Pattern Language Verification in Model Driven Design

Bahman Zamani[a,*], Greg Butler[b]

[a] *Department of Computer Engineering, University of Isfahan, Isfahan 81746-73441, Iran*
[b] *Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec H3G 1M8, Canada*

Abstract

This paper addresses the problem of verifying the application of a Pattern Language in a design that is built based upon the patterns of the language in a Model-Driven approach. We propose a process named Pattern Language Verifier (PLV) which consists of four phases, working on a design model, to 1) verify the structure of every single pattern, 2) verify the relationships between the detected patterns, 3) verify the semantic aspects of the patterns, and 4) report the problems and help the designer fix them. Building a PLV for a given Pattern Language, requires the Structural, Syntactic, and Semantic rules of the language to be precisely defined using the presented formalism.

For the case study, a group of enterprise architectural patterns is selected as the Pattern Language. The Structural, Syntactic, and Semantic rules of the language are defined using the proposed formalism. A PLV is designed and implemented as an integration into an open source modeling tool. The tool is then utilized in designing a sample web application. The usefulness of the tool is represented by walkthrough scenarios that show finding the mistakes in the model and helping the designer repair the detected problems.

1. Introduction

The emergence of model-driven approaches [Sel03, Sch06, Obj09] for software development, e.g., Model-Driven Engineering (MDE), has shifted the focus from code-centric to model-centric. Since models are the main artifacts that drive software development [Béz06], the quality of models has direct impact on the quality of software. The verification and validation of evolving models is an issue in model-driven approaches, hence, providing processes and tool support for software development is a real need. Amongst them are the tools to verify the evolving artifacts such as design.

Designers, in the hope of producing quality models, apply best practices that are introduced by the experts as *Patterns*. "A pattern is a solution to a

*Corresponding author
Email addresses: zamani@eng.ui.ac.ir (Bahman Zamani), gregb@encs.concordia.ca (Greg Butler)

problem in a context" [GHJV95, p. 3]. A collection of patterns makes a Pattern Language (PL) if a) there is a starting pattern, b) there is a guidance on how to use one pattern after another, and c) the set of all patterns in the collection are sufficient to provide the design for a whole system [A+77]. *Designing with patterns* [BHS07, p. 248] is not an easy task. The problem is that patterns are not isolated islands and the application of the patterns is not arbitrary. Designing a system based upon the patterns of a PL, requires knowledge about how to apply an individual pattern correctly, how to put several patterns together correctly (weave patterns), and how to ensure that a pattern combination is semantically correct. Not all the PLs have precise rules that codifies such knowledge. For some PLs, e.g., Patterns of Enterprise Application Architecture (*PofEAA*) [Fow02], pattern relationships are embedded into the lengthy texts of pattern descriptions. Hence, designers are vulnerable in making mistakes in pattern selection, pattern application, and pattern weaving. Providing support for automatic verification of the models that have benefited from a PL will expedite the design process and results in better productivity.

This paper addresses the problem of verifying a *design model* which is built based upon the patterns of a PL. We believe that this problem is similar to using a compiler for checking a source program which is written in a programming language. As a solution, we propose a process called Pattern Language Verifier (PLV) that accepts a Unified Modeling Language (UML) design model as input and reports the structural, syntactic, and semantic errors in the model, considering the rules of the underlying PL. Furthermore, PLV helps the designer fix the problems either automatically or through guidelines. In *designing with patterns* we suppose that the designer knows which patterns he/she wants to apply. This way we eliminate consideration of the task of pattern selection in PLV. Our case study is two-fold: a) to validate the PLV process, i.e., to show how we can reach a PLV tool, given a PL, and b) to evaluate the obtained PLV, i.e., to show how the tool helps the designer in finding and fixing the design problems related to applying the patterns of the PL. In the first part of the case study, we select a subset of *PofEAA* [Fow02] as the PL, we define the required formal rules, and then we implement a PLV for *PofEAA*, called ArgoPLV, as an extension to the ArgoUML [Tig09] modeling tool. In the second part of the case study, we use the ArgoPLV tool to verify a design model of a web-based application: *online student registration system*.

The idea of PLV process is evolved from works in [ZB07, ZKB08, ZBK09], and the formalism for representing the rules of a PL is first introduced in [ZB09]. However, this paper includes novel contributions as per following: improvements on the PLV process and its modules, the idea of extracting advices from prose pattern descriptions and then converting advices into rules, dividing syntactic rules into two separate aspects, as well as more real case study. The previous work are cited where relevant.

This paper is organized as follows. Section 2 introduces the background knowledge and the related work. Section 3 describes the PLV process as a proposed solution to the problem. Section 4 shows the architecture proposed for the PLV. Section 5, the case study, first shows how a PLV can be defined for

a given PL, then explains how the obtained PLV can be applied in a real world situation to check a design model. Section 6 is dedicated to the conclusion and future work.

2. Background and Related Work

To the best of our knowledge, PLV is the first work which addresses the problem of verifying a design model from the PL view. Most of the related work falls into the category of single pattern detection, particularly detecting Gang of Four (GOF) design patterns [GHJV95]. The approach in [TCSH06] is based on similarity scoring between graph vertices, while [BP02] and [Wuy98] use Prolog rules to specify the structure of a pattern.

There are two works which focus on the PL aspects and are close to PLV: Pattern Enforcing Compiler (PEC) [Lov06] and Zdun's work [Zdu07]. The former is an extension to a Java compiler which verifies the application of GOF design patterns in the code. PEC only investigates individual patterns. It does not consider PL issues. The latter uses annotated PL grammars and design space analysis in systematic pattern selection. This work provides a pattern selection mechanism; It is not a verifying approach, and it does not address the models directly.

Due to the key role of models in MDE, quality assessment of models is an important issue. Several works have focused on the quality of models. Different people view the quality of a model from different aspects [Sel03, Sel06, Unh05]. From the patterns point of view, Buschmann et al. [BHS07, p. 131-132] see "high pattern density" as a characteristic of a good design.

Zdun and Avgeriou [ZA05] have identified a number of "architectural primitives" as recurring participants of architectural patterns and have defined a UML Profile containing Object Constraint Language (OCL) constraints that provide precise semantics of the primitives. In [KAZ11], by discovering and defining a set of pattern participants relationships, the authors try to address the problem of pattern integration that arises in integrating architectural patterns.

Egyed [Egy07] has considered model inconsistencies, e.g., the changes in the models that have undesired side effects or cause new bugs in the model. Liu et al. [LEM02] have developed a "Rule-Based Inconsistency Detection Engine" (RIDE) which helps detect and resolve the inconsistencies in the UML models. RIDE can also be used to detect misuses of design patterns.

Some researches exist on pattern relationships, since considering patterns independently results in low quality designs, i.e., designs which are more complex and hard to maintain [BHS07, p. 117]. Even in the pattern resources that have not focused on the PL aspects, e.g., in GOF, we can see indications of considering the dependency between patterns. Zimmer [Zim95] has divided the relationships between GOF design patterns into three categories: "uses," "is similar to," and "can be combined with." Zimmer has concluded that "Applying design patterns requires a fair knowledge of both single design patterns and their relationships" [Zim95]. James Noble [Nob98] has proposed a classification

scheme for describing pattern relationships, which consists of three *primary* and nine *secondary* relationships.

Using the vocabulary metaphor for the patterns of a PL leads to the grammar metaphor for the rules that dictate the correct sentences of the language [BHS07, p. 281]. Several formal notations can be used for representing such grammar, the Backus Naur Form (BNF) notation [Knu64], graphical notations such as "Feature Modeling" [KKL+98] or "Syntax Graph," to name a few. In [BHS07] four types of relationships that could exist between the patterns are presented: Competition (a.k.a. pattern alternatives), Completion (a.k.a. patterns in cooperation), Combination, and Compound.

3. The PLV Process

The challenges that a designer, in *designing with patterns*, faces include pattern selection, pattern application, pattern weaving, and pattern semantics. We propose a process named Pattern Language Verifier (PLV) to assist designers in verifying the application of a PL in the design. Since PLV is a verifying process, we exclude the "Pattern Selection" challenge from its duties. That means, we suppose that, the designers know which pattern they want to use, and they show their intention by using the name of the pattern. The PLV process, inspired by the compiler idea, requires the structural, syntactic, and semantic rules of the PL to be formally specified and drive the PLV process. In the following, we will elaborate on the nature of these rules and we give a guidance on how to specify the rules using the formalism proposed in [ZB09].

The *structural rules* are used by PLV to verify the structure of each individual pattern used in the model. By structural rules of a PL, we mean sets of rules where each set shows the essence of one pattern in the language. The structural rules must specify the constituting elements of the pattern as well as the way these elements are connected [GHJV95, p. 3]. As a formalism for defining these rules, we use an enumerated list of textual rules written in plain English, enriched with an optional UML class diagram. The textual rules are the main parts and must be clear and simple enough such that an intermediate Object-Oriented (OO) programmer can understand them [ZB09].

To facilitate the detection of elements in a model, and hence, to ease the work of PLV, we utilize a naming convention paradigm. We suppose that, the designers use the same names that are used by the pattern author in the *pattern form* for naming the patterns and their elements. From now on, we refer to the name of the pattern as "Sign" [ZBK09]. For instance, if the designer intends to use the Table Data Gateway pattern [Fow02, p. 144] in accessing the Person information, he/she may choose "PersonTableDataGateway" as the name of the class which corresponds to this pattern.

The *syntactic rules* deal with the relationship between patterns. We believe that syntactic rules must address two aspects: I) "Pattern-Layer Relationships," and II) "Pattern-Pattern Relationships." While the latter aspect is essential in every PL, the former aspect is optional and may not be identifiable in some PLs.

Patterns are normally organized into groups or layers. *Pattern-Layer Relationships* are the rules that enforce the correct organization for the patterns that are applied in a design model. We use a formalism [ZB09] inspired by context-free grammars, BNF, and set notations, for representing the organization of patterns into groups or layers (See Table 1). In terms of UML, since a "package" is often used to group elements [Lar05, p. 201], we correspond one layer/group to one package in the model. Hence, the syntactic rule that checks the membership of a pattern in a layer/group, should simply check that the main class (Sign) of the pattern is placed in the corresponding package.

Table 1: Formalism for Pattern-Layer Relationships [ZB09]

Notation		**Meaning**
$l\ m$	Layer	Lowercase letters or first-small words show the layers
$P\ Q$	Pattern	Capital letters or capitalized words show the patterns
$\supset$	Layer Inclusion	$l \supset m$ means layer l contains layer m
$\ni$	Pattern Membership	$l \ni P$ means layer l contains pattern P
,	Group Inclusion Group Membership	$l \supset m\ ,\ n$ means $l \supset m$ and $l \supset n$ $l \ni P\ ,\ Q$ means $l \ni P$ and $l \ni Q$
.	Layer dependency	$l \supset m\ .\ n$ means $l \supset m\ ,\ n$ and layer m is dependent on layer n, but layer n is not dependent on layer m
$*$	Optional Layer	l^* means layer l is optional
$?(c)$	Conditional Layer	$l^{?(c)}$ means existence of layer l is subject to condition c
$\{c\}$	comment	A *comment* can be attached to the above notations

Patterns can be related to each other in different ways: uses, conflicts, refines, to name a few. We use a grammar-like formalism [ZB09] for representing the *Pattern-Pattern Relationships* rules of a PL (See Table 2). Despite its formality, a grammar-like approach presumes a predefined set of relations between patterns, and this is more limited than representing those relations in prose description.

Note that every pattern combination has to start with one of the root patterns. A root pattern is an obligatory pattern and no other pattern is dependent upon it. The *uses* relationship is a basic relationship which can be found in most PLs. Three variants of *uses* are defined (*uses*, *conditional uses*, and *alternative uses*) to make it more usable. The *conflicts* relationship describes the situation where there is more than one solution to a specific problem, and those solutions are mutual exclusive. Two patterns can be conflicting either in the whole model or in a specific layer of the model. The *refines* relationship shows the case when one pattern is a more specialized version of another pattern.

The *semantic rules* are used by PLV to verify whether or not a pattern combination is semantically correct. To the best of our knowledge, there has been no discussion on the semantics of a pattern combination in the PL community. In [ZB09], we considered two categories of semantic problems, and we defined two simple notations for them (See Table 3). The first category shows the conflicts between the applied patterns and the context information. Examples of the context information are programming language, complexity of the system, and expertise of the designer [BHS07, p. 154]. The semantic rules should clearly say which pattern is in conflict with which context information. The second category shows the inconsistencies between the features of applied

Table 2: Formalism for Pattern-Pattern Relationships [ZB09]

Notation		Meaning
$\overline{P}$	root pattern	$\overline{P}$ means pattern P is a root pattern of the language
$\rightarrow$	uses	$P \rightarrow Q$ means pattern P uses pattern Q
$\overset{cond}{\rightarrow}$	conditional uses	$P \overset{cond}{\rightarrow} Q$ means pattern P uses pattern Q subject to $cond$
$\mid$	alternative uses	$P \rightarrow Q \mid R$ means pattern P may use pattern Q or pattern R
$\leftrightarrow$	conflicts	$P \leftrightarrow Q$ means patterns P and Q can not coexist in the model
$\overset{l}{\leftrightarrow}$	conflicts in layer	$P \overset{l}{\leftrightarrow} Q$ means patterns P and Q can not coexist in the layer l
$\uparrow$	refines	$P \uparrow Q$ means pattern P is a specialized version of pattern Q
$\{c\}$	comment	A *comment* can be attached to the above notations

patterns.

Table 3: Formalism for the Semantic Rules of a PL [ZB09]

Notation		Meaning
$\approx$	consistent	$P \approx \{c\}$ means pattern P is consistent with the condition specified by c
$\not\approx$	inconsistent	$P \not\approx \{c\}$ means pattern P is inconsistent with the condition specified by c

4. The PLV Architecture

The PLV process verifies a given UML design model from the structural, syntactic, and semantic viewpoints of the underlying PL. To fulfill this task, the architecture proposed for the PLV in [ZKB08] has three phases (modules) respectively. In addition, it has an advisory module to help the designer fix the problems. Hence, the PLV process includes four cooperating modules: Pattern Structural Verifier (PSV), Pattern language synTactic Verifier (PTV), Pattern language seMantic Verifier (PMV), and Pattern Language Advisor (PLA) as described in the following.

First module, PSV, accepts the Design Model as input and, by verifying the structural rules of the PL, looks for single patterns that are correctly applied in the model. The designer shows his/her intention of applying a particular pattern by using the "Sign" (class) of that pattern in the model. By detecting the "Sign", PSV initiates the verification process to check the structure of the pattern. If errors are found in the structure of the pattern, e.g., missing an element in a pattern, the PLA is invoked to report the errors and help the designer fix the problems. PSV applies the "structural match" strategy. That means, it matches the structure of the pattern given in the Design Model, with the structure of the pattern that is defined by the structural rules. For doing this task, PSV applies ideas introduced by the Sign/Criteria/Repair (SCR) process, except the repair part [ZB07, ZBK09].

Second module, PTV, verifies the model based on the syntactic rules of the PL. This verification includes both checking the layering of patterns (i.e., checking the Pattern-Layer Relationships Rules) and checking the relationships between the detected patterns (i.e., checking the Pattern-Pattern Relationships Rules). Therefore, there are two types of errors that can be caught by PTV.

First, placement of a pattern in a wrong layer or group. Second, missing or incorrect relationship between two patterns. In case of error, PTV invokes PLA to report the error and guide the designer in fixing the problem. Some of the syntactic errors can be fixed automatically by the PLA, but most cases need designer's decision and manual modifications on the model.

Third module, PMV, is responsible for verifying that the model adheres to the best practices of the PL. Specifically, by applying the semantic rules of the PL, PMV verifies that the model is consistent with the context information of the system, e.g., the implementation language. An example of an error that PMV can recognize is "discrepancy between the context environment and the used patterns." If any inconsistency is found in the applied pattern combination, the PLA is invoked to report the errors and help the designer fix the problems. Many of the problems are easily fixed by setting the appropriate value for the context information, e.g., selecting another tool for implementation. These repairs can be done automatically by the PLA. Other problems that are solved only be changing the applied pattern, should be solved manually by the designer.

Fourth module, PLA, is responsible for reporting the errors to the designer, displaying guidelines on how to fix the problems, fixing the detected problems in a systematic manner, and recording the model modifications into a *Design Rationale*. Reporting the errors and displaying the guidelines are important steps that foster a novice designer's knowledge in learning more about the patterns and PL. Upon detection of an error in the model, PLA is invoked, and having access to all the structural, syntactic, and semantic rules, guides the designer in stepwise fixing of the problems. PLA also gives the designer the opportunity for systematic repair, if possible. Therefore, PLA is the only module which is able to apply modifications to the model, e.g., pattern instantiation, or adding missing elements to a pattern. For problems that are hard to fix automatically or need expertise or designer's decision, the guidelines for fixing the problem should be given to the designer, and it is the designer's responsibility to modify the model accordingly. However, providing the designer with guidance and supporting comments can expedite the error recovery process.

5. ArgoPLV: A PLV for *PofEAA*

This section shows how to make a PLV for a Pattern Language. As the PL, we have selected Martin Fowler's book titled "Patterns of Enterprise Application Architecture" known as *PofEAA* [Fow02]. We discuss how to obtain formalized rules from the prose description of the patterns, then we show how to code PLV modules into a modeling tool, and at the end we show the usefulness of the resulted tool.

5.1. PofEAA Selected Patterns

In the *PofEAA* book, 51 patterns are introduced as solutions to the recurring problems in *designing* the architecture of a web-based enterprise application.

The pattern form used in *PofEAA* has eight items, including the name, the intent, and the sketch [Fow02, p. 11]. The patterns in *PofEAA* are decomposed into three main layers [Fow02, p. 19]. Also there are supporting patterns for the issues such as object to relational conversion and concurrency management.

We argue that *PofEAA* is a PL for the *design* of web-based applications. First, *PofEAA* patterns are closely related to each other and can be used to design the architecture of the application. Second, there are several recommendations in *PofEAA* about how to decide amongst various alternative patterns for a design problem. Third, the set of patterns in *PofEAA* is rich enough to describe the design of an application as a whole.

For the sake of simplicity and concreteness, we selected a subset of *PofEAA* that contains 23 patterns. Patterns that are filtered out (28 patterns) are mainly "Object-Relational" patterns. From the PLV perspective, considering those patterns does not add any knowledge to codifying process that we are going to present in this section. Figure 1 shows the placement of 23 selected patterns in a layered architecture.

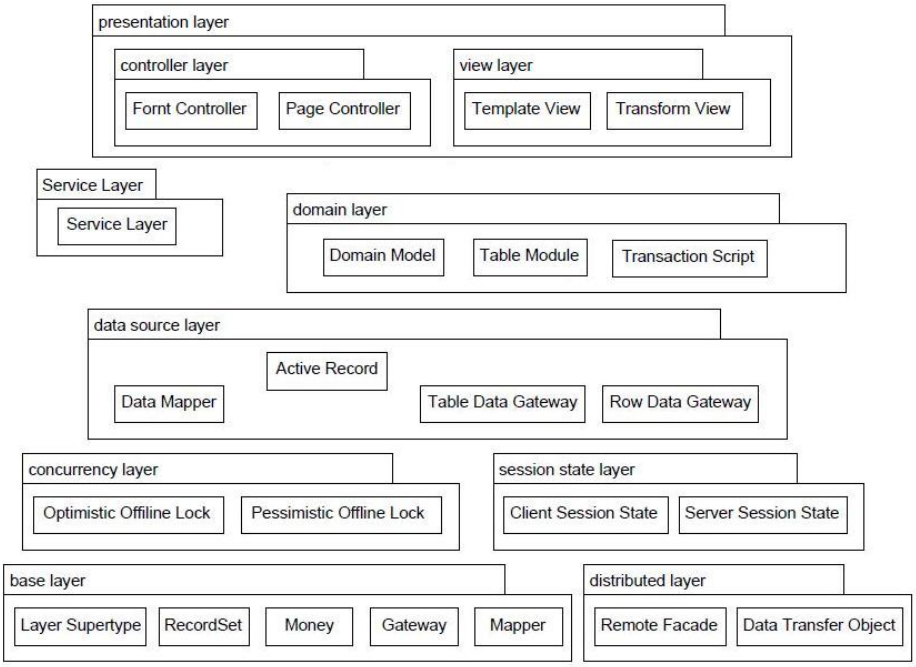

Figure 1: Selected Patterns from *PofEAA* in a Layered Architecture

5.2. PofEAA Rules

Due to the fact that the *PofEAA* book, like many other pattern books, describes patterns by means of prose description, defining the formalized *PofEAA* rules is not straightforward. Hence, we decided to break this task into two steps.

Table 4: Advices Extracted from the *PofEAA* Book [Fow02]

A#	**Type**	**Description (*PofEAA* book page#)**
A14	Syntactic	"A rich Domain Model is better for more complex logic, but is harder to map to the database." (p. 117)
A25	Structural	"A Front Controller handles all calls for a Web site, and is usually structured in two parts: a Web handler and a command hierarchy. The Web handler is the object that actually receives post or get requests from the Web server." (p. 344) "The Web handler is almost always implemented as a class rather than as a server page [...] The commands are also classes rather than server pages." (p. 345)
A46	Semantic	"The parameter list of the insert method must be a subset of the parameter list of the update method." (p. 144)
A47	Syntactic/ Semantic	"You probably don't need a Service Layer if your application's business logic will only have one kind of client-say, a user interface-and its use case responses don't involve multiple transactional resources" (p. 137)
A49	Syntactic	"For this book I'm centering my discussion around an architecture of three primary layers: presentation, domain, and data source." (p. 19)
A51	Syntactic	"Often you'll find that there isn't quite a one-to-one relationship between Page Controllers and views." (p. 61)

First, we extract some *advices* from the text, then, we finally turn advices into the rules that are the basis for the PLV. We have extracted 74 advices from the book and bracketed them into three classes: *Structural, Syntactic*, and *Semantic*. Table 4 is an excerpt from these advices. Some of the advices are not accurate enough, especially the syntactic and semantic ones. Hence, formalizing an advice into a rule for the PLV is not an easy task, and needs expertise.

Structural rules are those that describe the essence and the structure of an individual pattern. One important step in specifying the structure of a pattern is to select a "Sign" for each pattern. We select the names of the patterns that are indicated in Figure 1, as the "Sign" for the selected patterns of *PofEAA*.

Amongst the extracted advices, 23 of them are structural, which are converted into structural rules. For instance, the advice A25 describes the structure of the Front Controller pattern [Fow02, p. 344]. We define a set of eight criteria for the structural rules of this pattern, as shown in Figure 2. The same procedure is performed for all the 23 selected patterns, and the structural rules of all patterns are extracted. The result is called "PofEAA Rule Set - Structural Rules," which is not shown here due to lack of space. The resulted rule set is accessible via [Zam09].

For *Pattern-Layer Relationships*, we have two sources: 1) Figure 1 which divides 23 selected patterns into layers, and 2) the information that is recorded as syntactic advices in Table 4. For instance, advice A49 clarifies that there are three mandatory layers in the model, and advice A47 reveals that the "Service Layer" is not a mandatory layer and its existence depends upon the designer's choice. The following is the formal representation of the advices A49 and A47, supposing that all the layers lie in a root model named "pofeaa model."

$$pofeaa\ model \supset presentation\ .\ service^{?(\text{Needs Service Layer})}\ .\ domain\ .\ datasource$$

In total, there are 16 advices about the organization of patterns. Investigat-

1. There is a `Front Controller` (=`Handler`) class in the model.
2. There are at least two operations (`doGet` and `doPost`) in the Handler class.
3. The `Handler` class has a client dependency to a `Command` class.
4. The `Command` class is abstract.
5. The `Command` class has at least one `process` operation.
6. The `Command` class has at least one `Concrete Command` child class.
7. A `Concrete Command` class is concrete.
8. A `Concrete Command` class has at least one `process` operation.

Figure 2: A Structural Rule Showing the Structure of the Front Controller Pattern

ing those advices along with Figure 1, and converting them into formal rules, we obtained the rules shown in "Part A" of Figure 3.

For *Pattern-Pattern Relationships*, i.e., the relationship between patterns, there are 27 syntactic advices. Using the proposed formalism, we converted them into syntactic rules, and obtained the rules shown in "Part B" of Figure 3. As an example, consider advice A51 which says there are two alternative view patterns that can be used by a Front Controller pattern. Using the *alternative uses* formalism, we write the following rule.

$$Page\ Controller \rightarrow Template\ View \mid Transform\ View$$

Semantic rules of PLV aim to catch two types of errors: 1) conflicts between the applied patterns and the context information, and 2) the inconsistencies between the features of applied patterns. Amongst the extracted advices, 17 of them are semantic advices. By interpreting those advices into the formal rules, we obtained the rules shown in "Part C" of Figure 3. As an example, consider advice A14 about the effect of the domain complexity on the pattern used for the Domain Layer. This advice is formally represented as the semantic rule:

$$Domain\ Model \approx \{\text{Domain structure is complex }\}$$

pofeaa model $\supset$ *main layer* . *auxiliary layer** (Part A)
main layer $\supset$ *presentation* . *service*$^{?(\mathrm{C41})}$. *domain* . *datasource*
presentation $\supset$ *controller* . *view*
auxiliary layer $\supset$ *base** , *distributed*$^{?(\mathrm{C42})}$, *concurrency*$^{?(\mathrm{C43})}$, *sessionstate*$^{?(\mathrm{C44})}$
controller $\ni$ *Page Controller* , *Front Controller*
view $\ni$ *Template View* , *Transform View*
service $\ni$ *Service Layer*
domain $\ni$ *Domain Model*, *Table Module*, *Transaction Script*
datasource $\ni$ *Data Mapper*, *Active Record*, *Table Data Gateway*, *Row Data Gateway*
base $\ni$ *Record Set*, *Layer Supertype*, *Money*, *Mapper*, *Gateway*
distributed $\ni$ *Remote Facade*, *Data Transfer Object*
concurrency $\ni$ *Optimistic Offline Lock*, *Pessimistic Offline Lock*
sessionstate $\ni$ *Client Session State*, *Server Session State*

$\overline{\textit{Page Controller}} \rightarrow \textit{Template View} \mid \textit{Transform View}$ (Part B)
$\overline{\textit{Front Controller}} \rightarrow \textit{Template View} \mid \textit{Transform View}$
$\overline{\textit{Page Controller}} \overset{\textit{Tool=.NET}}{\rightarrow} \textit{Template View} \bullet \overline{\textit{Front Controller}} \overset{\textit{Tool=Java}}{\rightarrow} \textit{Template View}$
$\textit{Template View} \overset{\mathrm{C41}}{\rightarrow} \textit{Service Layer} \bullet \textit{Transform View} \overset{\mathrm{C41}}{\rightarrow} \textit{Service Layer}$
$\textit{Service Layer} \rightarrow \textit{Domain Model} \mid \textit{Table Module}$
$\textit{Template View} \overset{\neg\mathrm{C41}}{\rightarrow} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Transform View} \overset{\neg\mathrm{C41}}{\rightarrow} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Page Controller} \overset{\neg\mathrm{C41}}{\rightarrow} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Front Controller} \overset{\neg\mathrm{C41}}{\rightarrow} \textit{Domain Model} \mid \textit{Table Module} \mid \textit{Transaction Script}$
$\textit{Domain Model} \overset{\mathrm{C21}}{\rightarrow} \textit{Active Record} \bullet \textit{Domain Model} \overset{\mathrm{C23}}{\rightarrow} \textit{Data Mapper}$
$\textit{Table Module} \rightarrow \textit{Table Data Gateway} \mid \textit{Row Data Gateway}$
$\textit{Transaction Script} \rightarrow \textit{Table Data Gateway} \mid \textit{Row Data Gateway}$
$\textit{Table Data Gateway} \rightarrow \textit{Record Set}$ {C111} $\bullet$ Table Data Gateway $\overset{\mathrm{C42}}{\rightarrow}$ Data Transfer Object
$\textit{Data Mapper} \overset{\textit{datasource}}{\leftrightarrow} \textit{Active Record}$
$\textit{Table Data Gateway} \overset{\textit{datasource}}{\leftrightarrow} \textit{Row Data Gateway}$ {C112}
$\textit{Optimistic Offline Lock} \overset{\textit{concurrency}}{\leftrightarrow} \textit{Pessimistic Offline Lock}$ {C112}
$\textit{Client Session State} \overset{\textit{sessionstate}}{\leftrightarrow} \textit{Server Session State}$ {C112}
$\textit{FrontController} \uparrow \textit{Controller} \bullet \textit{PageController} \uparrow \textit{Controller}$
$\textit{Data Mapper} \uparrow \textit{Mapper}$
$\textit{Table Data Gateway} \uparrow \textit{Gateway} \bullet \textit{Row Data Gateway} \uparrow \textit{Gateway}$

Page Controller $\approx$ {C11} $\bullet$ *Front Controller* $\approx$ {C12} (Part C)
Template View $\approx$ {C61} $\bullet$ *Transform View* $\approx$ {C62}
Transaction Script $\approx$ {C11 and C21 and C31}
Table Data Gateway $\approx$ {*insert*() *parameter list* $\subseteq$ *update*() *parameter list*}
Active Record $\approx$ *Template View* {C121}
Domain Model $\approx$ {C23}
Service Layer $\approx$ {C41} $\bullet$ *Remote Facade* $\approx$ {C42}
Data Transfer Object $\approx$ {C42}
DomainModel $\approx$ {C23}
Optimistic Offline Lock $\approx$ {C43 and C51} , *Pessimistic Offline Lock* $\approx$ {C43 and C52}
Client Session State $\approx$ {C44} $\bullet$ *Server Session State* $\approx$ {C44}

C11: Tool is .Net , C12: Tool is Java
C21: Domain structure is simple , C22: Domain structure is moderate
C23: Domain structure is complex
C31: Designer is novice , C32: Designer is intermediate , C33: Designer is expert
C41: Designer wants Service Layer , C42: Designer wants Distributed Layer
C43: Designer wants Concurrency Layer , C44: Designer wants Session State Layer
C51: Chance of conflict is low , C52: Chance of conflict is high
C61: View is built using HTML , C62: View is built using XSLT
C111: Return type of every `find()` operation in Table Data Gateway pattern is Record Set
C112: Two patterns are applied for the same unit of work C121: The parameters of the operations of the Active Record pattern must match with the attributes of Template View

---

Figure 3: *PofEAA* Rule Set: Syntactic Rules (Parts A & B) and Semantic Rules (Part C)

5.3. ArgoPLV Architecture

In this section, we show how four modules of PLV, are hard coded into a modeling tool to build "A PLV for *PofEAA*." As the modeling tool, we have selected ArgoUML, hence, the resulted tool is called "ArgoPLV." At the end, we show the usefulness of ArgoPLV in designing with patterns.

ArgoUML [Tig09] is an open-source UML modeling tool as well as a design critiquing system. ArgoUML has predefined agents, called *critics*, that are constantly investigating the current model, as well as *wizards* which helps the designer solve the problem automatically. *Critics* and *wizards* are written in Java and are compiled as part of the tool, i.e., they are not user defined. Figure 4 shows the architecture of ArgoPLV as an extension to ArgoUML.

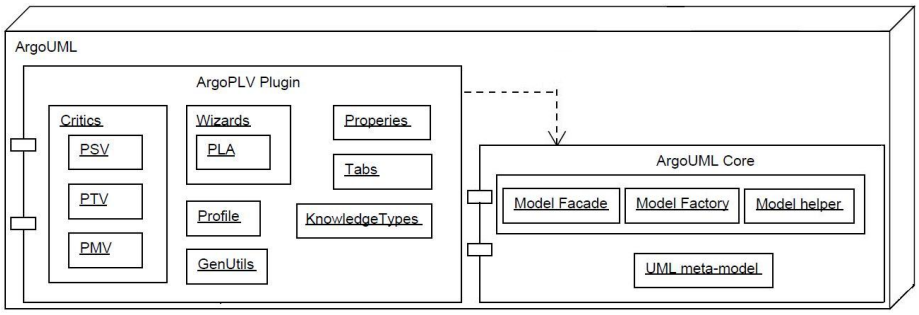

Figure 4: ArgoPLV Architecture

The PSV in ArgoPLV is built by hard coding the "*PofEAA* Rule Set - Structural Rules" into the *critics*. That means, for each *PofEAA* pattern, we have one *critic* which verifies the structure of the pattern and detects the errors. Hence, PSV is implemented by 23 *critics* (one *critic* per pattern).

The PTV in ArgoPLV is built by hard coding the two groups of syntactic rules of the *PofEAA* PL into 2 *critics*.

The PMV in ArgoPLV is built by hard coding the semantic rules of the "*PofEAA* Rule Set" into the *critics*. Dealing with semantic issues in the *critics* is almost similar to the syntactic ones, because most semantic *critics* need to check the context information. PMV is implemented in ArgoPLV by 10 *critics*.

The PLA module of ArgoPLV is built via several extensions to the ArgoUML. First, the UI is extended by adding a new tab named "**Detected Patterns**" to the Details Pane, as well as three new Knowledge Types to the ToDo List: ***PofEAA* Structure**, ***PofEAA* Syntax**, and ***PofEAA* Semantics**, to report the corresponding three groups of errors. Second, for each error, the error message along with the guidelines on how to fix the problem are defined in a uniform manner. Third, a Design Rationale is created that keeps track of each session of the ArgoPLV by recording the actions performed by the *wizards*. Fourth, to fulfill the most important responsibility of PLA, the *wizard* classes are written to fix the problems automatically. In total, 50 *wizards* are written as part of the PLA module for ArgoPLV.

5.4. Using ArgoPLV

How does the ArgoPLV tool help a designer in applying the *PofEAA*? ArgoUML, and hence ArgoPLV, is an interactive modeling tool. By using the "Sign" of the pattern, the designer shows his/her intention in applying a pattern. Then, the corresponding *critic* is activated and verifies the structure of the pattern (PSV module). If any of the structural criteria fail, the *critic* is triggered and a ToDo Item (*critique*) will be posted under ***PofEAA* Structure**. By selecting a ToDo Item, its description will be shown in the Details Pane, and upon the user's request, the *wizard* for the *critic* will be executed and the problems found in the pattern usage will be fixed (PLA module). The details of the correctly applied patterns is displayed in the **Detected Patterns** tab. If a syntactic problem is detected in the pattern combinations (by the PTV module), one of the syntactic *critics* is triggered and a *critique* will be posted under ***PofEAA* Syntax**. If any of the semantic criteria fail, one of the semantic *critics* is triggered (by the PMV module), and a *critique* will be posted under **PofEAA Semantics**. In either of the cases, the *wizards* might be available to fix the problem automatically, or the designer is guided to repair the error manually.

Due to lack of space, we abstain from showing an interactive session with the tool. Instead, we show how ArgoPLV helps a designer verify a (partial) model built for an *Online Student Registration System* based upon the *PofEAA* patterns. We suppose that the designer has utilized the stereotypes of a UML Profile to specify the patterns.

Step 1: Load the Model into ArgoPLV. After loading the model, the context information can be investigated. Figure 5 shows the context information, i.e., the tagged values of the «PofEAA» main package. Figure 6 shows the class diagram of the design loaded into ArgoPLV. Note that, we have merely shown the diagram as it is appeared in the Editing Pane of ArgoUML.

Target: Package (Main) TD

Tag	Value
ChanceOfConflict	Low
Complexity	Complex
ConcurrencyLayer	No
DistributedLayer	Yes
Expertise	Expert
ServiceLayer	Yes
SessionStateLayer	No
Tool	Java
ViewBuilt	XSLT

Figure 5: The Context Information Considered for the Model

Step 2: Check the Structural Problems of the Model. Several structural errors are detected by the PSV and reported by the PLA. The errors and their causes are as follows (See Figure 6).

1. Structural problem in using the Front Controller pattern; The causes are:
   (a) missing "doGet" and "doPost" operations in Handler ("MyWebServelet"),
   (b) having a non-abstract Command class ("CommandCls"), and
   (c) missing "process" operation in a Concrete Command ("CalculateGPA").
2. Structural problem in using the Template View pattern; Due to missing Helper class for one of the Template View classes ("BrowseProfsTV").
3. Structural problem in using the Domain Model pattern; The cause is that one of the Domain Model classes ("Professor") has no operation.
4. Structural problem in using the Data Mapper pattern: Due to missing the "delete" operation in one of the Data Mapper classes ("PersonMapper').

Step 3: Fix the Structural Problems. The designer asks PLA to fix the structural problems automatically. Then the following repairs are applied to the model.

1. (a) The "doGetOp" and "doPostOp" operations are added to the "MyWebServelet" class.
   (b) The "Command" class is set as an abstract class.
   (c) A "processOp" operation is added to the "CalculateGPA" class.
2. A "HelperCls" class is created as a supplier for "BrowseProfsTV."
3. A "newOp" operation is added to the "Professor" class.
4. A "deleteOp" operation is added to the "PersonMapper" class.

Step 4: Check the Syntactic Problems of the Model. The following list shows the syntactic errors detected by the PTV and reported by the PLA.

1. Syntactic problem in the layering of the model; Due to missing Distributed Layer in the model while the designer has shown his/her intention of having a Distributed Layer (See Figure 5).
2. Syntactic problem regarding Pattern-Layer Relationships; The cause is that Data Transfer Object ("CourseList") is not located in its corresponding package (Distributed Layer).
3. Syntactic problem in the Concurrency Layer; Due to existence of two conflicting patterns (two "AddressLock" classes) in the Concurrency package.

Step 5: Fix the Syntactic Problems. The designer asks for help from PLA. The following repairs are applied to the model (last one is manual).

1. PLA creates a Distributed Layer ("distributedPkg" package) inside the Main package.
2. PLA moves the "CourseList" class into the Distributed Layer.
3. The designer removes the "AddressLock" class (the class with stereotype «PessimisticOfflineLock») from the Concurrency package.

Step 6: Check the Semantic Problems of the Model. The following list shows the semantic errors detected by the PMV and reported by the PLA.

1. Semantic Problem regarding the View Layer; The cause is that setting "ViewBuilt=XSLT" contradicts with the usage of Template View pattern.
2. Semantic Problem regarding the Service Layer; The cause is that the designer wants to have a Service Layer in his/her design ("ServiceLayer=Yes") but, there is no such layer in the model.

Step 7: Fix the Semantic Problems. The designer asks for help from PLA. The following repairs are applied to the model, manually by the designer but via the support provided by PLA.

1. The designer sets "ViewBuilt=HTML" via the text box provided by PLA.
2. The designer decides not to have a Service Layer and sets "ServiceLayer=NO" via the text box provided by PLA.

Step 8: Final Design. Figure 7 shows the package diagram of the design model after all the errors caught by the ArgoPLV are fixed as described in the above.

6. Conclusion and Future Work

In this paper, we introduced a process, named Pattern Language Verifier (PLV), for verifying the use of a PL in model-driven design. We elaborated that building a PLV for a given PL, requires the structural, syntactic, and semantic rules of the PL to be explicitly and precisely defined. We used the three formalisms, defined by the authors, for specifying these three groups of rules. We utilized a naming convention to ease the detection of pattern elements, as well as eliminating the problem of pattern selection from the scope of the work. In our case study, we selected a subset of *PofEAA* as the PL. We extracted several advices from the *PofEAA* book, and transformed them into formal rules which are used by the PLV. Rules are hand coded into the ArgoUML modeling tool to obtain a PLV for *PofEAA*, named ArgoPLV. Then, we showed how the tool can be used to verify a design model, and to report the errors.

Comparing to the previous works of the authors [ZKB08, ZB09], this paper includes novel contributions as improvements on the PLV process and its modules as well as more detailed case study and examples. Separating the Rules from Advices is a new idea that we have found it very essential when trying to extract formalized rules out of prose pattern descriptions. We have improved the PTV module by focusing more on the PL issues and by dividing Syntactic rules into two separate aspects. The case study presented in this paper is closer to a real web-application comparing to our simple examples given in our previous works.

Due to the originality of the presented idea, and lack of similar research, no comparison is made between our work with other researches. However the validation of the process and the usefulness of the tool is done via the case study by anecdotal evidence.

There are several paths to extend and improve the work presented in this paper. Generalizing the idea of PLV and the experiences gained in this work towards a framework for "Pattern Language Verification" would result in a valuable contribution to patterns and PLs. The PLV process can also be enriched with the idea of systematic pattern selection presented by Zdun [Zdu07]. While people consider existing PLs, working on formalism of patterns and PLs is a real need, especially, if we look for more help from the CASE tools. The PLV process is mimicking the analysis part of a compiler. Investigating the synthesis (code-generation) part, may lead to a research which consolidates the PLV with the Model-Driven Software Engineering (MDSE) approaches that promote full code generation from the UML models, such as Executable UML (xUML) [MB02].

References

[A+77] Christopher Alexander et al. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[Béz06] Jean Bézivin. Model driven engineering: An emerging technical space. In Ralf Lämmel et al., editors, *GTTSE*, volume 4143 of *Lecture Notes in Computer Science*, pages 36–64. Springer, 2006.

[BHS07] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*, volume 5. John Wiley & Sons, 2007.

[BP02] Federico Bergenti and Agostino Poggi. Improving UML designs using automatic design pattern detection. In Shi-Kuo Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 2, pages 771–784. World Scientific Publishing, 2002.

[Egy07] Alexander Egyed. Fixing inconsistencies in UML design models. In *ICSE*, pages 292–301. IEEE Computer Society Press, 2007.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, first edition, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[KAZ11] Ahmad Waqas Kamal, Paris Avgeriou, and Uwe Zdun. The use of pattern participants relationships for integrating patterns: a controlled experiment. *Software Practice & Experience*, pages n/a–n/a, 2011.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, Jan. 1998.

[Knu64] Donald E. Knuth. Backus normal form vs. Backus Naur form. *Commun. ACM*, 7(12):735–736, Dec. 1964.

[Lar05] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, third edition, 2005.

[LEM02] WenQian Liu, Steve Easterbrook, and John Mylopoulos. Rule-based detection of inconsistency in UML models. In *Workshop on Consistency Problems in UML-Based Software Development*, pages 106–123, Dresden, Germany, Oct. 2002.

[Lov06] Howard C. Lovatt. *A Pattern Enforcing Compiler (PEC) For Java*. PhD thesis, Macquarie University, Australia, 2006. Online at `https://pec.dev.java.net/nonav/introduction/index.html`.

[MB02] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, first edition, 2002.

[Nob98] James Noble. Classifying relationships between object-oriented design patterns. In *Software Engineering Conference, 1998. Proceedings. 1998 Australian*, pages 98–107, Nov. 1998.

[Obj09] Object Management Group (OMG). Model Driven Architecture (MDA). `http://www.omg.org/mda/`, [July 1, 2009].

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, Feb. 2006.

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, Sep. 2003.

[Sel06] Bran Selic. Model-driven development: Its essence and opportunities. In *Proceedings of ISORC'06 Symposium*, pages 313–319, Los Alamitos, CA, USA, 2006. IEEE Computer Society Press.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Software Eng.*, 32(11):896–909, Nov. 2006.

[Tig09] Tigris.org. ArgoUML official web site. `http://argouml.tigris.org/`, [July 1, 2009].

[Unh05] Bhuvan Unhelkar. *Verification and Validation for Quality of UML 2.0 Models*. John Wiley & Sons, 2005.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the Technology of Object-Oriented Languages (TOOLS 26) Conference USA 1998*, pages 112–124. IEEE Computer Society Press, 1998.

[ZA05] Uwe Zdun and Paris Avgeriou. Modeling architectural patterns using architectural primitives. In *Proceedings of OOPSLA '05 Conference*, pages 133–146, New York, NY, USA, 2005. ACM.

[Zam09] Bahman Zamani. *On Verifying the Use of a Pattern Language in Model Driven Design*. PhD thesis, Concordia University, Montreal, Quebec, Canada, 2009.

[ZB07] Bahman Zamani and Greg Butler. Critiquing the application of pattern languages on UML models. In *Workshop on Quality in Modeling, MODELS2007 Conference*, pages 18–35, Nashville, TN, USA, 2007.

[ZB09] Bahman Zamani and Greg Butler. Describing pattern languages for checking design models. In Shahida Sulaiman and Noor Maizura Mohamad Noor, editors, *APSEC*, pages 197–204. IEEE Computer Society, 2009.

[ZBK09] Bahman Zamani, Greg Butler, and Sahar Kayhani. Tool support for pattern selection and use. *Electr. Notes Theor. Comput. Sci.*, 233:127–142, Mar. 2009.

[Zdu07] Uwe Zdun. Systematic pattern selection using pattern language grammars and design space analysis. *Software Practice & Experience*, 37(9):983–1016, Jul. 2007.

[Zim95] Walter Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.

[ZKB08] Bahman Zamani, Sahar Kayhani, and Greg Butler. A pattern language verifier for web-based enterprise applications. In Krzysztof Czarnecki et al., editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 553–567. Springer, 2008.

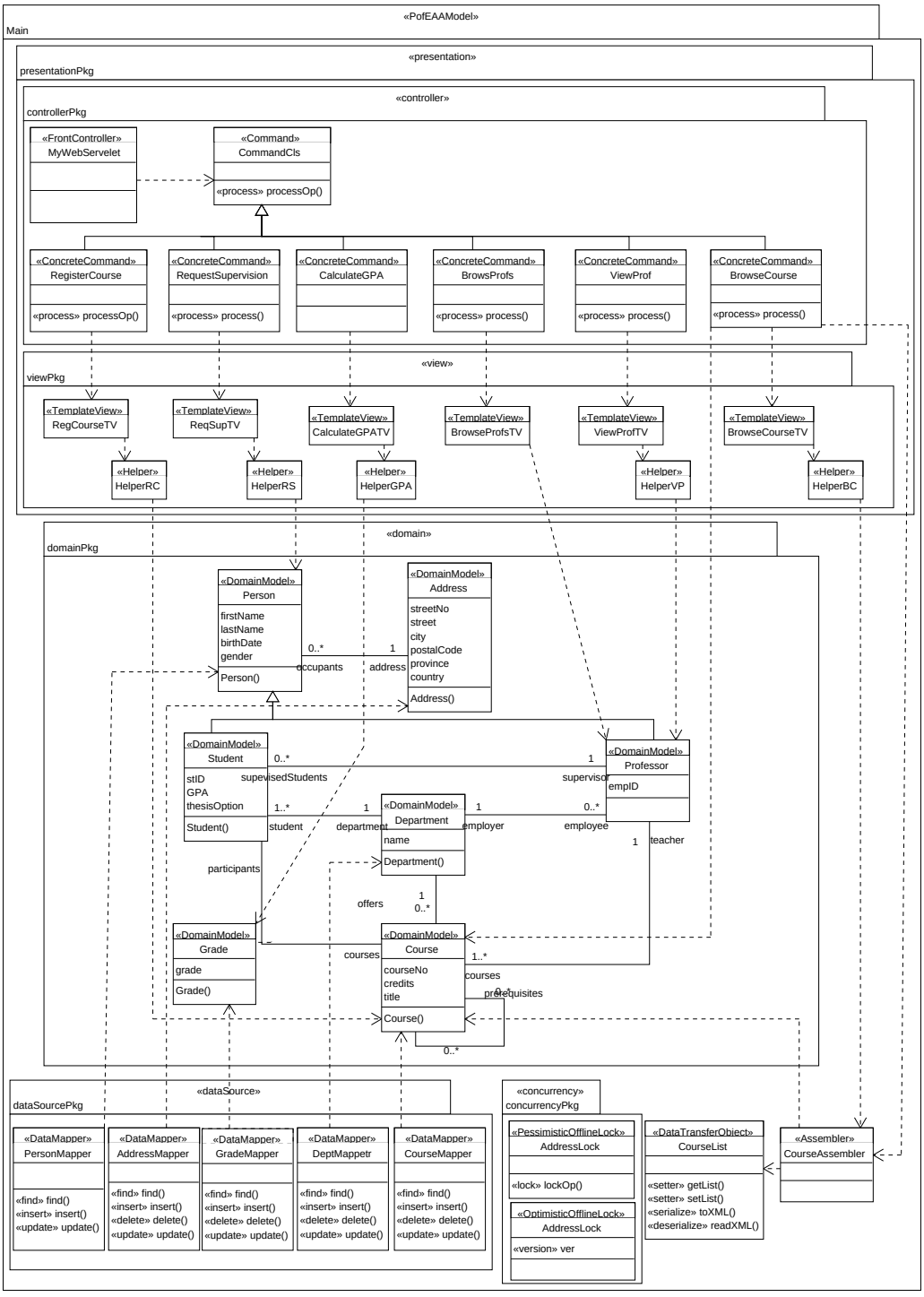

Figure 6: A Design Model for Online Student Registration System using *PofEAA* Patterns

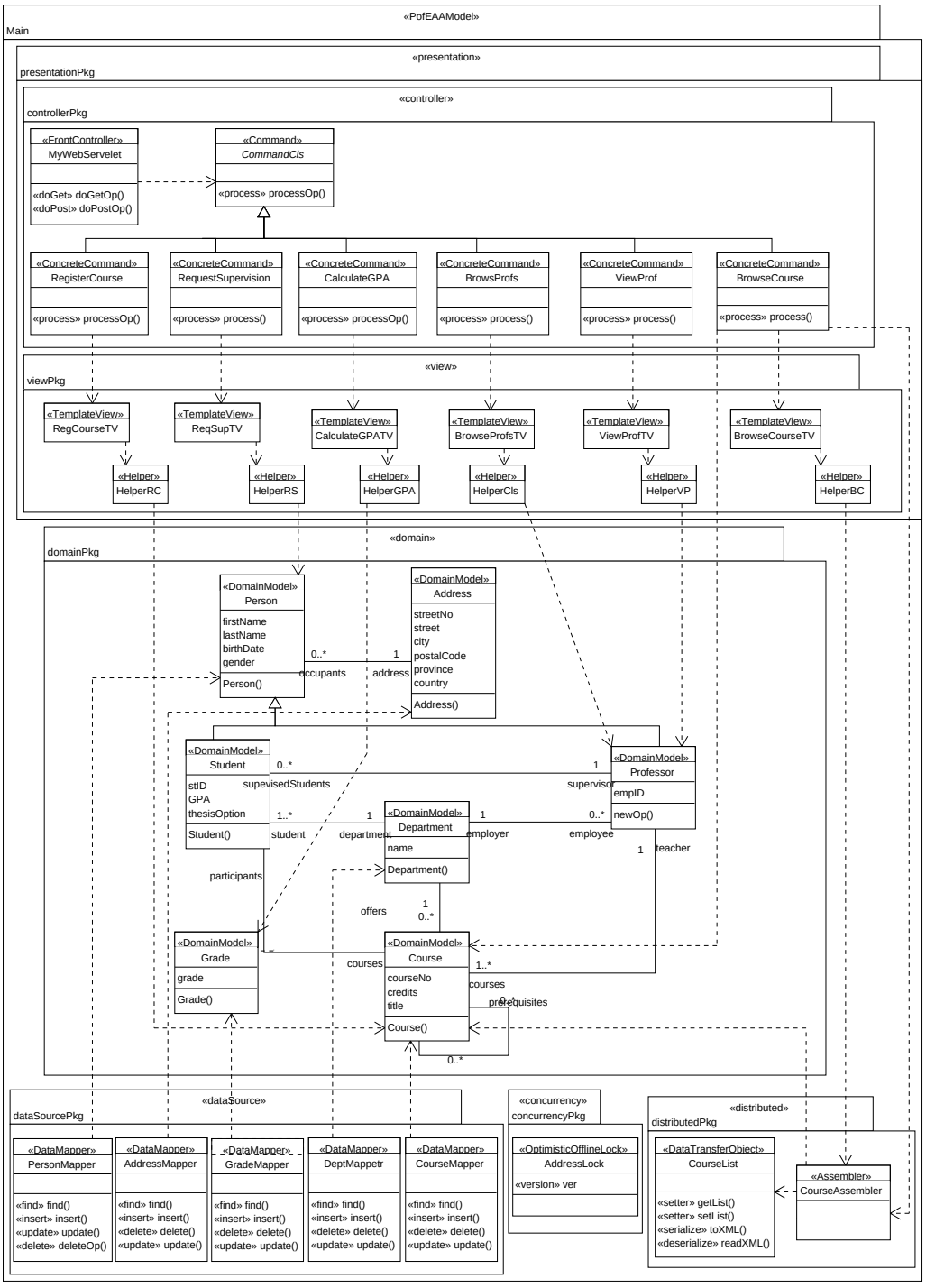

Figure 7: Design of Online Student Registration System - Refined by ArgoPLV