Abstract

More and more approaches propose to use Software Product Lines (SPLs) modelling techniques to implement dynamic adaptive systems. The resulting Dynamic Software Product Lines (DSPLs) present new challenges since the variability transformations used to derive alternative configurations have to be intensively used at runtime. This paper proposes to use the Common Variability Language (CVL) for modelling runtime variability and evaluates a set of alternative strategies for implementing the associated variability transformations. All the proposed strategies have been implemented and evaluated on the case-study of a smart-home system. Results show that the proposed strategies provide the same reconfiguration service with significant differences in quality-of-service.

1 Introduction

Runtime adaptation is becoming a key requirement in many modern software applications ranging from large-scale systems of systems to dedicated mobile applications. Such Dynamic Software Product Lines (DSPLs) must intensively use variability transformations at run-time in order to adapt their configuration to a changing context and environment. Examples of DSPL range from large crisis management systems to interactive entertainment applications. In this paper, the motivating example and case study we use is a smart-home application. The smart-home application includes a variety of sensors such as light sensors, temperature sensors or presence sensors and controls a set of devices such as heaters, lights, a TV and so on. While it is running, the smart-home system has to be able to accommodate with the connection or disconnection of devices and sensors and update its behaviours accordingly.

All modern middleware platforms and programming languages provide mechanisms to implement runtime variability. For example, the Java virtual machine allows loading and unloading code dynamically and components platforms allow loading, connecting and disconnecting component instances. However, these mechanisms are platform-specific and very low-level as they allow for any kind of changes in the running application. Implementing runtime adaptation at this level consists of writing ad-hoc scripts to program the adaptation policy which remains possible for small applications but becomes tedious and error-prone as the number of configurations for the system grows. To overcome this problem, several approaches [5, 18] have proposed the idea of modelling DSPLs early in the development cycle and to use model-driven techniques to derive their adaptation logic. Modelling a DSPL presents three major challenges:

- Modelling the variability. The problem is similar to variability modelling in product lines except for the variability transformation part which has to be dynamic instead of static. This implies that the variability transformation process has to be fully automated. Existing variability modelling techniques can be applied if they include all necessary links to the model of the application in order to automatically derive alternative models for the application. These techniques include approaches based on feature-models [6] which use model annotations in the application base model and the Common Variability Language (CVL) [13] which keeps the variability model separated from the base model and expressed the variability by modelling the differences between alternative configurations.

- Modelling the runtime adaptation policies, i.e. which configuration should be used and when should the system adapt. Several approaches have been proposed in the literature to express adaptation policies ranging from simple guard/action rule system to complex AI optimization algorithms. Finding the optimal formalism for modelling adaptation policies remains an open research question but is out of the scope of this paper.
The safe and efficient migration of the system from one configuration to another. At runtime, the variability transformation not only has to produce a plain model of the configuration to run but it has to carry the migration from the currently running configuration to the new one. This is a major difference with typical static variability transformations. The reconfiguration has to be efficient in order to avoid perturbations in the performances of the application and it has to be safe, i.e. avoid loss of data and should not disturb the actual services.

In this paper we propose and evaluate three alternative strategies for runtime variability transformations, i.e. for the migration of an adaptive application from one configuration to another. These strategies all ensure the same re-configuration service but have different extra-functional properties: for example they do not offer the same performances or they do not offer the same history capabilities. These strategies have been implemented on top of the CVL and evaluated on the smart-home case study. Results show that in different situations the proposed variability transformation strategies offer valuable quality-of-service trade-offs. The contribution of the paper is two-fold. Firstly we propose a technique for implementing runtime adaptation based on the common variability language. Secondly we propose and compare a set of alternative variability transformations strategies.

The paper is organized as follows. Section 2 presents an overview of the CVL. Section 3 presents the smart-home example and shows how CVL can be used to model runtime variability. Section 4 proposes alternative strategies for runtime variability transformations. Section 5 presents how these strategies were implemented and validated. Section 6 details the comparison criteria used to evaluate the proposed strategies. Section 7 presents the comparison of these strategies on the smart-home case study. Finally Section 8 provides an overview of related work and section 9 concludes.

## 2 Background

### 2.1 Common Variability Language (CVL) Approach

Haugen et al. [13] propose adding variability handling to Domain Specific Languages (DSL). The motivation of the approach is to facilitate SPL development by separating variability modeling from the base domain modeling. They define Common Variability Language (CVL) for modeling variability of models in any base language such as DSL or UML. The CVL approach leaves the base domain modeling to various DSLs while the variability is treated with CVL, as depicted in Figure 1.

As illustrated in Figure 1, a CVL model consists of one variation model and several resolution models that are applied to one base model. The variation model defines how base model elements can vary, and several variation models can be defined for a base model. Also several resolution models can be associated with a single variation model while each resolution model describes a variant of the base model, which is produced by executing the CVL description.

### 2.2 CVL Concepts

In CVL, each variation element is connected to some base model elements and provides a set of alternatives to these elements correspondingly. Thus the basic variation elements in CVL are substitutions. CVL includes constructs to compose the following three types of substitutions according to the type of alternatives it specifies:

- Value substitutions, which vary the value of the attributes of the base model elements.
- Reference substitutions, which vary the references of the base model elements.
- Fragment substitutions, which vary any arbitrary structure in the base model.

Since a fragment can be any arbitrary part of the base model including a set of elements and their relationships, the fragment substitutions are the most general and adopted substitutions in CVL variation models.

CVL resolution models are composed of choices from the substitution alternatives to configure a new version of the base model conforming to the variation model.
The nature of CVL transformations are model-to-model transformations that results in the resolved domain models conforming to related base model, variation models and resolution models, as illustrated in Figure 1. Therefore, CVL, in this sense, is capable of automating the production and configuration of the final software, which makes it a suitable technology for domain variability transformation.

3 Variability Transformation Using CVL

In this section we illustrate how CVL handles variability transformation in the domain of Smart Homes.

3.1 Smart Homes

Smart Homes consist of a number of physical peripherals, such as sensors (i.e. presence or light intensity), actuators (i.e. lamp or alarm) and multimedia devices (i.e. digiframe or set-top box). These devices are controlled and coordinated by higher level services to perform specific tasks in the context of Smart Homes. For instance, to realize presence simulation service, the service needs to coordinate lights, multimedia players, the TV and blinds in order to act as if there were people at home.

Smart Home services are subject to frequent reconfigurations since: (1) It is typical to add/remove devices into/from Smart Home environment. For instance, after the user gets an iPod, he/she can reconfigure the multimedia service at home in order to allow accessing the music library or playing music through the home speakers. (2) Context condition changing also leads to service reconfigurations, i.e. when the user leaves home, services at home should be reconfigured to prioritize security concerns, which we will elaborate more in the following example.

3.2 Smart Homes and CVL

As stated above, Smart Homes are a highly dynamic and variable domain since neither its physical devices nor high level services keep static throughout their lifecycle.

Assume the following scenario transition: as shown in the left of Figure 2, when the user is at home, presence sensors are deployed by lighting service via channels a and b; as shown in the right of Figure 2, when the user leaves home, the control of presence sensors are taken over by security service for intrusion detection (communication channels a and b are disabled, channels e and f are established). Also the lighting service is reconfigured to coordinate with multimedia service to realize presence simulation (channels c and d are established), such that the multimedia devices and lights will be on to act as if there is someone at home.

The configurations for each scenario in Figure 2 are defined by PervML DSL. We apply CVL to transform the first scenario configuration to the second one. In order to get presence sensors connected to security service instead of lighting service, two substitutions are needed: the channel a needs to be replaced by e, while channel b replaced by f. Also to add presence simulation service connected to multimedia service and lighting service, a fragment substitution can be applied. Assume that an empty fragment is connected to multimedia service and lighting service when the user is at home. By replacing the empty fragment with a fragment consisting of one model element (presence simulation) and channels references to multimedia service and lighting service (channel c and d), we can get the configuration of presence simulation and related services when the user leaves home. The CVL description that includes these substitutions is then executed using model transformation engine and results in a resolved model as the reconfiguration in the second scenario.

http://www.pros.upv.es/labs/projects/pervml
4 Strategies for Variability Transformation using CVL

The Variability transformation of CVL defines the updates of a base model from one configuration to a new one as illustrated in the Smart Home example. This Variability transformation takes as input a high level specification of the new configuration (namely Resolution according to the CVL terminology) and then it performs two operations:

1. **Synthesis.** This operation generates a new Base-model configuration that fulfills the given Resolution.

2. **Modifications.** This operation calculates the differences between the previous Base-model configuration and the new Base-model configuration.

Figure 3 shows this reconfiguration process using the State machines notation. The states represent different configurations of a Base-model. While the transitions indicate the possibility of Base-model reconfiguration. This reconfiguration is started by means of a Resolution (trigger of the transition), which specifies the Replacements for each Placement. Given this Resolution as input, CVL calculates the effects of the reconfiguration in terms of model increments/decrements (Modifications operation). Finally, the transition leads to a state of which Base-model configuration is calculated by the Synthesis operation.

Incrementally, more approaches apply SPLs to build runtime adaptive systems [12, 5, 18]. Although the details are different, these approaches share that they perform the variability transformation intensively at run-time. Furthermore, managing variability at run-time stresses concerns such as performance or reconfiguration debugging. We argue that the variability transformation can be realized by means of different strategies. These strategies implement the same functionality (synthesis and modifications) but they have different extra-functional properties. For example, they do not offer the same performances. These strategies enable SPL engineers to use the most suitable strategy for each concern. In particular, we have implemented three different strategies as follows:

- **Regenerative** (REG). Overall, the strategy takes as input a Resolution and it makes a copy of the Base-model which is updated to conform the given Resolution. Figure 4 shows the operations of the REG strategy graphically. In detail, the synthesis operation is implemented as follows. Given a Resolution (in terms of CVL Fragment Resolutions), first the REG strategy creates a copy of the Base-model. Then the strategy iterates all the Fragment Resolutions. Each Fragment Resolution indicates the Replacement Fragment of a Fragment Substitution in the Var-model. For each Fragment Substitution the strategy updates the copy of the Base-model. Those elements referenced by a Placement are deleted, and those elements referenced by a replacement fragment are copied from the Library to the copy of the Base-model. Once all the Fragment Substitutions have been processed, the updated copy of the Base-model is conforming to the given Resolution. To implement the modifications operation, the REG strategy calculates the model difference between the new Base-model and the previous Base-model.

- **Incremental - Copy** (INC-C). This strategy modifies the Base-model of the CVL specification to implement the synthesis and modifications operations. That is, the strategy does not make a copy of the Base-model. All the required modifications are directly applied to the Base-model of the CVL specification. Figure 5 shows the INC-C strategy graphically. The synthesis operation is implemented as follows. Given a Resolution, the INC-C strategy iterates all the Fragment Resolutions. For each Resolution the strategy updates the Base-model. Those elements referenced by a Placement are deleted, and those elements referenced by a Replacement Fragment are copied from the Library to the Base-model. Finally the up-
To implement the modifications operation, the strategy iterates all the Fragments of the Resolution. Those elements referenced by a Placement (which should be deleted from the Base-model) are copied to a list of decrements, and those elements referenced by a Replacement (which should be copied from the Library to the Base-model) are copied to a list of increments.

- **Incremental - Move** (INC-M). This strategy modifies both the Base-model and the Library of the CVL specification. The Library is updated because the model fragments are not removed from the CVL specification, instead, they are moved from the Base-model to the Library. Figure 6 shows the INC-M strategy graphically.

The synthesis operation is implemented as follows. Those elements referenced by a Replacement Fragment are moved from the Library to the Base-model and those elements referenced by a Placement are moved from the Base-model to the Library. Therefore, changes performed to the elements of the Base-model are not discarded by reconfigurations, because model changes are saved in the Library.

To implement the modifications operation, the strategy iterates all the Fragments of the Resolution. Those elements referenced by a Placement Fragment (which should be moved from the Base-model to the library) are copied to a list of decrements, and those elements referenced by a Replacement Fragment (which should be moved from the Library to the Base-model) are copied to a list of increments.

We have implement these strategies by means of the Model Query project of the Eclipse Modelling Framework [1], but the strategies as such can be implemented with other technologies such as ATLAS Transformation Language [15] or MOFScript [2]. In a previous work [13], we detailed the implementation of the variability transformation following a REG strategy by means of MOFScript.

Next section shows how we have validated the implementation of the above strategies. Then, we describe the extra-functional properties of each strategy and we also give recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.

## 5 Validating the Strategies Implementation

Given a Resolution, we have three different strategies to calculate the same operations (synthesis and modifications). We argue that simultaneously comparing the outputs of the strategies enables the validation of the strategy implementations.

Our approach tests for equality the operation results of the strategies. In our case, equality means: (1) all the strategies got the same model modifications for each reconfiguration and (2) there are no differences between the resulting base models.

To systematize the process, we perform the testing throughout the Possibility Space of a CVL specification. This Possibility Space is the representation of all the feasible configurations according to the CVL Specification. Top of Figure 7 shows a simple CVL specification and bottom of Figure 7 shows the Possibility Space using the State Machines Notation. States represent configurations and transitions represent reconfigurations as introduced in the previous section.

Our approach for validating the strategies implementation is a three steps process. First, from a CVL specification we calculate the skeleton of the Possibility Space. This skeleton is conformed by the empty states of the state machine and the transitions with their triggers (Resolutions). An empty state means that the Base-model associated to this state is not calculated yet.

Second, the Base-model associated to each state and the effects of the transitions are calculated by means of the strategy operations (synthesis and modifications). For each reconfiguration (transition), the strategies take the same Resolution as input (transition trigger) and they calculate the
Finally, our approach compares the model modifications and the Base-models among strategies in order to check their equality. We recommend this comparison among strategies when we have at least one reliable strategy and we are implementing new strategies. The Base-model comparison can detect differences between new implementations and a reliable implementation.

Furthermore, some states of the Possibility Space can be reached through different paths (see bottom of Figure 7). Independently of the followed path, all the strategies must generate the same Base-model. Our approach also compares the Base-models among paths in order to check their equality. In fact, this last comparison can be performed by means of only one strategy.

We recommend this comparison among paths when we do not have a reliable strategy yet. This comparison helps to refine the implementation of a strategy until the inconsistencies among paths have been eliminated.

The combination of these comparisons (among strategies and among paths) throughout a Possibility Space turns out to be a powerful tool to verify the implementation of strategies. We have applied this approach to verify the three strategies presented in the previous section. Furthermore, the approach enables us to validate the implementation of new strategies.

5.1 Tool Support for Testing Strategies

Calculating the skeleton of the Possibility Space and then executing and comparing the different strategies are tedious tasks. We have developed a tool to automate this process. Figure 8 shows the Testing tool for CVL Strategies, which is integrated with the CVL editor. This tool is structured in three tabs: Possibility Space, Strategies Management and Strategies Comparison.

In the Possibility Space tab (see left of Figure 8), the testing tool calculates the skeleton of the Possibility Space. To calculate this skeleton, the tool takes as input a CVL specification (see top of Figure 8). Then, the tool calculates all feasible Resolutions according to the VAR-model. These Resolutions are valid assignments of Replacement Fragments to Placements. For each Resolution the tool creates an empty state in the Possibility Space. Empty state means that the associated Base-model Configuration will be calculated later by the strategies.

Finally, the transitions are set among states. For each possible pair of states such as ResA and ResB, the tool creates two transitions: one transition from A to B that is triggered by Resolution A and other transition from B to A that is triggered by Resolution B. Once all the transitions have been set, the skeleton of the Possibility Space is ready.
The Strategies Management tab (see center of Figure 8) shows all the strategies that are available in the testing tool. The strategies introduced above (REG, INC-C and INC-M) are preloaded in the tool. Furthermore, the tool also provides functionality to load new strategies as Eclipse Plugins. These strategies are in charge of completing the skeleton of the Possibility Space.

The Strategies Comparison tab (see right of Figure 8) runs the testing process. First, the skeleton of the Possibility Space is completed by means of the selected strategies. The Synthesis and Modifications operations calculate the Base-model for each state and the Effect for each transition. Once the Possibility Space is completed, the tool runs the test for equality.

The criteria for the equality test can be selected by the user among the following options: Model Increments/Decrements, Base-model Comparison and Paths Comparison. The first two options implements the comparison among strategies while the last option implements the comparison among paths. The tool applies these criteria thought the Possibility Space to perform the equality test. Finally, the tool generates a report that summarizes the results of the test for equality.

### 6 Extra-Functional Properties of Strategies

Although all these strategies implement the same operations introduced at the beginning of this document, there are differences among them from the viewpoint of the extra-functional properties. Table 1 summarizes these differences by means of the following criteria: History, Performance and Persistency of the Base-model changes.

The History criterion evaluates if the strategies keep information about the Base-models that have been previously synthesized. This information is useful for techniques that debug invalid configurations and derive the minimal set of changes to fix flawed configurations [3]. Specially, when these techniques deals with SPLs which use staged configuration [7]. Staged configuration means that variability decisions are taken in multiple stages to form a complete configuration iteratively. For instance, the Configuration Understanding and REMedy (CURE) tool implements some of these techniques to debug invalid configurations [19].

Analyzing the introduced strategies from the viewpoint of the History criterion, we found the following results. On the one hand, the REG strategy stores previous configurations of the system, because this strategy works with copies of the Base-model. On the other hand, INC-C and INC-M just store the current state of the system, since these strategies apply all the modifications to the same Base-model.

Therefore, we recommend the use of the REG strategy for configuration debugging, since the REG strategy will provide more information for the analysis.

The Performance criterion evaluates how many elements are processed to synthesize a Base-model configuration. The performance of the synthesize operations is specially important for DSPLs. DSPLs products are adaptive systems, i.e. a product might pro actively adapt itself when changes are performed in its environment. For instance, [20] uses a DSPL to synthesize new system variants for mobile devices according to changes in the context.
From the performance viewpoint, executing the INC-C strategy only involves those elements that change from the previous configuration to the new one. The INC-M strategy additionally updates the Library in order to save Base-model changes. Finally, the REG strategy involves all the variations points, since this strategy regenerates the whole Base-model always.

Therefore, we recommend the use of the INC-C strategy for DSPLs, since in DSPL the performance of the synthesis operation impacts on the overall performance of the synthesized product.

The Persistence of the base Model Changes criterion evaluates the capability of strategies to save changes of the Base-model by run-time systems. Increasingly, some approaches are leveraging variability models at run-time [4]. A key benefit of using models at runtime is that models can provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns. For instance, [18] leverages variability models at run-time to achieve dynamically adaptive systems.

Analyzing the introduced strategies from the viewpoint of the Persistency criterion, we found that only the INC-M strategy saves changes to the Base-model. In a reconfiguration, this strategy moves the decrements of the Base-model to the Library instead of just deleting these model elements. Eventually, these elements will be back from the Library to the Base-model. The REG and INC-C strategies discard the base model changes, since they just delete the decrements of the Base-model.

Therefore, we recommend the use of the INC-M strategy for approaches that leverage models at run-time. These models can be modified by the run-time system without loosing the modifications in the next reconfiguration.

In this section, we have shown three strategies which support different extra-functional properties. Our intent is to develop new strategies that mix the above properties. For instance, INC-C with History support or REG with persistency of base model changes. Furthermore, we also plan to develop more strategies that support new extra-functional properties.

### 7 Applying the Strategies to Smart Homes

We have applied the previous strategies to a DSPL for adaptive Smart Homes [5]. This DSPL allows Smart Homes to use the variability modeling from the SPL design at run-time in order to determine the steps that are necessary to reconfigure the Smart Home. For instance, in the example of Figure 2 the Dynamic Product Line Architecture is re-targeted to a Nobody at Home configuration when the users leave the home.

This DSPL for Smart Homes use staged configuration since fragments are selected in multiple stages to form a complete configuration iteratively. At a late stage in the configuration process, developers may realize that a specific context condition cannot select some fragments due to reconfigurations in some previous stages. It is hard to debug the last configuration to figure out how to change reconfigurations in previous stages to make these fragments selectable [3]. To debug these staged configuration errors, we apply the REG strategy to synthesize the Smart Home configurations. The REG strategy synthesize configurations from an invariant CVL specification, keeping the history of configurations. Therefore, we are able to conduct a thorough analysis of the previous configurations for the purpose of debugging.

A fundamental problem in SPL engineering is that a real product line can easily incorporate several thousands of variation points [17]. The use of variability models to assist the system adaptation (as our DSPL for Smart Homes does) impacts the product performance. The incorporated latency comes from the synthesize operation that is performed at run-time when a context condition is fulfilled. For this reason, we use the INC-C strategy for deployment and we keep the REG strategy just for debugging. The INC-C strategy achieves better performance results since it only involves the subset of variation points affected by a context condition. On the other hand, the REG strategy always involves all the variation points independently of the context condition.

In an ongoing work [8], we also use this DSPL for service reconfiguration in the Ambient Assisted Living domain. In this domain we update the Base-model at run-time in order to save user preferences. The INC-M strategy is suitable for this purpose, since it enables the run-time system to modify the Base-model without losing the modifications in the next reconfiguration.

The realization of the variability transformation by means of interchangeable strategies enables SPL engineers to use the most suitable strategy for each concern. In this
section, we have illustrated how to take advantage of different strategies in a DSPL for adaptive Smart Homes. However, we can develop new strategies that provide other extra-functional properties in order to support more SPLs. Furthermore, the tool presented in this paper will help us to validate the implementation of these new strategies.

8 Related Work

Managing variability at run-time is not new, several SPL approaches [9, 14] delay variability decisions until run-time. The resulting products contain variation points that represent unbound options about how the software will behave. At run-time, product decisions are utilized to select among the options for each variation point. Once all the variation points are selected, the behavior of the final product is fully specified. We present a brief overview of these approaches as follows:

Goedicke et al. [9] presents an indirection architecture based on Component Wrapper objects and message redirection for dynamically composing and customizing generic components. In this product line architecture, each individual variability requirement can be implemented in a separate component. Such abstraction on details of other components reduces the knowledge that components need to perform customization.

Hoek [14] proposes an approach to support any-time variability, from statically at design time to dynamically at run-time. This approach is based on ubiquitous use of a product line architecture that organizes all abstractions throughout the lifetime of a software system. Within such architecture, variabilities in terms of both space (captured as explicit variation points) and time (captured as explicit versions of architectural elements) are specified.

In contrast to the above proposals, our work address the synthesis of a whole new product at run-time instead of binding some variation points at run-time.

Other SPL approaches also address the synthesis of a whole new product at run-time as follows:

Gomaa et al. proposed the Reconfigurable Product Line UML Based Environment (RPLUSEE) [10]. Their main contribution is provisioning software dynamic reconfiguration patterns. Depending on the location of dynamic reconfiguration information, these patterns are classified into master-slave, centralized, client-server and decentralized. This method also provides reconfiguration Statechart and reconfiguration transaction models for the dynamic reconfiguration. This approach focuses on high-level specifications of dynamic reconfigurable units.

Lee et al. proposed a systematic method to developing dynamically reconfigurable core assets and a reconfigurator that monitors and manages product configuration at run-time [16]. The method first analyzes a product line in terms of features and their binding time. Then, core assets are developed with the analysis results as key design driver. Finally, the developed reconfigurator address reconfiguration contexts, reconfiguration strategies and reconfiguration actions.

Compared to Gomaa’s and Lee’s work, our approach not only provides one implementation of the variability transformation but also different strategies. These strategies enable SPL engineers to use the most suitable strategy for each concern, because these strategies cover specific extra-functional requirements such as performance or support to reconfiguration debugging at run-time.

Finally, there are SPL approaches which address other extra-functional properties such as QoS:

Hallsteinsen et al. developed the MADAM approach [12]. This approach builds adaptive systems as component based systems families with the variability modeled explicitly as part of the family architecture. MADAM uses property annotations on components to describe their Quality of Service. For example a Video Streaming component may have properties such as start up time, jitter and frame drop. At run-time, the adaptation is performed using these properties and a utility function for selecting the component that best fits the current context.

White et al. developed the Scatter tool [20] to address efficient online variant selection. Scatter captures the requirements of the product line architecture and the resources of a mobile device and then quickly constructs a custom variant for the device. This tool also ensures that variant selection is optimal with regard to a configurable cost function.

In contrast to Hallsteinsen’s and White’s work, our approach covers several specific extra-functional properties, and we give recommendations to use the most suitable strategy for different concerns of run-time reconfiguration. Furthermore, we validate the implementation of our strategies for variability transformation using a testing approach which provides tool support.

9 Conclusions

Increasingly, more approaches apply SPLs to build run-time adaptive systems [12, 5, 18]. Although the details are different, these approaches share that they perform the variability transformation intensively at run-time. We
argue that the variability transformation can be realized by means of interchangeable strategies that have different extra-functional properties. These strategies enable SPL engineers to use the most suitable strategy for each concern, because these strategies cover specific extra-functional requirements such as performance or support to reconfiguration debugging at run-time.

In this work, we introduced three different strategies (REG, INC-C and INC-M) for realizing the variability transformation. We implemented these strategies by means of the Model Query project of the Eclipse Modelling Framework, and we validated these implementations using a testing approach which provides tool support. Then, we compared these strategies from the viewpoint of the extra-functional properties, and we also gave recommendations to use the most suitable strategy for different concerns of run-time reconfiguration.

Finally, we have evaluated the above strategies in a SPL for run-time adaptive Smart Homes. In this SPL we illustrated how we have take advantage of the different strategies in practice.

Our intend is to build a catalog of new strategies that cover extra-functional requirements. We believe that this catalog is useful for the SPL community since variability transformations are more and more applied to domains which require extra-functional properties. Furthermore, although the strategies presented in this work are based on CVL, the approach as such, can be applied by means of other languages for variability specification.

At http://www.autonomic-homes.com there are several videos available about the reconfiguration of our prototype smart home.

Acknowledgments. The work presented here has been co-developed within the MoSiS project (ITEA 2ip06035) and the DiVA project (EU FP7 STREP). This work has been also supported by MEC under the project SESAMO and the DiV A project (EU FP7 STREP). This work has been also supported by MEC under the project SESAMO and the DiVA project (EU FP7 STREP).

At the 13th International Workshop of Ambient Assisted Living (IWAAL09), LNCS, 2009.


References