Designing Domain Architectures for Model-Driven Engineering

Daniel Lucrédio*, Renata P. M. Fortes†, Eduardo S. Almeida‡ and Silvio L. Meira§

*Computing Department - Federal University of São Carlos, São Carlos - SP - Brazil
†Institute of Mathematical and Computing Sciences - University of São Paulo, São Carlos - SP - Brazil
‡Computing Science Department - Federal University of Bahia, Salvador - BA - Brazil
§Recife Center for Advanced Studies and Systems - Recife, PE, Brasil

Corresponding author: daniel@dc.ufscar.br

Abstract—Model-Driven Engineering (MDE) can leverage domain engineering by offering support to complex variability and automatic implementation. However, little attention is given to the process of designing a domain architecture that is well suited to MDE techniques such as domain-specific languages and software transformations. A domain-specific software architecture is normally developed based on a few selected and important requirements, called architectural drivers. This paper presents three types of architectural drivers that can be used to build a software architecture that can take full advantage of the benefits of MDE. It also presents some patterns that can be used to help in the architectural design. An evaluation is also presented, showing that, when used together in a model-driven domain engineering project, these drivers and patterns can lead to some benefits in terms of reusability and complexity, but that in some cases there are drawbacks that need to be considered in a trade-off analysis.

Keywords-model-driven engineering; domain engineering; architectural drivers; patterns; domain-specific languages; code generation;

I. INTRODUCTION

One of the main tasks of domain engineering is to define a domain-specific software architecture that supports commonality and variability [1]. The architecture must not only predict the variation points, but also effectively provide the required support, normally in the form of reusable components [2]. Model-Driven Engineering (MDE) can extend this process with domain-specific languages and transformations that provide support for more complex variability and automated implementation.

To make this model-driven domain engineering (MDDE) scenario possible, the software architecture must be built around the concept of MDE, being aware and giving support to its technologies, so that the application development process can take full advantage of its benefits.

However, most model-driven approaches for domain engineering and product line engineering focus on the product derivation task, after the architecture is already defined (two examples are [3], [4]). In general, not enough attention is given to the process of designing an architecture that is well suited for MDE. We attempt to address this issue in two ways: (i) by identifying three types of MDE-specific architectural drivers [2]; and (ii) by presenting some patterns to deal with them. When used together, they facilitate the task of designing architectures for MDE. The drivers guide the domain architect through the identification of the different domain variability possibilities, while the patterns help by providing solutions to some of the problems involved in model-driven domain design.

We also present the results of an evaluation conducted to demonstrate the viability, the benefits and drawbacks of using these drivers and patterns.

II. DOMAIN VARIABILITY AND MODEL-DRIVEN ENGINEERING

Variability is the key concept behind a successful domain architecture [1]. There are two kinds of domain variability [6]: routine configuration - a subset of features is selected to configure a product, using tree-like structures; and creative construction - models describe the variation through graph-like structures.

Most domain design approaches rely on the identification of variability as described in terms of features [7] – externally observable characteristics of a product. Feature modeling is located somewhere between routine configuration and creative construction [6], and has already been used in many successful cases. However, there are other kinds of variation, such as in entity models [8], which are more complex than it is possible to capture in feature models, because they are too generic [3]. These require a richer mechanism, with more expressive power to capture detailed relationships and constraints between entities. The technology of choice for this is normally a Domain-Specific Language (DSL) [3], [6]. Together with model transformations and code generators, DSLs form the basis of Model-Driven Engineering (MDE).

For example, consider a web authoring domain involving publish-navigate applications. An administrator publishes information, such as news, posts, messages, etc. that can be accessed by users. News websites, forums and blogs are some example applications of this domain. Figure 1A shows a feature model for this domain. Using this model, it is possible, for example, to configure a product by selecting optional features like simple search and user content submission (Figure 1B). But the four domain technology features in the bottom of Figure 1A cannot be merely selected or
deselected. Instead, they need to be combined in a creative way to configure a product (Figure 1C).

This is where a DSL comes in hand. It is a language that describes the domain concepts and how they can relate to each other. Using a DSL, one can create models that capture more detailed variability. In the previous example, a domain designer could use a web authoring DSL to specify which document types will be present in an application, and how they relate to each other, or which pages will be available, and how navigation between them occurs.

Since a domain usually comprises both types of variability, it can be divided into subdomains [9], each one described in terms of feature-based variability (routine configuration) or DSL-based variability (creative construction). Also, these different subdomains must effectively cooperate in order to support the variability, i.e. the individual feature models and DSLs must be integrated [10], [11].

III. DESIGNING A DOMAIN-SPECIFIC SOFTWARE ARCHITECTURE FOR MDE

The process of designing a domain-specific software architecture is usually carried out by an architect, together with the stakeholders, who can identify the main quality attributes that are important for a particular domain [2]. These are called architectural drivers - a combination of functional and quality requirements that “shape” the architecture. To identify them, the most important business objectives must be analyzed, and those that have most impact in the architecture must be chosen as drivers [2].

For Model-Driven Domain Engineering, we defined three types of architectural drivers that must be identified: (i) Feature-based variability, (ii) DSL-based variability and (iii) Subdomain integration. In the following sections we discuss some techniques to help to identify them. In the following discussions, we assume that domain analysis has already been conducted, so a feature model is available at this point.

A. Identification of feature-based variability drivers

Domain features identification and variability analysis is normally performed during domain analysis [12], [13]. So, at this point the domain architect already has a good idea about the similarities between domain applications and how they may differ from each other.

But for the architectural design to begin, more concrete information is needed. The architect needs to know details about what happens when a particular feature is present or absent, when two related features are combined, etc. A common technique for this particular task is to use scenarios. A scenario is a brief description of behavior that illustrates certain conditions - like the presence of a particular feature - and serves to test an architecture’s capacity in satisfying one or more quality attributes [2].

There are different ways to map the variability at the features level into scenarios that describe the variability. For example, use cases with inclusion and exclusion relationships or special tags that define variation points can be used [14]. Here we propose the use of another concept to derive these scenarios, called change cases [15].

A change case is a scenario that describes a variation point considering the presence of one or more variants. For example, consider a domain featuring simple and advanced search options. Figure 2A describes the use case for simple search, and Figure 2B describes advanced search. Note that, in Figure 2B, steps 2, 3 and 4 substitute the “normal” behavior when the advanced search variant is present. The related features and the affected scenarios are also registered, to indicate where the changes occur.

Change cases have the advantage of representing complete scenarios. This facilitates architectural evaluation, because the stakeholders can fully understand how a scenario works without having to interpret specific tags or analyze combinations between alternatives. However, they result in longer descriptions and some duplication.
B. Identification of DSL-based variability drivers

Some authors have proposed extensions to the basic feature modeling technique for this purpose [4]. However, even the extended feature model has a fixed, generic notation. This may not be a problem in the beginning, during variability analysis, but later in the MDE process this generic notation is unable to represent all the complexities and details required for code generation [3].

DSL development is a complex, time-consuming task [16], and thus it is not viable to build a complete DSL just for architectural design purposes. But a DSL’s underlying structure - its abstract syntax, or metamodel [17] - can be used to formalize the variation space in particular areas of the domain (subdomains). It can introduce constraints and rules that go beyond the information contained in the feature model, defining more precisely how the applications vary in these areas, up to a point where design can start. Differently from feature-based variability, scenarios are not useful here, because there are infinite variation possibilities.

Thus, we propose the identification and development of DSL metamodels to be used as architectural drivers. The identification of a DSL’s metamodel is based primarily on the feature model [6], but the domain expert’s knowledge [3] is also important. It also depends on the identification of the subdomains, by analyzing for example the feature dependencies [7] or their potential for model-driven automation1. Each subdomain corresponds to a subset of the domain, and has its own set of features.

First, it is necessary to identify the features that will compose the DSL’s metamodel. We propose that this task should begin by analyzing the domain technology features [7]. These are the features that represent domain-specific ways to implement services or operations in a domain. In other words, they are the domain’s building blocks upon which the domain functions and services (capability features) are normally built. These domain technology features are good candidates to be part of a DSL.

![Diagram of a change case introducing the advanced search variant](image)

**Figure 2.** Example of a change case introducing the advanced search variant

Consider for example the web domain shown in Figure 1A. In this domain, the authoring subdomain can be identified. The features representing this subdomain (Authoring, DocumentType and Relationship) will be the starting point of the DSL’s metamodel. These features are then further analyzed to determine how they relate to each other and if additional concepts are needed. For example, the metamodel shown in Figure 3 includes, in addition to Authoring, DocumentType and Relationship, obtained directly from the feature model, the relationships between them, and new concepts, such as a Author, Field and FieldType. This metamodel describes the essence of the DSL and the variability in that particular subdomain.

C. Identification of subdomain integration drivers

Having identified those drivers that describe feature-based and DSL-based variability, each one normally contained in a well-defined subdomain, the domain architect must deal with the integration between them. For example, in the web domain (Figure 1A) the navigation subdomain may contain references to the authoring subdomain (e.g. a page that points to a specific document type). These subdomain interdependencies must be made explicit, so that the architecture can be prepared for the existence of multiple subdomains and possibly multiple DSLs [10], [11]. It might be necessary, for example, to define a single metamodel for multiple subdomains, and then develop different concrete

![Diagram of the authoring subdomain metamodel](image)

**Figure 3.** Definition of the authoring subdomain metamodel.

---

1Subdomains identification is out of the scope of this paper. Further reading in [9]
syntaxes for each one of them, so that they can integrate well but still have different views.

To accomplish this task, every subdomain (feature models and DSL models) is inspected, looking for elements (features or DSL metaelements) that depend on an element in a different subdomain. Multiple uses of the same name, synonyms and homonyms can point these out. Next, the dependencies must be documented in a clear way, annotating the feature models and the DSL models, or using a separate document that lists these dependencies.

IV. ARCHITECTURAL PATTERNS FOR MODEL-DRIVEN ENGINEERING

The architectural drivers described in Section III provide detailed information about how variation occurs in the different subdomains. With this information in hand, the architect can devise a domain-specific software architecture that gives the necessary support for MDE.

The architectural design is a creative process, requiring experience and knowledge about the domain. This task is normally carried out with the help of patterns to facilitate the design [2]. It is up to the architect to decide how to combine the available patterns according to the identified drivers, or even propose new patterns to design the architecture.

A list of well-known patterns is available in the literature [2], [18], but they normally do not deal with variability and MDE-related issues. Thus, we present some architectural patterns for dealing with the identified drivers, highlighting how they can be integrated with MDE artifacts, particularly the code generators. Some of the patterns are based on [19], which are useful to MDE projects. But since these are not specific for architectural design or domain engineering, we provide further discussions on these aspects. Those patterns that do not appear elsewhere (thin layer and features data layer - Section IV-B) were observed at least in our case studies, but we are aware that more experience is needed to increase their generality.

A. Feature-based variability patterns

In a previous work [20], we presented how some of the GoF patterns [21] can help making the architecture ready for the different types of feature-based variability. Here, we extend that work by highlighting how MDE integrates with each pattern and specifying which parts can be automatically generated. The scenario is the following: a feature model describes the common and variable parts. A code generator takes as input a selection of features/sub-features that will be part of the generated application, and produces the corresponding code. Depending on the type of feature, a different pattern can be applied.

Alternative features. These are features where only one alternative from a group can be present in an application. When a feature can be directly mapped into a single class, we suggest the use of the Prototype pattern. Each alternative is implemented as a different subclass of a common prototype. The code generator is responsible for generating code that instantiates only the selected alternative.

If a feature must be implemented by different classes, we suggest the use of the Abstract Factory or Factory Method patterns. These allow, through inheritance of the factory, to build more than one class (ConcreteProduct) for the same feature. To automate this pattern, the alternative features are implemented as concrete products, and the generator is responsible for producing the concrete factory or factory method that corresponds to the selected alternative.

When the alternatives are different behaviors that can be mapped to a single method, the Strategy or Template Method patterns may be used. These are similar to prototype, but only a method is overridden. In this case, the generator must produce the method calls for the selected alternative.

Or features. These are alternative features where more than one alternative from a group may be present. The Decorator and Chain of Responsibility patterns may be used when different features have functionalities that are complementary to each other. The Decorator is a structural pattern, more indicated for the cases where the interaction between several features is complex, but well defined. The Chain of Responsibility is a behavioral pattern, i.e., the structure of the interaction is not well defined, and thus it is more indicated for simpler feature interactions. In both cases, the behavior is encapsulated in specific methods, classes, or the Command pattern, and the generator is responsible for producing the method calls that correspond to the selected features.

Optional features. For optional features, the same set of patterns used for the Or features can be used, with the difference that in this case it is not necessary to guarantee that at least one feature is present in the application.

Figure 4 shows an example of application of the abstract factory design pattern being used for alternative features.

In this example, there are two alternative features, each one for a different database management system: Apache
Derby and MySQL\textsuperscript{2}. Depending on which feature is selected, a different set of classes is generated.

The following listing shows a piece of JET\textsuperscript{3} code that generates class `DAOAbstractFactory`. This code uses JET tags `<c:iterate>` and `<c:get>`, together with XPATH [22] queries, to iterate through all document types and generate one creator method for each document type.

```java
public abstract class DAOAbstractFactory {
    <c:iterate select="/documentTypes" var="d">
        public abstract <c:get select="/d@name" @>create<c:get select="/d@name" @>Instance();
    </c:iterate>
}
```

For document types “News”, “Project” and “Reference”, the class on the upper left corner of Figure 4 is generated.

To illustrate how this pattern facilitates the task of the code generator, consider the following piece of JET code:

```java
<c:iterate select="/documentTypes" var="d">
    <c:choose>
        <c:when test="/featuresModel/@derby='true'">
            <java:class name="/d@name" template="DerbyDAOClass.java" @ />
        </c:when>
        <c:when test="/featuresModel/@mysql='true'">
            <java:class name="/d@name" template="MySQLDAOClass.java" @ />
        </c:when>
    </c:choose>
</c:iterate>
```

Here, JET tags `<c:choose>` and `<c:when>` are used to select among mutually exclusive alternatives, in this case, alternative features “derby” and “mysql”, which are obtained from the features model through XPATH queries. Depending on the choice, JET tag `<java:class>` calls the appropriate template. For document types “News”, “Project” and “Reference”, three from the six concrete classes on the lower right side of Figure 4 are generated. A similar construction produces the concrete factory (`DerbyDAOFactory` or `MySQLDAOFactory`) for the selected feature.

There are also two patterns that can be used for feature-based variability, based on well-known good practices [23] for writing code generators:

- **Visitor pattern**

  Figure 5. Visitor pattern applied to feature-based code generation.

  The first pattern (Figure 5) is known as the visitor approach [23]. In this pattern, the input model is traversed and every element is visited. For each element, a corresponding template is called, according to the model element’s type. In the domain engineering scenario, it can be particularly useful for different types of mandatory and optional features [7]. A visitor traverses the features model and, for each selected feature, calls the corresponding template. Normally, each template produces a single class fitting into the architecture through design patterns such as the above.

  The visitor approach is a good choice when it is possible to encapsulate a feature’s functionality in a single class. If not, a second pattern can be used, known as the template approach [23]. It consists of a single entry point, responsible for querying the models and calling other templates. This approach may be used in different types of variability, because it is more flexible, being particularly useful to implement or features [7]: a main template analyzes the feature models and decides which templates to call based on the selected features.

- **B. DSL-based variability patterns**

  The patterns in this category are focused on how the DSL-based tools and code generators can be integrated with the other domain artifacts. One particularity about these patterns is that, after the MDE process is finalized and the code generation takes place, the patterns “disappear” from the architecture of the final product. This happens because they belong to a different meta-level. However, they do have impact on domain success, helping to shape an architecture that is better prepared for tasks like domain-specific modeling and code generation.

  A first pattern we propose is called thin data layer, which facilitates the integration between the generator and the DSL tool/modeler. Normally, code generators read information directly from a DSL tool, which can be created manually or with a language workbench such as GME or the Eclipse modeling project\textsuperscript{4}.

  ![Figure 6. Thin data layer pattern.](image)

  This pattern, shown in Figure 6, advocates the use of a separate data layer, built in a technology that is independent from the DSL tool, and that contains the information that is essential to the generator, and nothing else. In this way, the information needed by the generator is made explicit, which facilitates the evolution of both the generator and

\textsuperscript{2}http://db.apache.org/derby/ and http://www.mysql.com
\textsuperscript{3}Java Emitter Templates (http://www.eclipse.org/modeling/m2t)
\textsuperscript{4}www.isis.vanderbilt.edu/projects/gme/ and www.eclipse.org/modeling/
the DSL tool. It also allows the development of both sides in parallel (one team developing the generator and another team developing the tools/metamodels). Finally, it frees the generator from a particular modeling technology, and restricts the need for learning the particularities of a specific modeling tool to one team only. The team working with code generation may focus on its own tasks.

A second pattern is the features data layer. Normally, the feature model is a central point of information for generators, even those that are exclusively dedicated to DSL-based variability. This pattern proposes that a common data layer holds all the information related to the features. This data layer must be designed to be accessed by any generator, allowing it to query feature information while generating code. If there is a tool for feature modeling, a separate thin data layer can be used to hold feature data without depending on that particular tool.

The second code sample in Section IV-A illustrates the features data layer pattern. Instead of directly reading from the feature model, the XPath queries $featuresModel/@derby='true' and $featuresModel/@mysql='true' access a separate data layer, stored in a temporary slot represented by $featuresModel. This temporary slot contains only the necessary information from the features model that is actually used by the code generators. The following code shows the general idea of how this layer can be populated using JET jags <c:load> and <c:set>. In this example, the “derby” feature is selected:

```java
<c:load var="conf" url="features.xmi"/>
<c:set select="$featuresModel[@derby='true']" name="derby"/>
<c:set select="$featuresModel[@mysql='true']" name="mysql"/>
```

C. Subdomain integration patterns

These patterns aim to provide a good integration between generated and non-generated software, particularly in those areas involving a subdomain’s boundaries. The patterns in this section are divided in four categories, according to the dependency between generated and non-generated code:

Generated code depends on non-generated code. This is the simpler type of interaction, and consists of making the generator produce code that uses existing, non-generated code, such as a framework or API.

The Facade [21] pattern can be used to simplify the interaction between generated and non-generated code. Instead of generating code that uses multiple classes and methods, a single Facade class can group all necessary classes and methods. This not only makes the dependencies more explicit, but can also provide some protection against changes in the non-generated code. Depending on how deep the change is, it might not be necessary to change the generator, which is a more complex, error-prone task.

To shield the generator from modifications in the non-generated code, the adapter [21] pattern may be used, to collect, filter and/or prepare the information needed by the generated code. The bridge [21] pattern may be used with the same purpose, creating an abstract representation of the referenced code, that is free to be modified to introduce simpler modifications. Partial classes are another way to accomplish this, as shows the following code:

```java
// This file is generated
class DefaultDAOFactoryProvider {
    private static DefaultDAOFactory theInstance = null;
    public static DefaultDAOFactory getDefaultDAOFactoryInstance() {
        if (theInstance == null)
            theInstance = new DerbyDAOFactory();
        return theInstance;
    }
}
```

In this example, written in C#, the compiler is responsible for integrating the two files in a single class.

Non-generated code depends on generated code. This happens when some non-generated code expects that some behavior or structure is generated. In most cases, patterns such as abstract factory, template method or factory [21] can be used, so that the non-generated code does not need to know details about how the classes or methods it uses are actually implemented.

The following non-generated Java code illustrates how the dependency is reduced through the abstract factory pattern. In this example, this non-generated Java code depends on the “derby” feature being selected, but there is no need for much detail, as the abstract factory pattern hides most of it:

```java
public class DefaultDAOFactoryProvider {
    private static DAOAbstractFactory theInstance = null;

    public static DAOAbstractFactory
    getDefaultDAOFactoryInstance() {
        if (theInstance == null)
            theInstance = new DerbyDAOFactory();
        return theInstance;
    }
}
```

Generated code depends on generated code. This usually happens when there are two subdomains that depend on each other. One issue that arises when multiple subdomains are related is how to ensure this relationship between them. One possibility is to use the names of the elements as references, i.e. the name of a reference in one model must match the name of the referenced element in another
model. Although not ideal, this simplifies the process of implementing cross-model references [11].

Another option are model bridges, which consist in creating duplicates of elements from the referenced metamodel into the referencing metamodel. In the web authoring example, this corresponds to the creation, in the navigation metamodel, of an element called ReferenceToDocumentType, or something similar, that can be used to establish references between metamodels. A separate checker can make sure that these references are valid [10].

V. Evaluation

To evaluate our approach, we conducted three case studies, described in Table I. The first study involved the web content authoring domain, which is a technical domain that includes applications for publishing and viewing information, such as news websites, forums and blogs. It was performed in a purely academic scenario. The second and third studies were performed in conjunction with the industry. The second study involved the domain of distributed cloud-based applications, which is also a technical domain, dealing mainly with peer-to-peer communication and discovery functionalities. The third case study involved the domain of scientific events, which is a business domain.

All three case studies took a similar amount of time, although the third study had smaller duration and only part-time availability of the participants. The third case study was the largest in terms of lines of code of the reference implementation, but smaller in terms of the generation artifacts, which reflects the fact that the participants had less time to implement MDE automation.

The objective of these studies was to evaluate if the proposed drivers and patterns can help to produce an architecture that is better suited for MDE, so the idea was to make a comparison between existing implementations, developed without them, and the resulting MDE-based implementation, developed with them. For all studies, the same participants were responsible for developing the two versions (with and without our approach).

There are few model-oriented metrics available, and experience have shown that these are often not reliable [24]. Some authors have used classic object-oriented metrics to evaluate models [25], [26], arguing that they are more stable and can detect the same problems that will appear in the code, but earlier in the process. We decided to follow this second line of thought and use more reliable metrics:

\[ M_1: \text{RP - Reuse Percent} \]

Reused LOC / Total LOC. This is a classic reuse metric [27], which counts as reused any line of code that is reused without modification.

\[ M_2: \text{RR - Reuse Ratio} \]

The same as \( M_1 \), but here artifacts that suffer only a small modification (less than 25% of the code is modified) are considered reused as well [28].

\[ M_3: \text{Undesired Reuse Percent} \]

The reuse of large pieces of unused code may distort the reuse percent metric. For this reason, we defined metric \( M_3 \), which consists of calculating the lines of code that are reused (copied to the new context) but are not being used. IDE functions that find methods that are not being called can be used to collect these data.

\[ M_4: \text{Module instability} \]

This metric aims at evaluating how difficult it is to modify a piece of software [29]. The metric ranges from zero to one, where zero means a stable (easy to modify) module, and one means an unstable (difficult to modify) module.

\[ M_5: \text{Cyclomatic complexity} \]

This is a classic complexity metric [30], based on the number of paths inside a program. Values between 1 and 10 indicate simplicity, and values above 50 represent complex modules. In the context of this research, complexity is an important indicator of how the architecture is facilitating tasks like implementation, maintenance and software reuse.

\[ M_6: \text{Maintainability index} \]

This metric is an attempt to determine how easily a module can be maintained, by combining a series of empirical measurements, like the Halstead volume and cyclomatic complexity [31]. Values below 65 indicate low maintainability, while values above 85 indicate good maintainability.

These metrics were collected with the help from tools.

---

**Table I**

**Summary of the case studies**

<table>
<thead>
<tr>
<th>Study number</th>
<th>Domain</th>
<th>Location</th>
<th>Participants</th>
<th>Duration</th>
<th>Num. DSLs</th>
<th>Num. generation artifacts</th>
<th>Size (LOC) generation artifacts</th>
<th>Size (LOC) ref. implementation</th>
<th>Num. of domain features</th>
<th>Implementation technologies</th>
<th>MDE technologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Web Authoring</td>
<td>ICMC/UFRJ - São Carlos/SP - Brazil</td>
<td>1 (author)</td>
<td>3 months (full-time)</td>
<td>4</td>
<td>38</td>
<td>1610</td>
<td>12127</td>
<td>15</td>
<td>Apache Tomcat, MySQL, Java, JSP, JSTL, Servlets, XML, SQL, Eclipse</td>
<td>Eclipse Modeling, JET (Java Emitter Templates)</td>
</tr>
<tr>
<td>2</td>
<td>Cloud-based applications</td>
<td>Microsoft Research - Redmond/WA - USA</td>
<td>2 (incl. author)</td>
<td>3 months (full-time)</td>
<td>4</td>
<td>39</td>
<td>2847</td>
<td>71873</td>
<td>29</td>
<td>Visual Studio 2008, SQL Server, C#, .NET, .NET Remoting, Volta, PNRP, DBML, LINQ</td>
<td>Visual Studio 2008, Microsoft Text Templates, Generic Modeling Environment</td>
</tr>
<tr>
<td>3</td>
<td>Scientific events</td>
<td>Aptron Systems - São Carlos/SP - Brazil</td>
<td>2 (incl. author)</td>
<td>2 months (part-time)</td>
<td>4</td>
<td>39</td>
<td>1375</td>
<td>1</td>
<td>3</td>
<td>Adobe Dreamweaver, PHP, MySQL.</td>
<td>Eclipse Modeling, JET (Java Emitter Templates)</td>
</tr>
</tbody>
</table>

---

**Reference**


Eclipse Metrics, JHawk and Net2Java\textsuperscript{5}. Manual extraction was also necessary for those artifacts, such as code generation templates, not supported by the automatic tools.

Using these indirect metrics, we tried to obtain a picture of how the approach can influence different factors in software development. Knowing whether the observed results are directly caused by the approach and the produced architecture or by other factors remains a threat to the study. To soften this uncertainty, an interview was also conducted with the participants of the third case study. The interview consisted of open questions about the participants’ impressions on the benefits and difficulties of using the approach.

A. Results and analysis

Table II shows the results of the evaluation. Metrics $M_4$, $M_5$ and $M_6$ could not be extracted for Study 3 because we could not find an automated tool to extract them from non-OO PHP code, which was used in this domain.

In the first two studies, we observed an increase in the reuse level, in terms of LOC, either considering reuse without modification ($M_3$) and with minor modifications ($M_2$). The degree of undesired reuse ($M_4$) was also slightly reduced in this first two studies. For the third study, however, the reuse level in terms of LOC was reduced when the approach was used. In fact, the reuse level was already high without the approach. On the other hand, the undesired reuse is considerably smaller when the approach was used.

For study 1, which is a relatively simple technical domain, we observed that the resulting architecture had an improved overall stability of the modules. However, they were, in average, more complex and more difficult to maintain. For study 2, which involved a more complex technical domain, the complexity of the architecture was reduced, without affecting too much the maintainability.

For study 3, an interview was conducted. During this interview, the participants reported that the approach helped to solve some problems that could not be solved by hand, in particular the many customizations that must be made every time a product is sold. They tried to parameterize some of these in the code, but this often caused loss of generality, and so this solution was abandoned. With MDE and the use of a higher level architectural design approach, the reusable code can remain generic, while the final code is made specific through code generation. The automation also helped to reduce errors in customization.

Another aspect raised by the participants was the fact that reuse was high without the approach, but at the cost of having large pieces of unused code copied into the product. This was causing some confusion, specially in maintenance. The approach helped to reduce that.

The participants also reported that the modeling and code generation technologies are difficult to learn, requiring a shift of thought to correctly use them. The participants had basic modeling skills and worked as programmers for two years, never having worked with MDE before. However, they managed to implement the required functionality after a 16-hours training about MDE.

Table III summarizes our observations, which indicate that the proposed drivers and patterns may bring benefits in many situations, but a trade-off analysis must be carried out to evaluate its drawbacks of increased complexity, decreased maintainability and a high learning curve. Projects similar to study 2, i.e. involving more complex technical domains, seem to be the best candidates for using our approach.

<table>
<thead>
<tr>
<th>Std</th>
<th>Main observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Architecture resulted in increased reuse, but at the cost of more complex and less maintainable modules</td>
</tr>
<tr>
<td>2</td>
<td>Architecture resulted in increased reuse, less complex and more maintainable modules</td>
</tr>
<tr>
<td>3</td>
<td>MDE-based approach is difficult to learn but facilitates product customization and makes reuse more focused</td>
</tr>
</tbody>
</table>

Table III

Main observations in the case studies

B. Threats to validity

There are some threats to the validity of this study. The participation of one of the authors of this paper in two of the studies may have influenced the results. Also, the fact that we were not able to collect the metrics for the third study made our conclusions less generalized. The use of indirect metrics is also a threat to this kind of experiment.

We consider that the fact that the same participants developed the domain infrastructure with and without the approach did not influence the results, because the approach actually encourages the reuse of existing software to develop the DSLs and code generators.

\textsuperscript{5}metrics.sourceforge.net, www.virtualmachinery.com/jhawkprod.htm and net2java.dev.java.net
Finally, the realization of experiments in an industrial scenario makes it difficult to control all the influencing factors and variables, and thus the results can be strongly biased. This is also aggravated by the fact that the studies were realized in more than one country, in different environments and with different cultures. However, this is also a good thing, because it increased the diversity and overall confidence in the results.

VI. RELATED WORK

Czarnecki et al. [32] present a template-based approach for generating models from feature models. Through mappings between features and general models, such as data or behavioral models, richer information can be associated with features. When applied to architectural models, more detailed variability can be specified, which is similar to our approach. However, differently from Czarnecki et al.’s work, we also address the issues that result from the inclusion of MDE-specific artifacts, such as domain-specific modelers and code generators.

Perovich et al. [33] propose an approach that uses features to modularize architectural decisions. These are made concrete through model transformations that automatically produce architectural fragments that correspond to the selected features. As in our approach, the architecture is an artifact that sits on a different meta-level, including transformations and code generators that will disappear from the final product’s architecture. The main difference from our work is that we include an explicit concern about more detailed variability, applying DSLs in parts of the domain to achieve more expressive power, while Perovich et al. use only feature models as a starting point for product configuration.

In [34], the authors present a model-driven approach for requirements engineering in a product line context. As in our research, they defend the idea that features should be complemented with additional models to better represent variability. Composition rules between use cases are used similarly to the concept of change cases, but without the benefit of producing single scenarios that can be used in evaluation. Also, their work do not present details about how the architecture is obtained, with patterns, for example.

The work described in [35] presents an approach to derive the architecture from analysis models using semi-automatic transformations. However, it is assumed that the analysis models are rich enough to be automatically transformed into an architecture. Also, only a single architectural style (C2 [35]) is supported. And finally, the approach is focused on single product development, not covering reuse-related issues such as variability support and product derivation.

VII. CONCLUDING REMARKS AND FUTURE WORK

MDE can leverage domain engineering through techniques that support more complex variability and automatic implementation. In general, the literature is more focused on the automatic product derivation process. There is no doubt that this is the final goal to be achieved, and we agree with that. However, little attention is given to the process of producing a domain architecture that is prepared for this kind of automation, and we believe that the issues we discussed in this paper are important.

Our research aims at providing a more systematic way to perform domain design considering the requirements of a domain engineering effort that will benefit from MDE techniques. The main contributions of this research are the identification of specific architectural drivers and patterns that can be used to produce a domain-specific software architecture, and the description of how to deal with them using existing and new patterns.

We also provided the results of an evaluation. Despite the threats to its validity, we consider that valuable observations could be made. The qualitative and quantitative data pointed out that the approach is feasible, and may lead to benefits in terms of software development. The resulting architecture helped to increase the reuse level, as observed in two of the three case studies, and also make it more focused, as observed in one of the studies. The proposed patterns also seems to help dealing with the design of architectures for more complex domains, like the one involving distributed cloud-based applications. But we observed some drawbacks that need to be considered in a trade-off analysis prior to using our approach.

As future work, we are planning further experiments with these drivers and patterns, to refine and extend them. The results from the first case studies are promising, and we expect that the completion of more experiments can deliver more solid results to the interested community.

ACKNOWLEDGEMENTS

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES - www.ines.org.br), funded by CNpq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.

REFERENCES


