Utilizing domain models for application design and validation

Iris Reinhartz-Berger^a,c, Arnon Sturm^b,1

^aDepartment of Management Information Systems, University of Haifa, Carmel Mountain, Haifa 31905, Israel
^bDepartment of Information Systems Engineering, Ben-Gurion University of the Negev, Beer Sheva 84105, Israel

ABSTRACT

Domain analysis enables identifying families of applications and capturing their terminology in order to assist and guide system developers to design valid applications in the domain. One major way of carrying out the domain analysis is modeling. Several studies suggest using metamodeling techniques, feature-oriented approaches, or architectural-based methods for modeling domains and specifying applications in those domains. However, these methods mainly focus on representing the domain knowledge, providing insufficient guidelines (if any) for creating application models that satisfy the domain rules and constraints. In particular, validation of the application models which include application-specific knowledge is insufficiently dealt. In order to fill these lacks, we propose a general approach, called Application-based Domain Modeling (ADOM), which enables specifying domains and applications similarly, (re)using domain knowledge in application models, and validating the application models against the relevant domain models. In this paper we present the ADOM approach, demonstrating its application to UML 2.0 class and sequence diagrams.

1. Introduction

Domain Engineering, also known as software product line engineering, is concerned with building reusable assets, such as specification sets, patterns, and components, in a specific domain [6,8,33]. A domain in this context is an area of knowledge that uses common concepts for describing requirements, problems, capabilities, and solutions. The purpose of domain engineering is to identify, model, construct, catalog, and disseminate the commonalities and differences of applications in a domain [40]. Domain engineering methods and domain-specific languages (DSLs) receive nowadays special attention from communities that deal with reuse, validation, and knowledge representation [1,29,30]. One of the reasons for this tendency might be the increasing variability of information and software systems and the need to acquire expertise in different, evolving domains.

A core activity in domain engineering is domain analysis, which identifies a domain and captures its ontology [52]. 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 [8]. Departing from “regular” reuse, domain analysis is expected to provide some kind of support to the specification of variability within the domain and not just to the commonality. Several methods and architectures have been developed to support the domain analysis through modeling. However, these mainly focus on the specification and representation of the domain knowledge and lack in guiding and validating the reuse of domain knowledge in particular application models. These lacks are especially emphasized when application-specific elements are introduced to fully specify the application at hand.

The Application-based Domain Modeling (ADOM) approach aims at addressing these lacks by handling both reuse and validation according to the domain knowledge. The main observation of ADOM is that domain and application models have much in common: they both define concepts, relations, and structural and behavioral constraints. However, they differ in their abstraction and in the required flexibility levels: domain models are more abstract than application models and they should be more flexible in order to handle commonalities and differences of the applications within their scope (domain).

ADOM’s framework consists of three layers, language, domain, and application, each of which includes models in the corresponding abstraction level. Furthermore, the models in each layer make use of the models in the more general layer: the domain models are expressed using the modeling constructs specified in the language layer, while the designated applications are modeled using the knowledge and constraints presented in the domain layer and the modeling constructs specified in the lan-
guage layer. In this way, ADOM can be used with different modeling languages. However, when adopting ADOM with a specific modeling language, this language is used in both application and domain layers, easing the tasks of application design and validation by employing the same constructs in both application and domain layers. In this paper, we concentrate on a specific dialect of ADOM, called ADOM–UML, in which UML 2.0 class and sequence diagrams [38] are used as the modeling language. We chose these diagrams due to their popularity and familiarity in the software engineering community and due to their support for both structure and behavior modeling.

The contribution of this work is in providing a general approach for creating and validating complete application models according to domain models. Different from other domain analysis approaches, ADOM provides a powerful validation mechanism that prevents application developers from violating domain constraints after or while (re)using the domain artifacts in the context of particular applications. This mechanism also handles application-specific elements that can be added in various places of the specifications in order to fulfill the particular application requirements. Furthermore, in the context of ADOM–UML, we provide a method for defining and constraining domain-specific extensions (stereotypes) in UML, extending the expressiveness and usefulness of UML profiles.

The structure of the rest of the paper is as follows. Section 2 generally describes and demonstrates ADOM–UML and its principles. The examples used in this section are of two process control systems: a home climate control application and a water level control application. Section 3 formalizes ADOM–UML, referring to its reuse guidance and validation capabilities, as well as to its supporting CASE tool, while Section 4 refers to related work and discusses ADOM’s benefits and limitations. Finally, Section 5 concludes and refers to future research plans.

2. ADOM–UML in a nutshell

As noted, ADOM in general, and ADOM–UML in particular, is concerned with three layers. The language layer comprises metamodels and specifications of modeling languages, UML 2.0 in the case of ADOM–UML. The application layer consists of models of particular systems, including their structure (scheme) and behavior. Finally, the domain layer consists of specifications of various application families (domains), such as web applications, multi-agent systems, and process control systems. These specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain. The approach also explicitly enforces constraints among the different layers: the domain layer enforces constraints on the application layer, while the language layer enforces constraints on both domain and application layers. In particular, both application and domain models in ADOM–UML are expressed using UML 2.0.

The domain model in ADOM–UML is constructed based on the knowledge gained from developing (previous) applications in the domain and studying the relevant literature. The approach provides means for specifying the generic knowledge (know-how) of a domain in terms of common elements and allowed variants. UML stereotypes mechanism is used in the domain layer in order to denote the multiplicity variability of the different domain model elements: each domain model element is associated with a multiplicity indicator, which is a stereotype defined in the language layer and attached to the Element meta-class in the Kernel package of the UML metamodel. This means that multiplicity stereotypes (indicators) can be associated to all UML elements, including classes, attributes, methods, parameters, associations, messages, and objects.\(^2\) This kind of stereotypes, which aims at representing how many times a model element of this type can appear in a specific context, has two associated tagged values, min and max, which define the lowest and the upper most multiplicity boundaries, respectively. For clarity purposes, we defined four commonly used multiplicity groups on top of this stereotype, <<optional many>>, <<optional single>>, <<mandatory many>>, and <<mandatory single>>, whose meanings are summarized in Table 1. Note that other multiplicity constraints can be specified using the general <<multiplicity>> stereotype with its associated tagged values.

Besides multiplicity-related constraints, ADOM employs the modeling language expressiveness and semantics (UML in case of ADOM–UML) in order to specify additional constraints and dependencies in the domain layer. These constraints and dependencies are expressed in Object Constraint Language (OCL), which is part of UML [38].

Having a domain model, it is used as a reference for guiding the development of the target application model. By guiding we refer to the set of operations that can be performed on models in the domain layer in order to create models in the application layer. This guidance includes indications on mandatory and optional elements, their structure, and behavior, the ability to have different variants of a domain element in the application layer, the ability to customize and adapt a domain element to the application at hand, and so on.

The relations between a generic (domain) element and its specific (application) counterparts are maintained by UML stereotypes mechanism, such that a domain element serves as a stereotype of an application element. However, a domain element may serve as a stereotype of an application element only if their meta-classes in the language layer are the same. Domain classes, for example, can serve as stereotypes of application classes, domain attributes and operations can respectively serve as stereotypes of application attributes and operations, domain associations and messages can respectively serve as stereotypes of application associations and messages, and so on. Some optional generic elements may be omitted and may not be included in the application model and some new specific elements may be inserted to the specific application model (these are termed application-specific elements and are not stereotyped in the application layer). Nevertheless, the domain knowledge embedded in the generic model must be maintained in the specific one.

As an example to a domain, consider process control systems (PCS). Applications in this domain monitor and control the values of certain variables through a set of components that work together to achieve a common objective or purpose [13]. Application areas within this domain include engineering and industrial control systems, control systems in the human body, and financial derivation-tracking products. Figs. 1 and 2 present an ADOM–UML model of the PCS domain expressed in terms of UML 2.0 class and sequence diagrams, respectively.

According to this model, all applications in the domain should define their sensors, controlled elements and values, and controlled devices, as well as the relationships among them. In addition, the applications should specify scenarios of monitoring the controlled values and activating the controlled devices accordingly. In particular, each application in the PCS domain has to have ex-

\(^2\) We chose to use UML stereotype mechanism for defining multiplicity constraints over UML multiplicity mechanism since the latter is applicable only for association ends and attributes and we wish to express multiplicity constraints on all the model elements. Furthermore, we decided not to use UML multiplicity mechanism even when applicable in order to maintain uniformity and to avoid confusion between its different meanings in the application layer (where multiplicity refers to instances) and the domain layer (where multiplicity refers to concepts or roles).
actly one controller, which exhibits at least one operation for monitoring and acting and possibly several notification operations. A PCS application should also have at least one controlled element and at least one controlled value class. Each Controlled Element class exhibits at least one attribute specifying its identity, zero or more enumerated attributes specifying its statuses, at least one Boolean operation checking certain conditions, and at least one operation for monitoring and activating the controlled element at hand. Each controlled value class exhibits two or more attributes specifying its range constraints and at least one operation for getting these ranges. As the OCL constraint specifies, each controlled value class can be connected to either the Controller class or to Controlled Element classes. Furthermore, in a typical monitoring and acting scenario, a controller object may appear or not. If it appears, it may activate the monitoring and acting operation of its controlled elements, which in turn may sample their controlled values and sensors and record them, if required.

The variety within the PCS domain is quite large. Applications in the domain defer in the number of the controlled elements, the number and type of controlled values and sensors, whether the history of measurements is recorded or not, whether the range constraints are specified to the entire system or separately for each controlled element, whether the system is configurable, etc. These varieties are captured in the PCS domain model. In order to demonstrate them, two different PCS applications, a home climate control system and a water level control system, are briefly explained, modeled, and discussed next.

The Home Climate Control (HCC) application, the ADOM–UML model of which appears in Figs. 3 and 4, ensures that the temperature and humidity in the rooms of a house satisfy certain constraints. For this purpose, the system records for each user U its own preferable limit values: minimal temperature (LTU), maximal temperature (HTU), optimal humidity ( OHU), and possible humidity mistake (PMU). These values are independent of the visited room. For detecting the users, a human identifier component is installed in each room. This component returns the cornea and body halo of each user who enters the room. If several users stay in the same room, the actual temperature and humidity are required to satisfy the preferences of the first user who entered the room, U1, i.e., the temperature should be in the range [LTU1,H TU1] and the humidity should be in the range [OHU1 +PMU1,OHU1]. When the person leaves the room, the system (through the human identifier component) will automatically be adjusted to satisfy the preferences of the second person who entered the room and so forth. The actual levels of temperature and humidity are measured by thermometers and humidity gauges, respectively. In each room, one thermometer and one humidity gauge are installed. An air conditioner and a water sprayer, which are also installed in each room, enable changing the temperature and humidity at will in order to satisfy the user preferences.
The HCC model shows that this application (1) controls one type of elements (rooms), (2) monitors two values, temperature and humidity, which are specific to a user and relevant for all controlled elements, (3) uses three types of sensors, namely thermometers, humidity gauges, and human identifiers, (4) controls two devices, air conditioners and water sprayers, (5) adjusts the temperature and humidity in the room to the preferences of the first human (user) who entered the room (and is still there), and (6) includes three types of monitoring and acting scenarios, heating, cooling, and spraying, the first one shown here in Fig. 4. The application does not require saving history of measurements or turning to an external assistance (in the form of an exception handler).

The purpose of the water level control (WLC) application, the ADOM–UML model of which is given in Figs. 5 and 6, is to monitor and control the water levels in tanks, ensuring that the actual water level of tank \(i\) is always in the closed range \([\text{Low}_i, \text{High}_i]\). If a problem occurs and some of the tanks do not satisfy their boundary constraints, the system tries to resolve the problem internally, for example, by rebooting the system. However, if the problem cannot be resolved internally and at least one of the tanks, tank \(i\), goes beyond its extended boundaries \([\text{Lowest}_i, \text{Uppermost}_i]\), an external hazard handling system is activated for the purpose of trying to resolve the problem externally.

The actual levels of the different tanks are measured by boundary sticks. In each tank, several boundary sticks are installed at different locations. The water level of a tank is considered as the average of all the boundary stick measurements. Each tank is also coupled to filling and emptying faucets which respectively inject and drain water when the water height in the tank reaches its low or high desirable limits. The history of the measured water heights is recorded, so periodical reports on the tanks and their water heights can be retrieved.

The WLC model shows that the application (1) controls one type of elements (tanks), (2) monitors a single value, the water height in tanks, while each tank has its own desirable water height, (3) uses a single sensor type, boundary sticks, (4) activates two (simple) types of devices, filling and emptying faucets, (5) requires a special treatment of an external exception handler if the water height in a tank goes beyond predefined extended boundaries, (6) handles one type of monitoring and acting scenarios which deals with both closing and opening faucets, and (7) records the history of measurements.

### 3. Formalizing ADOM–UML

#### 3.1. ADOM’s foundations

The classical framework of metamodeling is based on four layers [37]. The first layer (M0) is the information layer, which comprises the desired data. The model layer (M1), which is the second layer, includes the metadata that describe data in the information layer. The third metamodel layer (M2) consists of the descriptions that define the structure and semantics of metadata. Finally, the meta-metamodel layer (M3) comprises a description of the structure and semantics of meta-metadata. Atkinson and Kühne [2] found several inaccuracies in this framework and suggested a two-dimensional classification approach, which integrates two orthogonal dimensions: physical and logical classifications. The physical dimension, which is the dominant viewpoint of tool builders, views the metamodel layer (M2) as defining the physical classifiers from which models are constructed. In this dimension, the P1 level defines (lingual) concepts from which user models, defined in P0 level, are created. The logical dimension, on the other hand, which is the dominant viewpoint of modelers, focuses on the classification within a domain and is not concerned with the physical representation. According to this dimension, there are three levels, L0, L1, and L2, which respectively represent concepts, concept types, and classifiers. Logical instantiation is considered as
going down this hierarchy, i.e. elements in L1 are logical instantiations of elements in L2, while elements in L0 are logical instantiations of elements in L1.

ADOM’s three layers are mapped to the basic metamodeling framework, and its extensions presented in [2], as follows. The language layer consolidates with the metamodel layer (M2) and is included in the physical layer P1. These metamodels are defined with languages that are specified in the meta-metamodel layer (M3), also included in the physical layer P1, such as Meta-Object Facility (MOF) [37]. The application layer is included in the model layer (M1), the physical layer P0, and the logical layer L1. Finally, the domain layer is included in the model layer (M1), the physical layer P0, and the logical layer L2. Furthermore, the domain layer enforces logical constraints on the application layer, while the language layer enforces physical constraints on both the application and domain layers. Table 2 summarizes the relations between the different layers in the different frameworks.

The connection between the different layers in ADOM, and especially between the domain and application layers, both of which belong to P0 and M1, is done through classification mechanisms provided by the languages used with ADOM. In ADOM–UML, the classification is done by the stereotypes extension mechanism. According to UML documents [38], stereotypes can be used in all metamodeling levels, namely M1, M2, and M3. Furthermore, in M1, two types of models can be specified: profiles and “regular” models. However, different from UML profiles, domain models are specified closely to application models (rather than to the language layer) and enforce constraints on these models. This point is elaborated in Section 4.1.

The rest of the section elaborates on the representation of domain and application models in ADOM–UML, the creation of application models from domain models, and the validation of application model correctness and completeness with respect to the domain rules and constraints.3

3.2. Representing domain models in ADOM–UML

A domain model in ADOM is a syntactically correct model that captures the understanding gained in a specific knowledge area (domain). In particular, it includes the specification of the commonality and variability allowed among applications in the domains. In order to determine whether an application model fulfills the domain constraints, we need to identify what are valid logical instantiations of domain elements. For this purpose, we distinguish among three types of elements.

Fig. 3. The HCC application model: the application structure expressed in terms of a class diagram.

3 Note that the actual construction of domain models in ADOM, including extraction and gathering of domain knowledge, is out of the scope of this paper. We focus on the activities that can be carried out after having the domain knowledge and constraints, namely representing this knowledge, instantiating it in specific applications, and validating that the applications fulfill the domain constraints.
Definition 1 (relational elements). A relational element re is an explicit binary directional relationship between two other elements. Notation: \( re = (s, t) \) connects a source element \( s \) to a destination element \( t \).

A relational element is often associated with a lingual construct and is represented by a meta-class in the language layer. In case of bidirectional relationships, the model is assumed to have two relational elements, \( re_1 = (s, t) \) and \( re_2 = (t, s) \). Note that n-ary relationships can usually be split into binary relationships. Associations and messages are examples of relational elements in UML.

Definition 2 (dependent elements). A dependent element \( d \) in model \( M \) is an element which has a binary directional relationship with another element \( d' \) in model \( M \), such that the omission of \( d \) from \( M \) implies the omission of \( d' \). \( d' \) is termed the dependee of \( d \) in model \( M \).
The relationship between $d$ and $d'$ is usually implicit and is represented by a meta-relationship in the language layer. Attributes, operations, sub-packages, and nested combined fragments are examples of dependent elements: attributes and operations depend on their owning classes, sub-packages depend on their owning packages, and combined fragments depend on their owning combined fragments. Messages are both relational and dependent elements: they connect two objects and depend on their owning combined fragments.

**Definition 3** (*first-order elements*). A *first-order element* in model $M$ is an element which is neither relational nor dependent in model $M$.

Top-level packages are considered as first-order elements.

**Definition 4** (*domain model*). A *domain model* is a triple $(E_D, MULT, mi)$ such that $E_D$ is a set of model elements constructing a syntactically correct model, $MULT \subseteq N \times (N \cup \{\ast\})$ is a set of multiplicity pairs (where $N$ is the set of the natural numbers and $\ast$ represents

---

**Fig. 6.** The WLC application model: a typical emptying/filling scenario expressed in terms of a sequence diagram.
Regarding operations, domain models can constrain their entire UML (e.g., integer, Boolean, float, String, and Date) or a set of those in the domain level, defining it as any atomic or composite type in elements. For example, the type of an attribute can be constrained to be an atomic type, such as an integer or a string, or it can be a composite type, such as a collection or a set. The multiplicity indicator of an association in the domain model may impose additional (non-multiplicity) constraints, such as the range of times this domain element can be logically instantiated in any application model of that domain. Specifying, for example, the multiplicity indicator of a top level package as \( \text{mi} = (n, m) \), where \( n \leq m \), implies that any application in the domain must have between \( n \) and \( m \) packages that logically instantiate the package.

A multiplicity indicator of a relational element specifies the range of times this domain element can be logically instantiated in any application model of that domain giving that its source and destination have been logically instantiated. Specifying, for example, the multiplicity indicator of an association \( r \) between class \( A \) and class \( B \) as \( \text{mi}(r) = (n, m) \), where \( n \leq m \), implies that in any application in the domain that logically instantiates both \( A \) and \( B \), each logical instantiation of \( A \) is connected within the range of times \( n \) and \( m \) logical instantiations of \( B \) via associations that logically instantiate \( r \) and vice versa.

Finally, a multiplicity indicator of a dependent element specifies the range of times this domain element can be logically instantiated in any application model of that domain giving that its dependence has been logically instantiated. Specifying, for example, the multiplicity indicator of an attribute \( A \) of class \( C \) as \( \text{mi}(A) = (n, m) \), where \( n \leq m \), implies that in any application in the domain, a class that logically instantiates \( C \) must have between \( n \) and \( m \) attributes that logically instantiate \( A \).

A domain model may impose additional (non-multiplicity) constraints. We divide these constraints into two groups: language-related and dependencies. Language-related constraints utilize the expressiveness of the modeling language and the fact that the same modeling language is used in both application and domain layers in order to specify additional features for the different model elements. For example, the type of an attribute can be constrained in the domain level, defining it as any atomic or composite type in UML (e.g., integer, Boolean, float, String, and Date) or a set of those. In case the type of the attribute (or parameter) does not have to be constrained in the domain level, its type will remain empty. Regarding operations, domain models can constrain their entire signatures, including the number of parameters, their types, and the operation returned types. Another example of a language-related constraint is the order and type of messages in a sequence diagram, which can also be constrained in the domain layer.

The second type of domain constraints is dependencies, which can be further divided into inclusion and exclusion. An inclusion dependency can be specified as “\( A \) requires \( B \)”, while an exclusion dependency can be specified as “\( A \) excludes \( B \)”. In the context of ADOM–UML, both can be specified using the “implies” construct of OCL: “\( A \) implies \( B \)” for inclusion and “\( A \) implies not \( B \)” for exclusion.

Table 2: The relations between the different layers in the different frameworks.

<table>
<thead>
<tr>
<th>Physical dimension</th>
<th>Framework</th>
<th>Logical Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P0</td>
<td>Lo</td>
</tr>
<tr>
<td>Classical</td>
<td>M0 layer</td>
<td>M1 layer</td>
</tr>
<tr>
<td>ADOM’s</td>
<td>N/A</td>
<td>Application layer</td>
</tr>
<tr>
<td>Classical</td>
<td>M2 layer</td>
<td>Language layer</td>
</tr>
<tr>
<td>ADOM’s</td>
<td>Language layer</td>
<td>N/A</td>
</tr>
</tbody>
</table>

N/A, not applicable.

\( \infty \), and \( \text{mi} : E_D \rightarrow \text{MULT} \) is a function. The elements in \( E_D \) are termed domain model elements and the elements in \( \text{MULT} \) are termed multiplicity indicators. The multiplicity indicators satisfy \( \forall \text{MULT} = (k, m) \in \text{MULT}, m \geq k, m > 0 \).

As noted, the multiplicity indicators define the minimal and maximal numbers of times a domain element may be logically instantiated\(^\text{4}\) in a particular application model. Graphically, the constraint \( \text{mi}(E) = (n, m) \) is specified in UML by associating the stereotype \(<\text{multiplicity} \min = n \max = m>>> \) to the relevant element \( E \). However, their meanings are slightly different for the three defined elements types. The multiplicity indicator of a first-order element specifies the range of times this domain element can be logically instantiated in any application model of that domain. Specifying, for example, the multiplicity indicator of a top level package \( P \) as \( \text{mi}(P) = (n, m) \), where \( n \leq m \), implies that any application in the domain must have between \( n \) and \( m \) packages that logically instantiate \( P \).

\(^{\text{4}}\) Logical instantiation was informally defined in Section 3.1. A formal definition of this concept is given in Section 3.3 (see Definition 6).
tion model may further restrict these cardinality constraints. In particular, a many-to-many association in a domain model can be logically instantiated by many-to-many, many-to-one, or one-to-one associations in an application model in that domain.

To guide and constrain the behavioral specification of process control systems, Fig. 2 uses the sequence diagram notation for specifying a typical scenario of monitoring and acting in the domain. Here again, the appearance of the model elements (i.e., objects, messages, and combined fragments) and their multiplicities in a particular application are constrained by the multiplicity indicators (stereotypes) in the domain model. Note that these multiplicity constraints may be different from their counterparts in the class diagram, as the multiplicity constraints in a sequence diagram refer to objects and messages (as opposed to classes and operations in class diagrams) in specific scenarios (and not in the whole application). The domain sequence diagram can also define additional language-related constraints and dependencies, for instance, on the order of the messages, their types (e.g., procedure calls vs. asynchronous messages), their embedding within different combined fragments, and their sources and destinations. The scenario in Fig. 2, for example, starts with zero or more messages of type monitor and act that arrive from an unknown source to the only object of type Controller that may participate in the scenario. This can be followed by a loop of at least one sequence in which a Controller-type object sends at most one message of type monitor and act. Each monitor and act message is followed by at least one sequence in which the measured values and possible ranges are retrieved and optionally recorded. Finally, for each such block, a condition is checked and if fulfilled an action of the (external) controlled device is taken. Furthermore, in some cases, a notification message can be sent to an (external) exception handler if a problem is found.

Note that in sequence diagrams, combined fragments are the dependees of their included messages and embedded combined fragments. Thus, the meaning of a mandatory message within an optional combined fragment can be interpreted as “if the combined fragment is logically instantiated in an application model, then the message is mandatory for each such instantiation”.

3.3. Creating application models from domain models in ADOM–UML

Application models in ADOM are syntactically correct models that specify particular applications or systems. Each element in an ADOM application model may be associated with its domain counterparts (using the stereotype mechanism). Elements that have no domain counterparts are considered as application-specific elements.

**Definition 5 (application model).** An application model is a triple \((E_A, C, cl)\) such that \(E_A\) is a set of model elements constructing a syntactically correct model, \(C\) is a set of model elements for which there exists a domain model \((E_D, \text{MULT}, \text{mi})\), \(C \subseteq E_D\), and \(cl:E_A \to C\) is a mapping. The elements in \(E_A\) are termed application elements and the elements in \(C\) are termed domain classifiers.

A domain model may be used as a starting point for guiding the developer about the needed and recommended elements, their structure, and behavior in the context of a particular application. Each one of the elements that appears in the domain model can serve as a stereotype of an application element of the same type (e.g., a class that appears in a domain model may serve as a classifier of classes in an application model). The number of times a particular classifier can appear in an application model is constrained in the domain model by the multiplicity indicators. Furthermore, the application elements are required to fulfill the structural and behavioral constraints induced by their classifiers in the domain model.

Figs. 3–6 exemplify two possible application models that maintain the constraints specified in the domain model given in Figs. 1 and 2. The main operation for creating application models from domain models is logical instantiation, which is defined next.

**Definition 6 (logical instantiations).** An element \(e_A\) in an application model logically instantiates an element \(e_D\) in a domain model iff:

1. The type (meta-class) of \(e_A\) is identical to that of \(e_D\).
2. If \(e_A\) is a first-order element then its classifier is the name of \(e_D\).
3. If \(e_A\) is a relational element then:
   a. it has no classifier or its classifier is the name of \(e_D\), and
   b. there exist application elements \(s_A\) and \(t_A\) and domain elements \(s_D\) and \(t_D\) such that \(e_A = (s_A, t_A)\), \(e_D = (s_D, t_D)\), and \(s_A, t_A\) logically instantiate \(s_D, t_D\), respectively.
4. If \(e_A\) is a dependent element then:
   a. its classifier is the name of \(e_D\), and
   b. there exist an application element \(d_A\) and domain elements \(e_D\) and \(d_D\) such that \(e_A\) depends on \(d_A\) in the application model, \(e_D\) depends on \(d_D\) in the domain model, and \(d_A\) logically instantiates \(d_D\).

Examples of logical instantiations in the HCC and WLC models are given in Table 3.

3.4. Validating application models against domain models in ADOM–UML

After creating a draft of an application model from a domain model, the developer continues improving the model so it will fulfill the requirements of the application at hand. Possible operations at this stage are adding new first order, relational, and dependent elements and changing the sources or destinations of the existing relational elements (e.g., adding sources of found messages, adding destinations of lost messages, and replacing direct associations with indirect associations through newly added application-specific elements). Checking the validity of the resultant application model after carrying out these operations is not a trivial task. Most of the works in the area of domain analysis provide techniques for reusing domain artifacts in applications rather than validating domain constraints in applications (see Section 4 for details). ADOM implements an automatic validation procedure which refers to the adherence of the application model to the domain model. It does not refer to the verification of the specific application requirements in the application model. In other words, the validation capability of ADOM checks the fulfillment of the domain constraints and guidelines in the application model.

The validation of an application against its domain model utilizes the fact that both models are specified in the same modeling language. It is performed in three phases: element reduction, element unification, and model matching. In this process, we first relax the application models to the domain terminology (the element reduction phase) and then we validate that this relaxation satisfies the domain constraints and rules (the element unification and model matching phases).

3.4.1. Element reduction

In this iterative phase, a reduced model that does not include application-specific elements is created as follows. Initially, the reduced model equals to the application model. In each step of this phase, an application-specific element from the reduced model, first order, dependent, and relational elements in this order, is chosen and omitted (from the reduced model), imposing compensating operations to the reduced model.
**Definition 7** (application-specific element). An application-specific element is an element which does not logically instantiate a domain element.

**Definition 8** (compensating operation). A compensating operation defines the changes that should be introduced to a model as a result of omitting an application-specific element. In particular:

1. If the application-specific element, e, participates in two relational elements \( e_1 = (s, e) \) and \( e_2 = (e, t) \) in the reduced model before the omission, and a relational element of the form \( e' = (s, t) \) can be considered a logical instantiation of a domain element, then the relational element \( e' = (s, t) \) is added to the reduced model (after the omission of \( e \)). All the relational elements of the form \( (s, e) \) or \( (e, t) \) are omitted from the reduced model.

2. If there exists a dependent element \( de \) such that \( de \) depends on the application-specific element \( e \) in the reduced model before the omission, then the omission of \( e \) implies the omission of \( de \) and all the elements that depend on \( de \).

Table 3

<table>
<thead>
<tr>
<th>Diagram type</th>
<th>Domain element</th>
<th>Its logical instantiations in the HCC model</th>
<th>Its logical instantiations in the WLC model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class diagram</td>
<td>Controlled Element</td>
<td>Room, Human Identifier, ThermometerItem, HumidityGaugeItem</td>
<td>Tank, BoundaryStick</td>
</tr>
<tr>
<td></td>
<td>Sensor</td>
<td>monitorAndAct message</td>
<td>checkHeightAndAct of WaterController</td>
</tr>
<tr>
<td></td>
<td>The attribute cdStatud of ControlledDevice</td>
<td>SprayerStatus of WaterSprayer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The operation monitorAndAct of Controller</td>
<td>monitorAndHeat, monitorAndStopHeating, monitorAndCool, monitorAndStopCooling, monitorAndSpray of ClimateController</td>
<td></td>
</tr>
<tr>
<td>Sequence diagram</td>
<td>The first monitorAndAct message</td>
<td>The first monitorAndHeat message</td>
<td>No instantiation</td>
</tr>
<tr>
<td></td>
<td>The alt combined fragment</td>
<td>The alt combined fragment</td>
<td>The (four) alt combined fragments with the condition ‘act’</td>
</tr>
<tr>
<td></td>
<td>The opt combined fragment</td>
<td>No instantiation</td>
<td>The (two) opt combined fragments with the condition ‘act’</td>
</tr>
</tbody>
</table>

Table 4

<table>
<thead>
<tr>
<th>Diagram type</th>
<th>( re_1 = (s, e) )</th>
<th>( re_2 = (e, t) )</th>
<th>( e )</th>
<th>Changes to the model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class diagram</td>
<td>A uni-directional association from A to e</td>
<td>A uni-directional association from e to B</td>
<td>A class</td>
<td>Adding a uni-directional association from A to B in case a domain counterpart to this association can be found and omitting ( re_1, re_2, ) and ( e )</td>
</tr>
<tr>
<td></td>
<td>A uni-directional association from A to e</td>
<td>A bi-directional association between e and B</td>
<td>A class</td>
<td>Adding a uni-directional association from A to B in case a domain counterpart can be found and omitting ( re_1, re_2, ) and ( e )</td>
</tr>
<tr>
<td></td>
<td>A uni-directional association from e to A</td>
<td>A bi-directional association between e and B</td>
<td>A class</td>
<td>Adding a uni-directional association from B to A in case a domain counterpart can be found and omitting ( re_1, re_2, ) and ( e )</td>
</tr>
<tr>
<td></td>
<td>A bi-directional association between A and e</td>
<td>A bi-directional association between e and B</td>
<td>A class</td>
<td>Adding a bi-directional association between A and B in case a domain counterpart can be found and omitting ( re_1, re_2, ) and ( e )</td>
</tr>
<tr>
<td>Sequence diagram</td>
<td>A message from an object A to e</td>
<td>A message, which immediately follows ( re_1 ), from e to an object B</td>
<td>An object</td>
<td>Changing the source of message ( re_2 ) to A in case a domain counterpart can be found and omitting ( re_1, re_2, ) and ( e )</td>
</tr>
<tr>
<td></td>
<td>A message from e to an object B&lt;sup&gt;a&lt;/sup&gt;</td>
<td>An object</td>
<td>Changing the source of message ( re_2 ) to be a found message in case a domain counterpart can be found and omitting ( re_1, re_2, ) and ( e )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A combined fragment nested in e</td>
<td>A combined fragment</td>
<td>Omitting the nested combined fragment</td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup> The cardinality of the added associations in all these cases is determined as the less restricting cardinalities of elements \( re_1 \) and \( re_2 \). In other words, if \( re_1 \) or \( re_2 \) are many-to-many associations, so will be the added association; if \( re_1 \) and \( re_2 \) are one-to-one associations, so will be the added association; otherwise (\( re_1 \) or \( re_2 \) are one-to-many associations, but none of them is a many-to-many association), the added association will be a one-to-many association.

<sup>b</sup> There is no message immediately after \( re_2 \) from e to a domain-classified object (i.e., an application object which is classified according to the domain).
### 3.4.2. Element unification

In this phase, reduced model elements of the same meta-class (e.g., classes, messages, combined fragments, and so on) that have the same classifier are unified, leaving only one element for each category (as appears in the domain model).

A new stereotype that represents the actual multiplicity of these elements is used to denote the number of elements that are actually classified the same in the reduced model. Similarly to the multiplicity stereotype in domain models, the actual multiplicity stereotype has two associated tagged values, min and max, which respectively specify the minimal and maximal (actual) numbers of application elements that are classified as the corresponding domain element.

The actual multiplicities are calculated according to the model element type. The actual multiplicity of first-order elements is exactly the number of application elements that are classified as the same domain concept. In other words, the min and max tagged values of the actual multiplicity stereotype of first-order elements are equal.

The way of calculating the actual multiplicity of relational elements is as follows. Let \( r_{ij} \) be relational elements of the form \((s_i, t_j)\), such that all \( s_i \) are classified as the domain element \( s \) and all \( t_j \) are classified as the domain element \( t \). Denote SRC, as the group of all \( r_{ij} \) such that \( s_i \) is their source and DST, as the group of all \( r_{ij} \) such that \( t_j \) is their destination. Then, the minimal actual multiplicity of \( r = (s, t) \) is \( \min(|\text{SRC}|, |\text{DST}|) \) and its maximal actual multiplicity is \( \max(|\text{SRC}|, |\text{DST}|) \), where \( |\text{SRC}| \) and \( |\text{DST}| \) respectively denote the sizes of the groups SRC and DST.

Similarly, the actual multiplicity of dependent elements is calculated as follows. Let \( d_j \) be dependent elements, each of which depends on \( d_j \), all \( d_e \) are classified as the domain element \( d_e \) and all \( d_e \), are classified as the domain element \( d_e \), set \( d_j \) that depends on a specific \( d_j \). Then, the minimal actual multiplicity of \( d \) is \( \min(|\text{DEP}|) \) and its maximal actual multiplicity is \( \max(|\text{DEP}|) \), where \( |\text{DEP}| \) denotes the size of the group DEP.

The resultant model from this phase is termed the **verifiable model**. Fig. 7 demonstrates the verifiable model of the HCC application. The actual multiplicity of the class Sensor, for example, is set to \((3, 3)\), since there are three different types of sensors in the HCC application model: Human Identifier, Thermometer Item, and Humidity Gauge Item. Each one of them has exactly one attribute capturing the sensor identity, between 1 and 3 attributes classified as measured values (Thermometer Item and Humidity Gauge have one each and Human Identifier has three), and between 1 and 3 operations for getting the measured values (here again, Thermometer Item and Humidity Gauge have one each and Human Identifier has three). The actual multiplicity of the associations between the class Sensor and the class Controlled Element is set to \((1, 3)\), since each Sensor class in the HCC application model is connected to exactly one class classified as Controlled Element, while the latter is associated to the three Sensor classes. Note that the multiplicity indicator of the message get MeasuredValue in the sequence diagram is \((1, 2)\), since in the HCC model this message is logically instantiated once with Thermometer Item as its destination and twice with Human Identifier as its destination.

Fig. 7. The verifiable model of the HCC application: (a) the class diagram and (b) the heating sequence diagram.
One should bear in mind that the verifiable models, as well as the reduced models, are (automatically) created only as a means for validating the application models against the domain models.

3.4.3. Model matching

This phase matches the verifiable model with the domain model. In particular, the matching model phase checks the following conditions:

1. All the elements in the verifiable model are named as domain model elements.
2. For each element in the verifiable model, the boundaries of the actual multiplicity tagged values do not exceed the values of the multiplicity tagged values of the corresponding domain model element.
3. Each element in the domain model that has no logical instantiations in the application model (and consequently does not appear in the verifiable model) has minimal multiplicity of 0 in the domain model.
4. All inclusion and exclusion dependencies are fulfilled.

Checking the above-mentioned conditions on the verifiable model given in Fig. 7 shows that the HCC application maintains the PCS domain constraints.

3.5. ADOM–UML supporting CASE tool

In order to support developing domains and applications in ADOM–UML, we develop a plug-in to an existing UML tool, called TOPCASED [51]. TOPCASED, which is open source, promotes model-driven engineering and formal methods as key technologies. It uses the eclipse modeling framework [14] for manipulating the modeling tool and models. ADOM–UML plug-in adds the following functionality to TOPCASED: domain model creation, application model guiding, and application model validation. At its current stage, this ADOM-related functionality is supported only in UML class and activity diagrams.

3.5.1. Domain model creation

The creation of domain models is supported by defining an ADOM–UML profile that includes the different multiplicity stereotypes with their associated tagged values. These stereotypes are assigned to the top level Element class in the UML metamodel, allowing specification of multiplicity-related commonality and variability of all domain elements (e.g., mandatory and optional elements).

3.5.2. Application model guiding

When creating a new modeling project, the modeler requests the tool to semi-automatically create an application model from the selected domain model. A profile based on the selected domain model is created, including the different domain model elements, each of which is attached to the relevant element types. Domain elements that are described by classes, for examples, are translated to stereotypes that are attached to the Class meta-class. The tool adds to the current application model the minimal (logical) instantiations for each mandatory first-order domain element, as specified by the multiplicity indicators (stereotypes) in the domain model. For each such instantiated element, all its mandatory dependent elements are instantiated (the minimal number of times each). After creating the initial application model, the modeler can continue developing the application model by adding, removing, and updating various model elements, as well as assigning the proper domain classifiers (stereotypes) to them.

3.5.3. Application model validation

At any moment in the application development process, the modeler can choose to activate this option, which executes the three-step algorithm specified in the previous section, and results with a report of errors that refer to violation of domain model constraints. Fig. 8 is a screenshot from the tool, showing an error report that resulted when validating an erroneous HCC application model against the PCS domain model.

4. Related work

Different approaches to domain analysis in general and domain modeling in particular have been proposed over the years. They range from reuse techniques to product line methods. In this section, we review these works concentrating on their support in creating and validating application models according to domain models. Guiding the creation of application models should refer to issues like the relationships between a domain and its applications, the way according to which variability is managed [1,53], the constraints that should be held when constructing an individual application [1,52], the way according to which decisions are made during application construction [44], and the means for selecting components for a specific application [22]. Validation is essential for helping reduce errors, faults, and consequently costs and difficulties. In this context, a domain model may be perceived as enforcing constraints on its applications [1,39]. The validation types that a domain specification may offer are derived from the commonality and variability it supports.

4.1. Comparison of related works in domain analysis

Architectural-based methods for domain analysis, such as [7,16,29,35], define the domain knowledge in components, libraries, or architectures. These various domain artifacts are reused in an application as they are, but can be modified to support the particular requirements at hand. They usually do not provide the designer with guidelines to support a specific application design; rather they allow selecting the relevant elements required by the designated application, while adaptation and assembly of these elements are usually out of the method scope. Furthermore, these methods do not support validation of specific applications according to the domain constraints and requirements.

In the recent years the term domain-specific languages (DSLs) has grasped for describing languages specially tailored to specific domains [30]. Compared to general-purpose programming languages, DSLs offer substantial gains in expressiveness and ease of use. As the development of DSLs requires both domain knowledge and language development expertise, the decisions to develop DSLs are rare and most DSLs never get beyond the implementation of libraries for developing applications in the domains. Hence, the primary contribution of DSLs is to enable the reuse of software artifacts, such as language grammars, source code, software designs, and domain abstractions.

Feature-oriented domain analysis methods, such as those presented in [19,40], FODA [23,22], and PLUS [20], suggest that a system specification will be derived by tailoring the domain model according to the features required by the specific application. That is, a specific system uses the reusable architecture and instantiates a sub-set of features from the domain model. Variability is modeled and managed in most feature-oriented methods through variation points that determine one or more locations at which variants may occur, where variant is one way to realize a particular
variability and bind it in a concrete way [21]. Each variation point exhibits the following main characteristics: cardinality (indicating the number of variants that have to be selected for the specific variation point), openness (specifying whether it is possible to add non-listed variants at this variation point), binding time (constraining the software lifecycle phase at which the variants should be bound), variants (specifying the set of possible features to be included in the specific variation point; a variant can be either mandatory or optional), and dependencies (restricting the variant selection). van Deursen and Klint [11] suggest a formal textual notation for feature diagrams, which can be used as a basis for tool development and as mediation between the options provided by software applications and the user requirements. They further show how feature diagrams can be directly mapped to UML class diagrams and consequently be generated to Java code. These methods usually guide the application designer of how to select the required features (variants), while validation is supported by checking whether the feature constraints defined in the domain model hold in the specific application [3,5,9]. Benavides et al. [5], for example, defined twelve operations for automating the analysis of feature models, including determining feature model satisfiability, finding a product from a feature model, and obtaining all products. Another example is the work of Mannion [27,28] that suggests representing a feature model as a logical expression and using this expression as a vehicle for validating feature sets derived from the feature model. However, the main limitation of these methods is that they support variability only through variation points and specialization, while neglecting the addition of new application features which neither violate the domain constraints nor specialize the different variation points. In this way the variety of applications suitable for a specific domain model is limited, imposing constraints on the definition of a domain.

Metamodeling techniques for domain analysis enable definition of domains as metamodels that serve both for capturing domain knowledge and for validating particular applications in a domain. Following these techniques the domain and application models are described in two abstraction levels: metamodel (M2) and model (M1). Examples for this type of approach are the studies by Schleicher and Westfechtel [43], Gomaa and Eonsuk-Shin [18], Morisio et al. [32], MetaEdit [31], and the Generic Modeling Environment (GME) [10,36]. Most of these methods use standard visual modeling languages, such as UML. Recent works that use UML 2.0, e.g., [46,53], recruit the profile mechanism for the purpose of adapting an existing metamodel to constructs that are specific to a particular domain, platform, or method. In this context, stereotypes are meta-classes, tagged values – metaattributes, and profiles – special kinds of packages. However, although profiles provide a better solution than earlier textual-based stereotypes, they are still not geared towards the mission of domain analysis, including application creation and validation tasks. Furthermore, profiles are mainly expressed in terms of UML class and package diagrams, while (application) models are expressed using different structural and behavioral diagrams types, enlarging the gap between the application and domain layers and making the application creation and validation tasks difficult.

The UML-based language for specifying domain-specific patterns [17,26,24], which can be considered as a metamodeling approach, defines a Role-based Metamodeling Language (RBML) that modifies UML metamodel in order to express the domain variability in terms of element multiplicity. The approach has been applied to UML class, sequence, and state diagrams. Similarly to ADOM, when specifying a particular application, stereotypes are used for connecting the application elements to the relevant domain (pattern) elements. However, the approach supports variability to a limited extent. In particular, application-specific additions and their correct usage are not explicitly supported, although allowed. In later work, Kim and Shen [25] suggest a conformance mechanism which validates the application models against the relevant domain models. However, this mechanism does not include special treatments for application-specific additions. For example, if a direct association in the structural domain model (termed Static Pattern Specification, SPS) is replaced by indirect associations
through an additional application-specific class in the class diagram, the conformance mechanism will result with the conclusion that the class diagram does not conform to the SPS, limiting the possible variants of a SPS.

Pattern-based modeling approaches, such as [28,34], can also be considered for domain analysis. They examine common knowledge and utilize it for developing applications in order to help construct better applications in less time and efforts. They promote reusing solutions for recurring design problems. However, these are usually too abstract to be used directly, refer mainly to the common features of the solutions in the different contexts, and require further expertise in order to correctly apply the patterns.

In the area of business process design and management, the domain analysis is encouraged through reference models, which are models used for supporting the construction of other models. Reference models were originally suggested as a vehicle for enhancing the development of information systems (e.g., [4,15,45,50]), but they also provide generic knowledge about business processes in order to assist in their design in specific enterprises. While much attention has been given to the content of these models, the actual process of reusing this knowledge has not been extensively addressed [42].

4.2. ADOM benefits and limitations

ADOM, which can be considered as a metamodelling domain analysis approach that combines feature-oriented and architectural ideas, has the following three main advantages over other domain analysis methods reviewed in the previous section: (1) it provides a complete method that focuses on two of the main usages of domain models: creating and validating application models according to their corresponding domain models, (2) it uses the same languages and techniques for the application and domain layers, utilizing to the fullest the modeling language expressiveness and bridging the gap between the abstraction levels of these layers, and (3) it does not depend on a specific modeling language; rather it can easily be adapted to different modeling languages with small changes (if any) to their specification or metamodel. These three advantages, as well as their derived limitations, are discussed in this section.

ADOM provides support and guidance for the creation of application models in specific domains by enabling the specification of rules and “best practices” in the form of mandatory and optional domain elements. An experiment we conducted, and is reported in [41], advocates that the availability of a domain model helps achieve more complete models without reducing the comprehension of these models. However, many domain analysis methods provide means for reusing domain artifacts and knowledge in particular applications, while enabling changing and modifying these artifacts in the context of the particular application. This flexibility dramatically harms the ability to validate the correctness and completeness of applications within a domain. Validating an application model throughout gradual system development stages reduces the development cost as errors are detected in early stages. These errors cannot be automatically found when syntactic modeling languages are used alone. Most of the reviewed methods provide simple validation mechanisms (if any) that only check whether a specific feature appears in an application and maintains the feature dependencies as expressed in the domain model. ADOM’s validation procedure handles more complicated cases in which application-specific elements are introduced in different parts of the application model and connected to variants of domain elements.

ADOM treats domains similarly to applications, but at different abstraction levels. Hence, the expressiveness of ADOM is directly derived from the modeling language expressiveness (UML in this paper). Since UML supports specifying both application structure and behavior, ADOM–UML supports specifying domain structure and behavior. Furthermore, using the same terminology, languages, and techniques in both domain and application layers might reduce the communication problems between the different stakeholders in the system development process [1], most relevantly software and domain engineers.

Finally, ADOM, as presented in Section 3, is not tightly connected to the modeling language. As such, it can be adapted to different modeling languages, for different tasks, in different environments, and by different stakeholders. The only requirement from the modeling language used in conjunction with ADOM is that it will have a classification mechanism with which ADOM can specify commonality and variability of applications in a given domain and associate application elements with relevant domain elements. If such a classification mechanism does not exist, ADOM requires modification to the language specification or metamodel. Indeed, we already successfully applied ADOM to other UML diagram types, including use case, state, and activity diagrams. In [42], for example, ADOM is used in conjunction with UML activity diagrams for modeling reference models and business processes. Lately, we extended this work to Event Process Chain (EPC) notation [47] and Business Process Modeling Notation (BPMN). In [48], an ADOM-based approach is applied to the Object-Process Methodology (OPM) [12]. OPM, which departs from the object-oriented approach, treats objects and processes as equally important elements that are used together to describe structural and behavioral aspects of systems in a single view. This ADOM–OPM approach enables the specification of domain models in OPM, their definition as design policies to other OPM (application) models, and the validation of application models against the relevant domain models. A particular application makes use of domain models by specifying the roles the different application elements play according to the domain concepts and constraints. We also adapted ADOM to Tersus [49], a commercial product that enables development of Web applications by visual modeling. The domain layer in Tersus is termed prototype and is mainly used for defining executable design patterns. The three last mentioned languages, namely EPC, OPM, and Tersus, had no built-in classification mechanism, so slight modifications were required to their specifications. In OPM, for example, a Role element was added to the metamodel and connected to Element which is the top level element in the OPM hierarchy. In EPC, modifications were required to all the four core (top level) elements: events, processes, connectors, and arcs.

Note that the attempt to make ADOM as general as possible, and in particular language-independent, might result in over-complicated application models that are developed too closely to the domain model. This problem can be mainly resolved (or at least reduced) by adding a language-dependent phase whose aim is to simplify and improve the application models achieved by the ADOM approach according to the selected language semantics.

5. Summary and future work

In this paper we presented the Application-based Domain Modeling (ADOM) approach, focusing on its instantiation and validation capabilities. ADOM treats domains similarly to applications and enables creating application models from domain models, improving the draft applications to fulfill the requirements at hand, and validating the fulfillment of the domain constraints in the final application models. Furthermore, it can be adapted by different modeling languages involved in different development stages, utilizing to the fullest the expressiveness of the modeling language in both application and domain layers. As opposed to other domain
analysis techniques, ADOM provides a complete method that focuses not only on the creation of application model skeletons, but also guides the development of complete and valid application models. In particular, ADOM’s validation mechanism, which includes element reduction, element unification, and model matching stages, enables detecting incorrectness or incompleteness in an application model in early development phases. The approach is supported by a CASE tool, making it accessible for both application (software) and domain engineers.

In the future, we plan to further examine ADOM, extend it to different fields, including software engineering, method engineering, and business process design, and improve its supporting tools. In particular, we plan to check the completeness of the supported constraints in ADOM and to provide techniques for the creation and specification of domain models. We started to work on the latest direction by developing algorithms for construction of domain models from existing applications in the domain and from other analog domain models. These algorithms, which use both structural and semantic techniques, should reduce the costs of domain engineering activities and improve their profitability.

References