International Workshop on

Domain Engineering

DE@CAiSE'2010

Hammamet, Tunisia, 8 June 2010

In conjunction with the CAISE'10
22nd International Conference on Advanced Information Systems Engineering

Workshop Proceedings

Editors

DE@CAiSE'2010 Chairs

Iris Reinhartz-Berger
University of Haifa, Israel

Arnon Sturm
Ben Gurion University of the Negev, Israel

Yair Wand
University of British Columbia, Canada

Jorn Bettin
Sofismo, Switzerland

Tony Clark
Thames Valley University, UK

Sholom Cohen
Software Engineering Institute, Carnegie Mellon University, USA

CAiSE'10 Workshop Chairs

Jolita Ralyté
University of Geneva, Switzerland

Pierluigi Plebani
Politecnico di Milano, Itally

Workshop DE@CAiSE'2010

Proceedings

This volume contains the original articles presented at the International Workshop on Domain Engineering – DE@CAiSE'2010. The workshop was held in conjunction with the 22nd International Conference on Advanced Information Systems Engineering, in Hammamet, Tunisia, June 8, 2010.

CEUR Workshop Proceedings, CEUR-WS.org, ISSN 1613-0073.

Preface

Domain Engineering, also referred to as product line engineering, deals with developing reusable assets that can be adjusted and adapted to families of applications, rather than to particular systems. A domain in this context can be defined as an area of knowledge that uses common concepts for describing phenomena, requirements, problems, capabilities, and solutions. The purpose of domain engineering is to identify, model, construct, catalog, and disseminate artifacts that represent the commonalities and differences within a domain, as well as to provide mechanisms, techniques, and tools to reuse these artifacts in the development of particular applications and systems.

Although being applicable to different engineering disciplines, domain engineering methods and domain specific languages (DSL) receive nowadays special attention from the information systems and software engineering communities who deal with artifact reuse, application validation, and domain knowledge representation: different kinds of reuse mechanisms, such as customization, configuration, specialization, and template instantiation, are introduced; ways to capture and manage variability are developed; and guidelines for creating consistent and correct applications and systems in certain domains are emerged. The aims of all these up-and-coming methods and techniques is to help reduce time-to-market, product cost, and projects risks on one hand, and help improve product quality and performance on a consistent basis on the other hand.

As an interdisciplinary field, domain engineering deals with various topics such as conceptual foundations, semantics of domains, development and management of domain assets, lifecycle support, variability management, consistency validation, and theoretical and empirical evaluation of domain engineering techniques. The purpose of this series of workshops is to bring together researchers and practitioners in the area of domain engineering in order to identify possible points of synergy, common problems and solutions, and visions for the future of the area. In particular, the specific workshop focuses on the use of domains for improving development processes in these domains.

The workshop will start with an invited talk entitled "Domain Engineering: What is it?" and given by Arne Sølvberg. This talk will be followed by 4 accepted papers dealing with domain semantics and profile-based development:

Domain Semantics:

1. Wolf Fischer and Bernhard Bauer, Domain Dependent Semantic Requirement Engineering.

Profile-based Development:

2. Jugurta Lisboa-Filho, Gustavo Breder Sampaio, Filipe Ribeiro Nalon, and Karla A. de V. Borges, A UML Profile for Conceptual Modeling in GIS Domain.
3. Saoussen Rekhis, Nadia Bouassida, Rafik Bouaziz, Bruno Sadeg. A UML-Profile for domain specific patterns: Application to real-time

4. Oded Kramer and Arnon Sturm, Bridging Programming Productivity, Expressiveness, and Applicability: a Domain Engineering Approach.

Iris Reinhartz-Berger, Arnon Sturm, Yair Wand,
Jorn Bettin, Tony Clark, and Sholom Cohen
DE@CAiSE'2010 Organizers

For more information on the workshop, see our website http://www.domainengineering.org/, or contact Iris Reinhartz-Berger (iris@mis.haifa.ac.il) or Arnon Sturm (sturm@bgu.ac.il)

Organization

DE@CAiSE'10 Organizers

Iris Reinhartz-Berger University of Haifa, Israel	Arnon Sturm Ben Gurion University of the Negev, Israel	Yair Wand, University of British Columbia, Canada
Jorn Bettin Sofismo, Switzerland	Tony Clark Thames Valley University, UK	Sholom Cohen, Software Engineering Institute, Carnegie Mellon University, USA

Program committee

Colin Atkinson	University of Mannheim, Germany
Mira Balaban	Ben-Gurion University of the Negev, Israel
Balbir Barn	Middlesex University, UK
Jorn Bettin	Sofismo, Switzerland
Tony Clark	Thames Valley University, UK
Sholom Cohen	CMU-SEI, USA
Kim Dae-Kyoo	Oakland University, USA
Joerg Evermann	Memorial University of Newfoundland, Canada
Jeff Gray	University of Alabama, USA
Atzmon Hen-Tov	Pontis, Israel
John Hosking	University of Auckland, New Zealand
Jaejoon Lee	Lancaster University, UK
David Lorenz	Open University, Israel
John McGregor	Clemson University, USA
Klaus Pohl	University of Duisburg-Essen, Germany
Iris Reinhartz-Berger	University of Haifa, Israel
Michael Rosemann	The University of Queensland, Australia
Julia Rubin	IBM Haifa Research Labs, Israel
Bernhard Rumpe	Braunschweig University of Technology, Germany
Lior Schachter	Pontis, Israel
Klaus Schmid	University of Hildesheim, Germany
Keng Siau	University of Nebraska-Lincoln, USA
Pnina Soffer	University of Haifa, Israel
Il-Yeol Song	Drexel University, USA
Arnon Sturm	Ben Gurion University of the Negev, Israel
Juha-Pekka Tolvanen	MetaCase, Finland
Yair Wand	University of British Columbia, Canada
Gabi Zodik	IBM Haifa Research Labs, Israel

Additional reviewers

Ingo Weisemoeller, RWTH Aachen, Software Engineering Group, Germany
Kim Lauenroth, University of Duisburg-Essen, Germany
Ornsiri Thonggoom, Drexel University, USA

Table of Contents

Domain Engineering: What is it?

Arne Sølvberg

Department of Computer and Information Science
NTNU – The Norwegian University of Science and Technology
7491 Trondheim, Norway
arne.solvberg@idi.ntnu.no

Abstract The term *domain engineering* has different meanings. In software engineering it is used for describing the software relevant features of the domain for which software is developed. In general the term denotes the act of creating the domain itself, including its artifacts. The paper argues that the use of the term in software engineering is not a happy choice and contributes to the terminological confusion often observed when disciplines meet in multi-disciplinary projects.

Keywords: information systems engineering, domain engineering

1 Introduction

When computers first came around, it made sense to distinguish between a domain and its computer applications. Around 40-50 years the so-called "Scandinavian School" of Information Systems made clear distinctions between the "total system", the "information system", and the "data system", the latter consisted of computer hardware and software.

These were pristine times, and it was simple to make such a distinction. An example of a "total system" could be a bank, with its vaults for the coins and bills, the "information system" would be the people and machines that exchange messages. A major task of the information system was to make sense of the numbers and texts that reflected the operation of the bank. The "data system" was the computers, the software and the data which could be stored and processed according to predetermined rules. It was a comparatively simple world.

As the years have gone by, and computers are everywhere, this simple distinction is no longer straightforward.

2 Domain layer and application layer, is this an appropriate distinction?

The call-for-papers explains the relationship between software and non-software as follows:

I. Reinhartz-Berger, A. Sturm, Y. Wand, J. Bettin, T. Clark, S. Cohen, J. Ralyté, and P. Plebani (Eds.):
CAiSE 2010 Workshop DE@CAiSE'10, Hammamet, Tunisia, pp. 1-5, 2010.

Domain engineering deals with two main layers: the domain layer, which deals with the representation of domain elements, and the application layer, which deals with software applications and information systems artifacts. In other words, programs, applications, or systems are included in the application layer, whereas their common and variable characteristics, as can be described, for example, by patterns, ontology, or emerging standards, are generalized and presented in the domain layer.

I find this distinction to be somewhat disturbing and potentially misleading. Man-made systems are always composed of parts. Most parts in our technical world increasingly consist of interrelated software and non-software. Take for example an automobile. The brakes consist of mechanical components and of process control software. So do other parts, like the fuel injection system, the power transmission system and many more. A car is an assembly of parts where many of them have very clear autonomous features. The various parts are made to interact through a combination of software and mechanical connections. How can one distinguish between a domain layer and an application layer for the car, when every component of the car is a system of both mechanical parts and software parts?

Approximately 50 % of the production cost of a modern automobile reflects the cost of electronics and software. So what is the domain and what is the (software) application? Does this distinction make sense?

The term domain engineering is used in contemporary software engineering. One example of a definition is "*Domain Engineering* is seen as a process for creating a family of programs so that programs in the family can be created efficiently" [3]. So domain engineering is seen as a technique for creating components with a wider applicability than the components would have when engineering makes one-of-a-kind solutions. The use of the term is seen more clearly in [4], where domain engineering is used in the meaning of reflecting software relevant features of a domain for the purpose of building software to be reusable for a wider market.

The use of the term *domain engineering* in this context is a typical example of picking a general term and applying it in a restricted setting. Such practices change the meaning of terms and lead to confusion and difficulties when communicating over discipline borders. A somewhat wider definition is given by [5], but the restriction to the software perspective is evident also here.

3 "Domain engineering" - old wine in new bottles?

Domain engineering has always been there, as long as people have built artifacts to satisfy human wishes and desires. The design of a road is domain engineering, as is the design of a business organization. The term "domain engineering" in the current context comes out of information technology and software engineering. The question is whether domain engineering in this restricted context means the same as domain engineering in the wider context. I propose that this is not the case and that this discrepancy leads to confusion.

If domain engineering means the engineering of a domain this is the same as the engineering of the total system of which the information system and the associated

software are parts. This is very different from using the term domain engineering for describing the external properties of some part of the domain for the purpose of designing re-usable software solutions for this part of the domain.

An information system is usually seen as giving support to some other system, by keeping track of its state-of-affairs, by supporting the exchange of information between the other system and its environment, and by providing information needed for changing the behavior of the other system, either through direct intervention or through making information available for other change agents [1]. In general, "the other system" is known by many different names, e.g., the user system, the user domain, the Universe of Discourse (UoD), the real world, the business system.

The term *domain engineering* when used in this context implies that the software based information system and its domain should be seen as two different entities? But it is increasingly difficult to separate these two parts of "the total system" through all system layers, and collect the software parts in one bundle and the non-software parts in another. So, can domain engineering exist on its own, and be separated from information system engineering?

4 Traditional approaches to information systems engineering in organizations

Implicit to the traditional approaches to information systems engineering is that there is a primary system, the domain system, which is to be served by a secondary information system. Most of the practice of information systems engineering is done in the domain of administrative organizations, e.g., bank, insurance, public services. The information systems have been so central to the design of the administrative routines that it has been difficult to distinguish between the two.

The information services are to be determined by the needs of the primary system. So, the first step in designing an information system is to do a requirement analysis. The next step is to find out whether the requirements can be satisfied. This is done by developing concrete solution proposals, and evaluating those relative to the requirements. Traditional approaches recognize that solution proposals and requirements must be co-developed. The solution proposals will give rise to modifications in the requirements, and to modifications of the features of the primary system. So, domain engineering is implicit in the traditional approach to information systems engineering.

Traditional approaches to total systems realization are based on

- application platforms
- software- and process libraries
- application languages

These have been with us in different shapes and quality since we started to use computers.

The ERP's are among the most successful application platforms. Almost all sizable companies depend on an ERP. Process libraries are among the most important tools for consultancy companies who mostly make their profit by re-using solutions for every new customer. Software libraries have been with us from the start and

provide for the reuse of software. Application specific languages have enjoyed less success.

Characteristic for all these approaches are that the tools that they apply reflect knowledge about the domain, that the tools limit the functional and structural properties of the "total system" through their own limitations to treat data, and therefore lead to a design process where the limited functional properties of the information processing software platforms, - libraries, and languages, provide a limited solution space for the design of "the total system".

5 Domain modeling in multidiscipline oriented approaches

Computer science is probably one of the few disciplines that cannot render useful results unless being related to another discipline. In many cases of traditional information systems engineering the computer part is large compared to the rest of the application system, and the modelling discipline of computer science dominates. For other situations there is more of a balance between the importance of domain disciplines and the IT discipline. Relevant domains are everywhere, in the material world, the biological world, and the worlds of organised and creative humans.

In practical system development cross-competence cooperative activities relate the IT core technologies to the application areas. Different modelling cultures meet, and sometimes they clash. We need to better clarify the relationship between the IT as a modelling discipline, and the modelling disciplines of the domains where IT is applied. We need a better framework for thinking about cross-competence systems design.

6 Conclusion: On the need and possibility of model integration

Computers are increasingly found everywhere, in almost every artifact, in the background of almost every organized human activity. This will have to result in a change in approach, from viewing the role of IT to mainly support "the other system", to become an integral part of "the other system". We should search for ways to avoid treating the domain discipline separate from the information system discipline, towards the integration of IT concepts, tools and theory into the modeling theories of the supported (domain) disciplines. If we manage this the two layers of domain engineering and application engineering may merge.

Fortunately the basic modeling concepts of the "domain sciences" and of information technology overlap. Information systems are concerned with data that represent facts in a Universe of Discourse. A fact is what is known -or assumed - to belong to reality. In science and technology one usually distinguishes the following kinds of facts: state, event, process, phenomenon, and concrete system e.g., a magnetic field [2]. We see that the basic modeling ontology is the same in the sciences as for IT. This gives some reason for optimism.

Because information technology provides component solutions to almost every other discipline we experience increasing fragmentation pressures on the discipline of IT itself. Every domain where IT is used seems to contain seeds for creating their own

kind of discipline where IT is integrated with the domain specific knowledge. We often see labeling like, e.g., medical informatics, organizational informatics, and industrial informatics. And we sometimes see that common IT knowledge is reinvented in new application settings. Harmonization of domain modeling ontology with IT modeling ontology may be one approach to better sharing of IT knowledge among the various disciplines thus avoiding massive “wheel-re-invention”.

References

1. Boman, M.; Bubenko, J.A. jr.; Johannesson, P.; Wangler, B.: Conceptual Modelling. Prentice Hall, 1997,269 p.
2. Bunge, M: The Philosophy of Science, Transaction Publishers, 1998, ISBNN 0-7658-0415-8
3. http://craigc.com/cs/de.html Accessed 23 April 2010
4. http://www.domain-specific.com/ Accessed 23 April 2010
5. http://burks.brighton.ac.uk/burks/foldoc/76/33.htm Accessed 23 April 2010

Domain Dependent Semantic Requirement Engineering

Research-In-Progress

Wolf Fischer and Bernhard Bauer

Programming Distributed Systems Lab, University of Augsburg, Germany
[wolf.fischer|bauer]@informatik.uni-augsburg.de

Abstract. Requirements Engineering is one of the most important phases in any product development life cycle since it is the basis for the complete software or product design and realization. Therefore in a worst case scenario any error within a requirement can result in a project loss. Computer support for requirement engineers is still premature as most RE applications can not cope with more sophisticated techniques like semantics, natural language processing etc. In this paper we will present the ongoing research work in semantic requirement engineering which will try to close the gap between computer understandable semantics, namely ontologies, and the natural language input of a human.

1 Introduction

Requirements engineering is known to be one of the most important phases in the development of a product. Correctly acquired requirements can lead to a solid and fluid creation of the to be developed artifacts whereas incorrect, contradictionary or incomplete requirements can result in a complete project loss in a worst case scenario (e.g. 63% of all errors in the medical sector result from the requirement engineering phase [4]). In the past different solutions have been proposed to treat these problems. They can basically be classified in two categories: Informal and strongly formal ones. Informal systems are such which rely on natural language and somehow try to cope with the existing problems (the majority of requirements is written in natural language [11]). Their advantages are the overall simple usability, at least in the beginning. However, especially in larger projects the increasing complexity makes it difficult to cope manually with requirements documents written in natural language. Strong formal systems rely on the specification of requirements using logical formulas having the advantage that they can be analyzed and verified by computers. Their main drawback is that on the business level most of the people are not familiar using logical formulae. Moreover it is a complex and time-consuming undertaking. Some approaches have tried to combine the advantages of both worlds and created ways of capturing requirements in models (e.g. i* [17] and UML [12]). Although these

I. Reinhartz-Berger, A. Sturm, Y. Wand, J. Bettin, T. Clark, S. Cohen, J. Ralyté, and P. Plebani (Eds.):
CAiSE 2010 Workshop DE@CAiSE'10, Hammamet, Tunisia, pp. 6-17, 2010.

techniques are easier to understand than logical formalisms there is still the problem of many humans adapting to new technologies. Therefore the system has to adapt to humans, i.e. the system has to be able to cope with text in natural language, extract relevant information from it and use it to support the user. Such a system could be used not only by professional requirement engineers but also by customers to identify their intentions and guide them to find previously gathered solutions automatically, as we will show in this paper.
We describe a highly integrated approach to requirements engineering, combining semantic technologies and NLP for requirements engineering. The rest of the paper is structured as follows: In section 2 the related work relevant to our approach is described. Next we present the overall approach to this topic in section 3. Section 4 exemplary shows how the presented concept works by showing it on a small example. Next an outlook on future work as well as a discussion on the possibilities and boundaries of this approach is outlined in section 5. Finally, section 6 concludes the paper.

2 Related Work

For treating requirements in a formal manner there have been different approaches presented in the past. In [17], Yu used i* to model requirements and reason on them. This approach however does not cope with the textual description of requirements and how one can come from this informal representation to a more formal one. Many different approaches use the advantage of ontological technologies but did not close the gap between a textual description of the requirements and the semantic representation: Lee and Ghandi [9] developed an ontology acting as a common language for all stakeholders of a domain. It allows the modelling of different viewpoints, goals and the requirements itself. Dobson et al. ([5]) presented an ontology for describing non-functional requirements.
However some approaches tried to combine semantic technologies with NLP in the requirement engineering field. Kof ([7]) relied on ontology extraction from text without additional domain information in the beginning. His main goal is to give the user some hints on how to write more precisely. However, there is no consequent combination of semantics and syntax. In [13], the NLP analysis was done without semantic information at hand based on a clustering approach. Thus problems like resolving synonyms remain.
Lohmann et al. [10] developed a semantic wiki for requirement engineering, especially for a large number of participants. In this case the semantics are based on the connections between different requirements (each requirement has its own URI). A deeper semantic representation of requirements is possible, as every term within a text can be linked to another wiki page. However there is no automatic mechanism to annotate a requirement semantically.
The market offers many different tools for eliciting requirements. The most prominent one is probably IBM Rational Doors[1]. It allows an efficient, hierarchical management of textual requirements as well as creating dependencies

[1] http://www-01.ibm.com/software/awdtools/doors/productline/

between them. Requirements can be annotated with attributes like priority etc., but there is no semantic annotation of the textual descriptions.

2.1 NLP and Construction Grammars

Construction grammars are a field which originated from cognitive linguistics. It is based on the idea that humans process the syntax and semantics in parallel, i.e. language can not be understood by separating both. This is also called the Syntax-lexicon Continuum (see [3] and [8] for more information). This concept is central to every construction grammar and differentiates it from state-of-the-art NLP concepts, which mostly rely on pipelined approaches (i.e. first the syntax is being analyzed and afterwards the semantics). There have been different experiments which support the idea of an inseparability of syntax and semantics in human language. Computational approaches have been created in the past, namely Fluid Construction Grammars (short FCG, for more see [14], [15] and [16]) and Embodied Construction Grammars (short ECG, for more see [1]). The goal of FCG is to create a linguistic formalism which can be used to evaluate how well a construction grammar approach can handle open-ended dialogues. To evaluate the approach it has been implemented within autonomous, embodied agents. Some of the key assumptions of FCG are that it is usage-based (inventories are highly specialised), the constructions are bi-directional (i.e. FCG can handle parsing as well as production of language), it uses feature structures (which are directly incorporated within the constructions) and there is also a continuum between grammar and lexicon. The production as well as parsing process is handled by a unify and merge algorithm, which allows for an emergent creation of either the semantics or the syntax using best-match-probabilities in the unification of the constructions. This leads to a high robustness of the algorithm and more natural creation of language.
ECG follows another goal by evaluating how embodiment affects language, i.e. how the human body, its shape, movement, etc. affects the mind and therefore language itself.

3 Concept

Our approach allows the combination of semantic and linguistic information as well as the analysis of corresponding text in natural language. Figure 1 shows a schematic overview of our approach. At the beginning the text is split into its single terms which are then analyzed according to constructions. Constructions combine the syntactic (language knowledge) and the semantic information of the domain ontology. The result of the process is an interpretation of the text.

3.1 Requirements Ontology

In order to identify concepts which are relevant to requirements engineering we are currently developing a requirements ontology which is able to capture the

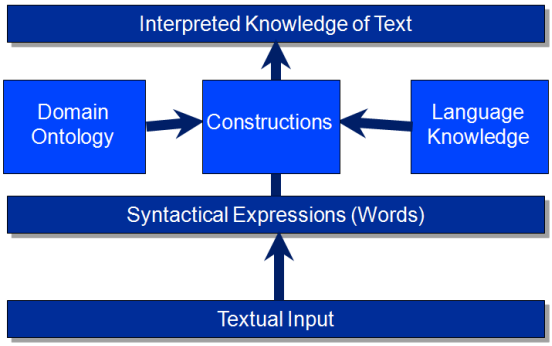

Fig. 1. Big picture of the analysis process

life cycle of requirements, i.e. from early requirements (e.g. a customer request) over the resulting business requirements to the delivered product. The main concepts of the requirements ontology can be seen in figure 2. An **AbstractRequirement** is anything which can contain a requirement, a wish or a problem description. There are further specializations like a **Request** (intended for initial user requests) or a 'typical' **Requirement** (which is further endetailed by a **NonFunctionalRequirement** and a **FunctionalRequirement**). Each requirement is created by a specific **Stakeholder**. Examplatory a Stakeholder can also be a **User** or an **ExternalStakeholder**. There are multiple other types of stakeholders possible, but are not described in detail here because of lack of space. Information about the competence of the user can be gathered by the **UserCompetence**. Finally, specific Stakeholders are responsible for a requirement which is indicated by the 'isResponsibleFor' association. To describe a requirement it contains a **RequirementDescription**. This one references different **RequirementResources**. A RequirementResource is a very generic concept which can represent anything domain specific. In the figure there are four specializations, i.e. the **Product** (which a requirement is about), an **Event** (e.g. to specify a certain point when a problem occurs), a **Document** (which is e.g. the source of the requirement) or a **System** (which could also act as a requirement source).

3.2 Linguistic Knowledge

This requirements ontology serves as a reference for the application and the ontology engineer: The application 'knows' the type of certain domain specific information (e.g. that the CEO of the company is a Stakeholder). The ontology engineer has a reference for which information the system might expect from a user and can model the domain ontology accordingly (an example is given in section 4).
In order to use the purely semantic information it has to be enriched with syntactic information. The basic concept behind it is described in [6]. The basic

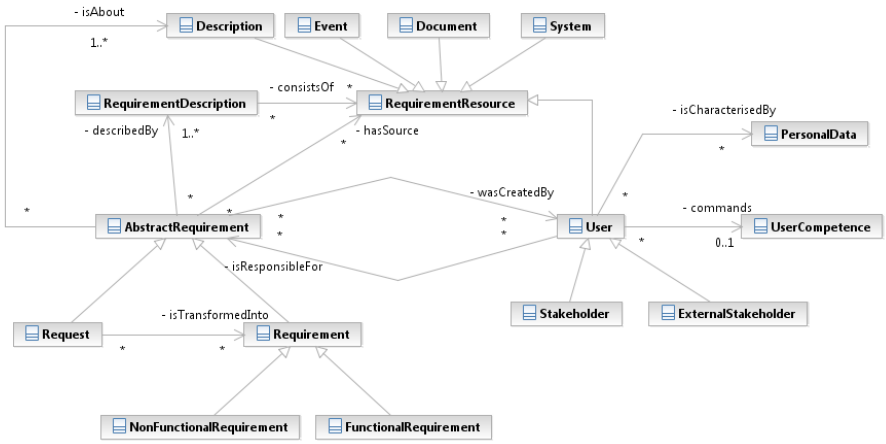

Fig. 2. Excerpt of the requirement ontology

idea is based on Construction Grammars, a paradigm which has its origins in Cognitive Linguistics. Therefore our concept provides a structure to enrich all kinds of concepts with linguistic information. A short excerpt is shown in figure 3. The main concept is the **Construction** which basically consists of a set of **Symbol**s, **Statement**s and **SymbolMapping**s. A Symbol is more or less a proxy referencing another Construction, Syntactic Element or a Semantic Element. A Statement allows the definition of further restrictions (e.g. stating that a set of syntactic symbols comes in a certain word order). A SymbolMapping allows the mapping of a syntactic to a semantic element (e.g. stating that the string 'car' represents the semantic element 'Vehicle').
Based on this structure the semantic information of the requirements ontology can be enriched with linguistic information sufficient to parse full sentences. Of course there are certain limitations and difficulties. One is the amount of linguistic information itself, therefore we integrate as much information as possible from existing linguistic data sources like the TIGER Corpus[2].

4 Example

In this section we describe an example how the concept works. The user request to be analyzed is:

> My car starts to rattle when driving at a speed of 120 km/h.

As can be seen in figure 4, we have a small excerpt of a domain ontology, holding information about a car manufacturer and the request of the customer. It describes relevant concepts of a car (e.g. that it is driven by a customer) and states

[2] http://www.ims.uni-stuttgart.de/projekte/TIGER/TIGERCorpus/annotation/

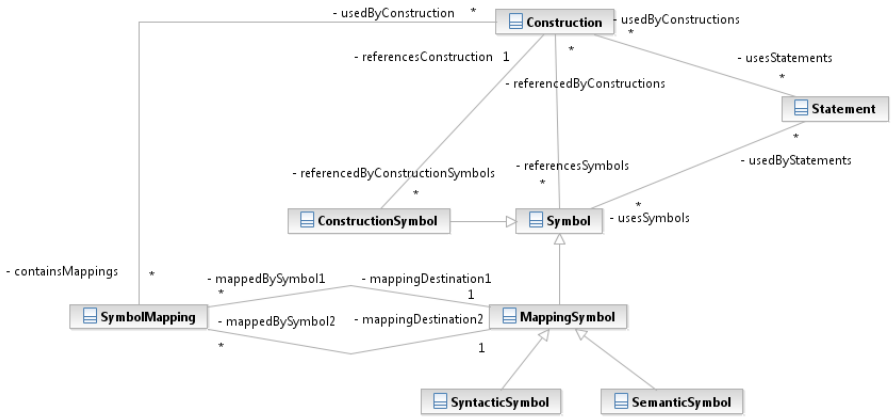

Fig. 3. Excerpt of the construction structure

a specific problem (i.e. the car rattles if it reaches a specific speed range). Further information is available in the ontology, i.e. that the cylinders are connected to the rattling. Concepts which are entangled with a blue rectangle, belong to the previously described requirements ontology.

To use these information in an NLP application the information is enriched with linguistic information as shown in figure 5. On the upper left side of the figure the semantic element 'Car' is displayed. This element is mapped on a possible string representation 'Vehicle'. In order to use this mapping it has to be referenced by a construction (in this case the **Car Construction**). To use a semantic element in a construction a symbol is introduced which is referenced by the construction. The symbol itself then references the corresponding semantic element. The cause for this structure is that the same syntactic or semantic information has to be used twice or more within a construction. A symbol can therefore be seen as a variable name and the element it references is the variable type. Mapping a semantic element to a syntactic representation is a simple task in contrast to describing more complex phrasal structures. The right side of the figure contains a construction which specifies a simple subject, predicate, object construction. It references the syntactic category noun twice ('Noun Symbol 1' and '2') as well as the verb category. To specify more concrete structures e.g. the order in which specific terms must have been written in the text can be done by using so called 'Statement's. A statement is a function which gets different symbols as its arguments. In this example the 'InOrder' statement checks if the given symbols appear in the correct order. Statements can also be used to check for certain semantic conditions (e.g. if a certain word has a specific semantic meaning which is useful in case of homonyms) or to create specific semantic constructions (which will result in the interpretation of a text): In figure 5 an additional statement in the Subj→Pred→Obj construction could define that the

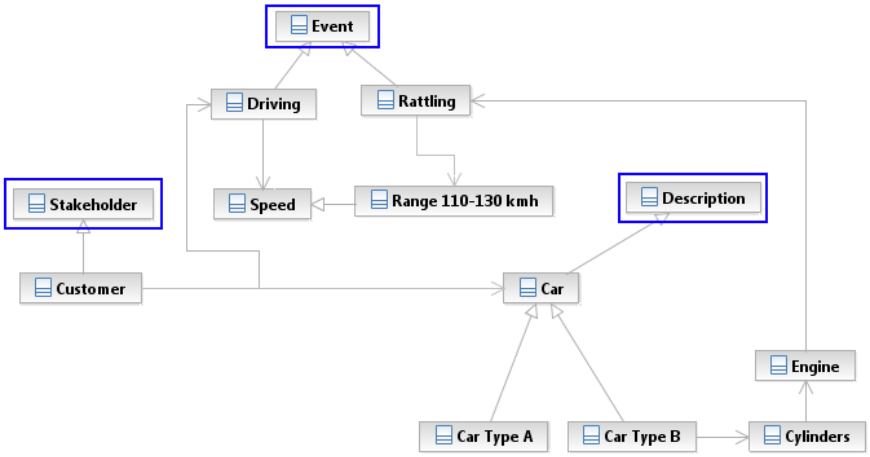

Fig. 4. Domain ontology example. Elements from the requirement ontology are encircled with a blue rectangle

meaning of the sentence's subject must be directly connected to the meaning of the object by the meaning of the predicate.

The annotation of semantic information with syntactic knowledge (as seen in figure 5) can be done automatically to some extent. However there is still a large part of information which has to entered manually. We are currently developing a concept which uses existing linguistic information from treebanks to create constructions and their corresponding statements (we use the TIGER Treebank [2]). The general mechanism can be adapted to other treebanks. This automatic transformation process helps with the creation of omplex phrasal structures and therefore facilitate the process of enriching the semantics with linguistic information. However the syntactic description of semantic elements will still have to be done manually as it is unknown in which domain the system will be used and therefore how the semantic elements will be represented in this domain.

Next, the system analyzes the sentence according to the available information at hand and tries to form a consistent semantic interpretation. Therefore it first identifies the possible concepts of the text. Next, step by step, constructions are chosen from the ontology and evaluated according to their statements and symbols. If all of the condition statements of a construction apply to a certain situation, the construction is used and its effect statements are being executed. This mechanism results in an interpretation which can be seen in figure 6. The text is analyzed according to the structure of the requirements described in section 3. The system detected that 'My car' probably refers to the customer being related to the concept of a car (more specific to 'Car Type B', as this is the only one having the rattle problem, according to the domain ontology). This is done

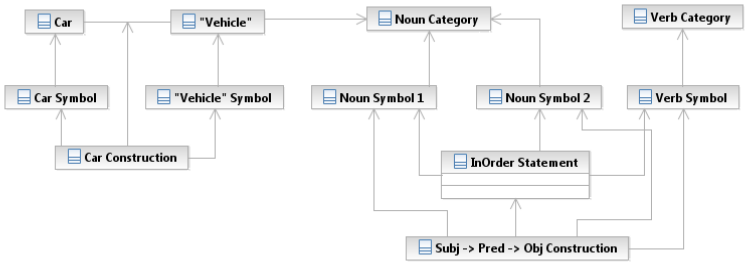

Fig. 5. Enrichment of the domain with linguistic information

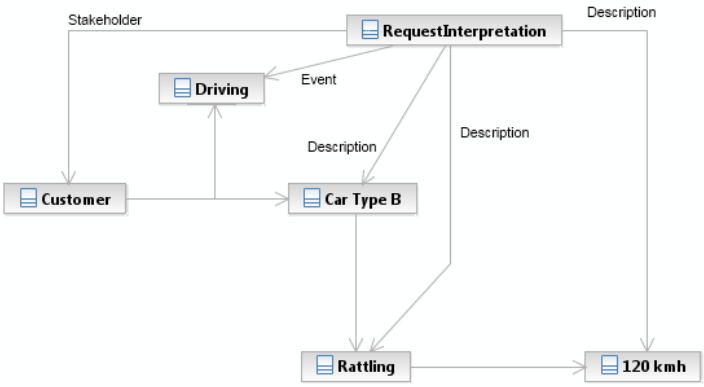

Fig. 6. Interpretation of the text resulting in a possible requirement

by a single construction which identifies 'My' as a word which is related to the semantic element 'Customer' and 'car' as a word which is (initially) mapped to the semantic element 'Car'. As this construction has all its condition statements fulfilled (e.g. the words are in the correct order), its effect statements are being executed. One of the effect statements clarifies how the semantics have to be put together. In this case the statement just says that the 'Customer' is related to the 'Car', therefore an edge between the Semantic Element 'Customer' and the Semantic Element 'Car' has been inserted. Further, as the user has stated that he is 'driving at a speed of 120 km/h' this information is also being extracted and added to the interpretation. The semantic element 'Driving' is added as the type of the edge between 'Customer' and 'Car'. The remaining information is added the same way.
After identification of the corresponding concepts their roles within the requirement (i.e. their relation to the requirement within the ontology as described in figure 2) are gathered and added to the request accordingly. Therefore, the 'Customer' seems to be the 'Stakeholder' of this requirement (as it can be seen in figure 2), the 'Driving' in this case is marked as an 'Event' and the remaining concepts are part of the 'Description'. This role-elicitation process is part of a probability method based on which concept seems to take in which role the most likely.
The so gathered information within the request can now be enriched with further information in order to make the request more suitable for engineers. Therefore the request can be transformed into a requirement, where additional information like the 'Cylinders' could be added.

5 Outlook, Possibilities and Limitations

Currently we are in the process of developing a prototype which shows the advantages of our concept. Besides the creation of the semantic interpretation there are two specific applications we will use the interpretation for:

1. Identifying previous requirements: The semantic interpretation presents a result which resolves different problems like homonyms and synonyms and therefore can provide better results than purely syntactic based search mechanisms. Therefore a semantic search helps to reduce redundancy. In case of customer requests it could help to identify prior and probably already answered requests and thus speed up the answering process within a company's support department.
2. Semantic traceability: The semantic interpretation can improve the overall results and usability of the traceability aspect. Due to the semantic representation, not explicitly stated information can be gathered through inference (e.g. by using transitive and symmetric properties as well as the taxonomy) which makes it easy to identify the origin of specific products or follow a request to the artifact it resulted in.

The possibilities that this approach can support are tremendous. It could help simplify every days requirement engineering as well as management by automat-

ically detecting overlaps and duplicates. Further erroneous as well as incomplete requirements could be identified automatically and therefore help the requirements engineer doing his / her work. However there are several limitations to this approach. As for every knowledge intensive system there must be an intensive amount of knowledge, especially linguistic information. At the moment there are many different treebanks available which represent a good foundation for this linguistic source of knowledge. We are currently developing an import mechanism for that task. It will be interesting to see how good the information from these sources can be adapted to our systen and how much time this process will consume (i.e. how many changes a human has to make to this imported information for them to be usable). Further, this knowledge needs to be updated regularly (i.e. either existing information has to be changed or new one has to be added as domains are dynamic systems). This is a tedious and time consuming task and one of the biggest problems with these system types. A way to circumvent this problem are learning components, i.e. concepts and algorithms which help the system to adapt to new and unknown situations. This will be a part of our future work especially as the core of our concept proves a very promising approach to this task (i.e. its combination of syntax and semantics). Another limitation is metaphorical reasoning, i.e. the possibility to resolve metaphors and what they mean in the corresponding context. In a linguistically limited domain this might not be a big problem (as a clear linguistic description is required and therefore metaphors should not be used) but e.g. business / early requirements might contain an imprecise linguistic description (i.e. metaphors).
Next detecting references within text is very difficult for any NLP system. Our approach also faces this problem however due to the usage of semantics from the very beginning we hope to be better suited to this problem, as we can resolve dependencies not only on a syntactic but on a fact driven level as well.
A fifth boundary (one we cannot tackle in the near future) is pragmatics. Pragmatics can be described as semantics in context. An example would be a scenario, consisting of a room with an opened window, a person A standing at the window and another person B standing at the door. B utters 'It's cold in here', which from a purely semantic point of view is the statement that the air in the room is cold. From a pragmatic point of view it also is a request to person A to close the window. To identify and manage pragmatics a system would have to 'imagine' the situation which is even more difficult than just interpreting semantics (and computer science has problems to do even this). The last sections contained many theortical aspects and examples that we are working on. Some of our ideas have not yet been implemented and therefore it will take some more time until we can deliver a prototype which is capable of delivering the results that we have described in this paper. What our prototype is currently capable of is analyzing simple sentences and creating an interpretations for it. In figure 7 a full and automatically created interpretation of the sentence 'The car is driven by the CEO' can be seen. The lower part mostly contains complex (preceded by a 'Con' prefix) or mapping constructions ('Mapping' prefix), whereas the upper part contains the different syntactic (prefix 'SynSym') as well as semantic infor-

mation (this is the data within the blue rectangle). The system can evaluate the sentence both with an active or passive verb phrase, the semantics will always remain the same. The system yet consists of the basic algorithm for eliciting constructions and evaluating them based on a given sentence. Yet the elicitation of the elements role as well as a more precise disambiguation mechanism is missing.

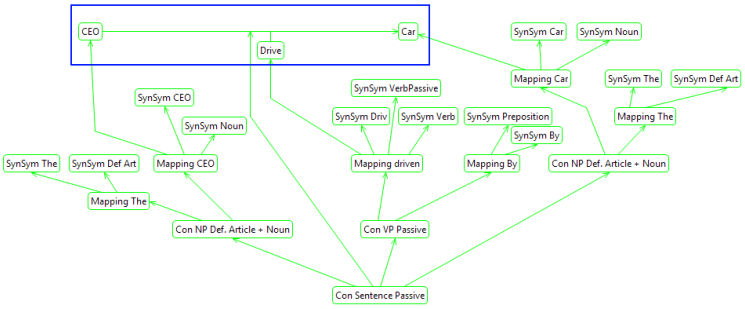

Fig. 7. Actual interpretation of a simple sentence by the prototype

6 Conclusion

In this paper we presented a novel approach for requirements engineering by directly combining semantics and natural language processing. The overall architecture has been described as well as been evaluated exemplarily. The first results are very promising and we will refine the approach and apply to several case studies in the near future.

References

1. B. Bergen and N. Chang. Embodied construction grammar in simulation-based language understanding. *Construction grammars: Cognitive grounding and theoretical extensions*, pages 147–190, 2005.
2. S. Brants, S. Dipper, S. Hansen, W. Lezius, and G. Smith. The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*, Sozopol, 2002.
3. W. Croft. *Radical construction grammar: Syntactic theory in typological perspective*. Oxford University Press, USA, 2001.
4. C. Denger. Studie zum stand der software-entwicklung in der medizintechnik management zusammenfassung. *IESE-Report*, Nr. 07/D, 2007.
5. G. Dobson, S. Hall, and G. Kotonya. A domain-independent ontology for non-functional requirements. In *IEEE International Conference on e-Business Engineering, 2007. ICEBE 2007*, pages 563–566, 2007.

6. W. Fischer and B. Bauer. Cognitive-linguistics-based request answer system. Springer, 2009.
7. L. Kof. Natural Language Processing for Requirements Engineering: Applicability to Large Requirements Documents. In *Proc. of the Workshops, 19th International Conference on Automated Software Engineering*. Citeseer, 2004.
8. R. Langacker. An introduction to cognitive grammar. *Cognitive Science: A Multidisciplinary Journal*, 10(1):1–40, 1986.
9. S. Lee and R. Gandhi. Ontology-based active requirements engineering framework. In *Proc. 12th Asia-Pacific Soft. Engg. Conf.(APSEC 05), IEEE CS Press*, pages 481–490, 2005.
10. S. Lohmann, P. Heim, S. Auer, S. Dietzold, and T. Riechert. Semantifying Requirements Engineering–The SoftWiki Approach. In *Proceedings of the 4th International Conference on Semantic Technologies (I-SEMANTICS)*, pages 182–185, 2008.
11. M. Luisa, F. Mariangela, and N. Pierluigi. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56, 2004.
12. L. Nielsen. *Requirements Engineering: Anforderungsdefinition mit Hilfe von UML bei der Entwicklung einer ERP Software*. GRIN Verlag, 2008.
13. J. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. A linguistic engineering approach to large-scale requirements management. *Managing Natural Language Requirements in Large-Scale Software Development*, 22(1):145, 2005.
14. L. Steels. Fluid Construction Grammar Tutorial. Tutorial. 2004.
15. L. Steels and J. De Beule. A (very) brief introduction to fluid construction grammar. In *Proceedings of the Third Workshop on Scalable Natural Language Understanding*, pages 73–80. Association for Computational Linguistics.
16. L. Steels and J. De Beule. Unify and merge in fluid construction grammar. *Lecture Notes in Computer Science*, 4211:197, 2006.
17. E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, page 226. Citeseer, 1997.

A UML Profile for Conceptual Modeling in GIS Domain

Jugurta Lisboa-Filho[1], Gustavo Breder Sampaio[1],
Filipe Ribeiro Nalon[1] and Karla A. de V. Borges[2]

[1]Departmento de Informática, Universidade Gederal de Viçosa
36570-000 - Viçosa, MG, Brazil
jugurta@ufv.br, gustavobreder@gmail.com, frnalon@gmail.com

[2]Prodabel – Empresa de Informática e Informação do Município de Belo Horizonte
Av. Pres. Carlos Luz, 1275 – 31230-000 – Belo Horizonte – MG - Brazil
karla@pbh.gov.br

Abstract. After many years of research in the field of conceptual modeling of geographic databases, experts have produced different alternatives of conceptual models. However, still today, there is no consensus on which is the most suitable one for modeling applications of geographic data, which brings up a number of problems for field advancement. A UML Profile allows a structured and precise UML extension, being an excellent solution to standardize domain-specific modeling, as it uses the entire UML infrastructure. This article proposes an UML profile developed specifically for conceptual modeling of geographic databases called GeoProfile. This is not a definite proposal; we view this work as the first step towards the unification of the various existing models, aiming primarily at semantic interoperability.

Keywords: UML profile, GIS, Conceptual data model, Geographic database.

1 Introduction

One of the current concerns in software development is to better understand the domain of the problem, about which it is intended to create solutions that meet satisfactorily the real needs of users. To aid in this task, one of the techniques used is the conceptual modeling, which consists in to extract from the real world only those essential elements observed, leaving out implementation aspects.

The process of conceptual modeling allows a better understanding of the system being designed and is performed with the aid of specific modeling languages, which are languages whose syntax and semantics are focused toward the conceptual representation of a system [3]. The Unified Modeling Language (UML) has been widely used and accepted by the scientific community and industry, as a tool for design and specification of systems [18].

One area that has currently received much attention includes the geographic applications domain, given its wide range of usefulness to society and the scientific community and whose systems have particular characteristics that need to be taken into account in developing such applications.

I. Reinhartz-Berger, A. Sturm, Y. Wand, J. Bettin, T. Clark, S. Cohen, J. Ralyté, and P. Plebani (Eds.):
CAiSE 2010 Workshop DE@CAiSE'10, Hammamet, Tunisia, pp. 18-31, 2010.

Parent et al [20] emphasize that the conceptual modeling has several advantages for the design of geographic applications. It allows, for instance, users to express their knowledge on the application using concepts that are closer to them, without the need to use computational expressions.

For the past 20 years, several research groups have been studying the requirements for database conceptual modeling of Geographic Information Systems (GIS) [1]. Some conceptual models specific to this area were proposed. OMT-G [4], MADS [20], GeoOOA [14], UML-GeoFrame [15] and the Perceptory's model [2] are important among these models.

Despite the maturity of this research field, to date, there is no consensus among designers and users as to which model best meets the requirements for modeling a geographic database (geoDB). The lack of a standard model brings up serious problems in the development of the field, as for instance, communication difficulties among different projects. For example, considering CASE tools that support conceptual models specific to geoDB, data conceptual schemas cannot be migrated between different tools, as it happens with conventional database designs.

These problems would not exist if there was a standard for modeling such applications that incorporated the main features of the existing models. The creation of a UML profile is one option to standardize this type of models. UML profile is a feature that allows for a structured and precise extension of the UML elements so that it can fit into a specific domain [12].

This paper aimed to initiate the specification of a UML profile for the conceptual modeling of geoDB taking into account the requirements imposed on this application domain. Some models in the literature provided the basis for this task. The remaining of the paper is structured as follows. Section 2 presents the concept of UML profile. Section 3 describes the requirements of geoDB conceptual modeling, as well as the main current models, while Section 4 details the proposal to the GeoProfile and usage examples. Section 5 presents the final considerations and future work.

2 UML Profiles

Despite being a general purpose language, which can be used in different application domains, there are situations in which the UML elements are not able to express all the peculiarities of a given domain. Therefore, to prevent the UML became too complex, it was specified as an extensible language [10].

The OMG defines two ways of extending the UML. The first is based on the modification of the UML metamodel, thereby creating a new language, in which the syntax and semantics of the new elements are adapted to the intended domain. The second way is to adapt the UML to specific domains or platforms using the mechanism of profiles. In this second alternative, the elements of language are specialized, but respecting the UML metamodel and maintaining the original semantics of the elements unchanged [12].

In this first form of UML extension, the new language is created using MOF. In the second alternative, the language elements will be specialized by using the extension mechanisms provided by UML, which are:

- **Stereotypes**. A stereotype defines how an existing metaclass may be extended and enables the use of specific terminology for a domain or different platform in place of or in addition to the terminology used for the extended metaclass. Stereotypes can also change the appearance of the elements of the extended model using graphic icons;
- **Tagged values**. They are additional meta-attributes associated with a metaclass of the metamodel extended by a profile and add information to elements of the model;
- **Constraints**. These are restrictions associated with the corresponding elements of the metamodel. They can be written using natural language or OCL, which is also standardized by the OMG.

A UML profile is a set of extension mechanisms grouped in an UML package stereotyped as <<profile>>. As mentioned earlier, these mechanisms allow the extension of the syntax and semantics of the UML elements, but without violating the original semantics of UML and, therefore, consistent with MOF.

The idea of extending the UML for specific purposes is not new. UML 1.1 could already easily assign stereotypes and tagged values to model elements. However, the notion of profile was defined to provide a more structured and precise extension [18]. UML profile is already adopted as a standard modeling in some domains, such as CORBA architecture [19]. Other profiles are in the process of being adopted by the OMG or are being created by private organizations, software companies and research centers.

OMG [18] emphasized that there is no simple answer to the question of when to create a new metamodel or when to use the mechanism of profiles. Each alternative has its advantages and disadvantages, but the use of UML profiles provides a better cost-benefit ratio, by utilizing the entire structure of the UML tools and training materials. Fuentes and Vallecillo [12] mention that the benefits of using UML profiles undoubtedly exceed their limitations.

A UML Profile allows a structured and precise extension of UML constructors to customize UML for a particular domain. A well-specified UML Profile will have direct support of CASE tools. In other words, once the Profile is defined there is no need to implement new CASE tools. *Enterprise Architect* [9] and *Rational Software Modeler* [21] are examples of CASE tools with support for UML Profiles.

Hence, the development of a UML Profile has proven an excellent method to standardize modeling of specific domains, as it uses the language's popularity and tools compatible with UML 2.0, favoring standard acceptance and reducing time for training in new languages.

3 Conceptual Modeling of Geographic Database

The term Geographic Information Systems (GIS) is applied to systems that perform a computational analysis of geographic data. The main difference between GIS and a conventional information system is the ability of GIS to store both the descriptive attributes and the geometries of different types of spatial data [24].

GIS use has grown and continues to grow rapidly throughout the world due to advances in hardware and software and the increasingly easier access to these technologies. Worboys [24] points out that among the main components of a GIS is the storage component, which is called geographic database. Its function is to

structure and store data in order to enable carrying out the analysis with spatial data. Applications developed with GIS are highly complex and a major problem in developing these applications has been designing the geoDB [16].

The classical approach to project database is to divide the process into three stages: conceptual design, logical design and physical design [8]. In conceptual design, the conceptual database is drawn up on the basis of conceptual models that provide high-level abstraction builders to describe the requirements for application data.

One of the principles of conceptual modeling is that a conceptual schema should only contain the elements of the domain, discarding implementation aspects. The process of database conceptual modeling includes a description and definition of possible contents of data, as well as structures and rules that apply to them [15]. In the case of geoDB, the specific nature of geographic information led to the development of specific solutions for modeling spatial data.

Friis-Christensen et al [11] describe a survey of requirements for modeling spatial data. These requirements are classified into five groups, as follows:

- **Spatiotemporal properties**. Include the spatial requirements (coordinates in a reference system, representation of points, lines and polygons), time (need to record the existence time and the changes undergone by an object); need for representation of object attributes, and a unique identifier; and difference between fields (the real world is perceived as a set of space-varying attributes as a continuous function) and objects (the real world consists of entities with unique identity);
- **Roles**. A same geographic object can be defined in different ways depending on the universe of discourse. That is, the role of an object is dependent on the application. It should be possible the indication of roles based on the same type of object;
- **Associations**. Include topological relationships (e.g., overlap, touch), metric (involving distance and depending on the absolute position of objects in a reference system), semantic (e.g. “all lots must have access to roads”), and relationships to indicate that an object is composed of other objects;
- **Constraints**. It should be possible to attach constraints to objects (e.g., limiting the value of an attribute to a certain range) and associations (e.g., preventing a building from being located on a lake). Constraints are related to data quality, which is negatively affected when constraints are not met.
- **Data quality**. This information is important in order to know the source credibility and data accuracy. It should be compared with the application specifications to determine whether the data is accurate enough at that time.

Another list of requirements is shown in [17]. This study mentions eight groups of requirements, five of which are equivalent to those presented by Friis-Christensen et al [11]: possibility of modeling phenomena in the field and object view, spatial aspects, spatial relationships, temporal aspects, and quality aspects. The other requirements, not explicitly mentioned in the previous work, are: possibility of differentiating between geographical phenomena and objects without spatial reference; the need to organize the phenomena by theme; and the possibility of modeling phenomena with more than one spatial representation (multiple representations).

Friis-Christensen et al [11], compare some models with these requirements to show advantages and disadvantages of each model. One of the conclusions of this

study shows the importance of balancing ease-of-use of the model notation with its comprehensiveness. The posed challenge is to balance these two characteristics or improve them, and the development of a standard model provides the basis for data exchange.

The profile proposed in this paper is based on contributions from a number of models existing in the literature, as well as the concepts defined in Goodchild [13]. The models that have contributed most significantly to the GeoProfile development are cited below, but certainly other predecessor models also had their contribution.

The OMT-G (Object Modeling Technique for Geographic Applications) model [4] has a rich collection of conceptual constructors, the strong point of which is modeling spatial relationships, including spatial aggregation. The GeoOOA model [14] supports the abstraction of spatial classes, whole-part topological structures, network structures and temporal classes. MADS (Modeling of Application Data with Spatio-temporal Features) [20] approaches objects and relationships in its diagram, with structures very similar to the Entity-Relationship model. Its main feature is the orthogonality, in which spatial and temporal characteristics can be added either to objects or attributes or relationships. The Perceptory's model was the pioneer in the use of pictograms. These pictograms are grouped into the languages Spatial PVL and Temporal PVL (Plug-in for Visual Languages), which allow the addition of spatial-temporal characteristics not only to UML, but also to other visual modeling languages. The UML -GeoFrame model is based on a structured hierarchy of classes that make up the GeoFrame, providing the basic elements present in any geographic database [15]. The proposal of ISO-191xx Standard [6] differs from the models above mentioned for addressing more the logical level (records) than the conceptual level (abstractions).

Finally, Clementini et al [7] formally describe a small set of relationships capable of reproducing all the possible topological relationships that can occur between spatial elements with the representation of point, line or area. Although not proposing a model, this work has considerable importance in the scope of the GeoProfile design. Defining a minimum set of relationships, one eliminates the possible use of two relationships with different names, but having the same meaning. This set includes the following relationships: *touch, in, cross, overlap* and *disjoint.*

4 GeoProfile

GeoProfile is a UML profile built for the conceptual modeling of geographic databases. According to the proposed methods to guide the construction of a UML Profile (Section 2), two artifacts are generated during profile development: the domain metamodel and the profile itself. While the first is useful to understand the addressed problem, the second presents the extensions received by the UML metaclasses.

In order to check the validity of the GeoProfile specification, this profile has been implemented in RSM [21]. Mechanisms for creation of stereotypes were successfully tested, as well as automatic validation of schemas by checking OCL constraints.

Section 4.1 defines a metamodel for the geographical domain. Section 4.2 proposes a set of stereotypes for the proposed profile. Section 4.3 shows a way to specify additional integrity constraints. Section 4.4 shows the implementation of the GeoProfile in a CASE tool, and Section 4.5 presents examples of GeoProfile use.

4.1 Defining a metamodel for geographical domain

At the beginning of the metamodel specification, elements are identified in a conceptual schema, observing the requirements of this type of conceptual modeling.

The way each considered conceptual model in this proposal (GeoOOA, MADS, UML-GeoFrame, OMT-G and Perceptory's model) meets the found requirements was examined. The inclusion of the main mechanisms present in each of these models into the GeoProfile allows it to meet most requirements of a geoDB. Table 1 summarizes the results obtained in the comparative analysis between requirements and conceptual models, but also displays in its last column the models that most influenced GeoProfile construction in each requirement.

Among the discussed conceptual models, the UML-GeoFrame shows the closest organization to a metamodel. GeoFrame is defined in a class hierarchy representing the elements present in a geoDB. Thus, the metamodel development started from a GeoFrame adaptation (Figure 1).

A geoDB comprises a number of themes, which is characterized by the metaclass *Theme*. A theme can be formed by the aggregation of other themes or objects with or without spatial representation, characterized by the classes *GeoPhenomenon* and *ConventionalObj* respectively.

When one chooses to associate a spatial representation with objects of a class, it is possible that the phenomenon is perceived in the geographic field view (*GeoField*) or object view (*GeoObject*). Depending on the technique used in geographic information acquisition in the field, its representation be selected from six options as described in [13]: *AdjPolygons*, *Isolines*, *TIN*, *GridOfPoints*, *GridOfCells* or *IrregularPoints*. Representation of geographic objects can be of the types point, line, polygon or complex (the object geometry consists of other geometries).

To specify multiple representations, it is possible to use more than one stereotype in the same class of the conceptual schema, as in the Perceptory`s model.

Table 1. Comparison between requirements and models presented, and major contributions to the GeoProfile.

Models X Requirements	GeoOOA	MADS	OMT-G	Perceptory	UML-GeoFrame	Contribuition for GeoProfile
Geographical phenomena and conventional objects	Yes	Yes	Yes	Yes	Yes	Perceptory
Field visions and objects	Partial	Partial	Yes	No	Yes	OMT-G
Spatial aspects	Partial	Yes	Yes	Yes	Yes	OMT-G, UML-GeoFrame
Thematic aspects	No	No	Yes	Yes	Yes	UML-GeoFrame
Multiple representations	Partial	Yes	Yes	Yes	Yes	UML-GeoFrame
Spatial relationships	Partial	Yes	Yes	Partial	Partial	MADS, OMT-G
Temporal aspects	Partial	Yes	No	Yes	Partial	MADS, Perceptory

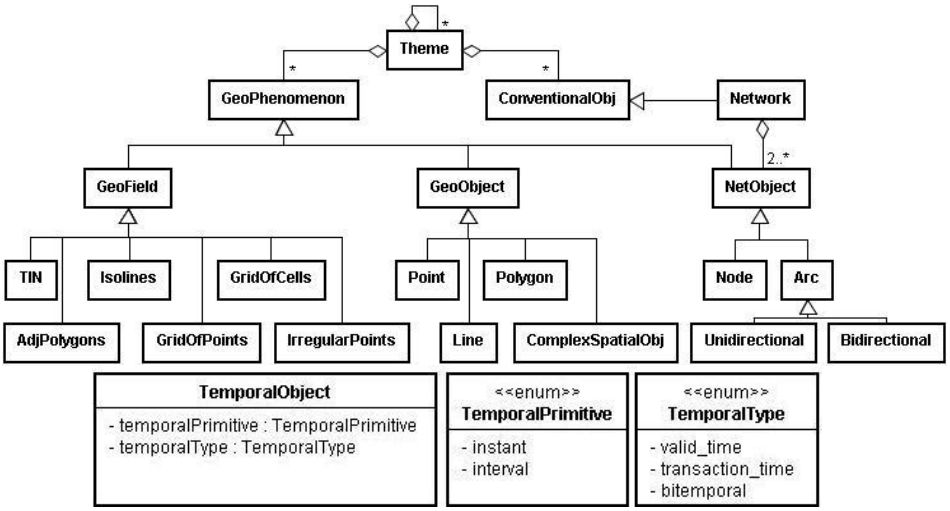

Fig. 1. Metamodel for the geographical domain

The requirements related to the roles and metadata are not considered in the GeoProfile proposal. Despite representing important information relating to spatial data, it is believed that they need not necessarily be demonstrated during the conceptual modeling of a geoDB.

Topological and composition are the main types of spatial relationships to be represented in a conceptual schema. There was no need to add new constructors to the GeoProfile to characterize composition, as the UML can indicate whether an association is a composition or aggregation. However, it was necessary to add new constructors to model topological relationships, including the capacity to represent networks.

With basis on GeoOOA and OMT-G models, which provide more detailed solutions for network representation, [23] proposed an extension of GeoFrame to address the requirement. This extension was incorporated into the metamodel.

The classes in charge of storing alphanumeric data and information on which elements participate in the network are represented by the metaclass *Network*. Since this metaclass does not have spatial information, it was defined as a *ConventionalObj* specialization. The networks are formed by network objects (*NetObject*), which can be nodes (*Node*), unidirectional arcs (*Unidirectional*) or bidirectional arcs (*Bidirectional*).

The other types of topological relationships are directly defined in the creation of stereotypes and OCL constraints. This is because a large number of possible relationships between spatial objects of the type point, line and polygon would overburden the metamodel.

The MADS and Perceptory approaches stand out among temporal aspects. Although they do not consider transaction time, icons added at different positions of the class diagram can indicate that the object's existence time, its spatial evolution or the evolution of values of certain attributes in that class should be kept in the database. Despite being an interesting solution, it can visually overload the schema.
Another solution adopted by GeoProfile is indicated only whether a class is considered temporary or not, as in the GeoOOA model. In this case, it is implied that

both the attributes and spatial data of an object can vary, and these changes must be maintained in the database.

In this way, the metaclass *TemporalObject* was added to the metamodel. This metaclass has two attributes that characterize temporal information. One of these attributes indicates the temporal type (validity time, transaction time or bitemporal time), whereas the other defines the used temporal primitive type (instant or interval). There are two enumerations (*TemporalType* and *TemporalPrimitive*) for the possible values these attributes can assume.

4.2 GeoProfile stereotypes

After creating the domain metamodel, the next step is to extend the UML metaclasses to create the profile itself. Figure 2 illustrates the stereotypes of GeoProfile, generated from the metamodel shown in Figure 1.

The UML allows the definition of graphic and textual («...») stereotypes. The authors believe that the choice of graphic stereotypes is a matter of personal taste (or customary within an organization) and there is no need of standardization. For example, two designers, one familiar with the MADS model and the other with the notation used in the Perceptory tool, might start to use GeoProfile, but keep the original graphical representation of the stereotypes of their preferred model. CASE tools that support profile may allow different graphical views of the same data schema, enhancing conceptual interoperability. Thus, initially, it was decided not to propose graphic stereotypes for GeoProfile, leaving the standardization to future decision.

It is worth noting that not all metaclasses of the domain metamodel have a corresponding stereotype, as it happens with *Theme* and *ConventionalObj*. Themes can be represented by packages. Classes of conventional objects are, however, modeled by UML classes without addition of stereotypes. Therefore, the UML constructors themselves can reproduce these two concepts.

Another important observation is that some stereotypes are abstract (*GeoObject, GeoField, NetObject* and *Arc*). During GeoProfile use, these stereotypes are not available to be used. They are, nevertheless, useful for organizing profile elements, allowing addition of constraints common to all the other stereotypes created as their specialization. For example, a constraint that is common to the stereotypes *UnidirectionalArc* and *BidirectionalArc* can be added to *Arc*.

Geographic phenomena, extending the metaclass *Class*, are defined in a similar hierarchy to that found in the domain metamodel. The stereotype *Network* directly extends the metaclasse *Class*, since there is no stereotype defined for representation of conventional objects.

To deal with temporal aspects, the stereotype *TemporalObject* was added to GeoProfile, as well as two enumerations (*TemporalPrimitive* and *TemporalType*). In addition, designers are allowed to indicate that an association between two objects is only valid for one period and this history should be kept in the database. This is done by simply assigning the stereotype *Temporal*, which extends the metaclass *Association* to an association of the schema.

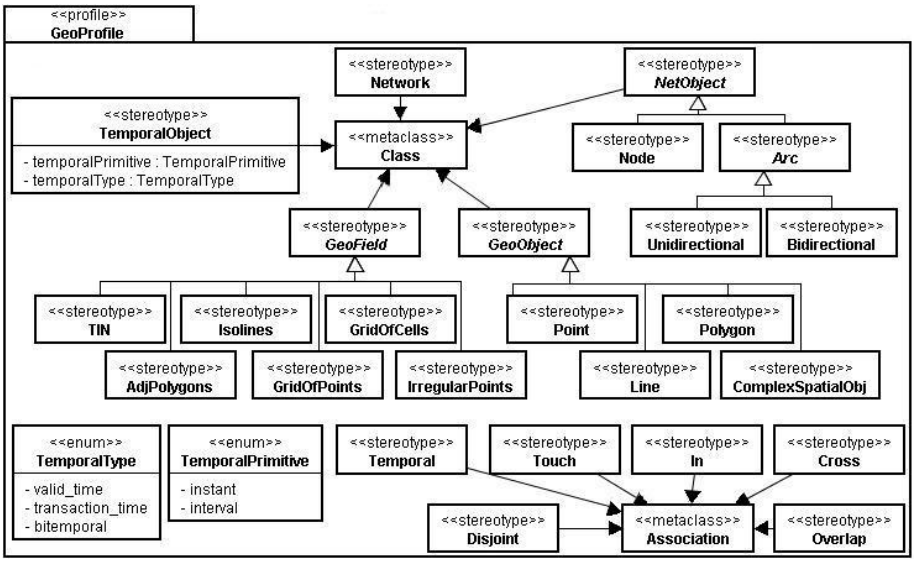

Fig. 2. GeoProfile Stereotypes

Finally, stereotypes were created to represent the topological relationships that were not considered during drawing up of metamodel. We chose to use the set of five relationships proposed by [5], as they are capable of representing any topological relationship between objects of type point, line or polygon. Thus, the stereotypes *Touch*, *In*, *Cross*, *Overlap* and *Disjoint*, all extending the metaclass *Association*, were added.

4.3 OCL constraints

The constraints included in the GeoProfile focuse on the validation of the designer's conceptual schema. Consequently, they always have a stereotype of the GeoProfile as context, as well as being invariants.

Those constraints basically prevent the occurrence of three error types: addition of incompatible stereotypes with a same element, poor network construction and addition of impossible topological relationships between two elements (e.g. *Cross* relationship between two geographic objects with point representation). These three constraints groups were analyzed and a set of OCL expressions was specified. There is no limitation to the inclusion of the stereotype «TemporalObject» in classes of the schema or «Temporal» in their associations. Because of space limitation, this article describes only one of the OCL constraints as example.

The constraint (a) evaluates the use of incompatible stereotypes. Each class that receives a stereotype of geographic field (*context GeographicField*) must have all its applied stereotypes captured (*getAppliedStereotypes*). Stereotypes of the geographic object type are selected from the result using the *select* method, and the returned set must be empty (*isEmpty*), since a class cannot have object and field representation at the same time.

```
context GeographicField
inv: self.getAppliedStereotypes() -> select(s |
s.name = 'Point' or s.name = 'Line' or s.name =
'Polygon' or s.name = 'ComplexSpatialObj') ->
isEmpty()
```

4.4 Implementation of GeoProfile in a CASE tool

One of the greatest advantages in using a UML profile as a basis for modeling of a specific field is to use the entire UML infrastructure. Therefore, an implementation of this profile in the RSM [21] was carried out to verify the validity of the GeoProfile specification. Mechanisms for stereotype creation were successfully tested, as well as automatic validation schemas from the verification of OCL constraints. OCL constraints assist the designer in identifying basic errors.

RSM is produced by IBM® and supports UML 2.1. This work used the version 7.0.5. The tool interface can be changed according to user's preferences. Figure 3 illustrates the RSM interface with support for GeoProfile.

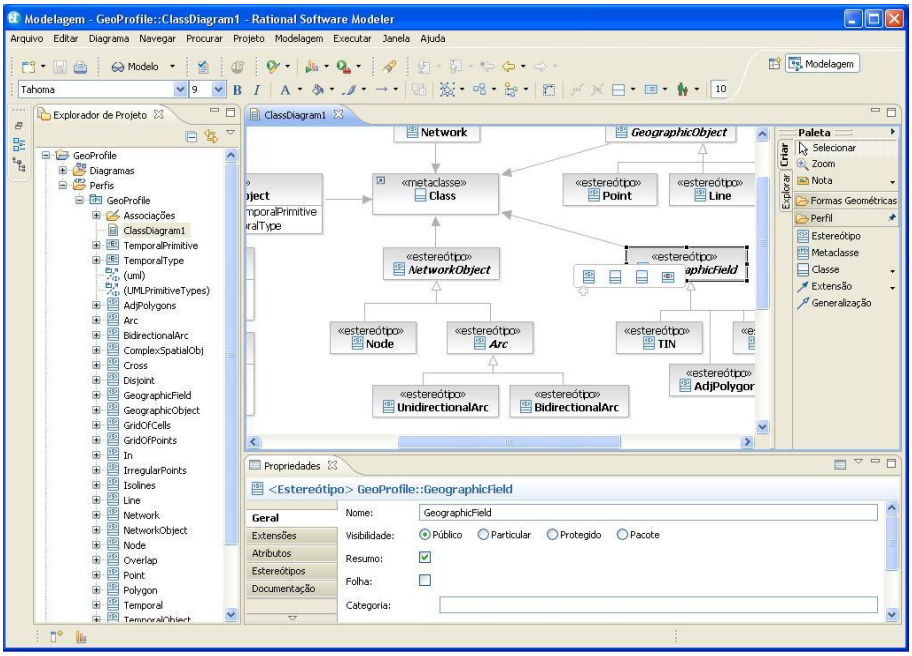

Fig. 3. RSM interface with GeoProfile

4.5 Usage examples of GeoProfile

Considering the results obtained after the establishment of the GeoProfile and its implementation in the CASE tool RSM, this section presents examples of conceptual modeling using GeoProfile. To allow a comparison between the GeoProfile and conceptual models that were the basis for its definition, each example also shows the corresponding conceptual schema in the other model.

Figure 4 illustrates part of the conceptual modeling of a system for pollution control in parcel (or plot) of land. The diagrams (a) and (b) display the model developed using the model GeoOOA and GeoProfile, respectively. The parcels have a polygonal representation. A non-geographic object providing information on the owners of each parcel must be stored. In addition, each parcel may contain several pollution controls, which are geographically represented by points. In the association between parcels and points of pollution control, the restriction that each control point must be contained in the area of the parcel with which it is associated is represented in the conceptual schema.

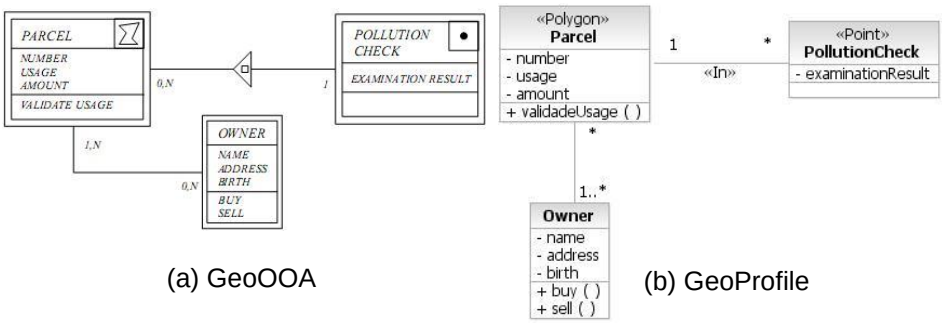

Fig. 4. Comparison between GeoOOA and GeoProfile (Source: (4-a) [14])

Another example of topological relationship involving parcels (or plots) of land is showed in Figure 5, which compares (a) the MADS model with (b) GeoProfile. In this schema each plot may contain several buildings, and both classes have polygonal representation. Furthermore, a restriction is imposed that the buildings belonging to a particular plot must have their geographical area within the area of the plot. In the case of the topological relationship «In», it may be important for a correct interpretation of the schema, to state which of the objects involved in a particular association must have its geometry contained in the geometry of the other object that participates in the association. In Figure 5-b, roles of the association were used for this purpose. Another option is to indicate the navigability of the association. Both solutions use resources of the UML specification.

Finally, the last example explores temporal aspects. In Figure 6-a, a class House is modeled using the Perceptory CASE tool [2]. The temporal pictogram located on the top right of the diagram of this class shows that the period in which the house exists in modeled reality (e.g., date of construction until the date of demolition) should be stored in the database. However, when the same pictogram is added to the side of the spatial representation pictogram or next to an attribute, it indicates that the historical of the object spatial evolution or the evolution of the values of an attribute, respectively, must be kept into the database. As discussed above, in GeoProfile, these

concepts are grouped into just one stereotype called «TemporalObject». In this case, it is implied that both the period of existence and the historical evolution of the attributes or geometry of an object must be kept in the database.

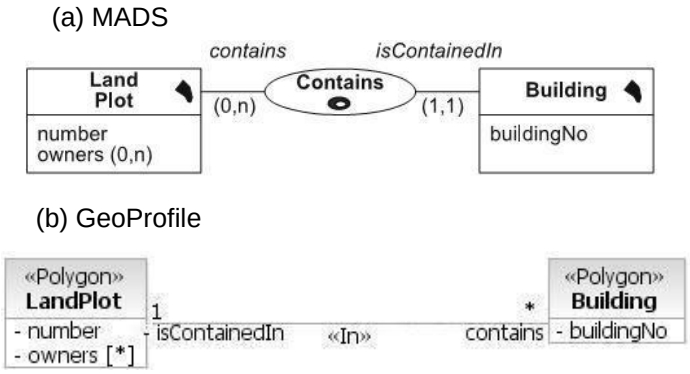

Fig. 5. Comparison between MADS and GeoProfile (Source: (5-a) adapted from [20])

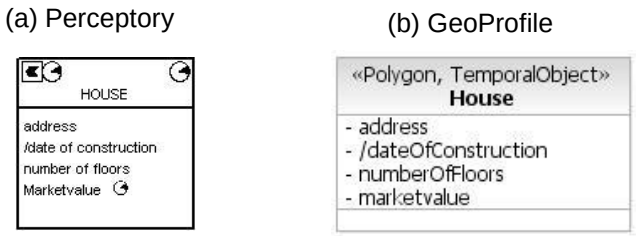

Fig. 6. Comparison between Perceptory and GeoProfile (Source: (6-a) [2])

5. Final Considerations

The idea of this paper is not to propose one more new conceptual model for GIS, but rather to propose a set of constructors, extracted from existing models toward a standard geographic profile for database modeling in GIS domain.

The existence of several alternative conceptual models of geographical databases prevents users and designers to migrate their projects from a CASE tool to another. Another major problem brought up by the lack of standardization is the difficulty in training designers, since although the models have been produced for the same purpose; each one has its differences and particularities. Users who are familiar with a model and its respective CASE tool (e.g. Perceptory [2] and ArgoCASEGEO) show strong resistance to accept a new one.

The use of a UML profile will solve these problems. Besides the wide UML acceptance by software developers, the availability of CASE tools with support for profiles rule out the need for implementing specific tools for a particular model.

A subject for future work is the logical-conceptual transformation of schemas produced with GeoProfile. The existence of logical standards, as defined by OGC and

the series ISO 191xx [6], will have a strong link with the level of conceptual modeling. Finally, the great challenge is to make authors of the existing conceptual models contribute to improve the GeoProfile. Moreover, to know the opinion of the users is important, because in many cases the database of a GIS application is designed by then. Thus, it is also important to measure the GeoProfile use's facility and its learning curve.

Acknowledgments. This project was partially funded by CAPES, FAPEMIG and CNPq / MCT / CT-Info.

References

1.Bédard, Y., Larrivée, S., Proulx, M., Nadeau, M.: Modeling geospatial databases with plug-ins for visual languages: a pragmatic approach and the impacts of 16 years of research and experimentations on Perceptory. In:CoMoGIS 2004. LNCS. vol. 3289, pp. 1148-1158. Springer (2004)
2.Bédard, Y.: Visual modeling of spatial databases: towards spatial PVL and UML. Geomatica, 53(2), 169--186 (1999)
3.Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language user guide. 2. ed. Addison-Wesley, Boston (2005)
4.Borges, K.A.V., Davis Jr., C.A., Laender, A.H.F.: OMT-G: an object-oriented data model for geographic applications. GeoInformatica, 5(3), 221--260 (2001)
5.Brodeur, J., Bédard, Y., Proulx, M.-J.: Modeling geospatial application database using UML-based repositories aligned with International Standards in geomatics. In: ACMGIS, Washington DC (2000)
6.Brodeur, J., Badard, B.: Modeling with ISO 191xx standard. In: Shekhar, S.; Xiong, H. (Eds.). Encyclopedia of GIS. Springer-Verlag, pp. 691--700 (2008)
7.Clementini, E., Di Felice, P., Oosterom, P.: A small set of formal topological relationships suitable for end-user interaction. In: Int. Symp. on Advances in Spatial Databases. Springer-Verlag, London. pp. 277-295 (1993)
8.Elmasri, R., Navathe, S.B.: Fundamentals of database systems. 4 ed. Addison-Wesley, Boston (2003)
9.Enterprise Architect. http://www.sparxsystems.com/products/ea/
10.Erikson, H., Penker, M., Lyons, B., Fado, D.: UML 2 Toolkit. OMG Press, Indianapolis (2004)
11.Friis-Christensen, A., Tryfona, N., Jensen, C.S.: Requirements and research issues in geographic data modeling. In: ACM Int. Symp. on Advances in GIS, Atlanta, pp. 2--8 (1993)
12.Fuentes, L., Vallecillo, A.: An introduction to UML profiles. UPGRADE, The European Journal for the Informatics Professional, 5(2), 6--13 (2004)
13.Goodchild, M. F., Yuan, M., Cova, T. J.: Towards a general theory of geographic representation in GIS. Int. Journal of Geographic Information Science, 21(3), 239--260 (2007)
14.Kösters, G., Pagel, B., Six, H.: GIS-Application development with GeoOOA. Int. Journal of Geographical Information Science, 11(4), 307--335 (1997)
15.Lisboa Filho, J., Iochpe, C.: Modeling with a UML Profile. In: Shekhar, S.; Xiong, H. (Eds.). Encyclopedia of GIS. Springer-Verlag, pp. 691--700 (2008)
16.Lisboa Filho, J. et. al.: A CASE tool for geographic database design supporting analysis patterns. In: CoMoGIS/ER`2004, LNCS vol. 3289, Springer (2004).
17.Lisboa Filho, J.: Iochpe, C.: A study about data conceptual models for geographic database design. Informática Pública. 1(2), 67-90 (1999) (In Portuguese)

18. Object Management Group. Unified Modeling Language: Infrastructure. V. 2.1.2 (2007)
19. Object Management Group. Profile Catalog. http://www.omg.org/technology/documents/profile_catalog.htm
20. Parent, C., Spaccapietra, S., Zimányi, E.: Modeling and multiple perceptions. In: Shekhar, S.; Xiong, H. (Eds.). Encyclopedia of GIS. Springer-Verlag, pp.682--690 (2008)
21. Rational Software Modeler. http://www-01.ibm.com/software/awdtools/modeler/
22. Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE Int. Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07), pp. 2--9 (2007)
23. Stempliuc, S. M., Lisboa F., J., Andrade, M. V. A., Borges, K. V. A.: Extending the UML-GeoFrame data model for conceptual modeling of network applications. In: Int. Conf. on Enterprise Information Systems (ICEIS), Milão pp. 164--170 (2009)
24. Worboys, M,, Duckhan, M.: GIS: A computing perspective. 2 ed. CRC Press, Boca Raton (2004)

A UML-Profile for domain specific patterns: Application to real-time

Saoussen Rekhis[1], Nadia Bouassida[2], Rafik Bouaziz[1], Bruno Sadeg[3]

[1,2] MIRACL-ISIMS, Sfax University, BP 1088, 3018, Sfax, Tunisia.
[3]LITIS, UFR des Sciences et Techniques, BP 540, 76 058, Le Havre Cedex, France.

[1]{saoussen.rekhis, raf.bouaziz}@fsegs.rnu.tn
[2]nadia.bouassida@isimsf.rnu.tn
[3]bruno.sadeg@univ-lehavre.fr

Abstract. The design of Real-Time (RT) applications is a difficult task since it must take into account the specification of time-constrained data and time-constrained transactions. The design of these applications can be facilitated through the reuse of RT design patterns that improve software quality and capture RT domain knowledge and design expertise. However, the difficulty of RT design patterns comprehension reinforces the need for a suitable design language. This language has to express concepts modeling RT features and distinguishing the commonalities and differences between RT applications.

This paper presents new UML notations that take into account the design of both RT specific concepts and the variability of domain specific patterns. The UML extensions are, then, illustrated in the RT context using an example of a controller pattern.

Keywords: UML notation, domain specific patterns, instantiation, real-time applications.

1 Introduction

A design pattern [1] is a description of a solution to a common problem in software design. It captures the design expertise necessary for developing applications and allows the reuse at both the design and code levels. Design patterns can be general and cover different domains of application (e.g. patterns of GoF [1]) and they can, also, be intended for a particular domain, in this case they are called domain-specific patterns [24].

Despite their advantages, to benefit from design patterns, a designer must spend a lot of time in understanding and then reusing the design pattern in a certain application. To facilitate the reuse and instantiation phase, many design pattern notations have been proposed ([8], [4], [3]). The proposed approaches offer essentially UML extension mechanisms such as stereotypes, tags and constraints to cope with the pattern variability and to show the pattern specificities.

I. Reinhartz-Berger, A. Sturm, Y. Wand, J. Bettin, T. Clark, S. Cohen, J. Ralyté, and P. Plebani (Eds.):
CAiSE 2010 Workshop DE@CAiSE'10, Hammamet, Tunisia, pp. 32-46, 2010.

These design languages with their UML extensions remain insufficient when they deal with a specific domain. In fact, in the design of a specific domain, the design language has to take into account not only the variability and the aspects relative to the pattern, but also the extensions and specificities of the domain itself. For example, when considering the Real Time (RT) domain, we found that this domain has many details that must be taken into account by the design pattern notation.

In fact, RT applications, which manipulate voluminous quantities of data, have two main features: i) they manipulate RT data that must closely reflect the current state of the controlled environment, and ii) they must be able to meet RT constraints of transactions. These two features must be considered by RT design patterns.

This paper proposes a new UML-profile that extends UML with concepts related to RT design patterns. The motivations behind these extensions are three-folds. The first motivation is to have flexible patterns that distinguish the fixed parts from the optional and variable elements in the pattern. The second motivation is to facilitate the comprehension of design patterns instantiation and to guide a designer to derive a specific application. The third motivation is to present design patterns for the RT domain using the proposed profile which is extended with RT specific concepts.

The remainder of this paper is organized as follows. Section 2 overviews and evaluates currently proposed design languages and their extensions. Section 3 presents our proposition to represent an UML profile for RT design patterns. Section 4 illustrates the design language with a RT controller pattern and presents an example of a freeway traffic management system reusing it. Section 5 concludes the paper and outlines future work.

2 Overview of current works

In order, to propose a RT pattern profile, we have been inspired in our work from RT profiles and existing pattern notations. Thus, in this section we, first, overview current design languages for pattern's representation. For this reason, we define a set of criteria necessary for pattern notations and then we present their advantages and limits. Second, we briefly present in Subsection 2.2 the RT profiles and the UML extensions taking into account the real-time system requirements.

2.1 Overview of UML extensions for design patterns representation

Several criteria have to be taken into account to evaluate the currently proposed languages for pattern representations. These criteria are used to compare current UML-based pattern notations, for the specification of general and domain-specific design patterns and for their instantiation.

- Criteria for design pattern representation at the specification level

C1. Expressivity: Design patterns have mostly been described using natural language, complex mathematical or logic based formalisms [5] [6] which are not easily

understood by an inexperienced designer. This leads to complications in incorporating design patterns effectively into the modelling of a new system. To remediate to this difficulty, the solution is using an expressive visual notation based on UML to specify patterns. This improves the pattern specification quality because UML allows to easily visualise, define and document the artefacts of the system under development.

C2. Variability: The design patterns have to incorporate flexibility and variability in order to guide the designer in determining the variable elements that may differ from one application to another. In fact, variability is classified into optional and alternative characteristics. So, it is important to show the optional elements which can be omitted in a pattern instance. It is also necessary to clarify the variability points (called *hot-spots*) which describe the elements that can vary according to a specific context.

C3. Constraints definition: The correct instantiation of patterns is a major problem when we want to design a new system by composing design patterns. The validity of an instantiation depends on respecting the properties inherent to the solution. These properties are specified by constraints that are generally expressed in OCL (Object Constraint Language) [7]. They are presented on the class diagram using *notes*.

- Criteria for design pattern representation at the instantiation level

C1. Traceability: The traceability consists of easily identifying design patterns when they are applied and composed with other patterns. In fact, we not only need to identify each pattern in a design, but also we want to show the methods and attributes that play important roles in the pattern. Explicit representation of the key methods and attributes can assist on the traceability of a pattern since it allows us to trace back to the design pattern from a complex design diagram [8].

C2. Composition: The development of applications using design patterns as design components requires a careful look at composition techniques, which are categorized as: behavioural composition techniques and structural composition techniques. Indeed, the behavioural techniques show how dynamic specifications of patterns can be composed using sequence diagram, whereas structural techniques show how the static architectural specifications of instantiated patterns can be composed using class diagram [16].

- UML notations for design patterns

There are several UML notations which proposed extensions to present general design patterns and domain models. Many of them can be used to express concepts relative to domain-specific design patterns such as their flexibility. A comparison of the most recent notations, using the specification and instantiation criteria is proposed in Tables 1 and 2.

Table 1. Comparison of current notations using the specification criteria.

	Design pattern specification criteria		
	Expressivity	**Variability**	**Definition of constraint**
Dong & Yang UML profile [3]	This profile proposes notations that focus more on the pattern applicability context than on the pattern specification.	Unlike several others notations [8] [4], the proposed profile does not focus on specifying the variability of a pattern solution.	These notations don't specify constraints which delimit the pattern applicability.
P_UML profile [8]	P_UML proposes extensions showing the pattern hot-spots in a class diagram and guiding the designer in instantiating a pattern. However, it does not distinguish between the extensions used in pattern instantiation from those used in pattern specification, which reduces the expressivity of notations.	This profile is characterized by: -The definition of tagged values to extend the static view: {*variable*} indicates that the method implementation varies according to the pattern instantiation; {*extensible*} indicates that the class interface may be extended by adding new attributes and/or methods; -The applicability of the {*incomplete*} constraint on generalization relation to indicate that new classes may be added during the pattern instantiation	These notations propose to define the pattern constraints through notes containing OCL constraints.
Arnaud profile [4]	This profile is not very expressive since the static view of a pattern is presented by very elementary separated packages which contain one or two classes. This reduces the understanding and makes the composition more difficult.	Unlike all previous notations, this profile focuses on the variability in the functional, dynamic and static views. The use case diagram is the entrance point for the instantiation process, where the application designer selects a functionality variant. However, the use case diagram is too abstract and can not be used as an input model for the patterns instantiation. In fact, the use case diagram is at a high level of abstraction and thus the designer cannot identify, for example, the optional attributes or methods according to its needs.	Similar to P_UML, this profile uses notes that contain OCL constraints. These latter must be fulfilled by a pattern to be applied correctly.
ADOM-UML [18]	ADOM-UML is an Application based DOmain Modeling approach, in which UML 2.0 is used as the modeling language of both: the domain and	ADOM-UML defines new stereotypes in order to denote the multiplicity variability of the different domain model elements. The multiplicity stereotypes aim to represent how many times a model element can appear in a	The constraints are well defined in ADOM-UML among the different layers: the domain layer enforces

	application models. Unlike the previous works [3] [4] [8], the ADOM-UML enhances the expressivity of the proposed notations since it well differentiates between the extensions used in the language, domain and application layers. This means that each layer includes modeling constructs that will be used in the more specific layer.	specific context. Particularly, the authors define four stereotypes: <<optional single>>, <<optional many>>, <<mandatory single>> and <<mandatory many>>. Each stereotype has two associated tagged values, min and max, which define the lowest and the upper most multiplicity boundaries. However, the word 'many' used in these stereotypes doesn't enhance the semantic of UML model since each element in a model can be instantiated implicitly many times.	constraints on the application layer, while the language layer enforces constraints on both domain and application layers. Besides, ADOM-UML specifies additional constraints and dependencies in the domain layer expressed in OCL.

Table 2. Comparison of current notations using the instantiation criteria.

	Design pattern instantiation criteria	
	Traceability	**Composition**
Dong & Yang UML profile [3]	This profile proposes new stereotypes and tagged values for the explicit representation of design patterns in software designs. These extensions show the pattern name, the role names of the classes, the attributes and the operations in the pattern and how many instances of a design pattern are applied.	This profile deals with the composition of patterns statically. That is, when two or more classes represent the overlapping part of the composition, the proposed notation shows the roles that these classes play in each pattern.
P_UML profile [8]	This notation proposes to show the pattern participant roles by using an ellipse in the bottom of a class that indicates the pattern name and the role through which this class participates in the pattern. Thereby, it provides support for traceability of pattern instantiation. However, the class diagram may seem to be overloaded since the notation presents an association between ellipses to join the elements of the same pattern.	Like the previous work [3], this profile proposes extensions showing the composition of patterns presented by class diagrams. It does not present notations to deal with the composition of patterns dynamic specifications.
Arnaud & al. UML profile [4]	This profile defines a process to show the steps of patterns instantiation. However, it does not permit the visualization and preservation of pattern-related information in patterns instances in a design model. Consequently, it does not deal with the traceability criterion.	This profile does not present mechanisms to compose neither static, nor dynamic specifications of patterns.
ADOM-UML [18]	The connection between the domain and application layers is done through the stereotypes extension mechanism. This means that a domain element can serve as a stereotype of an application	The composition criterion is not taken into account in domain models. In fact, an application model is created

	element if their meta-classes in the language layer are the same (e.g., a class that appears in a domain model may serve as a classifier of classes in an application model). Thereby, ADOM-UML provides support for traceability criterion and enhances the readability of an application model.	according to the adaptation of one domain model and doesn't deal with the composition of many reusable domain artefacts, such as patterns.

In summary, none of the proposed notations satisfies all the different specification and instantiation criteria, when representing patterns. Moreover, none of them proposes extensions showing the behavioral composition.

2.2 Overview of UML extensions for RT applications

Several works have proposed UML extensions to take into account the real-time system requirements such as, *RT-UML* [20] and *ACCORD/UML* [21]. The basic concepts of *RT-UML* were integrated in the UML standard through the UML profile for **S**chedulability, **P**erformance, and **T**ime (denoted SPT profile) [22]. Recently, *MARTE* profile [10] for **M**odeling and **A**nalysis of **R**eal-**T**ime **E**mbedded systems has been standardized by the OMG. It is intended to replace the existing UML Profile for SPT profile [22]. MARTE consists in defining extensions that provide high-level modelling concepts to deal with RT and embedded features modeling as well as specific modeling artifacts to be able to describe both software and hardware execution supports.

Another work proposed the *UML-RTDB* profile [23] to express real-time database features in a structural model. Unlike the previous profiles, it supplies concepts for real-time database modeling such as RT attributes, RT methods and RT classes. In addition, UML-RTDB specifies two kinds of real-time attributes, *sensor* attributes and *derived* attributes, in order to satisfy the requirements of current real-time applications. However, some proposed stereotypes overlap with the UML extensions presented by MARTE profile especially those relative to the RT methods. In fact, the UML-RTDB stereotypes *<<Periodic>>*, *<<Sporadic>>* and *<<Aperiodic>>* that express respectively periodic, sporadic and aperiodic methods in the class diagrams, has the same meaning as the tagged value *Occurrence Kind* of the *<<rtFeature>>* stereotype defined in MARTE. Thereby, we adapt some MARTE stereotypes modeling RT aspects instead of the other UML extensions proposed for the modeling of RT applications since MARTE is a standardized profile.

Nevertheless, the only use of UML notations modeling RT application characteristics is insufficient to specify RT design patterns. That is, RT patterns must be generic designs intended to be specialized and reused by any application in RT domain. For this reason, in addition to the UML extensions representing RT aspects, we need new notations distinguishing the commonalities and differences between applications in the pattern domain. Moreover, we need new concepts for the explicit representation of the pattern elements roles for the traceability purpose.

In the next section, we describe the extensions that we propose to take into account these new concepts.

3 The UML profile for RT design patterns

In the present work, we extend the unified modeling language "UML 2.1.2" [9] to represent design patterns for RT applications. These extensions allow (i) to express the variability in a pattern, (ii) to identify the roles played by each pattern element in the application instantiating it and (iii) to specify RT applications constraints and their non functional properties. The proposed extensions are described in the next section.

3.1 UML extensions for specifying domain-specific patterns

In this section, we propose new stereotypes showing the optional and fundamental elements participating in a pattern and assisting the designer in pattern reuse. Thus, the class diagram Metamodel is extended with the following stereotypes:

• **Stereotype <<*optional*>>** (applied to the *Feature* UML Metaclass): This stereotype is inspired from <<optional single>> and <<optional many>> stereotypes defined in [18]. In fact, the variety of applications within the RT domain is quite large. For this reason, we can not specify **exactly** how many times a pattern element can appear in a specific RT application. Thus, we use <<optional>> stereotype to represent the optional features (i.e. attribute or method) that can be omitted in a pattern instance.
Each method or attribute which is not stereotyped <<optional>> in a fundamental classifier (i.e. class, interface …) means that it is an essential element that plays an important role in the pattern.

• **Stereotype <<*mandatory*>>** (applied to the UML Metaclasses: *Class, Association, Interface, Lifeline and ClassAssociation*): This stereotype is inspired from <<mandatory single>> and <<mandatory many>> defined in [18]. We propose the <<mandatory>> stereotype to specify a fundamental element (association, aggregation,…) that must be instantiated at least once by the designer when he models a specific application. For the clarity purpose, a fundamental element in the pattern is drawn with a highlight line like this class  .

Besides, each pattern element which is not highlighted means that it is an optional one, except the generalization relation that permits to represent alternative elements. All the attributes and methods of an optional class are implicitly optional.

• **Stereotype <<*extensible*>>** (applied to the UML Metaclasses: *Class, Interface and ClassAssociation*): This stereotype is inspired from {extensible} tagged value proposed in [8]. It indicates that the class interface may be extended by adding new attributes and/or methods. Moreover, two properties related to the extensible stereotype are proposed, in order to specify the type of features (attribute or method) that may be added by the designer.

- *extensibleAttribute* tag: It takes the value *false*, to indicate that the designer cannot add new attributes when he instantiates the pattern. Otherwise, this tag takes the value *true*.
- *extensibleMethod* tag: It indicates if the designer may add new methods when he instantiates the pattern. The default value is *true*.

• **Stereotype** <<***variable***>> (applied to the *Operation* UML Metaclass): This stereotype has the same meaning with the {variable} tagged value proposed in [8]. It indicates that the method implementation varies according to the pattern instantiation.

3.2 UML Extensions for instantiating domain-specific patterns

Some of the existing notations (Dong & Yang UML profile [3] and P-UML profile [8]) provide support on how to keep trace of the pattern when instantiated. These notations focus only on generic design patterns for which it is difficult to recognize the pattern instance when it is composed with others in a particular design. Thus, it is essential to hold the pattern name and the role played by each element (class, attribute and method) in the instantiation.

However, a domain specific pattern is instantiated in the scope of a domain. Therefore, it is easy to retrieve the pattern-related information even after the pattern is applied or composed with other patterns. We assume that omitting both the name and the role of pattern attributes and operations will not create any ambiguity. For this reason, we propose to present only the pattern name and the role names of the classes in order to avoid overloaded models. In fact, pattern-related information should be minimized in the class and sequence diagrams for readability [3].

We propose to define two new stereotypes for the explicit visualization of patterns in an application design:

• <<***patternClass***>> stereotype: It is applied to the *Class* UML metaclass in order to indicate that it is an instantiated pattern class and not originally defined by the designer. We propose to define two properties related to this stereotype:
- *patternName* tag : indicates the pattern name,
- *participantRole* tag : indicates the role played by the class in a pattern instance.

• <<***patternLifeline***>> stereotype: It is applied to the *Lifeline* metaclass in order to distinguish between the objects instantiated from the pattern sequence diagram and those defined by the designer. This stereotype has the same properties than <<patternClass>> stereotype.

These stereotypes allow to eliminate any confusion when patterns are composed. That is, when two or more classes represent the overlapping part of the composition, the proposed stereotype shows the roles that these classes play in each pattern.

3.3 UML extensions for modeling RT aspects

In addition to the above described stereotypes distinguishing the fixed parts from the optional and variable parts in the pattern, the specification of RT design patterns needs UML extensions supporting the modeling of RT aspects. Thus, we import stereotypes from HLAM (High Level Application Modeling) and NFP (Non Functional Properties) sub-profiles of MARTE [10] (cf. figure 1). Note that MARTE provides support required from specification to detailed design of RT embedded systems characteristics. However, only the extensions describing RT applications features at a high level of

abstraction are taken into account since RT patterns can be instantiated to model many RT applications and not only the embedded systems.

From HLAM sub-profile, we import the <<rtFeature>> stereotype in order to model temporal features. This stereotype extends the metaclasses: message, action, signal and behavioral features. It possesses nine tagged values among which: relD1 (i.e. specification of a relative deadline), absD1 (i.e. specification of an absolute deadline), Miss (i.e. percentage of acceptance for missing the deadline), occKin (i.e. specification of the type of event: periodic, aperiodic or sporadic)… . We propose to annotate each model element that has real-time features with the previously described stereotype.

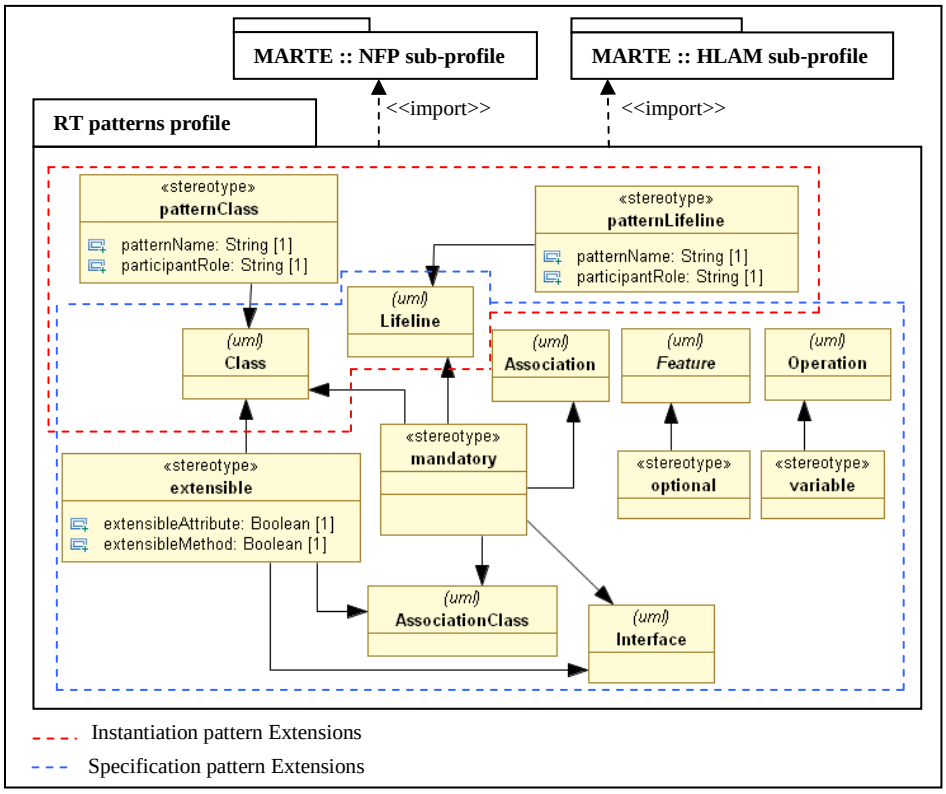

Fig 1. RT pattern profile Metamodel

From NFP Modeling sub-profile of MARTE, we import two stereotypes: <<Nfp>> and <<NfpType>>. The first one extends the Property metaclass. It shows the attributes that are used to satisfy non functional requirements. The second stereotype extends the DataType metaclass. There is a set of pre-declared *NFP_Types* which are useful for specifying NFP values, such as NFP_Duration, NFP_DataSize and NFP_DataTxRate.

In the following section, we illustrate the RT design pattern profile through the specification of RT controller pattern.

4 A RT design pattern example

In this section, we propose to illustrate the proposed extensions through an example of a reusable RT design pattern that explicitly shows the generic data which are fundamental and which represent the core of RT applications, on the one hand, and the allowed variants, on the other hand.

4.1 RT controller pattern

RT applications perform several RT processes among which: the RT data acquisition and the data control processes. We focus in this paper on modeling the static as well as the dynamic view of RT data control process through the definition of RT controller pattern.

- Interface:

Name: controller pattern

Context: This pattern is applicable in all RT applications which need to be managed by Real Time Database (RTDB) systems. In fact, a RTDB has all the requirements of traditional databases, but it also requires management of time-constrained data and time-constrained transactions [11].

Intention: The pattern aims to model the control of the data acquired from environment and the initialization of corrective action(s) if a violation is found.

- Solution:

Static specification: Figure 2 presents the controller pattern static view.

Participants*:*

- Observed_element: This class represents the description of a physical element that is supervised by the controller. It can be an aircraft, a car, a road segment, and so on. One or more measure types (i.e. Temperature, Pressure, etc) of each observed element could determinate its evolution. These measures are classified into either base measures or derived measures. Base measures stand for RT data that are issued from sensors, whereas derived measures stand for RT data that are calculated by the controller using base measures. The refreshment of each derived RT data is required every time one of the base data is updated.

The *ObservedElement* class has the *ElmentID* and *ElementStatus* fundamental attributes. In addition, it has an *UpdateStatus ()* method allowing to update the status of observed element according to the variation of the captured values.

- Controller: A controller has to monitor physical elements for responding to conditions that might violate safety. It takes periodically the value captured for each observed element as well as the minimum value and the maximum value that define the interval for which the controller does not detect an anomaly. If a captured value does not verify the boundary constraint, then the controller initiates some corrective actions, such as a reset and a shut-down, or sends an alarm to notify an operator.

On the other hand, the controller receives periodically an update message from an observed element to notify it about the modification of its measures. In this case, the

controller is waiting for a message. If this message does not arrive on time, then the controller performs appropriate recovery actions [14].
As illustrated in Figure 2, the controller class has four methods. The only fundamental method is *VerifyValue()*since it is essential to check that the boundary constraints are fulfilled for all RT applications. This method is performed periodically. In addition, it must be achieved before a deadline. Thus, the *VerifyValue()*method is stereotyped <<rtFeature>> in order to define the periodicity, the relative and absolute deadlines that are tagged respectively period, relDl and absDl. The method *CalculateDerivedValue()* is optional since it can be omitted in a pattern instantiation, when the designed application does not have derived measures. It is stereotyped <<rtFeature>> since it is sporadic and has to meet the deadline defined by the designer. The methods *notify()*, *initiateCorrection()* are optional since the choice of the appropriate recovery action depends on the application instantiating the pattern.

- Operator: The alarm signals sent by the controller are supervised by the operators. These latter provide decisions to validate reported incidents in case the controller only reports errors and does not have the responsibility to take further actions; or in case the confirmation of an operator is needed to achieve the correction.

The Operator class is optional since the controller can take the correction initiative without the intervention of an operator.

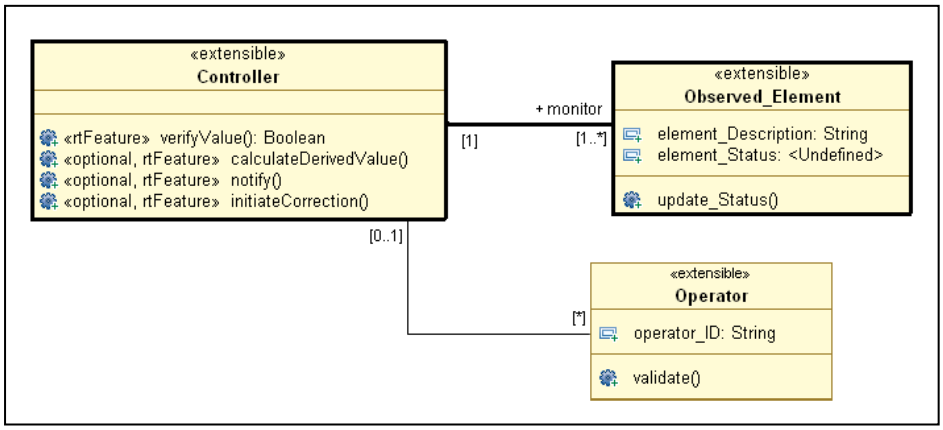

Fig 2. Specification of RT controller pattern static view

Dynamic specification: Figure 3 presents the controller pattern dynamic view.
In order to verify the validity of each observed element measure, the controller takes the current captured value and the value thresholds in parallel. Then, it verifies that each measured value is in the closed range [Minimum-Value, Maximum-value]. If this constraint is violated or the update message received from an observed element occurs too late, then the controller notifies the operator or initiates the appropriate recovery actions.

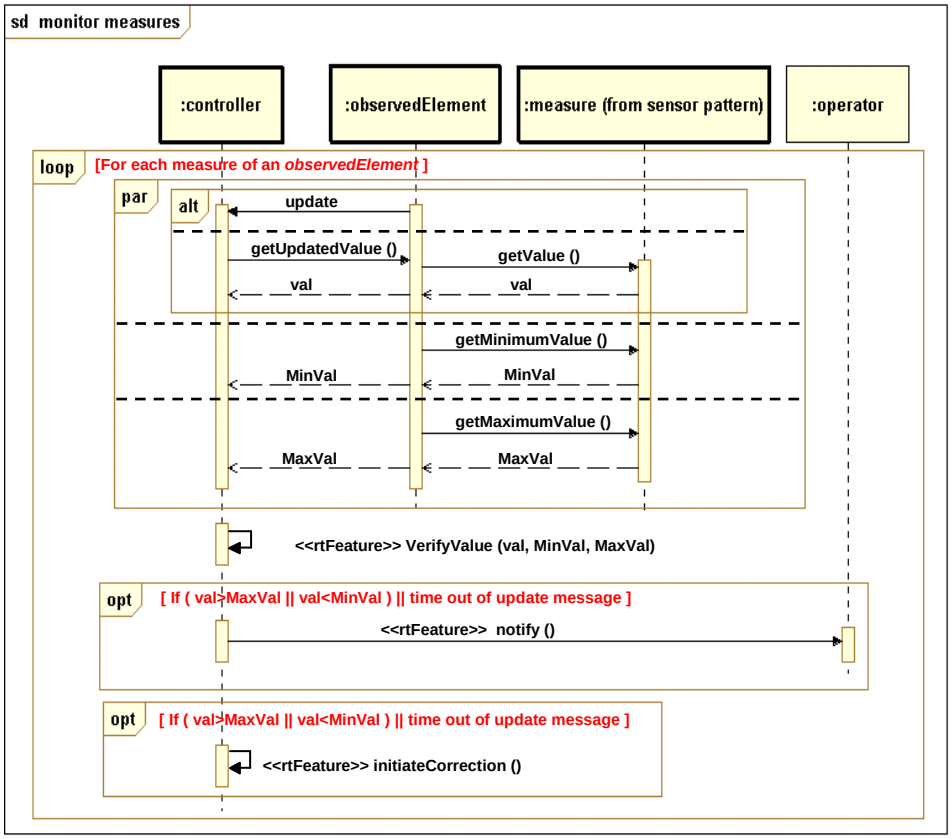

Fig 3. Specification of RT controller pattern dynamic view

4.2 RT sensor pattern instantiation: an example

This section proposes to illustrate the reuse of RT controller pattern through the design of freeway traffic management system.

The increasing road transport traffic and the incessant rise of the number of vehicles have caused a great growth of the magnitude of traffic flows on public roads. In consequence, freeway traffic management systems have become an important task intended to improve safety and provide a better level of service to motorists. We describe, in the following an example of a freeway traffic management system: COMPASS [19]. We focus precisely on modeling the compass control data subsystem and we explain how this design issue can be facilitated by the reuse of the RT controller pattern.

The current traffic state is obtained from the essential sources: inductance loop detectors and supervision cameras. In fact, vehicle detector stations use inductance loops to measure speeds and lengths of vehicles, traffic density (i.e. number of vehicles in a road segment) and occupancy information. Whereas, the supervision cameras are

used to supplement and confirm the data received through the vehicle detector stations and to provide information on local conditions which affect the traffic flow. The processed data are then transmitted at regular time intervals to the Central Computer System to monitor traffic and identify traffic incidents, when they occur.
Figure 4 illustrates the class diagram of the freeway traffic management system reusing RT controller pattern. It indicates that the *controller* monitors two types of elements (*Road_Segment* and *vehicle*). In addition, the *Operator* optional class is instantiated since it is essential to notify the operators of any detected events in the COMPASS system.

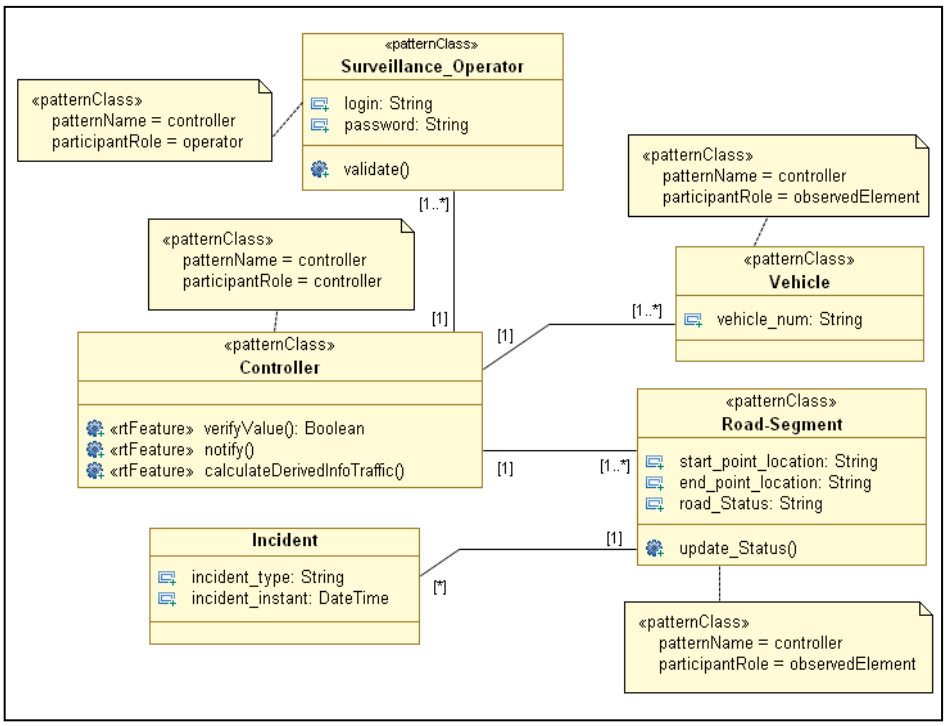

Fig 4. Example of controller pattern instance

5 Conclusion

The design of RT applications differs from the design of classical applications. RT applications have to guarantee that each action (transaction) meets its deadline, and that data are used during their validity interval. Thus, it is necessary (i) to give a great importance to RT applications design and (ii) to benefit from previous experiences of developers by reusing the knowledge previously acquired in the design practices. For this reason, dealing with RT domain engineering becomes a necessity since it allows to identify reusable patterns which reduce the complexity of RT applications design.

In order to represent RT design patterns in a more readable manner, this paper proposed UML-based extensions distinguishing clearly between the different parts constituting the pattern. These extensions help the designer in determining the variable elements that may differ from one application to another and allows to identify, easily, design patterns when they are applied to model a particular RT application. Besides, this paper proposed to guide the designer in modeling features specific to RT domain through the use of stereotypes imported from MARTE profile. These stereotypes provide facilities to model RT applications characteristics at a high abstraction level, being independent from the nature of tools used for the implementation of RT systems. The paper illustrated the proposed notations through the specification of RT controller pattern and its instantiation to design a freeway traffic management system.

Our future works include two axes. Firstly, we are looking into the formalization of RT design patterns. Secondly, we must examine how to integrate the design patterns in the context of the model driven architecture in order to add more assistance when generating models by reusing patterns. This could bring new benefits and impulse for both the knowledge capturing techniques and the software development process quality.

References

1. Gamma E., Helm R., Johnson R.E, Vlissides J., Design patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Edition, 1994.
2. Fowler M., Analysis Patterns – Reusable Object Models, Addison-Wesley, 1997.
3. Dong J. and Yang S., Visualizing design patterns with a UML profile, proceedings of IEEE Symposium on Human Centric Computing Languages and Environments, pp: 123-125, 2003.
4. Arnaud N., Front A.and Rieu D., Expression et usage de la variabilité dans les patrons de conception, Revue des sciences et technologies de l'information, série : Ingénierie des Systèmes d'Information, vol. 12/4, pp. 21-24, 2007.
5. Eden A.H., Gil J., Hirshfeld Y., Yehudai A., Towards a mathematical foundation for design patterns, Technical report, dept.of information technology, U.Uppsala, 1999.
6. Mikkonen T., Formalizing Design Patterns, Proc. 20th International Conference on Software Engineering— ICSE, pp. 115–124, 1998.
7. OMG, UML 2.0 OCL specification, 2003.
8. Bouassida N., Ben-Abdallah H., Extending UML to guide design pattern reuse, Sixth Arab International Conference On Computer Science Applications, Dubai, 2006.
9. OMG, Unified Modeling Language (UML) Infrastructure: v2.1.2, formal/2007-11-04, 2007.
10. OMG, A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, OMG document number: ptc/2008-06-09, 2008.
11. Ramamritham K., Real-Time Databases. Journal of Distributed and Parallel Databases, 1(2):199–226, 1993.
12. Ramamritham K., Son S., and DiPippo L., Real-Time Databases and Data Services. Real-Time Systems, 28:179–215, 2004.
13. Kim D.K., France R., and Ghosh S., A UML-based language for specifying domain-specific patterns, Journal of Visual Languages and Computing, pp. 265–289, 2004.
14. Douglass B. P., Real-Time Design Patterns: Robust Scalable Architecture for Real Time Systems, Addison-Wesley Edition, September 27, 2002
15. M. Amirijoo, J. Hansson, and S. H. Son. Specification and management of QoS in real-time databases supporting imprecise computations. IEEE Transactions on Computers, 55(3), 2006.

16. Yacoub S. M., Ammar H., Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems, Published by Addison-Wesley Professional, August 2003.
17. Czarnecki K., Eisenecker U.W., Generative Programming – Methods, Tools, and Applications, Addison-Wesley, 2000.
18. Reinhartz-Berger I., Sturm A., Utilizing domain models for application design and validation, Information and Software Technology, vol 51, pages 1275-1289, 2009.
19. COMPASS Website, Available from: http://www.mto.gov.on.ca/english/traveller/compass/main.htm
20. Douglass B. Real Time UML, Third Edition : Advances in The UML for Real-Time Systems. Pearson Education, Inc, 0-321-16076-2, 2004.
21. Lanusse A., G´erard S., and Terrier F.. Real-time modeling with UML: The ACCORD approach. In J. B´ezivin and P.-A. Muller, editors, The Unified Modeling Language, UML'98- Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers, volume 1618 of LNCS, pages 319–335. Springer, 1999.
22. OMG. "UML Profile for Schedulability, Performance and Time, v1.1", formal/2005-01-02, January 2005.
23. Idoudi N., Louati N., Duvallet C., Bouaziz R., Sadeg B. and Gargouri F., How to model a real-time database. Proceedings of 12th IEEE International Symposium on Object-oriented Real-time distributed Computing (IEEE ISORC'2009), Tokyo, Japan, pages 321-325, March 17-20, 2009.
24. Port D., Derivation of Domain Specific Design Patterns. USC Center for software engineering, 1998.

Bridging Programming Productivity, Expressiveness, and Applicability: a Domain Engineering Approach

Oded Kramer and Arnon Sturm
Department of Information Systems Engineering, Ben-Gurion University of the Negev
Beer-Sheva, Israel
odedkr@bgu.ac.il, sturm@bgu.ac.il

Abstract. Productivity is the ability to create a quality software product in a limited period with limited resources. The software engineering community advocates that the future of productivity lies in the field of domain engineering. However, existing domain engineering approaches suffer from the tension between productivity and applicability. In this paper we propose an approach that reduces this tension by adopting a domain engineering method called Application-based DOmain Modeling (ADOM) as an infrastructure for a new programming approach. The adopted ADOM is applied on Java as its underlying language. This approach will offer guidance and validation for application developers as mechanisms for improving their productivity. This is done by keeping the regular Java development environment and thus maintaining the developer's expressiveness and not compromising the overall applicability of the approach.

Keywords: Domain engineering, software productivity,

1 Introduction

Today's software development is a complex process involving a set of activities that require orchestration. One of the most resource consuming activities is programming. In order to better utilize the programming activity we should seek for ways to increase its productivity. Productivity according to [13] is "the ability to create a quality software product within a limited period with limited resources". The productivity of a programmer is affected by many factors. Jones [8] presented several of these: the design for reusability, experience, bugs or errors, management, creeping requirements, code structure and complexity, application size, supportive tools, and programming languages.

Many efforts have been made in order to increase the programmers' productivity from the technical point of view. These efforts are focused on providing techniques for increasing the code reusability, thus saving programming time. These techniques include generic programming which enables reuse by parameterizations, design patterns which provide solutions for specific situations, meta programming which enables programming at various levels of abstraction, as well as utilizing reflection

I. Reinhartz-Berger, A. Sturm, Y. Wand, J. Bettin, T. Clark, S. Cohen, J. Ralyté, and P. Plebani (Eds.): CAiSE 2010 Workshop DE@CAiSE'10, Hammamet, Tunisia, pp. 47-60, 2010.

mechanisms, and frameworks which provide partial design and implementations but are difficult to compose [3]. However, most of these efforts are related to general purpose reuse techniques, thus they do not exploit the commonalities among similar applications of a given domain.

Nowadays, the software engineering community advocates that the future of productivity lies in the field of domain engineering [3, 4, 11, 19]. According to Haarsu [7], domain engineering is a systematic process to provide common core architecture for similar applications. Its purpose is to provide reuse capabilities among these applications.

Indeed significant productivity achievements have already been reported [11, 12], but the quest for better software development solutions is far from over. We claim that one of the reasons for this is the inherent tension between productivity and applicability that current domain engineering approaches suffer from. Solutions that offer potentially promising productivity results tend to be expensive and require radical changes to the accustomed programming paradigms, thus their applicability is low.

A key factor that can aid in resolving this tension is expressiveness, which is the ability of developers to express desired semantics. Expressiveness is highly correlated with applicability. Solutions that reduce significantly the developer's expressiveness often require new development tools and processes. These tend to be expensive and require a learning curve that might seem to managers as risks that should be avoided. We assert that desired solutions should strive to keep the level of expressiveness as in general-purpose languages.

In this paper we propose an approach that aims at partially resolving the above mentioned tension by adopting a domain engineering method called Application-based DOmain Modeling [16, 17] (ADOM) as an infrastructure for a new programming approach. This approach offers guidance and validation for application developers as mechanisms for improving their productivity. The novelty of the proposed approach lies in using a standard programming language (including it supporting tools), thus maintaining the developer's expressiveness and increasing the applicability of the approach.

The structure of the rest of the paper is as follows. Section 2 discusses related work concerning DSLs and feature-oriented programming approaches, delving into the tension between productivity and applicability. Section 3 briefly introduces ADOM - the underlining framework of the proposed approach, which is presented in details in Section 4. Finally, Section 5 concludes and refers to future research directions.

2 Related Work

As domain engineering provides the platform for increasing productivity, in this section we analyze domain specific languages and feature-oriented approaches in view of the above mentioned tension.

2.1. DSLs

Domain Specific Languages (DSLs) are computer languages that are tailored to specific domains [11, 14]. This reduction to a specific domain allows the elevation of the language abstraction level. A higher abstraction level is a sought out goal in the fields of DSLs [6, 9, 11, 14]. It leads to many benefits such as: increased productivity, improved quality, better maintainability, and reuse of experts' knowledge. DSLs are divided into two distinct types: external and internal DSLs.

2.1.1. External DSL

The basic premise of external DSLs is that the underlying principles of a higher abstraction level and tailoring to specific domain necessitate the development of the DSL from scratch. Typically, there would be a domain expert whose expertise is on the semantics of the domain and an expert programmer whose expertise is on developing complicated and sophisticated software[1] working on this process [11]. The design process includes defining domain concepts and their relationships, semantics, notations, and constraints. The implementation process includes building a code generator, an optional domain specific framework, and the DSL's integrated development environment (IDE) which includes the DSL's supporting tools.

The main two advantages of external DSLs are the improved productivity; reports have shown of increase in productivity of 300%-1000% [11] and enhanced application quality; due to a preliminary check of the model's consistency according to domain rules. This means that many of the programmers' mistakes can be detected and thus can be avoided at this early stage of development. The developers specify the solution on a higher level, which is then transformed automatically to another form of code. This means that they can avoid dealing with important but complicated issues such as design principles and architecture, as these are handled by the code generator.

Yet, external DSLs suffer from various limitations. As mentioned, the design and implementation of external DSLs is by no means simple, it is complicated and time consuming. Even if the work is done by experts (both domain and programming), and some supporting tools are available it might not be enough to ensure a successful working DSL. According to [6] most DSLs are usually abandoned in the development process and the work is done eventually in regular general purpose languages. Additionally, to justify economically the investment of the DSL development process a quota of applications has to be exceeded. While this is true for all domain engineering techniques it is as harsh as the amount of emphasis that is put on the domain engineering process [6, 9]. Moreover, introducing the notion of DSL based development into an organization requires a significant change in the organization's development paradigm. This change requires both new tools and new processes. While some managers will be able to see the long terms advantages of DSLs, other might be reluctant to introduce radical, expensive and time consuming changes to

[1] obviously, they could be the same person, however both kinds of expertise are required

their natural development process. All of these indicate that the applicability of external DSLs is problematic.

Another limitation of external DSLs is the limited expressiveness of the application developer. Usually, this is considered to be an advantage – limiting the application developer's expressiveness means guiding him and controlling the quality of his work, and by that increasing his productivity and the overall quality of the end products. However, we consider this to be a disadvantage since the application developer has many constraints. The restrictions imposed on application developers are achieved by designating the domain to include a set of pre formulated commonality and variability. In case the application developers wish to express a newly encountered feature, they have to inform the DSL developers to update the DSL and wait for the change to be done. This process is time consuming and more importantly will make the procedure of incorporating new variants into the domain difficult, ultimately leading to narrow domains. Furthermore, this limitation is directly linked to the necessity to incorporate new tools and processes which lead to the problematic applicability of the DSL.

2.1.2. Internal DSLs

Internal DSLs drew their inspiration from the recognized drawbacks of external DSLs. Their basic premise is that DSLs should not be developed from scratch; rather they should be embedded on existing proven general purposed programming languages (GPPLs). In this sense internal DSLs are no different than regular domain specific application programming interfaces. However, they are different in the sense that the APIs are designed to have a language like flow to them. This is achieved by advanced coding techniques such as method chaining, expression builders, interface chaining, generics, etc. When these techniques are used correctly some domain semantics could be validated in compile time.

The main advantage of internal DSLs is that they do not suffer from the above mentioned drawbacks of external DSLs. This is caused by three main reasons: (1) The development of internal DSLs is much easier with respect to external DSLs, mainly because the GPPL facilities already exist; (2) Internal DSLs do not necessitate a radical change in the organization's natural development paradigm as they permit using the same set of tools (such as a programming languages, IDEs, and compilers); and (3) Internal DSLs do not limit application developers' expressiveness as they are allowed to use the GPPL regularly. These reasons indicate that internal DSLs are more applicable than external DSLs.

However, internal DSLs introduce the following limitations: (1) Current reports [6, 9] of internal DSLs focused on code readability and maintainability. Although this should have positive effects over productivity it is hard to see how sophisticated APIs raise the level of abstraction similarly to external DSLs; (2) External DSLs achieved improved code quality through pre code generating validation algorithms and higher abstraction levels. Although internal DSLs can exploit coding techniques to assure some domain semantics, they cannot implement validation algorithms that examine the specified code according to domain constraints. Ultimately, application

programmers can use the API in any desirable way. Thus, in that sense internal DSLs are less productive than external DSLs.

2.2. Feature-oriented approaches

Feature oriented approaches rely on features which are system properties that are relevant to the stakeholders and are used to capture commonalities or discriminate among systems in a product family [3]. The various approaches consist of a feature model that contains all features covered by the product family along with their dependencies and their variability [15]. Each application will be comprised by a unique subset of the features presented in the feature model. Typically, the feature model will be expressed using the tree diagram that was firstly introduced by the Feature-Oriented Domain Analysis (FODA) method [10].

The different feature oriented approaches focus on different levels of abstractions and on different stages of the development cycle. For example: FODA focuses on the domain analysis phase, Hyper/UML [15] and the work presented in [5] focus on feature oriented design by mapping features to other models (e.g., UML models). Feature Oriented Programming (FOP) [1] and HyperJ [18] focus on mapping features to code increments.

Many feature-oriented approaches suffer from the tension that was presented in the previous section. For example, feature modeling can help facilitate DSL design and DSLs may be used to specify the family members [4]. In that case, the applicability of the feature-oriented approach is problematic, similarly to that of the external DSLs. Furthermore, some approaches limit the expressiveness of application developers to the extent of only selecting appropriate features that are mapped automatically to code pieces (e.g., FOP and HyperJ [15, 18]). These, as in external DSLs, also may suffer from extensive domain engineering efforts, radical changes to the programming paradigm and narrow domains which will presumably lead to poor applicability.

To overcome the aforementioned limitations with respect to the tension between productivity and applicability, we utilize a domain engineering approach called Application-based Domain Modeling (ADOM).

3 The ADOM Approach

The Application-based Domain Modeling (ADOM) is rooted in the domain engineering discipline [16, 17], which is concerned with building reusable assets on the one hand, and representing and managing knowledge in specific domains on the other hand. ADOM supports the representation of reference (domain) models, construction of enterprise-specific models, and validation of the enterprise-specific models against the relevant reference models.

The architecture of ADOM is based on three layers: The ***language layer*** comprises metamodels and specifications of the used languages. The ***domain layer*** holds the building elements of the domain and the relations among them. It consists of

specifications of various domains; 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 structure and the behavior of the domain layer are modeled using the language that was defined in the language layer. The ***application layer*** consists of domain-specific applications, including their structure and behavior. The application layer is specified using the knowledge and constraints presented in the domain layer and the constructs specified in the language layer. An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain should be applied in any application of that domain. In order to achieve this goal, any element in the application is classified according to the elements declared in the domain.

For describing variability and commonality, ADOM uses a multiplicity indicator that can be associated to all elements, including classes, attributes, methods, and more. The multiplicity indicators in the domain aim to represent how many times an element of this type may appear in an application. This indicator has two associated tagged values - min and max - which define the lowest and the upper most multiplicity boundaries.

The relations between a generic (domain) element and its specific (application) counterparts are maintained by a classification mechanism: each one of the elements that appear in the domain can serve as a classifier of an application element of the same type (e.g., a class that appears in a domain may serve as a classifier of classes in an application). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain. Some optional generic elements may be omitted and not be included in the application, while some new specific elements may be inserted in the specific application; these are termed application-specific elements and are not classified in the application.

ADOM also provides validation mechanism that prevents application developers from violating domain constraints while (re)using the domain artifacts in the context of a particular application. This mechanism also handles application-specific elements that can be added in various places in the application in order to fulfill particular application requirements.

While ADOM is general and language-independent, a specific 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 language is to have a classification mechanism that enables categorization of elements.

4 The ADOM-Java Dialect

Since we refer to programming, in this paper we select Java as the language used in conjunction with ADOM. We will refer to that ADOM dialect as ADOM-Java. In this case, the required classification mechanism will be fulfilled by Java's annotation construct due to its meta data qualities. Listing 1 demonstrates the usage of the Java annotation in both the domain and application layers. In the domain layer the

multiplicity indicator is used to constrain the domain's applications to have classes classified as someDomainClass at least A times and no more than B times. This type of constraints in ADOM is referred to as the multiplicity constraint. In the application layer the someClassApplication class is classified by the someDomainClass class.

```
// domain layer code
@multiplicity(min = A, max = B)
public class someDomainClass {
    …
}
// application layer code
@someDomainClass
public class someApplicationClass {
    …
}
```

Listing 1: The Java annotation classifications

4.1. Structural constraints

Using the multiplicity indicator one can express a great deal of the structural commonality and variability captured and identified in the domain. For example, small scale information systems based on three layered architecture may be considered as a domain.
Applications in that domain use a relational DBMS, the JDBC API to interface with it, and the Java Swing API for the presentation layer. Applications in that domain may include a conference management system, a university registration system, and a laboratory management system.

In Figure 1, the applications of a conference management system and a laboratory management system are depicted along with their corresponding domain[2]. In this case the domain layer consists of five different types of classes GUI, Controller, and DBmapper, which represent the three classic layers, and SingleStatedObject and, MultiStatedObject which represent domain elements that have a single state or more, respectively.

In Listing 2, it is shown that the applications are expected to have exactly one class classified as a controller. This is indicated by the multiplicity annotation assigned to the class declaration as noted above. Moreover, it is shown that this class must have exactly one field classified as db, which is of a type that is classified as a DBmapper. This is noted by the DBmapper type of db in the domain code. This is effectively the composition relationship between these two classes that is shown in Figure 1. If the matching application field will be of any other type it will be a

[2] Note that domain models in ADOM-Java are expressed in Java. In Figure 1 we use UML to visualize the structural outline that was extracted.

violation of the code presented above. This type of constraints goes one step further than the multiplicity constraint as it deals with the syntactic structure of the application, for this reason it is referred to as the language constraint.

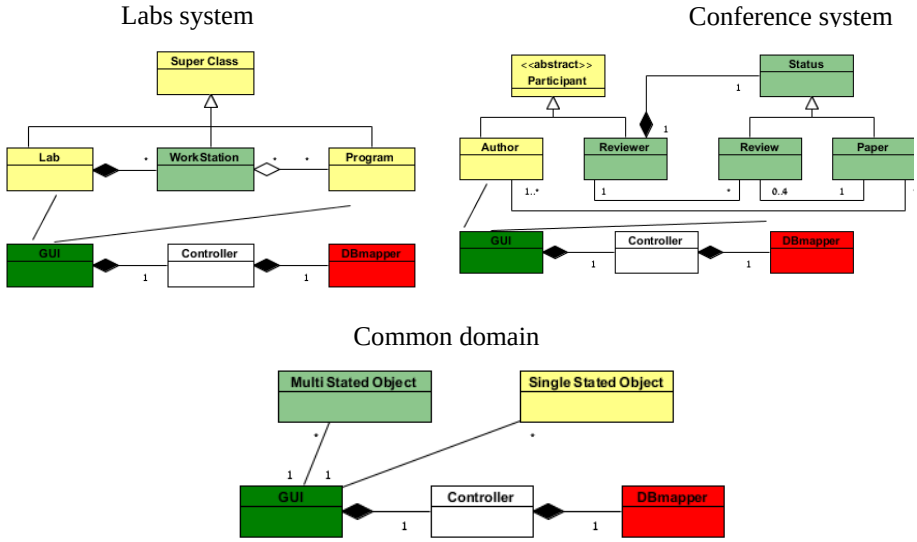

Figure 1: example of two applications and their common domain

The rest of Listing 2 expresses three methods, each of which are expected to appear at least once in the controller class. All of these methods should be pubic, the last two should return the boolean primitive type, and the first one should return a type classified as SingleStatedObject. These constraints are indicated by the methods' modifiers. Any other setting will be a violation of the language constraint. These method attributes are stated explicitly, however, those that are stated implicitly will be constrained by the language constraint as well. Effectively, all of the application methods classified as one of this three will have to be non-static and non-final, moreover the field that will be classified as db will have to be private-package.

ADOM-java enables to use other indicators to raise the flexibility of the language constraint. For example, the addDomObject method which is responsible of adding new objects to the system and returns the newly added object should be able to return both the SingleStatedObject and the MultiStatedObject types. This requires a correction to the code represented in Listing 2 as shown in Listing 3.

The typing indicator expresses this. It should be noted that the indicator overrides the return type. This was used because there are no multiple return types in Java. It is important to notice that this is not to say that the respected application method will return both types, indeed it will return a single type, as accustomed in Java. However, this type will be either classified as SingleStatedObject or as MultiStatedObject. This is just one example of additional indicators that raise the flexibility level of the

language constraint. Others can be used to express that methods can be of a combination of different access levels, final or non-final, and static or non-static. Actually ADOM-Java supports all the Cartesian products of the different members' modifiers.

```
//domain layer code
@Multiplicity(min = 1, max = 1)
public class Controller {

 @Multiplicity(min = 1, max = 1)
  DBmapper db;

@Multiplicity(min = 1)
 public SingleStatedObject addDomObect(String...
ObjectsData)

@Multiplicity (min = 1)
 public boolean addDomainAssociation ()

@Multiplicity(min = 1)
public boolean changeStatDomObj(MultiStatedObject mso) }
```

Listing 2: The controller class in the domain layer

```
//domain layer code
@typing ({" SingleStatedObject ", " MultiStatedObject "})
@Multiplicity(min = 1)
 public singleStatedObject addDomObect(String…
ObjectsData)
```

Listing 3: the addDomObject method from Listing 2 with the typing indicator

Listings 2 and 3 are neither a complete description of the entire domain model, as the other 4 classes from Figure 1 are missing, nor a complete description of the entire controller domain class. The full implementation of this class has more methods and goes into the methods declarations themselves.

The matching application code regarding its controller class from the labs management system is presented in Listing 4.

First of all, the AppController class is classified as the Controller class from the domain; this is noted by the Controller annotation assigned to the class declaration. Following there is a field declaration which is classified as the db field from Listing 2. If the AppMapper type will be classified as DBmapper (not shown here) the language constraint will be fulfilled. Following, there are two methods declarations classified as addDomObect. Their public, non-final, and non-static modifiers indicate an adherence to the language constraint in Listing 2. Their return types' classifications are not

shown here as well, however in the full implementations they are of type SingleStatedObject and MultiStatedObject and therefore correct. These methods are responsible for adding two (Lab and Program) of the three classes that were presented in Figure 1 to the labs management system. The number of these methods indicates an adherence to the multiplicity constraint in Listing 2. Following, there is a method classified as changeStatDomObj which adheres to both the multiplicity and the language constraints on Listing 2. Finally, there is the reportMalfuctionWS method, which is responsible of changing the state of workstation from functional to malfunction, removing the workstation from its lab and updating the DB if the change took place, and which has no classification. This method does not match any of the domain's method types; therefore it can be of any linguistic structure and multiplicity, and is considered as an application specific extension.

```
//application layer code
@Controller
Public class AppController {

   @db
   AppMapper appDB;

   @addDomObject
   public Lab addLab()
   @addDomObject
   public Program addProgram()

   @changeStatDomObj
   public boolean fixWS(WorkStation ws)

   public boolean reportMalfucntionWS(WorkStation ws)
```

Listing 4: The matching application code with respect to Listings 2 and 3

The same goes for all other Java constructs: classes, fields, etc. Obviously, the code presented in Listing 4 is not a complete description of the application's controller implementation. Some method declarations were omitted and the bodies of the methods were not presented due to space limitations. These will be presented and elaborated in the next section

4.2. Behavioral constraints

In the previous section we presented how the multiplicity and language constraints are used to impose structural knowledge over the applications. Language constraints by nature cannot be extended to behavioral knowledge as they refer to syntactic structure of the Java constructs. However, the notion of multiplicity constraints can be

introduced to behavioral aspects as well. This will be shown by yet another drill down, this time to the controller's changeStatDomObj method as appears in Listing 5.

```
//domain layer code
@Multiplicity(min = 1)
public boolean changeStatDomObj(
@Multiplicity(min = 1, max = 1) MultiStatedObject mso) {

   @Multiplicity(min = 1, max = 1)
   if (dso.changeState()) {
        @Multiplicity(min = 1, max = 1 )
       db.updateDomainObject(dso);
           @Multiplicity(min = 1, max = 1)
            return true;
        }
   @Multiplicity(min = 1, max = 1 )
   return false;
}
```

Listing 5: The controller's changeStatDomObj method

First of all, this Listing presents this method's signature as it appeared in Listing 2 with the addition of the multiplicity indicator to the received parameter. This method's responsibility is to receive a business logic object, to change its internal state, update the DB if the transition was successful, and finally to return a Boolean statement indicating whether the action was successful or not. For this reason, the matching application methods will have to receive a single parameter classified as MultiStatedObject.

This is noted by the type of the mso parameter and by its multiplicity. Therefore, this example illustrates that constraints over methods' parameters can be defined in the same manner as over classes' fields. Following, inside the body of the method, there are four execution statements, each with a multiplicity[3] indicator constraining the statement to appear once. This specification constrains any application method classified as changeStatDomObj to contain each of these four statements exactly in the order as they appeared in the domain and with the same scoping structure, with the exception of method calls. Each method call in the domain will be replaced in the application code by a call to a method that is classified as the called method in the domain. For example, dso.changeState() method call in Listing 5 is replaced by the p.accept call in Listing 6. This is correct only because p is of type Paper (as presented in Figure 1), which is classified as statedObject and accept() is a call to its method

[3] Notice that this use of Java annotation is not supported in standard Java and requires an extension called @Java [2].

that is classified as changeState(). Thus, the code in Listing 6 adheres to behavioral constraints specified on Listing 5.

```
//application layer code
@changeStatDomObj
 public boolean acceptPaper(Paper p)  {
        if (p.accept()) {
            db.updatePaperStatus(p);
            return true;
        }
        return false;
 }
```

Listing 6: an application method that adheres to the method presented in Listing 6

The example in Listing 6 presents an application method that did not introduce application specific statements. These statements could have been introduced anywhere in method body as long as the constraint mentioned above would not have been violated.

4.3. Extension constraints

Up until now we presented two different kinds of types: primitive types that are part of the language, and domain classified types, which means types that are classified as one of the classes from the domain layer (Figure 1 presents these). However, there is a third kind, types that belong to horizontal domains which are parts of software systems that can be classified according to their functionality [4]. Examples of these types are those from the Swing, JDBC, and collection APIs. Listing 7 presents the DBmapper class (figure 1) which uses this kind of types.

```
//domain layer code
@Multiplicity(min = 1, max = 1)
public class DBmapper {

    @Multiplicity(min = 1, max = 1)
    Connection connection;
}
```

Listing 7: Horizontal domain types

This Listing specifies that a matching application class will have a single field named connection of type Connection (a type from the JDBC API). This is not to say that the Application's respected field will be of a type classified as connection, as was

demonstrated in Listing 3, rather that the type itself will be Connection. In fact, this is a specification of how to interface with JDBC; in this case to (re)use by composition of the Connection class

An application class that will be classified as DBmapper will violate the specification in Listing 7 if it doesn't have a single connection field of type Connection. ADOM-java offers an additional way to constraint APIs extension. This is referred to as the extension constraints. The usage of some APIs can be a quite a difficult task [18]. This has many reasons. For example, the volume of some of the APIs can be overwhelming (the Swing API has hundreds of classes). Moreover, inheriting from a framework necessitates an understanding of its inner structure. This can become quite difficult as the interdependencies of the classes force developers to learn all the classes at once rather than each class at a time. ADOM-Java realizes that for some domains only a small subset of the API will suffice. For example, of the entire Swing API only a dozen classes are used in the aforementioned applications. Here lies the motivation for the extension constraint (not shown here). It will be used to define if a framework class can be extended in the application and by which mechanism, where the possible mechanisms are: composition, inheritance, and none. For example, some Swing components can be found too complicated or unnecessary thus can be marked as not to be used for a given domain at all, others can be marked as not to be extended (i.e., used only by composition).

5 Summary

In this paper we presented the tension between productivity and applicability in common domain engineering approaches. We pointed that a key factor for reducing this tension is the expressiveness of the application developer. To address this tension, we utilize a domain engineering approach called ADOM based on the Java programming language for guiding the application developer by providing models that express the expected structure and behavior of the domain's applications. Moreover, ADOM-Java validates the developer's code according to these models. Thus, it enables error detections at an early stage of development. These factors, presumably will lead to increased productivity. Furthermore, ADOM-Java is embedded into a general purposed programming language (Java), thus it ensures that the expressiveness of the application developer will not be compromised and that the overall approach, as it does not necessitate radical expansive changes to the programming paradigm, will be applicable.

While ADOM-Java looks promising in bridging the gap between productivity, expressiveness and applicability. It is clear that additional examination is required. In the near future, we plan to conduct and experiment that aims at checking the applicability of ADOM-Java.

6 References

1. Batory, D., Sarvela, J.N., and Rauschmayer, A. Scaling step-wise refinement. Proceedings of the 25th International Conference on Software Engineering, 187–197, 2003.
2. Cazzola, W. @Java: A Java Annotation extension, http://homes.dico.unimi.it/~cazzola/atjava.html, 2010.
3. Czarnecki, K. and Eisenecker, U. W. *Generative Programming - Methods, Tools, and Applications*, Addison-Wesley, 2000.
4. Czarnecki, K. Overview of Generative Software Development. Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, September, 15–17, 2004.
5. Czarnecki, K. Mapping features to models: A template approach based on superimposed variants, in GPCE'05, Lecture Notes in Computer Science 3676, 422-437, 2005.
6. Freeman, S. and Pryce, N. Evolving an embedded domain-specific language in Java. In Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, 855-865, 2006.
7. Harsu, M. A survey on domain engineering, Report 31, Institute of Software Systems, Tampere University of Technology, 2002.
8. Jones, C. *Estimating Software Costs*, McGraw-Hill, 2007.
9. Kabanov, J. and Raudjärv, R. Embedded typesafe domain specific languages for Java. Proceedings of the 6th international Symposium on Principles and Practice of Programming in Java, 189-197, 2008.
10. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., 1990.
11. Kelly, S. and Tolvanen, J-P. Domain-Specific Modeling: Enabling Full Code Generation, Wiley, 2008.
12. Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. A software engineering experiment in software component generation. Proceedings of the 18th international Conference on Software Engineering, 542-552, 1996.
13. Lowell, A. J. *Programmer Productivity: Myths, Methods, and Murphology. A Guide for Managers, Analysts, and Programmers.* John Wiley and Sons, 1983.
14. Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. ACM Comput. Survev. 37 (4), 316-344, 2005.
15. Philippow, I., Riebisch, M., and Boellert, K. The hyper/UML approach for feature based software design. The 4th AOSD Modeling With UML Workshop, 2003.
16. Reinhartz-Berger, I. and Sturm, A. Enhancing UML Models: A Domain Analysis Approach, Journal on Database Management, 19 (1), special issue on UML Topics, 74-94, 2007.
17. Reinhartz-Berger, I., Sturm, A. Utilizing Domain Models for Application Design and Validation. Information and Software Technology, 51(8), pp. 1275-1289, 2009.
18. Tarr, P. and Ossher, H.: Hyper/J User and Installation Manual. In: Multi-Dimensional separation of Concerns: Software Engineering using Hyperspaces, 2001.
19. Weiss, D. M. and Tau, C., and Lai, R. *Software Product Line Engineering: A Family-Based Software Development Process*, Addison-Wesley, 1999.