ArchSPL-MDD: An ADL-based Model-Driven Strategy for Automatic Variability Management

Ana Luisa Medeiros¹, Everton Cavalcante¹,², Thais Batista¹, Eduardo Silva¹
¹Federal University of Rio Grande do Norte, Natal, Brazil
²IRISA-UMR CNRS/Université de Bretagne-Sud, Vannes, France
analuisafdm@gmail.com, {evertonrsc, eduardoafs}@ppgsc.ufrn.br, thais@ufrnet.br

Abstract—Model-driven strategies have been used in the development of software product lines (SPLs) to facilitate product customization and to generate the source code of the derived products through variability management. In this context, the architecture description of the SPL is essential to make it clear how the architecture realizes the feature model and to represent both domain and application engineering architectural artifacts. Moreover, it is important to establish the association between the architectural specification and the artifacts involved in the SPL development process towards code generation. In this paper, we present ArchSPL-MDD, a model-driven strategy to support explicit modeling and automatic management of variabilities in SPLs. ArchSPL-MDD is associated to a generic process with systematic activities aimed to generate customized source code from the product configuration. Furthermore, the proposed strategy uses the LightPL-ACME architecture description language to support the architectural specification of the SPL, which is input of model transformations towards automatically generating SPL products. To evaluate the efficiency and applicability of ArchSPL-MDD, we conducted a controlled experiment by using GingaForAll, an SPL for the Ginga digital TV middleware.

Keywords—software product lines; variability management; model-driven development; model transformation; architecture description language; code generation

I. INTRODUCTION

The synergy between software product lines (SPLs) [1] and model-driven development (MDD) [2] brings a variety of benefits in terms of variability management, such as an easier definition of the product line creation process [3, 4]. In this context, the artifacts generated in the SPL development activities can be represented as models refined by MDD transformations, thereby increasing the degree of automation in such activities.

The architectural description of an SPL is one of the relevant models associated to the SPL development. Indeed, describing an SPL architecture is an important activity as it allows anticipating important decisions regarding the system design as well as it establishes how the architecture can realize the feature model. As in any software-intensive system, architecture description languages (ADLs) [5, 6] can also be used to represent architectural characteristics of an SPL, thus providing a better understanding about the SPL itself and fostering the generation of other artifacts throughout the development life cycle.

Aiming at properly exploring the benefits provided by ADLs in the SPL context, a model-driven process can encompass systematic activities able to generate artifacts described in an ADL and refine them towards the customization of products of an SPL and generation of their respective source code. To come with such a process, this paper presents ArchSPL-MDD, an MDD-based strategy that comprises a generic process with activities to model and manage variabilities in an SPL as well as to support the generation of derived products and their source code. The proposed strategy uses architectural models described in LightPL-ACME [7], a simple, lightweight ADL for representing SPL architectures and establishing relationships between the variabilities described in the feature model and the SPL architecture in an easy way. Additionally, ArchSPL-MDD has an associated tool aiming at assisting SPL architects to handle the models involved in the process and to automatically generate other artifacts, such as source code of products. In order to evaluate the efficiency and applicability of ArchSPL-MDD, we have conducted a controlled experiment by using GingaForAll [8], an SPL for the middleware adopted by the Brazilian Digital Television System (SBTVD).

The remainder of this paper is organized as follows. Section II provides the background of this work in terms of basic concepts related to SPL, the GingaForAll SPL used as running example, and an overview about the LightPL-ACME ADL. Section III presents the ArchSPL-MDD strategy and its respective tool as well as its application for the GingaForAll SPL. Section IV describes an evaluation of ArchSPL-MDD by means of the controlled experiment and the obtained results. Section V discusses related work. Finally, Section VI contains some concluding remarks and directions to future work.

II. BACKGROUND

A. Software Product Lines

SPLs allow creating a family (or product line) of similar products by using a common software infrastructure to mount and configure parts designed to be reused among products [1]. SPL approaches usually identify commonalities (similarities) between all members of the family as well as characteristics that vary among them, the so-called...
variabilities. Therefore, the members of a family have a basic set of common functionalities and associated variations individualizing each of these members.

The SPL development process follows two main activities, namely domain engineering and application engineering [9] (see Fig. 1). The former aims to systematize the gathering, organization, and storage of reusable and consistent information in the form of artifacts (such as requirements, architectures, components, etc.), so that development is designed for a planned reuse of these artifacts and similarities are explored to build different products. In turn, the latter aims to specify and customize different products by (re)using the artifacts generated by the domain engineering activity upon specific-product requirements. The construction of a software product (also called instantiation or derivation) is made from a configuration of core assets consisting of an arrangement of the software artifacts that implement the product.

In the SPL development, similarities and variabilities among products of a family are typically modeled in terms of features. A feature is a concept visible to any stakeholder involved in the application development and can be a requirement, a function, or a non-functional feature depending on the interest of such a stakeholder [9]. Features are organized in feature models representing commonalities, variabilities, and constraints related to the relationships between these elements. Feature models often have a tree structure in which features are represented by tree nodes and the variations between features are represented by edges and feature groups, so that the hierarchical organization of the diagram describes the key concepts from more general to more specific concepts as they descend the tree. Features can be [10]: (i) mandatory, i.e., the feature must necessarily be included in a product; (ii) optional, i.e., the feature may or may not be included if the feature from which it derives is selected; (iii) or-inclusive, so that at least one feature is selected from a set of related features; and (iv) alternative, so that exactly one feature is selected among a set of related feature. Fig. 2 shows part of feature model for the GingaForAll SPL, used as running example throughout this paper and described in Section II.B.

B. Running Example: The GingaForAll SPL

Ginga [11] is the middleware adopted by the Brazilian Digital Television System (SBTVD) whose main functionalities are receiving and processing the digital TV signal as well as providing specific services to declarative and imperative applications running atop the middleware. In the Ginga architecture, this set of common digital TV services is called Ginga-CC (Ginga Common Core), which is responsible for tuning channels in, exhibiting audio and video, and managing user preferences.

Some deficiencies in the original architecture of Ginga-CC have motivated the refactoring of this architecture in order to define an SPL-based architecture while increasing the configuration capability of Ginga through an automatic variability management. As a result, the GingaForAll SPL [8] was generated by applying a domain engineering methodology in which common and variable features were identified from the application domain. Fig. 2 shows the GingaForAll partial feature model. The Tuner feature is responsible for selecting the physical channel used for signal transmission, which can be terrestrial (Terrestrial feature), satellite (Satellite feature) or over IP (IP feature). The Tuner feature also supports reading from the file system, represented by the File System feature. The IP feature has variants represented by the IPTV, InternetTV, and P2PTV features. The Application Manager mandatory feature is responsible for loading, instantiating, configuring, and executing applications, besides controlling their life cycle and managing resources and the access control. Finally, the Data Processing optional feature is responsible for accessing, processing, and providing data streams to other components of the middleware, besides notifying other components about application updates, synchronization, etc.

C. LightPL-ACME

LightPL-ACME [7] is a lightweight, simple language for SPL architecture description. This language was ex-
tended from ACME [12], a well-known general-purpose language that provides a basis for developing new domain-specific ADLs. LightPL-ACME promotes the separation of the domain and application engineering activities within SPL development by creating specific abstractions for features (related to the domain engineering) and products (related to the application engineering). Moreover, it was designed envisioning the representation of the architecture and its relationship with the features. As new abstractions, LightPL-ACME introduces the ProductLine, Feature, and Product elements and allows representing the referenced architecture concept. LightPL-ACME is supported by LightPL-ACME Studio [13], a graphical and textual editor for describing SPL architectures.

**ProductLine and Feature elements.** The ProductLine element represents the SPL and its specific elements such as features and products. Within the ProductLine element, the Feature element is defined to represent the features that compose the SPL, either similarities or variabilities. In this context, the Feature element is specified by defining an identifier and an associated type related to the types of features that may occur in an SPL (i.e., alternative, optional, or-inclusive, and mandatory). LightPL-ACME also provides the extends mechanism, which allows establishing an inheritance/dependency relationship between features and their respective father features. The description of features is important because it is used to guide the development of the system architecture as the features represent the functionalities that must be provided by architectural elements of the system. Fig. 3 illustrates the definition of the features of the GingaForAll product line. For example, the Demultiplexer feature is mandatory (line 4) and the Hardware and Software features (derived from the Demultiplexer feature) are alternative (lines 5 and 6).

**Product element.** The Product element corresponds to the concept of products that can be generated by an SPL. The specification of this element consists of an identifier and a set of identifiers regarding the Feature elements that compose it. The inclusion of Feature elements also occurs by hierarchy, so that the inclusion of a Feature element in a Product element includes all of its direct dependencies, which take place through the inheritance relationship. As shown in Fig. 3, the Ginga _NCL LUA Product_ (line 14) includes the ContextManager, Hardware, NCLLua, and HTML features. Since the direct dependencies of these features are implicitly included, it is not necessary to include the MediaProcessing and Demultiplexer features, from which the HTML and Hardware features are derived.

**Referenced architectures.** In LightPL-ACME, a referenced architecture consists of a description of the system architecture enriched with references to the Feature elements previously described in a ProductLine element. In this perspective, they aim to allow a general view of the system and clearly define which architectural elements correspond to the features of the SPL. The notion of referenced architecture is part of the well-known concept of configuration knowledge [9], which represents all available information used to support the product derivation process and includes such a mapping between SPL artifacts and features as well as dependencies and illegal combinations between features.

Another relevant concept refers to the so-called base architecture, i.e., a conventional software architecture defined in terms of elements such as components, connectors, and system configurations. The mapping mechanism provided by LightPL-ACME aims to map such architectural elements to the features of the described SPL. This can be made by adding the MappedTo keyword and a list of Feature elements after the names of the architectural elements. It is worth highlighting that the system element representing the base architecture must adhere to the architectural style described by a ProductLine element is required to make Feature elements accessible in the referenced architecture.

Fig. 4 illustrates the mapping between the base architecture and the GingaForAll architectural description in order to compose the referenced architecture. In this example, the GingaFull system (base architecture) adheres to the GingaForAll ProductLine, thereby composing the referenced architecture. Through the MappedTo mechanism provided by LightPL-ACME, the ContextManager and MediaProcessing components described in the base architecture are respectively mapped to the CxtManager and MdProcessing features specified in the GingaForAll ProductLine.

It is important to mention that all features described in the SPL associated with the referenced architecture must be mapped to at least one element of the architecture, otherwise the architecture does not fulfill all Feature elements of the SPL and hence it could not be considered a valid architecture for this system. However, there may be architectural elements not mapped to Feature elements of the SPL, so that they can be understood as implementation-specific elements.

### III. THE ARCHSPL-MDD STRATEGY

#### A. Process Definition

The ArchSPL-MDD strategy comprises a MDD process with generic activities to specify SPL architectures...
and generate products. This process separates the domain engineering and application engineering activities as well as its respective models aiming at simplifying variability management and product derivation. In order to specify SPL architectures, the process realized by ArchSPL-MDD uses the LightPL-ACME ADL, thus providing a greater abstraction level to the models. With LightPL-ACME, it is possible to support an explicit representation of structural concepts related to SPLs (features, products, and the SPL itself) and the definition of dependency and mutual exclusion constraints for the elements that represent the features.

Fig. 5 shows an activity diagram with the general overview of the ArchSPL-MDD process, with its activities and artifacts. The process consists of the five following systematic activities used to automatically transform and refine models:

1. **Build Architecture Model**: this activity is realized by the domain engineer, who specifies the referenced architecture model resulted from the mapping of architectural elements of the base architecture to features of the SPL.
2. **Configure Product**: this activity is realized by the application engineer and results in a product family model that will guide the development and generation of products.
3. **Generate Product**: the purpose of this activity is to generate a product model containing one or more products specified by the application engineer.
4. **Generate product source code**: the purpose of this activity is to obtain a product source code skeleton from the product model according to the architecture.
5. **Reuse product source code**: this activity aims to generate the product source code according to the product model by reusing existing code available in a software repository.

When instantiating the ArchSPL-MDD process, the domain engineer models the referenced architecture (Activity 1). In turn, the application engineer is responsible for choosing the product configuration (Activity 2) and which features will be part of the product (Activity 3). In the final stages of the application engineering activity, the product source code is generated from the scratch (Activity 4) or it can be generated by reusing existing source code (Activity 5).

### B. ArchSPL-MDD Process Artifacts

This section details some artifacts specified in the LightPL-ACME ADL that are part of the ArchSPL-MDD process. For illustrative purposes, we use the *GingaForAll* SPL described in Section II.B.

As shown in Fig. 3, the feature model contains the description of the *GingaForAll* SPL, which specifies its features and possible products resulted from combinations of these features in order to compose the product model. In turn, the base architecture model (see Fig. 6) contains the definition of the system architecture in LightPL-ACME. The adherence of the *GingaOverview* system (described in the base architecture) to the *GingaForAll ProductLine* results in the referenced architecture shown in Fig. 7. As previously mentioned, the relationship between architectural elements and features is specified through the *MappedTo* mechanism provided by LightPL-ACME.

Fig. 8 shows part of the C++ product source code generated in the Activity 4 based on the features specified in the product. In this example, the *Tuner* class is equivalent to the *Tuner* component described in Fig. 7 and associated to the *Tuner* feature.

### C. The ArchSPL-MDD Tool

Fig. 9 depicts the architecture of the ArchSPL-MDD tool, which is structured upon five main elements: (i) the Eclipse Platform, which supports the tool as a whole; (ii) LightPL-ACME studio, textual and graphical editor based
System GingaOverview {
  Component Tuner = {
    Port ReceiveData;
    Port ListenersPort;
  }
  Representation TunerRep = {
    System TunerRep = {
      Component TunerIp = {
        Port ReceiveData;
        Port ListenersPort;
      }
      Representation TunerIp_Rep = {
        System TunerIp_Rep = {
          Component IPTV = {
            Port ReceiveData;
          }
        }
      }
    }
  }
  Binding IPTV.ReceiveData to TunerIp.ReceiveData;
}

System GingaOverview : GingaForAll {
  Component Tuner MappedTo Tuner = {
    Port ReceiveData;
    Port ListenersPort;
  }
  Representation TunerRep = {
    System TunerRep = {
      Component TunerIp = {
        Port ReceiveData;
        Port ListenersPort;
      }
      Representation TunerIp_Rep = {
        System TunerIp_Rep = {
          Component IPTV MappedTo IPTV = {
            Port ReceiveData;
          }
        }
      }
    }
  }
  Binding IPTV.ReceiveData to TunerIp.ReceiveData;
}

class Tuner {
  public:
    Tuner();
    virtual ~Tuner();
  private:
    void ReceiveData();
    void ListenersPort();
};

transforms the textual model of the referenced architecture in LightPL-ACME to a corresponding XMI file, which will be used as input of the ATL transformation rules implemented in the ArchSPL-MDD Transformer. These rules will execute a model-to-model (M2M) transformation in order to transform the referenced architecture model into a product model containing one or more products specified in LightPL-ACME by the application engineer. Finally, the product source code generated by ArchSPL-MDD can be a code skeleton obtained from the product model and/or a source code retrieved from an existing repository. The produced source codes are obtained from model-to-text (M2T) transformations performed in the Source Code Editor.

IV. EVALUATION

This section presents a controlled experiment [17] conducted to evaluate the applicability of ArchSPL-MDD when compared to the GingaForAll strategy [8] for generating products of the GingaForAll SPL (see Section II.B). The GingaForAll strategy is a model-driven process in which models related to the GingaForAll SPL are defined using UML and UML profiles. It was chosen to be compared with ArchSPL-MDD in this study due to the similarity between the strategies, thereby allowing us to outline more direct comparisons.

A. Design

Goals, questions, metrics, and hypothesis. We have organized and established the specific goal, questions, and associated metrics to answer the questions by using the Goal–Question–Metric (GQM) approach [18]. Fig. 11 shows the hierarchical organization of these elements of the controlled experiment. It is worth mentioning that while metric M1 is quantitative, the others (M2 and M3) are qualitative and were assessed based on the feedback received from the subjects participating in the experiment. The refinement of
the stated questions relies on the analysis of three hypotheses (each one with null and alternative forms) related to the defined metrics, as synthesized in Table I.

Subjects. In this experiment, six Computer Science graduate students from the Federal University of Rio Grande do Norte (Natal, Brazil) were selected by convenience based on a minimal previous knowledge on SPLs, MDD, UML, and ADLs. To improve the knowledge of the participants, they received training on: (i) SPLs, with focus on the GingaForAll SPL; (ii) ADLs, in particular the LightPL-ACME language; (iii) MDD, in the context of model-driven processes and model specification; (iv) UML, used for model specification in the GingaForAll strategy; (v) the GingaForAll strategy and its respective tool; and (vi) the ArchSPL-MDD strategy and its respective tool. The six students were randomly organized in two groups: the first one (Group I) started the experiment using the ArchSPL-MDD strategy while the other (Group II) started with the GingaForAll strategy. The order defining which strategy would be used first was disposed at random. When using ArchSPL-MDD, the participants had to describe the models regarding the GingaForAll SPL and both base and referenced architectures in LightPL-ACME, as well as to specify a product whose code would be generated by the strategy. In turn, the use of the GingaForAll strategy implied in modeling the base architecture and application of the feature UML profile and selecting the variabilities to customize a product. After the first modeling round, the groups exchanged the strategies.

Dependent and independent variables. This experiment comprises one independent variable, i.e., the strategy used to specify the models. Such an independent variable has two related alternatives, namely (i) to specify models by using the LightPL-ACME ADL with ArchSPL-MDD and (ii) to specify models by using UML with the GingaForAll strategy. In addition, there are three dependent variables, which refer to the metrics to be assessed in the experiment: (i) efficiency, related to the time spent to specify the models when applying each strategy; (ii) complexity, which regards the observed difficulty and verbosity when using each strategy; and (iii) expressiveness, which refers to the sufficient representation of the elements related to the SPL.

B. Conduction

The experiment was conducted in three phases. In the first one, the participants were trained in terms of the elements mentioned in Section IV.A in order to improve their knowledge. Afterwards, the execution environment was prepared by installing the Eclipse IDE plug-ins regarding the GingaForAll and ArchSPL-MDD tools. It is important to highlight that the tools were used only for specifying the models and evaluating the strategies, and not for assessing issues regarding their usability.

The second phase of the experiment was conducted in terms of specifying the models by using UML in the GingaForAll strategy and describing the models by using the LightPL-ACME ADL in ArchSPL-MDD. Before starting this phase, the participants answered an initial form to characterize their profile and collect personal information about professional expertise and skills in software development and familiarity with involved concepts. Additionally, a partial feature model of the GingaForAll SPL and a product resulted from combinations of some of these features was
To evaluate the applicability of ArchSPL-MDD against the GingaForAll strategy for generating products of the GingaForAll SPL

Q1: Is the model specification with LightPL-ACME (using ArchSPL-MDD) efficient when compared to the use of UML (using GingaForAll)?

M1: Time spent to specify models when applying each strategy

Q2: What is the degree of complexity for applying the GingaForAll strategy compared to the use of ArchSPL-MDD?

M2: Observed difficulty and verbosity when using each strategy

Q3: What is the degree of expressiveness for representing models in the GingaForAll strategy compared to ArchSPL-MDD?

M3: Sufficient representation of the elements related to the SPL

Legend: 
- Goal
- Question
- Metric

Figure 11. Hierarchical organization of the goal, questions, and metrics of the controlled experiment.

<table>
<thead>
<tr>
<th>Question</th>
<th>Hypotheses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>H1: ( \text{time(UML)} = \text{time(LightPL-ACME)} )</td>
</tr>
<tr>
<td>Q2</td>
<td>H2: ( \text{complexity(GingaForAll)} = \text{complexity(ArchSPL-MDD)} )</td>
</tr>
<tr>
<td>Q3</td>
<td>H3: ( \text{expressiveness(GingaForAll)} = \text{expressiveness(ArchSPL-MDD)} )</td>
</tr>
</tbody>
</table>

Table I: Hypotheses related to the questions investigated in the controlled experiment

provided to the participants, as well as a list of tasks to be performed: (i) to specify the base architecture; (ii) to select the features for customizing a product; and (iii) to specify the referenced architecture according to the given specifications.

Next, the participants specified the input models and the product in LightPL-ACME and UML with the respective strategies. Finally, the participants answered another form in which they registered their opinion about the evaluated strategies and listed difficulties, facilities, and other relevant observations. All materials used in this experiment are publicly available at https://sites.google.com/site/archsplmdd/experimental-reports/. It is worth mentioning that the models described by the participants were equivalent with minor differences with respect to organization. Nevertheless, they were checked and have shown to correctly describe the SPL.

C. Results

This subsection reports the results obtained in the controlled experiment aiming at verifying the hypotheses stated in Section IV.A based on the feedback provided by the participants. To verify such hypotheses (H1 to H3), it is necessary to answer the formulated questions (Q1 to Q3) using the respective metrics (M1 to M3).

Q1: Efficiency. The time spent for specifying the models using LightPL-ACME/ArchSPL-MDD was smaller than the time spent for specifying them with UML in the GingaForAll strategy, thus validating the alternative hypothesis H1. More precisely, the average time for specifying the models by using ArchSPL-MDD was 23 minutes whereas the average time spent with the GingaForAll strategy was 38 minutes, a difference of 39.47% in favor of ArchSPL-MDD.

Q2: Complexity. The participants were unanimous when reporting that UML was complex, non-intuitive, and mentally tiring for specifying the GingaForAll models. In addition, they regarded the GingaForAll strategy as heavyweight for model specification in comparison to ArchSPL-MDD. As a result, the alternative hypothesis H2 can be considered as true.

Q3: Expressiveness. The participants unanimously reported that the degree of expressiveness of UML in the GingaForAll strategy is very low compared to the LightPL-ACME ADL in the ArchSPL-MDD strategy. Therefore, the alternative hypothesis H3 can be accepted.

D. Discussion

In general, the participants reported in the answered forms that the model specification using LightPL-ACME within ArchSPL-MDD is able to express important elements related to the SPL context (such as features and respective types and relationships as well as the products that can be generated from them) and the base architecture in a simple, easy way. The participants unanimously stated that LightPL-ACME is better for specifying models and products in comparison to UML as it allows a direct relationship to the SPL elements, thereby making the description simpler, more readable, and more intuitive. These aspects were not observed in UML.
However, the absence of a mechanism for expressing cardinality in the features was pointed as a desirable improvement for LightPL-ACME. In the GingaForAll strategy, a greater knowledge about the strategy and the elements composing UML diagrams is required to establish the relationship between base elements and features. Additionally, the participants reported that the GingaForAll strategy is difficult for visualization, non-intuitive, and error-prone.

Another important aspect highlighted by the participants concerns the expressiveness for specifying models. For instance, to construct the referenced architecture, the comprehension becomes harder since UML profiles need to be created and applied to reference the base architecture model, thus requiring a more complete tool support. In LightPL-ACME, a greater easiness was observed since the relationship between the base architecture and the SPL features towards creating the referenced architecture can be established by using only the MappedTo mechanism provided by the language. On the other hand, there is a lack in terms of supporting abstract features defined for organization purposes [19] since it to map all defined features to architectural elements.

Finally, the participants were asked if they would apply ArchSPL-MDD in the development process of an SPL. They provided a positive feedback by stating that the proposed strategy is easy to understand and has a high degree of generality. Despite the GingaForAll strategy uses UML, the participants regarded it as extremely specific and pointed out that the association between base elements and the SPL features was not intuitive.

E. Threats to Validity

The conducted controlled experiment and its results may have been affected by some threats to empirical validity [17, 20]. We briefly discuss some of these limitations in the following.

Internal validity. Internal validity is mainly concerned to unknown factors that may influence the results. To reduce the threat that the subjects may not have the needed skills, we have ensured that they had a minimum knowledge on the topics related to this study. We have also performed a pilot study and collected feedback about the difficulty of performing the required tasks. In addition, we carefully designed, written, and iteratively refined the evaluation forms to be answered by the participants in order to (i) mitigate the risk of ambiguous and poorly phrased questions and (ii) ensure that all participants could understand well the proposed questions. We have also designed the experiment aiming at not postponing its duration, an issue that could discourage or physically and/or mentally tire the participants. Nevertheless, the results may still be affected by making a different organization of the participants, e.g., keeping them in individual work instead of assigning them into groups, as done in this experiment.

External validity. External validity is related to claims for the generality of the presented results. We acknowledge that the number of participants is small and hence may be a non-representative sample. The participants were chosen by convenience and also have a similar, homogeneous profile, a fact that would not probably be observed if the experiment was performed in other conditions encompassing subjects with different expertise levels. For these reasons, the obtained results cannot be generalized. Despite the use of students instead of professionals could raise another threat to the external validity of this study, we agree with Salman et al. [21] in that similar performances can be observed with students and professionals in a scoped experiment on an approach that is new to both of them. Nonetheless, further studies using a larger sample and applied in other contexts are needed to increase the external validity of our study. Finally, although further studies targeting other SPLs are required, ArchSPLMDD was conceived as a domain-independent strategy for SPL development mainly due to the use of LightPL-ACME as a general-purpose ADL for describing SPL architectures and well-known MDD technologies.

Reliability validity. Reliability validity refers to the possibility of replicating the study. Aiming at mitigating this type of threat, all materials used in the controlled experiment were made publicly available at https://sites.google.com/site/archsplmdd/experimental-reports/.

Construct Validity. Construct validity refers to the relationship between the concepts and theories behind the experiment and what is measured and affected. Metrics M2 and M3 (see Section IV.A) are qualitative and were assessed based on the subjective responses provided by the participants in the evaluation forms. Moreover, the applied forms also contain discursive (open) questions, which could yield different interpretations. In order to reduce this type of threat, we have discussed the answers provided to these questions.

V. RELATED WORK

The literature presents some strategies related to the concepts addressed by ArchSPL-MDD. However, most of them do not consider variability management or make use of a more complex process and tools to generate SPLs and UML resources. Moreover, the existing derivation tools focus on the definition of products at the code level rather than high-level models such as the ones described in an ADL.

Buchmann and Schwäger [22] show the use of FAMILE for mapping a feature model to implementation objects and generating Java code fragments. Due to the lack of a structured process for the proposed refinement, the associated tool requires using several Eclipse plug-ins and requires the manipulation of many files, thereby making FAMILE difficult to execute and apply. In turn, ArchSPL-MDD comes up with a strategy based on a MDD process.
that uses few models and a set of semi-automated activities to generate source code. Furthermore, the ADL used in ArchSPL-MDD supports features specification and product configuration through specialized models, thus simplifying the management of features that will be part of each product.

Nascimento et al. [23] present a model-driven infrastructure for implementing SPL architectures through the refinement of coarse-grained models (feature model) to fine-grained models (source code), similarly to the ArchSPL-MDD strategy. Within such an approach, the FarM (Feature Architecture Mapping) method [24] uses a sequence of transformations to map a feature model to a product line architecture by refining the features of an initial model based on the analysis of functional features and business logic that can implement the architectural components. However, the approach implemented by ArchSPL-MDD is simpler in terms of tools, languages, and processes. Additionally, ArchSPL-MDD defines a process that clearly isolates the activities related to domain engineering and application engineering, an issue that is not properly handled in the Nascimento et al.’s infrastructure.

Ouali et al. [25] propose the definition of a domain-independent SPL development process encompassing the domain engineering and application engineering activities. In such a proposal, the domain engineering activity addresses the creation of base models representing requirements. Moreover, the feature model represents the SPL structure and a state-machine diagram is used as means of modeling SPL behavioral aspects. The process is based on the automatic transformation of models until obtaining executable code and enables developers to generate a suitable SPL to the requirements elicited in the initial phases of the development and new arisen ones. However, this proposal lacks of a tool enabling developers to perform the activities defined in such a process and specify the corresponding artifacts. Despite it has the same goal of ArchSPL-MDD (i.e., to generate source code of products), it is not clear in terms of how to implement the product generation as well as the domain engineering and application engineering activities could be differentiated.

The Genarch tool [26] aims at assisting software engineers in the SPL creation and derivation activities. This approach defines three models that are used to represent variabilities and elements related to the SPL implementation, namely (i) the product line implementation model, (ii) the feature model, and (iii) the configuration model. Similarity to Genarch, ArchSPL-MDD uses a generative approach, i.e., it aims to automatically generate software from a high-level specification. Both approaches deal with models at the architectural level, but ArchSPL-MDD takes advantage on the simplicity and expressiveness provided by the LightPL-ACME ADL. Finally, ArchSPL-MDD has an associated MDD process whereas Genarch uses a configuration file for mapping features to code level.

VI. Conclusion

In this paper, we presented ArchSPL-MDD, a strategy based on the integration of MDD and SPL approaches for specifying an SPL with customized products. The proposed strategy comprises a process with activities used to transform and refine the models specified in LightPL-ACME ADL. Moreover, ArchSPL-MDD is generic and encompasses the creation and representation of the feature models and the system architecture by using LightPL-ACME for specifying the referenced architecture and the product family, product generation, and the product source code. In addition, we have developed the ArchSPL-MDD tool to support the proposed MDD process and the automatic product derivation.

The ArchSPL-MDD strategy was evaluated in a controlled experiment using GingaForAll, an SPL for the Ginga digital TV middleware. The performed experiment pointed out that the ArchSPL-MDD strategy showed to be better in terms of efficiency, complexity, and expressiveness when compared to GingaForAll [8], an UML-based MDD strategy. As future work, we intend to verify the quality of the code generated by the ArchSPL-MDD tool and the applicability of the proposed strategy and tool in other case studies of a different domain.

REFERENCES


