International Journal of Software Engineering
and Knowledge Engineering
Vol. 18, No. 8 (2008) 1–28

APPLICATION-BASED DOMAIN ANALYSIS APPROACH AND ITS OBJECT-PROCESS METHODOLOGY IMPLEMENTATION

ARNON STURM
Department of Information Systems Engineering,
Ben Gurion University of the Negev, Beer Sheva, 84105, Israel
sturm@bgu.ac.il

DOV DORI
Faculty of Industrial Engineering and Management,
Technion — Israel Institute of Technology, Haifa, 32000, Israel
dori@ie.technion.ac.il

ONN SHEHORY
IBM Haifa Research Lab, Haifa University Campus, Haifa, 31905, Israel
onn@il.ibm.com

Submitted 14 September 2006
Revised 18 May 2007
Accepted 25 August 2007

Domain engineering can simplify the development of software systems in specific domains. During domain analysis, the first step of domain engineering, the domain is modeled at an abstract level, providing guidelines for modeling applications within that domain. Drawbacks of existing domain analysis approaches include poor guidelines for domain-specific application modeling, insufficient validations capability, and limited usability. In this paper we apply the Application-based Domain Modeling (ADOM) to the Object-Process Methodology (OPM). This application requires the extension of OPM with a classification mechanism. Showing that the ADOM-OPM approach overcomes limitations of existing approaches, we further verify experimentally that the level of correctness of an ADOM extended OPM model is higher than that achieved without the extension. Finally, we ensure that the proposed extension does not degrade the generic vanilla form of OPM.

Keywords: Domain engineering; domain analysis; software product line engineering; object-process methodology; information systems development.

1. Introduction

Domain engineering is concerned with building reusable software core assets and components in a specific domain of human interest [1, 2]. Domain engineering activities include domain analysis, domain design, and domain implementation.

Domain analysis can be defined as a process by which information used in developing software systems in a specific domain is identified, captured, and organized with the purpose of making it reusable when creating new systems in that domain. Taking an object-centered viewpoint, Valerio *et al.* have viewed domain analysis as the activity of identifying objects and operations of a class of similar systems in a particular domain [3].

Domain analysis should "carefully bound the domain being considered, consider commonalities and differences of the systems in the domain, organize an understanding of the relationships between the various elements in the domain, and represent this understanding in a useful way" [1]. The initial phases of domain analysis concern the identification of a domain (or a set of related domains) and capturing the domain ontology and its variations within the domain. Subsequent stages of domain engineering, namely domain design and domain implementation, are concerned with mechanisms for translating the requirements into systems that are made up of components with the intent of reusing these components effectively.

Methods developed to support domain analysis, reviewed by Czarnecki and Eisenecker [4] and by Sturm and Reinhartz-Berger [5], suffer from a number of weaknesses, including the following:

1. These domain analysis methods lack formality, making it difficult to validate domain-specific applications against their domain models.
2. The notions and notations these methods employ for the domain models are different from those used for the application models, limiting the possibility of collaboration among the various stakeholders engaged in the development process.
3. These methods address primarily the static aspect, characteristics, and constraints of the domain; their treatment of the domain's dynamic aspect is limited.
4. These methods require the use of several views for the specification of both the domain and the application within the domain, resulting in limited accessibility.

The Application-based Domain Modeling (ADOM) approach [5, 6] addresses the above-mentioned problems. In particular, it addresses the first two limitations. ADOM treats a domain as an application in its own right that needs to be modeled before systems in that domain are specified and designed, yet the entire domain is modeled as a regular application that serves as a reference to applications in that domain. The same paradigm, along with its semantics — the set of concepts, and its syntax — the set of symbols, is used for specifying domains and the applications within them. The modeled domain structure and behavior serve to define and enforce static and dynamic constraints on models of application in that domain.

ADOM consists of three layers:

1. The language layer, which is concerned with the underlying modeling language, pertinent ontologies, and their constraints.

2. The domain layer, which uses the language defined within the language layer to model the various domains, including the building blocks of each domain and the relationships among them.
3. The application layer, which consists of domain-specific system models.

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. As we elaborate later, some of these constraints are syntactic, while others are semantic, requiring understanding of the selected modeling language.

To address the above limitations, using the language-independent characteristics of ADOM, in this paper we have elected to implement the ADOM approach using Object-Process Methodology [7] as the modeling language. We refer to this implementation as ADOM-OPM. OPM is an integrated approach to the study and development of systems. The choice of OPM for that purpose is due to its adequacy to carry out the task at hand, as argued below. As a general-purpose systems modeling language, OPM has been used to model systems in various domains, including pattern recognition in mechanical engineering drawings [8], computer integrated manufacturing documentation and inspection [9], and Web applications [10]. These systems and others were modeled without first devising a domain-specific ontology infrastructure, as the ADOM approach advocates. OPM was selected as the modeling language due to its advantages with respect to comprehension and construction of system models compared with multiple-view languages such as OMT and UML [11–13]. In particular, OPM addresses the third and the fourth limitations presented above.

The contribution of this paper is three-fold. First, we validate the suitability of implementing the ADOM approach using a modeling paradigm and language other than UML. Second, we extend OPM with facilities to support domain analysis principles, making it more accessible and efficient for modeling domain-specific systems and products. Third, experimenting with ADOM-OPM, we provide an empirical proof of the advantage of carrying out domain modeling and engineering using this approach for domain modeling over the generic version of OPM.

The rest of this paper is organized as follows. In Secs. 2 and 3 we present the ADOM approach and an overview of OPM, respectively. In Sec. 4 we introduce ADOM-OPM and demonstrate its application by modeling the domain of access control systems and two particular systems within this domain: the drinks vending machine and the automatic teller machine. In Sec. 5 we describe an experiment we performed in order to put to test the suitability of ADOM-OPM for application modeling compared with OPM and report the results of this experiment. Section 6 discusses the benefits and drawbacks of the ADOM-OPM approach, and Section 7 concludes with a summary and future research.

2. Application-Based Domain Modeling

The Application-based Domain Modeling (ADOM) approach is founded on a three-layered architecture: the language layer, the application layer, and the domain layer. Following the OMG-MOF [14] metamodeling framework, the *application layer*, which corresponds to the MOF model layer (M1), consists of models of particular applications and may include their structure and behavior. The *language layer*, which corresponds to the MOF metamodel layer (M2), includes metamodels of modeling languages, such as UML, OPM, etc. The intermediate *domain layer*, which we label M1.5, consists of specifications of models of various domains. ADOM enforces constraints among the different layers: The domain layer enforces constraints on the application layer, while the language layer enforces constraints on both the domain and the application layers.

The ADOM architecture is generic, so it can be used for various purposes and with various modeling languages. Figure 1 depicts an instance of the ADOM architecture, in which the application layer includes three application models: a poker game, a Web-based black jack game, and an auction site. The domain (intermediate, M1.5) layer on top of the application layer includes two domain models: gambling games and Web applications. By constraining these applications, the corresponding domain models provide guidelines for instantiating applications in the application layer of each domain. The language layer in this example includes OPM as the modeling language. The arrows in Fig. 1 show that the poker game is constrained by the gambling games domain, the auction site is constrained by the Web applications domain, and the Web-based black jack is constrained by both domains. The

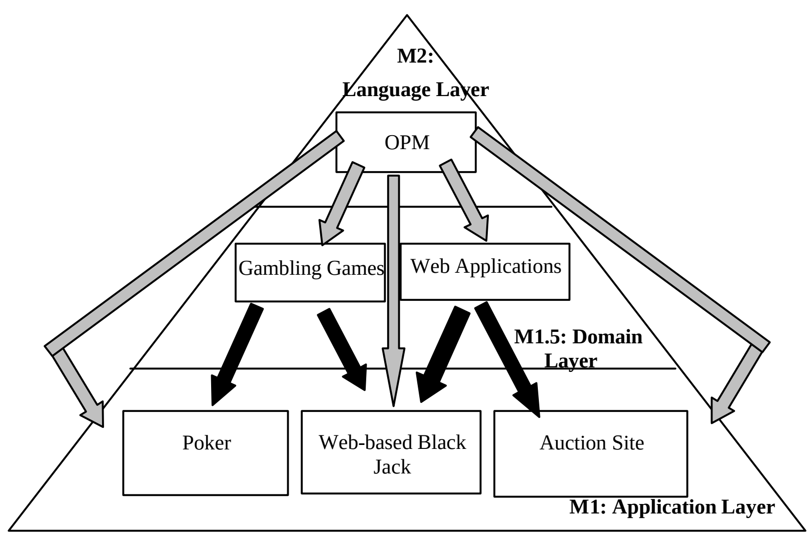


Fig. 1. An instance of the ADOM architecture with two domains and three applications.

gray arrows in Fig. 1 indicate syntactic constraints enforced by the language layer on the domain and application layers in terms of modeling language usage, while the black arrows indicate semantic constraints enforced by the domain layer on the application layer.

ADOM has three purposes: (1) presenting domain models along with their constraints; (2) providing guidelines for instantiating application models in a specific domain; and (3) validating application models against their corresponding domain models.

While ADOM is general and language-independent, a specific modeling language needs to be selected as a basis for a workable dialect of ADOM. In order to apply ADOM, the only requirement from the associated modeling language is to have a classification mechanism that enables categorization of elements within a model. Mechanisms such as UML stereotypes and profiles do exist, but if the modeling language has no classification mechanism, a modification of its specification (via a metamodel) is required. In ADOM, the classification mechanism is used within both the intermediate domain layer and the application layer. Within the intermediate domain layer, the classification mechanism is used for variability management of domain model elements. Variability management in ADOM uses a multiplicity indicator to denote whether and to what extent a model element is optional or mandatory in applications in that domain. Within the application layer the classification mechnism is used for associating domain model elements with application model elements. The application model element has the role of the associated domain model element. For example, in the domain model of the gambling games, a player is a mandatory entity, which has to appear at least once within an application model in that domain. In the Web-based black jack application model, the dealer will play the role of a player, which appears in the domain model.

Since domain and application models are specified in ADOM in practically the same way, the validation of a particular application model against a relevant domain model can be done automatically. Validation in this context refers to the adherence of the application model to the domain model after the instantiation of the domain model to fulfill the requirements of the specific application. To avoid confusion, it should be noted that the ADOM validation procedure does not refer to verifying the fulfillment of the specific application requirements in the application model. Rather, it refers to checking the fulfillment of the domain constraints in the application model. As the domain model captures the domain knowledge, the intention of the specific application model should be retained. The validation of an application against its domain model is performed in three steps: element reduction, element unification, and model matching. For that purpose a virtually new model is built.

In the ***element reduction*** step, things (objects or processes) that have no role assigned to them (since they are particular to the specific application) are removed from the application model. As a consequence, we might need to remove or change things in the application model. The result of this step is the reduced application model.

During the ***element unification*** step, things having the same role in the reduced application model are unified, leaving only one thing for those that are relevant from the domain model and changing its name to its corresponding role name. The multiplicity of these things denotes the number of distinct classes of things in the application model having the same role. The result of this step is termed verifiable model.

Model matching is the third and final step, during which the verifiable model is checked for agreement with the domain model. When validating an application model in a specific domain, the multiplicity indicators constraints are checked. The matching procedure checks the following:

1. All the things (objects or processes) in the model are termed as domain model elements.
2. For each thing in the verifiable model, the boundaries of the actual multiplicity do not exceed the values of the multiplicity of the corresponding domain model element.

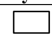
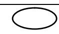
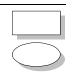
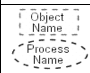
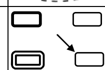
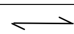
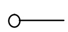
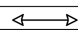
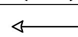
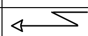
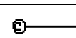
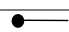
   Formally expressed, $\forall \text{e} \in \text{VerifiableModel} \, \exists \, \text{de} \in \text{DomainModel e.name} = \text{de.name} \wedge \text{e.actualMultiplicity.min} \wedge \text{de.multiplicity.min} \wedge \text{e.actualMultiplicity.max} \leq \text{de.multiplicity.max}$.
3. Each thing in the domain model that does not appear in the verifiable model has minimal multiplicity (in the domain model) of 0.

Additional language-specific constraints may also be checked (e.g., logical connectors, guards, flow, and order). However, since such constraints are specific to the language selected for implementing ADOM, it is not dealt with in this section. We demonstrate the various capabilities of ADOM in Sec. 4.

3. Object-Process Methodology

Object-Process Methodology [7] is an integrated approach to the study and development of systems. The basic premise of OPM is that (possibly stateful) objects and processes are two types of equally important classes of things, which together describe the function, structure and behavior of systems in a single framework in virtually any domain. OPM unifies the system lifecycle stages — specification, design and implementation — within one frame of reference, using a single diagramming tool — a set of Object-Process Diagrams (OPDs) and a corresponding subset of English, called Object-Process Language (OPL), that is intelligible to executives and domain experts who are usually not familiar with system modeling methods, let alone programming jargon. At the same time, OPL is amenable to machine compilation for code generation and database schema specification due to its formal definition by a context-free grammar. In summary, the OPM framework, expressed visually by the OPD-set and textually by the OPL script, provides the ability to grasp the entire system through one visual language, which is useful pri-

Table 1. Main OPM concepts.

Concept Name	Symbol	Definition
Object		An entity that has the potential of stable, unconditional physical or mental existence.
Process		A pattern of transformation that objects undergo.
Essence		An attribute that determines whether the thing (object or process) is physical or informational.
Affiliation		An attribute that determines whether the thing is environmental or internal.
Initial/ Regular/ Final/ Default State		A situation at which an object can exist for a period of time.
Exhibition-Characterization		A fundamental structural relation from a thing (object or process) or a tagged structural relation to a feature (attribute or operation) it exhibits.
Generalization-Specialization		A fundamental structural relation, which denotes the fact that a thing generalizes one or more specialized things.
Aggregation-Participation		A fundamental structural relation, which denotes the fact that a thing aggregates (i.e., consists of, or comprises) one or more things, each of which is referred to as a part of it.
Tagged structural relationship		An association between two things that holds true in the system irrespective of the time of inspection.
Instrument link		A procedural link indicating that a process requires an object for its execution (has a "wait until" meaning).
Effect link		A procedural link indicating that a process changes an object.
Result/ Consumption link		A procedural link indicating that a process creates/consumes an object.
Invocation link		A procedural link indicating that a process activates (invokes) another process.
Event link		A procedural link representing an event (data change, external, etc.) which activates a process.
Condition link		A procedural link representing a condition required for a process execution (has an "if" meaning).
Agent link		A procedural link indicating that a human actor is required for a process execution.

marily for system architects and developers, and one textual language, which is useful for domain experts.

Table 1 presents the main OPM concepts with their definitions and graphical symbols,, including the OPM/T [15] and OPM/Web [10] enhancements used in this paper. A comprehensive specification of OPM is provided by [7].

In OPM, an object class can be either systemic (internal) or environmental (external to the system). Furthermore, an object class can be either physical or informatical (logical). An object class can be at one of several states, which are possible internal status values of the class objects. At any point in time, each object is at some state, and an object is transformed (generated, consumed, or change its state) through the occurrence of a process. A process can affect (change the state of) an environmental or a physical device, or the state or value of a specific attribute of an object class (rather than the state of the entire class, as in object-oriented methods).

Unlike the object-oriented approach, behavior in OPM is not necessarily encapsulated within a particular object class. Allowing the concept of a stand-alone process, one can model behavior which involves several object classes that enable and/or are transformed by the process. Processes can be connected to the involved object classes through procedural links, which are classified into three groups: enabling links, transformation links, and control links. OPM also supports the definition of multiple scenarios which use the same entities. This is done by labeling paths composed of chains of procedural links. These path labels are also used to remove the ambiguity arising from multiple outgoing procedural links from the same process.

OPM's two main built-in refinement/abstraction (also called scaling) mechanisms, unfolding/folding and in-zooming/out-zooming, help manage system complexity. Unfolding/folding is applied by default to objects for detailing/hiding their structural components (parts, specializations, features, or instances). In-zooming/out-zooming is applied by default to processes for exposing/hiding their sub-process components and details of the process execution. A third scaling mechanism, state expression/suppression, provides for showing or hiding any subset of states of an object class. OPM's scaling mechanisms facilitate focusing on a particular subset of things (objects and/or processes), elaborating on their details by refining each thing to any desired level of detail, and managing the complexity of a system model.

The ability to use a single OPM model for specifying the important system aspects — function, structure and behavior — prevents compatibility and integration problems among different diagram types by avoiding the model multiplicity problem [11]. In addition, expressiveness is increased as there is a possibility to define system changes at all levels of granularity.

4. ADOM-OPM Demonstrated Using the Access Control Domain

To implement ADOM-OPM we found it necessary to extend OPM with a classification mechanism by introducing only two new features: (1) A role, which is a stereotype-like element, used within the application model to denote additional semantics for an OPM thing, and (2) a multiplicity indicator, used within the domain model to constrain the number of OPM things (objects or processes) of some class that can be modeled in an application. The rest of this section presents the domain and application models of the Access Control (AC) using the ADOM-OPM approach and the validation of the application model with the corresponding domain model.

Applications within the AC domain are concerned with problems related to accessing objects, and resources using well-defined access policies and procedures. Examples include systems that access databases using batch and interactive interfaces, and local (batch or interactive) and remote access to software and hardware objects in a computer network [16]. Application areas within the AC domain include such devices as product vending machines, automated teller machines (ATMs), and gambling machines.

In AC applications, clients interact with a machine to acquire specific products, and their interactions are recorded. In addition, a human operator usually performs maintenance operations. Since applications in the AC domain manage items and their monetary related aspects, an item identification process may be required. For any interaction, item and cash availability validation is performed, followed by an update of the relevant information upon successful interaction.

In what follows we demonstrate an ADOM-OPM deployment by presenting an AC domain model in Sec. 4.1 and examples for corresponding application models in Sec. 4.2, while in Sec. 4.3 we demonstrate the validation of application models against the domain model.

4.1. *The ADOM-OPM domain layer*

ADOM advocates modeling the domain as a bona fide application. Hence, as Fig. 1 shows, OPM is used as the modeling language for both the domain model and the application model, and each will be constructed as an OPM model with its OPD set.

Figure 2, which depicts the system diagram (SD, top level) of the AC domain, shows that it consists of three environmental (external) entities — **Client**,[a] **Machine**, and **Maintenance Entity**, two processes — **Operation** and **Maintenance**, and four systemic objects — **Owner**, **Company**, **Transaction**, and **Machine Info**. The symbols "*" and "+" at the bottom right edge of some OPM things (objects or processes) are the multiplicity indicators, which respectively indicate zero to many and 1 to many multiplicity constraints of these things within an application model. For example, the + symbol within **Client** in Fig. 2 indicates that at least one object classified (with a role) as **Client** should appear

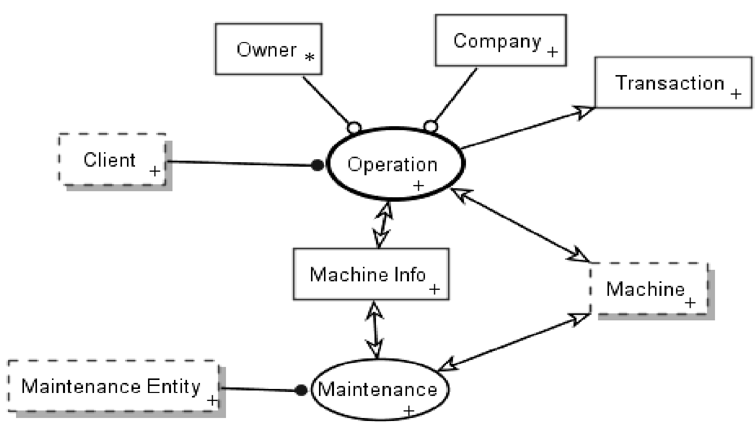


Fig. 2. System Diagram of the Access Control (AC) domain.

[a]Bold face font indicates a thing name within the model.

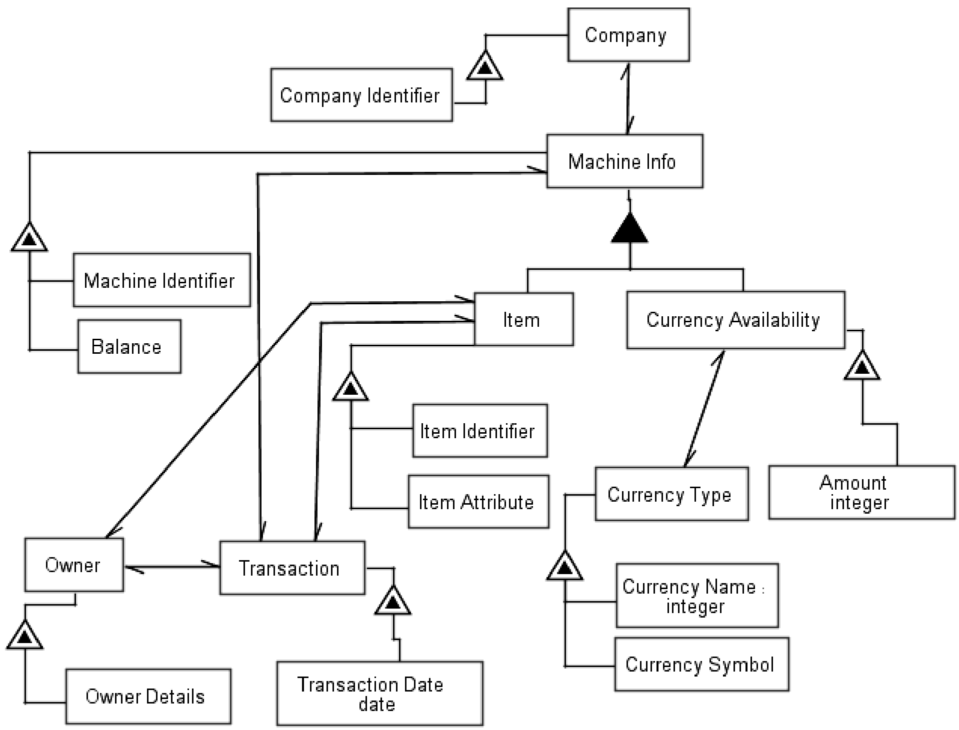


Fig. 3. Machine Info from the top-level AC domain model in Fig. 2 unfolded.

within any application models in the AC domain. In addition, links in the OPM domain model induce constraints on the pertaining application models between respective things. For example, the **Operation** process yields a **Transaction**. This domain model assertion induces a constraint that should hold in any application within the AC domain.

Unfolded in Fig. 3, **Machine Info** is shown to consist of at least one **Item** object and many **Currency Availability** objects. **Item** exhibits (i.e., has the attribute of) **Item Identifier** and at least one **Item Attribute**, and it refers to many **Transactions** and to at least one **Owner**. **Currency Availability** exhibits **Amount** and refers to **Currency Type**, which exhibits **Currency Name** and at least one **Currency Symbol**. **Machine Info** exhibits at least one **Machine Identifier** and an optional **Balance**. **Machine Info** refers to a **Company** and to many **Transactions**. **Company** exhibits **Company Identifier**. **Transaction** exhibits **Transaction Date** and optionally refers to **Owner**, which exhibits at least one **Owner Details**.

In Fig. 4, the **Operation** process is elaborated using the in-zooming mechanism of OPM. The time line within an in-zoomed process flows from top to bottom and determines the order of process execution. Hence, the sub processes in Fig. 4 occur in the following order:

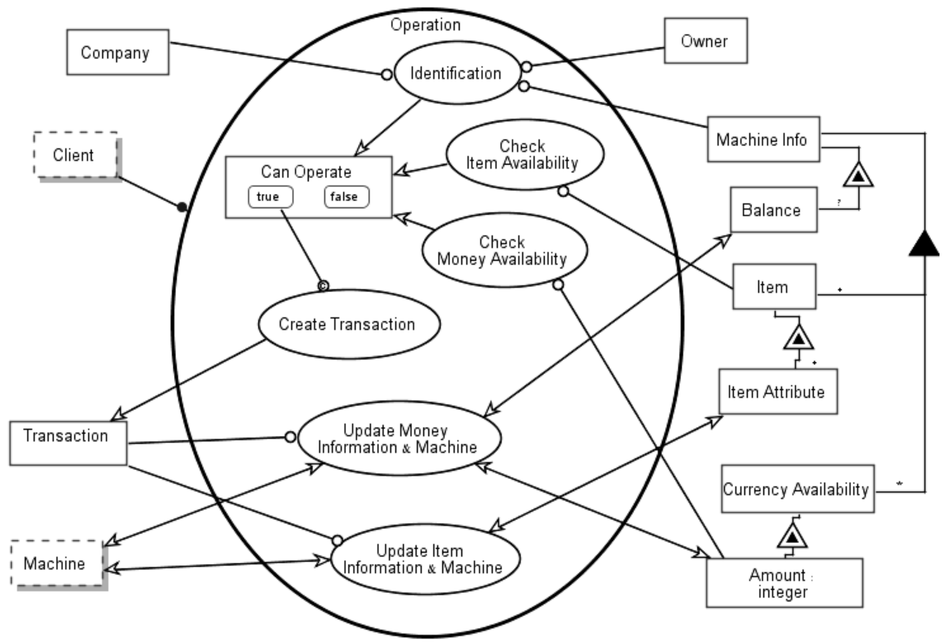


Fig. 4. The process Operation in-zoomed.

1. An **Identification** process, which, as indicated by the multiplicity indicator "*", is optional, requires the objects **Company**, **Owner**, and **Machine Info**, and yields the Boolean object **Can Operate**.
2. At least one **Check Item Availability** process (as indicated by the multiplicity indicator "+"), which requires an **Item** object and yields a **Can Operate** object.
3. At least one **Check Money Availability** process, which requires an **Amount** object and yields a **Can Operate** object.
4. A **Create Transaction** process, which is activated if the **Can Operate** object is **true**, in which case it yields a **Transaction** object.
5. At least one **Update Money Information & Machine** process, which requires the **Transaction** object and affects **Balance**, **Money Amount**, and **Machine**; and
6. At least one **Update Item Information** and **Machine** process, which requires **Transaction** and affects **Item Attribute** and **Machine**.

4.2. *The ADOM-OPM application layer*

The AC domain model resides within the domain layer (M1.5) and serves as a basis for constructing application models within that domain in the application layer (M1). The domain model provides guidelines for modeling an application, and inducing constraints on the application model, it also serves as a validation template. We next model two AC domain applications: drink vending machine (DVM) and automatic teller machine (ATM) [16]. Using these applications, we demonstrate the correspondence between the domain model and its applications, and show the commonalty and variability among applications.

4.2.1. *The Drink Vending Machine application*

The DVM application manages machines that belong to various companies. Each machine is identified by its location and the company that owns it. The application stores the name and telephone number of each company. Each machine can be operated using several coin types. The products sold by each machine are identified by their name and producer. When a customer buys a drink, the system first needs to check whether the product is available and, if needed, whether coins for change are available. When the customer buys a drink, the system generates a transaction, updates the relevant information within the machine, and notifies the (physical) machine about the product and any change coins it needs to deliver. A machine operator can fill drinks and load coins into the machine.

Figure 5 presents SD, the system (top-level) diagram of the DVM application. In the application layer model, each thing (i.e., an object or a process) is associated with a role. For example, the object **Customer** is associated with a *Client*[b] role, which is an object in the domain layer model. The **Buy A Drink** process is associated with the *Operation* role, a process in the domain layer model.

Note that the domain model objects **Owner** and **Company** do not appear as roles in the application model, since they are not required in this application, demonstrating the ability of ADOM-OPM to capture variability within a

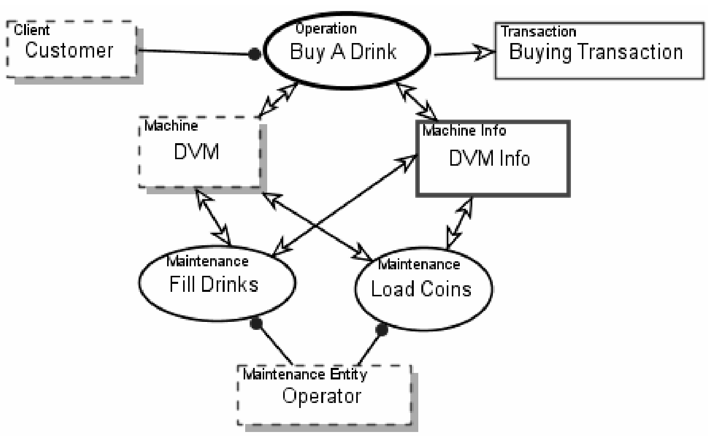


Fig. 5. System diagram of the Drink Vending Machine.

[b]Italic font indicates a classifying role within the application model.

domain using multiplicity constraints. They are not required since their dependee process — identification is redundant as shown in Fig. 7.

As Fig. 5 shows, the system features three top-level processes:

1. **Buy A Drink**, which is triggered by **Customer**, yields **Buying Transaction**, and affects (i.e., changes the state of) **DVM** and **DVM Info**.
2. **Fill Drinks**, which is triggered by **Operator** and affects **DVM** and **DVM Info**.
3. **Load Coins**, which, like **Fill Drinks**, is triggered by **Operator** and affects **DVM** and **DVM Info**.

Note that **Buy A Drink** process conforms to the constraints induced by the *Operation* process, while **Fill Drinks** and **Load Coins** processes conform to the constraints induced by the *Maintenance* process in the domain model shown in Fig. 2.

Constrained by the domain model OPD of Fig. 3, Fig. 6 is an OPD in which **DVM Info** is unfolded. The roles specified within the domain model are mapped to the object and process application classes. For example, the **Producer**, labeled with the role *Owner*, exhibits **Producer Name** and **Producer Address**, which are labeled with the role *Owner Details*, demonstrating how the domain model guides the modeling of the application.

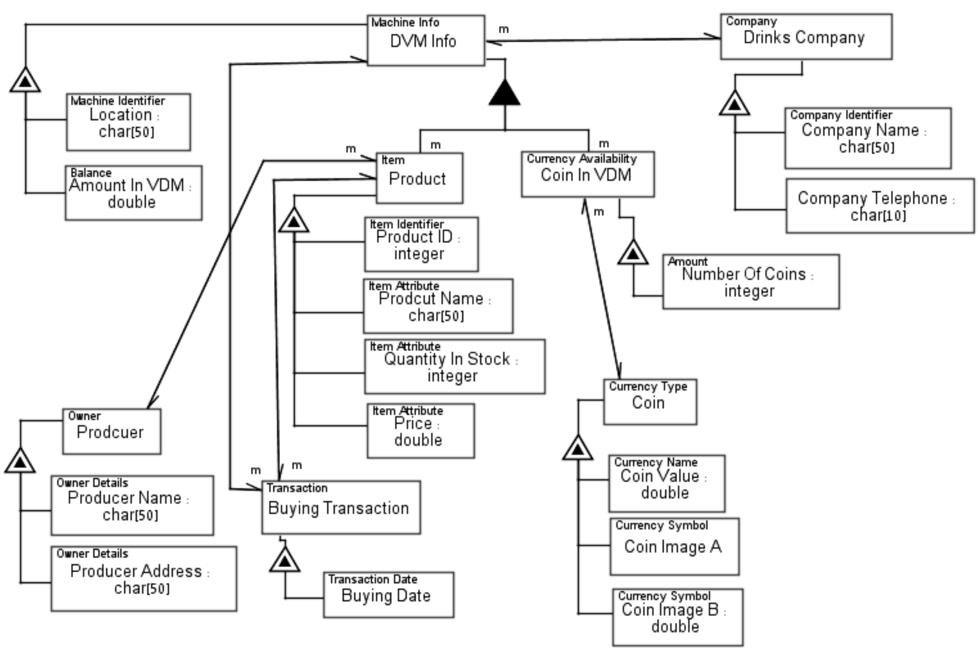


Fig. 6. DVM Info from the OPD in Fig. 5 unfolded.

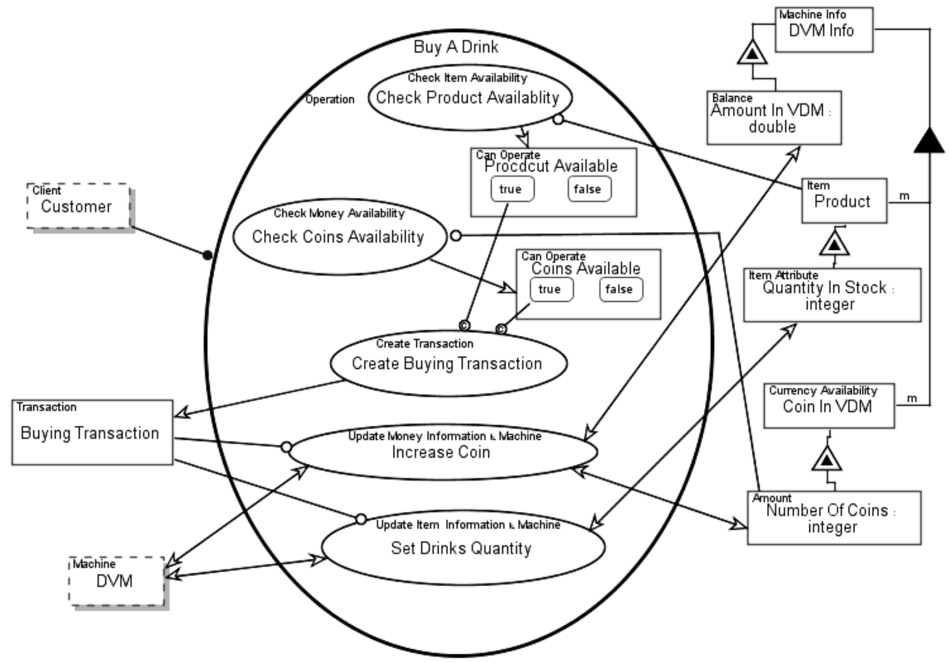


Fig. 7. The process Drink Buying in-zoomed.

The **Buy A Drink** process, which is in-zoomed in Fig. 7, conforms to the constraints specified in the domain model, as described in Fig. 4. Overall, the sequence of application processes follows the pattern specified in the domain model. The lack on an *Identification* process is acceptable, since it was specified as optional in the domain model.

4.2.2. *The Automatic Teller Machine application*

The ATM application manages several machines that belong to a bank. Each machine is identified by its location and the bank that owns it. The system stores the bank name. Each ATM recognizes and can handle several bill types. An account can be accessed using a customer card. When a customer wishes to withdraw money from the ATM, he or she is identified, the associated account eligibility is checked, and bills availability is determined. When the customer withdraws money, the system creates a transaction, updates the ATM and the account balance, and notifies the machine about the bills it needs to deliver. A bank clerk can fill the ATM with bills. The system diagram of the ATM application is depicted in Fig. 8.

Like the DVM, the ATM application model adheres to the Access Control domain model. In the ATM model, however, a non-domain process (a process not represented in the domain model), **Process Operation**, was added. This additional process does not violate the constraints defined by the domain model, hence

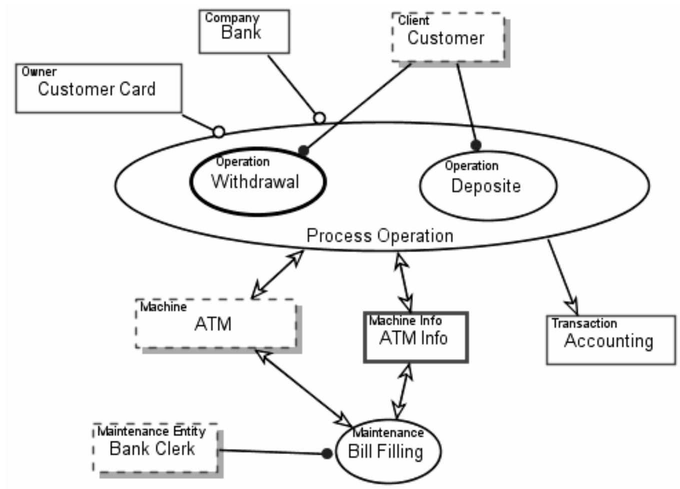


Fig. 8. System diagram of the Automatic Teller Machine.

it is valid. The links in the ATM model also conform with those in the domain model, as links attached to an aggregate process (e.g., **Process Operation**) are also linked to each internal process, as defined within the domain model, e.g., between **Owner** and **Operation**.

Using the AC domain OPD in Fig. 3 as a template and guidelines for modeling applications within the AC domain, **ATM Info** is unfolded in the OPD depicted in Fig. 9. The **Withdrawal** process, which is in-zoomed in Fig. 10, follows the constraints induced by the domain model, as specified in Fig. 4. Overall, the sequence of application processes follows the pattern specified in the domain model, including an *Identification* process, which was missing in the DVM application model.

4.3. *Validating the application model*

As noted above, a major benefit of a good domain analysis method is its ability to validate an application model with respect to its domain model. ADOM-OPM-based validation of an application against its domain model is performed in three steps: element reduction, element unification, and model matching as discussed in Sec. 2. In this section we demonstrate the validation process on the case study of the drink vending machine.

In the element reduction step, nothing is done since all elements within the application model are classified with their corresponding domain model elements.

Figures 11–13 present results of the element unification step in terms of the verifiable model. This model is created only for validation purposes and may even be syntactically incorrect. Yet, it fulfils its validation goals. This model serves as the vehicle for validating the application model against its corresponding domain model.

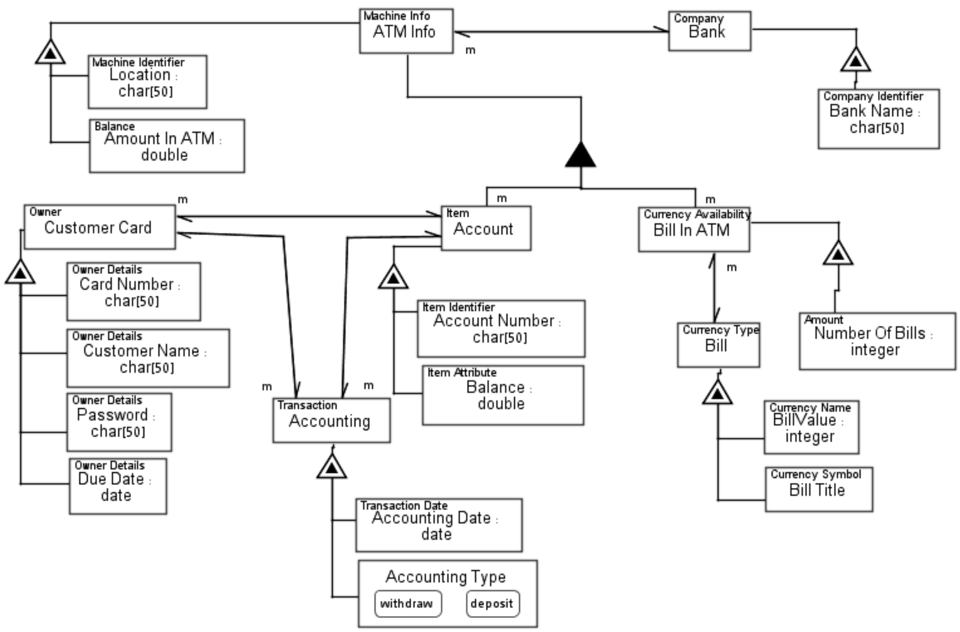


Fig. 9. ATM Info Unfolded.

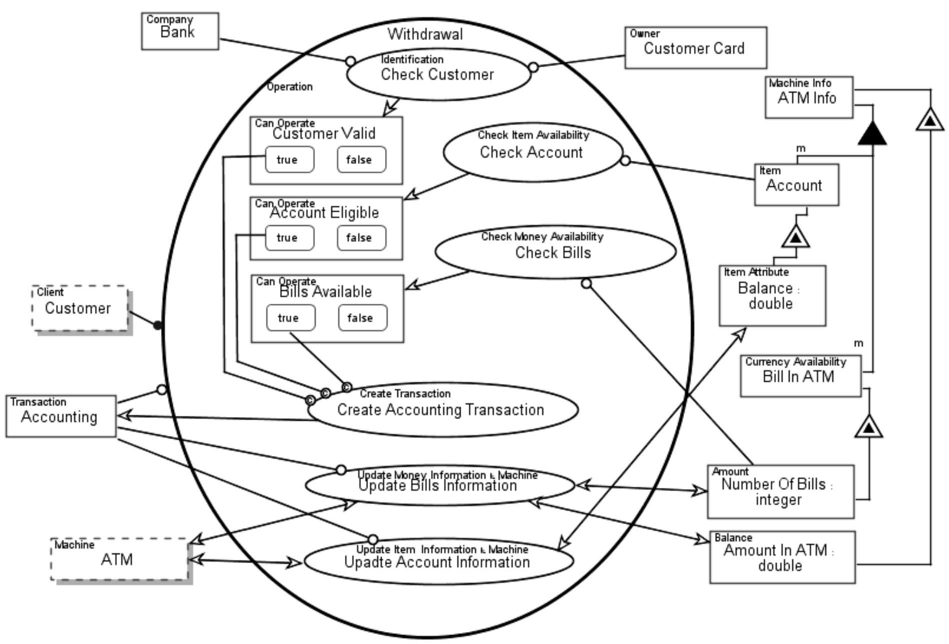


Fig. 10. Withdrawal process in-zoomed.

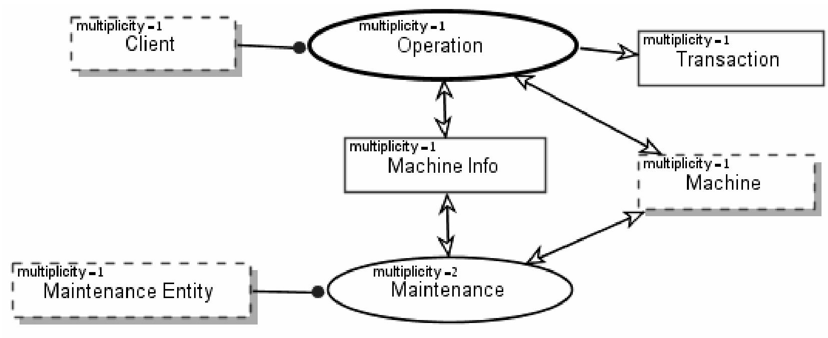


Fig. 11. System Diagram of the verifiable model for the Drink Vending Machine.

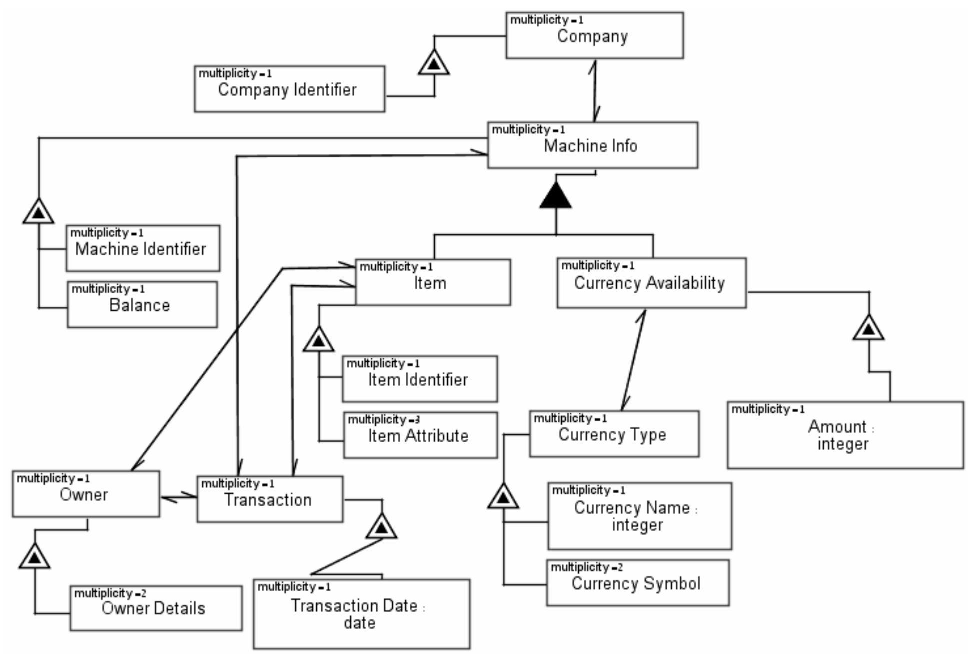


Fig. 12. Machine Info unfolded OPD of the verifiable model for Drink Vending Machine.

Following the matching algorithm provided in Sec. 2 these facts hold:

1. All elements in the verifiable model are termed as domain model elements.
2. For each element in the verifiable model, the boundaries of the actual multiplicity do not exceed the values of the multiplicity of the corresponding domain model element.
3. Each element in the domain model that does not appear in the verifiable model has minimal multiplicity (in the domain model) of 0. For example, the

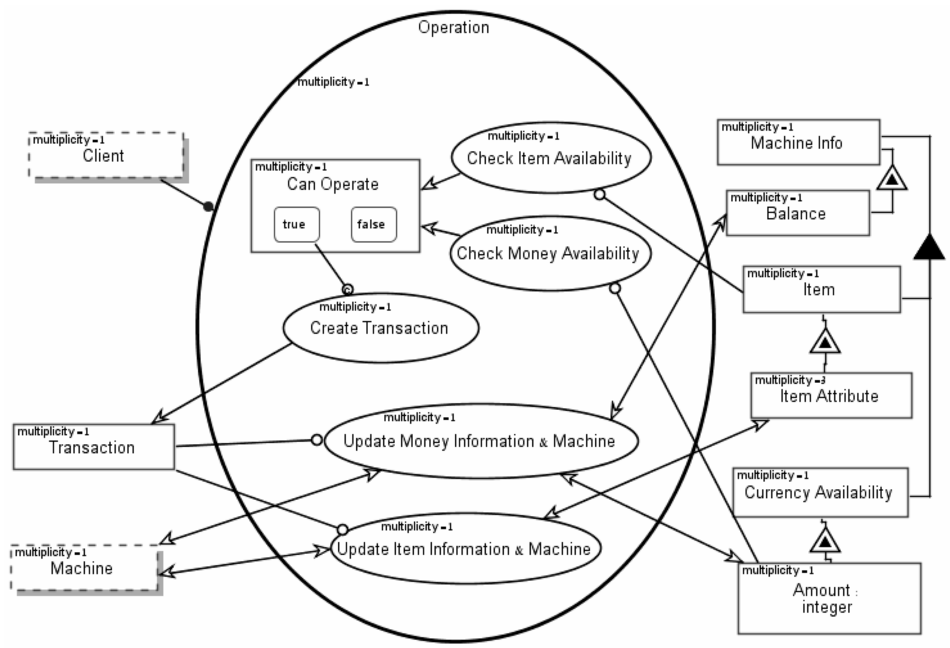


Fig. 13. Operate in-zoomed OPD of the verifiable model for Drink Vending Machine.

identification process does not appear in the DVM application model. As it has a minimal multiplicity of 0, the application model also adheres to the domain model regarding that process.

Analyzing these results, none of the specified constrains induced by the domain model are violated by the DVM verifiable model, implying that the DVM application model is in agreement with its AC domain model.

5. Evaluating ADOM-OPM

In order to evaluate ADOM-OPM, we conducted an experiment that compares it with "vanilla" OPM. The goal of the experiment was to determine whether application modeling that is based on and guided by a domain model improves the resulting application model compared with an application model that is developed without the support of a domain model. In this section, we present the experiment and its results.

5.1. *Experiment hypothesis*

The null hypothesis of our experiment was that an application model constructed using ADOM-OPM is not more complete (in terms of requirements coverage) and not more accurate (in terms of syntax and semantics of OPM models) than the model of the same system resulting from using OPM alone. The alternative hypothesis is that there is a significant difference between the completeness and accuracy of the models constructed with and without the domain models. Possible reasons for accepting this conjecture may be that the ADOM-OPM domain model provides a framework that guides the modeler in creating the application model within the domain of discourse and helps her avoid making mistakes that may occur without such guidance.

5.2. *Experiment settings*

The subjects of the experiment were 118 third-year students in a four-year engineering B.Sc. program at the Technion, Israel Institute of Technology, who took the course "Specification and Analysis of Information Systems" during the winter semester of the 2004–5 academic year. The students had no prior knowledge or experience in system modeling and specification. During the course, the students studied various modeling techniques, including Data Flow Diagram (DFD), UML, Statecharts, and OPM. The last lecture of the course was devoted to the ADOM approach and its applications in UML and OPM.

The experiment took place during the final examination of the course. The examination consisted of three problems, each relating to a different domain: (1) The Resource Allocation and Tracking (RAT) systems domain which refers to applications that register requests from outside sources, assign resources to resolve requests and inform interested client systems of the status; (2) the Process Control (PC) systems domain, which refers to applications that monitor and control the values of certain variables; and (3) the Access Control (AC) systems domain.

The RAT domain problem contained requirements calling for modeling of an elevator system that needs to allocate an elevator for a specific call. The PC domain problem required modeling of a water tank's filling control system and the AC domain problem called for modeling a drink vending machine akin to that described in Sec. 4.2.1. The requirements in all three problems were similar to those specified for the DVM application.

We prepared three different examination versions, labeled V1, V2, and V3, each with two problems. In each problem (and its respective domain) about half of the students were also provided with the OPM-based domain model, while the other half did not get it. This way, each version included one question with a domain model and one question without a domain model. The questions were checked and validated by three OPM and ADOM experts.

The students were divided arbitrarily into three groups, each responding to a different examination version. The distribution of students by the three examination questions and the three question domains is provided in Table 2, where the numbers of students who responded to each question in each version appear in parenthesis. Examining the student groups using the Kruskal-Wallis ANOVA test on students' cumulative average grade over their three years of study, we found no statistically significant differences between the groups ($H(2, N = 118) = 3.45$, $p = 0.177$).

Table 2. Students' distribution by examination version and question domain.

Question Domain \ Examination Version	V1	V2	V3
Resource Allocation and Tracking (RAT)	OPM (32)		ADOM-OPM (39)
Process Control (PC)		ADOM-OPM (38)	OPM (28)
Access Control (AC)	ADOM-OPM (41)	OPM (32)	

5.3. *Experiment results*

All the questions were graded by the course staff. Each one of the graders checked a question in one domain for all students according to a predefined set of criteria. Each question could score up to 34 points. Table 3 summarizes the average scores students achieved for each question in OPM and in ADOM-OPM. As no correlation was found among the questions within a single subject, we performed a t-test[c] to evaluate the differences between the achievements of students who were provided with the domain model in addition to the requirements, and those of students who did not get the domain model. This evaluation was done separately for each one of the three domains.

Table 3 clearly shows that using the ADOM-OPM, the students achieved better results than with OPM alone, and that these results are domain independent. Performing a mean comparison statistical analysis (i.e., t-test) we found that the differences between the two methods were significant for each domain separately. Hence we accept the alternative hypothesis regarding the benefits of modeling with ADOM-OPM compared with generic OPM modeling. Examining the results in detail we found out that the models done using ADOM-OPM scored better than models done with OPM alone in terms of the correct use of objects, processes, relations and links, and in terms of model completeness.

Table 4 presents the detailed analysis results. Each model was examined by four correctness and completeness criteria: objects, relations, processes, and links. Analyzing the results, we found out that the domain model helped identify mainly

Table 3. Average total scores for each domain with OPM and ADOM-OPM.

Method \ Domain	RAT	PC	AC
OPM	23.06	27.07	25.06
ADOM-OPM	**25.32**	**30.64**	**28.19**
Significance	$p < 0.01$	$p < 0.01$	$p < 0.02$

[c] A *t*-test is a statistical test of the null hypothesis that the means of two normally distributed populations are equal.

Table 4. Average scores for objects, relations, processes, and links.

Domain / Method	RAT				PC				AC			
	Objects	Relations	Processes	Links	Objects	Relations	Processes	Links	Objects	Relations	Processes	Links
OPM	12.02	**2.93**	4.72	3.39	13.52	3.18	3.41	6.96	13.33	1.72	6.36	3.66
ADOM-OPM	**13.1**	2.67	**5.27**	**4.28**	**16.2**	**3.29**	**4.03**	**7.13**	**14.76**	**2.15**	**6.76**	**4.54**
Significance	$p < 0.02$	$p < 0.11$	$p < 0.01$	$p < 0.02$	$p < 0.01$	$p < 0.63$	$p < 0.1$	$p < 0.52$	$p < 0.03$	$p < 0.03$	$p < 0.3$	$p < 0.01$

the objects and processes of the specific application. The drink vending machine problem (from the access control domain) is an exception with respect to process identification since the processes in this problem were described explicitly, so their identification was relatively easy. Consequently, no significant differences were found between the grades of the students who got the domain model and those who did not get the domain model.

Students who used the domain model also achieved better results with respect to relationships and links correctness and completeness, but these differences were not significant in two of the three cases. A possible reason for this lack of significant difference is that once the correct processes and objects have been identified, their associations are quite straightforward, so there is no need for extra help from the domain model. In one case, the RAT domain problem — the scores with respect the relationships were in favor of OPM alone rather than the ADOM-OPM. This might occur due to the fact that the correlation between the domain model and the application model was not easy to achieve.

Summarizing the experiment, we found out that ADOM-OPM had improved the resulting model, compared with a model that is constructed with OPM without the domain model support, and that the improvement is mainly due to better identification of the objects and processes in the model.

6. Discussion and Reference to Related Work

We have also examined the ADOM-OPM approach and compared it to current domain analysis approaches based on the criteria of specification of abstraction levels, variability management, application modeling guidelines, and validation of application models against their domain models. In what follows we discuss each criterion separately in ADOM-OPM as it relates to pertinent literature.

Specification of different abstraction levels: It is desirable for a domain analysis method to be able to define both a domain and applications in that domain. While a domain specification is usually generic in order for it to be able to support different types of domain-specific applications, the specification of an application within a domain must be more concrete and specific [17]. ADOM-OPM provides the ability to model the domain and application models using the same language at different abstraction levels. Within the domain and application models, ADOM-OPM provides abstraction capabilities utilizing OPM abstraction/refinement (scaling) mechanism. The ADOM-OPM specification of these two abstraction levels via the same method — OPM — is important for defining consistency, validation, and integrity constraints between domains and their applications.

Variability management: Some domain analysis approaches provide a generic architecture or model, which depicts only the commonality of the domain, ignoring variability. In such approaches, each application starts with the generic architecture and adapts it as required. Developers using such an approach are in a better starting point compared with developing a system from scratch without reuse, but it fails to capture knowledge about the variability in the domain. Such knowledge is one of the major factors that should be handled by a domain analysis approach [18]. Webber and Gomaa [19] identified four types of techniques for modeling variability:

(1) Parameterization: the application designer may change the values of attributes defined as parameters in the domain model;
(2) Information hiding: the application designer uses the same interface for defining similar components;
(3) Inheritance: the application designer may extend the interface of domain elements within a specific application and even override them; and
(4) Variation points definition: the application designer may create new variants and connect them to the variation points specified within the domain.

When referring to variation points, van Gurp and Bosh [20] made a distinction between closed and open variation points. Closed variation points consist of a predefined set of variants, from which the application designer can choose. Conversely, in open variation points, new variants that are not part of the domain specification can be introduced for an application model.

ADOM-OPM enables to specify an application using elements defined within the domain model with various multiplicities, as it is constrained by the domain model. This is the main means by which ADOM-OPM specifies its variability management. ADOM-OPM also enables adding new model elements, which were not defined within the domain model, as long as they do not violate the constraints defined by the domain model. These new elements are specific for the modeled application. This shows the ability of the ADOM-OPM to support open variation points. Commonality, on the other hand, is expressed by assigning a mandatory constraint (i.e., a multiplicity indicator of at least one) on a model element.

Application modeling guidelines: A domain analysis method is expected to provide guidelines for constructing an application model using relevant domain knowledge. These guidelines should refer to issues such as the relationships between a domain and its applications variability management [19], domain-specific constraints that should hold in an application model [20], decisions that need to be made during application construction [21], and selecting components for a specific application [22]. In ADOM-OPM, application modeling guidelines are provided by the domain model as it serves as a template for specifying application models.

Validation: The validation of an application against its domain specification is essential for reducing errors and faults, and consequently costs and difficulties. In this context, the domain specification enforces constraints on its applications [23]. It captures constraints to be followed by applications in the domain and possibly refers to variability among applications. Domain models enable checking that all relationships and constraints are maintained and fulfilled within the domain-specific applications [24]. A particular validation type is completeness checking, i.e., determining whether the application model includes all the (mandatory) elements required by the domain model. ADOM-OPM provides a validation algorithm to check the adherence of an application model to the domain model.

Comparing the ADOM-OPM approach with other current domain analysis approaches, we argue above that the proposed approach addresses all four criteria, while other approaches miss at least one of these criteria. *Architecture-based domain analysis* methods define the domain knowledge in components, libraries, or architectures. These various domain artifacts are reused in an application as they are, but they can also be modified to support the particular requirements at hand. Examples of this type of approach include that of Draco [25], a multiple view method for domain analysis presented in Meekel et al. [26], the Domain Analysis and Reuse Environment (DARE) [27], and a generic modeling technique that uses UML extensions for variability [28]. These architecture-based domain analysis approaches do not explicitly refer to commonality and variability within a domain, nor do they provide the designer with guidelines to support a specific application design. Rather, they allow him or her to select the relevant elements required by the designated application. The modeling of both domains and applications is done at the same level of abstraction, expecting the application designer to select relevant elements for the application model from the domain model. As a consequence, no validation is enforced by these methods. The lack of validation facilities as part of these architecture-based domain analysis methods undermines their value as domain analysis approaches.

Feature-oriented methods, such as FODA [22, 29] and PLUS [18] suggest that a system specification be derived by tailoring the domain model according to the features desired in a specific system. That is, a specific system reuses parts of a reusable architecture and instantiates a subset of features from the domain model. These methods enable modeling domains and applications at the same level of abstraction. They handle variability by means of parameterization, generalization, and implicit variation point definition. They provide guidance for the application designer of how to select the required features. However, they do not allow adding to an application new features which were not modeled within the domain, i.e., they support only the concept of closed variation points, narrowing the variety of applications suitable for a specific domain. These methods support validation by checking whether the feature constraints defined in the domain model hold in the specific application.

An approach similar to the feature-oriented one is that of Morisio *et al.* [30], who proposed an extension to UML that includes a special stereotype, indicating that a class may be altered within a specific system. This extension is demonstrated by applying it to UML class diagrams. This approach also uses only one level of abstraction in specifying the domain and its application. Thus, 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 associated classes, but not whether the class is correctly connected.

Metamodeling techniques for domain analysis enable definition of domains as metamodels that serve for both capturing domain knowledge and validating particular domain-specific applications. The validation rules that are induced by the metamodels enable avoiding syntactic and semantic mistakes during the initial stages of application development, reducing development time, and improving system quality. Examples for this type of approach are the studies by Schleicher and Westfechtel [31], Gomma and Eonsuk-Shin [32], and GME — the Generic Modeling Environment [33, 34]. Most of these methods use standard visual modeling languages, such as UML. Some of them enhance the visualization by constraint languages, such as the Object Constraint Language (OCL) [35].

The metamodeling methods discussed above support close variation points, guide the application designer in constructing applications, and provide validation facilities. Variability in these methods is managed using multiplicity constraints among business elements. However, these methods suffer from limited accessibility, as different jargons are used within the domain and application models, giving rise to "impedance mismatch," which arises from ambiguous, poorly defined translations between the domain and application models [36]. Moreover, while these metamodeling methods support modeling at two different abstraction levels, they focus on specifying structural elements, leaving out support for dynamic constraint specifications. Furthermore, specifying new element types are forbidden, thus when applying these methods the support of open variation points is missing.

7. Summary and Future Work

ADOM-OPM has been presented as an application domain modeling approach that extends Object-Process Methodology with a classification mechanism with two elements: roles, which are stereotypes-like elements, and multiplicity indicators. We demonstrated the use of ADOM-OPM by applying it to the domain of Access Control and two applications in this domain: the drink vending machine and the automatic teller machine. ADOM-OPM has been applied in several domains, including multi-agent systems, in which a new modeling language has been suggested for specifying this type of systems [37], discrete simulation events, and databases, in which database schemata can be easily generated including triggers and stored procedures.

To evaluate ADOM-OPM, we examined it via a controlled experiment and established that it helps create better models than those obtained using OPM alone. Analyzing the empirical results and the theoretical aspects, we have concluded that the ADOM-OPM approach addresses the following problems.

1. The multiplicity aspect problem: ADOM-OPM by its nature supports both static and dynamic aspects of the domain and application models.
2. The multiple view problem: OPM supports system specification in a single, unifying view, or diagram type. Since a domain is modeled just like an application within that domain, domain modeling benefits from all the advantages of OPM, including its single view, the combination of formal and intuitive model presentation, and the bimodal graphic-textual representation.

3. The domain-application relationship problem: The ADOM-OPM approach utilizes the domain model while modeling the application in two ways: (1) classifying the application model entities with roles defined in the domain model, and (2) validating the relationships among the application model elements (entities and links) according to their classifying roles and link constraints defined in the domain model.
4. The models incompatibility problem: both the domain and the application OPM models use the same notations and semantics, eliminating the need for mental model transformations.

Moving forward from domain analysis, domain design in OPM is similar to domain analysis, as it employs the same terminology while deepening the level of details and shifting the focus from the problem area to the solution area. The transformation to domain implementation can be done using the Generic Code Generator (GCG) [38] associated with Object-Process CAse Tool (OPCAT) [39]. Utilizing the GCG and roles within a domain can be a basis for developing infrastructure components and using them to generate application code.

The implementation of the ADOM-OPM analysis approach is currently being integrated into OPCAT. We also plan to add negation constraints and extend the application model so that it can incorporate more than one domain model. For evaluation purposes, we intend to experimentally compare the ADOM-OPM approach with ADOM-UML approach and other domain analysis approaches.

References

1. Carnegie Mellon — Software Engineering Institute, *Domain Engineering: A Model-Based Approach*, http://www.sei.cmu.edu/domain-engineering/, 2002.
2. C. Cleaveland, *Domain Engineering*, http://craigc.com/cs/de.html, 2002.
3. A. Valerio, G. Succi, and M. Fenaroli, Domain analysis and framework-based software development, *Applied Computing Review* **5**(2) (1997) 4–15.
4. K. Czarnecki and U. W. Eisenecker, *Generative Programming — Methods, Tools, and Applications* (Addison-Wesley, Reading, 2000).
5. A. Sturm and I. Reinhartz-Berger, Applying the application-based domain modeling approach to UML structural views, in *Proc. 23rd Int. Conf. Conceptual Modeling* (*ER'2004*), LNCS 3288, 2004, pp. 766–779.
6. I. Reinhartz-Berger and A. Sturm, Behavioral domain analysis — The application-based domain modeling approach, in *Proc. 7th Int. Conf. Unified Modeling Language* (*UML'2004*), LNCS 3273, 2004, pp. 410–424.
7. D. Dori, *Object-Process Methodology — A Holistic Systems Paradigm* (Springer Verlag, 2002).
8. D. Dori, Representing pattern recognition-embedded systems through object-process diagrams: the case of machine drawing understanding system, *Pattern Recognition Letters* **16**(4) (1995) 374–384.
9. D. Dori, Object-process analysis of computer integrated manufacturing documentation and inspection, *Int. J. Computer Integrated Manufacturing* **9**(5) (1996) 339–35.

10. I. Reinhartz-Berger, S. Katz, and D. Dori, OPM/Web — Object-process methodology for developing web applications, *Annals on Software Engineering — Special Issue on OO Web-based Software Engineering* (2002) 141–161.
11. M. Peleg and D. Dori, The model multiplicity problem: experimenting with real-time specification methods, *IEEE Trans. Software Engineering* **26**(8) (2000) 742–759.
12. I. Reinhartz-Berger and D. Dori, OPM versus UML — Experimenting comprehension and construction of web application models, *Empirical Software Eng. J.* **10**(1) (2005) 57–80.
13. K. Siau and Q. Cao, Unified modeling language: A complexity analysis, *J. Database Management* **12**(1) (2001) 26–34.
14. OMG-MOF, Meta-Object Facility (MOF$^{\text{TM}}$), version 1.4, 2002.
15. M. Peleg and D. Dori, Extending the object-process methodology to handle real-time systems, *J. Object-Oriented Programming* **11**(8) (1999) 53–58.
16. D. J. Duffy, *Domain Architectures: Models and Architectures for UML Applications* (John Wiley & Sons, 2004).
17. J. D. McGregor, Software product lines, *J. Object Technology* **3**(3) (2004) 65–74.
18. H. Gomma, *Designing Software Product Lines with UML* (Addison-Wesley Professional, 2004).
19. D. L. Webber and H. Gomaa, Modeling variability in software product lines with variation point model, *Science of Computer Programming* **53** (2004) 305–331.
20. J. Van Gurp, J. Bosch, and M. Svahnberg, On the notion of variability in software product lines, in *Proc. Working IEEE/IFIP Conf. Software Architecture* (*Wicsa'01*), 2001, pp. 45–54.
21. K. Schmid and I. John, A customizable approach to full lifecycle variability management, *Science of Computer Programming* **53** (2004) 259–284.
22. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, FORM: A feature-oriented reuse method with domain-specific reference architectures, *Annals of Software Engineering* **5**(1) (1998) 143–168.
23. E. Addy, A framework for performing verification and validation in reuse-based software engineering, *Annals of Software Engineering* **5**(1) (1998) 279–292.
24. P. Padmanabhan and R. R. Lutz, Tool-supported verification of product line requirements, *Automated Software Engineering* **12**(4) (2005) 447–465.
25. J. Neighbors, Draco: A method for engineering reusable software systems, in *Software Reusability. Volume I: Concepts and Models*, ed. T. Biggerstaff and A. Perlis (Addison-Wesley, Reading, 1989), pp. 295–319.
26. J. Meekel, T. B. Horton, R. B. France, C. Mellone, and S. Dalvi, From domain models to architecture frameworks, in *Proc. 1997 Symposium on Software Reusability*, 1997, pp. 75–80.
27. W. Frakes, R. Prieto-Díaz, and C. Fox, DARE: Domain analysis and reuse environment, *Annals of Software Engineering* **5**(1) (1998) 125–141.
28. M. Clauss, Generic modeling using UML extensions for variability, in *Proc. Workshop on Domain Specific Visual Languages, Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA'01*), 2001, pp. 11–18.
29. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, CMU/SEI-90-TR-021 ADA235785, 1990.
30. M. Morisio, G. H. Travassos, and M. Stark, Extending UML to support domain analysis, in *Proc. Fifth IEEE Int. Conf. Automated Software Engineering*, 2000, pp. 321–324.

31. A. Schleicher and B. Westfechtel, Beyond stereotyping: Metamodeling approaches for the UML, in *Proc. 34th Annual Hawaii Int. Conf. System Sciences*, 2001, pp. 1243–1252.
32. H. Gomma and M. Eonsuk-Shin, Multiple-view meta-modeling of software product lines, in *Proc. Eighth IEEE Int. Conf. Engineering of Complex Computer Systems*, 2002, pp. 238–246.
33. J. Davis, Model integrated computing: A framework for creating domain specific design environments, in *Proc. Sixth World Multiconference on Systems, Cybernetics, and Informatics* (*SCI*), 2002.
34. G. Nordstrom, J. Sztipanovits, G. Karsai, and A. Ledeczi, Metamodeling — Rapid design and evolution of domain-specific modeling environments, in *Proc. IEEE Sixth Symp. Engineering Computer-Based Systems* (*ECBS*), 1999, pp. 68–74.
35. J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML* (Addison-Wesley, 1998).
36. J. Evermann and Y. Wand, Toward formalizing domain modeling semantics in language syntax, *IEEE Trans. Software Engineering* **31**(1) (2005) 21–37.
37. A. Sturm, D. Dori, and O. Shehory, Single-model method for specifying multi-agent systems, in *Proc. Second Int. Joint Conf. Autonomous Agents and MultiAgent Systems*, 2003, pp. 121–128.
38. I. Reinhartz-Berger and D. Dori, Object-Process Methodology (OPM) versus UML: A code generation perspective, in *Proc. CAiSE Workshops*, 2004, pp. 275–286.
39. D. Dori, I. Reinhartz-Berger, and A. Sturm, OPCAT — A bimodal CASE tool for object-process based system development, in *Proc. IEEE/ACM 5th Int. Conf. Enterprise Information Systems* (*ICEIS '03*), 2003, pp. 286–291. Academic version is downloadable from www.opcat.com.