the *MyTextEditor* member. This is an example of the traditional static and deterministic way of expressing architectures. The second line expresses that a laser printer with a speed greater then 8ppm should preferably be used (any other one or none if not available). Third line says that we need an English *SpellChecker*. Fourth line indicates that the *SpellChecked* selection must be done later.

The application model is specified using our constraint language [5] briefly introduced bellow (3.3).

3.3 The constraint language

We have defined a selection and constraints language that may look like OCL, since it allows navigating the links and evaluating logical expressions on the attributes and links found in the database. Unlike OCL, our expressions can be associated with any individual objects, either types of instances, and act on types as well as on instances, since group heads are types and they are declared dynamically. But the main difference lies in the purpose of the language. Like OCL, constraints may be used to enforce internal database consistency but the main purpose of the language is to record and enforce the compatibility constraints that often exist between objects. For example the *FrenchSC* implementation may indicate:

```
Select Dictionary(language = French);
```

meaning that any application containing *FrenchSC* should also contain a French dictionary. Constraints can be more general, like

```
Select  Dictionnary(language  =  self.language);
```

which means that whenever a *SpellChecker* implementation is selected, must be also be selected a *Dictionary* for that implementation language: it factorizes constraints at group level.

Note that since group heads are types, the constraints interpreter knows that "*language : String*" is an attribute declared for the *Dictionary* group. Constraints are strongly typed despite the fact that types can be created dynamically.

An implementation that executes only on Linux should not be associated with an implementation that does not run on Linux. It can be expressed as follows:

```
Select Implementation (Not  ((Self.
attributeDef.name="OS")and(Self.OS!= "Linux"));
```

Which means that, if attribute *OS* is defined in a group, then later on must be selected only implementations runing on Linux.

An automatic composition may fail because it has not been possible to resolve a group, either because no member satisfies the constraints (the application will be *incomplete*) or because some constraints are in conflict (the application will be *inconsistent*). For that reason, the automatic selection manager can be executed in "backtrack" mode, with means that if a resolution fails, the manager undoes previous selections and tries another composition. This ensures that if a solution exists, it will be found, but it may be very expensive or impossible in practice for large databases.

Universal and Contextual constraints

The above constraints are called *universal constraints* because they must be satisfied by any application making use of the associated object. They can be set on types, like in OCL as well as on individual objects. It guaranties that any application will be made of components for which no incompatibility is known.

A composite can add other constraints, which purpose is to indicate the intention, properties and features the application must show. The language used is the same, but the constraints and properties are interpreted only in the context of the current application; they are *contextual constraints*.

The application defined above (*MyEditorTest*) simply indicates that an English spell checker is required but of course much more complex definitions are possible.

It is also possible to set some attribute values, and to instantiate relationships that will be visible only in the context of the composite that define them:

```
Set SpellChecker (&(member.EnglishSC.mode=extended)(member.EnglishSC.requires = Grammar));
```

This expression changes the value of attribute *mode* of member *EnglishSC* of group *SpellChecker*, and creates a new relationship *requires* between *EnglishSC* and group *Grammar*.

3.4 Delayed selection

The application manager (tries to) resolve all groups found in the transitive closure of the dependency relationship, ending if successful in a full list of components. An application definition can indicate which resolutions should be left to a later phase in the life cycle. This is indicated by the expression like `Delay SpellChecker;` which means that group *SpellChecker* must not be resolved now. It is the user, and ultimately the execution machine, that turns off the delay property.

This feature is useful in two cases: when resolution must be left to the execution (opportunistic computing), and when a composite is intentionally defined to gather the common structure, properties and constraints shared by a set of applications. It is a template.

3.5 Composite Templates

A composite C_i can be created as a refinement of another composite C, called the template. It means that C_i initial value is set to the current value of C. The new composite C_i can add constraints and remove *Delay* expressions in order to make its own specific resolution for these undelayed groups. If all delays are removed, C_i will be an application that satisfies the C constraints; it can be said that C_i is a member of the C family. If C_i does not remove all the Delay expressions, C_i will be a template that refines the C template; it will be a C sub family.

4. SCOPE AND MODULARIZATION

Software engineering good practice clearly indicates that encapsulation and information hiding are the most powerful means to master complexity and scalability. Therefore in Selecta, a group may have member that are "traditional" components or a group can be a PL of its own, which members are its PL members. A group whether supporting a PL should be seen as atomic from "above", hiding its internal domain model, variability, component database and so on. This section discusses how it is done in Selecta.

4.1 Composite factory

A composite is an implementation (see Figure 2); therefore, for those specifications (like *TextEditor*) for which various composite can be defined, it is possible to set the specification attribute instanciable=true and to associate a factory. That factory is called when a resolution is required for that specification, and no existing implementation satisfies the constraint. The factory creates a new composite with attributes that satisfy the constraint, a selection constraint and optionally a predefined template when common constraints must be set.

If the composite manager succeeds in building that composite, the resolution returns the newly created composite, otherwise the specification is left unresolved. In this way, only the needed composites are built, not all the possible composites (their number can be very large!).

4.2 Composite and encapsulation

The above is insufficient in general; for instance, it is not possible (and not convenient) to give as constraint to the *TextEditor* group expressions like the one in our example (*config MyEditorTest)*, because:

- the selection expression is a simple expression using only the attributes defined in the group, and
- the caller is not supposed to know that the result will be a composite, and
- the caller is not supposed to know the structure, content and attributes of the future composite elements.

A composite must act as a black box; it must appear, for its users, as a "normal" implementation. A composite must act as an abstraction level, which means that the properties it provides (its attributes) result from the correct combination of components of lower abstraction level, therefore holding attributes describing lower level properties.

For example, the *SpellChecker* group can define the attribute `Boolean incremental`, meaning that its implementations can perform (or not) spell checking while the user is typing text. Suppose that implementations of the *SpellChecker* group are only composites that are built from a specific database of components holding specific attributes, like `buffered`, `memsize`, *`delay`* and so on, and that the *SpellChecker incremental* property only comes from a correct composition of lower level components holding these low levels attributes. The same occurs for the *TextEditor* group; its implementation may very well be composites made of components stored in a specific database with specific and low level attributes. But applications that require a text editor do not need to know how a *TextEditor* implementation is built, but only that it holds the required properties. In this case, the *SpellChecker* factory is in charge to transform each selection expression on the *SpellChecker* group in a set of lower level selection expressions on its internal components such that, if successful, the required properties will "emerge" from the selected components composition.

In the current Selecta implementation, this transformation is pretty simple: single attributes values (like "incremental = true") are transformed into a predefined set of composite selections constraints, and a few simple expressions like `(&(A="a")(B>3))` are translated. A more general work on expression transformation will be undertaken.

The factory, in the transformation process, can also generate code and implementations, or even fully generate, based on the selection expressions, the components that fits the needs. This is a "traditional" generative approach. Indeed, the *instanciable* property and the factory mechanism have been originally designed with this purpose in mind.

4.3 Nested visibility, composite hierarchy

The above mechanism allows splitting the database in non overlapping "partitions", each one with its own types, properties,

models and instances. This fits pretty well with the composition approach where each sub-family can be designed and managed as an autonomous and independent PL fragment, and composed dynamically in various larger scope PLs. For example, the *SpellChecker* group can be designed as an independent PL that can be composed with other PLs to form large scopes PLs that may require a spell checker.

Each PL fragment appears, from "above" as a "normal" group (a selection point), and is a PL on its own right, with its domain model, feature model, component database and so on.

This approach has important characteristics in what concern evolution, since each PL fragment can evolve independently without impact on it "users", at least as long as the visible attributes (i.e. the attributes defined in its group head) are not changed. New features and new components can be added, different architectures and constraints can be set without visible impact on larger PL, nor on existing products using this PL fragment.

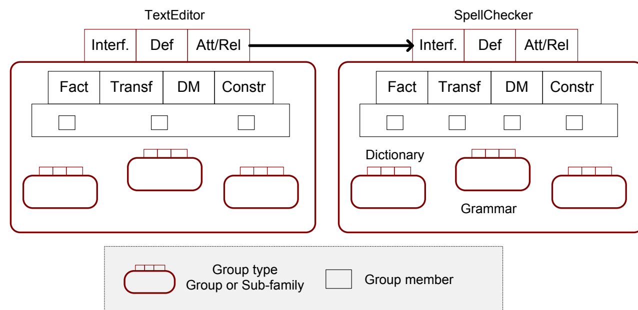

Figure 5. Groups and sub-famillies

In Figure 5, we represent schematically how groups and sub-families are structured. A Group has a visible definition (its type) containing an interface ("Interf."), a number of attributes and relationships ***definition*** ("Def"), and a number of valued attributes and links ("Attr/Rel"). The valued attributes and links are shared by all the group members, and the attribute and relationship definitions are instantiated differently in each member.

Simple groups only contain their members, which are not visible from outside. A PL fragment is a group having composites as members, containing a factory ("Fact"), a transformator ("Transf" from external selection expressions to internal composite definition), a Domain Model ("DM"), and universal and contextual constraints and templates ("Constr").

5. PRODUCT VERSIONING AND EVOLUTION

Most product lines use traditional versioning systems for their versioning and evolution control. It is clearly unsatisfactory, the versioning tools not having been created for PL evolution. In the CADSE system, we have deeply reworked model and artifact versioning, in order to provide more flexibility and reliability what building PL member using (versions of) components.

5.1 Product versioning

In a model based system, as CADSE and Selecta, not only individual (source) files need to be versioned and controlled, but also the different models created for the PLE definition (see Figure 1) and for the PL member definition.

In the CADSE system, evolution and versioning strategies are explicitly defined along with the domain model. These strategies are defined at a very fine grain: each individual attribut, and each individual relationship definition has its own evolution characteristic (*mutable*, *immutable*, *final*), and each relationship destination also has its own evolution characteristics (*mutableDestination*, *immutableDestination*, *finalDestination*, *branch* and *effectiveDestination*) [6]. The destination relationship characteristics allow one to control how a change in an entity (model or metamodel entity) propagates to other entities following relationships.

For example, suppose that we have a dependency relationship between an implementation A.i (entity A revision i) and an interface I.j. It the dependency relationship has characteristic *immutableDestination*, whenever the interface changes (creating I.j+1), a new revision of A must be created; therefore leading to two dependencies (A.i -> I.j and A.i+1 -> I.j+1). If the dependency between A.i and B.j has characteristic *effectiveDestination,* if B changes (creating B.j+1), relationship becomes (A.i -> B {j, j+1}) meaning that both B.j and B.j+1 are valid destinations for that relationship.

Combining the different characteristics on the different attributes and relationships, very fine grained and powerful evolution strategies can be defined. Since metamodels are models, and since the composition technology we use is based on the definition of links between models and between metamodels, the same evolution control mechanisms can be used to control how metamodels evolve (for example adding new concepts, or changing concept definition in the domain metamodel), and how changes in a model propagate to other models.

5.2 Product evolution

We believe that our constraints (contextual and universal), our selection language and composite factory mechanism solve a number of issues already identified in the current PL approaches with respect to evolution.

Feature management. Most users consider feature management as a must for any PLE. Our functional attributes can be used like feature definition, and our expression like feature selection. But attributes are set on entities and the scope for an attribute is the group in which it is defined, while a feature may span over a large set of components. The composite factory approach with attribute transformation is a convenient solution.

- Attributes can really be used as features because the expression transformation can clearly express how to provide the feature using the characteristics of the potential components.
- The CADSE can provide, as part of its interaction model, a dedicated interface for feature selection (using a form for example).

Feature Scope and dependencies. Mentioned very often is the issue of feature scope, since most formalism (FODA [9] and others) do not provide any scoping mechanism. Another frequent problem is the identification of feature dependencies [3]. Combining different features may lead to assemblies in which some components do not fit together. The composite factory

approach, database partitioning and universal constraints is a convenient solution since:

- Feature dependencies can be made explicit either during the expression transformation process or based on the constraints set on each individual component. Any invalid assembly will be detected (by constraint violation), and backtrack mechanism will try to find a valid solution, if existing.
- Feature scope is enforced through the database partitioning capability, allowing scalability and encapsulation.

Product technical dependencies. Difficulties are reported concerning the compatibilities between components, since it is difficult, if not impossible, to test any possible composition. Our system does not make any difference between attributes expressing (functional) features, and those expressing technical characteristics. Similarly, constraints can express either feature dependencies as well as any technical or compatibility problem.

We believe that our mechanism based on universal constraints solve a part of this problem, because it is the component programmer, which knows the low level issues, that is in charge of expressing the universal components constraints. Once set, these constraints are satisfied whatever the product to be built.

Versioning and evolution. The evolution system automatically records version dependencies (if these dependencies are set as *immutableDestination* or *effectiveDestination*). It ensures that any composition will be consistent with respect to version compatibility. For example if A.i pertains to a selection, the system will accept only B.j or B.j+1 to be part of that selection (see section 5).

6. RELATED WORK

Different tools and systems addressed model-based SPLs, however SPL evolution, as well as model/metamodel evolution are not the most popular research areas. We have split the topic into (1) PL environment evolution, (2) scope evolution and (3) product and variability evolution. In each category, two aspects are to be addressed: how to manage the change itself, and scalability issues.

Scalability is traditionally addressed by modularity, hierarchical nesting and information hiding. Some authors have proposed concepts like model fragment [11] or multiple DSL and DSM [13], or model specialization [14], but no one we know assimilate a product line itself as a component with variability, which allows nesting and encapsulation, and no one propose a complete PL environment for each PL fragment.

Domain evolution (domain change and scope evolution) is recognized as an issue but not much addressed in practice; it is because it changes the domain (meta) model, and subsequently it changes most models and the PL environment. Similar to us, [13] uses partial domain models and combine them through references but they seems no to be able to build the corresponding PL environment nor to continue using the "old" models unchanged. In [12] the problem is identified, but metamodel change is only syntactically propagated to changes in corresponding models.

Model evolution is addressed in the MDE community as model transformation (from XSLT to QVT) model diff and merges (SiDiff, EMF Compare, UML diff, DeltaProcess, etc.) and is the subject of an intense literature. Despite these efforts, model evolution is very weakly supported today, which hampers significantly the management of model-based PL evolution. In our case, there is never any transformation not any merge; domain model evolution as well as scope evolution is managed by model composition. Model evolution and versioning has been the topic of a large research effort as presented in section 5.

7. IMPLEMENTATION AND VALIDATION

A number of the ideas and techniques used in the Selecta system have been implemented and tried in the Adele system [7][6] in real exploitation for years (for example for all airbus development for more than 10 years). During this exploitation period we have intensively used an automatic selection process, including advanced automatic selections and universal constrains [5]. We have shown how powerful the approach can be. For example, the largest Airbus A320 configuration was expressed in a few lines of feature expression, but its computing involved thousands of internal attributes and constraints. But we also realized that, out of critical systems, (we also had Bull, Ericsson and many other customers), the cost of maintaining the component database was too high. In practice, this is a major limitation that explains why many advanced technologies and systems like architecture languages, system models, and automatic selection, never had success so far. Indeed, all these systems rely on a model that must be exact at all time. Experience shows that, out of a few niches, the effort to keep manually this information up to date overcomes the expected benefit.

This is one of the reasons why we have developed the CADSE technology used in many research projects. The model, in a CADSE, is the interface by with developers are interacting with the software application. Dependencies are automatically detected and set; attributes and constraints are directly set by the developer itself, because he/she is the only one who knows them, and because they are needed for building and testing the software. The model information is reliable and always up to date; the database, including version compatibility control, is also reliable and up-to-date. We believe this is a prerequisite for any automatic selection mechanism.

The Selecta system is developed with the CADSE technology which is an extension of the Eclipse platform. Based on the CADSE system and its component database, the Selecta system has been designed to overcome the issues identified with the Adele system (which still runs with thousands of seats used daily in production mainly in the Dassault System Company for CATIA development). CADSE and Selecta systems are developed as Eclipse plug-ins, the databases are Subversion databases for the Eclipse artifacts, and a number or relational databases for the model elements (MySQL).

We believe that the CADSE technology and the Selecta system, the Group mechanism, universal and contextual constraints and selection expression transformation do solve a number of the issues identified.

Selecta is fully implemented and running but still in a development and experimental stage.

CADSE is an open source project available at http://cadse.imag.fr

8. CONCLUSION

Since long, we have identified evolution as the major difficulty in practice in Software engineering. Our work on configuration management led us to see product lines, in the 80s, as a triple <component database, versioning mechanism, automatic selection>. The work presented here is an improvement and the current state of our work on the topic. We have added, since the 80s, the engineering environment point of view (CADSE), the model and metamodel approach, and the different CADSE composition techniques for better reuse and evolution, the concept of group with and extended selection and encapsulation mechanism : the Selecta system.

We believe these mechanisms have improved the PL evolution control on the environment, scope and product evolution areas.

Environment evolution. The *model approach* used for defining a CADSE through models, and using specific editors and wizards, makes much easier a PL environment design. The *generation technique* we are using (CADSE generator) makes very easy the development of a PL environment, and very easy to change it. In our experience, even the domain metamodel change rather frequently; it was one of the reasons to introduce the group mechanism which, de facto, extends the metamodel dynamically without any regeneration.

Scope evolution. The *composition technology* we have developed gives us much flexibility. It allows composing reusable CADSEs; it allows widening the scope of a given family by composing with another CADSE or extending the current one with new concepts, new components, new features, and new constraints.

Product evolution. The *universal and contextual constraints* allow to make explicit the knowledge at the level of any individual component (universal constraints), at the level of types and groups (universal constraints), and for any composite (contextual constraints). The *encapsulation mechanism* provided by composite factories and specialized component databases allows managing sub-families independently, and to hide the changes. It allows also scalability.

Not only our approach has improved evolution control, but also we have made significant progresses in what concern consistency, completeness and minimalism.

Consistency is improved because (1) feature selection is performed through expression transformation, (2) compatibility consistency, is enforced through universal and contextual constraints, and (3) version consistency is enforced through our version control mechanism. *Completeness* is enforced because the selection process follows all dependency links. *Minimalism* is enforced since only the relevant dependency links are followed.

However, the system ensures that if a solution exist it will be found, because in backtrack mode all the possible combinations are tried. However, for efficiency reasons, this facility can only be used in tiny databases. Heuristics could be used to speed up that process.

Our CADSE technology is running in operation for years, but no full size and operational validation is available yet with our Selecta system, but we hope it is already a contribution in the field of scalable, flexible and highly automated Product Lines.

9. REFERENCES

[1] Clements, P. C., and Northrop, L. 2003. A Framework for Software Product Line Practice – version 4.2. Carnegie Mellon, Software Engineering Institute.

[2] Deelstra, S., Sinnema, M., and Bosch, J. Product derivation in software product families: a case study. Journal of Systems and Software 74, 2 (2005), 173–194.

[3] Deelstra, S., Sinnema, M., and Bosch, J. Experiences in software product families: Problems and issues during product derivation. In SPLC (2004), 165–182.

[4] Deelstra, S., Sinnema, M., Gurp, J. V., and Bosch, J. Model driven architecture as approach to manage variability in software product families. In Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications (2003), Springer, 109–114.

[5] Estublier, J., Dieng, I. A., and Simon, E. Automating Component Selection and Building Flexible Composites for Service-Based Applications. In Proc 4th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE) (Milan, Italy, 2009).

[6] Estublier, J., and Vega, G. Reuse and Variability in Large Software Applications. Published in Proc ESEC/FSE. (Lisboa, Portugal, September 2005).

[7] Estublier, J. Configuration management, the notion and the tool. In Proc. of the ACM workshop on Software Version and Configuration Control, (Grassau, Allemagne, January 1988).

[8] Estublier, J., Vega, G., Lalanda, P., Leveque, T. Domain Specific Engineering Environments. In APSEC'08 Asian Pacific Software engineering Conference (2008), IEEE, 553–560.

[9] Gomaa, H., and Shin, M. E. Automated Software Product Line Engineering and Product Derivation. In HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences (Washington, DC, USA, 2007), IEEE Computer Society.

[10] K. Kang, S. Cohen, J. H. W. N., and Peterson, S. Feature-Oriented Domain Analysis (FODA) feasibility study. Tech. rep., Technical Report CMU/SEI-90-TR-21, Software Engineering Institute (SEI), (November 1990).

[11] D. Dhungana, T. Neumayer, P. Grunbacher, R. Rabiser. Supporting Evolution in Model-based Product Line Engineering. In SLPC 2008, San Francisco, USA p319-328.

[12] M. Pussinen. A survey on software product-line evolution. University of Tempere, December 2002. http://practise2.cs.tut.fi/pub/papers/swpl_evo.pdf

[13] W. Warme, A. Kleppe. Building a flexible software factory using partial domain specific models. In Proc 6th OOPSLA workshop on Domain Specific Modeling (DSM06) Portland 2006.

[14] C. Kim, K. Czarnecki. Synchronizing cardinality-based features models and their specialization. 8th IEEE Conf. On Engineering of complex Computer Systems. Greenbelt, USA, 2002.