Towards Supporting SPL Engineering in Low-Code Platforms using a DSL Approach

Alexandre Bragança
atb@isep.ipp.pt
Institute of Engineering of Porto
Porto, Portugal
Polytechnic of Porto
Porto, Portugal

Isabel Azevedo
ifp@isep.ipp.pt
Games, Interaction and Learning Technologies
Porto, Portugal
Institute of Engineering of Porto
Porto, Portugal
Polytechnic of Porto
Porto, Portugal

Nuno Bettencourt
nmb@isep.ipp.pt
Interdisciplinary Studies Research Center
Porto, Portugal
Institute of Engineering of Porto
Porto, Portugal
Polytechnic of Porto
Porto, Portugal

Carlos Morais
carlos.morais@omnialowcode.com
NumbersBelieve
Matosinhos, Portugal

Diogo Teixeira
diogo.teixeira@omnialowcode.com
NumbersBelieve
Matosinhos, Portugal

David Caetano
david.caetano@omnialowcode.com
NumbersBelieve
Matosinhos, Portugal

Abstract

Low-code application platforms enable citizen developers to autonomously build complete applications, such as web applications or mobile applications. Some of these platforms also offer support for reuse to facilitate the development of similar applications. The offered mechanisms are usually elementary, they allow module reuse or building a new application from a template. However, they are insufficient to achieve the industrial level reuse necessary for software product lines (SPL). In fact, these platforms were conceived to help build standalone applications, not software families and even fewer software product lines. In this paper, we argue that the major limitation is that these platforms seldom provide access to their metamodel, the access to applications' models and code is also limited and, therefore, makes it harder to analyze commonality and variability and construct models based on it. An approach is proposed to surpass these limitations: firstly, a metamodel of the applications built with the platform is obtained, and then, based on the metamodel, a domain-specific language (DSL) that can express the models of the applications, including variability, is constructed. With this DSL, users can combine and reuse models from different applications to explore and build similar applications. The solution is illustrated with an industrial case study. A discussion of the results is presented as well as its limitations and related work. The authors hope that this work provides inspiration and some ideas that the community can explore to facilitate the adoption and implementation of SPLs in the context, and supported by, low-code platforms.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; *Model-driven software engineering*; **Software product lines**.

Keywords: low-code platforms, software product line engineering, domain specific languages

ACM Reference Format:
Alexandre Bragança, Isabel Azevedo, Nuno Bettencourt, Carlos Morais, Diogo Teixeira, and David Caetano. 2021. Towards Supporting SPL Engineering in Low-Code Platforms using a DSL Approach. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3486609.3487196

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10.
https://doi.org/10.1145/3486609.3487196

1 Introduction

Low-code application platforms are systems that do not require users to be professional developers to rapidly create applications. These platforms can also be useful for highly qualified professionals who do not need to spend a lot of effort on easy, but time-consuming tasks, and can thus focus their attention on other considerations that require expertise. The Low-code application platform (LCAP) market has high growth expectations [25]. Gartner estimates that by 2023 over 50% of medium to large enterprises will have adopted an LCAP as one of their strategic application platforms [30]. As these types of software development platforms are more and more used, either by citizen developers [14] or more

technically qualified professionals, the applications developed will probably become more complex and aim to solve more difficult problems.

The work presented in this paper is a side result of an ongoing applied research project whose main goal is to add support for a textual domain-specific language (DSL) in an LCAP named OMNIA [31]. During this project, one of the main issues identified by users is that they frequently need to do repetitive tasks in their modeling activities, because they often need to solve similar problems with the platform. Therefore, the solutions for these problems are also very similar, basically requiring models where most of their elements are similar, with only small differences for specific variants or features. In an LCAP, this may require repeating manual tasks in a web modeling application of the LCAP. And thus, one of the major advantages that these users see in a textual DSL is the possibility of reusing parts of models by copying and pasting them between solutions. Although copy and paste are probably the most used techniques for solving similar problems, it is rarely a good solution. A more systematic and manageable approach is required, like the one possible with software product lines (SPL).

This paper reports on the experience of adding support for a software product line engineering approach for a low-code application platform using domain-specific languages. For that, a DSL capable of representing all the modeling concepts of the LCAP is created (we will call it low-code DSL or LC-DSL). The LC-DSL is then used to model groups of applications of the LCAP that may be managed as an SPL, i.e., the LC-DSL is a model that covers all the applications of the LCAP that are treated as an SPL. To express variability, a variability DSL is also required (we will call it Variability DSL). The Variability DSL can be used to express the variability of the product line as well as the configurations used for each application of the SPL. With this approach, users can model applications either by using the LCAP or the LC-DSL, but variability is expressed only outside the LCAP, by using annotations in the LC-DSL that reference elements of the Variability DSL. Specific models of the LCAP (and, therefore, applications) can be produced by removing elements in the LC-DSL that are annotated with variability elements of the Variability DSL that are not included in a variability configuration model of the application, also expressed in the Variability DSL. To integrate the existing LCAP with the new LC-DSL, an import/export process was implemented to obtain the LC-DSL from the models in the LCAP and vice-versa.

This approach was experimentally tested by applying it in a case study that is used as a tutorial to the OMNIA low-code platform. The tutorial was expanded and adapted to include variability requirements and, as such, became more suitable for a solution based on SPL engineering.

Although LCAP adoption by the industry is growing, its study by the academy as a research topic is still only starting. There are, of course, recent exceptions, such as the European Lowcomote research project, which, according to its proponents "aims to train a generation of professionals in the design, development, and operation of new LCDPs" [24, 36]. However, these recent research publications usually focus on other relevant aspects to allow building new LCAPs, with improved architectures and features. As described previously, our proposed approach is based on an almost orthogonal solution regarding the LCAP, i.e., it does not require changes in the LCAP, which usually would result in significant costs for the LCAP author. Furthermore, according to our knowledge, there is no published work that directly focuses on the same problem. Therefore, we hope that this research may bring the discussion of this topic to the community, and, eventually, inspire the community to improve the solution and propose and discuss alternatives.

The remainder of this paper is structured as follows. Section 2 presents the context and motivation for this work. Section 3 presents the proposed approach and details all its major components. Section 4 is dedicated to the case study. In Section 5, a discussion of the results of the work is presented, including lessons learned. Section 6 includes an analysis of related work. The paper is concluded in Section 7, where some near future work is also presented.

2 Context and Motivation

This section presents the context and motivation for the approach presented in the paper.

2.1 SPL Engineering

According to Paul Clements and Linda Northrop, "a software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [6]. This definition is commonly accepted by the community and it has set the conceptual foundation for the field, as confirmed in other reference books, such as the ones from Klaus Pohl *et al.* [34] and Frank J. van der Linden *et al.* [23]. The software product line engineering approach is thus composed of two integrated sub-processes: domain engineering and application engineering, as illustrated in Figure 1.

From the previous definition and illustrated process, it is reasonable to assume that in a software product line engineering approach: (1) it is required to capture the common and variable features in the domain(s) of the product line; (2) there are core assets that are the main, shared, building blocks of the product line; (3) a process is required to be in place to enforce the prescribed way of building the products of the product line. Regarding (1), the capture of common and variable features is commonly done using specific models, such as feature models and domain-specific languages [37].

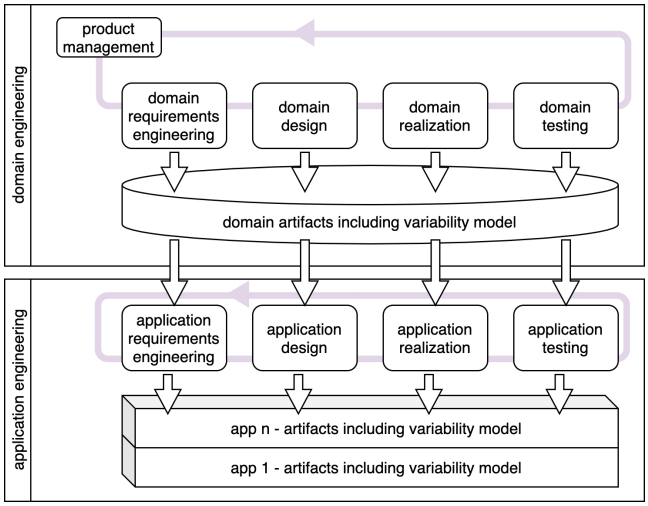

Figure 1. The software product line engineering process according to [34].

Regarding the core assets referenced in (2), that are (re)used in building the products of the product line, these may have diverse support and formats but, usually, take the form of a software platform. As stated by Klaus Pohl et al., "the domain engineering process produces the platform including the commonality of the applications and the variability to support mass customisation" [34]. Finally, (3) is a reference to a required process that needs to be followed to implement a software product line engineering approach.

From this analysis, we may assume that there are three main requirements for establishing a software product line: a process; a supporting platform; and a variability model. From this assumption, we explore, in the next section, the possible relationships between SPL engineering and low-code platforms.

2.2 Relating Low-Code and SPL

Low-code application platforms enable a fast time to market by reducing the amount of hand-coding required for the development of applications [5]. This is possible because the development platform is based on visual programming with a graphical interface as well as model-driven design [1]. The mentioned gains in productivity and ease of use facilitate the adoption of these platforms by non-professional developers and domain experts [8]. Usually, these platforms are also cloud-based and focused on a specific domain, commonly the domain of business applications [18], e.g., Mendix [27], Outsystems [33] and PowerApps [29].

If we compare the previous description of LCAPs with the discussion of SPL engineering in Section 2.1, it becomes clear that there are several common aspects. In fact, both LCAP and SPLs are based on core domain artifacts that are (re)used in the development of each product/application. The difference is that in SPL the development of these core artifacts is included in the overall process, whereas in LCAP these core artifacts are part of the LCAP and, usually, not possible to update by the end user of the LCAP. Also, both are used in the context of a specific domain or set of domains. In the case of LCAPs, there is usually a narrower context, since they are usually constrained to specific technical domains, such as web or mobile business applications. Regarding the process of SPL engineering, or the "prescribed way", as stated by Paul Clements and Linda Northrop [6], this is also a characteristic of LCAPs, since they usually support the overall life cycle of applications, including their deployment, monitoring and update in cloud-based contexts. We argue that the major feature missing in LCAPs is the explicit support for variability modeling, which is fundamental in a software product line.

An LCAP will usually provide a cloud-based environment that its developers use to build applications. LCAPs are also based on models. Developers use one or more models to develop the applications. The editors of such models are usually graphical and based on simple construction blocks that can be combined by drag and drop actions. While the execution platform may differ for each LCAP, if the solution is based on model interpretation, then the execution platform is the runtime required to execute the models. Yet, if there is a code generation approach, then the execution platform may be much more simple or even non-existent.

When building applications, the platform is used, as well as the generated code or produced models. This process has similarities to those used in SPL, where core assets are (re)used. In LCAPs, these core assets that can be reused are the platform and models, or part of them.

Although low-code platforms have some support for reuse, they lack support for variability realization and modeling at a similar level as SPLs. Regarding the similarities between LCAPs and SPLs, we explore an approach to assist the development of SPLs with LCAPs based on domain-specific languages to express both the models of the applications and the variability. We believe this approach can provide insights and lessons on possible integrations between LCAPs and SPLs that could maximize the best of both. Our approach is presented in the next section.

3 Proposed Approach

The approach taken to support SPLs in low-code platforms is orthogonal to the LCAP, not requiring any modification in the LCAP. The only requirement in the LCAP is for it to provide a mechanism that allows importing and exporting models of the applications. Figure 2 illustrates this approach with some detail, in a manner like the one used to represent SPL engineering (see Figure 1). In this way, similarities between LCAP and SPL, as discussed in Section 2.2, become more evident. In Figure 2, the set of DSLs required to support the approach is identified (LC Model DSL; LC DSL; Variability DSL), inside a component identified as SPL IDE

(Integrated Development Environment). The import/export model mechanism of the LCAP is required to communicate models between the LCAP and the SPL IDE.

One major goal of our work was to provide a fast solution to tackle the reuse problems faced by the end users of the project. As such, we selected a model-driven development (MDD) solution. To rapidly produce a proof-of-concept both for stakeholders and end users, the envisaged solution is based on models and model transformations. With this in mind, and because using textual DSLs was a strong requirement for the end users, the Xtext [13] framework for building DSLs was selected. Since Xtext is based on the Eclipse Modeling Framework (EMF) [11] this choice also allowed us the integration with more MDD tools, such as the ATL Transformation Language (ATL) [10] for model transformations. Because our team had previous experience with all the mentioned tools, the development of a proof-of-concept solution was rapidly achieved.

3.1 Process Overview

From a users' viewpoint, the process starts by using the LCAP to create a model that expresses the domain of the SPL, i.e., all the possible applications of the SPL. This is because, as we will discuss later, our solution is based on what is known as *negative variability*, where elements from a whole (i.e., the domain model) are removed based on *presence conditions* that annotate these variable elements. Therefore, we need to start with a model of a complete SPL. This domain model – which is identified as LC Model SPL in Figure 2 – is automatically generated (by the user action) from the LCAP export mechanism. This will usually result in a file, or set of files, in popular data formats, such as JSON or XML. Despite being popular data formats, these files, and their support in IDEs, are not adequate or understandable enough for being edited by end users. We require a *full* language support in the IDE, such as the ones provided by the Xtext framework. Therefore, the domain model is converted into a new low-code DSL (LC DSL). This full language must support variability annotations and, thus, users can mark language elements with the presence conditions mentioned previously. Hence, a language for expressing the variability of the SPL is also required. We call this language Variability DSL. By using the Variability DSL, the user can now model the variability of the SPL as well as use it to annotate the LC DSL with presence conditions that represent all the possible applications of the domain. To complete the process, the users can use the Variability DSL to create application configurations. These will include only elements that satisfy the presence condition for a specific application of the SPL. Then, the App Generation process in Figure 2 is used to generate models of applications by removing elements from the domain model that are not present in the configuration. The resulting DSL is then converted to the LCAP model format and imported into the LCAP. In the LCAP, the user can then use the features of the platform to generate and deploy the application.

3.2 LC-Model DSL

The approach proposed in this paper has two main characteristics: it is orthogonal to the LCAP and it is based on MDD. By being orthogonal, it does not require any modifications in the LCAP. It only requires that the LCAP has some sort of import/export mechanism for the application models. LCAPs usually provide such mechanisms and use popular data file formats, such as JSON and XML. In Figure 2, we identify these files as LC Model (either LC Model SPL or LC Model App).

However, since our approach is MDD, we want to use, as much as possible, models and transformations. As previously mentioned, we use Xtext and the modeling framework it is based upon, EMF. These are integrated at a very deep level. Xtext uses EMF models as the underlying semantic models for its DSLs and, automatically, provides the DSLs representations as persistence formats for EMF models. Therefore, all the tools that use EMF models can also use Xtext DSLs as EMF Models. As such, by using Xtext DSLs we could also use any of the model transformation tools compatible with EMF. For its declarative capabilities, we selected ATL as the model-to-model transformation tool used in the implementation of the approach.

As a result of these options, we produced an Xtext DSL to support the grammar of the LC Model. We call it the LC Model DSL. In this way, it is possible to directly use the data files that compose the LC Model, in model transformations that convert them into/from instances of the new LC DSL.

3.3 Variability Metamodel and DSL

To model variability and express it in a DSL our choice was to design a specific metamodel for variability using EMF and use Xtext to automatically generate the grammar for the DSL. Therefore, the metamodel is specific to the modeled SPL. This comes with some limitations because every single SPL requires a specific metamodel and will result in a specific variability DSL. However, this also enables some interesting advantages: (1) a very user-friendly syntax can be used for the variability DSL, simplifying users' work; (2) the expressiveness of the language is not limited as some more conventional variability metamodels, such as feature models. Regarding (2), we have followed an approach to variability as discussed in [37].

Figure 3 illustrates the variability metamodel for our case study. EMF metamodels are treated as class models, and we use their features to express variability. For instance, in Figure 3, we see how a mandatory feature Orders was modeled as an abstract class that has two child features (subclasses): Sales and Purchases. Since the cardinality of the containment relationship to Orders is 1..2, this means that the SPL can have one or both subfeatures. This metamodeling approach

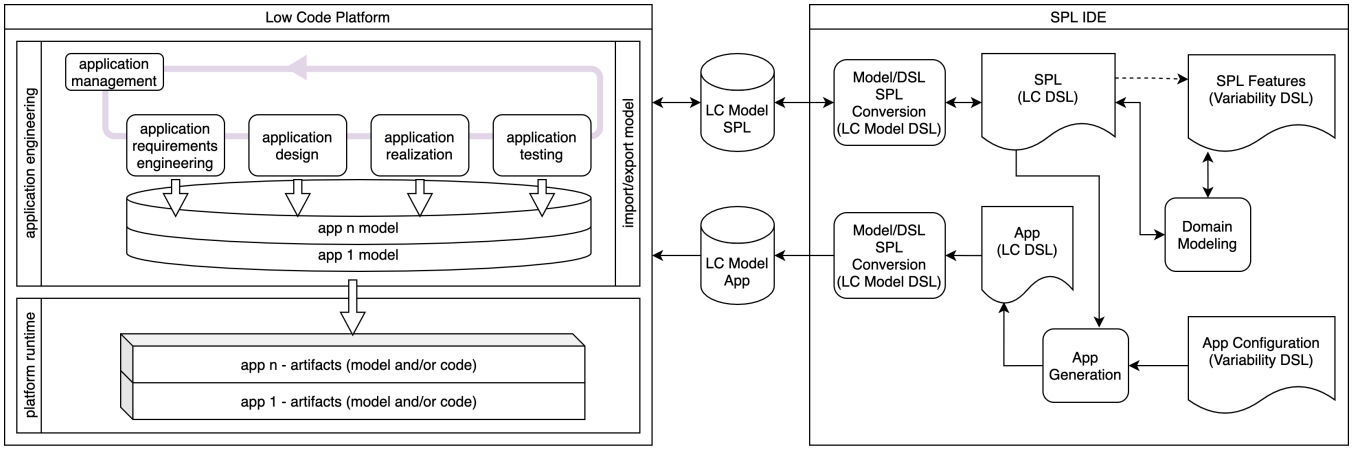

Figure 2. Approach to integrate low-code platforms and SPL engineering.

allows more interesting possibilities, such as using attributes to enrich the variability model. One example presented also in Figure 3 is the teamManagerLimit attribute that is used to express the purchase monetary limit that the role of team manager of the application is allowed to approve. Also, using metamodels allows the use of constraints in the metamodel using, for instance, the OCL language [12] that is supported by EMF.

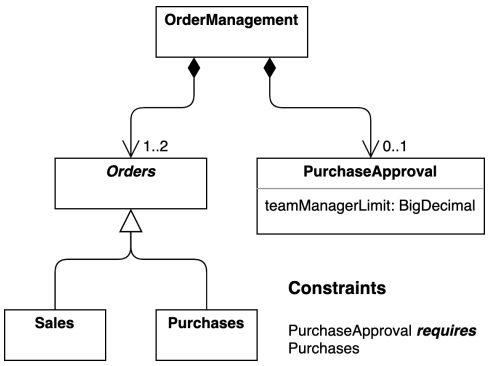

Figure 3. A simplified variability metamodel for an order management SPL.

3.4 Deducing the Low-Code Metamodel

One of the main tasks of the approach is the creation of the DSL for the LCAP. This DSL must be able to express all the elements of all possible exported models of the LCAP. The problem with this task is that LCAPs usually do not give access to their metamodel. They usually provide export and import functionalities, but only for models of applications, not for their metamodels. To circumvent this limitation, we try, as much as possible, to infer, or deduce, the metamodel by inspecting the contents of exemplary models of the LCAP. Finding the structure of the composition of elements may be simple, as they are explicit in the structure of the models (i.e., the structure of the JSON or XML file). Even the identification of primitive data types may be achieved automatically. However, things will become more complex as we try to identify references between elements, hierarchy relationships, or even deduce the cardinality of a relationship. This problem has been discussed and several approaches have been proposed [19, 20, 39]. However, to our knowledge, there is no complete automatic solution for the problem. A semi-automatic solution was devised in line with that already used by other authors, such as Izquierdo and Cabot [19]. Figure 4 illustrates our approach to deducing the metamodel of the LCAP and, from that, generating a DSL.

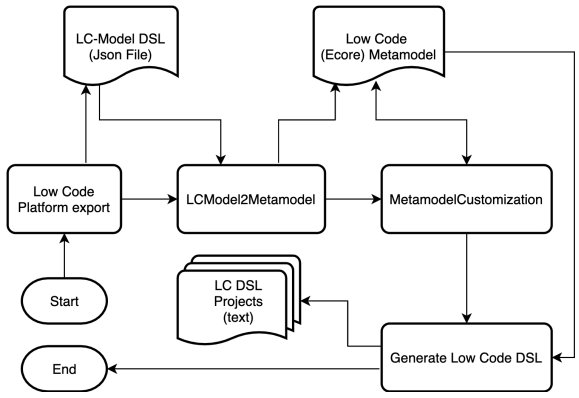

Figure 4. The process of deducing the low-code metamodel and generating the low-code DSL.

The process starts by exporting the model of the SPL from the LCAP. As stated previously, the model is exported in a common data file format, such as a JSON document (or a set of JSON documents). As described in Section 3.2, this document format is treated as an EMF model by handling it as an instance of the LC-Model DSL (developed with Xtext). This allows for a first model-to-model transformation (i.e., LCModel2Metamodel identified in Figure 4) that takes as

input the LC-Model and produces the first version of a low-code metamodel.

3.4.1 LCModel2Metamodel. LCModel2Metamodel is a model-to-model transformation that uses ATL. The goal of this transformation is to create the first version of a metamodel for the LCAP. The transformation is based on the following main rules:

- Each object or array of objects becomes an EClass in the resulting metamodel;
- Each containment in the origin model results in a containment relationship in the resulting metamodel (i.e., an EReference). The process tries to deduce the cardinality of the relationship by the number of instances in the input model;
- For elements of simple data types, the result is a compatible EAttribute.

These rules automatically produce a valid metamodel. It is valid since it can be used as the base for constructing a DSL with Xtext, and this DSL could be used for expressing the models of the SPL. However, the metamodel is probably incomplete, since the process is not capable of automatically identifying specifics such as the correct cardinality of the relationships, and the hierarchy between EClasses or elements that are wrongly identified as strings or arrays of strings, not references.

3.4.2 MetamodelCustomization. For solving the previously identified issues, the process includes a semi-automatic task called MetamodelCustomization. This is also a task supported by ATL. The idea for the task is to have an expert on the LCAP to verify and correct the resulting metamodel. This needs to be done, at the moment, with the help of someone familiar with the ATL language. The process is based on a *refining transformation*, that applies refactoring rules to the resulting metamodel to remove the issues identified by the LCAP expert. By using this refining model, it is only required to specify the rules for the refactoring since the untouched model elements remain in the resulting model. Some examples of common refactoring rules are:

- Correct non-identified class hierarchies;
- Adding new missing classes (for instance, abstract classes in hierarchies);
- Correcting wrong cardinalities;
- Correct wrongly identified data types.

We call this task semi-automatic because it may also include automatic rules that do not require user intervention to edit them. For instance, we included in the task the refactoring of the metamodel so that all the elements can have *annotations* regarding variability (e.g., *presence conditions*). For that, the resulting metamodel must have a reference to the variability metamodel (see Section 3.3). Also, some new elements are created, and existing elements need to be updated to include the referred *annotations*. After the metamodel is verified and corrected it can be used as input to the creation of the Xtext DSL for the LCAP: LC DSL.

3.5 Low-Code DSL

One of the many advantages of using Xtext is that it provides a series of defaults for language engineering. One of them provides the ability to generate the grammar definition for a metamodel. Given the metamodel that results from the previous task of MetamodelCustomization, it is very simple to generate the low-code DSL. The result of the Xtext wizard should provide a functional implementation of the DSL suitable for working with the Eclipse IDE and, with some additional work, with other IDEs and editors using the Xtext implementation of the language server protocol [28].

For this specific task, we have found a few common issues that may require manual intervention, such as:

- **Customization of terminal rules**. The default implementation of terminal rules may not be suitable for the DSL. For instance, we had to customize the rule that is used to recognize numeric values.
- **Identification of elements**. Xtext assumes a property with *name* as a default to identify elements and support cross-references as well as scope. If this property is missing, it may generate problems with the DSL. The workaround may require customizing the solution to use another property.
- **Order of parsing**. Xtext will generate a grammar that parses the elements of an EClass following a static order, which may not be suitable, but can be fixed by manual editing the parser rules.
- **Variability Annotations**. Since the metamodel of the DSL has references to the variability metamodel, these are also automatically translated to references between the two DSLs. However, some customizations may be justified, for instance, to provide specific graphical highlights in the IDE for the variability annotations that will appear along with the DSL.

One possible approach to avoid the previous issues is to customize the Xtext wizard so that it may follow other conventions when generating the grammar file from a metamodel. However, at the moment, we have not yet explored this possibility, and it remains future work.

3.6 Conversion between LC-Model and the Low-Code DSL

One essential part of the proposed solution is the conversion between the low-code model and the low-code DSL (i.e., *Model/DSL SPL Conversion* in Figure 2). This task provides the conversion of the model of the SPL that comes from the LCAP into the format of the new textual low-code DSL. After that, the user may edit the low-code DSL that models the SPL, for instance, adding variability annotations or any

other type of modification. As depicted in Figure 2, after generating a model for an application of the SPL, the user will want to use this conversion task to make the reverse conversion, from the low-code DSL to the low-code model (in the format expected by the LCAP) and import the new application into the LCAP.

For this task, the goal was to build a generic solution, that depended neither on the format of the data model of the LCAP nor on the specific produced low-code DSL. So, we wanted a model transformation that was independent of both the input and the output models and metamodels. The solution was based on the use of the reflection features of the EMF API. Since, in our approach, there is a direct mapping between elements in the metamodel of the LC-Model and elements in the low-code metamodel of the LC DSL, this approach is feasible. Also, the names used in the elements of both metamodels are identical, which facilitates the transformation.

It should be noted that in the conversion from low-code DSL to LC-Model all the elements regarding variability (i.e., the possible variability annotations) are ignored by the transformation. This is a choice made by design since the configuration process (i.e., the process that generates the model of one application of the SPL) will always result in a model of an application with all variability annotations removed. Besides, even in the situation that the user has edited the LC DSL and wishes to update the SPL model in the LCAP, it makes no sense to convert the variability annotations since the context for this work is that the LCAP has no support for SPL engineering, particularly for variability modeling.

3.7 Configuration Process

The configuration process is a key part of generating an application in the SPL. As depicted in Figure 2, to generate an application in the SPL it is necessary to have the LC DSL annotated with variability annotations (i.e., *presence conditions*) and have an instance of the variability DSL that represents the configuration for the specific application. The Figure 5 shows a concrete example. In the central window of the SPL IDE, the LC DSL is annotated with a presence condition (i.e., *inc* keyword in green) stating that the *SaleOrderForm* will only be included in the generated model of the application if the *SPLOrderMan.Sale* element exists in the configuration model. The configuration model is presented in the Figure at the bottom window of the SPL IDE.

To implement this process, an approach like the one described for DSL metamodel customization was used (see Section 3.4.2). The process is based on the refining mode of ATL. Since in this mode ATL only applies rules to matched elements of the models and leaves the remaining elements untouched, it was used to only deal with elements that have variability annotations and leave all the other elements untouched. The simple rules for the transformation are:

- **Remove Element**. This rule only applies to elements that have variability annotations. It will verify if the variability elements referred to in the annotation are present in the configuration model and, if not, will remove the element from the resulting model.
- **Remove Annotations**. For all the elements with variability annotations for which the include condition is false (i.e., that are not included in the previous rule) the variability annotation is also removed. In this way, the resulting DSL is free from all variability annotations and can the imported into the LCAP.

If required by the variability metamodel, the actual process can also support attributes in the features of a variability model. These attributes usually express values that can be used to customize behavior in SPL, such as in workflows or processes modeled in the LCAP. In the example of Figure 3, the attribute *teamManagerLimit* can the used to model the purchase limit of the team manager in the purchase approval process. At the present moment of our research, these kinds of behaviors are usually expressed using a general programming language (e.g., C#) and codified inside a string in the model exported from the LCAP. Therefore, our process supports a macro string substitution in the model, that will replace referenced attributes inside the strings, such as *teamManagerLimit*, by their specific values in the configuration model. Similar to macros in languages such as C and C++, this macro substitution process is executed before the ATL transformation.

4 Case Study

The incentive for our proposal started from a project whose main goal is to develop a web-based textual DSL for the OMNIA low-code application platform. In the context of such work, it was identified that some (intermediate and advanced) users of the LCAP regularly exported their models to JSON format and, using a regular text editor, modified those models to rapidly obtain similar models, for similar applications. They then imported the edited JSON files into the LCAP to generate the new, similar applications. When this process came to our attention, we searched for a more convenient approach to deal with that kind of issue and reuse technique and the result is what is shared in this paper. The case study presented in this section is the first applied experiment of our approach[1].

This case study is based on a common use case scenario, also used as a tutorial by the OMNIA LCAP, that implements an application for order management, and it is easy to understand. This application may include functionalities such as client management, supplier management, product management, salesperson management, purchasing management

[1]The artifacts are available in https://github.com/BAMoL-ISEP/low-code-sple

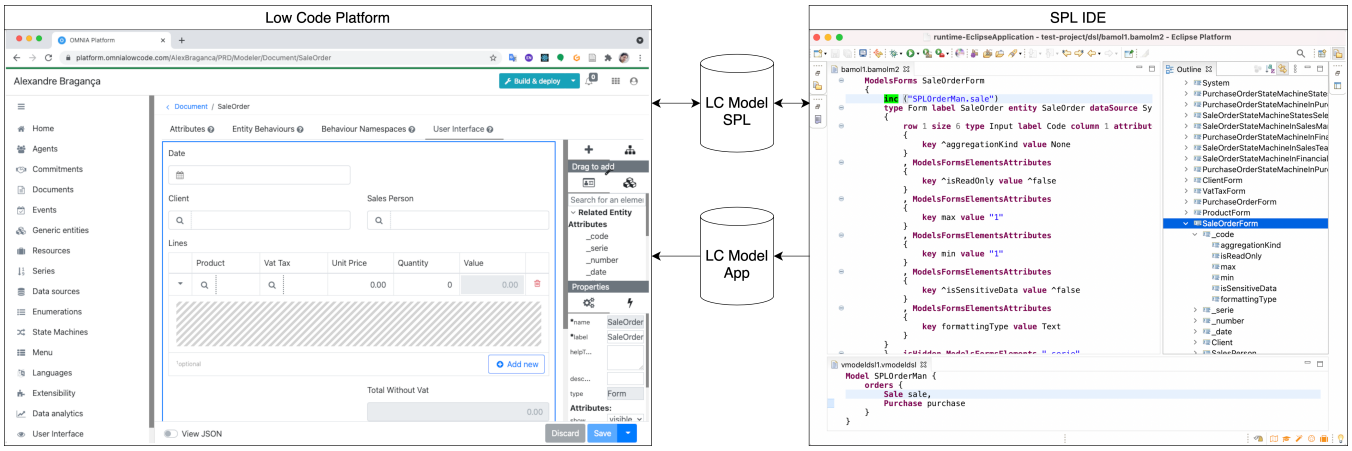

Figure 5. Integration of the low-code platform and the SPL IDE.

from suppliers, customer sales management, and VAT management. However, quite often, the modeled application only includes parts of these functionalities. For instance, an application for sales management will not include functionalities regarding suppliers. Also, behavior such as the workflow for purchase approval may vary according to specific rules of the organization that will use the application. Therefore, this seemed a good case to apply SPL engineering

Figure 5 illustrates the result of the case study implementation. On the left side of the Figure, we see the LCAP platform, that users normally use, without any kind of change. The process does not require changes in the LCAP. Users can use the LCAP to model individual applications and create models that model completely or partially an SPL, without modeling variability, since the LCAP, as is, does not support it. Variability can be modeled with the created SPL IDE, as depicted on the right side of the figure. Within the SPL IDE users may edit the model of the SPL as well as create and edit variability models and application configuration models. Within the low-code DSL, the user may add variability annotations that reference elements in the variability DSL. The user may export the generated application models to the LCAP to verify if they are correct and, eventually, use the LCAP to deploy them.

Table 1 presents some data and measures regarding the main tasks of the case study. Its duration was around 2 months and the longer tasks were the ones fundamentally performed by researchers. These tasks mainly refer to the setup of the core tools that support the SPL engineering process and can be significantly reused for other SPL cases with the same LCAP. Although in this case these tasks were performed by researchers, those essentially require users with knowledge and experience with Xtext and EMF-related tools.

Next, we present the details of the case study implementation. The description is divided into domain engineering and application engineering, although the boundaries between these two contexts can be blurred. For example, there is an initial task that results in the creation of a complete domain model containing all possible applications for this case study. But this was not mandatory, the applications could be added to the model later, in the context of application engineering, using the LCAP or the produced DSL.

4.1 Domain Engineering

This section presents the main tasks of the case study that we consider as being part of the domain engineering process. However, some tasks are more generic and can be reused for other SPLs. These include, for instance, a significant set of Eclipse plugins, that can be easily reused to support other SPLs. This set of plugins – used as a whole – is what we call the SPL IDE. As stated earlier, for constructing the DSLs we used Xtext. ATL was used extensively for model-to-model transformations, also EMF as it is the base modeling framework for Eclipse. Some customizations in Xtext and the ATL projects required the use of the Java programming language.

4.1.1 LCAP Domain Model. Based on the previous behavior of the LCAP users, i.e., using JSON representations of LCAP models as a foundation to build new similar application models, a request was made to the OMNIA team to provide a model that was illustrative of the described situation. The idea was that the OMNIA team produced some kind of domain model in the LCAP, i.e., a model that supported several similar applications. They had to create a model of the domain and model the variability of the domain. A meeting of approximately 2 hours was scheduled to explain the goal of this task and to explain the fundamentals of SPL engineering and provide some references to the OMNIA team. The OMNIA technical team was able to provide an illustrative model for a simple order management SPL, only one week after the request. The team also described

Table 1. Some measures about the tasks performed during the case study

Task	User	Duration	Manual code
LCAP Domain Model	LCAP staff	1 week	NA (modeled in the LCAP)
Variability DSL	Researcher	2 days	NA (ecore metamodel)
Low-Code DSL	Researcher	1 month	˜400 LOC ATL + ˜80 LOC Java
Conversions	Researcher	3 weeks	˜1000 LOC Java
Configuration	Researcher	1 week	˜30 LOC ATL + ˜50 LOC Java
Application Generation	End User	minutes	˜10 LOC Variability DSL

Notes: The symbol ˜ means "approximately". LOC stands for "Lines of Code". NA stands for "Not Applicable".

the variability related to that model in a natural language format.

4.1.2 Variability DSL. From the natural language description of the variability associated with the domain model, a metamodel was achieved and, after that, a textual DSL to model that kind of variability. This was easily done in about 2 days. An EMF modeling project was used to create the variability metamodel by creating and editing the ecore metamodel (i.e., the metamodel format used in EMF). After that, the variability DSL was built using Xtext. Xtext was able to generate a functional DSL based on the previous metamodel and we did not need to customize any code generated with Xtext.

4.1.3 Low-Code DSL. For the approach of an SPL IDE to be successful, a DSL that models what is possible with the LCAP was needed, but also its integration with the variability DSL. Thus, users were able to annotate the models with variability *presence conditions* (as discussed in Sections 3.6 and 3.7). A semi-automatic process was devised to deduce a metamodel from the JSON models of the LCAP, such as discussed in Section 3.4. To use ATL to transform the JSON models into the metamodel of the new DSL we also needed to create an Xtext DSL for JSON (as described in Section 3.2). As discussed in Section 3.4.2, the process to create the new low-code DSL is not fully automatic. We required the help of the LCAP staff to gather metadata that was not available in the JSON models, such as identification of class hierarchies and some date data types (namely references). We called this activity metamodel customization and we also used it to include in the LC DSL metamodel references to the variability metamodel. The model-to-model transformations (i.e., LCModel2Metamodel and MetamodelCustomization, as depicted in Figure 4) were developed using ATL. With the resulting metamodel, Xtext was used to implement the low-code DSL. Here, some customization of the Xtext generated code was necessary, for instance, to customize the format used for the *presence conditions*. The duration for all these activities was, approximately, 1 month.

Note that the produced DSL can be used for any SPL based on the LCAP since its syntax should support all the modeling capabilities of the LCAP. However, the variability DSL can only express variability for the SPL of the case study.

4.1.4 Conversions. This project required some conversions between files and models. Particularly, the LCAP was only able to export and import models of applications using JSON. Therefore, two processes were required: one to convert from JSON to the low-code DSL and another to do the reverse. Details of these processes are included in Section 3.6. In this case, the implementation was based on the EMF reflection API that enabled us to convert dynamically between the two DSLs, using the metamodels as guidance. The language used was Java and the activity took, approximately, 3 weeks. These processes also included some minor activities, such as converting to and from the specific JSON multiple file organization that the LCAP requires.

4.1.5 Configuration. The configuration process supports the activity of generating one single application of the SPL. Basically, given a model of the domain of the SPL that includes variability annotations and a specific variability configuration model for one application, it can generate the resulting model for that specific application. The process is based on a refining ATL transformation, as described in Section 3.7. The ATL transformation depends also on some Java code that implements the verification of the *presence conditions* as well as deals with the substitution of feature attributes. At least for this use case, the process was very simple to implement using ATL and required little Java code. The process took roughly 1 week.

4.2 Application Engineering

This section refers to tasks during the case study that we deem related to application engineering. Since, to prepare for the case study, the OMNIA staff produced a complete domain model for all the SPL, there was only one specific task exclusively related to application engineering, the task of application generation. In another scenario, for another case

study, users usually would also update the domain model because of requirements for new applications.

4.2.1 Application Generation. Application generation is the process that automatically generates one application of the SPL by applying a variability configuration model to the SPL domain model. Given the existing SPL domain model with variability annotations, users must create a variability configuration model (using the variability DSL) containing the specific variability configuration for the application they wish to generate. Then, the configuration process is invoked (see Sections 3.7 and 4.1.5) to generate the application. This process takes as input the SPL domain model and one variability configuration model and results in a model for the specific application. This model can then be imported into the LCAP for verification and, possibly, deployment. For this case study, the editing of the variability configuration model was done by the users in few minutes.

5 Discussion

When applying the approach to the case study, important issues were found, and some lessons were learned. In Section 5.1, the evaluation of the approach is discussed, referring to its main contributions as well as possible issues and limitations to its application. These are further analyzed in Section 5.2, where specific issues and lessons learned are presented.

5.1 Evaluation

The main goal of the approach presented in this paper was to provide an SPL IDE that could be used to implement SPL engineering based on LCAPs. As presented earlier, this was achieved for the case study. The users that were using very simple mechanisms before, such as copy and paste in JSON files, are now using the created SPL IDE to build similar applications, using an approach based on SPL engineering. These users are now able to edit their models either in the LCAP or using the low-code DSL in the SPL IDE. This DSL provides much more support, hints, and verifications when compared to editing a JSON file (e.g., propose referenced elements or type validation). Also, users can model variability, annotate the domain models with presence conditions, and automatically generate applications of the SPL that are conformant with configuration models, all this without any change in the LCAP.

Another promising measure is the total time required to apply the approach, as presented in Table 1. We are aware that the case study is simple and, as such, its implementation may be, naturally faster. However, the tasks that took more time (such as the process to build the low-code DSL or the process to make model/file conversions), are tasks with the potential to be reused, almost as they are, in other contexts, with other LCAPs. So, we consider this as an advantage of our approach. However, for the time being, we have not done any other complete experiences with other LCAPs to validate this claim but, this is in our short-term plans.

5.2 Lessons Learned

In this section, some of the most important lessons learned from the case study are presented.

5.2.1 LCAP Independency. One of the main characteristics of the approach is that it does not require any change in the LCAP, i.e., it is orthogonal to the LCAP. This option is by design since we aim at exploring a solution that can be used with as many LCAPs as possible. Also, our research project with OMNIA - to build a web-based textual DSL - required that the DSL could be used outside the LCAP, in as many IDEs as possible (with a focus on Visual Studio Code). This brings some advantages, such as being possible to apply the approach to many LCAPs, most of them being closed source projects. Nevertheless, this also limits the way the solution can be integrated with the LCAP. For instance, at the moment, we integrate only via import and export mechanisms of the LCAP. If the LCAP provides other types of integration, such as REST APIs that enable a more granular edition of LCAP application models, then it is possible to provide more integration, for instance, updating the LCAP model as the user edits fragments of the low-code DSL in the SPL IDE. But there is a limit to what it is possible to integrate when we cannot change the LCAP, as in our proposed approach.

5.2.2 Target Users. LCAPs are usually targeted at citizen developers, especially those platforms that are classified as no-code. Since our approach is based on textual DSLs and the use of an IDE, this is arguably the best context for those kinds of users. However, LCAPs are also in evolution, and some are offering more powerful features focusing on more professional developers [22]. Eventually, at a certain point, LCAP applications evolve to more complex solutions that will also require more advanced tools and professional users. We only had one case study to evaluate our solution. In this case study, the users were senior consultants at an ERP (Enterprise Resource Planning) software company. Although they are not professional developers, they are users with several years of experience working with tools such as ERPs and LCAPs. They were able to explore and use our solution without any major problems. Nonetheless, it is doubtful that citizen developers could have done the same, at least without specific training.

5.2.3 Solution Replication. The proposed solution may only be replicated if the LCAP supports an import/export mechanism for the application models. This is the only strong requirement. To our knowledge, several LCAP solutions offer such a mechanism. Some minor requirements may impact the replication of this solution. For instance, the approach requires the construction of textual DSLs and their IDE plugins. This was done using Xtext, but other language workbenches

could be used. In all cases, strong expertise in such workbenches is required. This may be minimized by applying automatic transformations, such as the ones presented in the paper for deducing metamodels or generating the grammar of the DSLs. These will require specific expertise to be built but after that could be reused by less specialized users.

A global and ready-to-use solution applicable to any LCAP can be achieved with the approach described in this article, when standards that make possible to address interoperability between multiple platforms became available. Our proposal could even facilitate platforms' interoperability, something that currently is hardly possible [35].

6 Related Work

LCAPs are just now starting to capture the attention of the scientific community [24, 36]. Low-code can be seen as a synonym of model-driven development [5]. In fact, LCAPs use models and model transformations to generate applications. This is like MDD. One could argue that the major difference is that in LCAPs the metamodels, languages, and transformations used are not exposed and remain static, whereas in a more conventional MDD solution it is possible to access and change them. Although, to our knowledge, there are no approaches that directly integrate LCAPs and SPL engineering, we will address related work that refers to SPL engineering and MDD since, as stated before, LCAPs can be seen as a specific MDD solution.

The development of software product lines with a model-driven approach is not new [2, 7, 21]. Gomaa authored one of the first books on designing SPLs with the Unified Modeling Language (UML) [15]. Since then, several works on the theme have been published. SPLs rely on modeling variability, especially by using feature models, and its integration with design models, such as UML, has been explored [2, 3, 9, 16, 32]. In our proposal, domain-specific languages were used to model variability, as opposed to using feature models.

Markus Voelter and Eelco Visser explore the use of domain-specific languages to model variability [38]. They argue that feature models, because of their strict metamodel, provide a particular advantage, since they can be mapped to logic, and SAT solvers can be used to check valid configurations or provide automatic completion for partial configurations. In contrast, the use of DSLs to express variability provides support for general variability, repetition, nesting, references between elements of the variability model, and modeling variability related to the behavior of the system. The authors conclude that DSLs fill the gap between feature models and programming languages. In our approach, we use DSLs to model variability, similarly. We also find that one of the main advantages of this approach is the possibility to specify the concrete syntax of the variability language, providing a way to adapt it to the user of the LCAP.

MDD approaches to SPL engineering rely on using standard modeling languages, such as UML, or domain-specific models (or languages) for which the metamodel is known. This is a requirement to achieve traceability between models and to implement model transformations [2]. Since LCAPs do not provide access to their metamodels, we explore an approach to help deduce the metamodels from existing models exported from the LCAP. The idea is to construct a metamodel by inspecting models and try to deduce metadata, such as datatypes, cardinalities, or references. Similar approaches were presented earlier by other authors, such as [4, 19, 20, 39]. One problem with deducing metamodels from data models, such as JSON, is that some metadata is almost impossible to deduce, such as class hierarchies (not represented in JSON) or references between classes (usually, the reference is represented as a string value or array). In our approach, a manual step for metamodel customization was used so that, with the help of LCAP experts, such metadata could be added (if required).

Medeiros *et al.* propose an approach to apply MDD to SPLs [26]. The toolchain is like ours, but the approach is based on using architecture description languages to model the base architecture of the domain. They use feature models to express variability and generate product models and source code. In their work, they do not have to deduce the metamodel of the domain (as we do). Their solution is mainly forward-only engineering, while our is more a combination of forward and reverse engineering. Horcas *et al.* presented a solution for the development of web-based SPLs that is focused on support for variability annotations in several file artifacts used in web applications [17]. Our approach differs from theirs since our focus is on annotating the new DSL of the LCAP. They use variability annotations inside comments in the diverse artifacts (e.g., HTML or JavaScript) while we extend the DSL of the LCAP with variability elements. As such, while dealing with models, variability elements are treated in the same way as regular elements of the models (we do not use comments to incorporate variability). After the configuration of an application, all variability elements are removed from the model. As a result, application models imported to the LCAP do not contain variability.

7 Conclusion

This paper presents an approach that provides support for SPL engineering in low-code application platforms. The approach does not impose any change in the LCAP and only requires that the LCAP has some mechanism to import and export application models.

The solution is based on a set of DSLs and respective plugins to be used in, what we call, an SPL IDE. One DSL is required to model the low-code platform. This DSL is produced semi-automatically, deducing part of its metamodel by inspecting models exported from the LCAP. This DSL is

also manually customized to incorporate metadata that is not automatically identified and to integrate with a second DSL that deals with variability. The variability DSL is used to express the variability of the SPL as well as express configuration models specific to particular applications, or products, of the SPL. There may be also supporting DSLs, such as DSLs to manipulate the data that is exported and imported from the LCAP (e.g., JSON files). Once all these tools are available, the LCAP users may model the domain and applications of the SPL using either the LCAP or the SPL IDE. Variability modeling and application configuration are available only in the SPL IDE.

A case study is presented and discussed. It addresses the application of the approach to OMNIA, an LCAP of a startup company, and its results are very promising. The duration of the case study was about 2 months, including the time to develop all the supporting tools. The users were able to implement an SPL for a scenario that before was only roughly addressed by elementary reuse mechanisms.

There are, however, topics that will be addressed as near future work, such as validating the approach with more case studies that include other low-code application platforms and user types. We also plan on evolving the metamodel deduction mechanism, for instance, by providing hints to possible reference relationships as well as improve the user interface in the customization of metamodels. Supporting some form of customization for the syntax of the DSLs is planned to make the process more independent from the capabilities offered by the language workbench.

Acknowledgments

This work is supported by "Fundo Europeu de Desenvolvimento Regional (FEDER)" funds through the "Programa Operacional Competividade e Internacionalização and Portugal2020" program, under the project BAMoL Low-Code Platform and the consortium BAMoL – LCP (POCI-01-0247-FEDER-39661).

References

[1] M. Al Alamin, S. Malakar, G. Uddin, S. Afroz, T. Haider, and A. Iqbal. 2021. An Empirical Study of Developer Discussions on Low-Code Software Development Challenges. In *2021 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) (MSR)*. IEEE Computer Society, Los Alamitos, CA, USA, 46–57. https://doi.org/10.1109/MSR52588.2021.00018

[2] Nicolas Anquetil, Birgit Grammel, Ismenia Galvao, Joost Noppen, Safoora Shakil Khan, Hugo Arboleda, Awais Rashid, and Alessandro Garcia. 2008. Traceability for Model Driven, Software Product Line Engineering. In *ECMDA Traceability Workshop (ECMDA-TW) 2008*. SINTEF, 77–86. ECMDA Traceability Workshop, ECMDA-TW 2008, ECMDA-TW ; Conference date: 12-06-2008 Through 12-06-2008.

[3] Alexandre Braganca and Ricardo J. Machado. 2007. Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines. In *11th International Software Product Line Conference (SPLC 2007)*. 3–12. https://doi.org/10.1109/SPLINE.2007.17

[4] Alexandre Bragança and Ricardo J. Machado. 2008. Transformation Patterns for Multi-Staged Model Driven Software Development. In *Proceedings of the 2008 12th International Software Product Line Conference (SPLC '08)*. IEEE Computer Society, USA, 329–338. https://doi.org/10.1109/SPLC.2008.41

[5] Jordi Cabot. 2020. Positioning of the Low-Code Movement within the Field of Model-Driven Engineering. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 76, 3 pages. https://doi.org/10.1145/3417990.3420210

[6] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.

[7] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. 2005. Model-Driven Software Product Lines. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 126–127. https://doi.org/10.1145/1094855.1094896

[8] Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. 2020. Democratizing the Development of Recommender Systems by Means of Low-Code Platforms. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 68, 9 pages. https://doi.org/10.1145/3417990.3420202

[9] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. 2005. The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations. *9th International Conference on Software Product Lines, SPLC'06*, 33–44. https://doi.org/10.1007/11554844_5

[10] Eclipse Foundation. 2021. *ATL - ATL Transformation Language*. Retrieved June 8, 2021 from https://www.eclipse.org/atl/

[11] Eclipse Foundation. 2021. *EMF - Eclipse Modeling Framework*. Retrieved June 8, 2021 from https://www.eclipse.org/modeling/emf/

[12] Eclipse Foundation. 2021. *OCL Language*. Retrieved June 8, 2021 from https://projects.eclipse.org/projects/modeling.mdt.ocl

[13] Eclipse Foundation. 2021. *Xtext*. Retrieved June 8, 2021 from https://www.eclipse.org/Xtext/

[14] Gartner. 2021. *Gartner Glossary, Citizen Developer*. Retrieved June 8, 2021 from https://www.gartner.com/en/information-technology/glossary/citizen-developer

[15] Hassan Gomaa. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., USA.

[16] Salvador Trujillo Gonzalez. 2007. *Feature Oriented Model Driven Product Lines*. Ph.D. Dissertation. PhD thesis, Department of Computer Sciences, University of the Basque Country.

[17] Jose-Miguel Horcas, Alejandro Cortiñas, Lidia Fuentes, and Miguel R. Luaces. 2018. Integrating the Common Variability Language with Multilanguage Annotations for Web Engineering. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. Association for Computing Machinery, New York, NY, USA, 196–207. https://doi.org/10.1145/3233027.3233049

[18] Felicien Ihirwe, Davide Di Ruscio, Silvia Mazzini, Pierluigi Pierini, and Alfonso Pierantonio. 2020. Low-Code Engineering for Internet of Things: A State of Research. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, New York, NY, USA, Article 74, 8 pages. https://doi.org/10.1145/3417990.3420208

[19] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2013. Discovering Implicit Schemas in JSON Data. In *Proceedings of the 13th International Conference on Web Engineering* (Aalborg, Denmark) *(ICWE'13)*. Springer-Verlag, Berlin, Heidelberg, 68–83. https://doi.org/10.1007/

978-3-642-39200-9_8

[20] Faizan Javed, Marjan Mernik, Jeff Gray, and Barrett R. Bryant. 2008. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology* 50, 9 (2008), 948–968. https://doi.org/10.1016/j.infsof.2007.08.003

[21] Jean-Marc Jézéquel. 2012. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering* 2012 (12 2012). https://doi.org/10.5402/2012/670803

[22] Frederic Lardinois. 2021. *Microsoft launches Power Fx, a new open source low-code language.* https://techcrunch.com/2021/03/02/microsoft-launches-power-fx-a-new-open-source-low-code-language-for-its-power-platform/

[23] Frank Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* https://doi.org/10.1007/978-3-540-71437-8

[24] lowcomote consortium. 2021. *Lowcomote objectives.* Retrieved June 8, 2021 from https://www.lowcomote.eu/objectives/

[25] MarketsandMarkets. 2021. *Low-Code Development Platform Market by Component (Platform and Services), Application Type, Deployment Type (Cloud and On-Premises), Organization Size (SMEs and Large Enterprises), Industry, and Region - Global Forecast to 2025.* Retrieved June 8, 2021 from https://www.marketsandmarkets.com/Market-Reports/low-code-development-platforms-market-103455110.html

[26] Ana Luisa Medeiros, Everton Cavalcante, Thais Batista, and Eduardo Silva. 2015. ArchSPL-MDD: An ADL-Based Model-Driven Strategy for Automatic Variability Management. In *2015 IX Brazilian Symposium on Components, Architectures and Reuse Software.* 120–129. https://doi.org/10.1109/SBCARS.2015.23

[27] Mendix. 2021. *Mendix.* Retrieved June 8, 2021 from https://www.mendix.com

[28] Microsoft. 2021. *Language server protocol.* Retrieved June 8, 2021 from https://microsoft.github.io/language-server-protocol/

[29] Microsoft. 2021. *PowerApps.* Retrieved June 8, 2021 from https://powerapss.microsoft.com

[30] Yefim Natis, Jason Wong, Akash Jain, Saikat Ray, Paul Vincent, Adrian Leow, and Kimihiko Iijima. 2019. Magic Quadrant for Enterprise Low-Code Application Platforms. (August 2019). https://www.gartner.com/en/documents/3956079/magic-quadrant-for-enterprise-low-code-application-platf

[31] NumbersBelieve. 2021. *OMNIA Low Code Platform.* Retrieved June 8, 2021 from https://omnialowcode.com

[32] Sami Ouali, Naoufel Kraiem, Zuhoor Al-Khanjari, and Youcef Baghdadi. 2013. A Model Driven Software Product Line Process for Developing Applications. *First International Workshop on Variability Support in Information Systems (VarIS) (to be held in conjunction with CAiSE 2013)* 148. https://doi.org/10.1007/978-3-642-38490-5_40

[33] OutSystems. 2021. *OutSystems.* Retrieved June 8, 2021 from https://www.outsystems.com

[34] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, Berlin. https://doi.org/10.1007/3-540-28901-1

[35] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. 2020. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* 171–178. https://doi.org/10.1109/SEAA51224.2020.00036

[36] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan De Lara, Esther M Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms. In *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019) (CEUR Workshop Proceedings (CEUR-WS.org)).* Eindhoven, Netherlands. https://hal.archives-ouvertes.fr/hal-02363416

[37] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org. 1–558 pages.

[38] Markus Voelter and Eelco Visser. 2011. Product Line Engineering Using Domain-Specific Languages. In *2011 15th International Software Product Line Conference.* 70–79. https://doi.org/10.1109/SPLC.2011.25

[39] Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S Kolovos, and Richard F Paige. 2019. Type inference in flexible model-driven engineering using classification algorithms. *Software & Systems Modeling* 18, 1 (2019), 345–366. https://doi.org/10.1007/s10270-018-0658-5