Achieving Knowledge Evolution in Dynamic Software Product Lines

Lorena Arcega*†, Jaime Font*†, Øystein Haugen‡, Carlos Cetina*
**San Jorge University, SVIT Research Group, Zaragoza, Spain*
Email: {larcega,jfont,ccetina}@usj.es
†*University of Oslo, Department of Informatics, Oslo, Norway*
‡*Østfold University College, Faculty of Computer Science, Halden, Norway*
Email: oystein.haugen@hiof.no

***Abstract*—Dynamic Software Product Lines (DSPLs) offer a strategy to deal with software changes that need to be handled at run-time. In response to context changes, a DSPL capitalize on knowledge about the architecture variability of the software system to shift between configurations. Similar to any other kind of software, a DSPL needs to evolve over time but current approaches require software engineers to manually perform the DSPL evolution. Our work addresses the evolution of the architecture variability that makes up the knowledge of the DSPL. Given a new version of the architecture variability, we calculate its configuration space and propose strategies that allow migration from the current version to the new version. Our strategy solves the collision of the realization layer resulting from the integration of the new version of the variability specification. We evaluate our dynamic evolution strategy using the Goal-Question-Metric method for a Smart Hotel case study with 2^{39} possible configurations as starting point. Our experiment indicates that the proposed technique would enable automatic evolution in 9 out of 10 cases. In the rest of the cases, all of the DSPL configurations changed between the old and the new version, which frustrates an automatic evolution.**

I. Introduction

Dynamic Software Product Lines (DSPLs) offer a strategy to deal with software changes that need to be handled at run-time. Specifically, DSPLs shift between different configurations triggered by context changes and are driven by means of the architecture variability knowledge. A recent survey [1] reveals that normally the knowledge of a DSPL is formalized by a variability model and an architecture model described in a Domain Specific Language (DSL). The infrastructure that uses this knowledge for the run-time reconfigurations is a MAPE-K loop [2]. DSPLs exist in several domains such as transportation system's production and warehousing environments [3], recommendation systems [4], autonomous navigation in robots [5], environmental monitoring [6], automotive systems [7] and smart homes [8], [9], [10].

Nevertheless, similar to any other kind of software, a DSPL needs to evolve over time. However, the DSPL evolution has some specific characteristics: (1) the co-evolution of the variability model and the architecture model (if the variability model evolves the architecture model must also evolves and vice versa), and (2) the running system has to be available for the interaction with the context throughout the evolution.

Current research efforts in DSPLs propose the evolution of DSPLs that are already deployed by developing and deploying new software bundles that represent alternative implementations [11], [1], [12], [13]. The integration of new bundles into the DSPL has to be performed manually by software engineers. They have to manually inspect and manipulate the specification of variability and architecture and do not guarantee that the system remains available throughout the evolution of the knowledge of the DSPL.

Our work addresses the knowledge evolution of DSPLs. We propose to address the DSPL knowledge evolution by means of the configuration space of a DSPL. Given a new version of the models of a DSPL, we calculate its configuration space and propose strategies that allow migration from the current version to the new version taking into account shared configurations between their configuration spaces.

One problem that can arise during the evolution is the collision between model elements. A collision is when a feature of the DSPL is realized differently in the current version of the DSPL compared with the new version of the DSPL. Our strategy solves the collision of the realization layer resulting from the integration of the variability specification by taking into account each type of collision depending on its nature. We develop a set of rules for solving each type of collision.

We evaluate our DSPL knowledge evolution strategy using the Goal-Question-Metric method through a simulated evolution of a Smart Hotel DSPL. The Smart Hotel presents thirty-nine features in the feature model, and thirteen services, twenty devices and thirty-five channels in the architecture model. That is, the configuration space of the Smart Hotel has 2^{39} potential configurations. The results of our work demonstrate that the proposed strategies complement the current implementations by solving the automatic evolution of the variability knowledge at run-time in 9 out of 10 cases. In 1 out of 10 cases, our strategy cannot evolve the DSPL automatically because the input models differ greatly from each other, that is, they have no common configurations.

The remainder of the paper is structured as follows. In Section 2, we present the models and the implementation of

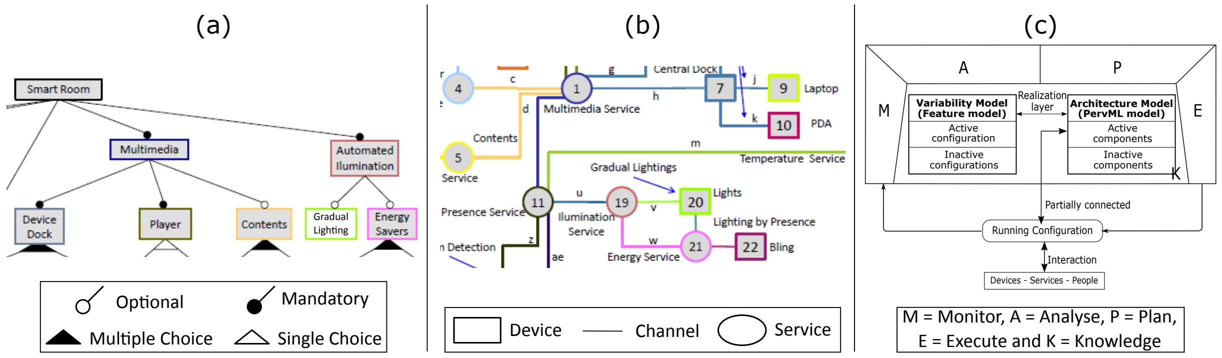

Figure 1. (a) Subset of the Smart Hotel Feature Model, (b) Subset of the PervML model, and (c) MoRE.

the MAPEK loop. In Section 3, we introduce our evolution strategy and the different evolution scenarios. In Section 4, we explain the implementation details of our strategy. In Section 5, we present our evaluation for the Smart Hotel DSPL. In Section 6, we examine the related work, and we present the conclusions in Section 7.

II. Background

DSPLs move existing Software Product Line (SPL) engineering processes to run-time, ensuring that each reconfiguration of the system reach a valid configuration state [1]. Therefore, a DSPL generates a single product which is able to adapt its behaviour at run-time.

Variability modelling, which consist in defining the commonalities and variabilities, is the central activity of SPLs and DSPLs. In a DSPL the variability model describes the variations that can be produced at run-time. The variability model refers to the system architectural components. In DSPLs the system architecture supports all possible configurations defined by the variability model.

The evaluation of this approach is performed through a reconfigurable DSPL for a Smart Hotel [14]. The run-time reconfigurations are performed by an implementation of a MAPE-K loop [2] named Model-based Reconfiguration Engine (MoRE) [14]. Recent surveys reveal that MAPE-K is the most common implementation for reconfiguration loops in DSPLs [1], [15]. This will enable other software engineers to apply these evolution ideas to their MAPE-K DSPL.

A. *Smart Hotel Variability Modeling*

In the Smart Hotel DSPL, the variability model is expressed through a feature model. The architecture model is defined using a DSL. The realization layer defines the connection between the variability model and the architecture model [16], [17]. Finally, the output system is obtained through a model (DSL) to text (Java code) transformation.

Feature models describe the common and variable characteristics of a system [18]. In feature models, features are hierarchically linked in a tree-like structure through variability relationships (optional, mandatory, or single choice), and are optionally connected by cross-tree constraints (requires or excludes). Our feature model represents all of the different features that the Smart Hotel has implemented.

A feature model contains the set of all features (selected or unselected). A Running Configuration (RC) of a system is defined as the set of all selected features in its feature model at a given time. The subset of the Feature Model in Figure 1 shows a small part of the entire Smart Hotel. The grey features represent the running features of the Smart Hotel, while the white features represent potential variants since they may be activated in the future. For instance, the system can potentially be upgraded with a Gradual lighting.

Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely placeholders. We use a weaving model as the realization layer, that is, for mapping features to architecture model elements. The weaving model expresses a link between a feature model and model elements. This weaving approach enables us to configure architecture models from a set of given features.

We use Pervasive Modelling Language (PervML) [19] to describe the Smart Hotel. PervML is a DSL that describes pervasive systems using high-level abstraction concepts based on Meta-Object Facility (MOF) [20]. However, other MOF-based DSLs for other domains can be used equally well with our approach.

Due to space constraints, in this work, we only focus on the subset of PervML that specifies the relationships among devices and services. This subset specifies the components that define a particular system (services and devices) and how these components are connected with each other (channels). Services are depicted by circles, devices are depicted by squares, and the channels connecting services and devices are depicted by lines (see Figure 1 PervML Model).

B. *DSPL Reconfiguration Loop: MoRE (Model-based Reconfiguration Engine)*

Control loops have been identified as essential elements to realize the adaptation of software systems. IBM suggested a reference model for autonomic control loops [2], which is

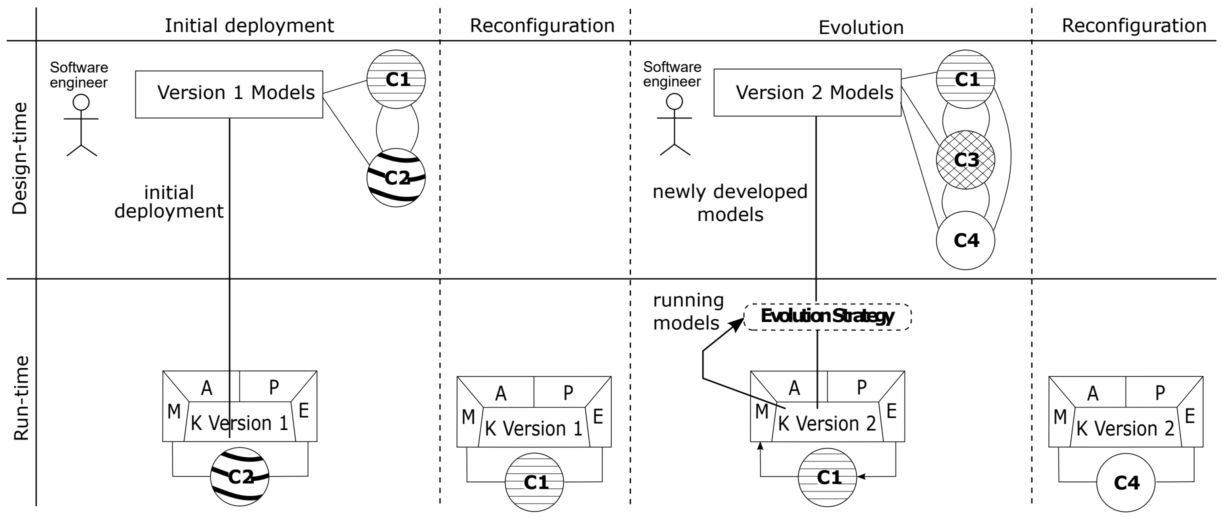

Figure 2. Operations of the DSPL.

called MAPE-K loop. This loop is very useful to researchers that work on run-time variability to make their systems autonomous. The adaptation is based on models (the K element), which means that models are present at run-time [1]. To enable autonomic behaviour, the system must change from one configuration to another by itself. These changes are then translated into reconfiguration actions that modify the system components accordingly.

We use an implementation of the MAPE-K loop called Model-based Reconfiguration Engine (MoRE) [14]. MoRE translates context changes into changes in the activation/deactivation of features. These changes are then translated into the reconfiguration actions that modify the system components accordingly.

In the previous section, we have presented the variability model and the architecture model that MoRE uses as Knowledge (K) to switch between configurations (see Figure 1 (c)). That is, the Smart Hotel DSPL knowledge is composed by the feature model and the PervML architecture model. In MoRE, the Monitor (M) uses the run-time state as input to check context conditions. If any of these conditions are fulfilled, the Analyzer (A) uses the associated resolution and the previous model operations to query the run-time models about the necessary modifications. The response of the models is used by the Planner (P) to elaborate a reconfiguration plan. This plan contains reconfiguration actions, which modify the system architecture and maintain the consistency between the models and the architecture. The Execution (E) of this plan modifies the architecture in order to activate/deactivate the features specified in the resolution.

The reconfiguration of the system is performed by executing reconfiguration actions that deal with the activation and deactivation of components and the creation and destruction of channels among components. For example, the Java virtual machine allows loading and unloading code dynamically and component platforms allow loading, connecting and disconnecting component instances.

The feature model specifies the possible configurations of the system, while the PervML architecture model can be rapidly retargeted to a specific configuration in response to changes in the context. MoRE calculates the architecture increments and decrements in order to determine the actions necessary to modify the system architecture. These operations take a feature resolution as input, and they calculate the modifications to the architecture in terms of devices, services, and channels.

Moreover, it is absolutely necessary to have a way to analyze the reconfigurations before performing them. MoRE validates the configurations resulting from the simultaneous fulfillment of context events at design-time. Therefore, unexpected configurations can be avoided. Specifically, MoRE analyzes variability models by means of the FAMA framework [21] for variability analysis. FAMA framework integrates some of the most commonly used logic representations and solvers proposed in the literature. This framework enables to determine if a system configuration is valid, and it can also provide explanations about invalid configurations.

III. Our Evolution Strategy

Figure 2 shows the operation of the DSPL presented. The upper part depicts the actions performed at design-time by software engineers while the lower part shows the impact that those actions have on the running DSPL.

The first column presents the initial deployment of the DSPL; the first versions of the models are created by the software engineer (the upper part). The initial deployment of the DSPL is performed using those models. The lower part of the first column shows how the DSPL is conformed

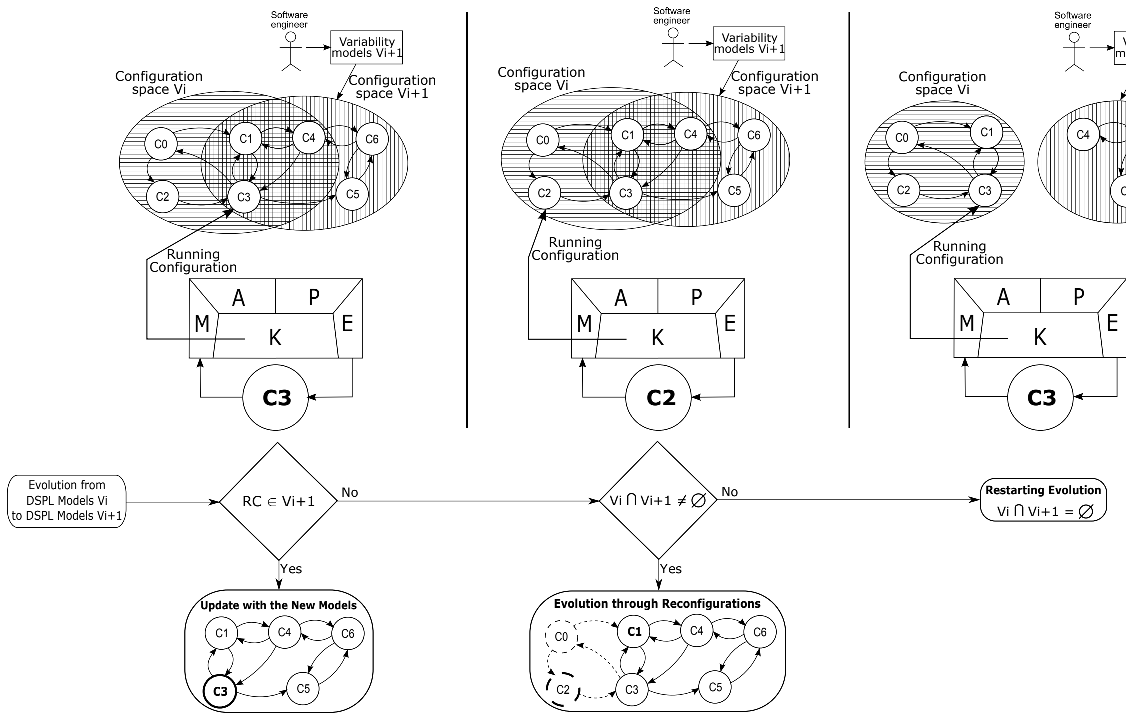

Figure 3. Different Evolution Scenarios.

with the MAPE-K loop. The DSPL can be reconfigured at run-time by hot-swapping existing components.

In the second column of Figure 2, the system is reconfigured and starts using a configuration described by the variability knowledge that is different from the original one (in this case, a shift from Configuration 2 (C2) towards Configuration 1 (C1)). This reconfiguration is performed at run-time; an event from the physical world triggers the use of a new configuration. For example, in the Smart Hotel, C1 is the configuration when a room is empty (i.e., the sensor of the room are used for security purposes). When the client of the hotel enters in the room, MoRE performs the reconfiguration and changes from C1 to C2 that is when the client is in the room (i.e., the sensors of the room are used for illumination purposes).

The third column presents an example of evolution of the DSPL. The software engineer creates the new models (the top part of third column). Our strategy is used to translate those changes to the running DSPL. The current DSPL (Version 1) will be evolved with the new DSPL (Version 2) as indicated by the strategy in order to modify the running DSPL without suspending its execution. That is, some of the configurations are added and some others are removed without stopping the system. When a configuration is modified, we assume that the configuration has been removed and a new configuration has been added.

After the evolution, the system can perform new reconfigurations. In the fourth column of Figure 2, the system is reconfigured from Configuration 1 (C1) to Configuration 4 (C4). The system is able to reach configurations of the new version of the models.

This work focuses on the knowledge evolution of a DSPL. We consider evolution to be the integration of newly developed components without having to stop the system.

We show the evolution by means of the configuration space that is defined by the models. A configuration space is composed of a set of configurations and transitions between configurations. Each of the configurations is composed of a set of active features defined in the variability model. In turn, an active feature is related to a set of architecture model elements by means of a realization layer. The system source code is obtained through a model to text transformation taking as input the complete architecture model.

Our evolution starts with a design-time evolution when the software engineer develops a new configuration space at design-time. In our case, the design-time evolution is the enabler for the DSPL knowledge evolution [22].

Once the evolution at design-time is developed by the software engineer, our strategy allows the run-time activation of the new configurations without having to stop the system. Our strategy distinguishes between three main scenarios.

A. Evolution scenarios

Figure 3 depicts the different evolution scenarios. The software engineer develops a new variability model which defines a new configuration space. We distinguish three scenarios taking into account the current configuration space, the newly developed configuration space, and the running configuration.

In the first case, there is an intersection between the two configuration spaces, that is, some configurations belong to both. In addition, in the first case, the running configuration belongs to this intersection (i.e., the running configuration belongs to both configuration spaces). Hence, the running configuration is also in the new configuration space.

When this case occurs, our strategy performs an **Update**. An update requires the evolution of the model elements that are not involved in the running configuration. Our strategy removes the old configurations and adds the new ones.

The first column in Figure 3 shows an update when the running configuration is C3. Configuration C3 is in version Vi models and in version Vi+1 models; hence, C3 is in the new version. The resultant configuration space only contains the configurations of version Vi+1 (C1, C3, C4, C5, and C6).

In the second case, there is also an intersection between the two configuration spaces. However, the running configuration does not belong to this intersection, that is, the running configuration is not part of the new version of the configuration space. The running configuration is only in the old version of the configuration space.

In this scenario, our strategy performs an **Evolution through Reconfigurations**. Our strategy composes the old configuration space with the new configuration space. The old version is used in a transient period until a configuration of the new version is reached. In other words, the two versions coexist until the running configuration reaches a configuration of the new version, which allows the safe removal of the old version of the configuration space.

The second column in Figure 3 shows the evolution when the running configuration is C2. Configuration C2 is not present in version Vi+1 models. The resultant configuration space is a composition of the old configurations (C0, C1, C2, C3, and C4) with the new configurations (C1, C3, C4, C5, and C6). This is only a transient situation; the final configuration space only maintain the new configurations (C1, C3, C4, C5, and C6).

The third column in Figure 3 shows that there is no intersection between the two configuration spaces. Hence, the strategy cannot reach the new configuration space; there are no transitions between old configurations and new configurations. Therefore, our strategy is not able to perform an automatic evolution without stopping the system. This scenario needs a **Restarting Evolution**.

B. The special case of a bug

This evolution also allows some configurations to be blocked. For example, if a serious bug has been spotted in the running version and some of its configurations should be avoided (also in the transient situation), the software engineer can develop a new configuration space without the configuration that contains the bug.

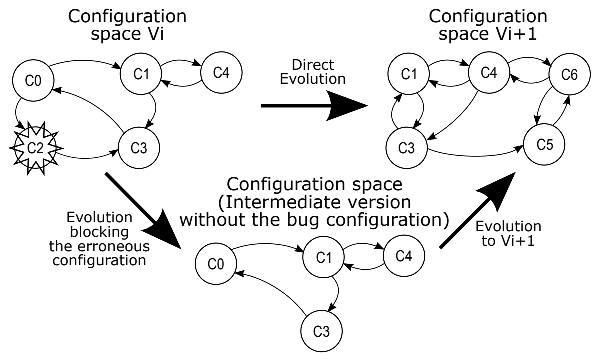
Configuration space Vi
C0
C1
C4
C2
C3
Direct Evolution
Configuration space Vi+1
C1
C4
C6
C3
C5
Evolution blocking the erroneous configuration
Configuration space (Intermediate version without the bug configuration)
C0
C1
C4
C3
Evolution to Vi+1

Figure 4. Example of the Special Case of a Bug.

In the same way as in the evolution scenarios, if the erroneous configuration is not running, our strategy simply performs an **Update** discarding the erroneous configuration. However, if the erroneous configuration is running, our strategy performs an **Evolution through Reconfigurations**. Once the system leaves the erroneous configuration, the system cannot return to it. This is how the system eliminates the bug.

In the example of Figure 4, the software engineer wants to evolve the DSPL from version Vi models to version Vi+1 models. However, there is a bug in configuration C2. Instead of performing the evolution directly from version Vi to version Vi+1, the best way to block the erroneous configuration, C2, is to develop an intermediate version of the configuration space that does not contain this erroneous configuration. Then, the first evolution is from version Vi to the intermediate version. Our strategy applies one kind of evolution or the other depending on the configuration that is running.

Once this intermediate evolution is performed, the configuration that contains the bug has been removed. Thus, the system is prevented from repeatedly reaching the erroneous configuration. Finally, another evolution from the intermediate version to the version Vi+1 is needed to achieve the desired configuration space.

IV. Model Operations to Realize the Evolution Strategy

This section introduces the model operations of our evolution strategy. The evolution starts when the software engineer develops the new version of the models at design-time. These new models define the new configuration space. Then, our strategy evolves the configuration space of the Smart Hotel DSPL. The strategy takes as input the new version of the models developed at run-time and the running models of the Smart Hotel.

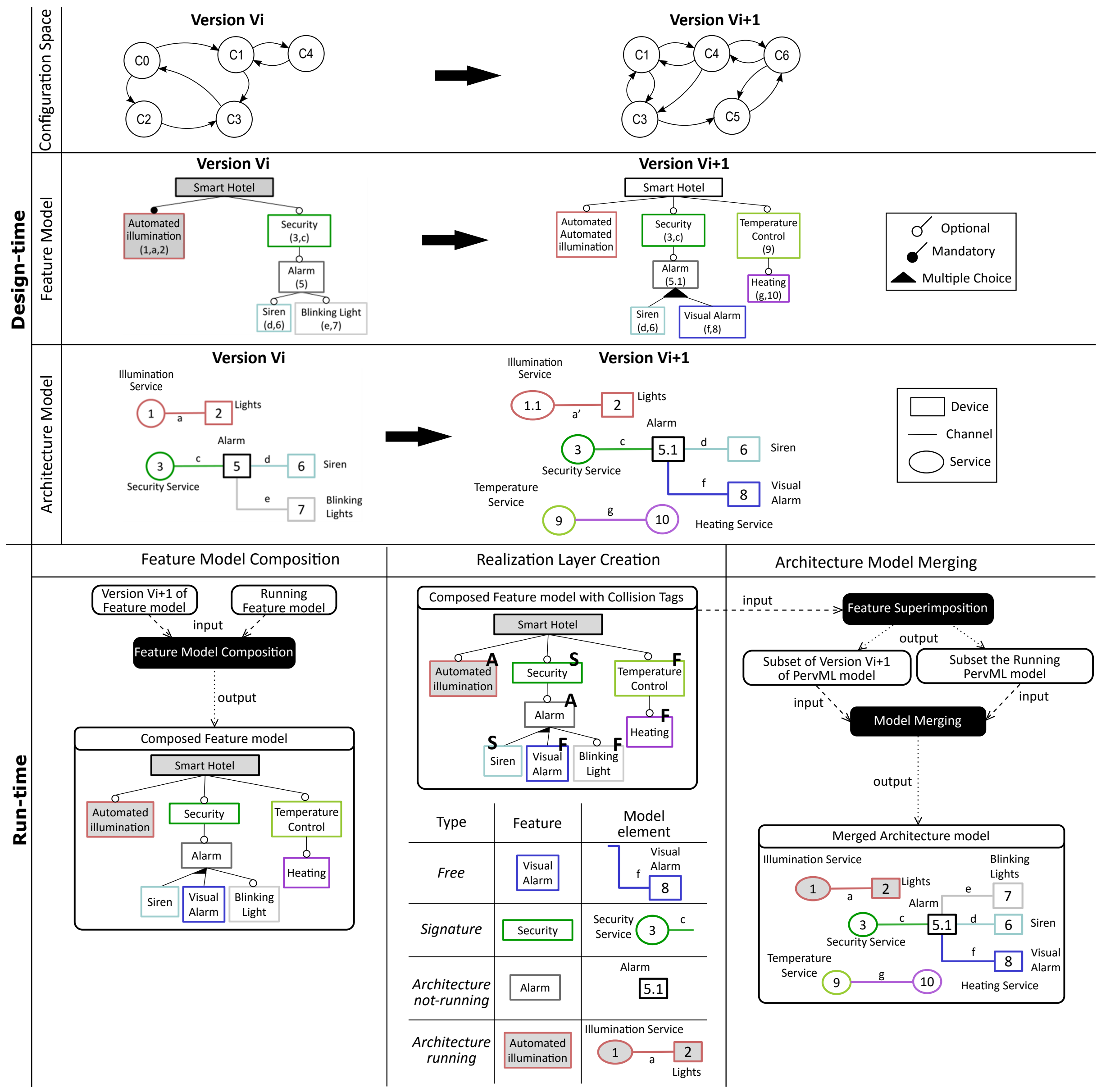

Figure 5. Evolution of the Smart Hotel

A. Design-time Evolution

This section shows the evolution of a Smart Hotel DSPL example performed by the software engineer at design-time. The upper part of Figure 5 depicts the model versions and the evolutions performed at design-time. Figure 5 shows the configuration space, the DSPL Feature Model, and the DSPL PervML Model. The information in brackets after the name of each element in the feature model corresponds to the links to the architecture model.

The upper part of Figure 5 shows the changes performed by the software engineer at design-time over the configuration space, the variability model, and the architecture model. Version Vi shows the first version of the Smart Hotel models. In this small example, the Smart Hotel has two services, automated illumination and security services. The automated illumination feature enables lights to be automatically controlled. The security service is used to alert people of critical notifications such as fire or water leaks in the room. The security service can have a traditional audible alarm or blinking lights.

Version Vi+1 of Figure 5 shows the new models created by software engineers. They create the temperature control service, which takes into account the occupancy of the room and the actual temperature in order to adjust the heating system. They also change the blinking light alarm feature to the visual alarm feature. If any critical situation occurs, the visual alarm shows a message with instructions; therefore, the visual alarm shows necessary information that the blinking lights do not show.

B. DSPL Knowledge Evolution at Run-time

Our strategy distinguishes three scenarios to perform the evolution. In the **Update** evolution, the running models are changed for the new ones. An update requires the evolution of the model elements that are not involved in the running configuration. Our strategy removes the old configurations and adds the new version of the configuration space.

In the **Evolution through Reconfigurations** scenario, our strategy has a transient situation before reaching the final configuration space. The transient situation is formed by a composition of the new models developed at design-time and the running models of the DSPL.

To get the transient situation, our strategy composes the new version of the feature model and the running feature model, and merges the new version of the architecture model and the running architecture model. A new realization layer is created to map each feature with the architecture elements that are related to it. Our strategy implements this composition of the models as indicated below.

1) Feature Model Composition: Our strategy combines the running feature model and the new version of the feature model to produce the new version of the running feature model. This composed feature model specifies configurations that may be reached after the DSPL knowledge evolution. To do the variability composition, our strategy extends the *Reference-based and Slicing* composition process implemented in [23].

The composition process consists of two main phases: (1) the matching phase identifies model elements that describe the same concepts in the input models; (2) the merging phase groups the matched elements together to create new elements in the resulting model. We chose the hybrid implementation, *Reference-Based and Slicing* presented in [23], because it is the most customizable implementation in their study.

The key idea of *Reference-Based* implementation is to build a separate feature model that contains features with the same names as the input feature models. The features are then related to input features through a set of logical constraints. We use *Slicing* to eliminate internal variables, which are needed to perform the composition (because they increase the computational time). The result is a feature model that joins the input variability models and the constraints.

Since conflicts between constraints from different versions of the feature model can occur, our strategy is driven by configuration semantics (the architecture configuration set of the composed variability model is the union of the architecture configuration sets of the input variability models). However, our current implementation of the strategy does not take into account ontological semantics to determine the most appropriate variability hierarchy. Given a set of configurations, there still exist different candidate variability models with other different hierarchies [18].

The Feature Model Composition in Figure 5 depicts an example of the feature model composition of our Smart Hotel example. The inputs of the composition are the running feature model and the version Vi+1 of the feature model developed by the software engineer. The output of the operation is the composed feature model (see Figure 5 lower part).

2) Realization Layer Creation: In order to link the composed variability model to the architecture model, the strategy generates a new realization layer. The realization layer creates a link between features and architecture model elements. In the feature model composition, some collisions between features can occur if the features are not unique. In our case, a feature is not unique when in the composition there is another feature with the same signature. Our strategy only considers the name property of a feature model element as signature. In other words, two features match if they have the same name property.

We follow a set of rules to create the realization layer depending on the collisions among features detected during the matching phase of the feature model composition. A collision depends on the uniqueness of the features. To distinguish the type of collision, we extend our implementation of the feature model composition to add a tag to each feature.

- A feature of type *free (F)* means no collision, the feature is only found in one of the feature models.
- A feature of type *signature (S)* means a collision of the features, where their references to the architecture model are the same. In other words, there are two features (one in each feature model) with the same signature, where the model architecture elements related to them are the same for both features.
- A feature of type *architecture (A)* means a collision of the features and of the architecture model elements related to them (i.e., the architecture elements related to these features are different).

The composed feature model in the realization layer creation column of Figure 5 shows the features and the type of each feature.

The creation of the realization layer follows one rule with each type of feature. For features of type *free (F)* and *signature (S)*, our strategy holds the current architecture elements related to them. This is because there are no collisions in the architecture elements related to the features.

For features of type *architecture (A)*, we differentiate two cases. For active features (involved in the running configuration), our strategy keeps the architecture elements denoted by the run-time variability model before the DSPL knowledge evolution. For inactive features (those not involved in the running configuration), our strategy updates the architecture elements with the new ones that come from the new variability model of the design-time.

The realization layer creation column of Figure 5 shows the different types of features and the model elements associated with each one after creating the realization layer. For example, the illumination service and the alarm have changed in the new version of the architecture model (see Figure 5, upper part). However, since the illumination service is active in the running configuration, the architecture model related to this feature remains unchanged. In contrast, the alarm is changed to the new version because it does not belong to the running configuration; it is inactive.

3) Architecture model merging: Finally, our strategy merges the architecture models that are driven by the realization layer. It performs the superimposition operation ($\odot$) [14] over the variability model. The superimposition takes a feature and returns the set of architecture model elements that are related to this feature. The result is a subset of the running architecture model and another subset of the new version of the architecture model. These two subsets are merged to create the new version of the running architecture model. Our strategy uses a signature-based model composition [24] to achieve the merging.

The model merging is structured in two phases, the matching phase and the merging phase. In the matching phase, model elements that are described in different models are identified. In the merging phase, matched model elements are merged to create a new model element.

To support automated element matching, each element is associated with a signature type that determines the uniqueness of each element. Two elements with equivalent signatures cannot coexist in a model. Our strategy only considers the name property of a model element as signature. In other words, two elements match if they have the same name property.

The merged model contains the union of the model elements in the source models; matching elements are included only once in the merged model. The new version of the running architecture model in Figure 5 shows an example of a merge model.

The superimposition operator ($\odot$) [14] is used to query a realization layer to identify which architecture model elements support a certain feature. The superimposition takes a feature and returns the set of components and channels that are related to this feature. By means of the superimposition operation, it is possible to project a particular feature to the architecture components.

Our strategy performs the superimposition operation taking the composed feature model. This operation returns different model elements to support all possible feature model configurations. These elements are matched and merged as a result of the composed model.

The resultant models are only a transition. Once our reconfiguration engine reaches a new configuration, our strategy removes the old elements. The final models only contain elements of the new version. Therefore, the system can only reach configurations belonging to the new version.

V. Validation of the DSPL knowledge evolution of the Smart Hotel

In this section we aim to show the applicability of our evolution strategy, by showing that it can support in practice the evolution of a Smart Hotel DSPL. We do not focus on computational complexity or scalability because these are properties that emerge from the choice of expressive power of the language used to express the variability and the choice of the reconfiguration loop implementation.

We conducted a validation of our strategy using a Smart Hotel case study. The Smart Hotel used in the previous sections to explain our strategy is a subset of the real Smart Hotel that we used in this validation. This Smart Hotel was developed previously [14]. The Smart Hotel reconfigures its services according to changes in the surrounding context. A hotel room changes its features depending on users activities to make their stay as pleasant as possible.

According to the feature model, the Smart Hotel presents thirty-nine features and six cross-tree constraints. The main concepts of the Smart Hotel DSPL architecture are services, devices, and the communication channels among them. The Smart Hotel has thirteen services, twenty devices and thirty-five channels. That is, the configuration space of the Smart Hotel has 2^{39} possible configurations.

We defined the experimental design of our study using the Goal-Question-Metric method (GQM) [25] and its template [26]. The GQM method was defined as a mechanism for defining and interpreting a set of operation goals using measurements. In this experiment, our goal was the following:

- Object : Our Smart Hotel DSPL
- Purpose: Validation
- Issue: The applicability of the automatic evolution strategy
- Context: Evolution of the DSPL architecture variability knowledge

To fulfill this goal, we focused on answering this research question: Do the scenarios of our strategy cover the evolution of the Smart Hotel DSPLs?

Basili [25] and Travassos [27] describe four kinds of studies: in-vivo, in-vitro, in-virtuo, and in-silico. In our case, we chose to carry out in-silico experiments, where subjects and the real world are described as computer models. The environment was composed entirely of computer models, with which human interaction was reduced to a minimum. This offers major advantages regarding the cost and the feasibility of replicating a real-world configuration. In addition, some scenarios such as fires or floods cannot be replicated in the real world.

Moreover, we simulated the evolution by means of a simulation approach for exploring the effects of product line

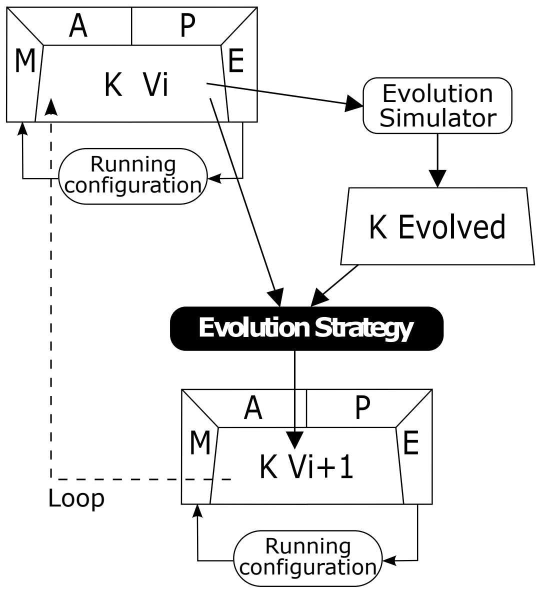

Figure 6. Evaluation Steps.

evolution [28]. The simulator tool allows generating random models based on product line profiles.

The evolution profiles are *Product line refactoring* that leads to the adaptation of existing components or the development of new ones, *Product placement* that changes the problem space to allow different configurations of the existing components, without changing them, and *Continuous evolution* that is subject to continuous changes in both spaces by adding/removing components or reorganizing decisions.

We implemented the above evolution profiles in a Evolution Simulator. This Evolution Simulator is based on Ecore-mutator [29], which is an EMF-based framework to mutate EMF models that conform to an Ecore metamodel. In particular, we implemented custom mutators that, given a model as input, add, remove or modify model elements according to each evolution profile. Our Evolution Simulator randomly chooses which evolution profile of the ones presented in this section is going to be applied.

Figure 6 shows the steps followed in this evaluation. The Evolution Simulator takes one version of the models as input and returns an evolved version of these models. Next, our evolution strategy performs the DSPL knowledge evolution. The input for the strategy was the running models and the new version of the evolved models developed in the Evolution Simulator. The output of the strategy is an evolved DSPL knowledge placed on the reconfigurable system. We performed simulations until all evolution profiles were covered.

For example, in the first evolution, the Evolution Simulator took as input the first version of the Smart Hotel DSPL (2^{39} configurations). The new version of these models (output of Evolution Simulator) had forty-two features in the feature model, fifteen services, twenty-four devices, and forty-five channels in the architecture model and 2^{42} potential configurations in the configuration space. This evolution profile corresponded to a *Continuous evolution* because five new configurations were added, two old configurations were removed and six of the old ones were modified.

Next, our evolution strategy evolved the models from the Smart Hotel DSPL. Our strategy took as input the new version of the models. As the running configuration was not in the configuration space of the new version of the models, the evolution was performed by an **Evolution through Reconfigurations**. Finally, the Smart Hotel DSPL could reach 2^{42} different configurations with the new version of the models instead of the 2^{39} configurations that the Smart Hotel DSPL could reach with the previous models.

A. Scenarios covered by the strategy

After analysing the data obtained from the evolutions, we obtain that in 1 out of 4 cases of the evolutions, the running configuration remains in the configuration space of the new version of the models. The strategy can perform a direct **Update** scenario without interrupting the running configuration.

In 2 out of 3 cases of the evolutions, the running configuration does not belong to the configuration space of the new version of the models. The strategy is able to find a way from the running configuration to reach a configuration that belongs to the configuration space of the new version of the models. Hence, the strategy performs an **Evolution through Reconfigurations** scenario.

However, in 1 out of 10 cases of the evolutions, the strategy cannot find a way to reach a configuration that belongs to the configuration space of the new version of the models. The system cannot be evolved with our automatic strategy and must be stopped to perform the evolution.

We have observed that the models corresponding to this 1 out of 10 cases that cannot be evolved automatically are models that differ greatly from each other. The **Restarting** scenario is required when none of the old configurations are retained in the new version of the configuration space. In a no simulated environment, the evolution of a system in which all of the configurations change without keeping at least one old configuration in the new version is very uncommon. Although it has occurred in an exhaustive simulation, it means not having in common any of those 2^{39} configurations between the old and the new version.

B. Threats to Validity

Our validation exhibits some applicability. Regarding the generalization, our results and findings are based on a single DSPL in the domain of the Smart Hotels. However, given the scale and complexity of our Smart Hotel (thirty-nine features, thirteen services, twenty devices, and thirty-five channels in the PervML model and 2^{39} possible configurations in the configuration space), we consider our validation a good starting point representing a realistic case.

Furthermore, the election of the Smart Hotel DSPL for the validation was motivated because the Smart Hotel adheres to the the core ideas of DSPLs such as MAPE-K loop and architecture variability knowledge [1]. Although, the evolution strategy was validated by means of the Smart hotel, the ideas of the evolution strategy are general and they can be applied to other DSLPs which are base on MAPE-K loop and architecture variability knowledge.

Concerning the evaluation runs, our validation may not seem sufficient for a continuously operating system. However, each run covers one complete continuous life-cycle of the system.

Finally, to ensure the validity of the case study, the validation was done by a student in his last year of his master as part of his master's thesis. The participation of the authors of the strategy was limited to explaining the operation of the strategy so that the student could correctly perform the validation. This student was responsible for developing the Evolution Simulator and for conducting the validation with the Smart Hotel case study. Thus, we have achieved that the validation was independent of the main research.

VI. Related Work

Hallsteinsen et al. [11] develop the MUSIC framework. MUSIC supports the dynamic addition and removal of components and service variants, as well as compositions with their own set of model fragments that describe internal variability. Their evolution is focused on software by means of developing new software bundles. However, their evolution must be performed manually by a software engineer, while our evolution strategy allows an automatic evolution.

Pascual et al. [30] present an approach that provides support for the dynamic reconfiguration of mobile applications, optimizing their architectural configuration according to the available resources. They model the variability of the application's software architecture using Common Variability Language (CVL) [16] while we use a feature model and a Domain Specific Language (DSL) to express the variability. They evolution is performed thought a genetic algorithm whilst we develop different operation (i.e., feature model composition or model merging) for each one of the models.

Perrouin et al. [31] use a MAPE loop to manage another MAPE loop. Their approach not only adapts the system, it also adapts both its adaptation mechanism and its adaptation policies. They define a dynamically reconfigurable adaptation loop. The dynamic reconfiguration of this adaptation loop is achieved by employing adaptation techniques that are similar to the ones used to adapt the system itself. They focus on evolving adaptation rules or reconfiguration scripts, while we focus our work on DSPL knowledge evolution that incorporates knowledge from new versions of design-time models into run-time models. The combination of the two approaches may improve current DSPL implementations.

Hussein et al. [32] develop an approach to enable the run-time evolution of context-aware adaptive services. Their approach captures a service's model from three aspects: functionality, context, and adaptive behaviour. Thus, these aspects and their relationships can be captured and manipulated at run-time. With this approach, the software engineer can perform the service's run-time changes at the modelling level. However, they compute the differences between the evolved model and the initial model and generate adaptation actions. Then, this actions are applied to the service's run-time artifacts. We take into account the run-time model and the run-time configuration to apply our strategy, hence we only need to perform adaptation action when the running configuration does not allow the evolution.

Capilla et al. [15] provides an overview of the state of the art and current techniques that face the challenges of run-time variability in the context of Dynamic Software Product Lines. They propose a solution for the automation of changes in the structural variability (i.e., adding or removing features at run-time). They use the notion of super-types [33], [34] while we use the feature model composition. In addition, they propose some techniques to check the feature model and the configuration model [21], [35], [36] once they are modified. These techniques could be applied to our strategy in the future to ensure that the evolved models are valid.

VII. Conclusions

This work addresses the knowledge evolution of DSPLs by means of the configuration space of a DSPL. Specifically, our evolution strategy distinguishes three main scenarios taking into account the running configuration. In addition, our strategy solves the collision between components resulting from the evolution. Finally, the system evolves automatically thus enabling the DSPL to reach new configurations.

The proposed automatic evolution strategy is not restricted to the case study we have chosen to evaluate, it can also be applied to a wide range of DSPL domains. The ideas of the evolution strategy can be applied to the most common infrastructure of DSLPs [1], which combines MAPE-K loop and architecture variability knowledge.

In this paper, we have validated the benefits of performing the evolution automatically. The validation of our strategy in the Smart Hotel DSPL has shown promising results in 9 out of 10 cases, which confirm the potential of applying our automatic evolution strategy.

In the near future, we would like to explore automatic evolution in other kinds of systems with other types of variability management, such as systems that use the Common Variability Language (CVL) [16]. We plan to investigate an automatic evolution that covers the maximum number of possible cases.

References

[1] N. Bencomo, S. O. Hallsteinsen, and E. S. de Almeida, "A view of the dynamic software product line landscape," *IEEE Computer*, vol. 45, no. 10, pp. 36–41, Oct 2012.

[2] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2006.

[3] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. V. Betsbrugge, "Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines," in *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL '09)*, San Francisco, California, USA, Aug 2009, pp. 19–27.

[4] G. H. Alfrez and V. Pelechano, "Context-aware autonomous web services in software product lines," in *Proceedings of the 15th International Software Product Line Conference (SPLC '11)*, Munich, Germany, Aug 2011, pp. 100–109.

[5] M. Kim, J. Kim, and S. Park, "Tool support for quality evaluation and feature selection to achieve dynamic quality requirements change in product lines," in *Proceedings of the 2nd International Workshop on Dynamic Software Product Lines (DSPL '08)*, Limerick, Ireland, Sep 2008, pp. 69–78.

[6] S. Hallsteinsen, S. Jiang, and R. Sanders, "Dynamic software product lines in service oriented computing," in *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL '09)*, San Francisco, California, USA, Aug 2009, pp. 28–34.

[7] H. Shokry and M. A. Babar, "Dynamic software product line architectures using service-based computing for automotive systems," in *Proceedings of the 2nd International Workshop on Dynamic Software Product Lines (DSPL '08)*, Limerick, Ireland, Sep 2008, pp. 53–58.

[8] R. Ali, R. Chitchyan, and P. Giorgini, "Context for goal-level product line derivation," in *Proceedings of the 3rd International Workshop on Dynamic Software Product Lines (DSPL '09)*, San Francisco, California, USA, Aug 2009, pp. 8–17.

[9] J. Lee, J. Whittle, and O. Storz, "Bio-inspired mechanisms for coordinating multiple instances of a service feature in dynamic software product lines," *The Journal of Universal Computer Science*, vol. 17, no. 5, pp. 670–683, 2011.

[10] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Designing and prototyping dynamic software product lines: Techniques and guidelines," in *Proceedings of the 14th International Software Product Line Conference (SPLC '10)*, Jeju Island, South Korea, Sep 2010, pp. 331–345.

[11] S. O. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. A. Papadopoulos, "A development framework and methodology for self-adapting applications in ubiquitous computing environments," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2840–2859, Dec 2012.

[12] C. Cetina, P. Giner, J. Fons, and V. Pelechano, "Autonomic computing through reuse of variability models at runtime: The case of smart homes," *IEEE Computer*, vol. 42, no. 10, pp. 37–43, Oct 2009.

[13] M. Hinchey, S. Park, and K. Schmid, "Building dynamic software product lines," *IEEE Computer*, vol. 45, no. 10, pp. 22–26, Oct 2012.

[14] C. Cetina, "Achieving autonomic computing through the use of variability models at run-time," Ph.D. dissertation, Universidad Politcnica de Valencia, 2010.

[15] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey, "An overview of dynamic software product line architectures and techniques: Observations from research and industry," *Journal of Systems and Software*, vol. 91, pp. 3–23, May 2014.

[16] Ø. Haugen, B. Mller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Proceedings of the 12th International Software Product Line Conference (SPLC '08)*, Limerick, Ireland, Sep 2008, pp. 139–148.

[17] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, y. Haugen, B. Mller-Pedersen, and G. K. Olsen, "Developing a software product line for train control: A case study of cvl," in *Proceedings of the 14th International Conference on Software Product Lines (SPLC '10)*, Jeju Island, South Korea, Sep 2010, pp. 106–120.

[18] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, Waikiki, Honolulu , HI, USA, May 2011, pp. 461–470.

[19] J. Muoz, "Model driven development of pervasive systems. building a software factory," Ph.D. dissertation, Universidad Politcnica de Valencia, 2008.

[20] "Meta object facility (mof), 2.0 core specification," 2003, version 2.

[21] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE '05)*, Porto, Portugal, Jun 2005, pp. 491–503.

[22] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change: Research articles," *Journal of Software Maintenance and Evolution: Research and Practice - Unanticipated Software Evolution*, vol. 17, no. 5, pp. 309–332, Sep 2003.

[23] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. France, "Composing your compositions of variability models," in *Model-Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schtz, J. Gray, A. Vallecillo, and P. Clarke, Eds. Springer Berlin Heidelberg, 2013, vol. 8107, pp. 352–369.

[24] R. B. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh, "Providing support for model composition in metamodels," in *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC '07)*, Annapolis, Maryland, USA, Oct 2007, pp. 253–266.

[25] V. R. Basili, “The role of experimentation in software engineering: Past, current, and future,” Berlin, Germany, Mar 1996, pp. 442–449.

[26] V. R. Basili, G. Caldiera, and H. D. Rombach, “The goal question metric approach,” in *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[27] G. H. Travassos and M. de Oliveira Barros, “Contributions of in virtuo and in silico experiments for the future of empirical studies in software engineering,” in *Proceedings of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering (WSESE ’03)*, Roman Castles, Italy, Sep 2003.

[28] W. Heider, R. Froschauer, P. Grnbacher, R. Rabiser, and D. Dhungana, “Simulating evolution in model-based product line engineering,” *Information and Software Technology*, vol. 52, no. 7, pp. 758–769, Jul 2010.

[29] “Eclipse foundation, ecore-mutator.” [Online]. Available: https://code.google.com/a/eclipselabs.org/p/ecore-mutator/

[30] G. G. Pascual, R. E. Lopez-Herrejon, M. Pinto, L. Fuentes, and A. Egyed, “Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications,” *Journal of Systems and Software*, vol. 103, pp. 392–411, May 2015.

[31] G. Perrouin, B. Morin, F. Chauvel, F. Fleurey, J. Klein, Y. Le Traon, O. Barais, and J. M. Jezequel, “Towards flexible evolution of dynamically adaptive systems,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE ’12)*, Zurich, Switzerland, Jun 2012, pp. 1353–1356.

[32] M. Hussein, J. Han, J. Yu, and A. Colman, “Enabling runtime evolution of context-aware adaptive services,” in *Proceedings of the IEEE 10th International Conference on Services Computing (SCC ’13)*, Santa Clara, CA, Jun - Jul 2013, pp. 248–255.

[33] O. Ortiz, A. B. García, R. Capilla, J. Bosch, and M. Hinchey, “Runtime variability for dynamic reconfiguration in wireless sensor network product lines,” in *Proceedings of the 6th International Workshop on Dynamic Software Product Lines (DSPL ’12)*, Salvador, Brazil, Sep 2012, pp. 143–150.

[34] J. Bosch and R. Capilla, “Dynamic variability in software-intensive embedded system families,” *IEE Computer*, vol. 45, no. 10, pp. 28–35, Oct 2012.

[35] J. White, D. Benavides, D. C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes, “Automated diagnosis of feature model configurations,” *Journal of Systems and Software*, vol. 83, no. 7, pp. 1094–1107, Jul 2010, {SPLC} 2008.

[36] P. Trinidad, “Automating the analysis of stateful feature models,” Ph.D. dissertation, Universidad de Sevilla, 2012.