Behavioral Domain Analysis – The Application-based Domain Modeling Approach

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

[1]Department of Management Information Systems,
University of Haifa, Haifa 31905, Israel
iris@mis.hevra.haifa.ac.il
[2]Department of Information System Engineering,
Ben-Gurion University of the Negev, Beer Sheva 84105, Israel
sturm@bgumail.bgu.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. Domain analysis should consider commonalities and differences of systems in a domain, organize an understanding of the relationships between the various elements in that domain, and represent this understanding in a formal, yet easy to use, way. Several studies suggest using metamodeling techniques for modeling domains and their constraints. These metamodels are basically structural and present static constraints only. We propose an Application-based DOmain Modeling (ADOM) approach for domain analysis. This approach treats a domain as a regular application that needs to be modeled before systems of that domain are specified and designed. This way, the domain structure and behavior are modeled, enforcing static and dynamic constraints on the relevant application models. The ADOM approach consists of three-layers: the language layer handles modeling language ontologies and their constraints, the domain layer holds the building elements of domains and the relations among them, and the application layer consists of domain-specific systems. Furthermore, the ADOM approach defines dependency and enforcement relations between these layers. In this paper we focus on applying the ADOM approach to UML and especially to its class and sequence 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 the commonalities and differences of the domain applications [21]. As such, it is an important type of software reuse, verification, and validation [14, 15].

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 [24]. Hence, it should identify 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 continuous improvement. Hence, domain analysis models usually remain valid for long periods.

Several methods and architectures have been developed to support domain analysis. Most of them rely on Unified Modeling Language (UML) [3] and metamodeling techniques [25]. However, most of these works are concerned with domain structural elements and relations, neglecting domain behavioral constraints. Other techniques for domain analysis (e.g., [7, 23]) use UML extension mechanisms, or 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 specification of both structural and behavioral aspects of any application within a domain. 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 verification and 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 the structure and behavior of domains and for validating application models against domain models. Secondly, basing the ADOM approach on UML, we provide a formal framework for defining and constraining stereotypes.

The structure of the rest of the paper is as follows. In Section 2 we review existing works in domain analysis, dividing them into single-level and two-level domain analysis approaches. Section 3 introduces our three-level ADOM approach, elaborating on its applicability to UML class and sequence diagrams. We exemplify the approach stages and validation constraints on a domain of sensor-based machines and an elevator system. Finally, Section 4 summarizes the strengths of this approach and refer to the future research plan.

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. [15] 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 repository (model), and validates that model. Since the domain keeps evolving as new requirements are introduced, domain analysis in not a one-shot affair [5, 6]. Gomaa and Kerschberg [12] agreed that the domain model lifecycle is constantly evolving via an iterative process. Supporting this domain evolution concept, Drake and Ett [9] 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) [19], 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 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. [15], for example, propose a domain analysis process that is based on multiple views. They used Object Modeling Technique (OMT) [22] notations to produce a domain-specific framework and components. Gomaa and Kerschberg [12] 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) [13] 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 [10] 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 language fragments 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 enable renaming of the structural elements when reusing them in particular systems.

2.2 Two-Level Domain Analysis Approaches

In the two-level domain analysis approaches, connection is made between the domain model and its use 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. [20], 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 [23] 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.

Gomma and Eonsuk-Shin [11] suggest a multiple view meta-modeling method for software product lines. They solve model commonalty and variability problems within a specific domain (the product line) 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 modeled in Object Constraint Language (OCL) [26]. The main shortcoming of this method is in changing the core UML notation (e.g., by adding alternating paths) and in using structural metamodels that capture only the static constraints of the extension.

Morisio et al. [16] 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 the UML class diagram. 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 [17] using UML and OCL. The application models are created from the domain metamodel, enabling validation of their consistency and integrity in terms of the domain analysis [8]. While the ISIS project provides an environment for performing the tasks of building domain metamodels and domain application models, it is not clear how the domain dynamics is specified and validated, as the metamodel technique is basically static.

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: 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 [18]. 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 the 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 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 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 in this work.

Figure 1 also shows 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.

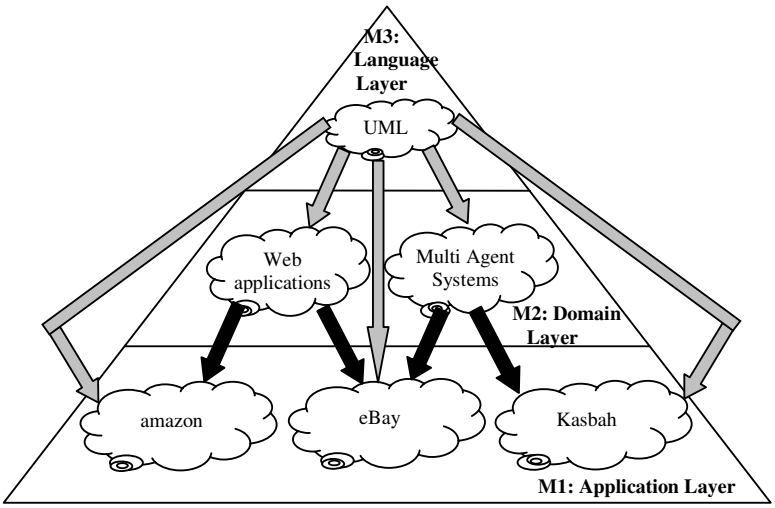

Figure 1. The Application-based DOmain Modeling (ADOM) architecture

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

1. A model element (e.g., attribute, operation, message, etc.) has an additional feature, "multiplicity", which represents how many times the model element can appear in a particular context. 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 The Domain Layer of the ADOM Approach

In the domain layer, the domain engineer specifies the structural and behavioral elements of a specific domain. This is done using UML structural and behavioral views. Figures 2-4 are a (partial) UML model that describes a domain of sensor-based machines. Figure 2 is the class diagram of the domain, which includes the basic elements of the domain and the static relations among them. The top-level class is **machine** which may have any number (including 0) of attributes of any type, as the

"**<<0..m>> anyAttribute: anyType**" declaration indicates. The scope of these attributes in the application models must be private. The order of scopes (from the least resticted 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].

Figure 2. A UML class diagram describing a domain of sensor-based machines

Figure 2 also specifies that **machine** has one or two operations of type **initialize** and at least one operation of type **work**. The **initialize** operations get no parameters and their return type is void, while the **work** operations can get any number of parameters of any type and their return types are not specified in the domain layer, as indicated by the "**: anyType**" declaration at the end of the operation signature. In addition to these types of operations, **machine** may have any number (including 0) of operations (as the reserved word **anyMethod** indicates) with any type of signature. All the operations of a **machine** (as all the operations in this domain model) are defined as public in the domain model and, hence, their scopes are not restricted in the domain-specific application models, i.e., they can be public, package, protected, or private.

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

A **machine** consists of one **controller**, which is composed of any number of **sensors** and any number of **data** items. A **controller** may have any attributes of any type, exactly one **start** operation with one **mode** parameter of type integer and any number of additional parameters of any type, at least one **work** operation, and other possible operations. A **sensor** should have at least one **test** operation and at least one **result** operation, in addition to other attributes and operations. A **test** operation gets any number of parameters of any type and returns a Boolean value, while a **result** operation may have any signature. Finally, **data** should have at least one **find** operation with any signature. Figure 2 also shows that an **operator** (a system actor) has a relation with the class **machine**, labeled "**operates**".

Using UML sequence diagram notation, Figure 3 describes a possible scenario of initialization in the domain of sensor-based machines. According to this scenario, the **operator** invokes an **initialize** operation of a **machine**. As a consequence, the **machine** invokes a **start** operation of its **controller**, which immediately invokes at least one **test** operation of the relevant **sensors**. All these messages are synchronous, enforcing that the corresponding messages in the application models will be synchronous too. Specifying a message to be asynchronous in a domain model enables defining this message as either synchronous or asynchronous in an application model of that domain[2].

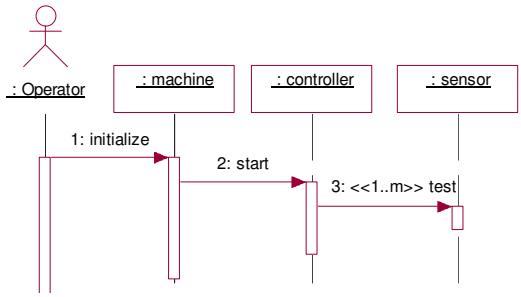

Figure 3. A UML sequence diagram describing an initialization operation in the domain of sensor-based machines

The ADOM approach also supports conditional messages and loops. Figure 4 describes a possible scenario of the machine work:

1. The operator invokes a work operation of a machine.
2. The machine invokes a work operation of its controller.
3. The controller invokes at least one result operation of its sensors. This invocation can be conditional and/or run in a loop, as indicated by [anyCondition] and [anyLoop], respectively.
4. The controller invokes zero or more find operations of its data items. This invocation can run in a loop, but it is not conditional.
5. Finally, the controller invokes at least one work operation of itself. This invocation can be conditional and/or run in a loop.
6. Steps 3-5 can run in a loop, as indicated by the <<1..m>> anyLoop note enclosing steps 3-5.

---

[2] Enforcing a message to be only asynchronous can be done by introducing a new type of an OCL constraint.

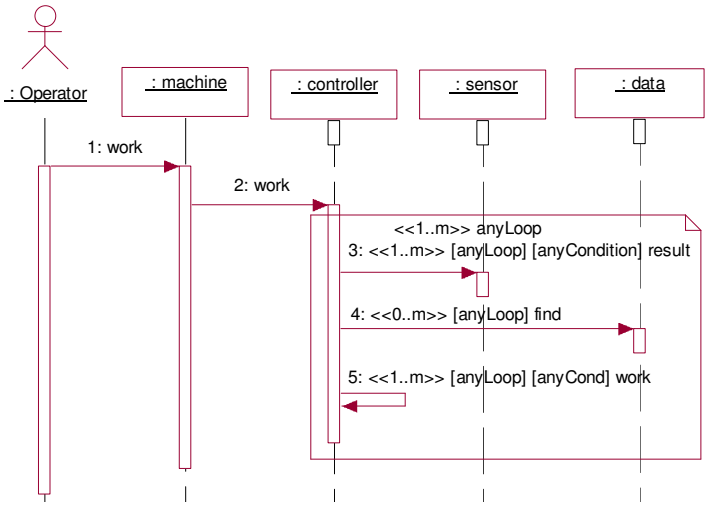

Figure 4. A UML sequence diagram describing a machine work operation in the domain of sensor-based machines

3.2 The Application Layer of the ADOM Approach

An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain model should be applied in 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 built-in stereotype mechanism. As defined in the UML user guide [3], a *stereotype* is a kind of a model element whose information content and form are the same as a 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 static and behavioral relations of its stereotypes in the relevant domain model(s).

Returning to our example of the sensor-based machine domain, we describe in this section a model of an elevator system in that domain. Figure 5 is the class diagram of the elevator system. Following the class hierarchy of the domain, shown in Figure 2, the **elevator** (classified as a machine) consists of an **elevator controller** (classified as a controller). The **elevator controller** consists of two types of sensors, a **door sensor** and a **button sensor**, and **music** items (classified as data items). Two types of operators are defined: **user** who **uses** the **elevator** and **technical person** who **fixes** the **elevator**.

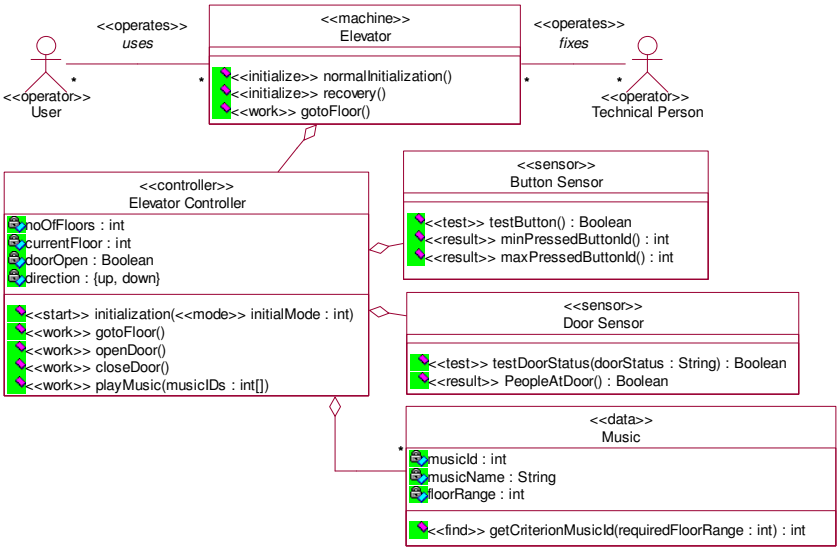

Figure 5. A UML class diagram describing an elevator system in the domain of sensor-based machines

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

Class	Feature	Feature Constraint	Allowed Feature Multiplicity	Actual Feature Multiplicity
Machine	General attribute	Private	0..∞	0
	Initialize operation	No parameters No return type	1..2	2
	Work operation	---	1..∞	1
	General operation	---	0..∞	0
Controller	General attribute	Private	0..∞	4
	Start operation	Mode:int parameter No return type	1..1	1
	Work operation	---	1..∞	4
	General operation	---	0..∞	0
Sensor	General attribute	Private	0..∞	0
	Test operation	Boolean return type	1.. ∞	1
	Result operation	---	1..∞	1* | 2**
	General operation	---	0..∞	0
Data	General attribute	Private	0..∞	3
	Find operation	---	1..∞	1
	General operation	---	0..∞	0

* for door sensor, ** for button sensor

Table 1 summarizes the domain constraints of the four basic domain elements, machine, controller, sensor, and data, and how they are correctly fulfilled in the class diagram of the elevator system. As can be seen, none of the constraints expressed in Figure 2 are violated by the application model shown in Figure 5.

Figure 6 includes two possible scenarios for initializing the elevator system: Figure 6(a) describes a normal initialization of the elevator, while Figure 6(b) describes an initialization operation that occurs after recovery (due to electrical power interruption, for example). These scenarios follow the guidelines of an initialization operation in the sensor-based machine domain as expressed in Figure 3. In these scenarios, no conditions and no loops are defined; hence, the ADOM approach validates only the order of the stereotyped operations and their multiplicities.

Table 2 summaries the constraints on an initialization operation in the domain of sensor-based machines and how the various constraints are fulfilled by the sequence diagrams of the elevator system specified in Figure 6. Note that the validation is done with respect to the stereotype of a model element in the application layer, which is itself an element in the domain layer.

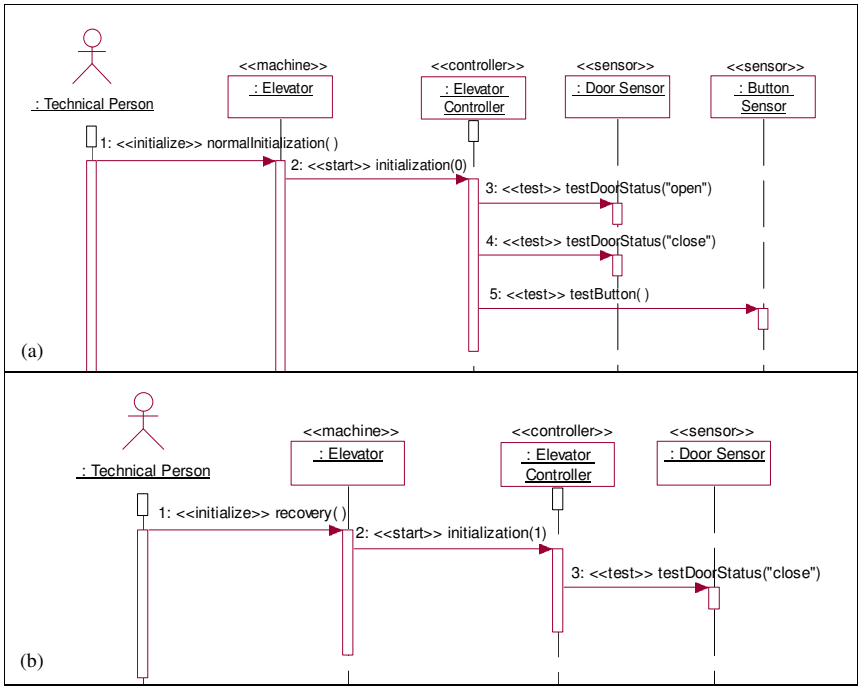

Figure 6. UML sequence diagrams describing initialization operations.
(a) Normal initialization. (b) Initialization after recovery.

Table 2. The domain constraints and their fulfillment in the elevator system model – comparing the sequence diagrams of the initialization operations

Layer/ Sequence	**Operation**	**Source**	**Destination**	**Mult.**
Domain/ Initialization	initialize	operator	machine	1..1
Application/ normal init.	normalInitialization	technical person	elevator	1
Application/ recovery	recovery	technical person	elevator	1
Domain/ Initialization	Start	machine	controller	1..1
Application/ normal init.	initialization(0)	elevator	elevator controller	1
Application/ recovery	initialization(1)	elevator	elevator controller	1
Domain/ Initialization	test	controller	sensor	1..∞
Application/ normal init.	testDoorStatus("open")	elevator controller	door sensor	3
	testDoorStatus("close")	elevator controller	door sensor	
	testButton()	elevator controller	button sensor	
Application/ recovery	testDoorStatus("close")	elevator controller	door sensor	1

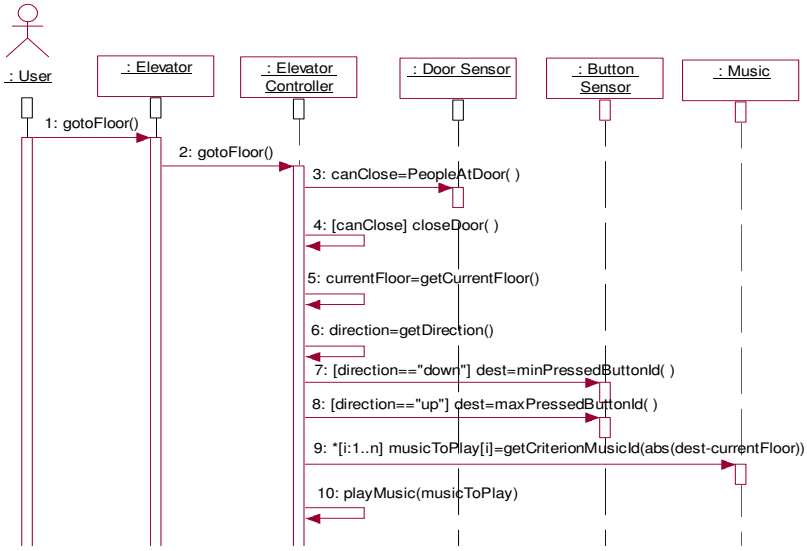

Figure 7. A UML sequence diagram describing an elevator work operation called gotoFloor

Figure 7 describes a possible work scenario of the elevator system. According to this scenario, called "go to floor", after the **user** pressed a button, the **elevator** (through its **controller** and **door sensor**) checks if the door can be closed, close it (if so), check the travel length, and choose **music** accordingly[3].

Table 3. The domain constraints and their fulfillment in the elevator system model – comparing the sequence diagrams of the work operation

Layer/ Sequence	Operation	Cond./ Loops	Source	Destination	Mult.
Domain/ work	work	None	operator	machine	1..1
Application/ go to floor	gotoFloor()	None	user	elevator	1
Domain/ work	work	None	machine	controller	1..1
Application/ go to floor	gotoFloor()	None	elevator	elevator controller	1
Domain/ work	result	Any	controller	sensor	1..∞
Application/ go to floor	peopleAtDoor()	None	elevator controller	door sensor	1
Domain/ work	find	AnyLoop	controller	data	0..∞
Application/ go to floor					0
Domain/ work	work	Any	controller	controller	1..∞
Application/ go to floor	closeDoor()	canClose	elevator controller	elevator controller	1
Domain/ work	result	Any	controller	sensor	1..∞
Application/ go to floor	minPressedButtonId()	direction= "down"	elevator controller	button sensor	2
	maxPressedButtonId()	direction= "up"	elevator controller	button sensor	
Domain/ work	find	AnyLoop	controller	data	0..∞
Application/ go to floor	getCriterionMusicId(abs(dest-currentFloor))	*[i:1..n]	elevator controller	music	1
Domain/ work	work	Any	controller	controller	1..∞
Application/ go to floor	playMusic(musicToPlay)	None	elevator controller	elevator controller	1

Table 3 summarizes the fulfillment of the constraints enforced by the domain sequence diagram expressed in Figure 4 on this particular scenario. Note that in Figure 7 there are several calls for operations that were not explicitly specified within

[3] For simplicity, the stereotypes of the messages and the objects in Figure 7 are suppressed.

the domain sequence diagram in Figure 4, for example, **getDirection()** and **getCurrentFloor()**. These operations are categorized as the additional methods allowed by the domain class diagram (Figure 2) for the **controller** class. They can be invoked in any stage in a sequence diagram, since they are basic operations (getter methods in this case).

4 Summary and Future Work

In this paper we introduced the Application-based DOmain Modeling (ADOM) approach which enables domain engineers to define structural and behavioral constraints that are applicable to all the systems within a specific domain. When developing a system, its domain (or domains) should be defined in order to enforce domain restrictions on the system. The advantages of this approach are twofold. First, it validates system models against their domain models in order to detect semantic errors in early development stages. These errors cannot be automatically found when using syntactic modeling languages alone. Secondly, the usage of the same modeling language for the application and domain layers reduces the ontological gap and the communication problems between the different stakeholders in the system development process. Applying ADOM specifically to UML, the standard object-oriented modeling language, also benefits from the maturity of the UML environment, including its CASE tools. Furthermore, this combination of ADOM and UML establishes a formal framework for defining and constraining stereotypes in UML. However, the separation of the ADOM architecture into three layers (application, domain, and language) makes it flexible enough to be applied to other languages as well.

Further work is planned to develop a domain verification 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 also 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 to the same application following 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. Davis, J. “Model Integrated Computing: A Framework for Creating Domain Specific Design Environments”, The Sixth World Multiconference on Systems, Cybernetics, and Informatics (SCI), 2002.
9. Drake, R. and Ett, W. “Reuse: the two concurrent life cycles paradigm”, Proceedings of the conference on TRI-ADA '90, p.208-221, 1990.
10. D’Souza, D. F., Wills, A.C. Objects, Components, and Frameworks with UML – The CatalysisSM Approach. Addison-Wesley, 1999.
11. 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.
12. Gomaa, E. and Kerschberg, L. "Domain Modeling for Software Reuse and Evolution", Proceedings of Computer Assisted Software Engineering Workshop (CASE 95), 1995.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. OMG, “Meta-Object Facility (MOF™)”, version 1.4, 2003, http://www.omg.org/docs/formal/02-04-03.pdf
19. OMG, "Model Driven Architecture (MDA™)", version 1.0.1, 2003, http://www.omg.org/docs/omg/03-06-01.pdf
20. 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).
21. Pressman, R.S. "Software Engineering: A Practitioner's Approach", 5th Edition, New York: McGraw-Hill, 2000.
22. 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
23. 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.
24. Valerio, A., Succi, G., and Fenaroli M. “Domain analysis and framework-based software development”, ACM SIGAPP Applied Computing Review, 5 (2), 1997.
25. Van Gigch, J. P. System Design Modeling and Metamodeling. Kluwer Academic Publishers, 1991.
26. Warmer, J. and Kleppe, A. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1998.