Applying the Application-based Domain Modeling Approach to UML Structural Views

Arnon Sturm[1] and Iris Reinhartz-Berger[2]

[1]Department of Information System Engineering,
Ben-Gurion University of the Negev, Beer Sheva 84105, Israel
sturm@bgumail.bgu.ac.il
[2]Department of Management Information Systems,
University of Haifa, Haifa 31905, Israel
iris@mis.hevra.haifa.ac.il

Abstract. Being part of domain engineering, domain analysis enables identifying domains and capturing their ontologies in order to assist and guide system developers to design domain-specific applications. Several studies suggest using metamodeling techniques for modeling domains and their constraints. However, these techniques use different notions, and sometimes even different notations, for defining domains and their constraints and for specifying and designing the domain-specific applications. We propose an Application-based DOmain Modeling (ADOM) approach in which domains are treated as regular applications that need to be modeled before systems of those domains are specified and designed. This way, the domain models enforce static and dynamic constraints on their application models. The ADOM approach consists of three-layers and defines dependency and enforcement relations between these layers. In this paper we describe the ADOM architecture and validation rules focusing on applying them to UML static views, i.e., class, component, and deployment diagrams.

1 Introduction

Domain Engineering is a software engineering discipline concerned with building reusable assets and components in a specific domain [4], [5], [6]. We refer to a *domain* as a set of applications that use a common jargon for describing the concepts and problems in that domain. The purpose of domain engineering is to identify, model, construct, catalog, and disseminate a set of software artifacts that can be applied to existing and future software in a particular application domain [22]. As such, it is an important type of software reuse, verification, and validation [15], [16].

Similarly to software engineering, domain engineering includes three main activities: domain analysis, domain design, and domain implementation. *Domain analysis* identifies a domain and captures its ontology [28]. Hence, it should specify the basic elements of the domain, organize an understanding of the relationships among these elements, and represent this understanding in a useful way [4]. *Domain design* and *domain implementation* are concerned with mechanisms for translating

requirements into systems that are made up of components with the intent of reusing them to the highest extent possible.

Domain analysis is especially crucial because of two main reasons. First, analysis is one of the initial steps of the system development lifecycle. Avoiding syntactic and semantic mistakes at this stage (using domain analysis principles) helps to reduce development time and to improve product quality and reusability. Secondly, the core elements of a domain and the relations among them usually remain unchanged, while the technologies and implementation environments are in progressive improvement. Hence, domain analysis models usually remain valid for long periods.

Several methods and architectures have been developed to support domain analysis. Some of them rely on Unified Modeling Language (UML) [3] and metamodeling techniques [29], for example Catalysis [11]. Using standard notations and techniques has many advantages, including accessibility, reliability, and uniformity. However, most of the suggested works to domain analysis use different notions, and sometimes even different notations, for defining domains and their constraints and for specifying and designing applications, weakening the mentioned standardization benefits. Other techniques (e.g., [7], [26]) use UML extension mechanisms, more accurately stereotypes. Yet, this mechanism provides no formal definition of domain models.

In this paper we present the Application-based DOmain Modeling (ADOM) approach, which enables modeling domains as if they were regular applications. This approach enables the validation of domain-specific application models against their domain models. The ADOM approach consists of three layers: the application layer, the domain layer, and the (modeling) language layer. In the application layer, the required application is modeled as composed of classes, associations, collaborations, etc. In the domain layer, the domain elements and relations are modeled as if the domain itself is an application. Finally, the language layer includes metamodels of modeling languages (or methods). We also provide a set of validation rules between the different layers: the domain layer enforces constraints on the application layer, while the language layer enforces constraints on both the application and domain layers. Thus, the contribution of this paper is twofold. First, we provide an approach for modeling various aspects of domains and for validating application models against domain models. This approach uses a single, standard modeling language, UML, and a standard technique, metamodeling. Secondly, applying the ADOM approach to UML, we provide a formal framework for defining and constraining stereotypes.

The structure of the rest of the paper is as follows. Section 2 reviews existing works in domain analysis, dividing them into single-level and two-level domain analysis approaches. Section 3 introduces our three-level ADOM approach. In this section, we elaborate on applying ADOM to UML class, component, and deployment diagrams, exemplifying the approach stages and validation rules on a domain of Web applications and a Web-based glossary application. Finally, Section 4 summarizes the strengths of this approach and refers to future research plans.

2 Domain Analysis – Literature Review

Referring to domain analysis as an engineering approach, Argano [1] suggested that domain analysis should consist of conceptual analysis combined with infrastructure specification and implementation. Meekel et al. [16] suggested that in addition to its static definition, domain analysis may be conceived of as a development process which identifies a domain scope, builds a domain model, and validates that model. Since the domain keeps evolving as the product users within its scope generate new requirements, domain analysis in not a one-shot affair [5], [6]. Gomaa and Kerschberg [13] agreed that the domain model lifecycle is constantly evolving via an iterative process. Supporting this domain evolution concept, Drake and Ett [10] claimed that domain analysis gives rise to two concurrent, mutually dependent lifecycles that should be correlated: the fundamental system lifecycle and the domain lifecycle. Becker and Diaz-Herrera [2] proposed that the two concurrent streams are the design for reuse (i.e., the domain model) and the design with reuse (i.e., the application model). Following this spirit, the Model-Driven Architecture (MDA) [20], which originally aimed to separate business or application logic from underlying platform technology, observes that system functionality will gradually become more knowledge-based and capable of automatically discovering common properties of dissimilar domains. In other words, the aim of MDA is to eventually build systems in which considerable amount of domain knowledge is pushed up into higher abstraction levels. However, this vision is supported in a conceptual level and not (yet) in a practical one.

Several methods and techniques have been developed to support domain analysis. We classify them into two categories: single-level and two-level domain analysis approaches.

2.1 Single-Level Domain Analysis Approaches

In the single level domain analysis approaches, the domain engineer defines domain components, libraries, or architectures. The application designer reuses these domain artifacts and can change them in the application model. Meekel et al. [16], for example, propose a domain analysis process that is based on multiple views. They used Object Modeling Technique (OMT) [25] to produce a domain-specific framework and components. Gomaa and Kerschberg [13] suggest that a system specification will be derived by tailoring the domain model according to the features desired in the specific system.

Feature-Oriented Domain Analysis (FODA) [14] defines several activities to support domain analysis, including context definition, domain characterization, data analysis and modeling, and reusable architecture definition. A specific system makes use of the reusable architecture but not of the domain model.

Clauss [7] suggests two stereotypes for maintaining variability within a domain model: <<variation point>>, which indicates the variability of an element, and <<variant>>, which indicates the extension part. These stereotypes seems to be weak when defining a domain model and validating a specific application model of that domain.

Catalysis [11] is an approach to systematic business-driven development of component-based systems. It defines a process to help business users and software developers share a clear and precise vocabulary, design and specify component interfaces so they plug together readily, and reuse domain models, architectures, interfaces, code, etc. Catalysis introduced two types of mechanisms for separating different subject areas: package extension and package template. Package extension allows definitions of fragments of language to be developed separately and then merged to form complete languages. Package templates, on the other hand, allow patterns of language definition to be distilled and then applied consistently across the definition of languages and their components. Both package extension and package template mechanisms deal basically with classes and packages and enable renaming of the structural elements when reusing them in particular systems. In addition, that work does not address the application model validation against its domain model(s).

2.2 Two-Level Domain Analysis Approaches

In the two-level domain analysis approaches, connection is made between the domain model and its usage in the application model. Contrary to the single-level domain analysis approaches, the domain and application models in the two-level domain analysis approaches remain separate, while validation rules between them are defined. These validation rules enable avoiding syntactic and semantic mistakes during the initial stages of the application modeling, reducing development time and improving system quality. Petro et al. [21], for example, present a concept of building reusable repositories and architectures, which consist of correlated component classes, connections, constraints, and rationales. When modeling a specific system, the system model is validated with respect to the domain model in order to check that no constraint has been violated.

Schleicher and Westfechtel [26] discuss static metamodeling techniques in order to define domain specific extensions. They divide these extensions into descriptive stereotypes for expressing the elements of the underlying domain metamodel, restrictive stereotypes for attaching constraints to stereotyped model elements, regular metamodel extensions, and restrictive metamodel extensions. They mostly deal with packages and classes, but not with behavioral elements. Furthermore, the semantics and constraints of the stereotypes used in this work are expressed in a natural language, weakening the formality of this approach.

Gomma and Eonsuk-Shin [12] suggest a multiple view metamodeling method for software product lines. They solve model commonalty and variability problems within the product line domain by defining special stereotypes which are used in the use case, class, collaboration, statechart, and feature model views. These stereotypes are modeled in the metamodel level by class diagrams, while the relations among them are specified in Object Constraint Language (OCL) [030]. The main shortcoming of this method is in using a new dialect of UML for modeling the domain elements and constraints (e.g., adding alternating paths).

Morisio et al. [17] propose an extension to UML that includes a special stereotype indicating that a class may be altered within a specific system. The extension is demonstrated by applying it to UML class diagrams. The validation of an application

model with respect to its domain model entails checking whether a class appears in the application model along with its associate classes, but not if the class is correctly connected.

The Institute for Software Integrated Systems (ISIS) at Vanderbilt University suggested a metamodeling technique for building a domain-specific model using UML and OCL [18]. The application models are created from the domain metamodel, enabling validation of their consistency and integrity in terms of the domain analysis [9]. However, the domain models are specified using UML class diagrams and OCL, while the application models use other notations, conceding the benefits of applying a standard modeling language to the application models as well.

3 The Application-Based Domain Modeling (ADOM) Approach

Application models and domain models are similar in many aspects. An application model consists of classes and associations among them and it specifies a set of possible behaviors. Similarly, a domain model consists of core elements, static constraints, and dynamic relations. The main difference between these models is in their abstraction levels, i.e., domain models are more abstract than application models. Furthermore, domain models should be flexible in order to handle commonalities and differences of the applications within the domain.

The classical framework for metamodeling is based on an architecture with four abstraction layers [19]. The first layer is the information layer, which is comprised of the desired data. The model layer, which is the second layer, is comprised of the metadata that describes data in the information layer. The third metamodel layer is comprised of the descriptions that define the structure and semantics of metadata. Finally, the meta-metamodel layer consists of a description of the structure and semantics of meta-metadata (for example, metaclasses, metaattributes, etc.). Following this general architecture, we divide our Application-based DOmain Modeling (ADOM) approach into three layers: the application layer, the domain layer, and the (modeling) language layer. The application layer, which is equivalent to the model layer (M1), consists of models of particular systems, including their structure (scheme) and behavior. The domain layer, i.e., the metamodel layer (M2), consists of specifications of various domains. The language layer, which is equivalent to the meta-metamodel layer (M3), includes metamodels of modeling languages. The modeling languages may be graphical, textual, mathematical, etc. In addition, the ADOM approach explicitly enforces constraints among the different layers: the domain layer enforces constraints on the application layer, while the language layer enforces constraints on both the application and domain layers.

Figure 1 depicts the architecture of the ADOM approach. The application layer in this figure includes three examples of applications: *Amazon*, which is a Web-based book store, *eBay*, which is an auction site supported by agents, and *Kasbah*, which is a multi-agent electronic marketplace. Each one of these systems may have several models in different modeling languages. The domain layer in

Figure 1 includes two domains: *Web applications* and *multi agent systems*, while the language layer in this example includes only one modeling language, *UML*. Since

UML is the current standard (object-oriented) modeling language, we apply the ADOM approach to UML.

Figure 1 shows also the relations between the layers. The black arrows indicate constraint enforcement of the domain models on the application models, while the grey arrows indicate constraint enforcement of the language metamodels on the application and domain models.

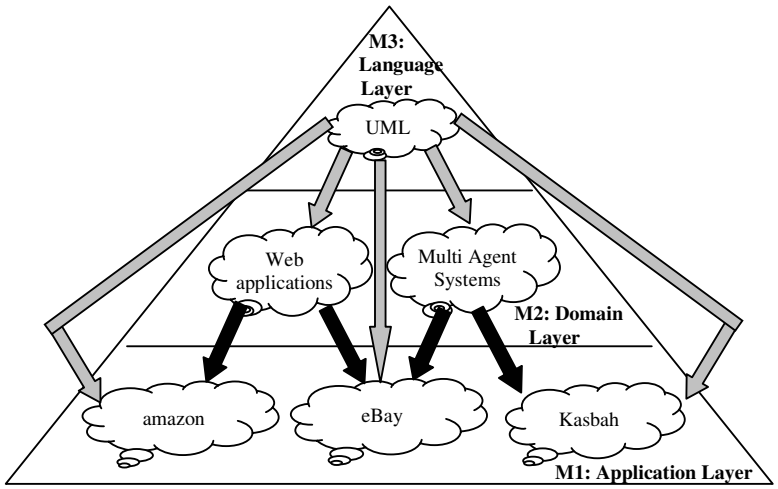

Figure 1. The Application-based DOmain Modeling (ADOM) architecture

The rest of this section elaborates on the domain and application layers, while the language layer is restricted to the UML metamodel [3] except of two minor changes:

1. A model element (e.g., attribute, operation, message, etc.) has an additional feature, called "multiplicity", which represents how many times the model element can appear in a particular system. This feature appears as <<min..max>> before a relevant domain element in a domain model, while <<1..1>> is the default (and, hence, does not appear).
2. A model element can have several stereotypes, which are separated by commas.

3.1 Applying UML Structural Views to the Domain Layer of the ADOM Approach

When referring to the static views of a domain, the domain engineer can use UML class, component, and deployment diagrams for specifying the domain elements and the (structural) constraints among them. In what follows, we demonstrate the ADOM approach on a part of the Web application domain as defined by Conallen [8]. Figure 2 cites a definition of a server page given by Conallen.

1. A server page represents a web page that has scripts which are executed by the server.
2. These scripts interact with resources on the server (databases, business logic, external systems, etc).
3. The object's operations represent the functions in the script, and its attributes represent the variables that are visible in the page's scope (accessible by all functions in the page).
4. Server pages can only have relationships with objects on the server.

Figure 2. A part of Conallen's specification for the Web application domain – A Server Page definition

As can be seen, the definition in Figure 2 includes logical and physical elements (classes, components, and nodes). Hence, modeling this particular domain element, a server page, requires UML class, component, and deployment diagrams.

Figure 3 is a partial class diagram that models the logical aspects of a server page: A **server page** is specified as a class the attributes of which are classified as **variables**. A **server page** may have any number (including 0) of **variables** which can be of any type recognized in UML. These constraints are modeled in the domain model as the attribute "**<<0..m>> variable: anyType**" of the **server page** class. Since these **variables** are visible only within the server page's scope (including its scripts), their scope is defined to be "package" in the domain model. The order of scopes (from the least restricted to the most restricted) is public, package, protected, and private. A scope of a model element defined in a domain model is the least restricted scope that this element can get in any application model of that domain[1]. In particular, a **variable** scope within an application model can be package, protected, or private.

Figure 3 also specifies that a **server page** may have any number of operations regardless of their signatures as indicated by "**<<0..m>> anyMethod (<<0..m>> anyParameter :anyType): anyType**" declaration. All the operations of a **server page** (as all the operations in this domain model) are defined as public in the domain model and, hence, their scopes are not limited in the application models, i.e., they can be public, package, protected, or private. A **server page** may have relations with any class (on the server, as will be constrained next), as indicated by the association between **server page** and **anyClass**. In addition, a **server page** may aggregate any number of **scripts**. A **script** has any number of operations regardless of their signatures, may have any relations with other scripts, as indicated by the self association labeled **anyRelation**, and interacts with any number of **resources** (on the server), as indicate by the dependency relation labeled "**interacts with**".

Similarly to scopes, the ADOM approach defines a precedence order between relations. The most general relation is an association, followed by a navigational association, an aggregation, a navigational aggregation, a composition, and a navigational composition. A relation specified in a domain model is the most general relation possible between the two model elements in any application model of that domain. Enforcing a specific relation type (e.g., aggregation) requires definition of a new type of an OCL constraint.

[1] Enforcing a specific scope on a model element (e.g., public) can be done by defining an OCL constraint.

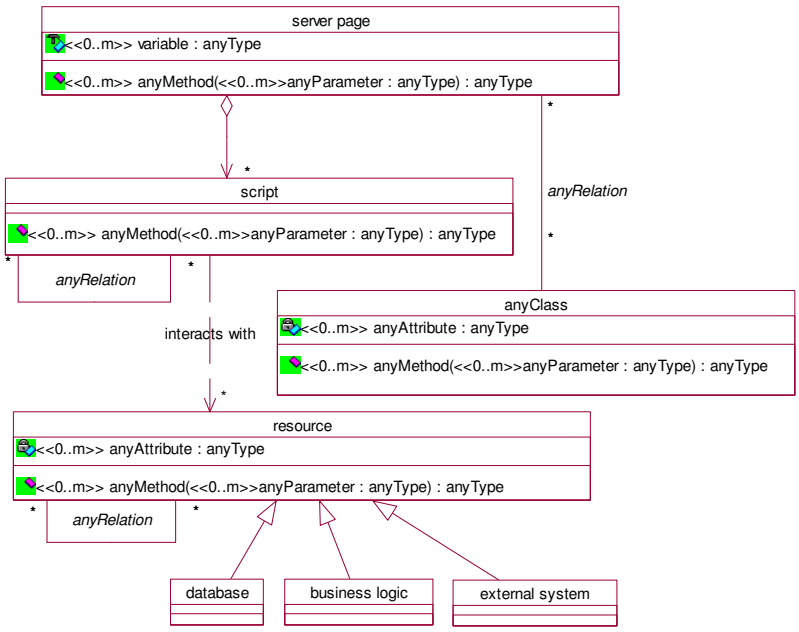

Figure 3. A partial class diagram of a Server Page in the Web application domain

Figure 3 does not limit the structure of a **resource** element, i.e., it may have any attributes, any operations, and any relations with other **resources**. However, this figure defines the hierarchy of resources: a **resource** is specialized into **database**, **business logic**, and **external system**, each of which is a special type of **resources**.

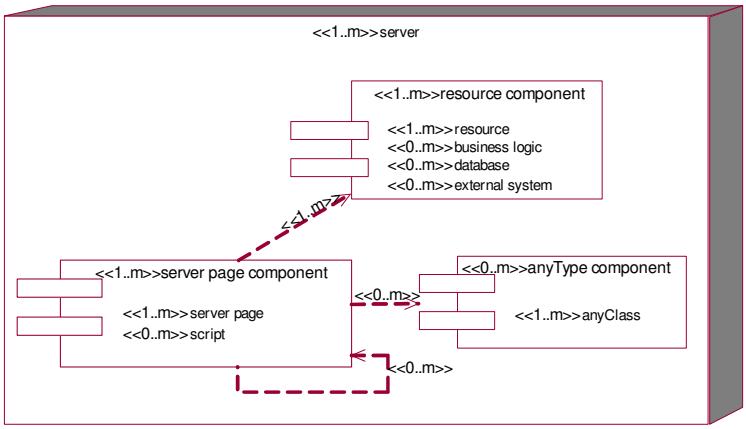

Figure 4. A partial merged component and deployment diagram describing the physical constraints on a Server Page in the Web application domain

Figure 4 presents a component diagram merged into a deployment diagram. The merged diagram expresses the physical constraints of the domain on server pages. The main domain node is a **server** from which at least one physical node exists as indicated by the multiplicity feature (**<<1..m>>**). The **server** hosts at least one **resource component** and at least one **server page component**. It may also host components of any type each of which implements at least one class (of any type). A **resource component** implements at least one **resource** class and may implement any number of other **resource** classes, i.e., **business logic**, **database**, and/or **external system**. A **server page component** implements at least one **server page** class and any number (including 0) of **script** classes. Figure 4 also defines dependency constraints among components: a **server page component** depends on at least one **resource component** and may depend on other components of any type, including other **server page components**.

3.2 Applying UML Structural Views to the Application Layer of the ADOM Approach

An application model uses a domain model as a validation template. All the constraints enforced by the domain model should be applied to any application model of that domain. In order to achieve this goal, any element in the application model is classified according to the elements declared in the domain model using UML stereotype mechanism. As defined in UML user guide [3], a *stereotype* is a kind of a model element whose information content and form are the same as the basic model element, but its meaning and usage are different. The ADOM approach requires that a model element in an application model will preserve the relations of its stereotypes in the relevant domain model(s).

Returning to our example of the Web application domain, we describe in this section a partial model of a Web-based glossary application (GLAP) in that domain. The GLAP system [8] provides an online version of a software development project's glossary of terms. The project's team members can access the database of terms, using a common Web browser. Team members may also update, add entries to the database, and remove entries from it, using the same browser interface. Figure 5 is a partial class diagram of the GLAP system. Following the server page definition in the Web application domain, shown in Figure 3, the GLAP system defines two types of server pages: **process search**, which uses the glossary API to search the glossary for words (or descriptions) that match a string, and **edit entry**, which builds an edit page for a specific entry in the glossary. **Process search** consists of **writeEntry** (classified as a script) and **getEntries** (also classified as a script). It also has four variables (attributes): **searchWord**, **searchDescription**, **nl** (the new line string), and **messageWord** (the word searched for, modified for use as a hyperlink parameter). All the variables of **process search** are of type string. The **Edit entry** server page consists of **getEntry** (classified as a script) and has three variables (**id**, **word**, and **description**). The **getEntries** script consists of a **getEntry** script. **writeEntry** and **getEntries** interact with the **glossary DB** (classified as a database), which in turn consists of many **glossary entries** (classified as "database" elements).

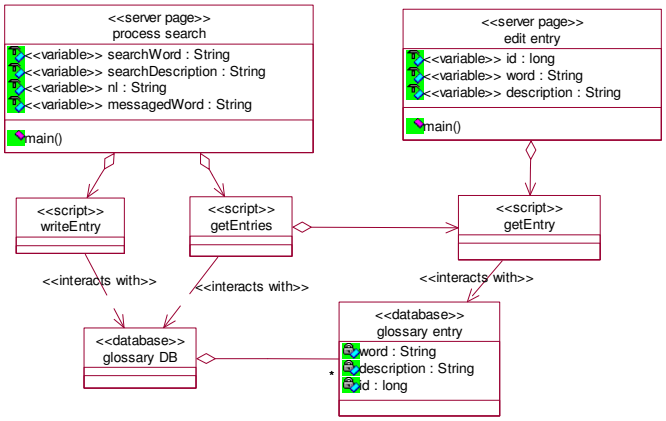

Figure 5. A partial class diagram of the GLAP system – A description of **process search** and **edit entry** server pages

Table 1. The Web application domain constraints and their fulfillment in the GLAP system model – comparing the class diagrams

Class	Feature Name	Feature Constraint	Allowed Feature Multiplicity	Actual Features
Server page	variable	Max scope: package	0..∞	• 4 package variables for process search • 3 package variables for edit entry
	general operation	Max scope: public	0..∞	• 1 public operation for each server page (process search & edit entry)
	relation to script	Type: navigational aggregation	0..∞	• 2 navigational aggregations for process search (writeEntry&getEntries) • 1 navigational aggregation for edit entry (getEntry)
	relation to any class	Type: association	0..∞	• 0 relation to other classes for both process search & edit entry
script	general attribute	None	0	• 0 attributes for both process search & edit entry
	general operation	Max scope: public	0..∞	• 0 public operations for each script
	relation to script	Type: association	0..∞	• 1 navigational aggregation for getEntries • 0 relations for the other scripts
	dependency to resource	None	0..∞	• 1 dependency relation for each script
resource	general attribute	Max scope: private	0..∞	• 0 attributes for glossary DB • 3 private attributes for glossary entry
	general operation	Max scope: public	0..∞	• 0 public operations for each resource
	relation to resource	Type: association	0..∞	• 1 aggregation for glossary DB • 0 relations for the other resources

The ADOM approach validates the structure of each application class and the relations among them using the domain model. Table 1 summarizes the domain constraints of the Web application elements, and how these are correctly fulfilled in the class diagram of the GLAP system. For each domain class, the table lists its features (in the "Feature Name" column), scope or relation type constraints (in the "Feature Constraint" column), and multiplicity constraints (in the "Allowed Feature Multiplicity" column). In addition, the table summaries the actual features of each class in the application model (in the "Actual Features" column). As can be seen, none of the constraints expressed in the domain model, shown in Figure 3, are violated by the application model, specified in Figure 5.

Figure 6 depicts the implementation view of the GLAP system. This diagram follows the guidelines of the Web application domain for components and their deployment as expressed in Figure 4. The ADOM approach validates the existence of the defined classes and their associations to components and nodes. It also validates the relationships among the various model elements and their multiplicities.

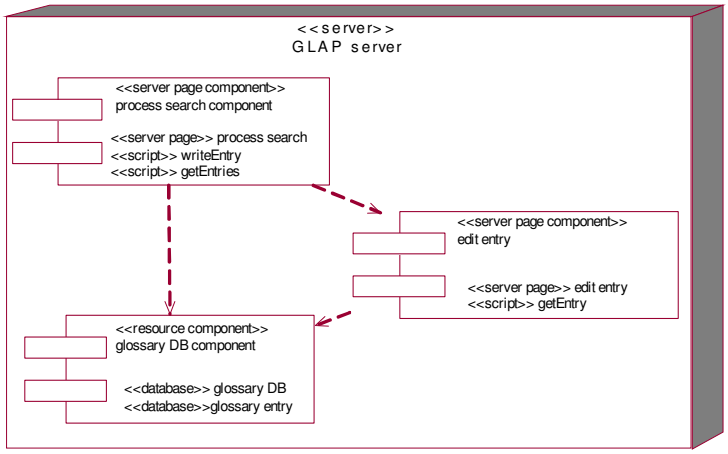

Figure 6. A partial merged component and deployment diagram of the GLAP system – Allocating the **process search** and **edit entry** server pages into components and nodes

Table 2 summarizes the physical constraints of the Web application domain (specified in Figure 4) and shows that none of them is violated by the GLAP system. For each component or node, the domain constraints (the "Feature Constraints" column) and the relevant features in the application model (the "Actual Features" column) are listed side-by-side.

4 Summary and Future Work

The Application-based DOmain Modeling (ADOM) approach enables domain engineers to define structural and behavioral constraints that are applicable to all the systems in a specific domain. When developing a system in ADOM, its domain (or domains) is first defined in order to enforce domain restrictions on particular systems.

Then, the application models are validated against their domain models in order to detect semantic errors in early development stages. These errors cannot be automatically found when using syntactic modeling language alone.

Table 2. The domain constraints and their fulfillment in the GLAP system model – comparing the component and deployment diagrams

Component/ Node	Feature Constraints	Actual Features
server	At least one node	One server called **GLAP server**
	Includes at least one resource component	One resource component called **glossary DB component**
	Includes at least one server page component	Two server page components called **process search component** and **edit entry component**
	Includes 0 or more components of any type	No other components
server page component	Includes at least one server page class	• The **process search component** includes one server page class (**process search**) • The **edit entry component** includes one server page class (**edit entry**)
	Includes 0 or more script classes	• The **process search component** includes two script classes (**writeEntry** and **getEntries**) • The **edit entry component** includes one script class (**getEntry**)
	Depends on at least one resource component	• The **process search component** depends on one resource component (**glossary DB component**) • The **edit entry component** depends on one resource component (**glossary DB component**)
	Depends on 0 or more server page components	• The **process search component** depends on one server page component (**edit entry component**) • The **edit entry component** does not depend on other server page components
	Depends on 0 or more other components of any type	• Neither the **process search component** nor the **edit entry component** depends on other components
resource component	Includes at least one resource class	The **glossary DB component** includes two resource classes of type database (**glossary DB** and **glossary entry**)
	Includes 0 or more business logic classes	The **glossary DB component** includes 0 business logic classes
	Includes 0 or more database classes	The **glossary DB component** includes two database classes (**glossary DB** and **glossary entry**)
	Includes 0 or more external system classes	The **glossary DB component** includes 0 external system classes

Two major techniques are usually used when applying UML to the domain analysis area: stereotypes and metamodeling. The main limitation of the stereotypes technique is the need to define the basic elements of a domain outside the model via a natural language, as was done, for example, by Conallen for the Web application

domain [8]. While using natural languages is more comprehensible to humans, it lacks the needed formality for defining domain elements, constraints, and usage contexts. The ADOM approach enables modeling the domain world in a (semi-) formal UML model. This model is used to validate domain-specific application models.

While applying a metamodeling technique, the basic elements of the domain and the relations among them are modeled. Usually, the domain and application models are specified using different notions (and even different notations). In the case of UML, the domain models are expressed using class diagrams, while the application models are expressed using various UML diagram types. This unreasonably limits the expressiveness of domain models. In the ADOM approach, the domain and application models are specified using the same notation and ontology. In other words, the ADOM approach enables specification of physical and behavioral constraints in the domain level (layer). Furthermore, keeping the same notation and ontology for the entire development team (which includes domain engineers and system engineers) improves collaboration during the development process.

In this paper, we applied the ADOM approach to UML static views. In [23], we have also applied the ADOM approach to UML interaction diagrams. In the future, we plan to develop a domain validation tool that will check a system model against its domain model and will even guide system developers according to given domain models. An experiment is planned to classify domain-specific modeling errors when using the ADOM approach and other domain analysis methods. This experiment will also check the adoption of several different domains within the same application utilizing the ADOM approach.

References

1. Arango, G. “Domain analysis: from art form to engineering discipline”, Proceedings of the Fifth International Workshop on Software Specification and Design, p.152-159, 1989.
2. Becker, M. and Diaz-Herrera, J. L. “Creating domain specific libraries: a methodology, design guidelines and an implementation”, Proceedings of the Third International Conference on Software Reuse, pp. 158-168, 1994.
3. Booch, G., Rumbaugh, J., and Jacobson, I. The Unified Modeling Language User Guide, Addison-Wesley, 1998.
4. Carnegie, M. “Domain Engineering: A Model-Based Approach”, Software Engineering Institute, http://www.sei.cmu.edu/domain-engineering/, 2002.
5. Champeaux, D. de, Lea, D., and Faure, P. Object-Oriented System Development, Addison Wesley, 1993.
6. Cleaveland, C. “Domain Engineering”, http://craigc.com/cs/de.html, 2002.
7. Clauss, M. "Generic Modeling using UML extensions for variability", Workshop on Domain Specific Visual Languages, Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), 2001.
8. Conallen, J., Building Web Applications with UML, First Edition, Addison-Wesley, Reading, MA, 1999.
9. Davis, J. “Model Integrated Computing: A Framework for Creating Domain Specific Design Environments”, The Sixth World Multiconference on Systems, Cybernetics, and Informatics (SCI), 2002.

10. Drake, R. and Ett, W. “Reuse: the two concurrent life cycles paradigm”, Proceedings of the conference on TRI-ADA '90, p.208-221, 1990.
11. D’Souza, D. F., Wills, A.C. Objects, Components, and Frameworks with UML – The CatalysisSM Approach. Addison-Wesley, 1999.
12. Gomma, H. and Eonsuk-Shin, M. "Multiple-View Meta-Modeling of Software Product Lines", Proceedings of the Eighth IEEE International Confrerence on Engineering of Complex Computer Systems, 2002.
13. Gomaa, E. and Kerschberg, L. "Domain Modeling for Software Reuse and Evolution", Proceedings of Computer Assisted Software Engineering Workshop (CASE 95), 1995.
14. Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A.,”Feature-Oriented Domain Analysis (FODA) Feasibility Study”, CMU/SEI-90-TR-021 ADA235785, 1990.
15. Massonet, P., Deville, Y., and Neve, C. “From AOSE Methodology to Agent Implementation”, Proceedings of the First Joint Conference on Autonomous Agents and Multi-Agents Systems, pp. 27-34, 2002.
16. Meekel, J., Horton, T. B., France, R. B., Mellone, C., and Dalvi, S. “From domain models to architecture frameworks”, Proceedings of the 1997 symposium on Software reusability, pp. 75-80, 1997.
17. Morisio, M., Travassos, G. H., and Stark, M. “Extending UML to Support Domain Analysis”, Proceedings of the Fifth IEEE International Conference on Automated Software Engineering, pp. 321-324, 2000.
18. Nordstrom, G., Sztipanovits, J., Karsai, G., and Ledeczi, A. “Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments”, Proceedings of the IEEE Sixth Symposium on Engineering Computer-Based Systems (ECBS), pp. 68-74, 1999.
19. OMG, “Meta-Object Facility (MOF™)”, version 1.4, 2003, http://www.omg.org/docs/formal/02-04-03.pdf
20. OMG, "Model Driven Architecture (MDA™)", version 1.0.1, 2003, http://www.omg.org/docs/omg/03-06-01.pdf
21. Petro, J. J., Peterson, A. S., and Ruby, W. F. “In-Transit Visibility Modernization Domain Modeling Report Comprehensive Approach to Reusable Defense Software” (STARS-VC-H002a/001/00). Fairmont, WV: Comprehensive Approach to Reusable Defense Software, 1995.
22. Pressman, R.S. "Software Engineering: A Practitioner's Approach", 5th Edition, New York: McGraw-Hill, 2000.
23. Reinhartz-Berger, I. and Sturm, A. Behavioral Domain Analysis – The Application-based Domain Modeling Approach, accepted to UML'2004, 2004.
24. Rix, M. “Case study of a Successful Firmware Reuse Program”, HP Software Productivity Conference, 1992.
25. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. Object-Oriented Modeling and Design, Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, 1991
26. Schleicher, A. and Westfechtel, B. “Beyond Stereotyping: Metamodeling Approaches for the UML”, Proceedings of the Thirty Fourth Annual Hawaii International Conference on System Sciences, pp. 1243-1252, 2001.
27. Troy R. “Software Re-Use”, Presentation at ObjectWorld conference, 1993.
28. Valerio, A., Giancarlo, S., and Massimo, F. “Domain analysis and framework-based software development”, ACM SIGAPP Applied Computing Review, 5 (2), 1997.
29. Van Gigch, J. P. System Design Modeling and Metamodeling. Kluwer Academic Publishers, 1991.
30. Warmer, J. and Kleppe, A. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1998.