Software Product Line Evolution with Cardinality-Based Feature Models

Nadia Gamez and Lidia Fuentes

Dpto de Lenguajes y Ciencias de la Comunicación, Universidad de Málaga
{nadia,lff}@lcc.uma.es

Abstract. Feature models are widely used for modelling variability present in a Software Product Line family. We propose using cardinality-based feature models and clonable features to model and manage the evolution of the structural variability present in pervasive systems, composed by a large variety of heterogeneous devices. The use of clonable features increases the expressiveness of feature models, but also greatly increases the complexity of the resulting configurations. So, supporting the evolution of product configurations becomes an intractable task to do it manually. In this paper, we propose a model driven development process to propagate changes made in an evolved feature model, into existing configurations. Furthermore, our process allows us to calculate the effort needed to perform the evolution changes in the customized products. To do this, we have defined two operators, one to calculate the differences between two configurations and another to create a new configuration from a previous one. Finally, we validate our approach, showing that by using our tool support we can generate new configurations for a family of products with thousands of cloned features.

Keywords: Software Product Lines, Feature Models, Evolution.

1 Introduction

Recently, pervasive systems and Ambient Intelligence environments are gaining popularity to support people's daily tasks. These systems are composed by a large variety of networked heterogeneous devices with embedded software. For instance, Ambient Assisted Living systems or Intelligent Transportation Systems (ITS) can be formed by a large number of sensor nodes (grouped in Wireless Sensors Networks, WSNs), smart phones, vehicles onboard computers or other devices with RFIDs or cameras. Application domains like pervasive systems, where heterogeneity is present at any level, can greatly benefit from Software Product Line (SPL) engineering [1], which is specifically focused on variability modelling. SPLs aim to provide techniques for creating infrastructures that allow the rapid and systematic production of similar software systems, promoting the reuse of common core assets.

Feature Models (FM) [2] have been widely adopted by the SPL community to specify which elements, or *features*, of the family of products are common, which are variable and the reasons why they are variable, i.e. if they are alternative elements or optional elements. Then, a feature model permits specifying where the variability is, independently of the core asset, and enables reasoning about all the different possible configurations of a family of products.

Specifically in heterogeneous pervasive environments, the most common variability is the *structural variability*, defined as variations in type, cardinality or naming of elements [3]. We propose using cardinality-based features models and *clonable features* [4] to model the structural variability present in the new generation of pervasive systems. The use of clonable features increases the expressiveness of FMs since they allow the creation of different configurations for the same kind of device using only one feature model. Using clonable features we can model so that a system has a variable number of different kinds of devices (e.g. *s* sensors, *c* cameras, *a* alarms, or *sm* smartphones). The cloning of these device features leads to the cloning of the related structure (e.g. for 3 sensors, the configuration will contain *s1*, *s2* and *s3* clones of the sensor feature, joint with its sub-tree), increasing the complexity of the resulting configurations, and moreover the number of possible configurations increases a lot. Then, as the FM evolves, the impact of propagating changes made in the FM to the possible configurations is much higher in a cardinality-based FM.

Evolving a FM may imply adding or removing a feature (e.g. adding a new encryption algorithm as a mandatory feature), which in a cardinality-based feature model may cause many changes in all the clones. Specifically in pervasive systems, configurations could have hundred of clones composing a single product configuration. So, considering the evolution of a concrete SPL, it would be useful to automatically obtain the evolved configurations according to the changes introduced to the FM. From the point of view of the SPL engineer, it would be useful to know the necessary effort to evolve a previous existing product configuration to a new valid configuration after a FM modification was performed. This effort could be calculated by comparing the previous and the list of new possible configurations; which is not trivial to do at first glance due to the high number of cloned features.

In this paper, we present how we manage automatically the evolution of an pervasive system software product lines using cardinality-based FM and clonable features. To do this, we have defined two operators between FM configurations that are not trivial for cardinality-based FM. The *create_configuration* operator allows the creation of a new configuration from a previous configuration and the features that must be added or removed in the new configuration. The *differences* operator calculates the differences between two configurations of a feature model. We use the *create_configuration* operator to create evolved configurations from the previous configuration and the evolved feature model. Furthermore, we use the *differences* operator to calculate the effort of evolving the product configurations of a SPL, reusing and preserving the elements of the previous configuration. Finally, we validate our approach showing that by using our tool support we can easily evolve FMs with clonable features, automatically generating new configurations, for configurations with a high number of clones.

The remainder of the paper is organized as follows. In Section 2, we present our motivation example and the challenges for evolving pervasive systems SPLs and how we achieve them. In Section 3, we show our approach and Section 4 details the *differences* and *create_configuration* operators. The validation and the tool support of our approach are presented in Section 5. In Section 6, we compare our approach with related work. Finally, in Section 7 we outline some conclusions.

2 Motivation

In this section we present a motivating example and we will discuss the special challenges of pervasive systems that make them good candidates to take advantage of the evolution process using SPL and cardinality-based FMs.

2.1 Motivation Example

One of the most popular pervasive systems are smart homes with a lot of appliances that helps the occupants of the house in their daily life. When the purpose of a smart home is to enhance the quality of life of dependant people, then we are talking about Ambient Assisted Living (AAL). In this paper our motivating example is a SPL of AAL homes, equipped with sensors, smartphones, alarms, and cameras as shows the FM of Fig. 1.a.

Fig. 1.a represents a FM in Hydra[1] (all the FM and configurations presented throughout this paper are modelled using our featuring modelling tool, Hydra). In a FM every feature has one parent except the *root feature* (as *AALHome* in Fig.1). The features can be *mandatory* (as *Encryption*), *optional* (as *VideoSurveillance*), or *clonable* (as *Sensor* that has a 0 to infinite cardinality). Apart from the features, Hydra also defines two groups of features: *xor*-group (as the group composed by the operating systems of the *Smartphone*: *Android* or *iPhone*) and *or*-group (as the one composed for the sensing units of the *Sensor*: *Accelerometer*, *Light*, *Humidity,* or *Temperature*). So in Hydra, we can distinguish two kinds of relations: between a feature and its children features (*and*-relationship, as in the relation between the *AALHome* and its *Services*) and between a feature and one group (as in the relation between the *Sensor* and its *xor*-group).

Fig. 1.b shows a valid configuration for the AAL home family. A configuration of a feature model is the selection of a set of features belonging to the feature model. A configuration is valid if all features contained in the configuration and the deselection of all other specific features contained is allowed by the feature model [5, 6]. So, a valid configuration must satisfy the *tree-constraints* and the dependencies or interactions between features (*cross-tree constraints*). In Hydra, the cross-tree constraints are expressed in a textual way using the combination of regular expressions, as for example, *VideoSurveillance implies any Camera*.

The home of the configuration shown in Fig. 1.b has video surveillance facilities to transmit periodically video to the health centre. Also an automatic control of the lights and heat is provided. Furthermore all the data transmitted must be encrypted. This configuration has 10 sensors: the sensor S1 has a temperature sensing unit and offers temperature monitoring, the sensor S2 has in addition a humidity sensing unit and the sensors from S3 to S10 are identical and are equipped with accelerometers and light sensing units and offer light monitoring facilities. In this configuration there are also 2 smartphones and 8 cameras. Note that the figure does not show all these devices for the sake of simplicity. The Phone 2 is an Android smartphone and provides an application to transmit the video received from the camera to the health centre. Similarly, the cameras must transmit the video to the smartphones. Finally, all the devices have an encryption algorithm installed since this feature is mandatory.

[1] http://caosd.lcc.uma.es/spl/hydra

Normally, only a subset of the family products are developed and marketed. Later, these products are mainly subject to two evolution scenarios: (1) one AAL home may focus on dependant people with movement difficulties. However, some of the dependant people may not have special movement problems, but problems due to diabetes, or both. So, this AAL home family of products must evolve in order to incorporate a glucose sensor device, specific for diabetic people. This means that the customers demand a new functionality to the family of products, so each product already developed, and even deployed in some houses, must be evolved in order to incorporate the new requirements; (2) the hardware and software technology for pervasive systems is continuously evolving. New operating systems (e.g. Android for mobile phones) or special sensors (e.g. new accelerometers) are frequently appearing. So, vendors must incorporate these new devices or facilities into their products already derived, in order to be competitive in the market.

2.2 Challenges

The heterogeneity present in pervasive systems is easily manageable with cardinality-based feature models. Furthermore, these kinds of systems are continually evolving, as new devices, application facilities or requirements appear, and as a consequence of this some obsolete features disappear. So, the evolution of these systems must be properly supported by advanced tools. Now, we enumerate and detail the specific **challenges** to manage the evolution of pervasive systems using SPLs.

- **C1 Structural Variability Evolution:** A special characteristic of pervasive systems is that many instances of the same device may compose the same product, but each device, although being of the same type may have a different configuration. In the AAL home presented, the device infrastructure would be similar for all products, but must be customized to the physical structure of each house or to the necessities of the dependant person. Such structural variability must be explicitly modelled in the SPL, but also its evolution must be part of a SPL engineering process. <u>Achieving C1</u>: We model such structural variability with clonable features, and manage its evolution, not at the feature, but at the clone level (see Section 3). So, it is possible to modify the configuration of sensor S1, but not of the other sensors.
- **C2 Automatic Change Propagation:** When a SPL evolves, the changes must be propagated to the customized products of the family. Nevertheless in these kinds of systems with a high number of devices, each one with very specific characteristics

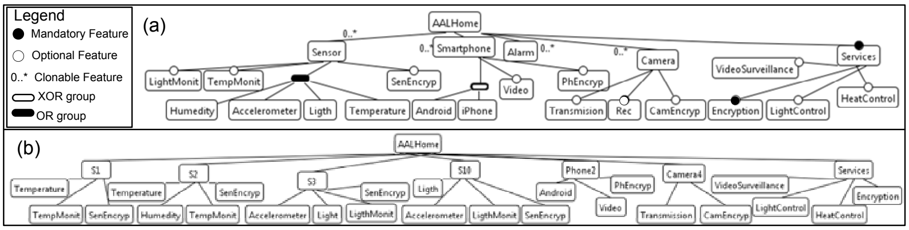

Fig. 1. AAL Home Family Feature Model (a) and Configuration (b)

(Fig.1.a) the propagation of changes is very complex. So, we need an automatic process that supports the evolution changes made in some characteristics of the SPL to all the derived products of this family. Achieving C2: We provide a tool support to automatically propagate the changes made at FM level into all customized configurations (Section 3.1). Note that representing configurations graphically with many cloned features will complicate the management of configurations evolution. This is important, since some pervasive systems may be composed of hundreds of devices of the same type, so it would be impossible to handle changes one by one manually.

- **C3 Evolution Effort:** Since most pervasive systems are composed of several or many different clones of each device, this implies that evolution changes must be performed in every clone in a different way. Let's imagine that we want to remove the encryption algorithm of the sensors in our AAL home family. But the application architecture may be different for every sensor, so, the way to remove the encryption is also different. So, it is necessary to automatically evaluate the required effort to make the changes in the products of the family when the family evolves, since when we have several instances of similar devices, but with different architectures, this is a very complex task. Achieving C3: We automatically calculate the differences between a previous configuration and the new evolved configuration for all the existing configurations (see Section 3.2). We use this difference, the FM and a mapping between every feature and the corresponding architecture to obtain which components of the architecture must be added or removed in every device. In this way, we can quantify the effort of evolving a product and the impact of change when the FM evolves. This may also help the SPL engineer to assess the persons per month required to produce upgraded versions of previous products.
- **C4 Preserve Compatibility:** In pervasive systems, the applications installed in all devices normally interact and collaborate between them. This means that the SPL process must guarantee that the configurations running in every device of a certain pervasive system are compatible with each other. An example of compatibility in the context of sensor nodes is that all of them have to use the same routing protocol, otherwise the communication is impossible. The SPL evolution process must ensure that new configurations of different kinds of devices are compatible. Achieving C4: We also use the cross-tree constraints to guarantee that the configurations of all the devices (i.e. *clones*) of a certain system are compatible with each other (see Section 3.1)**.** This novel use of the constraints specified between clones makes it possible to specify which architectural elements must be present in all devices that interact.
- **C5 Efficiency:** As many pervasive systems are composed of a large number of devices (as hundreds or thousands of sensor nodes executing several sensing tasks), the number of configurations of a simple FM would be really high. The FM configuration of a particular system may contain thousands of features due to the cloning of each device related structure (sub-tree) for every device. So, we must ensure that the tool support for creating new evolved configurations or for searching the difference between the evolved and the previous configuration has to be efficient. Achieving C5: We define and implement two operators *difference* and *create_configuration* (see Section 4) paying special attention to efficiency and as we will show in the evaluation (see Section 5), the execution time is efficient, being appropriate for thousands of features, typical of pervasive systems.

3 Evolution of Feature Models with Clonable Features

We have proposed an automatic process to derive different system configurations depending on the input constraints, determined by mainly hardware and software requirements [7]. We apply model-driven and SPL engineering techniques to automate this configuration process. The first step when creating a SPL is to analyze the variability inherent in the application domain and create the FM. In the next step the global architecture of the system (named *product line architecture* or PLA), which contains both the commonalities and the variabilities specified by the FM is defined. A *Feature Mapping* between the FM and the PLA defines the correspondence between features and architectural components. We propose the use of the variability language VML [8], which was defined specifically to do this mapping. The customization of the architecture is determined by a set of high-level parameters (e.g. number and type of sensors or the necessary services). Using this set of parameters as input features, Hydra is able to automatically infer the rest of the features needed for each product making use of tree and cross-tree constraints (i.e. feature interaction), defined as part of the FM. So the output of Hydra is then a configuration of a product. This product configuration and the mapping between the FM and the PLA specified in VML are the inputs of a model transformation that automatically generates a customised architectural model. Finally, the architectural model of a product is the input of a model-to-text transformation, which produces the code for deploying the specific application inside the devices. We detail how our process automatically propagates the changes made in a FM into current configurations, and also we evaluate the necessary effort to propagate those changes to the final architecture.

3.1 Feature Model Evolution

As we have mentioned previously, the SPLs need to evolve in order to satisfy new user or application requirement or to incorporate new technological advances as for instance, devices with new operating systems or new facilities to achieve energy efficiency or the security of the system. These evolution scenarios must be modelled as modifications in the FM. We have identified what elements of the FM may change as a consequence of an evolution scenario: (i) adding or removing features, (ii) adding or removing groups of features ('*or*' or '*xor*' groups), (iii) adding or removing constraints between features and (iv) modifying the variability of a feature (e.g. a mandatory feature is transformed in optional). Note that the modification of a feature can be defined by means of removing a previous feature and adding a new one. The same happens with the modification of groups of features and with the constrains.

Let's imagine that we want to evolve the AAL home family with new services: glucose control for diabetic people and fall detection for people with movement problems or other illnesses that may provoke falls. Furthermore, due to the rapid loss of energy of the sensors, the removal of the encryption algorithm in those sensors where it is not strictly required is recommended. Fig. 2.a shows the FM of the AAL home with these evolution changes. The two new services are added as new optional features (*GlucoseControl* and *FallDetection*) and since they can be used in any device, they are also added as children optional features of every device (as the glucose and fall monitoring in sensors or the diabetes application in the smartphones). Also the glucose sensing unit is added as a child of the sensor 'or' group. Furthermore, the *Encryption* mandatory feature is now an optional feature and we have added a new constraint for this feature: *Encryption implies (PhEncryp and CamEncryp)* to force both camera and mobile phone to transmit secure data. Finally, we have added other constraints related with the new services, for example, *GlucoseControl implies Diabetes* or *FallMonit implies Accelerometer.*

After evolving the FM we have to propagate the changes in all the previous configurations, as the one shown in Fig.1.b. Our process automatically obtains the new configuration from the previous one, the new FM and the requirements with respect the evolved features for this specific product. Consider as these requirements, that for the configuration shown in Fig. 1.b either the customers or the vendor needs the fall detection and glucose control services. The output of our process after this evolution will be the configuration shown in Fig. 2.b. The modifications are: (1) two new features are added as children of the *Services* feature; (2) the *Encryption* feature is removed; (3) in all the sensors the *SenEncryp* feature has been deleted; (4) a new sensor *S11* with a glucose sensing unit and with a glucose monitoring service is added; (5) in all the sensors equipped with accelerometers the fall monitoring feature is added; (6) the fall recording task is added to the cameras (7) and finally, the facility for controlling the diabetes and for transmitting the fall is added to the smartphones.

We can observe that with this small example we have to manage many changes in several features, so in systems with hundreds of nodes the number of changes increases exponentially. Therefore, we need a tool support that creates this new configuration in an efficient way, considering that the number of features of these kinds of systems may be really large. As is shown in Fig. 3.a, in order to automatically obtain this new configuration the **Create Configuration** facility of Hydra takes as input the evolved feature model, the previous configuration and the constraints with the requirements of this configuration for the evolved features, and it returns a set of constraints with all the features that we have to select in the new configuration. These constraints are used together with the new feature model by the Hydra facility to automatically generate a **Minimal Valid Configuration.** To implement the **Create Configuration** we use the *create_configuration* operator, defined in the Section 4, that obtains the features that must be selected in the new configuration for cardinality-based feature models. Also, how Hydra gets the **Minimal Valid Configuration** is explained in Section 4.

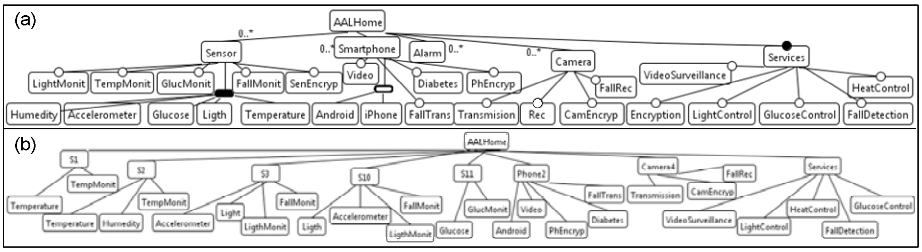

Fig. 2. AAL Home Family Evolved Feature Model (a) and Evolved Configuration (b)

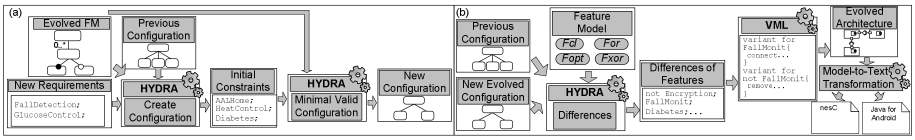

Fig. 3. (a) Evolving FM Configurations and (b) Obtaining Architectural Differences

3.2 Evolution of Existing Configurations

In order to evaluate the impact of change when a FM is evolved, we need to know the specific differences between the previous configuration and the new evolved configuration. To do this, as Fig. 3.b shows, the **Differences** facility of Hydra takes as inputs the previous, the new configurations and the four sets of variable features of a FM (clonable, optional, '*or*' and '*xor*' group) and it returns the difference between configurations by means of a set of features that must be selected and unselected in the new configuration. Obtaining the differences of FM configurations with clonable features is not a trivial task, since it cannot be calculated as a simple difference of sets as can be done for normal FMs. So, in the next section we have defined a *difference* operator for the special case of FM configurations with clonable features. Our process then uses this *differences set of features* and the mapping between FM and the PLA in VML to automatically produce the evolved architecture. Thanks to VML it is possible to automate the customization of the family architecture in an SPL context. Using VML, we specify which actions must be performed on the architectural model when a certain feature is selected or unselected. These mechanisms are basically adding and removing components, and connecting component interfaces. The family architecture is specified using the components model of UML 2.0.

Figure 4.a shows an extract of the mapping between our AAL home FM and the UML components of the family architecture, including all the variants of the FM (i.e. mainly devices and services). We only include in Figure 4.a the mapping for the features involved in the evolution changes, but in a full version of this VML file all the features which have an influence in the architecture are included. Lines 01-09 show the architectural modifications that must be performed when the *SenEncryp* feature is not selected, that is, the encryption-related components must be removed from all the applications installed in the sensor devices. Nevertheless, as we have explained previously, because the architecture in some sensors is different, how to remove the encryption-related components is also different for each sensor. We illustrate this for sensors *S1* and *S2* and sensors *S3-S10*. Lines 01-04 show the architectural mapping when *SenEncryp* and *LightMonit* features are not selected, which is the case of sensors *S1* and *S2*. In this case, the *elliptic curve cryptography* algorithm component must be removed (line 02) and also the component that composes output messages with the data collected by the sensor (*DataReady*) must be connected through *IData* interface with the component that transmits the data through the network (*DataTransmission*, line 03). We can see at the top of Fig.4.b the evolution of the architecture corresponding to these features. Nevertheless, as is shown at the bottom of Fig. 4.b., in sensors *S3-S10* where the *LightMonit* feature is selected, we also have a component that is responsible for fusing the data using an aggregation function, with the goal of reducing the number of messages that are sent through the network. In the case, where *SenEncryp* is not selected, apart from removing the *ECCALgorithm* component (line 06), the *DataReady* component must be connected with the *DataFusion* component (line 07) and this component in turn must be connected with the *DataTransmission* one (line 08). Here we show how we can manage automatically the architectural modifications in clones of the same feature (sensor *S1* and *S2* and sensors *S3-S10*) that have different architectures. In this figure we highlight only the components or architectural parts that are previously implemented so they can be reused in the new architecture.

(a)

```
01 variant for (not SenEncryp and not LightMonit){//Sensor
02  remove("ECCAlgorithm");
03  connect("DataReady", "DataTransmission", "IData");
04 }
05 variant for (not SenEncryp and LightMonit){//Sensor
06  remove("ECCAlgorithm");
07  connect("DataReady", "DataFusion", "IData");
08  connect("DataFusion", "DataTransmission", "IData");
09 }
10 ...
11 variant for FallMonit{//Sensor
12  connect("MovTracking", "FallAlgorithm", "IDetection");
13  connect("FallAlgorithm", "CameraSelection", "ICamSel");
14  connect("CameraSelection", "DataReady", "ICam");
15 }
16 variant for FallTrans {//Smartphone
17  connect("Video", "TransmissionHC", "IVideo");
18 }
19 variant for FallRec {//Camera
20  connect("Rec", "Transmission", "IVTrans");
21 }
22 ...
23 variant for GlucMonit {//Sensor
24  connect("GlRead", "DataReady", "IGluc");
25 }
26 variant for Diebetes {//Phone
27  connect("GlReceive", "GlAnalyze", "IGlAnalyze");
28  connect("GlucAnalyze", "GlucLimit", "ILimit");
29  connect("GlucLimit", "NotifyHC", "INotify");
30  connect("GlAnalyze", "GlRegister", "IRegister");
31 }
```

(b)

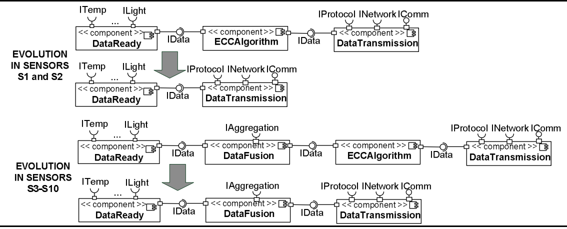
EVOLUTION IN SENSORS S1 and S2
ITemp
ILight
<< component >> DataReady
IData
<< component >> ECCAlgorithm
IData
IProtocol INetwork IComm
<< component >> DataTransmission
EVOLUTION IN SENSORS S3-S10
IAggregation
<< component >> DataFusion

Fig. 4. (a) Mapping between Features and Architecture and (b) Removing the Encryption

Now, we need to implement the new functionality. Firstly, an algorithm to detect possible falls and also that switches to the nearest camera when a fall is detected. This new component must be deployed in the 7 sensors equipped with an accelerometer. Also, the video captured by the camera has to be transmitted through the smartphone to the health centre. But, the components that implement this functionality were previously used for the video surveillance, so we simply reuse them. Lines 11-15 show the architecture corresponding with the selection of the *FallMonit* in sensor devices, lines 16-18 specify that in the smartphone we have to connect the *Video* with the component that transmits it to the health centre only if the *FallTrans* feature of the smartphone is selected. But all this functionality was already in the architecture, so we reuse the corresponding components. The same happens when the *FallRec* feature is selected in the camera (lines 19-21). Finally, a new sensor with glucose monitoring facilities must be incorporated (lines 23-25). Also, the components that analyze and register the glucose measures and that notify the health centre if any measure exceeds the limits is added in the smartphone (lines 26-31). These components are new, so the effort of adding this new sensor must take into account the effort of implementing all of them. In total, 9 components were added and the rest were reused. Our process is able to infer the list of these components automatically, helping the architect to assess the effort of evolving each existing product when the FM evolves.

4 Differences and Create Configuration Operators

Differences Operator. The *differences* operator obtains the set of different features between two configurations of a FM. This set of difference contains the features that were selected in the Previous Configuration (PC) and are not selected in the New Configuration (NC) and the features selected in the NC but were not selected in the PC. When considering FM without cardinality the problem for getting the differences can be simplified to the differences between two sets. Likewise, if we rename all the cloned features with a unique name, a first approach could be to also reduce the problem of the difference of cardinality-based FM configurations, to a simple difference between sets. In order to have a unique name for every feature, we can prefix the name of all the clones with the original name of the feature (e.g. *Sensor_S10*) and all the features of the cloned structures with the name of the clone (e.g. *S10_Accelerometer*). So, to obtain the differences firstly we have to cover all the features of the NC in order to know which ones are not in the PC and secondly, we have to cover all the features of the PC in order to know which ones are not in the NC. This would be correct since all the differences are returned but it is not efficient. Firstly, we have to rename all the features. Then, we have to navigate through all the features of the PC and find them in the NC. After that, we have to do the same with all the features in the NC by navigating through the PC to find them. So, the main disadvantage of this approach is that all the features selected in the two configurations have to be searched twice. Furthermore, we know in advance that mandatory features are selected in both configurations so we do not need to search for them in order to obtain the differences. Apart from the mandatory features, other features will be selected in both configurations, so the effort of searching the configuration tree twice is not justified. This approach would work well in small FMs and small configurations, i.e. configurations with a few cloned features, but this is not happening in real pervasive systems that may have hundreds of devices. Also, a FM representing a real SPL system usually has a big core asset, so they have many mandatory features. Finally, we may want to obtain the differences between two similar configurations, where only some of the features in a few clones are different, as is mainly happen with evolved configurations. To summarise, this approach for calculating the difference is extremely inefficient for real SPL of pervasive systems. And, as we will show in the evaluation section the time needed to find the differences between two configurations increases greatly when the number of clones increases. So, we have defined a more efficient algorithm in order to make our approach scalable to configurations with several thousands of total features.

Syntactic Definition. $differences\colon PC, NC, Fcl, Fopt, For, Fxor \rightarrow SEL, UNS$ It takes six sets of features as input arguments. The features selected in the PC and in the NC, the clonable and optional features and the features belonging to an 'or' and a 'xor' group. Also, it returns two sets of features. SEL is the set of the features that are selected in the NC but were not selected in the PC. Similarly, UNS is the set of the features that are not selected in the NC but were selected in the PC.

For the evolved FM shown Fig. 2.a the sets of features are:

$Fcl = \{Sensor, Smartphone, Alarm, Camera\}$
$Fopt = \{GlucoseControl, FallDetection, Encryption, \dots\}$

$For = \{Accelerometer, Ligth, Temperature, Glucose, Humidity\}$
$Fxor = \{Android, iPhone\}$

For the configurations presented in Fig. 1.b and Fig 2.b, we have:

$PC = \{AALHome, VideoSurveillance, Sensor\ Sensor10\colon \{SenEncryp, Ligth, \ldots\},$
$Smartphone\ Phone2\colon \{Video, \ldots\}, Camera\ Camera4\colon \{Transmission, \ldots\}, \ldots\}$
$NC = \{AALHome, GlucoseControl, FallDetection, VideoSurveillance,$
$Sensor\ Sensor10\colon \{FallMonit, Light, \ldots\},\ Smartphone\ Phone2\colon \{Video, Diabetes, \ldots\},$
$Camera\ Camera4\colon \{Transmision, FallRecord, \ldots\}, \ldots\}$

And as result of the *differences* operator, we will have to obtain:

$SEL = \{GlucoseControl, FallDetection,$
$Sensor\ Sensor10\colon \{FallMonit\}, Sensor\ Sensor11\colon \{Glucose, GlucoseMonit\},$
$Smartphone\ Phone2\colon \{Diabetes,\ FallTrans\}, Camera\ Camera4\colon \{\ FallRecord\}\} \ldots\}$
$UNS = \{Encryption, Sensor\ Sensor10\colon \{SenEncryp\} \ldots\}$

Semantics. It is represented by the relationship that exists between the PC and the NC and the selected and unselected features. Intuitively, the NC minus the PC is equal to the selected set of features. And in the same way, the PC minus the NC is the unselected features set. Then, $SEL = NC \setminus PC$ and $UNS = PC \setminus NC$.

Algorithm 1. *Differences*
returns two sets of features: one (SEL) with the features that are selected in the NC and not selected in the PC and other set (UNS) with the features that were selected in the PC not selected in the NC.
inputs six sets of features $PC, NC, Fcl, Fopt, For, Fxor$
output a tuple of two sets of features $diff = (SEL, UNS)$

```
1: $SimplePC := remove\_clones(PC)$
2: $SimpleNC := remove\_clones(NC)$
3: $(SEL, UNS) := diff\_simple\ (SimplePC, SimpleNC, Fopt, For, Fxor)$
4: foreach $f \in Fcl$ do
5:     $(ClPC, StPC) := extract\_clones\ (PC, f)$
6:     $(ClNC, StNC) := extract\_clones\ (NC, f)$
7:     for $i := 1..length(ClPC)$ do
8:         $c := ClPC(i)$ //clone of the i position
9:         if $(c \in ClNC)$ then
10:            $j := pos(c, ClNC)$ //search for the position of the clone c
11:            $(CSEL, CUNS) := diff\_simple\ (StPC(i), StNC(j), Fopt, For, Fxor)$
12:            $SEL := SEL \cup precede\_clones(c, CSEL)$
13:            $UNS := UNS \cup precede\_clones(c, CUNS)$
14:        else
15:            $UNS := UNS \cup precede\_clones(c, StPC(i))$
16:        end if
17:    end for
18:    for $i := 1..length(ClNC)$ do
19:        $c := ClNC(i)$
20:        if $(c \notin ClPC)$ then
21:            $SEL := SEL \cup precede\_clones(c, StNC(i))$
22:        end if
23:    end for
24: end for
25: $diff := (SEL, UNS)$
26: return $diff$
```

Algorithm. This algorithm firstly obtains the differences of the non clonable features. To do so, it uses the *diff_simple* algorithm (Algorithm 2) that obtains the difference between two configurations with non clonable features. *diff_simple* covers all the optional, '*or*' and '*xor*' features in order to know which ones are in the NC but were not in the PC to construct the *SEL* set, and which ones were in the PC but are not in the NC to construct the *UNS* set. Note that we avoid looking for the mandatory features, since they will be selected in both configurations. Also, using this algorithm we avoid covering all the features for the PC and the NC twice.

The *differences* algorithm uses *diff_simple* algorithm giving as input the PC minus the cloned structures, the NC minus the cloned structures and the optional, '*or*' and '*xor*' features (Algorithm 2, lines 1-3). Then, for each clonable feature, the algorithm extracts the clones of the PC and NC (lines 5-6). The *extract_clones* function returns a tuple with the names of the clones and with the cloned structures. The cloned structures are the features under a clone but only those that are not clonable again, since they will be considered later in the algorithm. Then, for each clone of the PC, if they appear in the NC, the *diff_simple* algorithm calculates the differences between both corresponding cloned structures (line 11). If the clone it is not present in the NC, all the structure (preceding the feature with name of the clone) must be added to the *UNS* set (line 15). Following on, for each clone of the NC, if it does not appear in the PC, all the structure must be added to the *SEL* set (lines 18-21).

Finally, we have to consider the possibility that some descendant features of a clonable feature will also be clonable (nested clones). Our algorithm takes into account this possibility since, as we mentioned before, the *extract_clones* function return all features in the cloned structure minus the clonable ones. And, as the algorithm covers all the clonable features, when the turn of a clone that belongs to a cloned structure comes, the *extract_clones* function obtain again the substructure of this clone features (without clonable features).

Algorithm 2. *diff_simple*
returns two set of features: one (SEL) with the features that are selected in the NC and not selected in the PC and other set (UNS) with the features that were selected in the PC and not selected in the NC.
inputs five sets of features $PC, NC, Fopt, For, Fxor$
output a tuple of two sets of features $diff = (SEL, UNS)$

```
1: $SEL := \emptyset$
2: $UNS := \emptyset$
3: foreach $f \in (Fopt \cup For \cup Fxor)$ do
4:     if $(f \in NC) \wedge (f \notin PC)$ then
5:         $SEL := SEL \cup \{f\}$
6:     elseif $(f \in PC) \wedge (f \notin NC)$ then
7:         $UNS := UNS \cup \{f\}$
8:     end if
9: end for
10: $diff := (SEL, UNS)$
11: return $diff$
```

Create Configuration Operator. The *create_configuration* operator creates a NC from a PC and the two sets of differences: the features that must be selected in a NC (*SEL*), and ones that must be unselected (*UNS*). To generate the NC, firstly, we have to remove the unselected features from the PC set. After, with this set plus the set of selected features we use the facility of Hydra to create a minimal valid configuration.

Syntactic Definition. $create\_configuration: PC, SEL, UNS, FM \rightarrow NC, Valid$ It takes three sets of features and the FM as input arguments: the features selected in the PC, the set of features that has to be selected in the NC but they were not selected in the PC, and the set of features that must not be selected in the NC but they were selected in the PC. The FM is given as input, represented by a propositional formula [6]. It returns the set features that must be selected in the NC and a Boolean that indicates if it is possible to create a valid configuration with those inputs.

Semantics. It is represented by the relationship that exists between the PC and the NC. Similarly to the *difference* operator, the NC is equal to the PC minus the unselected features plus the new selected features: $NC = (PC \backslash UNS) \cup SEL$.

Algorithm. This algorithm firstly assigns to the NC the features of the PC (line 1). Then, for each feature of the *UNS* set checks if it is a clone, to remove it using the *remove_clone* function (lines 3-4). This function has two inputs, the clone and the features of the configuration. If the feature is not a clone, i.e. it is simple feature, the algorithm removes it directly (lines 5-6). Similarly, for each feature of the *SEL* set, checks if it is a clone, to add it using the *add_clone* function (lines 10-11). This function has two inputs, the clone and the set of features of the configuration. Finally, if the feature is not a clone, the algorithm adds it directly (lines 12-13).

Algorithm 3. *create_configuration*
returns a tuple of a set of features of the new configuration and a Boolean value that indicates if for the inputs a valid configuration must be generated
inputs three sets of features PC, SEL, UNS and a feature model as a propositional formula FM
output a tuple with the set of features of the new configuration NC and a Boolean value $Valid$

```
1: $NC := PC$
2: foreach $f \in UNS$ do
3:    if ($is\_clone(f)$) then
4:       $NC := remove\_clone(f, NC)$
5:    else
6:       $NC := NC \backslash \{f\}$;
7:    endif
8: end for
9: foreach $f \in SEL$ do
10:   if ($is\_clone(f)$) then
11:      $NC := add\_clone(f, NC)$
12:   else
13:      $NC := NC \cup \{f\}$;
14:   endif
15: end for
16: $(NC, Valid) := minimal\_valid\_conf\ (NC, FM)$ // implemented by Hydra
17: return $(NC, Valid)$
```

After all the features in *UNS* and in *SEL* sets are covered, we have all the features that we want that will be selected in a NC. Then, this NC set is given as input together with the propositional formula of the FM to the Hydra minimal valid configuration function (*minimal_valid_conf*, line 16). This function returns *true* if a valid configuration can be generated (i.e. the NC given as initial constraints satisfy the tree and cross-tree constraints) or *false* in the other case. Also, this function returns the definitive NC. Maybe other features must be added to satisfy some constraints.

In order to check if a configuration is valid, Hydra uses a java library for Constraint Satisfaction Problems (CSP) [9], called Choco [10]. A CSP is defined by a triplet (X, D, C), where X is a set of **Variables**, D is a set of **Domains** for the variables and C is a set of **Constraints**. Hydra models the configurations by a CSP where the **Variables** are the features of the FM, the **Domain** is $\{0,1\}$ that corresponds with the semantic of the unselected feature or selected feature, and the **Constraints** include the implicit and the explicit cross-tree constraints. Furthermore, Hydra permits the automatic generation of the minimal valid configuration given a set of **initial constraints**. This is the valid configuration with less numbers of features that satisfy these initial constraints that are formulated in the same way that the explicit cross-tree constraints. This time, the **Constraints** include also these initial constraints and to get a minimal configuration Hydra uses the CSP **Objective Function**. For our purpose the function to **minimize** is the number of features selected, i.e. the number of variables with 1 value. So the objective function to minimize is $\sum_{i=1}^{n} v_i$.

5 Evaluation

Hydra, was first implemented as an Ecore-based Eclipse plugin [13, 14], to provide support for the modelling of cardinality-based feature models in an intuitive and graphical way. Hydra also provides support for the configuration, validation and automatic generation of minimal configurations of this kind of FM with clonable features. Within the scope of the present work, we extended the tool to implement the *differences* and the *create_configuration* operators to help the evolution of FMs. In this section, we present the experimental results of using the evolution support of our feature modelling tool, Hydra. We will show that Hydra works well with FM with a large number of cloned features, as required by challenge C5.

The time needed to create a configuration depends on the number of features selected for the configurations. So, for our small example, it depends very much on the number of clones, as is shown in Fig. 5.a. The experiments were done in a PC Intel Core 2 Quad, 2,5GHz, 2 GB of memory and with 1.6 JVM. In our evolved FM (Fig. 2.a), if we consider 30 sensors, 3 smartphones, 10 alarms and 10 cameras, the time needed to create a configuration is 1,7 seconds. It is a very reasonable time, since configurations of our feature model with 100 devices may have around 400 features.

So, if we clone 500 devices (we have 4000 features) the time is around 2 minutes. Instead, the time required to know the differences between two configurations is reduced (Fig. 5.b). Concretely, for 500 clones it takes 49 seconds. This happens because when Hydra creates a new configuration, executing the *create_configuration* algorithm it also has to paint the model of the configuration, which is the most time consuming task. By contrast, the *difference* only produces a file with the constraints. But both times are more than acceptable for huge configurations, so our approach is also scalable to configurations with around 4000 features.

Although the results presented here can be applied to any SPL, we specifically have applied them to a family of middleware for pervasive systems (FamiWare [7]). With FamiWare we have developed many case studies from the domain of pervasive systems. Specifically, we have implemented several versions of smart homes, AAL

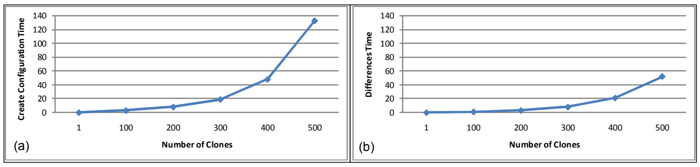

Fig. 5. Create Configuration (a) and Difference (b) operators time in seconds

homes and ITSs, with good results, although the number of clonable features, specifically in the ITS, was in the hundreds. For the ITSs, with a variable (without an upper limit) number of devices, an undetermined large number of different configurations were obtained. For this and for case studies similar in size, is not possible to manage the configuration evolution manually due to the high number of possible configurations and features per configuration. Since the ITS are novel, is very important that the SPL engineer can manage automatically the configuration evolution as proposed in our process, reducing the time to market for producing upgraded versions of this products. Although our process helps in the quantification of the effort required to produce the upgraded versions of previous products, it cannot calculate it in terms of the number of people per month. The output of our process for this is simply the list of components reused (those that were found in the component repository), and those which have to be implemented from scratch. So, the SPL engineer will have to assess the person per month per each new component and then make a final calculation of the estimated cost of evolving each product asset.

One desirable situation is that running products must continue their execution after evolution, so the initial requirements imposed by both the physical infrastructure (e.g. number of rooms) and the customer, which continue to be valid, must be preserved in the upgraded versions. Our process preserves the requirements and architecture, introducing the architectural modifications in the least intrusive way possible.

6 Related Works

Previous works [4,13,14] proposed some operations with cardinality-based FM. In [4] a cardinality-based notation for FM, on which Hydra is based, is presented. Also, the concept of staged configuration based on the specialization of FM is defined, where in each stage the products described by the specialized models is a subset of the products described by the FM. In [13] a verification of FM with clonable features using binary decision diagrams is presented. Both approaches are focused on cardinality-based FM but they do not deal with the evolution of this kind of FM. In [14] a synchronizing operation in cardinality-based FM is presented. They consider the possibility of propagating the changes produced in a FM to a existing specialization of this FM. So, in some way, they deal with the evolution of the FM and the corresponding changes to the specific products. Nevertheless, they do not provide a solution to the problem of propagating these changes at architecture level, as we do. Our model-driven process to evolve SPLs is one of the most important contributions of our work, since it allows the creation of new products and the evaluation of the effort of the evolution.

A classification of the evolution of a FM via modifications as refactoring, specializations, generalizations or arbitrary edits is presented in [6, 15]. So, an algorithm for classifying feature models with differences is defined. Similarly, in [16] an insert operator (to add a feature to a FM) and a merge operator (to compose two FM) were proposed. With these operators, the development of large feature models by composing smaller feature models is enabled. These proposals tackle the evolution of FM but they do not address how to propagate the changes made at FM level into the configuration level, as is the focus of our approach.

At configuration level, in [17] the work presented has similar motivations as our approach, since the authors propose the necessity of automated diagnosis of configurations in large FM. Also, they deal with the automatic configuration evolution. Nevertheless, they do not take into account the FM with clonable features, which are the main motivation of our work.

7 Concluding Remarks

We have presented an model-driven process for managing the evolution of SPL pervasive systems using a cardinality-based FM. Our process automatically propagates the evolution changes of the FMs into the existing configurations and also allows us to calculate the effort in performing the changes in every configuration. To do this, our tool Hydra creates new configurations from previous ones and the evolved FM. Furthermore, having the previous and the new configuration and using the variability language VML we can identify which parts of the architecture must be changed to evaluate the impact of the changes. We have defined the *differences* and the *create_configuration* operators and we have developed efficient algorithms to show their functioning. We have shown that Hydra is able to create new configurations and to see differences for configurations with a large number of clones.

References

1. Pohl, K., Böckle, G., Linden, F.: Software Product Line Engineering – Foundations, Principles, and Technique. Springer, Heidelberg (2005)
2. Lee, K., Kang, K., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, pp. 62–77. Springer, Heidelberg (2002)
3. Sánchez, P., Gámez, N., Fuentes, L., Loughran, N., Garcia, A.: A Metamodel for Designing Software Architectures of Aspect-Oriented Software Product Lines. Technical Report D2.2, AMPLE Project (2007)
4. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice 10, 143–169 (2005)
5. Batory, D.S.: Feature models, grammars, and propositional formulas. In: Obbink, J.H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
6. Thüm, T., Batory, D., Kästner, C.: Reasoning about edits to feature models. In: Proceedings of the 31st International Conference on Software Engineering (2009)

7. Fuentes, L., Gámez, N.: Configuration Process of a Software Product Line for AmI Middleware. Journal of Universal Computer 16(12), 1592–1611 (2010)
8. Loughran, N., Sanchez, P., Garcia, A., Fuentes, L.: Language Support for Managing Variability in Architectural Models. LNCS, vol. 49, pp. 36–51 (2008)
9. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press, London (1933)
10. Choco Solver Home Page (December 2010), `http://www.emn.fr/z-info/choco-solver/index.html`
11. Stephan, M., Antkiewicz, M.: Ecore.fmp: A Tool for Editing and Instantiating Class Models as Feature Models. Technical Report 2008-08, University of Waterloo (2008)
12. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley Professional, Reading (2003)
13. Zhang, W., Yan, H., Zhao, H., Jin, Z.: A BDD-based approach to verifying clone-enabled feature models' constraints and customization. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 186–199. Springer, Heidelberg (2008)
14. Kim, C.H.P., Czarnecki, K.: Synchronizing cardinality-based feature models and their specializations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 331–348. Springer, Heidelberg (2005)
15. Kuhlemann, M., Batory, D., Apel, S.: Refactoring feature modules. In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791, pp. 106–115. Springer, Heidelberg (2009)
16. Acher, M., Collet, P., Lahire, P., France, R.: Composing feature models. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 62–81. Springer, Heidelberg (2010)
17. White, J., et al.: Automated diagnosis of feature model configurations. Journal of Systems and Software 83(7), 1094–1107 (2010)