Extendable Multiplatform Approach to the Development of the Web Business Applications

Vladimir Balać*, Milan Vidaković**
* Faculty of Technical Sciences/Computing and Control Engineering, Novi Sad, Serbia
** Faculty of Technical Sciences/Computing and Control Engineering, Novi Sad, Serbia
sermasint@gmail.com, minja@uns.ac.rs

***Abstract*—We present the OdinModel framework - an extendable multiplatform approach to the development of the web business applications. The framework enables the full development potential of the application's model through the platform independent abstractions. Those abstractions allow the full code generation of the application from a single abstract model to the multiple target platforms. The model covers both the common and the platform specific development concepts of the different target platforms, which makes it unique. In addition, we can extend the model with any existing development concept of any target platform. The framework with such model provides both the generic and custom modeling of the complete Model, View and Controller parts of the application. OdinModel uses existing development tools for the implementation of the application from the model. It does not force any development technology over some other. Instead, the framework provides a hub from which the web application developers can choose their favorite approach. Currently, the framework covers the development for Java, Python and WebDSL platforms. Support of these three platforms and the extendibility of the framework guarantee the framework support for any development platform.**

I. INTRODUCTION

The development of the Model-View-Controller (MVC) web business applications with the general-purpose programming languages, like Java and Python, means that the realization of the domain problem solution is on the level of the programming language details. This means that, if we want to develop the same application on the Java and Python platforms, we outline the one same solution, yet write the code for it twice, for each platform.

Java and Python platforms have various development support tools that simplify the development as much as possible by doing the grunt work for the developers. These tools, however, produce the code that covers specific application's components, while the code outside those components the developers write manually. These tools also produce the code only for their target platform.

The development with the general-purpose programming languages generally has two main parts [1, 2]. In the first part, the developers create the models of the applications in textual and diagram forms. In the second part, the developers deal with the programming implementation of the models from the first part i.e. code writing. In practice, the developers give more importance to the code, than to the models [1, 2]. Consequently, when there are new changes to the application, the developers make the changes to the code, but not to the models, thus leaving the models inconsistent with the implementation of the application. This renders models practically useless for the further development cycle, and the invested work to make the models in the beginning a waste of time and effort [3].

To use the full development potential of the models, the developers may adopt one of Model-Driven Engineering (MDE) paradigms for the application development [2]. MDE offers higher levels of models abstraction, code writing automation, portability, interoperability and reusability than the programming languages [4]. MDE development principles propose use of the models as formal, complete and consistent abstraction of the applications [5]. From those models, the developers can generate the target application's code automatically. The abstraction improves the development process by allowing the developers to shift their focus from the programming languages to the models of the problem domain [2, 6]. The generation of the complete code of the application removes manual writing of the code during the implementation, hides complexity of the development and improves the quality of the application and its code [7, 8, 9].

We propose an extendable multiplatform MDE approach to the development that we call the OdinModel framework. What sets apart our framework from other solutions is that it encapsulates common features of three application's parts in a platform independent manner. We can develop the Model, the View and the Controller part of the application through the one platform independent model that we call Odin model. Our framework currently encapsulates common features of Java, Python and WebDSL [10] applications.

With the OdinModel framework, we write only the solution specific code, from which the accompanying Java, Python or WebDSL code the framework automatically generates. The OdinModel framework produces the complete code and eliminates the manual code writing. By using automation through the generators, we avoid direct work with any tool on any platform. However, the OdinModel framework recognizes that the use of the programming languages and their respective supporting development tools increases the developer's productivity by four hundred percent [11, 12]. In the light of that, the OdinModel framework combines MDE principles and use of proven

development tools with goal to improve the overall productivity, quality and amount of time needed for the development process.

With the OdinModel framework, we aim to make the application development more efficient with the right level of abstraction. We want to describe solutions naturally and to hide unnecessary details, as stated in [13]. Since there are many details in the application's code, we try to automate everything that is not critical, without loss of the expressiveness. We show that this concept can work for Java, Python and WebDSL platforms. Support of these three platforms and the extendibility of the framework, guarantees the framework support for any development platform.

II. RELATED WORK

Model-Driven Architecture (MDA) is MDE paradigm where the developers rely on the standards, primary Unified Modeling Language (UML) and Meta-Object Facility [14, 15]. At first glance, the approaches that adopt MDA are very similar to the OdinModel framework. Analyzing works such as MODEWIS [4, 11], UWA [3], UWE [16, 17], MIDAS [18], ASM with Webile [19], Netsilon [20] and Model2Roo [1], we recognize the same ideas as in the OdinModel framework. However, in MDA approach, the developers use UML to define the three distinct abstract platform independent application's models according to Meta-Object Facility principles [11, 15]. With the OdinModel framework, the developers use a custom modeling language to define one platform independent model. This is the key difference between MDA and OdinModel approach.

Another difference is that UML is not a domain-specific modeling language [21]. Since UML is not a domain-specific, the developers manually program the missing domain-specific semantics or use UML profiles, limited extensions of the language [5]. With the OdinModel framework, the abstract concepts are domain-specific.

With UML, the models and the underlying code are on the same level of abstraction [21]. The same information is in the model and the code i.e. visual and textual presentation. In contrast, OdinModel's modeling language has a higher level of abstraction and each symbol on the model is worth several lines of the code.

Domain-Specific Modeling (DSM) is MDE paradigm where the primary artifact in the application development process is the one abstract platform independent model and the full application's code generation from that model is obligatory [2]. In DSM approach, the focus is on the development in the one specific domain and the developers specify the domain problem solution using the domain concepts. In other words, the modeling language takes the role of the programming language. A modeling language, which directly represents the problems in the specific domain, is a Domain-Specific Language (DSL) [13, 15]. DSL is the integral part of DSM approach, along with the domain-specific code generator and the domain framework [2]. The OdinModel framework adopts DSM paradigm. We recognize the related works that adopt DSM paradigm in the next approaches: DOOMLite [22], WebML [23, 24], and WebDSL.

The main difference between our OdinModel framework and the related DSM approaches is that OdinModel provides the full code generation for the multiple platforms from the start. Odin model abstracts not just common features of the applications on a single platform, as the related approaches do, but common features of the applications with the different underlying platforms. Therefore, our model abstracts and covers both similarities and differences of the different platforms, which, to our knowledge, makes it unique. Other significant differences between OdinModel and the related DSM approaches we present in Table 1.

III. ODINMODEL SPECIFICATION

The key of OdinModel specification is Odin meta-model. It provides the specification of the abstractions of the features needed for the development of the Model, the View and the Controller parts of the applications. These abstractions are the result of the analysis of all the development concepts that the developers must define for each application's part separately. Odin meta-model is, essentially, union of these separate abstractions. Since we focus on multiplatform development, the abstractions cover both intersection and complement sets of the development features from different platforms. These two sets of features are a foundation for the specification of Odin DSL.

The OdinModel framework adopts the four-layered architecture of Meta-Object Facility standard. Essentially, this standard is a specification for definition of DSL [13]. Table 2 shows OdinModel's four-layered architecture. Odin DSL, in this stage, provides concepts that are abstractions of Java, Python and WebDSL features. There are two types of the features: common for all three platforms, and the platform specific. The Platform specific features are important because they allow customization and do not force the use of the generic solutions. However, we offer the generic solutions too.

The Model part of the application manages data access and persistence. With the OdinModel framework, we encourage the use of the tools, which automatically manage most of the database persistence. This means that the Model part of our meta-model only needs to cover the specification of entities, their attributes and their relations. Fig. 1 shows our definition of the *Entity* class. The meta-model class *EntityModel* is the root class and it contains the main domain classes i.e. all the other elements of the meta-model.

TABLE I.
Comparison of DSM approaches

Approach	Multiple target platforms	Custom user code	Custom user interface	Visual editor	Full MVC model
DOOMLite					
WebML		*	*	*	*
WebDSL		*	*		*
OdinModel	*	*	*	*	*

TABLE II.
OdinModel's four-layered architecture

Level	Layer	Implementation
M3	Meta-meta-model	Ecore meta-model
M2	Meta-model	Odin meta-model (Odin DSL)
M1	Model	Odin model
M0	Real world objects	Instance of Odin model

The meta-model classes *NumberField*, *StringField*, *EmailField*, *DateField* and *Fields* represent the attributes of the programming language classes i.e. entity fields. Fig. 2 shows the four types of the fields that OdinModel framework currently supports. Since the all field classes have some common attributes, we define those as the attributes of the super class *Fields*. We define the specific attributes of each field type in their own the meta-model class.

The Class *NumberField* defines the fields with the numerical values. This class can define the three types of the field: ordinary number, primary key and interval of numbers. When we define the primary key field, we can also define the type of the primary key generation through the attribute *generationType*. The same goes for the interval field where we can also define the type of interval through the attribute *intervalType*.

The Class *StringField* defines the fields with a string of characters as a value. It has five specific attributes, where two of them represent the constraints, and the other three define the combo box, a special type of the textual field.

The meta-model classes *OneToMany*, *ManyToOne*, *OneToOne*, *ManyToMany* and *Relations* specify the all four possible types of the relations between entities. The super class *Relations* specifies the common attributes of the relations.

The View part of the applications manages visual presentation. Through the code generation, the OdinModel framework provides the default Create-Read-Update-Delete (CRUD) user interface forms. The entities are the base for the generation of the CRUD forms. The CRUD forms contain the entity attributes as the input or the output form fields. The OdinModel framework provides navigation between these CRUD forms, through the default application's menu.

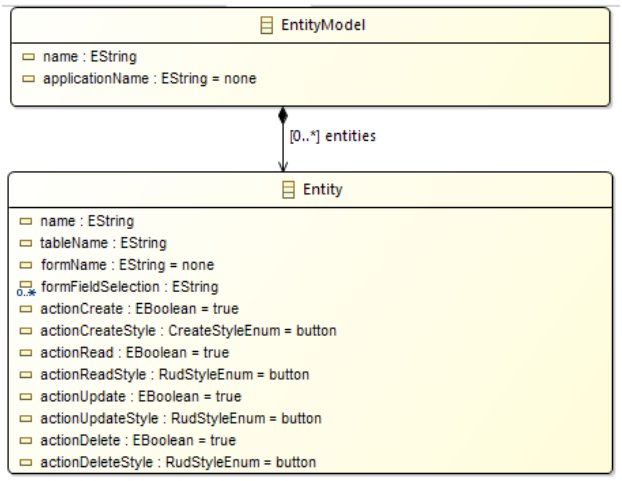

Figure 1. Entity class

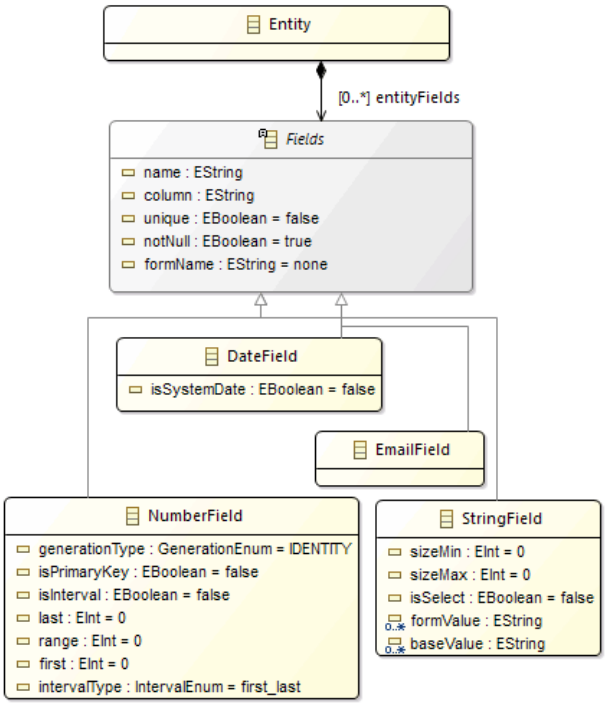

Figure 2. Field classes

The OdinModel framework also allows the customization of the content and the visual presentation of the CRUD forms and the application's menu. We can customize which CRUD operations will be visible on the forms and their visual style. The visual style covers the combinations of buttons, links, tables and fields. We define two meta-model classes with purpose to enable the menu customization, which we present in Fig. 3.

The Controller tier of the applications manages the page navigation, the input validation and the operations. The Odin meta-model specifies two sets of the operations. One set includes CRUD operations. The other set, which extends the CRUD set, includes the user's custom operations. We define the custom operations through the custom method classes. Through those classes, we define the control flow of the operation. We can declare the variables, assign the values to the variables, define IF conditions and define WHILE loops.

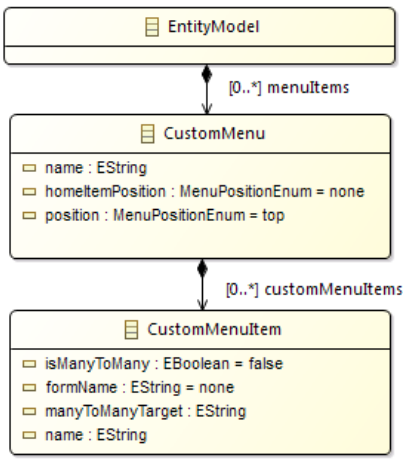

Figure 3. Custom menu classes

IV. ODINMODEL IMPLEMENTATION

The OdinModel framework provides the development environment, which contains Odin DSL, a visual editor for Odin DSL and the code generators. The developers through the visual editor use DSL to create the platform independent Odin model, which specifies the application. The code generators produce the complete application's code from the Odin model.

We now present the implementation of the OdinModel framework through the case study. In Fig. 4, we display the Odin model of a Sport center, which has five persistence objects and covers all four possible types of the relations between those objects.

The use of the OdinModel framework reduces the developer's work to the modeling of the domain concepts that exist in the Odin DSL. The Sport center model has all that is necessary for the specification of the Sport center application. Behind this model, there is an Extensible Markup Language (XML) code, not the programming language code. The code generators use that XML syntax to produce the application's code. In Fig. 5, Fig. 6, and Fig. 7, we present the generated code for the Member entity on the all target platforms.

OdinModel generators produce the code from the symbols, the arguments and the values of the symbols, and the relations between the symbols. If we make changes in the model, those generators apply changes to the all generated files. The generators are extendable. This means that whenever we define, for example, some new Java or Python domain concept in the meta-model, we adapt the corresponding generator. Java generator ignores Python and WebDSL specifics and vice-versa. In other words, if we specify the model with Java specifics, and then choose Python generator, the generator will generate Python application without problems. The generated application is ready-to-deploy. In Fig. 8, Fig. 9, and Fig. 10, we present the CRUD forms, which correspond to the generated codes.

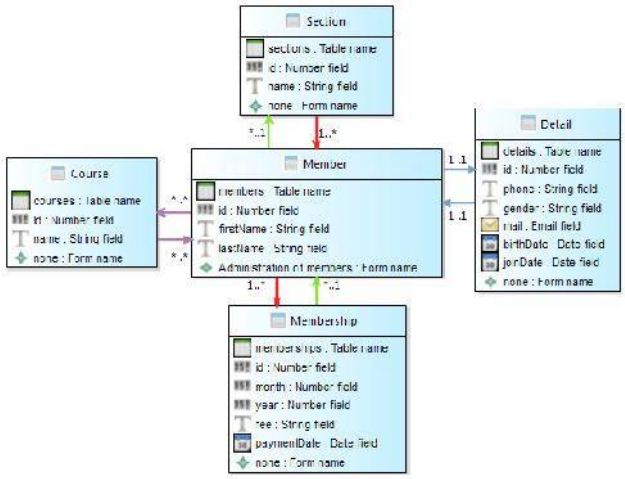

Figure 4. Sport center Odin model

```
… left out code …
@Entity
@Table(name = "members")
public class Member implements Serializable{

  @Id
  @GeneratedValue(strategy= GenerationType.IDENTITY)
  @Column(name = "id")
  private int id;

  @NotNull
  @Size(min = 3, max = 30)
  @Column(name = "first_name")
  String firstName;

  @NotNull
  @Size(min = 3, max = 30)
  @Column(name = "last_name")
  String lastName;

  @ManyToOne(cascade={CascadeType.REFRESH})
  public Section section;

  @ManyToMany(cascade={}, fetch=FetchType.EAGER)
  private Collection<Course>courses = new ArrayList<Course>();

  @OneToMany(cascade = {CascadeType.ALL},
  fetch=FetchType.EAGER, mappedBy = "member")
  @Fetch(value = FetchMode.SUBSELECT)
  public Collection<Membership> memberships;

  @OneToOne(cascade = {CascadeType.ALL})
  public Detail detail;
… left out code …
```

Figure 5. The generated Java class for Member entity

```
class Member(models.Model):
  id = models.AutoField(primary_key=True)

  first_name = models.CharField('First name',
  validators=[RegexValidator(regex='^.{3}$',
  message='Length has to be 3 ', code='nomatch')], max_length=30)

  last_name = models.CharField('Last name',
  validators=[RegexValidator(regex='^.{3}$',
  message='Length has to be 3 ', code='nomatch')], max_length=30)

  detail = models.OneToOneField('Detail')

  section = models.ForeignKey(Section)

  courses = models.ManyToManyField(Course)

  class Meta:

    db_table = "members"
… left out code …
```

Figure 6. The generated Python class for Member entity

```
… left out code …
  entity Member{
    firstName :: String(length = 3)
    lastName :: String( )
    name :: String := " " +firstName +" " +" " +lastName +" " +" "
    //1-1 relation
    detail <> Detail
    //m-m relation
    courses -> Set<Course> (inverse=Course.members)
    //m-1 relation
    section -> Section
    //1-m relation
    memberships -> Set<Membership> (inverse=Membership.member)
  }
… left out code …
```

Figure 7. The generated WebDSL class for Member entity

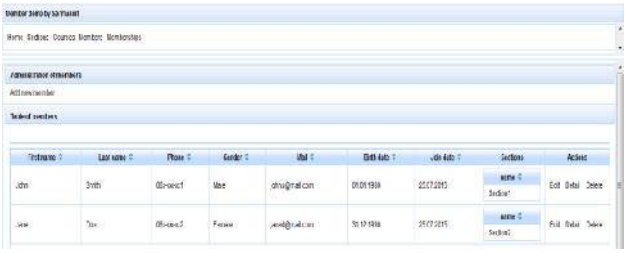

Figure 8. Java CRUD form Member

Figure 9. Python CRUD form Member

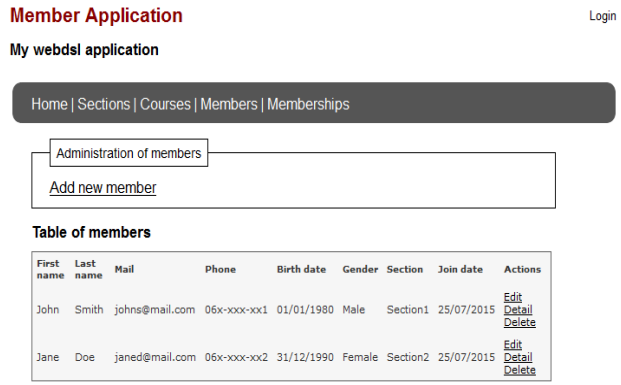

Figure 10. WebDSL CRUD form Member

V. CONCLUSION

The OdinModel framework improves productivity, portability, maintainability, reusability, automation and quality of MVC web business applications development process. The framework provides the visual modeling of the abstractions of the domain concepts in an original DSL. It provides the full multiplatform code generation from a single abstract model. We validate Odin model's level of abstraction by generating Java, Python, and WebDSL applications, directly from the model,. The incorporation of the proven development technologies shows openness and the extensibility of the approach. In addition to the default code generation, we provide the modeling of the custom user operations and the modeling of the custom user interface. The OdinModel framework does not force any development technology and approach over some other. Instead, it provides a hub from which the developers can choose their favorite development approach. Since the development tools incorporate best practices to produce code, we build on them.

The uniqueness of the OdinModel framework lies in its DSL, which covers both the similarities and the differences of the different target platforms. More precisely, it covers the common and the specific features of the target platforms relevant to the development. Odin DSL does not discard the specifics, but it does not force them either, which makes the Odin model platform independent, as well as composite. The DSL provides the developers with the abstract concepts and the platform specific details.

REFERENCES

[1] Castrejón, Juan Carlos, Rosa López-Landa, and Rafael Lozano. "Model2Roo: A model driven approach for web application development based on the Eclipse Modeling Framework and Spring Roo." In Electrical Communications and Computers (CONIELECOMP), 2011 21st International Conference on, pp. 82-87. IEEE, 2011.

[2] Kelly, Steven, and Juha-Pekka Tolvanen. Domain-specific modeling: enabling full code generation. John Wiley & Sons, 2008.

[3] Distante, Damiano, Paola Pedone, Gustavo Rossi, and Gerardo Canfora. "Model-driven development of web applications with UWA, MVC and JavaServer faces." In *Web Engineering*, pp. 457-472. Springer Berlin Heidelberg, 2007.

[4] Fatolahi, Ali, and Stéphane S. Somé. "Assessing a Model-Driven Web-ApplicationEngineering Approach."*Journal of Software Engineering and Applications* 7, no. 05(2014): 360.

[5] Voelter, Markus, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[6] Calic, Tihomir, Sergiu Dascalu, and Dwight Egbert. "Tools for MDA software development: Evaluation criteria and set of desirable features." In Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on, pp. 44-50. IEEE, 2008.

[7] Tolvanen, Juha-Pekka. "Domain-specific modeling for full code generation." *Methods & Tools* 13, no. 3 (2005): 14-23.

[8] Hemel, Zef, Lennart CL Kats, Danny M. Groenewegen, and Eelco Visser. "Code generation by model transformation: a case study in transformation modularity." Software & Systems Modeling 9, no. 3 (2010): 375-402.

[9] Rivero, José Matías, Julián Grigera, Gustavo Rossi, Esteban Robles Luna, Francisco Montero, and Martin Gaedke. "Mockup-Driven Development: Providing agile support for Model-Driven Web Engineering." Information and Software Technology 56, no. 6 (2014): 670-687.

[10] Groenewegen, Danny M., Zef Hemel, Lennart CL Kats, and Eelco Visser. "Webdsl: a domain-specific language for dynamic web applications." In Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pp. 779-780. ACM, 2008.

[11] Fatolahi, Ali. "An Abstract Meta-model for Model Driven Development of Web Applications Targeting Multiple Platforms."PhD diss., University of Ottawa, 2012.

[12] Iseger, Martijn. "Domain-specific modeling for generative software development." *IT Architect* (2005).

[13] Kosar, Tomaž, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. "Comparing general-purpose and domain-specific languages: An empirical study." *Computer Science and Information Systems* 7, no. 2 (2010): 247-264.

[14] Selic, Bran. "The pragmatics of model-driven development." *IEEE software* 20, no. 5 (2003): 19-25.

[15] Cook, Steve. "Domain-specific modeling and model driven architecture." (2004).

[16] Kroiss, Christian, Nora Koch, and Alexander Knapp. Uwe4jsf: A model-driven generation approach for web applications. Springer Berlin Heidelberg, 2009.

[17] Kraus, Andreas, Alexander Knapp, and Nora Koch. "Model-Driven Generation of Web Applications in UWE." MDWE 261 (2007)

[18] Cuesta, Alejandro Gómez, Juan Carlos Granja, and Rory V. O'Connor. "A model driven architecture approach to web

development." In *Software and Data Technologies*, pp. 101-113. Springer Berlin Heidelberg, 2009.

[19] Corradini, F., D. Di Ruscio, and A. Pierantonio. "An ASM approach to Model Driven Development of Web applications.", 2004.

[20] Muller, Pierre-Alain, Philippe Studer, Frédéric Fondement, and Jean Bézivin. "Platform independent Web application modeling and development with Netsilon." Software & Systems Modeling 4, no. 4 (2005): 424-442.

[21] Perisic, Branko. "Model Driven Software Development–State of The Art and Perspectives." In *Invited Paper, 2014 INFOTEH International Conference, Jahorina*, pp. 19-23. 2014.

[22] Dejanovic, Igor, Gordana Milosavljevic, Branko Perišic, and Maja Tumbas. "A domain-specific language for defining static structure of database applications." Computer Science and Information Systems 7, no. 3 (2010): 409-440.

[23] Wimmer, Manuel, Nathalie Moreno, and Antonio Vallecillo. "Systematic evolution of WebML models by coupled transformations." In *Web Engineering*, pp. 185-199. Springer Berlin Heidelberg, 2012.

[24] Ceri, Stefano, Piero Fraternali, and Aldo Bongio. "Web Modeling Language (WebML): a modeling language for designing Web sites." Computer Networks 33, no. 1 (2000): 137-157.