LATVIJAS UNIVERSITĀTES
RAKSTI

756. SĒJUMS

Datorzinātne un informācijas tehnoloģijas

SCIENTIFIC PAPERS
UNIVERSITY OF LATVIA

VOLUME 756

Computer Science and Information Technologies

SCIENTIFIC PAPERS
UNIVERSITY OF LATVIA

VOLUME 756

Computer Science and Information Technologies

University of Latvia

LATVIJAS UNIVERSITĀTES
RAKSTI
756. SĒJUMS

Datorzinātne un informācijas tehnoloģijas

Latvijas Universitāte

UDK 004(082)
Da 814

Layout: **Ilze Reņģe**
English language editor: **Māra Antenišķe**

ISSN 1407-2157
ISBN 978-9984-45-200-5

Contents

SOFTWARE DEVELOPMENT

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 9–32 P.

Ontology-Based Application for Domain Rules Development

Diana Kalibatiene, Olegas Vasilecas
Information Systems Research Laboratory, Vilnius Gediminas Technical University
Saulėtekio av. 11, Vilnius, LT-10223, Lithuania
diana@isl.vgtu.lt, olegas@isl.vgtu.lt

While there is a great interest in rule-based systems and their development, none of the proposed languages or methods has been accepted as a standard technology yet. Nowadays tools used in process of information systems (IS) development are not extended and adapted enough for modelling and implementation of application domain rules. A particular contingent of researchers proposes using of ontology for development of intelligent IS, since ontology is suitable to represent application domain knowledge. We are challenged in using ontology for the development of application domain rules. In this paper we present a method for ontology axioms transformation to application domain rules and describe how ontology-based development of application domain rules is integrated through IS development life cycle.

Keywords: ontology, axiom, application domain rule, transformation, OCL, PAL.

1 Introduction

Nowadays, ontologies representing application domain knowledge are used for the development of modern information systems (IS). A number of authors believe that the use of such ontologies, transformed and/or translated to IS components, help to 1) reduce the costs of a conceptual modelling [1] and 2) assure the ontological adequacy of the IS [1, 2, 3]; and allow to 3) share and reuse a domain knowledge across heterogeneous software platforms [2, 4], and 4) cognise of an application domain. If the IS is a traditional one, application domain knowledge will be just embedded in the standard components of the IS. If it is going to be an *ontology-driven* (or *ontology-based*) IS, then a separate component – *application domain ontology* – will be developed and included in the IS [1].

In the step of an IS conceptual modelling, researchers are challenged to transform application domain ontology to a conceptual data model, since their conceptualisation of a real world is similar. Both see an application domain in terms of concepts, presenting entities of an application domain, relationships between concepts, properties of concepts and rules (in ontology axioms), presenting constrains of an application domain. While a number of approaches and methods for the transformation of application domain ontology to a conceptual data model have been proposed, like [3, 5, 6, 7, 8] etc, there is lack of a formal theory and methods of ontology components transformation to application domain rules.

In the IS development, there is a great interest in the development of the application domain rules. Authors of [9, 10, 11] etc, organisations, such as the Object Management

Group[1] (OMG), have motivated the application of rules. Methods and languages proposed for the development and implementation of application domain rules are: Unified Modelling Language (UML) [12] with Object Constraint Language (OCL) [13], Demuth et al. method [14], the Ross method [15], CDM RuleFrame [16], Semantic Web Rule Language (SWRL) [17], etc.

However, the results of survey presented in [18] shows that a) a large part of rule-based systems are created without any specific development process, b) almost half of the respondents use an integrated development environment (IDE) (such as the Ontostudio[2], Ilog Rule Studio[3], the Visual Prolog IDE[4] or the SWRL tab [19] of Protégé[5]) that allows to edit, load, debug and run rules. For editing rules the most widely used tools are still textual editors (33%), a simple text editor or a textual rule editor with syntax highlighting (28%) and graphical rule editors (26%); c) verification is dominated by testing (90%) and code review (78%). 74% of respondents do testing with actual data, 50% test with contrived data. Advanced methods of test organisation are used by a minority, with only 31% doing regression testing and 19% doing structural testing with test coverage metrics.

None of the proposed languages or methods have been accepted as a standard technology yet, since they are not suitable for modelling all types of rules, as presented in [20], or limit the opportunity for business people to change them because the verbalization of formal languages is not mature enough. Only a few of them deal with reuse of knowledge acquired in the analysis of a particular application domain and automatic implementation of rules.

Current tools used for the development of IS are not extended and adapted enough for modelling and implementing of rules. For example, in MagicDraw[6] OCL is used for defining constraints. However, there is no suitable interface to facilitate the definition of OCL constraints. User should be familiar with OCL. PowerDesigner[7] is suitable for modelling structural rules (using integrity constraints, foreign keys, domains, checks) only. There is no mechanism for defining and validating of dynamic rules.

In this paper we propose using a domain ontology for the development of application domain rules and describe how ontology-based development of application domain rules is integrated with IS the development life cycle. Therefore, Section 2 overviews the related works according to application domain rules and their implementation and a concept of an ontology, Section 3 presents the comparison of ontology axioms with application domain rules, Section 4 describes ontology-based development of application domain rules in IS life cycle, Section 4 presents the application of the proposed in the previous section method of ontology axioms to information processing rules and its implementation to the Axiom2OCL plug-in, and Section 5 concludes the paper.

---

[1] http://www.omg.org/
[2] http://www.ontoprise.de/en/home/
[3] http://www.ilog.com/products/businessrules/
[4] http://www.visual-prolog.com/
[5] http://protege.stanford.edu
[6] http://www.magicdraw.com/
[7] http://www.sybase.com/products/modelingdevelopment/powerdesigner

2 Development of Application Domain Rules

An enterprise system can be viewed as a three-layered system: the business systems, the IS and the supporting software [21]. Any enterprise consists of several business systems, e.g. it is doing several businesses. A business system consists of several IS, e.g. an IS is created to support a business system. Finally, software systems are created to support IS. Consequently, an enterprise system is effective only then all layers of this system are integrated properly. Concepts should be mapped rightly from higher-level system to lower level systems and lower level systems should be constrained by rules governing processes in higher level systems. Therefore, the concept of a rule is analysed according these three levels of abstraction: business system, IS, and software.

At the business system level, rules are statements that define or constrain some aspects of a particular business domain in a declarative manner. For example, *a customer could not buy more than her / his credit limit permits*. At the IS level, rules are statements that define information processing rules using a rule-based language, like OCL. Expressions of information processing rules are very precise, e.g. terms used in expressions are taken from the particular data model [22]. For example, the following OCL expression "*context c: Company inv enoughEmployees: c.numberOfEmployees > 50*" constrains the number of employees in the Company that must always exceed 50. At the software system level, rules are statements represented using language of a specific execution environment, like Oracle 10g[8], Microsoft SQL Server 2008[9], ILOG JRules[10], etc.

According to the presented definitions of rules and three levels of abstraction, rules can be expressed in three different forms [10, 23]. They are:

- informal – rules are expressed using natural language;
- semi-formal – rules are expressed using rule templates, decision trees, decision tables or a graphical modelling language, like UML or Object Role Modelling (ORM) [24]. Ideally, this form is the basic from which to generate executable rule code [10]. Unfortunately, there is no standard rule specification language. There are various rule languages proposed as part of other modelling approaches;
- formal – rules are expressed using a particular formal language, like OCL, or a rule execution language, like SQL in relational database management systems (DBMS). Rules expressed in formal way can be processed automatically.

Rules expressed in the natural language are well understandable for business people. However, these expressions are ambiguous and can be interpreted in different ways. Authors of [10, 23] propose using rule templates, to avoid ambiguity. However, they do not present any approach of implementing such rules and how rules expressed by rule templates could be transformed to executable form. Nowadays, majority of methods describe ways of transforming formal rules to executable rules. Therefore, the main question is – which language is the most suitable for the development of the complete and integral set of rules? Unfortunately, there is no standard describing rule acquisition from the application domain, rule modelling by a suitable language and rule implementation in an executable environment.

[8] http://www.oracle.com/technology/software/products/database/index.html
[9] http://www.microsoft.com/sqlserver/2008/en/us/overview.aspx
[10] http://blogs.ilog.com/brmsdocs/2008/06/15/ilog-jrules-6-for-architects-and-developers-2/

According to the implementation perspective, it is proposed to classify application domain rules as follows.

- *Structural rules* (terms, definitions, facts, and integrity constraints). Terms, definitions and facts can be implemented by elements of a conceptual data model, for example an entity in an entity-relationship model or a class in a UML class model. Therefore, terms, definitions, facts can be regarded as concepts in an ontology and not as rules. Integrity constraints can be implemented by integrity constraints, like referential integrity constraints, cardinality constraints, and mandatory constraints, of a conceptual data model and in case of UML models expressed as OCL invariants. At software system level, integrity constraints can be implemented like SQL assertions, checks, and foreign keys.
- *Dynamic rules*, which can be expressed by ECA rules and implemented using language of a specific execution environment, like SQL triggers. A dynamic rule is: 1) a dynamic constraint, which restricts transitions from one state of the application domain to another, 2) a derivation rule, which creates new information from existing information by calculating or logical inference from facts, or 3) a reaction rule, which evaluates a condition and upon finding it true performs a predefined action.

Since implementation of structural rules is defined quite precisely (it can be seen from the precise definitions of integrity constraints in a conceptual data model, like CHECK, DOMAIN, NOT NULL, referential integrity and other constraints), we concentrate our research on the implementation of dynamic rules.

Methods and languages proposed to model and implement application domain rules can be classified according to their drawbacks as follows.

1 *Non-existence of any graphical notation* – languages and methods of this category do not have any graphical notation. OCL and all OCL-based languages and methods, like the method presented in [25], CDM RuleFrame environment [16], have no graphical notation.

Nowadays UML is the most popular for modelling of business and information systems [26]. UML has graphical notation, which gives a wide range of possibilities for representing objects and their static and dynamic relationships. The most appropriate diagram for describing structural rules is the class diagram. OCL is proposed as a formal language to express dynamic rules, since UML diagrams are typically not refined enough to express those rules explicitly. While UML with OCL satisfy the requirements of formality, expressiveness, preciseness, and unambiguity, OCL does not have any graphical notation and thus does not account for an easily comprehensible language.

Commercial organisations, such as Oracle[8], also present their own methods and languages for rules modelling. In [16] CDM RuleFrame environment is presented, where special OCL subset called RuleSLang is developed to represent rules. Later this representation is used for the automatic enforcement of rules using Oracle technologies. The main drawback of this approach is the lack of a graphical notation (the same as with pure OCL) and the tight coupling with commercial products of one vendor.

2 *Non-explicit implementation* – languages and methods of this category do not deal with a way rules be implemented (automated, semi-automated or manual). It is expected that many rules specified by the proposed language will likely be enforced in an automated way; and in such cases, the semi-formal or formal language or method is proposed. The Ross method [15], rule templates presented

by [10] and OMG proposed "*Semantics of business vocabulary and business rules*" (SBVR) [23] can be referred to this category.

The Ross method [15] proposes specific constructs for each of the rule families together with a big number of accompanying constructs, such as special symbols, invocation values, special interpreters, and special qualifiers. However, a big number of modelling constructs makes the language quite complicated. Moreover, Ross does not define any format for the rule model representation and interchange.

In [10] rules are expressed by *rule templates*, which are combination of rule clauses. A simple *rule clause* is of the form *<term1> <operator> <term2>*. A *term* is a noun or a noun phrase with an agreed-upon definition. These include a concept (for example, a customer), a property of a concept (for example, customer-credit-rating-code), a value (for example, female) and a value set (for example, Mon, Tues, Wed, Thurs, Fri). An *operator* is any operator that makes sense for the particular term type. The subsequent terms and operators will exist only if they make sense. According to [10], structural rules are expressed using single rule clauses. For example, a fact can be presented by the template *<term 1> IS COMPOSED OF <term 2>* (for example, *Window IS COMPOSED OF frames*) and a mandatory constraint can be presented by the template *<term 1> MUST BE IN LIST <a, b, c>* (for example, *Gender MUST BE IN LIST <F, M>*). According to (von Halle 2002), a dynamic rule is a combination of rule clauses. For example, reaction rule or action enabler can be presented as *IF <rule clause> THEN <action>* (for example, *IF ordering data = current data THEN insert new record*).

The OMG in [23] "is focused on SBVR as a vehicle for describing businesses rather than their information systems". The OMG proposes to use logical formulations of rules or logical rules, which provide abstract, language-independent syntax for capturing the semantics of a body of shared meaning. These logical formulations are presented by statements and definitions of structured English. Statements are recognised by being fully expressed using the four font styles. For more details see [23]. However, authors of [23] do not define the basic patterns or templates for rule definitions. They just suggest which keywords should be used in rules and how expressions of application domain rules should look like.

3 *Limited type of rules* – languages and methods of this category is limited on modelling a specific type of rules.

In [14] the templates of rules are presented to generate SQL views and triggers, but the trigger action part is not automatically generated. A method presented in [25] is suitable generating triggers from consistency rules defined using OCL, but the authors limit the usage of method to consistency rules only.

4 *Suitable for rule implementation at the lower levels of abstraction* – languages and methods of this category deals with implementation of rules expressed in a formal way. These methods do not deal with elicitation of rules from the application domain. They use rules already expressed in a particular formal language.

Authors in [27] briefly describe currently used methods for generating relational database schemas, their limitations and drawbacks, and propose a method which advances them by generating full-fledged relational database schemas from a conceptual model. The proposed method consists of metamodel-based and pattern-based transformations.

Principles of creating pattern-based transformations are defined for transformation of OCL expressions to corresponding SQL code.

The particular methodologies were selected in [20] and compared according to the possibility of rule modelling. Authors show that common methods are insufficient or at least inconvenient for a complete and systematic modelling of rules. Some relevant enhancements of these methods are more powerful but still emphasise only certain aspects and types of rules.

In [28] authors present how rules can be managed in enterprises and propose the managing scenario. Future works of authors are detailed design of the rule repository, development of the necessary facilities for extraction of rules from organization's business model and specification of the necessary operations for integration of IS repository with the rule repository.

In [29] authors identify the issues that they think are problematic in the context of rule explicit manipulation and present challenges for future research. The focus was put on five areas: the rule scope, acquisition, specification, implementation, and management. For each of the areas authors pointed out the issues that present obstacles for using rules as an approach to IS development.

2.1 The Main Problems Concerned with Domain Rules Modelling

The process of developing the application domain rules involves two main problems – determining the rules (their elicitation from the application domain) and developing ECA rules (their implementation).

First of all, it is necessary to determine the rules of a domain and ensure that they are appropriate. The process of determining which rules apply to a particular situation often involves an open-ended search through multiple sources: business speech, documents, laws, an application domain ontology, etc [11]. A set of application domain rules can be defined using different approaches. The main of them are analysis of documents and questionnaire of business employees. The consensus from all the domain stakeholders should be obtained on the problem of which the rules and their meaning should be used. It is suggested to use business vocabularies or application domain ontologies to ensure the one meaning of an application domain and its rules. When the application domain changes the rules should be properly adapted to new conditions. Capturing, documenting and retaining the domain rules prevent the loss of knowledge when employees leave an enterprise [30].

After the set of appropriate application domain rules is defined, it is necessary to determine which rules will be implemented in a computerised IS. Not all application domain rules are implemented in a supporting computerised IS. These rules are defined in business' documents. Application domain rules, which are going to be implemented in a computerised IS, can be implemented in different ways: by information processing rules and correspondent executable rules of software, as a part of a program code, using rule engines, etc.

The large amount of works on application domain rules elicitation from a domain and implementation in IS shows that this is an important and relevant topic in IS development. However, the lack of a standard method or a language for application domain rules modelling in IS development means that it is not a straightforward problem.

In this paper we propose using ontology for application domain rule modelling and implementation.

2.2 Ontology and Information Systems

Two main directions of this branch may be defined. One is about developing of application domain ontologies and other is about using ontologies for the development of IS. The first one is analysed in ontology engineering field and is not going to be discussed in this paper.

According to [1], every IS has its own ontology, since it ascribes meaning to the symbols used according to a particular view of the world. N. Guarino [1] distinguishes two orthogonal dimensions in IS: a temporal dimension, concerning whether an ontology is used at *development time* or at *run time*, and a structural dimension, concerning the particular way an ontology can affect the main IS components, like application programs, information resources like databases and/or knowledge bases, and user interfaces.

In this paper, the main attention is placed on the usage of ontology at IS development. One of the major trends in this context is using ontology for conceptual data modelling, since a conceptual data model and an ontology both include concepts, relationships between them and rules (in ontology – axioms).

However, it is typically the case that in ontology-based conceptual data modelling approaches the process of developing domain rules is not defined in a formal manner.

Ontology defines the basic concepts, their definitions and their relationships comprising the vocabulary of an application domain and the axioms for constraining relationships and interpretation of concepts [31]. Some authors, like [32], also distinguish properties from concepts. In the simplest case [1], application domain ontology describes a hierarchy of concepts related by particular relationships (e.g., is-a, part-of, etc). In more sophisticated cases, constraints are added to restrict the values of concepts and relationships, like cardinality constraints, possible length, etc. In the most sophisticated cases, suitable axioms are added in order to express and restrict complex relationships between concepts and to constrain their intended interpretation.

In mathematics [33], an axiom is any starting assumption from which other statements are logically derived. It can be a sentence, a proposition, a statement or a rule that enables the construction of a formal system. Axioms cannot be derived by principles of deduction, because they are starting assumptions.

Following the terminology used in [32] and [34], axioms in ontology can be classified as epistemological, consolidation, and derivation axioms. *Epistemological axioms* are defined to show constraints imposed by the way concepts are structured. These include all axioms which can be directly included by the use of modelling primitives and relations that are used in a structural specification of ontology (e.g., is-a relation, part-of relations, cardinality constraints). An example of epistemological axioms imposed by the most basic form of a part-whole relation is: *if exists x and y and x is a part of y, then y is not a part of x* ($\forall x,y\ partOf(x,y) \rightarrow \neg partOf(y,x)$). *Consolidation axioms* impose constraints that exclude unintended interpretations over the structure of the ontology specification. An example of the consolidation axiom from a software quality ontology presented in [35] is: if a product quality characteristic (*qc*) is decomposed in subcharacteristics (*qc1*), then these subcharacteristics should also be a product quality characteristic ($(\forall qc,qc1)(subqc(qc1,qc) \wedge prodqc(qc) \rightarrow prodqc(qc1))(C1)$). Finally, *derivation axioms* allow new knowledge to be derived from the previously existing knowledge represented in the ontology. Typically, derivation axioms are created in order to derive information which can be used to answer the ontology competence questions. An example of a derivation

axiom from [35] states that "if there is not a paradigm to which a quality characteristic *qc* is applicable, than *qc* is paradigm-independent" $((\forall qc)\ \neg(\exists p)(applicability(qc, p) \rightarrow pdgInd(qc))$.

If it is necessary, the fourth type of axioms can be defined in addition. They are *definitional axioms* that define the meaning of concepts in ontology.

According to [36], implementation of axioms in ontology modelling environments is:

- restricted in a framework of a description logics [37] or in some kind of logic language, like Knowledge Interchange Format (KIF) [38] in Protégé ontology [39] and SUMO [40], or
- axiom modelling is completely neglected in WordNet [41], which can be used as a lexical ontology, Protégé ontologies (not all), ontologies presented by [42] and [43], DBpedia [44].

This situation is detrimental to the modelling of large-scale ontologies, because it aggravates engineering and maintenance of large sets of axioms [36].

Authors of [36] propose using of objects and categories to represent axioms. They state that categorisation of axioms allows representing the semantics of axioms, and specifying axioms like objects provides a compact, intuitively accessible representation.

Authors of [45] attempt to reduce the difficulty of writing axioms by identifying groups of axioms that manifest common patterns creating templates that allows users to compose axioms by "filling-in-the-blanks". The method for collecting the templates is also presented in [45]. This method is implemented in Protégé ontology development and management tool.

E. Sirin and J. Tao [64] inspired of growing usage of OWL analyse the possibilities of defining integrity constraint semantics for OWL axioms. Authors implement the proposal in the prototype using Pellet. Authors show that integrity constraints validation can be reduced to SPARQL (Query Language for RDF) query answering using off-the-shelf reasoning. They state that the obtained results show that the goal of using OWL both as a knowledge representation and constraint language for data validation can be achieved without too much effort.

The analysis of ontology development tools, like Protégé, and ontologies, like SUMO, from the implementation perspective shows that epistemological axioms are implemented by structuring concepts in an ontology; consolidation and derivation axioms are not distinguished and they are implemented using some languages suitable for this purpose, like Protégé Axiom Language (PAL) [46] or Ontology Web Language (OWL) [8]. Some consolidation and definitional axioms are implemented by restricting definition of concepts in a particular ontology.

3 Ontology Axioms in Comparison with Application Domain Rules

Here we present differences between ontology axioms, application domain rules, information processing rules, and executable rules, expressed in the form of *event-condition-action* (ECA) rules. This comparison is necessary to define a correct mapping of ontology axioms to application domain rules.

As stated in Section 2, at the IS level rules are statements that define information processing rules using a rule-based language, like OCL, etc. They are taken from the business system level and implement application domain rules. Information processing rules should be precise and expressed as ECA rules to be implemented by executable rules. Therefore, it is necessary to develop ECA rules, which define when the rule should be applied, what should be checked and what to do after checking.

Application domain ontology axioms belong to a particular application domain. They define admissible states of a domain. In particular cases axioms can have conditions under which defined states should be taken.

Table 1 presents the comparison of rules and ontology axioms.

According to this comparison, the following conclusions could be done.

Since axioms can be formalised together with a domain ontology using a particular language, it is reasonable to use this formalisation to automatically transform the ontology axioms to information processing rules or even to executable rules.

Table 1

Ontology axioms in comparison with rules

Criteria of comparison	**Ontology axiom**	**Application domain rule**	**Information processing rule**	**Executable rules**
Level of abstraction	Application domain	Application domain	Information system	Software system (executable environment)
Level of formality	Formal	Informal	Formal	Formal
Languages used to define	PAL, OWL, logic	Natural language	OCL, RuleML, ORM, rule templates, decision trees, decision tables, etc.	A rule execution language, like SQL in relational DBMS
Event	Holds in all cases	Not defined	Insert, update, delete, select	Insert, update, delete, select
Condition	A predicate or a query over the ontology	Explicit or implicit	A predicate or a query over the data model	A predicate or a query over the data model
Action	No action	Explicit or implicit,	Data modification, application specific procedures, transaction operations	Data modification, application specific procedures, transaction operations
State	Predicate over the ontology	Explicit or implicit	No state	No state
Definition	Defined using ontology concepts	Defined using natural language	Defined using data model terms	Defined using data model terms

Protégé axioms and axioms from [35] and [46] were analysed and it was determined that consolidation and derivation axioms have structure *state* or *condition-state*.

A *state axiom* clearly defines a state in which a domain should be and which can be transformed to the condition of an ECA rule. An action can be understood in two ways:

1 if the condition is satisfied, then the transition from one state of the system to another is admissible;
2 if the condition is not satisfied, then the transition is forbidden.

An example of a state axiom, defined by PAL, is presented as follows. It constrains that the number of pages in a newspaper should not exceed 30. This axiom defines a possible state of a newspaper in a domain, i. e. it defines that for all instances of a class newspaper an attribute number_of_pages should not exceed 30.

```
defrange ?Newspaper :FRAME Newspaper
   forall ?Newspaper
(> (number_of_pages ?Newspaper) 30))
```

A *condition-state axiom* defines an admissible state of a domain under the defined condition. In the sense of the ECA structure, a condition-state axiom can be transformed into an ECA rule in two ways:

1 the condition of an axiom is transformed to the condition of an ECA rule, the state of an axiom is transformed to the action of an ECA rule;
2 the condition-state axiom is transformed to an ECA rule as in the case of a state axiom.

An example of a condition-state axiom, defined by PAL, is presented as follows. It constrains that only finished Content (an article or an advertisement) can be included in a Newspaper. This axiom defines a possible state of a newspaper under the defined condition, i. e. it defines that content (an article or an advertisement) can be included in a newspaper. However, it should satisfy a condition – it should be finished.

```
defrange ?Content :FRAME Content
defrange ?Content-SlotVal :FRAME Content 'published_in'
forall ?Content (forall ?Content-SlotVal
   (=> (not('isFinished' ?Content \"must contain\"))
   (instance-of ?Content-SlotVal Newspaper)))
```

Axioms hold in a domain in all cases. However, computer systems should have information when they apply rules. Therefore, according to the structure of an ECA rule, it is necessary to define important events and link them with corresponding rules during the transformation of ontology axioms to information processing rules or executable rules.

4 Ontology-Based Development of Application Domain Rules and IS Life Cycle

This section presents the method of transforming ontology axioms into information processing/executable rules and its mapping to IS development life cycle.

4.1 Transforming Ontology Axioms into Information Processing/Executable Rules

According to the results obtained in Section 2, Fig. 1 presents the basis for ontology axiom-based modelling of application domain rules. The comparison of ontology

axioms and application domain rules shows that a) consolidation axioms can be used to model dynamic constraints and / or reaction rules; b) derivation axioms – derivation rules, c) epistemological axioms – the structuring of entities in a conceptual data model, and d) definitional axioms – definitions of entities.

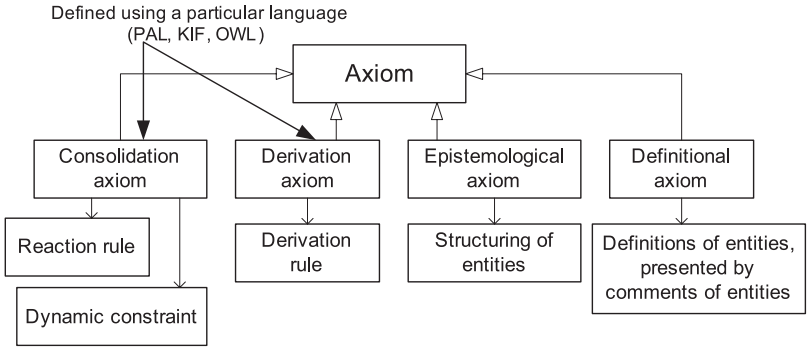


Fig. 1. Ontology axiom-based modelling of application domain rules

The main steps of applying the method of transforming ontology axioms into application domain rules are as follows (Fig. 2).

1 Check if axioms are in an ontology. It warranties that axioms are in an ontology. Otherwise, a user should define axioms.

Note that the creation of an ontology is not analysed here, since, it is not the topic of this paper. The method is based on the assumption that a user of the method has a necessary ontology.

2 Find an axiom.

3 Transform an axiom into a corresponding ECA rule:

3.1 define an event of an ECA rule as insert, update or delete;

3.2 determine the type of the axiom – is it a consolidation or a derivation axiom?

3.2.1 in the case of a consolidation axiom:

note that a consolidation axiom can be a state axiom or a condition-state axiom. However, in the both cases it is transformed to the condition of an ECA rule.

3.2.1.1 Transform an axiom to the correspondent condition of an ECA rule;

3.2.1.2 define an action as (a) if condition is true, then permit the change of a state in a domain, (b) if condition is false, then forbid the change of a state in a domain;

3.2.2 in the case of a derivation axiom:

note that a derivation axiom, which derives new information from the existing information, can be a state axiom or a condition-state axiom.

3.2.2.1 In the case of a state axiom – transform an axiom to the corresponding action of an ECA rule. A condition is always true.

3.2.2.2 In the case of a condition-state axiom: (a) transform a condition to the corresponding condition of an ECA rule, and (b) transform a state to the corresponding action of an ECA rule.

4 End of transformation.

The method is independent of particular languages, which can be used for the definition of axioms and application domain rules.

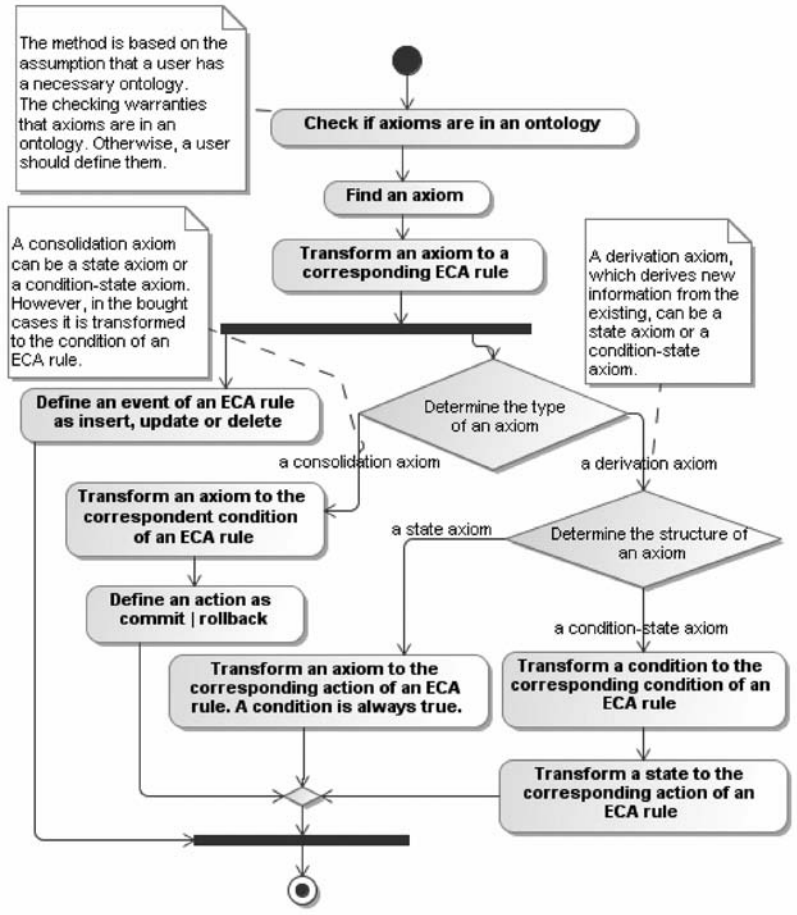


Fig. 2. The schema of the proposed method

The formal description of the method is presented in (Vasilecas et al., 2009) and is not discussed here.

4.2 The Mapping of the Proposed Method to IS Development Life Cycle

This sub-section presents the mapping of the proposed method to a system development life cycle. Fig. 3 presents the mapping schema. Business system is presented

by application domain ontology, which is created from business documents, laws and various knowledge sources [5]. This ontology with axioms is presented in a formal way, e.g. a particular formal language, like OWL, is used to define the ontology with axioms. Ontology axioms are used to present application domain rules, consolidation axioms are used to model dynamic constraints and/or reaction rules, derivation axioms – derivation rules, epistemological axioms – the structuring of concepts in the ontology, and definitional axioms – definitions of concepts in the ontology.

The ontology with axioms is transformed into the conceptual data model with information processing rules of an IS. The proposed method for transforming ontology axioms into information processing rules is used in this step. The method, described in [5, 8, 47], can be used to transform the ontology into a conceptual data model. However, it is necessary to integrate the obtained conceptual data model and information processing rules. Moreover, if both methods, the method used for the ontology transformation to a conceptual data model and the method used for the ontology axioms transformation to information processing rules, use the same conceptualisation of ontology, then we make an assumption that the obtained conceptual data model and information processing rules will be integral.

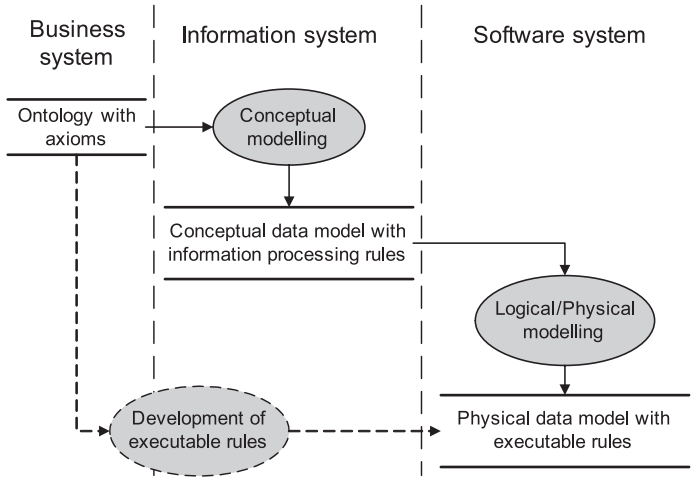


Fig. 3. The mapping of the proposed method to the system development life cycle

At the next step, the conceptual data model with information processing rules is transformed into the corresponding physical data model with executable rules. The transformation of a conceptual data model into a physical data model is not analysed here, since there exists a number of tools which support the automatic transformation of a conceptual data model into a physical data model, for example, Sybase PowerDesigner[7], Oracle[8], etc. However, it is important to discuss the possibility of transforming the ontology axioms into executable rules. Since the ontology with axioms is presented in a formal way, the proposed method can be adopted to transform the ontology axioms into executable rules. Such type of the experiment is presented in [31]. However, we believe that the transformation of ontology axioms into information processing rules is more

complete and correct, since ontology, in general, is closer to a conceptual data model than to a physical data model, but the transformation of ontology axioms to executable rules is also useful. It helps to facilitate the development of rules and ensure the same conceptualisation of rules at all levels of a system.

The obtained physical data model and executable rules can be implemented in an executable environment.

5 A Case Study of the Transformation of Protégé Axioms into OCL Constraints

This section presents the application of the proposed method for transforming ontology axioms into information processing rules.

5.1 Choosing an Appropriate Ontology Development Tool

First, a suitable ontology development and management tool (ODMT) should be chosen to apply the proposed method.

Currently there are more than 50 different ODMT. A number of authors, such as [48, 49, 50] etc, propose their criteria to assess different ODMT. However, the earlier proposed criteria of ODMT assessment are not enough, since they mainly concentrate on the modelling capabilities of the structure of an ontology and user interfaces. Therefore, according to the perspective of axioms, we select the following criteria, which will be used to analyse existing ODMT.

1 ODMT should support modelling of axioms:
  - 1.1 ODMT should support an axiom definition language.
  - 1.2 ODMT should support an axiom management language.
  - 1.3 ODMT should support syntactical checking of axioms.

2 ODMT availability – ODMT should allow free open source software, which can be installed locally.

3 ODMT usage:
  - 3.1 ODMT should be user-friendly.
  - 3.2 ODMT should support graphical notation.
  - 3.3 ODMT software should be supported by an active project.

4 ODMT should be extensible.

For a detailed study we chose the most popular *WebODE* [51, 52], *OilEd* [53, 54], *Ontolingua* [55, 56], *Protégé*, *Chimera* [57], *OntoSaurus* [58], *OntoEdit* [59] and *WebOnto* [60, 61] tools. The results of the ODML assessment according to the chosen criteria are presented in Table 2.

According to the results presented in Table 2, the Protégé ontology development and management tool is chosen to support our statement that information processing rules can be elicited from an ontology.

Table 2

Assessment of *WebODE, OilEd, Ontolingua, Protégé, Chimera, OntoSaurus, OntoEdit* and *WebOnto* ontology development and management tools

ODTM / Criteria	WebODE	OilEd	Ontolingua	Protégé	Chimera	OntoSaurus	OntoEdit	WebOnto
Axiom definition and management language (1.1, 1.2)	**Yes (WAB)**	**Yes (DAML + OIL)**	**Yes (KIF)**	**Yes (PAL)**	**Yes (KIF)**	**Yes (KIF)**	**Yes (F Logic)**	**Yes (OCML)**
Checking of axioms (1.3)	**Yes**	**Yes**	No	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Availability (2)	Free Web Access	[11]	Free Web Access	[11]	[12]	[12]	[13]	Free Web Access
User-friendly (3.1)	**Yes**	No	**Yes**	**Yes**	**Yes**	No	No	**Yes**
Graphical notation (3.2)	[14]	No	[15]	[14]	[14]	No	No	[14]
Active project (3.3)	No	No	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	No
Extensible (4)	No	No	No	**Plug-ins**	**Plug-ins**	No	**Plug-ins**	No

[11] Open source, installed locally.
[12] Open source and free Web access to evaluation version.
[13] Free Web access to free version. OntoEdit Professional needs software licence.
[14] Graphical taxonomy, graphical view.
[15] Graphical taxonomy, no graphical view.

5.2 Protégé Axiom Language (PAL)

Ontology axioms are implemented in Protégé ontology by the *Protégé Axiom Language* (PAL) constraints [46]. PAL is a superset of the first-order logic, which is used for writing strong logical constraints. PAL can be used to express constraints about a knowledge base, and it can be used to make logical queries about the contents of a knowledge base.

A PAL constraint (or a query) is a statement that holds on a certain number of variables, which range over a particular set of values. Therefore, a constraint or a query in PAL consists of a set of variable range definitions and a logical statement that must hold on those variables. The language of PAL is a limited predicate logic extension of Protégé that supports the definition of such ranges and statements.

The syntax of PAL is a variant of KIF. It supports KIF connectives, but not all KIF constants, predicates (i. e. the theory of arithmetic is much smaller), and statements, like (defrelation) and (deffunction).

PAL provides a set of special-purpose frames to hold the constraints that are added to a Protégé knowledge base, respectively the **:PAL-CONSTRAINT** class. The PAL constraint is an instance of the **:PAL-CONSTRAINT** class. The class has the following slots:

- **:PAL-name**, which holds a label of the constraint;
- **:PAL-documentation**, which holds a natural language description of the constraint;
- **:PAL-range**, which holds the definition of local and global variables that appear in the statement;
- **:PAL-statement**, which holds the sentence of the constraint.

The main part of the PAL constraint is the *PAL-statement*, which can be mapped to the information processing rule. The PAL-statement structure corresponds to the state or condition-state axiom. It has a clearly defined condition and a state. All constraints written by PAL define the state in which the domain should be. However, no information is provided about what should be done to implement a desirable state. The user triggers PAL constraints manually, when it is necessary. For more details about PAL see [46].

The EZPal Tab plug-in [62, 63] is used to facilitate acquisition of PAL constraints without having to understand the language itself. Using a library of templates based on reusable patterns of previously encoded axioms, the interface allows users to compose constraints using a "fill-in-the-blanks" approach.

5.3 Object Constraint Language (OCL)

We use OCL to support our statement that ontology axioms could be transformed into information processing rules, since UML is the most popular language for modelling business and information systems. However, there is no suitable interface to facilitate the definition of OCL constraints. User should be familiar with OCL. We attempt to propose a transformation which simplifies writing OCL constraints.

A UML diagram, such as a class diagram, typically is not refined enough to provide all the relevant aspects of a specification. There is a need to describe additional

constraints about the objects in the model. Therefore, OCL has been developed to fill this gap. OCL is a specification language [13]. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition).

According to [13], the following components of OCL constraints are defined:

```
Context TypeName [statement_type] {constrain name}:
[OCL statement]
```

Context introduces the context for OCL constraint. The context can be a particular class, attribute or method of a UML class diagram.

An OCL statement defines an OCL constraint. It is composed of a class, an attribute or a method, which is associated by a mathematical operator with a class, an attribute, a method or a value. An example of a statement is *self.numberOfEmployees > 50*, where *numberOfEmployees* is an attribute and 50 is a possible value of this attribute. *Note* that each OCL constraint is written in the context of an instance of a specific type. In an OCL expression, the reserved word ***self*** is used to refer to the contextual instance. For instance, if the context is the Company class, then self refers to an instance of the Company class.

`[Statement type]` is a possible type of statements in OCL constraints. It defines what kind of statement is used in an OCL constraint. Statement types can be stereotypes (like invariant (**`inv`**), precondition (**`pre`**), and postcondition (**`post`**)), which define stereotypes in an OCL constraint, an initial value (**`init`**), which is used to represent the initial value in an OCL constraint, and derived value (**`derive`**), which is used to represent the derivation rule.

5.4 Mapping PAL with OCL

According to the results presented in the previous section, Table 3 presents mapping of PAL to OCL.

Table 3

Mapping of PAL to OCL

Name of an element	OCL	PAL	Are they mapped?
Name of a constraint (for example, enoughEmployees)	**Yes** Defined after the statement type.	**Yes** Defined after the keyword %3APAL-NAME in quotes.	**Fully mapped**
Description of a constraint	**Yes, but not necessary** Defined in quotes.	**Yes, but not necessary** Defined after the keyword %3APAL-DOCUMENTATION in quotes.	**Fully mapped.** However, it is not the main part of a constraint.

Name of an element	OCL	PAL	Are they mapped?
Type of a constraint	Invariant (**`inv`**) – associated with a Classifier	**Yes**	**Fully mapped**
	Precondition (**`pre`**) – associated with an Operation or other behavioural feature	**No**	**No.** PAL has no operations.
	Postcondition (**`post`**) – associated with an Operation or other behavioural feature	**No**	**No.** PAL has no operations.
	Initial value (**`init`**) – indicate the initial value of an attribute or association end	**Yes**	**Mapped.** An initial value of an attribute can be defined
	Derived value (**`derive`**) – indicate the derived value of an attribute or association end	**Yes**	**Mapped.** A derived value of an attribute can be defined
Class, to which a constraint is attached (for example, Organization)	**Yes** Defined after the context.	**Yes.** Defined after the keyword %3APAL-RANGE in quotes.	**Fully mapped**
A statement of a constraint	**Yes** Defined after the statement type or a name, if a name is specified for the constraint. For example, self. numberOfEmployees > 50	**Yes** Defined after the keyword %3APAL-STATEMENT in quotes.	**Fully mapped.** In both constraints classes, attributes and mathematical operators are used.
Condition of a constraint statement	**Yes** Defined after the keyword **if**	**Yes** Defined after the symbol =>	**Fully mapped**
State part of a constraint statement	**Yes** Defined after the keyword **then**	**Yes** Any statement defined after the condition	**Fully mapped**

As can be seen from Table 3, PAL constraints can be transformed into OCL invariants, initial or derived values. Since an ontology and its elements, like classes, has no methods, preconditions and postconditions cannot be presented in an ontology.

An example of the mapping of a PAL statement to an OCL constraint follows.

- This part of a PAL-statement defines that a value of a slot `start_date` of the class `Employee` should be less than a value of a slot `end_date` of the same class.

```
(< ( 'start_date' ?Employee) ('end_date' ?Employee)))
```

- The presented OCL constraint corresponds to the PAL-statement. `?Employee` is transformed into the context of an OCL constraint. All slots of the PAL statement are transformed into the corresponding attributes in the OCL constraint. For example, "`end_date`" is transformed into "`self.end_date`".

```
context Employee inv:
self.end_date > self.start_date
```

For more detailed explanations, we present the mapping of Protégé ontology to UML class diagram in Table 4. It is used as a basis to define the mapping of PAL constraints to OCL constraints.

Table 4

Mapping of Protégé ontology elements to UML class diagram elements

Elements of a Protégé ontology	Elements of a UML class diagram
Protégé ontology	UML class diagram
"Thing"	Class
Class	Class
Slot	Attribute
Documentation	Comment
Value Types:	Data Types:
any	not defined
boolean	boolean
class	relationship with appropriate class
float	float
instance	relationship with appropriate class
integer	int
string	char
symbol	enumeration
Required	Multiplicity: 1
Minimum	Multiplicity:
Maximum	Multiplicity:
Default Values	Default Value
is-a relation (directed-binary-relation)	Generalization
PAL constraint	OCL constraint

5.5 An Example of Transforming PAL Constraints into OCL Constraints

The prototype of the *Axiom2OCL* plug-in is created to implement the proposed method of transforming PAL constraints into OCL constraints and to support the statement of authors that ontology axioms could be used for the development of information processing rules. The plug-in is developed according to the proposed mapping of PAL to OCL (Table 3).

The plug-in can be attached to MagicDraw UML 15.5 or Protégé 3.0 (or other version). Fig. 4 presents the Axiom2OCL plug-in attached to Protégé 3.4.

In this prototype the user should denote the input file, in which PAL constraints are stored, and may denote the output file, where OCL constraints will be stored. If the user does not denote the output file, the plug-in creates a default output file. After the denoting the input and output files, all PAL constrains from the input file will be automatically transformed into OCL constraints.

The plug-in is created in the Java development environment.

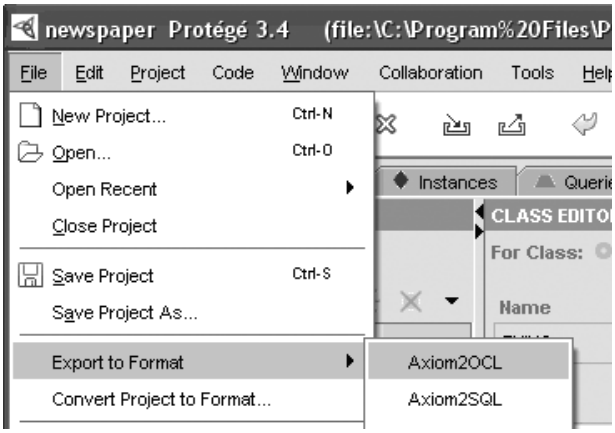


Fig. 4. The *Axiom2OCL* plug-in attached to Protégé 3.4

An example of transforming a PAL constraint, restricting that *the Employee end date should be after the start date*, into the corresponding OCL constraint, follows.

- A PAL constraint

```
(%3APAL-NAME "editor-employees-salary-constraint")
(%3APAL-RANGE "(defrange ?editor :FRAME Editor)\n(defrange
?employee :FRAME Employee responsible_for)")
(%3APAL-STATEMENT "(forall ?editor (forall ?employee\n (=>
(and \n (responsible_for ?editor ?employee)\n (own-slot-not-
null salary ?editor) \n (own-slot-not-null salary ?employee))
\n (> (salary ?editor) (salary ?employee)))))"))
```

- A corresponding OCL constraint

```
context Editor inv editor-employees-salary-constraint:
IF (self.responsible_for->notEmpty() AND self.salary ->
notEmpty() AND self.employee.salary -> notEmpty())
THEN (self.salary>self.responsible_for.salary) endif
```

The corresponding OCL constraint is attached to the part of a newspaper class diagram (Fig. 5), which is generated from the newspaper ontology using UMLBackend plug-in [47].

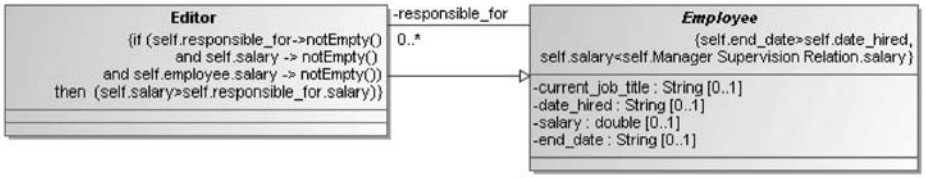


Fig. 5.Part of a newspaper class diagram

At this moment the prototype is not suitable for transforming all PAL constraints into the corresponding OCL constraints. Therefore, in the future it should be refined and adapted for the transformation of more difficult constraints.

The proposed transformation of PAL constraints into OCL constraints is applied in the High Technology Development Program Project "Business Rules Solutions for Information Systems Development (VeTIS)"[16] by extending MagicDraw tool to generate OCL constraints from PAL constraints.

6 Conclusions

The analysis of the related works shows that application domain rules are presented in ontology by axioms. However, the majority of authors analyse the use of ontology for the development of a conceptual data model, neglecting or not emphasising ontology axioms as a possible source for business rules development. The comparative analysis of ontology axioms and rules at the level of information systems let us to argue that ontology axioms can be used for modelling rules and consecutive implementation of such rules at the level of software systems.

Syntactic expressions of ontology and a conceptual data model were analysed and it was concluded that consolidation and derivation axioms, expressed in a particular language, can be mapped to dynamic rules, epistemological and definitional axioms – to the structure of a conceptual data model.

Thus, the method for transforming ontology axioms into OCL constraints, which can be defined as a part of a UML class diagram and which are most suitable for representing rules at intermediate level, and next to implementation level should be developed. Such a method is proposed in the paper.

The application of the method for transforming the Protégé ontology axioms (expressed as PAL constraints) to OCL constraints and their implementation in the Axiom2OCL plug-in shows that the method can be used for automatic generation of OCL constraints from ontology axioms.

The next step of the research is extending the developed plug-in.

References

1. N. Guarino. Formal Ontology and Information Systems. In: *Proc. of FOIS'98*. Amsterdam: IOS Press, 1998, pp. 3–15.

[16] http://www.verslotaisykles.lt/VeTIS/

2. M. Jarrar, J. Demey, R. Meersman. On Using Conceptual Data Modeling for Ontology Engineering. In: S. Spaccapietra et al. (eds.), *Journal on Data Semantics*. LNCS, Vol. 2800. Berlin/Heidelberg: Springer, 2003, pp. 185–207.
3. Y. Wand, V. C. Storey, R. Weber. An ontological analysis of the relationship construct in conceptual modeling. *ACM Transactions on Database Systems (TODS)*, Vol. 24(4), 1999, pp. 494–528.
4. T. R. Gruber. Toward Principles for the Design of Ontologies for Knowledge Sharing. *International Journal of Human and Computer Studies*, Vol. 43(4–5), 1995, pp. 907–928.
5. J. Trinkunas, O. Vasilecas. Ontology Transformation: from Requirements to a Conceptual Model. *Acta Universitatis Latviensis [Latvijas Universitates Raksti], Computer Science and Information Technologies*, Vol. 751. University of Latvia, 2009, pp. 54–68.
6. E. Bozsak et al. KAON – Towards a Large Scale Semantic Web. In: K. Bauknecht et al. (eds.), *Proc. of the Third International Conference on E-Commerce and Web Technologies (EC-Web 2002)*. LNCS, Vol. 2455. London: Springer-Verlag, 2002, pp. 304–313.
7. M. A. Goncalves, L. T. Watson, E. A. Fox. Towards a Digital Library Theory: A Formal Digital Library Ontology. *International Journal on Digital Libraries*, Vol. 8(2), 2008, pp. 91–114.
8. OMG: OntologyDefinition Metamodel, 2005. Available: http://www.omg.org/docs/ad/05-08-01.pdf. Accessed September, 2008.
9. T. Morgan. Business Rules and Information Systems: Aligning IT with Business Goals. Boston: Addison-Wesley, 2002.
10. B. von Halle. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. New York: John Wiley & Sons, 2002.
11. R. G. Ross. *Principles of the Business Rule Approach*. Addison Wesley, 2003.
12. OMG: Unified Modeling Language Specification. Version 1.4.2, ISO/IEC 19501:2005(E) (2005) Available: ftp://ftp.omg. org/pub/docs/formal/05-04-01.pdf. Accessed September, 2008.
13. OMG: UML 2.0 OCL Specification, 2003. Available: http://www.omg.org/docs/ptc/03-10-14.pdf. Accessed September, 2008.
14. B. Demuth, H. Hussmann, S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In: M. Gogolla, C. Kobryn (eds.), *Proc. of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML 2001)*. LNCS, Vol. 2185. London: Springer-Verlag, 2001, pp. 104–117.
15. R. G. Ross. *The Business Rule Book. Classifying, Defining and Modeling Rules*. Houston: Business Rules Solutions Inc., 1997.
16. L. Boyd. CDM RuleFrame – the Business Rule Implementation Framework That Saves You Work. In: *Proc. of ODTUG 2001*. Available: http://www.dulcian.com/odtug_conference.htm. Accessed November, 2006.
17. I. Horrocks, P. F. Patel-Schneider, H. Boley et al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C document, 2004. Available: http://www.w3.org/Submission/SWRL/. Accessed September, 2009.
18. V. Zacharias. Technical Report: Development and Verification of Rule Based Systems – a Survey of Developers. Technical Report, 2008. Available: http://vzach.de/papers/2008_SurveyTechReport.pdf. Accessed May, 2009.
19. C. Golbreich, A. Imai. Combining SWRL rules and OWL ontologies with Protégé OWL plugin, Jess and Racer. In: *Proc. of the 7th International Protégé Conference*, 2004. Available: http://galimed.med.univ-rennes1.fr/lim/doc_92.pdf. Accessed May, 2009.
20. H. Herbst et al. The Specification of Business Rules: A Comparison of Selected Methodologies. In: A. A. Verrijn-Stuart, T. W. Olle (eds.), *Methods and Associated Tools for the Information System Life Cycle*. New York: Elsevier, 1994, pp. 29–46.
21. A. Caplinskas, A. Lupeikiene, O. Vasilecas. Shared Conceptualisation of Business Systems, Information Systems and Supporting Software. In: H.-M. Haav, A. Kalja (eds.), *Databases and Information Systems* II. The 5th International Baltic Conference «BalticDB&IS'2002», Selected Papers. Dordrecht/Boston/ London: Kluwer Academic Publishers, 2002, pp. 109–120.
22. D. C. Hay. *Requirement Analysis. From Business Views to Architecture*. New Jersey: Prentice Hall PTR, 2003.
23. OMG: Semantics of Business Vocabulary and Business Rules (SBVR). Version 1.0, 2008. Available: http://www.omg.org/docs/formal/08-01-02.pdf. Accessed December, 2008.
24. T. Halpin. Object-Role Modeling: an Overview. 1998. Available: http://www.orm.net/pdf/ORMwhitePaper.pdf. Accessed November, 2009.

25. M. Badawy, K. Richta. Deriving Triggers from UML/OCL Specification. In: M. Kirikova (ed.), *Information Systems Development: Advances in Methodologies, Components and Management.* New York: Kluwer Academic/Plenum Publishers, 2002, pp. 305–316.
26. G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2000.
27. A. Armonas, L. Nemuraite. Using Attributes and Merging Algorithms for Transforming OCL Expressions to Code. *Information Technology and Control*, Vol. 38(4). Kaunas: Technologija, 2009, pp. 283–293.
28. M. Bajec, M. Krisper. Managing business rules in enterprises. *Electrotechnical Review*, Vol. 68(4), Ljubljana, 2001, pp. 236–241.
29. M. Bajec, M. Krisper. Issues and challenges in business rule-based information systems development. In: D. Bartmann, F. Rajola, J. Kallinkos (eds.), *Proc. of the 13th European Conference on Information Systems (ECIS 2005).* Regensburg: Institute for Management of Information Systems, 2005, pp. 1–12.
30. I. Valatkaite, O. Vasilecas. On Business Rules Approach to the Information Systems Development. In: H. Linger et al. (eds.), *Proc. of the 12th International Conference on Information Systems Development (ISD'2003)*. New York: Springer, 2004, pp. 199–208.
31. O. Vasilecas, D. Kalibatiene, G. Guizzardi. Towards a Formal Method for Transforming Ontology Axioms to Application Domain Rules. *Information Technology and Control*, Vol. 38(4). Kaunas: Technologija, 2009, pp. 271–282.
32. R. A. Falbo, C. S. Menezes, A. R. C. Rocha. A Systematic Approach for Building Ontologies. In: H. Coelho (ed.), *Proc. of the 6th Ibero-American Conference on AI (IBERAMIA'98).* LNAI, Vol. 1484. Berlin Heidelberg: Springer, 1998, pp. 349–360.
33. E. Mendelson. *Introduction to mathematical logic*. Belmont: Wadsworth & Brooks, 1987.
34. G. Guizzardi, R. A. Falbo, J. G. Pereira Filho. Using Objects and Patterns to Implement Domain Ontologies. *Journal of the Brazilian Computer Society*, Special Issue on Software Engineering, Vol. 8(1), 2002. Available: http://www.scielo.br/scielo.php?pid=S0104-65002002000100005&script=sci_arttext&tlng=en. Accessed September, 2008.
35. R. A. Falbo, G. Guizzardi, K. C. Duarte. An Ontological Approach to Domain Engineering. In: *Proc. of International Conference on Software Engineering and Knowledge Engineering (SEKE'02).* New York: ACM, 2002, pp. 351–358.
36. S. Staab, A. Maedche. Ontology Engineering beyond the Modeling of Concepts and Relations. In: N. Guarino et al. (eds.), *Proc. of the ECAI'2000 Workshop on Application of Ontologies and Problem-Solving Methods.* IOS Press, 2000, pp. 15–21.
37. D. McGuinness, P. Patel-Schneider. Usability issues in knowledge representation systems. In: J. Mostow, C. Rich (eds.), *Proc. of AAAI-98*. Madison, Wisconsin: American Association for Artificial Intelligence, 1998, pp. 608–614.
38. M. R. Genesereth. Knowledge Interchange Format (KIF). 2006. Available: http://logic.stanford.edu/kif/kif.html. Accessed October, 2006.
39. N. F. Noy, R. W. Fergerson, M. A. Musen. The knowledge model of Protégé-2000: combining interoperability and flexibility. In: R. Dieng, O. Corby (eds.), *Proc. of the 12th International Conference on Knowledge Engineering and Knowledge Management (EKAWÕ00)*. LNAI, Vol. 1937. Berlin: Springer, 2000, pp. 17–32.
40. SUMO: Suggested Upper Merged Ontology (SUMO). 2008. Available: http://www.ontologyportal.org/. Accessed December, 2008.
41. WordNet: Cognitive Science Laboratory. Princeton University, 2006. Available: http://wordnet.princeton.edu/. Accessed December, 2008.
42. R. Culmone, G. Rossi, E. Merelli. An Ontology Similarity Algorithm for BioAgent. In: S. Ercolani, M. A. Zamboni (eds.), *NETTAB Workshop on Agents and Bioinformatics*. Bologna, 2002. Available: http://www.bioagent.net/WWWPublications/Download/NETTAB02P1.pdf. Accessed March, 2007.
43. S. Lin et al. Integrating a Heterogeneous Distributed Data Environment with a Database Specific Ontology. In: E.H.M. Sha (ed.), *Proc. of the International Conference on Parallel and Distributed Systems, 2001 (ICPADS'01).* Washington: IEEE Computer Society Press, 2001, pp. 430–435.
44. C. Bizer. DBpedia. 2008. Available: http://dbpedia.org/About. Accessed December, 2008.
45. C. S. J. Hou, N. F. Noy, M. A. Musen. A Template-Based Approach toward Acquisition of Logical Sentences. In: M. A. Musen, B. Neumann, R. Studer (eds.), *Proc. of the Conference on Intelligent Information Processing (IIP-2002)*. Montreal: Kluwer, 2002, pp. 77–89.
46. W. Grosso, M. Crubezy. The Protégé Axiom Language and Toolset («PAL»). Stanford Medical Informatics, Stanford University, 2008. Available: http://protegewiki.stanford.edu/index.php/Protege_

Axiom_Language_%28PAL%29_Tabs. Accessed December, 2008.
47. H. Knublauch. UMLBackend. Stanford Medical Informatics, Stanford University, 2007. Available: http://protege.cim3.net/cgi-bin/wiki.pl?UMLBackend. Accessed December, 2008.
48. X. Su, L. Ilebrekke. A Comparative Study of Ontology Languages and Tools. In: A. B. Pidduck et al. (eds.), *Proc. of the 14th International Conference CaiSE*. LNCS, Vol. 2348. London: Springer-Verlag, 2002, pp. 761–765.
49. O. Corcho, M. Fernandez-Lopez, A. Gomez-Perez. Methodologies, tools and languages for building ontologies. Where is their meeting point? *Data & Knowledge Engineering*, Vol. 46(1), 2003, pp. 41–64.
50. L. Casely-Hayford. A comparative analysis of methodologies, tools and languages used for building ontologies. CCLRC Daresbury Laboratories, 2005. Available: http://epubs.cclrc.ac.uk/bitstream/894/OntologyReport.pdf. Accessed June, 2008.
51. J. C. Arpiirez, O. Corcho, M. Fernandez-Lopez, A. Gomez-Perez. WebODE: a scalable ontological engineering workbench. In: Y. Gil, M. Musen, J. Shavlik (eds), *First International Conference on Knowledge Capture*. New York: ACM, 2001, pp. 6–13.
52. A. Gómez-Pérez, M. Fernández-López, O. Corcho. WebODE Ontology Engineering Platform. WebODE Development Group, 2003. Available: http://webode.dia.fi.upm.es/WebODEWeb/index.html. Accessed June, 2008.
53. S. Bechhofer. OilEd Ontology Editor. University of Manchester, 2000. Available: http://xml.coverpages.org/oilEdANn20001204.html. Accessed June, 2008.
54. S. Bechhofer, I. Horrocks, C. Goble, R. Stevens. OilEd: a reasonable ontology editor for the Semantic Web. In: F. Baader, G. Brewka, T. Eiter (eds.), *Proc. of the Joint German/Austrian Conference on Artificial Intelligence (KIÕ01)*. LNAI, Vol. 2174. London: Springer-Verlag, 2001, pp. 396–408.
55. A. Farquhar, R. Fikes, J. Rice. The Ontolingua Server: A Tool for Collaborative Ontology Construction. *International Journal of Human-Computer Studies*, Vol. 46(6), 1997, pp. 707–727.
56. Ontolingua: Ontolingua – Software Description. Stanford University, 2005. Available: http://www.ksl.stanford.edu/software/ontolingua/. Accessed June, 2008.
57. D. L. McGuinness et al. The Chimaera Ontology Environment. In: *Proc. of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*. AAAI Press/The MIT Press, 2000, pp. 1123–1124.
58. B. Swartout et al. Toward Distributed Use of Large-Scale Ontologies. In: B. Gaines, M. Musen (eds.), *Proc. of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'96)*, 1997. Available: http://ksi.cpsc.ucalgary.ca/KAW/KAW96/swartout/Banff_96_final_2.html. Accessed April, 2009.
59. Y. Sure et al. OntoEdit: collaborative ontology engineering for the semantic web. In: *First International Semantic Web Conference (ISWCO02)*. LNCS, Vol. 2342. Berlin: Springer, 2002, pp. 221–235.
60. J. Domingue. Tadzebao and Webonto: Discussing, Browsing and Editing Ontologies on the Web. In: *Proc. of the 11th Knowledge Acquisition Workshop (KAW98)*, 1998. Available: http://ksi.cpsc.ucalgary.ca/KAW/KAW98/domingue/. Accessed December, 2008.
61. J. Domingue. WebOnto. 2008. Available: http://kmi.open.ac.uk/projects/webonto/. Accessed June, 2008.
62. C. S. J. Hou, N. F. Noy, M. A. Musen. EZPAL: Environment for composing constraint axioms by instantiating templates. *International Journal of Human and Computer Studies*, Vol. 62(5), 2005, pp. 578–596.
63. C. S. J. Hou. EZPAL. Stanford Medical Informatics, Stanford University, 2008. Available: http://protegewiki.stanford.edu/index.php/EZPal. Accessed December, 2008.
64. E. Sirin, J. Tao. Towards Integrity Constraints in OWL. In: R. Hoekstra, P. F. Patel-Schneider (eds.), *Proc. of OWL: Experiences and Directions 2009 (OWLED 2009)*, 2009. Available: http://www.webont.org/owled/2009/papers/owled2009_submission_35.pdf. Accessed March, 2010.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 33–57 P.

A Model-Driven Path from Requirements to Code

Audris Kalnins, Elina Kalnina, Edgars Celms, Agris Sostaks
University of Latvia, IMCS, Raina bulv. 29, Riga, LV-1459, Latvia
Audris.Kalnins@lumii.lv, Elina.Kalnina@lumii.lv, Edgars.Celms@lumii.lv, Agris.Sostaks@lumii.lv

Although there is a lot of support for model-driven development, few approaches offer support for a complete model-driven path from requirements to code. The approach proposed in this paper offers such a path fully supported by model transformations. The starting point is semiformal requirements containing behaviour description in a controlled natural language. A chain of models is proposed, including analysis, detailed design, and platform-specific models. A particular architecture style is chosen by means of selecting a set of appropriate design patterns for these models. We show how the required transformations can be informally defined and then implemented in the model transformation language MOLA. Thus, a prototype of the system is obtained which can then be extended in a model-driven way.

Keywords: model-driven development, transformations, requirements, UML.

1 Introduction

The main goal of this paper is to demonstrate how transformations could be used to support the full path from requirements to code in a model-driven development. Requirements are specified in the requirement specification language RSL [1, 2], which has been developed as part of the ReDSeeDS project [3]. A significant part of RSL is the specification of requirements for system behaviour in a controlled natural language. In this paper we demonstrate how such requirements can be used as the basis for transformations to code via Analysis, Detailed Design, and Platform-Specific Models. Models are generated according to a particular architecture style, including selection of appropriate design patterns for these models.

The Model Driven Architecture (MDA) approach [4] has evolved significantly since its launch in 2001. Now it has become just one of the versions of model-driven development (MDD) [5]. From the original chain of three models – CIM, PIM, and PSM – only the last two are used frequently, and typical MDA transformations support only the generation of PSM from PIM. The CIM model has got significantly less attention. In this paper, we assume that the proper contents of CIM are requirements.

The ReDSeeDS approach [3] used in this paper covers a complete chain of models for model-driven development – from requirements to code. Each transition in this chain is to a great degree assisted by formal model transformations. Although a specific chain of models is described here, the approach could be applied to any similar setting of models.

The first in the chain is the Requirements Model built in a special semiformal requirement language RSL (described in Section 4). The required behaviour specification

in this controlled natural language is sufficiently precise; therefore, this specification can be processed by model transformations in order to generate initial versions of the next models.

The next models are built using appropriate subsets (and profiles) of UML 2. The first corresponds more to the Analysis Model in the standard OOAD [6] approach. Therefore, we call this model the Analysis Model. In our approach, the main content obtained in this step is an analysis-level class model (the Domain Model). The Analysis Model is described in more detail in Section 5.

The most important model in the proposed model chain is the PIM, which is very close to the corresponding model in the MDA approach. This model is built according to the selected design patterns and contains the description of structure and detailed behaviour of the would-be system in a platform-independent way. Transformations which generate the initial version of this model use both Requirements and Analysis as inputs. Only this way the most sophisticated analysis of requirements can be performed. In the whole chain of transformations, this step contributes most to the rich system functionality inferred directly from requirements. The contents of PIM are described in Section 6.

It should be noted that for our models we use a pre-selected consistent set of design patterns and other design rules called an architecture style in our approach (this concept is described in Sub-section 3.2). Transformations are adjusted to this style to get maximum results in extracting the required behaviour from RSL. The best results are obtained if requirements are specified in RSL in an appropriate way – the RSL profile associated with the architecture style is used (see Sub-section 4.2).

The next model is the Platform Specific Model (PSM) in a fairly standard MDA style (Section 7). It is built by transformations from PIM by adding platform-relevant details. The paper demonstrates the combination of Java and Spring/Hibernate frameworks as the target platform, but any similar platform can be used as well. Finally, PSM is transformed to code (annotated Java EE in this case). The main value of the approach is that a large fraction of a non-trivial prototype of the system can be obtained from requirements without manual extension of intermediate models. Certainly, a true model-driven development should follow, where in each step the required details of the real system are filled in manually.

All model-to-model transformations in our approach are implemented in the model transformation language MOLA [7], which appears to be very appropriate for the given kind of tasks. If selection of patterns and the architecture style are changed, the transformations should be rebuilt too. The emphasis on transformation readability in MOLA would significantly facilitate this task. Another issue to be solved by transformations is the inevitable modifications of models and the necessity to reapply the transformations and merge the results. Transformation development is discussed in Section 8.

2 Related Work

The MDA Guide [4] states clearly that CIM means requirements for the system, although no formalism is proposed for this model. There also is an alternative view [8]

that CIM is just a business model of the whole business environment of the system; in this case, no transformations to PIM are possible. Because of that opinion, there are few approaches similar to ours.

Requirements in a controlled natural language, in particular behaviour scenarios, are used as a starting point in [9, 10, 11, 12, 13]. The approach closest to ours is described in [9], which proposes the Natural MDA language for description of behaviour in use cases. This language uses a large set of keywords; therefore, it is much closer to programming languages than RSL, and the transformation-based approach is only partial. The approach described in [10] is based on the Language Extended Lexicon and does not use the behaviour description thoroughly. The approaches proposed in [11, 12, 13] require an initial semi-manual transformation of natural language requirements into a more formal notation, which then can be processed by model transformations. An interesting approach of this kind is proposed in [12], where the initial requirements in a natural language are manually converted into a list of semiformal functional features, which then can be transformed formally using the topological functioning model. In [11] a manual XML-based initial marking of requirements is used before a grammar-based processing can be applied. In [13] a manual conversion into behaviour trees is used. Thus, the direct processing of requirements in a controlled natural language by transformations is the innovation offered by our approach.

There is so much work on transforming PIM to PSM that we do not comment on this subject since it is not the main topic of our paper.

3 General Principles of the Proposed Approach

3.1 Models

In this section, we present a short rationale behind our selection of the specific model chain.

Requirements are specified in the requirement specification language RSL [1, 2] which is developed as part of the ReDSeeDS project [3] and is the basis for the approach. We are interested mainly in requirements for the system behaviour specified by use case scenarios and draft domain concepts (which are called notions in RSL).

Starting from requirements, a chain of models for a model-driven development of the software system is proposed. To a great degree, this chain is inspired by the classical MDA approach. However, the specific structure and construction principles of models in our approach are determined by the chosen architecture style which most importantly includes the set of selected design patterns. A more precise description of the concept of the architecture style is given in Sub-section 3.2. All the models are built in UML using an appropriate profile.

Initially the Analysis Model is extracted by transformations from requirements. This model has no direct counterpart in the classical MDA chain. In the Analysis Model the most important part is a class diagram describing the main concepts of the software system to be created. Stereotypes are used to distinguish different types of concepts according to the Analysis Profile.

The next model in this chain is the PIM model. In this model, the implementation structure is represented according to the behaviour extracted from use case scenarios. This model is platform-independent and could be used as a basis for development of a code on any enterprise platform (Enterprise Java, .NET, etc). This is the model where the selected design patterns and sophisticated analysis of requirements permit to generate a non-trivial part of solution behaviour.

The final model in the chain is PSM. From this model code fragments for the selected platform can be generated. Currently the chosen platform is Java in the Spring/ Hibernate framework. In this model stereotypes corresponding to Spring-specific annotations are used. Data from this model are transformed to Java code with Spring/ Hibernate annotations.

It should be noted that in ReDSeeDS project an alternative model naming is used – PIM is also called the Architecture Model and PSM the Detailed Design Model.

3.2 Design Patterns and the Architecture Style

Nowadays, as a rule, large enterprise systems are developed using a set of design patterns. There are two types of design patterns: platform-independent and platform-specific. The traditional GoF design patterns [14] represent the former type. The modern Java EE environments (based on the POJO [15] idea and declarative ORM) also share a large set of common enterprise patterns (and so do the latest .NET environments based on POCO [16]). On the other hand, low level patterns such as an adequate usage of Spring framework annotations are still platform-specific.

Usage of design patterns is vital to efficient application of MDD and transformations. However, patterns alone are not sufficient for deciding how the generated models look like. Therefore, we use the concept of *architecture style*, which includes the structure of the system and model, a related set of design patterns (with indications where they should be used), the applied general design principles, and finally, the rules by which model elements are obtained from models preceding in the development chain. This last feature is formalized by a model transformation set associated with the architecture style. The most important content of an architecture style is the selected set of design patterns, tied up to the chosen model structure. Namely patterns are the style element which helps most in specifying efficient transformation rules. In addition, for transformations supporting the given architecture style to produce maximum results, the requirements must be specified in an appropriate style too; therefore, the concept of RSL profile (associated with the given architecture style) is introduced.

In this paper, we propose an architecture style named *Keyword-Based Style*. The main goal of this style is to extract as much as possible behaviour from the requirements. The in-depth analysis of requirements is based on keywords to be found in RSL sentences which the style is named after. The RSL profile associated with the Keyword-Based Style is described in Sub-section 4.2.

We start the description of the Keyword-Based Style with the model and system structure and some general design rules. We have chosen four-layer architecture because it is the most popular and accepted information system architecture style today. We use the following layers: Data Access or Repository layer, Service or Business layer, Application Logic, and User Interface. We also have domain objects as data containers (available to any layer, former DTOs [17]). Another general principle is that our approach

is based on a declarative object-relational mapping (ORM). The particular ORM in our approach will be Hibernate [18]. Whenever possible, we use an interface-based design style for all layers, meaning there is an interface (where the operations are specified) and its implementation class. The selected design patterns for the style will be described in the next sub-section.

It should be noted that there is already an architecture style defined in the ReDSeeDS project from the very beginning named the *Basic Style*. The goal of the Basic Style is to prove the feasibility of the approach in which model-driven development starting from requirements is combined with requirement-based reuse of software. The initial version of ReDSeeDS tool support is also based on this style. However, the possibilities to extract behaviour from requirements in the Basic Style are significantly weaker than in the proposed Keyword-Based Style.

In no case the selected architecture style should be considered the sole possible solution; other styles are also possible. To a great degree, the choice of the most appropriate architecture style depends on the domain of the system to be created. The proposed Keyword-Based Style could be an adequate solution for simple web-based information systems. The selection of architecture style could be formalized on the basis of non-functional requirements for the system; however, this topic is completely out of the scope of this paper. Furthermore, it should be reminded that creation of a new architecture style also requires creation of an appropriate transformation set.

3.3 Selected Design Patterns for the Keyword-Based Style

In this sub-section, we will describe the design patterns chosen for the Keyword-Based Architecture Style. The patterns are grouped according to models and system layers chosen for the style. The patterns used at the PIM level are as much platform independent as possible. Since we have chosen Java + Spring + Hibernate framework as the target platform, the design patterns popular in the Spring community are used at the platform-specific level. This choice has also slightly influenced our PIM level, when we had to choose one of several equivalent options.

We use the DAO design pattern [19] at the Data Access layer. Data access objects are introduced as the main actors for explicit ORM-related actions. Therefore, each DAO has the basic CRUD and typical Find operations. A data access object is created for each persistent domain concept. DAO classes are assumed to have the standard transaction support for their operations.

For business logic, the main design pattern used is Manager (see [20] for its version in the .NET world). It means that for each domain concept participating in business logic, a class (and interface) is created, which encapsulates all business level operations related to this concept.

The application logic and user interface layers are governed by the MVC pattern, which is used in almost every four-layer architecture. In addition, for application logic, the façade pattern [14, 17] is used. For each Use Case in requirements, we create one application logic interface and an implementing class. This class implements all operations invoked by MVC controllers within this use case.

The UI part is kept as simple as possible. It contains only calls to the application layer. This research does not include the specific issues of building user interfaces

from requirements, which is a separate topic in the ReDSeeDS project (currently in development [21]).

We also use the domain object design pattern. It means we use domain objects as data containers, in other words, as standard "POJO" (not mandatory Java) objects. Persistent domain objects are treated as the basis for ORM definition; therefore, platform-independent ORM features such as identifying attributes and persistent relations are included.

The design in general relies on the Dependency Injection Pattern (which will appear later as platform-specific dependency annotations) for referencing other classes; therefore, the Factory Pattern is not used explicitly.

Platform-specific design patterns are used in PSM and in the code. It is domain objects that have the most of platform-specific features. The POJO pattern is used, adapted to the Spring style. We use the declarative ORM definition (Spring + Hibernate) based on annotations. Annotations are coded as appropriate stereotypes in PSM. The transactionality of relevant classes is also defined by annotations. For reference initialization, the dependency injection pattern is used.

For UI layer, the MVC design pattern is used in a standard ("Spring-Basic") way.

4 The Requirements Model

The development of a software system in ReDSeeDS starts with definition of requirements for it in the Requirements Model.

4.1 Requirements Specification Language in ReDSeeDS

The Requirements Specification Language (RSL) [1, 2] is a semiformal language for specifying requirements for a software system. We briefly sketch here those elements of RSL which can be directly transformed into the system design.

RSL employs use cases for defining precise requirements for the system behavior. Each use case is detailed by one or more scenarios, which in turn consist of special controlled natural language sentences. The main type of sentences is the SVO(O) sentence [2], which consists of a subject, verb, and direct object (optionally, also an indirect object). These sentences express the actions to be performed in the scenario. In addition to SVO(O), there can also be conditions, rejoin sentences ("gotos" to a point in the same or another scenario) and invoke sentences (invoke another use case). Alternatively, the set of scenarios for a use case can be visualized in a natural way as a profile of a UML activity diagram. SVO(O) sentences serve as the nodes of the diagram, and conditions and rejoins as control flows (in addition to the natural "next sentence" control flow).

Another part of RSL is domain definition which consists of actors (system users), system elements, and notions. Notions correspond to elements (classes) of the conceptual model of the future system. It is also possible to define notion generalization and simple associations between notions.

The precise syntax of RSL is defined by means of a metamodel [1]. The behavior and domain parts in a valid RSL requirements model must be strictly related. The subject of an SVO(O) sentence must be an actor or system element. An object (direct or indirect)

must be a notion. The informal meaning of each noun and verb must be defined in a vocabulary (currently, WordNet [22]).

For the first version of RSL, a ReDSeeDS tool support [23] has been built including an RSL editor. This tool version only supported the Basic Architecture Style, with an appropriate set of model transformations for generation of PIM and PSM from requirements included. The Enterprise Architect (EA) tool [24] in ReDSeeDS is used for UML support. At present the final version of ReDSeeDS tool support has been built [25]. It includes the basic support for an extended RSL version, the main new feature being the introduction of keywords in order to support the Keyword-Based Style as well. Another aspect is the extension of the domain part – adding attributes to notions and extending association features so that a fully-fledged class model can be obtained. This paper is based on the extended RSL version – see an RSL example in Fig. 1.

4.2 The RSL Profile

Transformations described in this paper can be applied to any valid set of requirements in RSL for a system. However, in order to ensure that these transformations generate a really substantial fragment of the software system to be built, some more constraints on the requirements should be put. Thus, a concept of the RSL profile is introduced. The profile defines the set of keywords with predefined semantics to be used in scenario sentences (verbs, nouns, and prepositions) and some rules on how these keywords should be used. In addition, there are constraints on the order of these sentences (or nodes in the activity form). All these rules are "soft" rules in the sense that requirements do not become invalid if they violate some of these rules; simply, the transformations can do less. At the same time, profiles are defined so that they never make requirements less readable to domain area specialists (however, more skills may be required by requirement engineers to create them). A profile is always associated with an architecture style so that the corresponding transformation set can produce the largest possible part of the PIM and PSM models from requirements.

In fact, the default Basic Architecture Style together with the default RSL profile and the corresponding transformation set has already been used in ReDSeeDS [23]. This profile has no keywords, only some constraints on sentences. The usage has confirmed the feasibility of the used technologies; however, the part of a system generated by transformations is small.

In this paper, we propose a profile for the Keyword-Based Style. In this profile, the verb keywords for SVO(O) sentences are *show*, *select*, *build*, *add*, and *remove*. The noun keywords are *form* and *list* – when used as parts of complex notion names (and, consequently, objects in SVO(O) as well). Conditions (which otherwise are arbitrary sentences in RSL) can contain the verb keyword *click* and noun keywords *button* and *link*. The adjective (modifier in RSL terms) *empty* is also treated as a keyword.

Now we briefly describe the meaning of keywords and some context rules in scenarios. The keyword *show* means that the system must display a form defined by the direct object of this sentence. This object, in turn, must correspond to a notion whose complex name ends with the noun keyword *form*. For example, the SVO(O) sentence "System *shows* reservable facility list *form*" specifies that the form "reservable facility list *form*" must be displayed at this point.

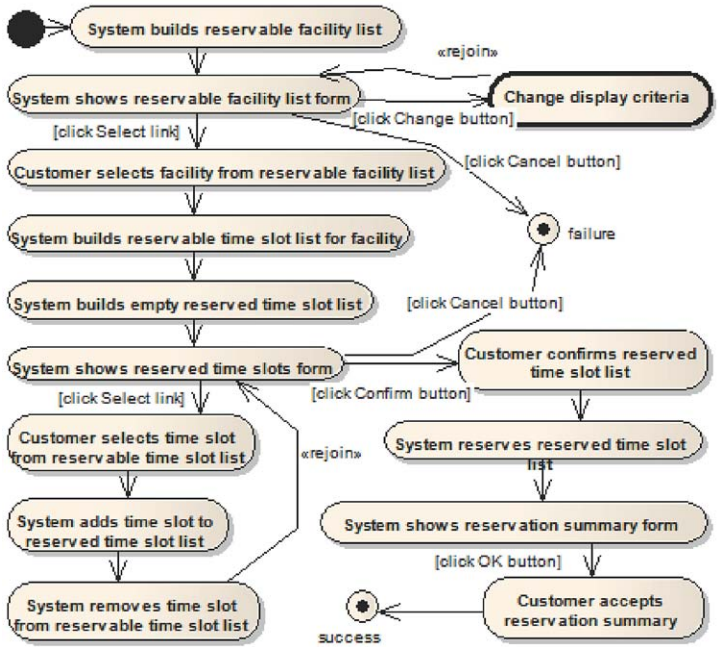


Fig. 1. Requirements – scenarios of the use case in a graphical form

Similarly, the sentence "System *builds* reservable time slot *list* for facility" uses the verb *build*, which means a data creation. The direct object "reservable time slot *list*" denotes a list, since the last noun in it is *list*.

The sentence "Customer *selects* facility from reservable facility *list*" means that the user has performed element selection from the data table in the form. The indirect object (after the preposition "from") specifies the data table contents ("reservable facility *list*", that is, a *list* notion), the selected element is an instance of the notion "facility".

The condition "*click* Select *link*" means that the user clicks on an active element (link) in a form table with selectable rows. Normally this condition should be on the control flow, which goes from the *shows* sentence/node (see the example above) to the *selects* sentence (the previous example). The condition "*click* Confirm *button*" means that the form button has been clicked.

The meaning of the remaining keywords is self-explanatory. The example in Fig. 1 completely complies with the rules described above.

The described profile for the Keyword-Based Style is supported in the current version of ReDSeeDS tools.

4.3 An Example of Requirements

The proposed ideas are illustrated on a fragment of an example of the Fitness Club system. One use case *Reservations* is taken – how a club customer can book regular access to the selected fitness facility of the club. The scenarios for this use case (in the form of one activity diagram) are defined in RSL (see Fig. 1). For this to be a correct requirements model, the relevant notions must also be defined (facility, reservable

facility list, etc). Fig. 2 presents two scenarios of this use case in textual form as they were entered using the RSL editor.

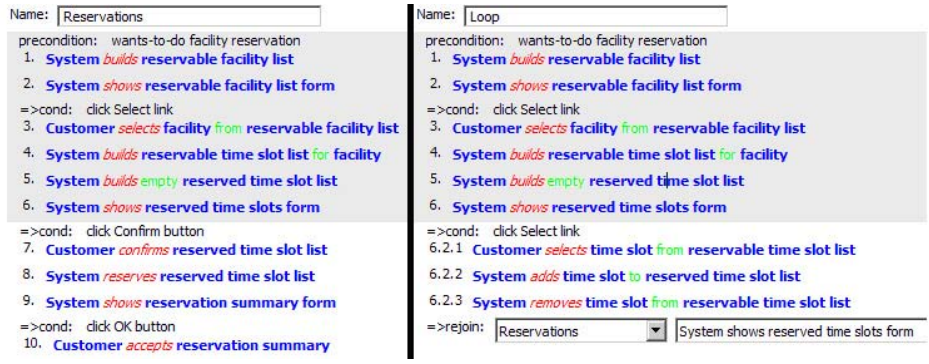


Fig. 2. Requirements – two scenarios in a textual form

The colour marking helps to distinguish more clearly the parts of SVO(O) sentences – subjects, verbs, and objects. Prepositions starting an indirect object are marked green. The whole continuous group of words marked blue is an object with a complex name (there must be an equally named notion in the domain part of the requirements). Note that in the textual syntax, each scenario is one continuous path in the diagram.

5 The Analysis Model

5.1 The Structure of the Analysis Model

The main part of the Analysis Model in the Keyword-Based Style is the Domain Model – a conceptual class model for the system to be built. The Domain Model is generated by appropriate transformations from the domain (notion) part of Requirements. It contains classes corresponding to all notions in Requirements. Class attributes and associations are also extracted from the notions part of Requirements (if they have been defined there). A special Analysis Profile is defined in ReDSeeDS which contains stereotypes to be applied to the Domain Model. Classes generated from persistent notions would have the <<entity>> stereotype (there also are some heuristic rules how to find persistent notions when they have not been properly marked in requirements). Other classes with the stereotype <<form>> would correspond to forms – notions with the suffix form in their names. In a similar way, collection classes (for example, ReservableFacilityList) will have the <<list>> stereotype. In the design stage, these classes will be converted into generic list classes. Control elements in forms (such as buttons and links) are also represented by stereotyped classes in the Domain Model, with stereotypes <<button>>, <<gridLink>>, <<link>>, and some others. Additional associations having a special meaning for the design model (e.g. aggregations linking a form to a list to be visualised as a data grid in this form) can also be generated. These associations are also given special stereotypes (<<owned>>, <<formElement>>, and some others). See more on the principles how the Domain Model is generated from Requirements by transformations

in Sub-section 5.2. Fig. 3 presents part of the generated Domain Model in the Fitness club example. It shows that the proposed approach can transfer a significant part of the intended semantics of requirements into the stereotyped Domain Model (this, in turn, will guarantee a rich behaviour to be generated into the PIM model).

The full strength of the transformations is revealed only if requirements are built in RSL according to the appropriate RSL profile (see 4.2). If requirements in RSL cannot provide sufficient information for building this Domain Model, it is highly recommended to extend this model manually in the Analysis Step. Only in this case the next steps will provide the desired results.

The structuring of the Domain Model is based on notion packaging (provided in RSL).

5.2 Transformation of Requirements to Analysis

The main task of this transformation is to create the Domain Model from the notion part of Requirements, taking into account some elements of scenarios as well. The basic transformation is very straightforward since notions, their attributes, and relationships in RSL actually are in one-to-one correspondence to the class model. The stereotypes <<*list*>> and <<*form*>> are added if the respective keywords are present in the notion names. An additional analysis is done for list classes. If an entity name is contained within the list notion name (such as "facility" within "reservable facility list"), the entity class is assumed to be the element of that list (a <<*listItems*>> association is generated).

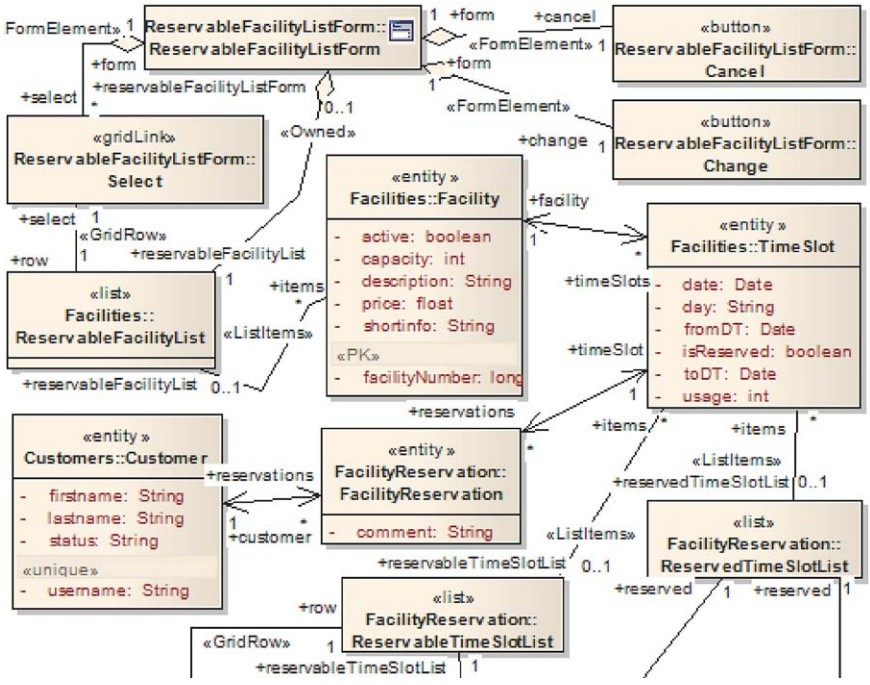


Fig. 3. Fragment of the generated Domain Model

Classes for control elements can be generated from scenarios. We are looking for a *click*-condition (*click … link* or *click … button*) which follows a *show*-sentence (… *shows … form*). If such (new) situation is found, a class is generated with the name equal to the name in the *click*-condition and the stereotype <<*gridLink*>> or <<*button*>>, respectively. The association (with the stereotype <<*formElement*>>) linking the control element to its form is also generated.

More form-related associations can be generated from scenarios. *Select*-sentences (such as ... *selects* facility from reservable facility *list*) let us conclude that the relevant form (that in the preceding *show*-sentence) permits to select elements exactly from this kind of list. Hence, this list (here, ReservableFacilityList) is visualized in the form (the <<*owned*>> association can be built), and each gridLink element in the form corresponds to a row in the list (the <<*gridRow*>> association is built).

Using these relatively simple principles, the domain model in the example in Fig. 3 can be generated from notions and the scenario in Fig. 1. The implementation of these transformations in the MOLA language is also quite straightforward.

6 The Platform-Independent Model

This model is the most important to our approach since all platform-independent functionality is generated in this model. This is done by revisiting the use case scenarios and analyzing them repeatedly, taking into account the (possibly manually extended) Domain Model from Analysis. In combination with the keyword-based sentence analysis, a significant part of application and especially business logic can be generated. This model is created according to the platform-independent design patterns described in Sub-section 3.3.

6.1 The Structure of the Platform-Independent Model

The main result of the PIM step is the design class model: packages and classes (and interfaces) with all attributes and operations. The operations will have all parameters defined. All the other data such as persistence info for ORM-related classes are coded by platform-independent stereotypes, which constitute the PIM profile.

The other essential results of this analysis are stored as sequence diagrams, also covering a significant part of business logic method bodies. All method invocations with appropriate parameters which can be generated are coded this way. Whenever possible, invocation logic up to the DAO level is documented. These sequence diagrams are kept in the *behaviour* package and are grouped in the same way as Use Cases in the Requirements Model. Some small practical extensions of sequence diagram syntax are used, for example, *FOREACH* iterator in *loop* fragments.

The design class model is split into the following packages: *applicationlogic*, *businesslogic*, *dataaccess*, *domainobjects*. The first three are further subdivided into *Interfaces* and *Implementation* parts containing interfaces and implementing classes, respectively. Each interface name has the prefix “I” added to the corresponding class name.

For application logic, the façade design pattern is used. For each use case, a class corresponding to this use case is generated (with the the suffix “Service” added to the

name). Further structuring of the *applicationlogic* package is done according to use case packages.

The content of *businesslogic* is generated according to the Manager Pattern. Here classes correspond to persistent classes (entities) whose usage in business logic can be inferred from sentences with keywords and the domain model. Classes/interfaces have the suffix "Service" added to the entity name.

For *dataaccess*, an updated version of the DAO pattern is used, and practically applicable methods are generated for DAO classes. Each class corresponds to a persistent domain object; the class name is generated from the object name with suffix "DAO". Classes are grouped in the same way as domain objects. For each class, CRUD and some typical find operations are generated. Bodies of these operations are similar in all classes, only types vary. Therefore, we propose to implement them once in a template class which contains parameterized types. All the other classes will inherit them from this template class (with parameters set to the relevant values in each case). We remind that this specialization of the classical DAO pattern is platform-independent since it can be directly implemented in most of typical platforms.

For the *domainobjects* package, the domain object design pattern is used. This package represents a platform-independent ORM (Object Relational Mapping) model for all entities, with platform-independent annotations. Associations (relations) are also included in a way typical of an ORM definition. A database schema for a specific platform can also be easily generated from this model (in the next PSM step). Names of domain objects are taken from the corresponding domain concepts. For each persistent class, a unique identifier attribute is defined as well.

6.2 Transformation of Requirements and Analysis to PIM

Transformations for building the platform-independent model are more complicated than for building the Domain Model in Analysis. They use the behavior part of the Requirements Model as input, as well as the updated Domain Model.

The transformation of domain objects is very straightforward. Domain classes are transformed to PIM domain objects, keeping all attributes. For each persistent class without primary key, an artificial primary key is created.

For each persistent domain class, a DAO class and its interface is created in the *dataaccess* package. They specialize the template-based implementation of CRUD and filter operations.

In the Business Logic layer, classes and interfaces have a similar structure as in DAO, with the exception that classes not having business level methods are excluded. The generation of business methods is done in the general context of behavior generation by analyzing scenarios in requirements.

In the Application Logic layer, for each use case, a class and interface is generated. For this interface/class, one "main" method is generated (which means invoking this use case from another one). Its name corresponds to the Use Case name. Other methods for this class are generated for UI-related sentences in the scenario. UI-related sentences are detected by analyzing the subject of the sentence. If the subject of the sentence is an actor, then it is Actor-system sentence (or UI-related sentence).

Now we will describe behavior generation. Behavior is grouped in the same way as Use Cases. For one Use Case, one or more sequence diagrams are generated by

processing its scenario. The behavior of a Use Case begins with invocation of the "main" method of the application logic class corresponding to the Use Case. In order to build an application logic method body, we look for consecutive scenario sentences with the subject System and recipient system (in other words, any verb other than "System *shows* …). All these sentences correspond to calls to the Business Logic layer. At first the verb used in this sentence is analyzed. If the verb is a keyword, the sentence is analyzed according to rules used for this keyword. If the verb used is not a keyword, the structure of the sentence is analyzed and object keywords are analyzed. Default behavior generation principles corresponding to the sentence structure are applied. The immediate recipient of this call depends on the sentence structure. If the indirect object (e.g., …*for* facility) is present, the call is directed to the manager of the corresponding entity (here, *FacilityService*). Another typical case is when an indirect object is absent and a direct object corresponds to a notion/class with the stereotype <<*list*>>. Then the invocation is created to the manager class corresponding to the entity class which is the list element. There also are some other "patterns" of sentences which correspond to business logic calls (or simple actions directly in the application layer). The grouping of the generated business logic calls is done in a simple way – all these calls up to the next UI call (corresponding to the next "System *shows* …" sentence) are included in the body of the current application logic method body (see Fig. 4). The "System *shows* … *form*" sentence generates a call to the user interface layer (to the controller of the relevant form), which completes the current body. The next sentence (which in fact follows the "*click* …" condition) corresponds to the invocation of another application logic method. Then building of the body of this method starts.

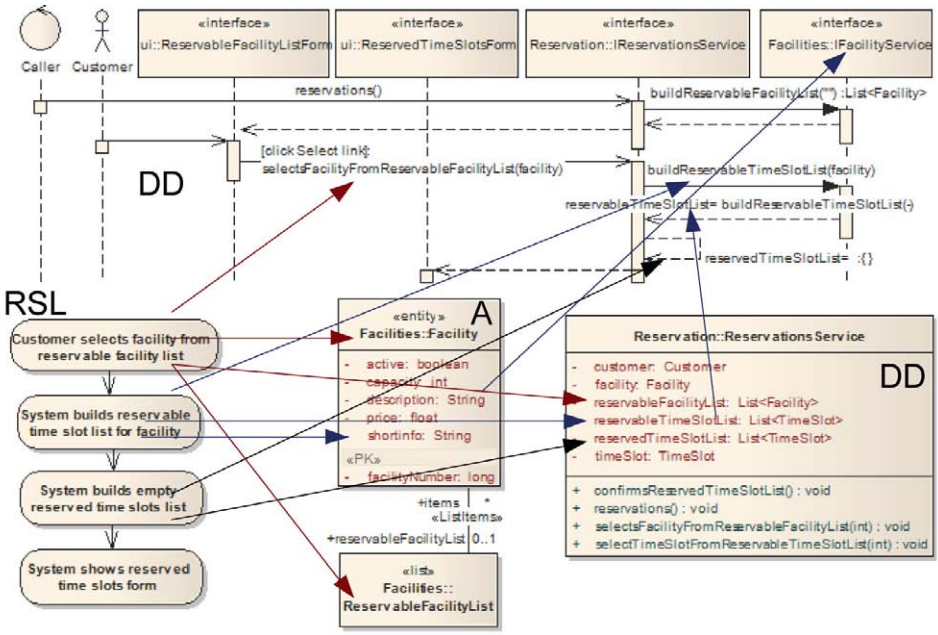


Fig. 4. An example of informal mapping describing transformations to Detailed Design

Fig. 4 illustrates in detail a typical application of the transformation rules described above by an informal "model mapping diagram", with arrows going from source model instances (bottom) to the corresponding target model instances (top). The first sentence in the scenario fragment ("Customer selects facility from reservable facility list") follows the "click Select link" condition; therefore, it implies the method invocation *selectFacili tyFromReservableFacilityList*() to the application logic class (*ReservationsService*). The next two sentences in the scenario correspond to the actions in the body of this application logic method. Fig. 4 shows detailed analysis of the first sentence. The sentence "System builds reservable time slot list for facility" implies the business logic method invocation *buildReservableTimeSlotList*(). According to the rules described above, there is an indirect object ("for facility"); therefore, the method must go to the corresponding manager class (to the class *FacilityService*). Because of *build*-semantics (*build* is a keyword) of the verb and *list*-semantics of the direct object, the return type of the method is *List<TimeSlot>*. The returned value must be stored in the attribute *reservableTimeSlotList* (of the same list type) of the invoking application class (*ReservationsService*). The next sentence corresponds to an action in the body (assignment to the attribute *reservedTimeSlotList*) because of the semantics of the keyword *empty*. Note that all lifelines correspond to interfaces because any invocation goes via the corresponding interface in our style (certainly, body behavior relates to the relevant class).

There are some more rules in the approach quite similar to those explained in the example. We do not examine the interaction with the UI layer in more detail.

7 The Platform-Specific Model and Code

This model is a specialisation of the platform-independent model to a specific platform. Java with Spring + Hibernate 3 was chosen, with declarative (annotation-based) style as much as possible.

7.1 The Platform-Specific Model

For this platform, the model is quite similar to the platform-independent model. The class structure in the PIM more or less corresponds to the required structure in PSM. The main task is to convert annotations to the specific style required by Spring and Hibernate. However, some new model elements should be added as well.

A new model is the database diagram generated from the domain objects. This is a typical database design diagram (with tables, columns, PK, FK, etc) in EA.

Domain objects themselves are "copied" with the same package structure. They are used to describe Hibernate-specific ORM functionality. All Hibernate- and Spring-specific annotations are added (coded as stereotypes) to domain classes, attributes, and operations. The relevant getters/setters and some predefined methods are added to classes. Traceability links between PIM and PSM elements are generated by transformations and used to maintain various annotations related to mappings between different parts of the model.

For each DAO class, the annotation <<*@Repository*>>is added. These classes also have annotations describing the transactional mode, by default "required" is used. The template-based mechanism is directly taken from PIM.

Application logic layer classes are included in the Business Logic layer. Classes in these layers are given the annotation <<*@Service*>> (to mark them as Spring beans). The annotation <<*@Autowired*>> is used to initialize references to other beans.

The structure of PSM corresponds directly to the potential Java class structure typically used in Spring (with packages *domain*, *repository*, *service*). These packages are further structured in accordance with the already defined model structuring.

In order to have a more or less complete design class structure and behaviour in sequence diagrams, some elements in the UI area also have to be specified. The basic source for that – forms, attached data, and actions (buttons and links) are available in the Analysis Model. Currently a rudimentary solution directly based on Spring MVC is proposed. In this solution, we can use JSP for data visualisation and controllers to manage user actions. We use one controller per form, with a method for each user action in the form. Typically a controller method directly calls the appropriate application logic method. Nevertheless, this should be treated only as a "stub" which can be replaced by a more appropriate UI feature definition. Such a prototype form structure definition could be incorporated in requirements since RSL language contains features for that purpose. Currently some experiments in this direction have been performed.

Sequence diagrams defining behaviour within method bodies are also refined according to Spring requirements. The most significant changes refer to the user interface part. At this level, a simple version of UI and application logic interaction can be precisely defined. In particular, a special "executable" solution (including DAO methods) could be provided for finding the object selected by the user via a data grid in a form. This way, the form behaviour sufficient for simple prototyping could be provided. We do not describe the UI aspects of PSM in more detail since tool support for them has not been fully implemented.

7.2 The Java Code

The provided PSM can be used for Java code generation. This generation is quite straightforward – at first all information must be transferred into a properly stereotyped class model using MOLA transformations (the body behaviour must also be transferred from sequence diagrams to code sections of operations in EA). Then properly modified EA Java code generation scripts can be used. The main issue of modification is to add scripts for processing all relevant annotations.

The structure of the Java code directly corresponds to the structure of PSM. Methods are generated according to the model. For some methods, predefined method bodies are generated. This is widely used for domain objects (almost all methods are generated). In particular, bodies of getters, setters, *hashCode*, *equals*, *toString* are generated. A template-based generator is used and the method body vary according to object properties for which the method is generated.

Predefined method bodies of the TemplateDAO class are also generated. Concrete DAO classes extending the TemplateDAO class with appropriate types are also generated. Appropriate Hibernate configuration file describing, for example, data base connection is also necessary. An initial version of this file can be generated. It should be noted that a data base script can also be generated from PSM.

Business logic- and application logic-related functionality is generated according to the class structure. The behaviour (described in sequence diagrams) is also generated.

For the UI part, currently only a placeholder is generated (due to reasons explained in Sub-section 8.1).

The generated Java project can be inserted into an Eclipse IDE project template containing references to the required Spring and Hibernate libraries. Thus, a ready-to-compile project is obtained. All this constitutes a significant part of a simple prototype – mainly the UI part has to be added manually. However, if the complete set of transformations described here was implemented, a "near to executable" prototype would be obtained.

Here are some examples of the generated Java code. The first example shows part of the code generated for the *Facility* entity.

```
@Entity
@Table(name="facility")
public class Facility {

private Boolean active;
private Boolean capacity;
private String description;
private String facilityNumber;
private String id;

@Override
public boolean equals(Object obj){
if (this == obj) return true;
if (!super.equals(obj)) return false;
if (getClass() != obj.getClass()) return false;
Facility other = (Facility) obj;
if (active == null) {
  if (other.active != null) return false;
} else if (!active.equals(other.active)) return false;
if (capacity == null) {
  if (other.capacity != null) return false;
} else if (!capacity.equals(other.capacity)) return false;
if (dscription == null) {
  if (other.description != null) return false;
} else if (!description.equals(other.description)) return false;
if (facilityNumber == null) {
  if (other.facilityNumber != null) return false;
} else if (!facilityNumber.equals(other.facilityNumber)) return
false;
 return true;
 }

@Column(name = "active", nullable = false)
public Boolean get_Active(){
return active;
}

...

public void set_Active(Boolean p){
active=p;
}

...

}
```

The next code fragment shows the code generated for Application logic methods. These are three methods for the Application logic class *ReservationsService*. To understand the context, one sequence diagram from the PSM model is shown in Fig. 5. There are three method invocations on the *ReservationsService* lifeline (*reservations*, *selectsFacilityFromReservableFacilityList*, and *selectsTimeSlotFromReservableTimeSlotList*). Methods invoked within the corresponding fragments of the lifeline (until the return) appear within the corresponding body.

A code fragment for *ReservationsService* class.

```
@Service("ReservationsService")
public class ReservationsService implements IReservationsService {

@Autowired
private IChangeDisplayCriteriaService iChangeDisplayCriteriaService_;
@Autowired
private IFacilityService iFacilityService_;
@Autowired
private IReservedTimeSlotListService iReservedTimeSlotListService_;
private List<Facility> reservableFacilityList;
private List<TimeSlot> reservableTimeSlotList;
private List<TimeSlot> reservedTimeSlotList;

...

public void reservations(){
reservableFacilityList=iFacilityService_.
buildsReservableFacilityList();
}

public void selectsFacilityFromReservableFacilityList(Facility
facility){
reservableTimeSlotList=iFacilityService_.buildsReservableTimeSlotLis
tFor(facility);
reservedTimeSlotList= new ArrayList<TimeSlot>();
}

...

public void selectsTimeSlotFromReservableTimeSlotList(TimeSlot
timeslot){
reservedTimeSlotList.add(timeslot);
reservableTimeSlotList.remove(timeslot);
}

}
```

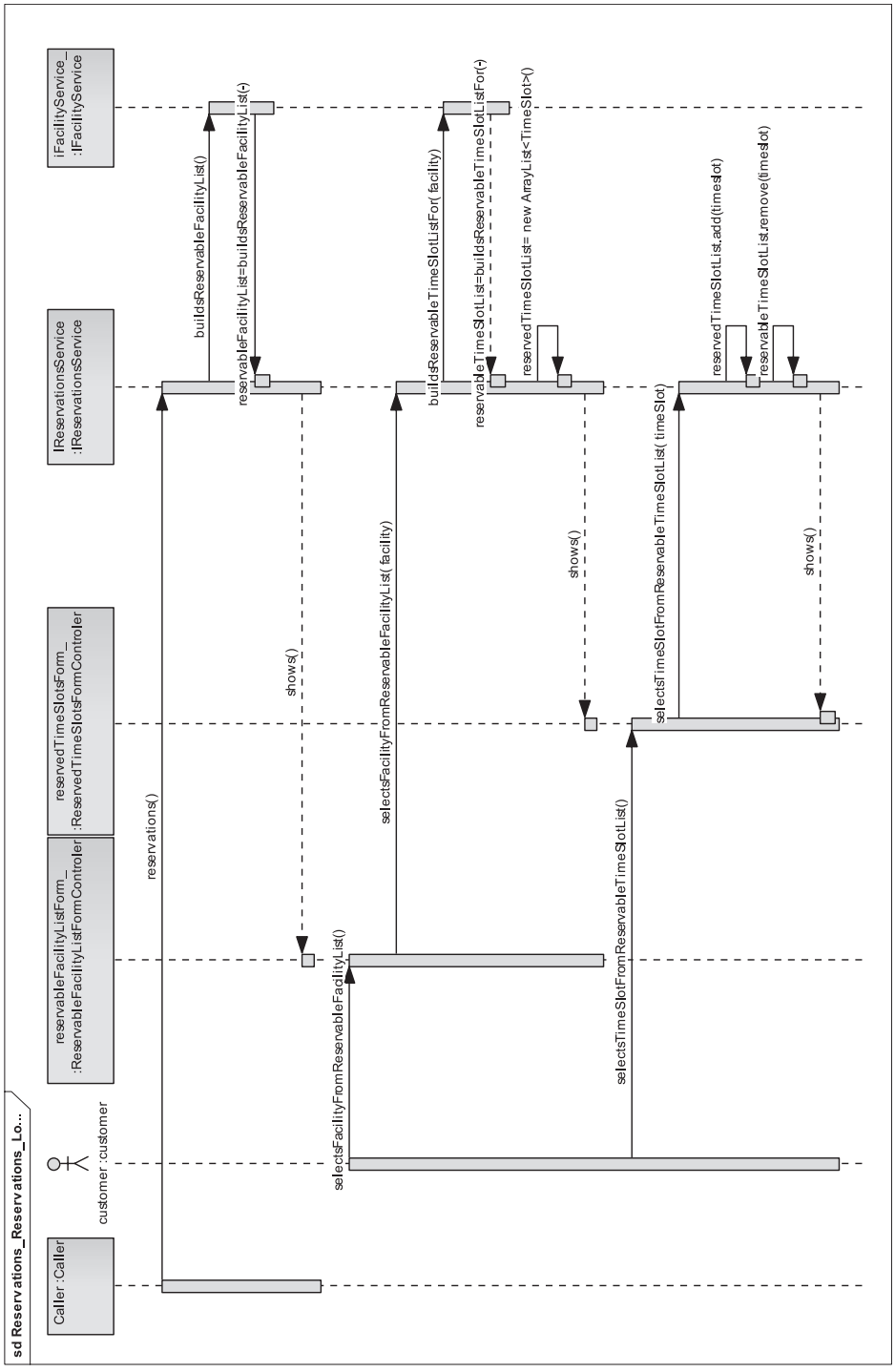


Fig. 5. An example of a sequence diagram for *ReservationsService* class

8 Implementation of Transformations

8.1 Model-to-Model Transformation Implementation

In this section, we briefly describe the implementation of transformation algorithms for building the chain of models in the Keyword-Based Style. The transformation language MOLA [7, 26] is used to define the transformations.

Although the current version of ReDSeeDS tools supports both the Basic Style and the Keyword-Based Style, not all new model transformation features described in previous sections are implemented in this version, mainly the features related to generation of UI functionality. For example, analysis of condition sentences in scenarios is not implemented since in the chosen RSL profile, conditions are mainly related to user interaction with forms. Similarly, the Analysis Model is created using only keyword-based analysis of notions, scenario-based analysis is not implemented in the current version. This again is related to the fact that scenario sentences can mainly contribute to finding relations between forms and their contained controls. The delay of transformation support for UI functionality is due to the fact that it would be natural to combine the generation of UI features from scenarios with direct specification of UI structure in RSL (as is usually done during requirements specification). Although this possibility is in the RSL language, as already said, currently there is minimum tool support for this.

Consequently, the UI part in generated models is implemented minimally; only some basic UI classes and interfaces are created. All the remaining details of UI such as form elements are not generated in the current version. Therefore, code generation for the UI part is not supported either although generation of some code skeletons is technically feasible.

One deviation from clean usage of UML in models is also visible in some of the examples. Assignments in sequence diagrams are emulated by message text and some tagged values because this feature is defined in UML in a very complicated way and supported in virtually no UML tools. This workaround has made some transformations more complicated.

The transformations are implemented using the MOLA tool [26]. Several different kinds of transformations are developed. Firstly, there are basic transformations supporting each software case development step: from RSL to Analysis, from Requirements and Analysis to PIM, from PIM to PSM, and from PSM to the PSM Code Model. There also are some technical transformations: export to EA, import from EA, keyword analysis, RSL scenario visualization by UML activity diagrams, and Simple Merge. Some transformation rules are re-used in several transformations.

The metamodel used for transformations is the same as for other ReDSeeDS tool components – it consists of an RSL metamodel merged with relevant parts of the standard UML metamodel and extended by special traceability elements. Transformations also build the relevant traceability links in every step.

Some non-trivial aspects of transformation implementation are described below.

Transformation for keyword analysis (which is the first to be applied in the chain) scans nouns, verbs, and modifiers used in scenario sentences, and fills in the keyword field of relevant RSL elements. This permits to specify the same keyword with several synonyms. It could be improved further by including Wordnet-based meaning analysis in this transformation.

The next transformation is from RSL to the Analysis Model. The logic of this transformation is relatively simple – it analyses the notion model in RSL and transforms it directly into a UML class diagram, adding stereotypes based on keywords set by the previous transformation.

The most important transformation is from the requirements and analysis model to PIM. This transformation has two logical parts. The first part is the creation of a static structure – package hierarchy, classes, and interfaces. The second part is the creation of behavior stored as UML sequence diagrams.

For creation of a static structure, a universal "package hierarchy copier" is used. This package hierarchy copier receives as input root of the source package hierarchy, the target package, and the copy mode. The package copier copies a hierarchy of packages and their elements (classes, interfaces, etc) in a way specific to the given mode. For example, it is possible to define that for some mode, a suffix should be added to the class name. It is also possible to define that for some mode class attributes should be ignored, etc. The universal package hierarchy copier is used in several contexts during creation of PIM and PSM models. In PIM Data Access objects and Business Logic objects are based on Analysis class diagram. In PIM Data Access class should be created for each persistent class in the Analysis Model. This is ensured using an appropriate copy mode. The same copy package hierarchy mechanism is even more widely used in creation of PSM since it is based on the PIM model with some modifications.

Another important part of PIM is the behavior description using UML sequence diagrams. In this case RSL scenarios are analyzed and sequence diagrams are created. For each scenario, one UML sequence diagram is created. The content of this sequence diagram depends on RSL sentences used in this scenario. Objects generated from a sentence depend on the kind of the sentence. There are three kinds of sentences: an "Actor-System" sentence defines interaction of an actor with the system. It can be recognized by the subject of the sentence – an Actor. The Subject of the two other kinds of sentences must be a system element. The next kind is a "System-Actor" sentence. Such sentence typically means that the system shows something to the user or asks for some input from the user. The third kind is "System-System" sentences. These sentences are used to describe internal actions of the system, typically some business logic. There are different subkinds of these sentences, depending on keywords used in the sentence.

The sequence diagram elements generated from a sentence depend on the kind and subkind of the sentence. At first the subkind of the sentence is determined; then elements of sequence diagrams are created. Since the UML sequence diagram metamodel is quite complicated, procedures for basic element creation are used. The procedure for one subkind of a sentence consists of calls to procedures for creating/finding basic sequence diagram elements. Fig. 6 demonstrates an example of procedure creating sequence diagram elements for "System-System" SVO sentence without keywords. At first the lifeline corresponding to the object is found or created. Then a message to this lifeline is created. Then operation corresponding to this message is found or created. Then this operation is associated with the message created. Then a return message is created. Each of these tasks is implemented as a MOLA procedure invoked by the given procedure. These procedures for sequence diagram element processing are used as building blocks. The content of one such MOLA procedure is shown in Fig. 7, which demonstrates the search of lifeline in a sequence diagram depending on the object used in the verb phrase.

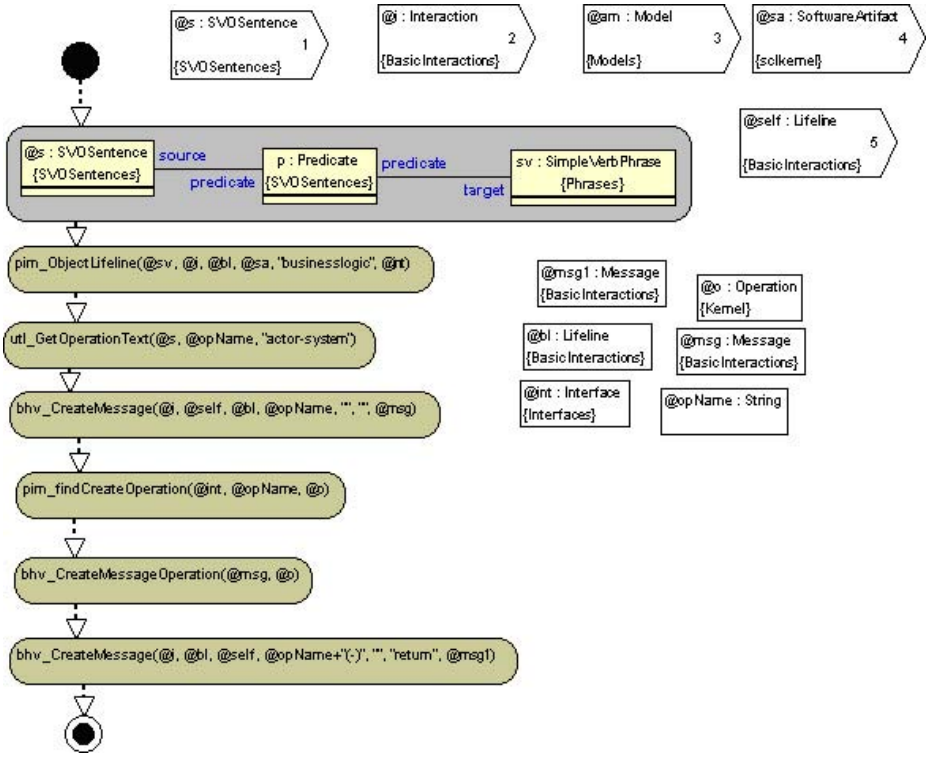


Fig. 6. Creation of a message for a "System-System" sentence without an indirect object

In the first rule, the notion corresponding to the noun used in a verb phrase is found (the long chain of associations necessary to locate this correspondence is implied by the RSL metamodel [1]). Then it is determined whether this notion or its parent should be used. Then the interface corresponding to this notion is found (it has been created during static structure generation). In this case, the Business Logic interface is found. Finally, lifeline for this interface is found or created. This procedure is very typical of transformation implementation in ReDSeeDS – it demonstrates the strength of MOLA patterns in finding complicated correspondences between model elements (such complicated correspondences are enforced by the structure of RSL and UML metamodels).

The next step in the chain is transition from PIM to PSM. For the creation of PSM, the package hierarchy copier described above is widely used. Only appropriate modes are defined.

The transformation from PSM to the initial code analyses sequence diagrams and creates the initial code. The code is attached to each relevant method. All messages from a lifeline starting from a method invocation on the lifeline to the return message (a message describing return to the caller of this message or a message to UI) are transformed to actions in the code for this method. For storing code corresponding to an operation, UML comments are used (the initial code is not a standard UML metamodel element). The transformation for code creation iterates through all messages in the sequence diagram. The search is performed in a recursive way (based on a stack). When

it detects a call of some operation, it means the following messages will constitute the body of this operation. If call to another operation follows this operation, the call to this other operation is added to the code body of this operation and this operation is added to the stack; and the newly created operation is set to be the current. If return from this operation to the previous operation is detected, the previous operation is popped out from the stack. If self messages are detected, an appropriate code is simply added to the message body. The stack is implemented using UML comments since it was not possible to extend the metamodel with temporary classes (due to requirements of other tool components).

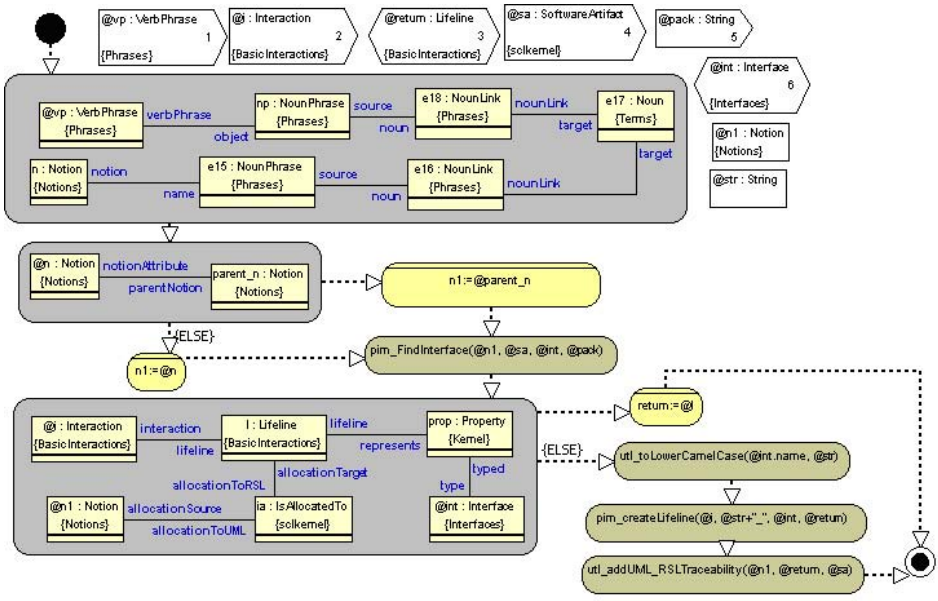


Fig. 7. The procedure of finding a lifeline in a sequence diagram depending on the object used in the verb phrase

Implementation of these transformation rules in the Keyword-Based Architecture style took approximately 3 person months. Implementation of these transformation rules consists of about 140 MOLA procedures (of size similar to those shown in Fig. 6 or 7). Thus, MOLA happened to be a very appropriate solution for implementation of transformations of such kind. The very low error rate during the development has to be singled out. Implementation of rules currently missing would be a small part of the existing code.

One more aspect of transformation implementation should be pointed out. All transformations in the chain must support repeated runs – the requirements ever change. What is even more important, for the same transformations to be applicable to manual model-driven development, all models in the chain should allow for manual modification. Therefore, support for various result merge actions must be included in the transformation set. In our approach, this support mainly relies on traceability links.

Currently one kind of the merge procedure – the so-called Simple Merge is implemented, but more sophisticated merge procedures could be implemented too.

8.2 Model-to-Code Transformation Implementation

Many MDD-based tools offer code generation from UML models. The Enterprise Architect (EA), the modelling tool used in the ReDSeeDS project, has the Code Template Framework (CTF) which also provides code generation features. The CTF consists of a number of code generation templates which generate a code for the most popular programming languages like Java, C++, etc. Each template transforms particular aspects of the UML to corresponding parts of the target language. Just like most of code generation tools in the MDD world, the EA does not provide full code generation, but code skeletons (classes, interfaces, field and operation declarations) can be obtained. Only packages, classes, and interfaces are used by these templates, other UML elements are ignored. These templates are called base templates. The latest versions of EA (not used in the project) provide some code generation features for behavioural UML diagrams as well (sequence, state).

Since the ReDSeeDS project uses the EA for UML support, there is a possibility to re-use all CTF capabilities of code generation. It is a significantly easier way to obtain a code than to generate a Java model as the first step and then convert this model to a proper code.

Base templates can be used directly for the default architecture style. These templates are applied to a detailed design model of this architecture style. The package hierarchy, declarations of all classes (DAO, DTO, etc), and methods are included in the generated code. Bodies of obtained methods should be filled in manually since the detailed design model in this style contains no behaviour.

For the Keyword-Based Architecture Style, significantly more code can be generated, including the behaviour aspects. Base templates do not generate the declarative annotations used in the Keyword-Based Architecture Style. We remind that these annotations are specified in the platform-specific model as appropriate stereotypes of classes, attributes, and associations. However, code generation templates are defined using the model-to-text language (the CTF language) in EA. Thus, it is possible to customize the way in which CTF generates a source code. The extension of the Java code generation template for Spring framework has been built. The generated code contains Spring annotations obtained from the stereotypes.

Although behavioural diagrams cannot be properly used for code generation in EA, they can be processed by model transformations before the code generation step. For example, a MOLA transformation converting a message and action sequence in a sequence diagram into part of the code of the appropriate method body has been implemented using an intermediate model. Then such an enriched intermediate model can be further processed by code generation templates in EA. Since such pre-processing is done, a great portion of the code (for example, method invocations from sequence diagrams) is being generated using EA. This way a meaningful executable prototype code could be obtained directly from requirements. If the models in the software platform-independent and platform-specific models have been extended manually, a true model-driven development can be carried out by this approach.

9 Conclusions

The paper shows the feasibility of a transformation-supported path from semiformal requirements to code in a model-driven way. The key aspects that have enabled this are selection of an appropriate architecture style (the general structure and an appropriate set of design patterns) for the system and an associated style for requirements – a profile for the requirement language. Then a corresponding set of transformations can be defined that can extract maximum facts from requirements and convert them into appropriate elements of models in the development chain. The most crucial of models in the chain is the Platform-Independent Model. To build this model, most sophisticated analysis of requirements has been done. The next model – PSM – is adapted to the selected platform – Java, Spring, and Hibernate. For models in the chain – Analysis, PIM, and PSM – appropriately defined UML profiles are used. The models obtained by this approach serve as the basis for further model-driven development, using the same transformations for support. All the transformations are implemented in the model transformation language MOLA.

We want to conclude with some thoughts on software design languages. Sequence diagrams used in our approach are the most natural UML facility for describing the required behavior. This notation is excellent for describing the way methods of another class are invoked. However, a more detailed data flow description soon becomes very awkward in this notation. An alternative could be the usage of UML activity notation with basic actions included, but then the class interaction is clearly less readable. Thus, the UML possibilities are slightly unclear. It may be that a special DSL for software design should be developed.

Acknowledgments. This work is partially funded by the EU Project "Requirements-Driven Software Development System (ReDSeeDS)" (contract No. IST-2006-33596 under 6FP). The authors would like to thank ReDSeeDS partners for valuable discussions. Special thanks to the ReDSeeDS partners from Warsaw University of Technology. The authors would also like to thank Oskars Vilitis for Spring-related consulting.

References

1. H. Kaindl, M. Smiałek, D. Svetinovic et al. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. Available: www.redseeds.eu (http://redseeds.iem.pw.edu.pl/index.php?option=com_remository&Itemid=7&func=fileinfo&id=58).
2. M. Smialek, J. Bojarski, W. Nowakowski et al. Complementary use case scenario representations based on domain vocabularies. *LNCS*, 4735, 2007, pp. 544–558.
3. Requirements-Driven Software Development System (ReDSeeDS) Project. EU 6th Framework IST Project (IST-33596). Available: http://www.redseeds.eu.
4. J. Miller, J. Mukerji et al. *MDA Guide Version 1.0.1*, omg/03-06-01. OMG, 2003.
5. T. Stahl, M. Voelter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, 2006.
6. C. Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 1998.
7. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. *Proceedings of MDAFA 2004*, LNCS ,Vol. 3599, Springer, 2005, pp. 62–76.
8. A. Queralt, E. Teniente. A platform-independent model for the electronic marketplace domain. *Springer SoSym*, Vol. 7, No. 2, May 2008, pp. 219–235.

9. L. Leal, P. Pires, M. Campos. Natural MDA: Controlled Natural Language for Action Specifications on Model Driven Development. *Proceedings of OTM 2006*, LNCS 4275, pp. 551–568.
10. M. C. Leonardi, M. V. Mauco. Integrating natural language-oriented requirements models into MDA. Workshop on Requirements Engineering, WER, 2004, pp. 65–76.
11. B. B. Bryant, R. R. Raje, M. Auguston et al. From Natural Language Requirements to Executable Models of Software Components. *Proceedings of the Monterey Workshop on Software Engineering*, 2003, pp. 51–58.
12. J. Osis, E. Asnina, A. Grave. Computation Independent Modeling within the MDA. ICSSTE07, pp. 22–34.
13. R. G. Dromey. Formalizing the Transition from Requirements to Design. In: Jifeng He and Zhiming Liu (Eds.) *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*. World Scientific Series on Component-Based Development, 2006, pp. 156–187.
14. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995.
15. R. Sriganesh, G. Brose, M. Silverman. *Mastering Enterprise JavaBeans 3.0*. Wiley Publishing, 2006.
16. J. Nilsson. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET.* Addison Wesley, 2006.
17. F. Marinscu. *EJB Design Patterns*. John Wiley, 2002.
18. C. Bauer, G. King. *Java Persistence with Hibernate*. Manning, 2007
19. C. Richardson. *POJOs in Action*. Manning, 2006.
20. V. P. Mehta. *Pro LINQ Object Relational Mapping with C# 2008.* Apress, 2008.
21. S. Kavaldjian, H. Kaindl, K. S. Mukasa, J. Falb. *Transformations between Specifications of Requirements and User Interfaces.* 4th Int. Workshop MDDAUI 2009, pp. 37–40.
22. Fellbaum, C. (ed.) *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
23. M. Rein, A. Ambroziewicz, J. Bojarski et al. Initial ReDSeeDS Prototype. Project Deliverable D5.4.1, ReDSeeDS Project, 2008. Available: www.redseeds.eu.
24. Sparx Systems, Enterprise Architect tool. Available: http://www.sparxsystems.com.au/.
25. M. Rein, A. Ambroziewicz, J. Bojarski et al. Final ReDSeeDS Prototype. Implementing the ReDSeeDS Engine prototype – 2nd iteration. Project Deliverable D5.4.3, ReDSeeDS Project, 2009. Available: www.redseeds.eu.
26. UL IMCS, MOLA pages. Available: http://mola.mii.lu.lv/.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 59–74 P.

Prototype for Traversing and Browsing Related Data in a Relation Database

Guntis Arnicans, Girts Karnitis
University of Latvia, Raiņa bulv. 19, Rīga, Latvia
{Guntis.Arnicans, Girts.Karnitis}@lu.lv

People who develop, test and maintain information systems often have to inspect the content of databases to make sure that data have been stored correctly or to find errors in the data. The most popular RDBMS and specialised database management tools usually offer single-table browsing. Sometimes SQL requests will be required before the necessary view of multiple tables can be ensured. This paper offers simple principles for inspection and traversing of databases, which may serve as a framework for the establishment of more effective tools for the visual inspection of database content. Browsing and traversing are based on the ER model of a database. Relationships among entities make possible definitions of various specialised views for each record from a table as well as displaying linked information from records in other tables. The relationships also define transitions for traversing the database so that the user can move from one data object to another. A prototype for data traversing and browsing based on these principles is developed.

Keywords: data browsing, database inspection, database traversing, transition graph.

1 Introduction

There are countless applications today which use a relation database management system (RDBMS). When information systems (IS) and databases are developed, that involves inspection of the relevant database. IS developers need to see real data that are stored in the relevant database – a process which is usually described as data browsing. We believe that the inspection of the contents of a database is a complicated process. That can be attributed to a lack of appropriate tools, and this both encumbers and delays the process of development. In this paper, we will review ways in which the data browsing process can be improved so as to make the work of developers easier and to reduce the possibility for errors.

We can divide all applications that work with database in two groups: *Business-specific applications* and *IS development-oriented applications* (Fig. 1). Main features of both groups are shown in Table 1.

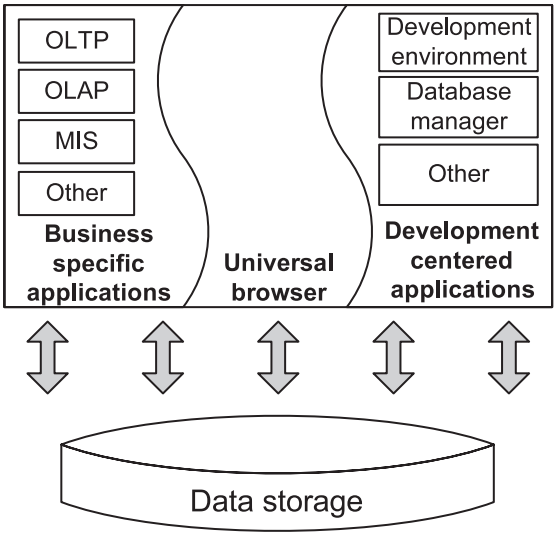


Fig. 1. Applications that work with database

Table 1

Features of the application groups

Feature	Business-specific application	Development-centered application
Target	Business requirements	Software engineering requirements
Data granularity	Low	High
Data view	Logical	Physical
Requires specific knowledge	Good knowledge on business, poor knowledge on IT	Poor knowledge on business, good knowledge on IT

There are situations when none of the applications mentioned above are suitable for use in practice. For instance, if we need to solve a problem with some IS and to ensure that complex object data is correctly stored in number of related tables.

Business specific information system represents data according to the business rules. It is hard to collect precise information what data is stored in which table in which format with these kinds of IS. Additionally, if we have a problem, there is a high possibility that IS is not working properly. It is impossible to ensure correct information retrieval from database tables if IS is working incorrectly.

Problems outlined above signify that we must inspect data by not using a business application. Most RDBMS have tools with good functionality for the management of databases in accordance with the ER model and with contemporary requirements. Unfortunately, RDBMS management tools also have certain shortcomings because they do not have all of the functionality that is necessary for data inspection.

RDBMS and the many tools that are used when working with a database will ensure data browsing at the entity level. The developer can review data from the table or view, as obtained after an SQL request. We believe the characteristics listed below are the most important that make data inspection more difficult than the user expects.

- Browsing of data from a table is a local process, one that is limited to only one table. Data values cannot be very expressive (e.g., numbers which are identifiers or keys to real world values stored in the other tables).
- Writing view can require effort and appropriate IT skills.
- View can be erroneous in the sense of not satisfying user requirements and creating a false impression of the true situation inside the database.
- View may not display those records which, at the logical level, are near to, but not fully in line with the selected data criteria (i.e., the low-quality data which we are seeking may not appear in the results).

Problems mentioned above indicate that we need supporting tools that ensure database inspection in a fast, easy and safe way during IS development and maintenance. In this paper we consider a simple tool which reduces the aforementioned problems in the inspection of data.

Our approach is based on the Entity-Relationship model that was introduced by Chen [1]. At first, tool obtains metainformation about database structure (tables and their attributes, relationships between tables) and allows for data browsing and traversing from one record to related records based on the data model. The user can select any table as source table and get records on the basis of criteria entered for this table. The user chooses a single record and the tool displays related records from other related tables. By selecting any relation, the related table becomes a new source table filled with related records.

2 The Problem of Database Inspection

When information systems which preserve their data in an RDBMS are developed and maintained, various specialists have to study their content – system analysts, software designers, coders, testers, database administrators, system maintainers, system users (in specific situations), etc.

Let us examine a few typical cases in which database inspection and debugging are necessary.

For example, the user of an information system reports that the system is returning incorrect results. System maintainers must find the cause. First of all, they must understand whether the data in the database is correct. This leads them to look for the data which is involved in the preparation of the result. This kind of data is usually found in various tables linked by relations.

System maintainers try to follow the algorithm for the extraction of results in a step-by-step way, writing individual SQL requests and testing the intermediate results at each step. If the intermediate results seem valid, then the next step can be taken. It might be necessary to make use of values from the fields of earlier intermediate results, and these have to be part of the next SQL request. The person who is doing the work must simultaneously record and store these values, either electronically (file, clipboard) or on paper. This is a process which takes a lot of time, and human errors are quite possible.

When an SQL request is based on multiple tables, the intermediate result can be erroneous. That can be the result of incorrect data in a table, or an incorrect SQL request.

For that reason, the system maintainer should use SQL requests which do not require the joining of multiple tables.

The work is similar for software developers who believe that new software is not returning the correct results.

When a system developer wishes to get a better sense of the design of relevant systems and particularly databases, the study of the data model must be supplemented by a survey of data in the actual database. Before relationships among data can be determined, the developer must write SQL requests, as described above.

System testers who use test cases will sometimes find that they need to take a look inside the database to make sure that the data have been stored correctly. If data are stored in multiple tables or are related to data in other tables, then the tester must write SQL requests, as described above.

All of these activities involve a user-accessible tool which usually makes it possible to review the structure of a database – tables and their fields, as well as primary and foreign keys. SQL requests for the database can be executed, and the results of such requests become visible. With such tools, the data browsing process is not always convenient or fast, and mistakes are possible.

The scenario for data examination (i.e., the series of SQL requests) and the data studied as part of that research will be different each time. That makes it more difficult to automate data examination processes on the basis of existing tools. At best we can save the series of SQL requests from the most typical incidents so that it can be reused with minor modifications.

If the software is being developed by a non-IT company, fundamentally important aspects of the testing duties are clearly handled by employees who have no specific IT-related knowledge [2]. The inability to obtain and analyse data from the database clearly reduces the quality of the testing process in such situations.

Let us now review an example that will be discussed in detail in this paper. There is a small university-based information system which is used to record whether students attend lectures and the grades that they receive on tests. The diagram of this entity-relationship system is presented in Fig. 2. The letters PK is used to show the primary key of the table and the letters FK is used to show the foreign key.

Let us look at a specific record in the table Grade, which contains the field values

```
<Examination_ID=5000001,
Student_ID=100002,
Grading_Teacher_ID=2001,
Grade=10>.
```

Let us call this the viewpoint for the database. The first three numbers are internal identifiers, and they offer us no semantic information. When it comes to the grade of 10, the student who received that grade, the examination to which the grade applies, the instructor who prepared and graded the examination, and the subject area in which the examination was taken – all those are of interest to us. In a graphical form it could look like Fig. 3.

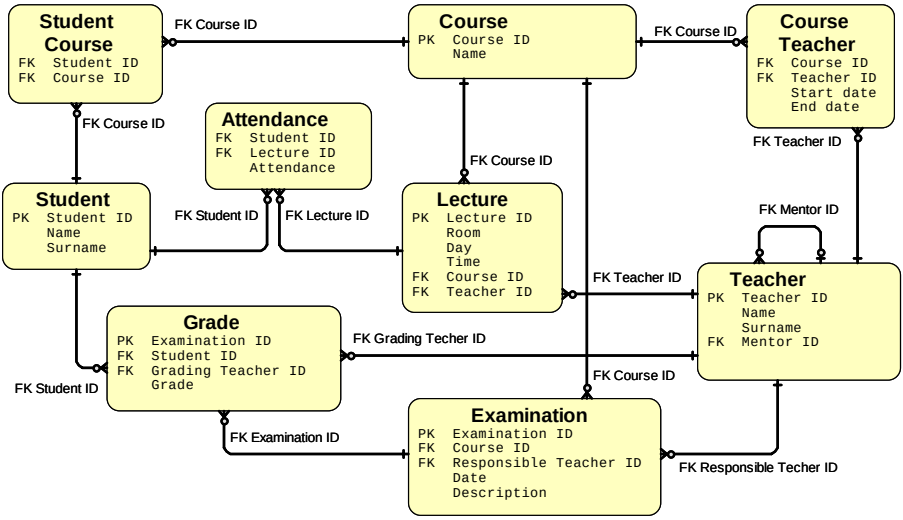


Fig. 2. A sample diagram of an entity-relationship database

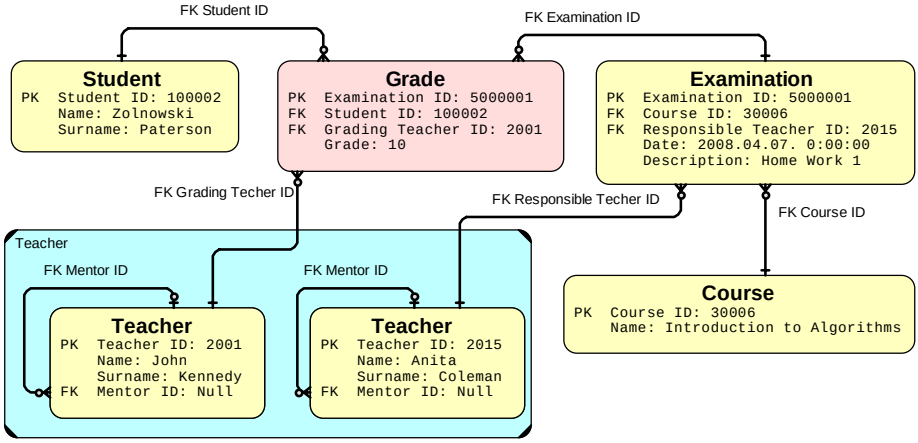


Fig. 3. A graphical form of related information for the fixed Grade record

Our first option here is to present an SQL request which joins five tables – Grade, Student, Examination, Course and Teacher twice. In this SELECT request, we can immediately spot two areas in which there might be problems. If the tables are joined with the ordinary JOIN function, as opposed to the appropriate LEFT JOIN or RIGHT JOIN function, then some resulting rows can be lost. Secondly, the user may fail to notice that the Teacher table is used two times in the SQL request – each case being independent of the other. This means that two different local names must be applied. A proper SQL request can be made only by someone who has knowledge exceeding that of the average programmer or tester. There is the risk that the incorrect request will lead to an incorrect result.

```
SELECT * FROM Grade G
LEFT JOIN Student S ON G.Student_ID=S.Student_ID
LEFT JOIN Examination E ON G.Examination_ID = E.Examination_ID
LEFT JOIN Course C ON E.Course_ID=C.Course_ID
LEFT JOIN Teacher T1 ON G.Grading_teacher_ID=T1.Teacher_ID
LEFT JOIN Teacher T2 ON E.Responsible_teacher_ID=T2.Teacher_ID
WHERE G.Examination_ID=5000001 AND G.Student_ID=100002
```

The second option is to prepare separate requests for data selection from specific tables, presenting the values of fields related to the selected parameters for this purpose. These requests would be as follows:

```
SELECT * FROM Student WHERE Student_ID=100002;
SELECT * FROM Teacher WHERE Teacher_ID=2001;
SELECT * FROM Examination WHERE Examination_ID= 50000001.
```

Here we assume that the response will be:

```
<Examination_ID=50000001,
Course_ID=30006,
Responsible_Teacher_ID=2015,
Date='2008.04.07',
Description=Home Work 1>.
```

Now we can select information about Course and Teacher, prepared Examination:

```
SELECT * FROM Teacher WHERE Teacher_ID=2105;
SELECT * FROM Course WHERE Course_ID=30006.
```

For an ordinary programmer, this is a scenario which should lead to fewer mistakes than in the first SQL request that was described. This is also a scenario which makes it easier to spot incorrect data.

Someone who is not familiar with SQL requests as such, however, would not be able to do this work at all.

Ideally we would want to obtain this information without preparing SQL requests at all. A change in the viewpoint might be another desired element. This means that we choose a linked record from a different table and imagine that we "transfer ourselves" to that record. In that case we can find information that relates to the new viewpoint.

3 Traversing and Browsing Databases

In RDBMS or tools that are used to work with databases, the concept of database browsing usually refers to an exploration of the structure of the relevant database so as to present the various elements therein – tables, views, stored procedures, indexes, relationships, etc. Browsing also identifies the relationship among the various elements, and it makes it possible to view data from a specific table. The elements in the structure of the database are usually presented as a tree. The content of the table is displayed in grids, which ensure the filtration and arrangement of the data, changes in the order in which columns are presented, the hiding of columns, and other activities that make it easier to review the data in the table. One can also review the content of several tables in individual grids which are mutually independent of one another.

What we need to see is not only the records that are in our selected table, but also related records from other tables. That would give us more information about the records in the selected table, and we could also assess the correctness of the data in our table and in the related tables. Where necessary, we can shift our viewpoint to a linked table, where we can find related information from a new position.

For the database which we are reviewing here, let us assume that we have opened the Student table and chosen the records of student X. We see that there are related records in the tables Student_Course, Attendance and Grade. All of the records that are related to student X are automatically filtered and made available for review in these tables. If we look at the Student_Course table, we can see only the records that apply to student X. Now we have a new viewpoint from the Student_Course table, and we also see the related tables – Student and Course. When we move from one record to another, we should see the records of the initially chosen student X in the Student table, as well as the courses which student X has chosen in the Course table.

The selected solution of this task should be sufficiently universal to be utilised in various relation databases. The one thing which relation databases have in common is that they are based on the ER model. Essentially this is a graph in which the vertex is in line with the entity, and the edge is in line with the relationship. In our example, the ER diagram can be presented in a simplified way in the graph that is seen in Fig. 4. The vertexes of the graph are marked with the first letters of words that are used in the name of the entity.

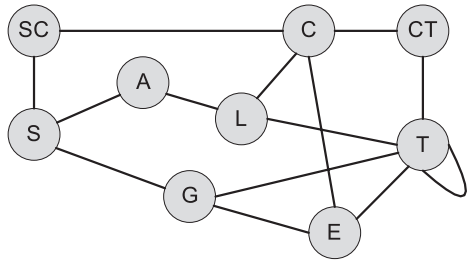


Fig. 4. The graph of entities and relations for the example

Traditional database management tools allow us to present the concepts of this simplified data model, and that means that we can review:

- the full list of vertexes (list of tables);
- the full list of edges (list of relationships);
- the list of edges for each vertex (the list of relationships for each table);
- the structure and properties of the vertex (a description of the table);
- the properties of the edge (a description of the relationship);
- the content of each vertex (viewing the table's record via filtration and various opportunities for visualisation);
- the content of several related vertexes via a special SQL request.

We offer new and standardised means to extract information from such a graph so as to reduce the shortcomings of the last point that we made. SQL requests require

IT skills. Time must be invested in order to understand a data model precisely and to write up an SQL request. An SQL request can be erroneous, and that will lead to an incorrect data view. These problems are reduced, but by no means eliminated by the use of predefined views or stored procedures or scripts.

From here on in, we will call the vertex of our graph an entity so as to use concepts that are more appropriate when discussing data models.

Our proposal on the design of new tools is based on a simple principle. We choose an entity on the graph which we can call the source entity. We can then make note of those records in the entity that are of interest to us. All records that can be reached from fixed records via the use of edges (the relations in database) can be called achievable records. Records that can be reached from an achievable record also are achievable records. Any entities that contain achievable records can be called achievable entities. We can set up an achievable graph which includes the selected source entity with fixed records, as well as all of the achievable entities with their achievable records.

Because this makes it possible to select all of the records in a database in a general case, it is necessary to set several sensible limitations on the selection of related records so that the set of records which is obtained is manageable. The limitations dictate those achievable entities from which we do not need to achieve following achievable entities. Let us call these destination entities. One of the simplest limitations in this kind will be this: we will define neighbouring entities as destination entities. We make use of only one record in the source entity and make it possible to review destination entities, automatically showing only the achievable records from this fixed record.

The graph in Fig. 4 is a very simplified model of the relevant ER diagram (Fig. 2) because the relations are depicted with a simple edge. Let us make use of the fact that each relationship between two entities in the ER diagram determines the information that is needed form a single entity, as well as the related information that can be obtained from the other entity. The cardinality of the relationship at both ends of the relation determines how many records from the other entity are or can be associated with or linked to the record in the first entity. Not to make the model which we are reviewing here too complex, let us assume that it is possible to form links from one field in one entity to one field in the second entity.

We can also imagine each relation as a transition from the specific record in one table to records in another table in accordance with the type of cardinality. We can use the ER diagram (Fig. 2) and the simple graph (Fig. 4) to prepare a new graph – the Browsing Transition Graph (BTG). Each edge or relation is changed into two oriented edges or transitions. If X is the source entity, and Y is the achievable entity, then the transition determines that if the record from entity X (the beginning of the transition) has been identified, then it is possible to find linked records in entity Y (the end of the transition).

This example offers us the image that is seen in Fig. 5.

The transition can also have properties that can be depicted visually and are obtained in accordance with the relation's cardinality, as well as the properties of the specific database:

1) a single-arrowhead edge – the record from entity X is linked to no more than one record from entity Y;

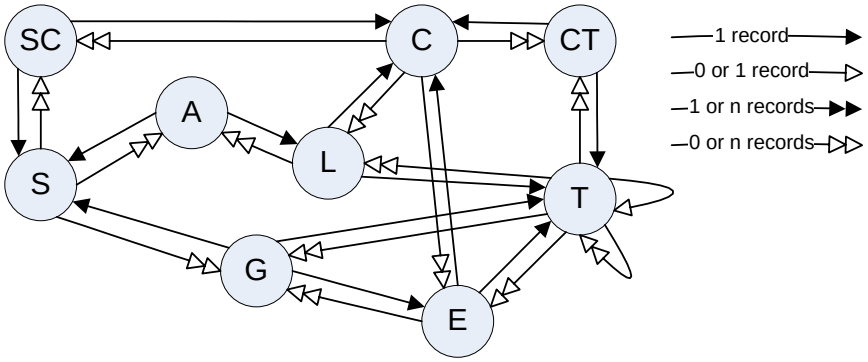


Fig. 5. A Browsing Transition Graph (BTG)

2) a double-arrowhead edge – the record from entity X is linked to an unlimited number of records from entity Y;
3) the full arrowhead – the record from entity X is not linked to fewer than one record from entity Y;
4) the empty arrowhead – the record from entity X might not be linked to any record from entity Y.

In our sample database (Fig. 2), each record in the Lecture entity describes a specific lecture and is linked to one specific Teacher entity. That, in turn, defines the one teacher who will teach the relevant class (in the BTG, this is a transition from entity L to entity T). In the opposite direction, the transition from entity T to entity L specifies that there might be many classes which are taught by the same teacher, while there can also be teacher who teaches no classes at all.

A double-arrowhead edge should make us cautious, because the number of linked records could be enormous. That would mean extra time to select the records and to deliver them to the end user. There would also have to be additional memory space to store and transport the relevant records.

Usually the records from the linked entity can be obtained relatively fast. This means that the necessary records from entity Y can be found on the basis of existing indexes in a period of time that is no longer than log(*n*), where *n* is the number of records in entity Y. Alternatively *n* might be so small that we can check every record from entity Y in order to select the right ones.

Sometimes there might be a need for a considerable amount of time to find the necessary records in entity Y. This means that we do not have the necessary indexes, and we will have to review all records from entity Y to select the ones that are necessary. If, for instance, we have information about attendance at lectures over the course of several years, and if there are many students to whom such information apply, then that means that we would have to inspect more than one million Attendance records (transition (S, A) in graph BTG) in order to specify the attendance of a specific student (a record from the Student entity).

The term database traversing refers to movement in the BTG. This involves specifying that the new source entity is one of the achievable entities, and then only the

records which can be accessed before that are depicted in this new source. As soon as a new source entity is defined, it identifies achievable entities, and records therein come from the records which are identified by the source of the new entity.

The term browsing view refers the view of the source entity, the transitions and the linked records in the achievable entities. The definition of the view defines the entities, transitions and records that are to be displayed and the way in which they are to be visualised for the end users. The tool used for browsing and traversing a database should ensure several pre-defined views so that the user does not have to do any additional work to receive the necessary information from the related entities.

Databases do not always contain definitions of all possible relations among entities. This means that we must allow the person who is configuring the tool to establish additional transitions in the BTG, indicating linked entities and linked fields.

When we define various browsing views, we must certainly think about protection against cycles which exist in the BTG. Simple protection involves defining the maximum visibility or distance of achievability (i.e., the number of transitions) from the source entity to the achievable entity. A slightly more intelligent protection would take into account the number of cycles in the BTG, as well as the total number of achievable records.

4 The Tool's Prototype

The authors of this paper have years of experience with various information systems. Many of those systems were based on database meta-models. They were used to integrate information in single systems and under the framework of several heterogeneous systems. Closest to the principle that is described in the previous chapter was a system which had the essential goal of integrating information [3].

The tools that have been developed in the past were information systems that were based on a logical data models. There was no tool, however, to survey a physical database. Principles of such tool were given in [4]. As the complexity of information systems increased, developers complained more frequently that traditional database administration tools did not ensure sufficiently convenient review of data. They asked for a simple and universal database browsing tool which would display tables of information related to the record without having to write up an SQL request.

4.1 Requirements for the Tool's Prototype

We developed a prototype for database browsing and traversing that was based on two simple browsing views. The views were chosen with the basic principle that the critical needs of users had to be satisfied as much as possible with as little developmental effort as possible.

These were the requirements for the tool's prototype:

- the tool must operate on the basis of the Browse Transition Graph (BTG) that was established for the database;
- the tool has to be able to read database metainformation from various popular RDBMS;

- the BTG has to be established automatically after a reading of meta information from the selected database;
- the system must provide means to choose a table that serves as the initial source entity for the BTG and makes it possible to select records on the basis of criteria entered for this table;
- the system must represent selected records and the related information on the basis of the browsing view;
- the system must ensure traversing to another entity in accordance with the browsing views.

The first browsing view provided for the display of the following data, as well as the following traversing opportunities:

- to create the selected source entity records on the basis of grid view control (grid view control ensures the filtration and arrangement of selected records, changes in the succession of columns, and other opportunities for visualisation);
- to display all transitions from the source entity to linked entities on the basis of grid view control;
- when user chooses a single record from the source entity and one transition, the system must display other relevant achievable records on the basis of grid view control;
- the determined transition must enable database traversing to the relevant achievable entity.

The second browsing view ensures the following data display and traversing possibilities:

- displaying fields of records from the identified source entity on the basis of tree view control (tree view control provides for a display of selected records in the form of a tree, with sub-tree collapsing and expanding, as well as other opportunities for visualisation);
- finding all achievable entities and achievable records that can be accessed on the basis of transitions with the single arrowhead transition and displaying them in a hierarchical tree;
- representing as child nodes all transitions to achievable entities in which the relevant field is used;
- enabling database traversing to the relevant achievable entity from any transition that is displayed on the tree.

4.2 The Architecture of the Tool

The architecture of the tool is represented in Fig. 6. The main elements of the tool are the Meta Database, the Main Engine, the Wrapper and the Presentation Engine.

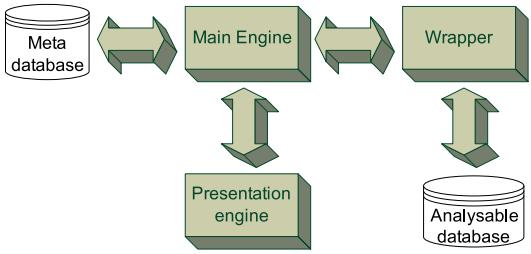


Fig. 6. The architecture of the prototype tool

The Meta Database contains a part of the description of the relevant database's structure, which is necessary to establish a BTG. The ER model for the Meta Database is seen in Fig. 7. Table "Tables" offers information about tables from the database. Table "Fields" offers information about fields in the table. Table "Relations" offers information about relationships among the tables. The Meta Database is filled-in when a connection to the database is established. The SQL requests that are necessary to fill in the Meta Database are specific to each specific RDBMS.

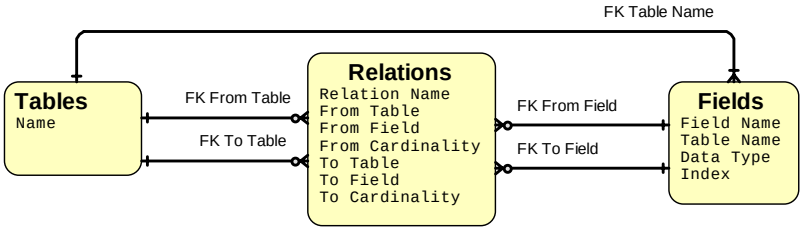


Fig. 7. A Meta Database

A Wrapper is a module which ensures the selection of metadata and data from the database that is being analysed. These are the functions which the wrapper provides in obtaining metadata:

- selecting the list of tables from the database, thus filling data in the "Tables" table;
- selecting the list of fields from the database, thus filling data in the "Fields" table;
- selecting the list of relationships from the database, thus filling data in the "Relations" table.

In order to obtain data, the wrapper provides the following request:

```
SELECT * FROM <Table> WHERE <Filter Expression>.
```

The Presentation engine is a module which displays information and offers navigation opportunities to traverse the database.

Let us explain operation of tool prototype on an example given in Chapter 2. Fig. 8 is a screenshot of the prototype we have developed.

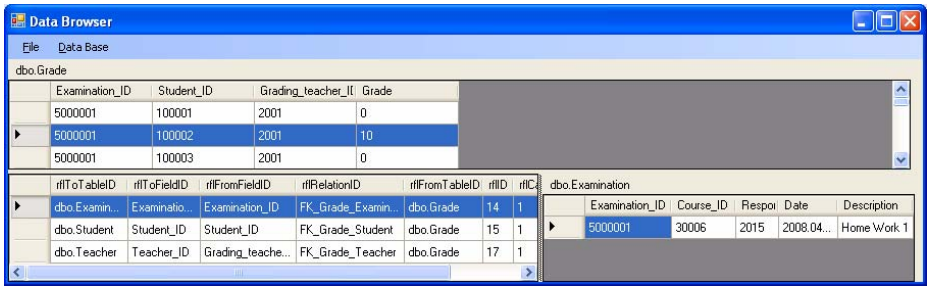


Fig. 8. A view of the Grade and Examination entities

At the top we see the records from the source entity (the Grade table), which are in line with the filter Grading_Teacher_ID=2001. One of the Grade records is fixed. At the bottom left, we see a list of transitions (relationships from the Grade table), which notes the transition to Examination, which is an achievable entity. To the lower right we see the achievable records from the Examination entity which is in line with the fixed source entity record and the fixed transition.

If we change the record of the source entity or transition, we also change the achievable records. The change in the fixed record involves a mouse click on the relevant grid view control record. In choosing the second record in the transition grid view control, for instance, we will find that records from the Student table will be displayed as achievable records. These will be in line with the filter Student_ID=10002 (Fig. 9).

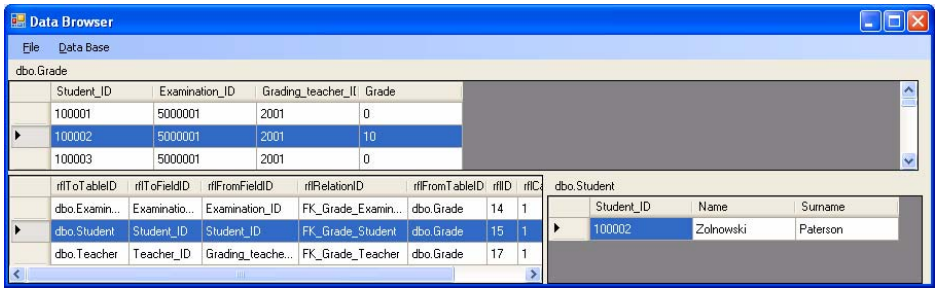


Fig. 9. A view of the Grade and Student entities

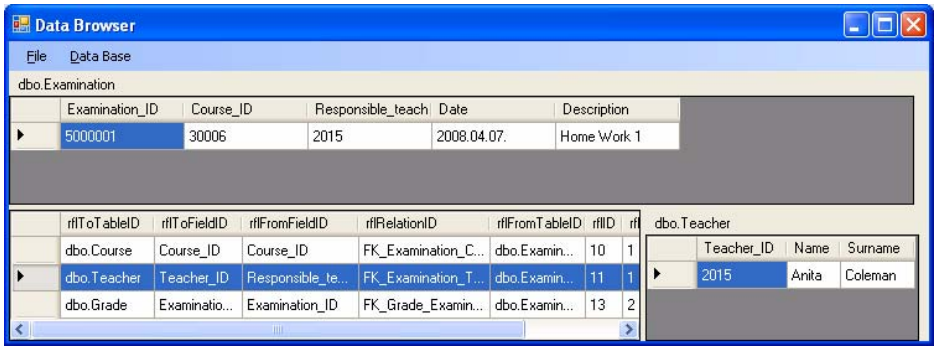


Fig. 10. A view of the Examination and Teacher entities

Database traversing here is ensured with a double mouse click on the relevant record from the transition grid view control. Here we have double-clicked on the first record from the transition grid view control, which means that our source entity is the Examination table. It in turn shows records which are in line with the filter Examination_ ID=5000001 (Fig. 10).

Fig. 11 shows how linked information is depicted in a tree. This is the second additional option for the browsing view.

The first level of the tree represents the source entity (the Grade table), with all relevant fields and values. If a field is involved in a transition, then that is represented as field node children under that field (<To_Table_Name> <To_Field_Name>). Depending on the properties of the transition, information about achievable records is displayed in different ways.

- If the transition has the single-arrowhead edge, which means that the number of achievable records is 0 or 1, the names and values of achievable record fields are attached as transition node children. For example, the Grade field Grading_ Teacher_ID is used in the transition to the Teacher_ID field of the Teacher entity. This means that under the node of this field, the child node is the relevant information about transition (dbo.Teacher Teacher_ID). It, in turn, has the values of achievable record fields as child nodes.

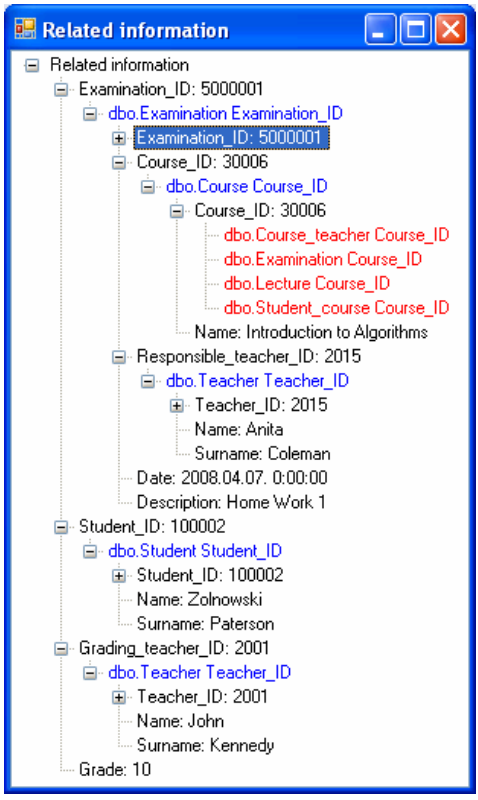


Fig. 11. A view of the Grade entity in the tree version

- If the transition has the double-arrowhead edge, which means that the number of achievable records is unlimited, then there are no transition node children. For instance, the Course_ID field in the Course achievable entity of the Grade entity is involved in four transitions (dbo.Course_Teacher Course_ID, dbo.Examination Course_ID, dbo.Lecture Course_ID, and dbo.Student_Course Course_ID).

In the example displayed in Fig. 11, information about transitions with the double-arrowhead edge is collapsed and is not visible. In the depiction of information about fields from achievable entities, transitions are arranged on the basis of the same algorithm that is used for the source entity.

The Main Engine provides the operating logic of the tool. At first, the Main Engine makes use of the Wrapper to hook up to the selected database and receive metainformation about its structure so that the BTG can be established. Then management is passed to the Presentation Engine component, and that allows the user to choose the source entity and the criteria for selecting records. The Main Engine then uses the Wrapper to select the relevant records so that they can be depicted by the Presentation Engine. The Main Engine maintains information about the current condition of the source entity and the selected browsing view, and it also supports the traversing mechanism.

In Fig. 11 we see the same record which was used in our example. All of the requested linked information is presented automatically.

The main shortcoming of the prototype from the perspective of the view functionality is that it has two "hard-coded" browsing views. Unfortunately there are situations in which a different browsing view is needed. When selecting a student, for example, we might want to see all the classes which the student is taking, but that's not possible with the current browsing view. If we want that information from the prototype, we will have to define the student, look at the linked records in the Student_Course table, and then select the records one by one to see the names of the courses.

5 Conclusions and Future Work

Over the course of the last 15 years, we have produced many informative systems that are largely based on a data model, are universal, and are fairly independent from the use to which they are put. We had to seek out general ways of obtaining, integrating and displaying information. The solutions were focused on the end users of information systems, but not on the developers of such systems.

We regularly received complaints from system developers about problems in the maintenance of databases, and we recommended the use of appropriate tools which better display the relevant data and their linkage to other data. Much to our surprise, our search for ready-made tools was unsuccessful. That does not meant that there are no such tools – perhaps we just didn't have the skills that were needed – but it is certainly a sign to show that many developers have to deal with problems on their own. Others may be as unskilled as we were in searching for an appropriate tool.

We applied the experience of our previous solutions to show how to put together a tool which offers not just the traditional view of physical data in a single table, but also a view of the nearest surroundings at the physical level of the database. We have presented

what we consider to be the necessary minimum to explain the essence of our idea. We demonstrated the two simplest views and traversing principles. This is a simple and even primitive tool, and it does not require vast resources. In a large project, it pays off to establish such a tool if an appropriate one is not available in the software market.

The prototypes which we developed were demonstrated to several developers and system maintainers who need to study data from databases on a daily basis. The positive evaluation of the prototype tool from all of the users was a surprise, and the tool already can be used in real life. Users reported that the tool makes it substantially easier and faster to select and review data. They also recommended a whole series of improvements, which would make the tool more effective. The primary functional improvements which users would like to see implemented are the following:

- the ability to configure the table fields that are viewed and the order in which they are presented, with full ability to store the configurations that are established;
- a back-forward system which would make it possible to return to data that have been viewed recently (in the reviewed prototypes, the transitions between two entities in the BTG are basically asymmetrical, and when one moves back, a different set of records is obtained);
- the ability to edit data in a way that allows maintainers and testers of databases immediately to apply necessary fixes when data mistakes are identified.

These are the requests which indicate the priorities of users who want to prepare such a database viewing and traversing tool. This might be more important than creating new and additional views or the many data integration and traversing opportunities which the transitions in the BTG might potentially offer. Work on a new version of browser with improvements mentioned above is in progress.

Acknowledgments.

This work has been supported by the European Social Fund Project No. 2009/0216/1DP/1.1.1.2.0/09/APIA/VIAA/044.

We wish to thank Karlis Streips for improving the English of this paper.

References

1. P. P.-S. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1), 1976, pp. 9–36.
2. V. Arnicane. Use of Non-IT Testers in Software Development. In: J. Münch, P. Abrahamsson (eds.), *Product-Focused Software Process Improvement*. Lecture Notes in Computer Science, Vol. 4589. Berlin-Heidelberg: Springer-Verlag, 2007, pp. 175–187.
3. G. Arnicans, G. Karnitis. Heterogeneous Database Browsing in WWW Based on Meta Model of Data Sources. In: J. Barzdins, A. Caplinskas (eds.), *Databases and Information Systems, 4th International Baltic Workshop BalticDB & IS 2000, Selected Papers*. Vilnius, Lithuania, May 1–5, 2000. Kluwer Academic Publishers, 2000, pp. 167–178.
4. G. Arnicans. Application generation for the simple database browser based on the ER diagram. In: Jānis Bārzdiņš (ed.), *Databases and Information Systems, Proceedings of the Third International Baltic Workshop*, Volume 1, Rīga, 1998, pp. 198–209.

LANGUAGES FOR MODEL-DRIVEN DEVELOPMENT

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 77–98 P.

Transformation Synthesis Language – Template MOLA

Elina Kalnina, Audris Kalnins, Edgars Celms, Agris Sostaks, Janis Iraids
University of Latvia, IMCS, Raina bulvaris 29, LV-1459 Riga, Latvia
Elina.Kalnina@lumii.lv, Audris.Kalnins@lumii.lv, Edgars.Celms@lumii.lv,
Agris.Sostaks@lumii.lv, Janis.Iraids@lumii.lv

Higher-Order Transformations (HOTs) have become an important support of the development of model transformations in various transformation languages. Most frequently HOTs are used to synthesize transformations from different kinds of models, for example, mapping models. This means that model-driven development (MDD) is successfully applied to transformations as well. The standard HOT solution is to create the transformation as a model using abstract syntax. However, for graphical transformation languages, a significantly more efficient solution would be to create the transformation using its graphical (concrete) syntax. An analogy here could be the textual template languages such as JET which directly create texts from a model in the concrete syntax of the target language. This paper introduces a new kind of language – a graphical template language for transformation synthesis named Template MOLA. This language is used for creation of transformations in the MOLA transformation language. Template MOLA is an adequate solution for many typical HOT applications.

Keywords: higher order transformations (HOTs), model transformations, template-based language, Template Mola.

1 Introduction

Model-driven development (MDD) has recently become a widespread technology for various kinds of software development. In addition to modeling itself, the key support feature of this technology is model transformations. This has given rise to various model transformation languages, both textual and graphical. We can state that transformation development has become an essential part of software development, with transformation languages being a domain-specific development environment. This domain is characterized by the fact that it itself is well defined by models. Therefore, MDD can be naturally applied to transformation development, i.e., transformations are used to create transformations, as a rule, in the same language. This kind of transformations is named Higher-Order Transformations (HOTs). The idea of HOTs can be applied to virtually any model transformation language. However, the largest number of HOTs important in practice has been created in the ATL language [1], probably due to the fact that the largest known number of transformations has been created in ATL. Automatic creation of transformations from various mappings between two models is especially popular. A large set of such mappings have been obtained by applying the ATLAS Model Weaver (AMW) [2] – a special framework for defining a mapping between two models on the basis of their metamodels. The mappings obtained with AMW can be considered a sort of high-level specification of the required model transformation. However, the idea of obtaining a transformation from a mapping is in no way restricted to AMW and ATL and refers to other transformation languages as well.

A comprehensive survey of HOT applications is given in [3]. They are classified into four types, according to the respective types of input and output models. One of the application types is transformation synthesis. This type is most relevant to the research presented in this paper. Transformation synthesis means transformation generation from different sources of information, including the model mappings mentioned above.

In the HOT approach, transformations must be treated as models conforming to the relevant metamodel. There is such a transformation metamodel for almost all transformation languages. If we want to generate transformations in a transformation language, the metamodel of this language will be the target metamodel of the particular HOT. In [3] synthesis of ATL [1] transformations is considered. An ATL model is created and then extracted as a transformation text (since ATL is a textual transformation language). The same task could be performed for graphical transformation languages, for example, MOLA [4]. A MOLA transformation in abstract syntax (the MOLA transformation model) could be created easily in the same way as the abstract syntax of ATL transformations. The transformation visualisation task is harder since graphical MOLA diagrams have to be created. However, it is also technically feasible. At first a transformation to the corresponding presentation model (graphical diagram) should be executed. Then some auto-layout creation library for graph diagrams should be used. It should be noted that for transformation execution visual representation is not needed. Consequently, for graphical transformation synthesis, MOLA (or ATL) could be used as a HOT. However, a better solution is proposed in this paper.

There are many template-based model-to-text languages. For example, popular languages are JET [5], mof2text [6], Xpand [7], Epsilon Generation Language [8]. The basic application of these languages is to create code (in Java, XML or in any other required language) from the PSM model in the standard MDD process. These languages typically contain facilities to navigate the given model according to its metamodel. However, the main advantage of these languages is the possibility to define the text fragment to be generated by the given rule as a textual template in the relevant concrete syntax (Java, XML or any other). The constant parts are fully defined by the template itself. The variable parts in the text to be generated are specified by means of template expressions which typically contain model class attributes and auxiliary variables. These languages have confirmed their practical applicability in code generation for several years.

Besides the approach described above, an ATL transformation text could also be created using some template-based model-to-text language. Since MOLA is a graphical transformation language and fundamentally model-based, textual template languages cannot be applied here. In this paper we address the problem of MOLA transformation synthesis using template-based mechanisms.

A new graphical template-based language Template MOLA for MOLA transformation synthesis is proposed in this paper. In this language elements to be created in MOLA can be defined explicitly in syntax close to traditional MOLA statements. The generation logic in Template MOLA is described by traditional MOLA facilities. This part of the description is executed during the generation process. The elements to be placed in the created transformation are described in a MOLA extension consisting of template statements. This extension again is similar to traditional MOLA, but with a possibility to incorporate template expressions as well. During generation, these expressions are

replaced by the corresponding generation time values based on the elements of the source model. Thus, the idea of textual template languages is leveraged to a graphical language. The main advantages of the template approach are retained – adequate facilities to process and navigate the source model and concrete syntax-based descriptions of elements to be created as a result. The proposed solution is shown to be significantly more convenient for transformation generation than pure use of MOLA as a HOT.

A short description of MOLA is given in Sub-section 2.1. Sub-section 2.2 describes Template MOLA in general. Section 3 describes the metamodelling aspects of Template MOLA. Section 4 describes Template MOLA in detail. Section 5 outlines general implementation principles.

2 A General Description

The Template MOLA language is an adaption of template mechanisms used for textual template languages (of the model-to-text kind) to a graphical language. It is based on the model transformation language MOLA. Template MOLA is used for easy generation of transformations in MOLA from various input models – as a substitute for the classical HOT approach.

All MOLA elements are retained in Template MOLA. Additionally, special template elements for easy MOLA transformation synthesis are included. With them it is possible to define explicitly in a graphical syntax which MOLA elements should be created.

Because of this close integration of MOLA and Template MOLA, we start this section with a short MOLA description. We continue with a description of basic Template MOLA concepts.

2.1 MOLA

MOLA [4] is a graphical transformation language developed at the University of Latvia. It is based on traditional concepts of transformation languages: pattern matching and rules defining how the matched pattern elements should be transformed. The formal description of MOLA as well as a MOLA tool can be downloaded at [9].

A MOLA program transforms an instance of a source metamodel into an instance of a target metamodel. The two metamodels are specified using the EMOF compliant metamodelling language (MOLA MOF). These metamodels, which may also coincide, are both part of a transformation program in MOLA. Mapping associations may be added to link the corresponding classes in source and target metamodels.

MOLA is the model transformation language which combines the imperative (procedural) programming style with declarative means of pattern specification. A transformation written in MOLA consists of several *MOLA procedures* where one of them is *the main.* An example of a MOLA procedure is given in Fig. 1. The execution of a MOLA program starts with the main procedure. Procedures in MOLA may be called from the body of another procedure using *call statements*. Like in most transformation languages, class instances, primitive and enumeration-typed variables can be passed on to the called procedures as parameters. There are other types of statements in MOLA as well, i.e. *rule, foreach loop, text statement,* etc. The execution of a MOLA procedure starts with the *start statement.* The next statement to be executed is determined by the outgoing control flow.

The rule in MOLA represents the classical branching (*if-then-else*) construct of imperative programming. A rule contains a declarative pattern that specifies instances of which classes must be selected and how they must be linked. Only the first valid pattern match is considered. The action part of a rule specifies which matched instances must be changed and what new instances must be created. The instances to be included in the search or to be created are specified using *class elements* in the MOLA rule. The traditional UML instance notation (instance_name:class_name) is used to identify a particular class element and specify the class the instance must belong to. Class elements included in a pattern may have attribute constraints – simple OCL-like expressions. Expressions are also used to assign values to variables and attributes of class instances. Additionally, the rule contains association links between class elements. A class element may represent an instance, matched previously by another pattern. Such class element is called a reference class element and is specified using the name of the referenced class element, prefixed with "@" symbol.

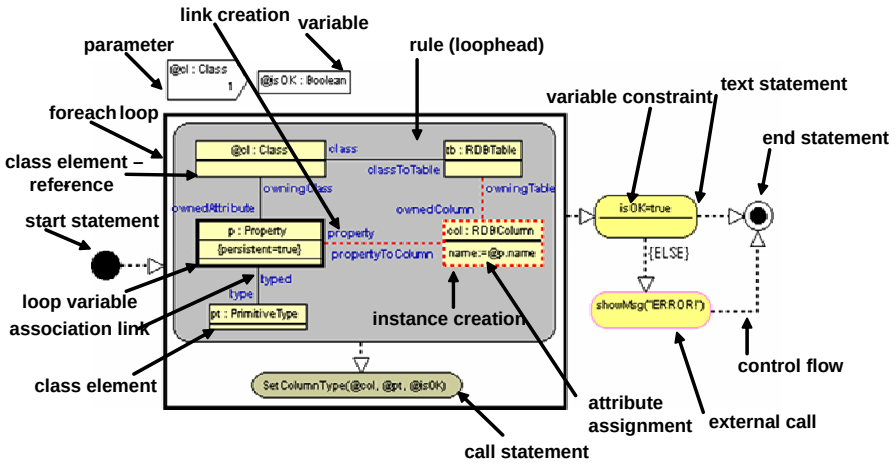


Fig. 1. A MOLA diagram example. The loop is executed over all Property instances which have Primitive Type and belong to referenced Class instance if it is already mapped to an RDBTable

Typical transformation algorithms require iteration through a set of the instances satisfying given constraints. In order to accomplish this task, MOLA provides the *foreach loop* statement. The *loophead* is a special kind of rule used to specify a set of instances to be iterated in the foreach loop. The pattern of the loophead is given using the same pattern mechanism used by an ordinary rule, but with an additional important construct. It is the *loop variable* – the class element that determines the execution of the loop. The foreach loop is executed for each distinct instance that corresponds to the loop variable and satisfies the constraints of the pattern. In fact, the loop variable plays the same role as an iterator in classical programming languages.

The above example demonstrated the concrete graphical syntax of MOLA. The MOLA language also has an abstract syntax defined by means of a metamodel containing several packages (see the MOLA reference manual available in [9]). The abstract syntax of MOLA MOF is defined in the *Kernel* package, with elements *Class*,

Type, *Property*, *Association* and others (actually, it is a subset of a UML 2 class diagram metamodel). The abstract syntax of MOLA procedures is defined in the *MOLA* package (containing *Rule*, *ClassElement*, *AssocLink*, etc). In the next sections, this abstract syntax is referenced where necessary, for example, *Kernel::Class* means a metamodel (MOLA MOF) class.

2.2 Template MOLA

In this sub-section, the basic constructs of Template MOLA are described. The proposed Template MOLA language contains two kinds of MOLA statements: generation statements and template statements.

Generation statements are executed during the transformation generation process. They are used to define the logic of generation process on the basis of the provided input metamodel. All ordinary MOLA statements may be used as the generation statements.

Template statements are meant to be "copied" to the generated "MOLA code" (in fact, model) with template expressions replaced by the appropriate generation time values. Template statements look similar to ordinary MOLA statements but can be distinguished by their graphical style – green color. The most used template statements are template rule and template loop; however, other MOLA statements may be used as template statements too.

Statements in Template MOLA are organized into procedures in the same way as in the traditional MOLA described in the previous section. A procedure may contain both generation and template statements; however, the generation statements alone should constitute a valid MOLA procedure. Template statements may be interspersed between generation statements. Thus, the general idea of Template MOLA is that the "generation part" of a procedure is executed in the same way as the traditional MOLA. The only difference is that template statements to be executed in this process are copied to the resulting traditional MOLA procedures (instead of directly executing them). Certainly, there are some more complex situations to be described further, but at the first glance, Template MOLA means exactly that.

The most used template statement is *template rule*. In generation time it is copied to the generated "code" (i.e., to the relevant generated MOLA procedure). Elements of the template rule may contain variable textual parts – *template expressions* (expressions enclosed in angle brackets followed (preceded) by a percent sign). These expressions are replaced by the corresponding generation time values.

Example of a template rule can be seen in Fig. 2. In this rule, the constraint in class element *b:Class2* contains the template expression *<%@p.name%>* where *@p* is a known generation time reference (defined in the procedure containing this rule). Another kind of a variable part in a rule is a template expression specifying the class of a class element (here *c:<%@tc:Class%>*). The generation time reference *@tc* must point to an appropriate metamodel class, i.e., it must point to an instance of *Kernel::Class* (the *::Class* suffix in the syntax emphasizes that), and it must be set before the rule under discussion is to be executed. In the resulting traditional MOLA rule, this template expression is replaced by the referenced class name. Association links may also be specified by a template expression in order to adapt to a variable class element at the end. This template expression (*<%@assoc:Association%>* in Fig. 2) must reference an association in the metamodel. The value of this reference must certainly be set correctly

during the generation; in the presented example only the association linking classes *Class2* and *Class3* is valid. In the generated rule, the standard MOLA notation for association links (both role names) is used.

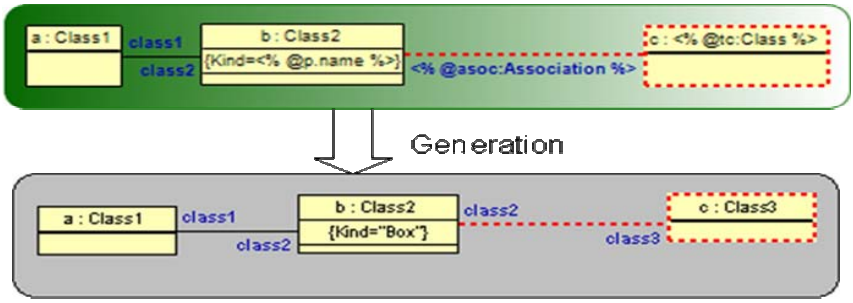


Fig. 2. An example of a template rule and the MOLA rule generated from it

The lower part of Fig. 2 shows the generated MOLA rule obtained from the template rule above. Here we assume that the reference *@p.name* has a string value "Box", the reference *@tc* points to the class *Class3* and *@assoc* to the association with role names (*class2*, *class3*).

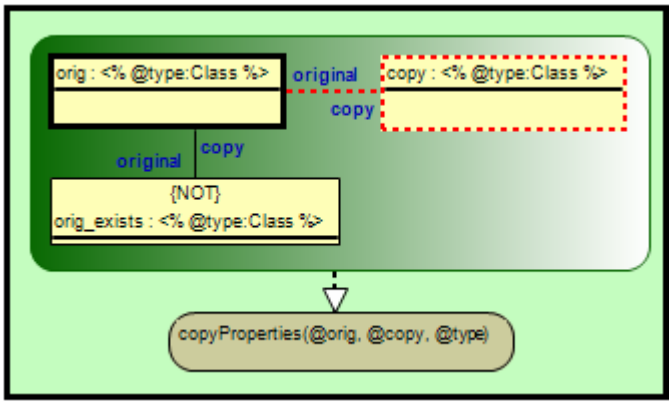


Fig. 3. An example of a template loop

Similarly to rules, the loop constructed in MOLA – the *foreach* loop statement – also has its template form in Template MOLA. The *template loop* is copied to the generated procedure during the generation process, including its body (which may also contain generation statements, see an example in Section 4). The template loop in its *loophead* rule can use all the extensions introduced for the template rule. Fig. 3 shows an example of a template loop, a simple construct for creating copies of all instances of an arbitrary class. In the loophead of this loop, the class to be used in all class elements (including the loop variable *orig*) is defined by a template expression *<%@type:Class%>* which means that the reference *@type* must be set to the required class before the given template loop.

Then a traditional MOLA loop is generated from this template loop, and the generated loop performs instance copying for the given class. The additional class element *orig_exists* with NOT constraint is used as a NAC (negative application condition) preventing from copying the copies again. The example presents a very simple case of another area of typical application of HOTs for transformation generation in [3] – building a generic transformation for a previously unknown metamodel.

The body of the loop in Fig. 3 contains another template-related construct – a MOLA procedure call with arguments of previously unknown types (*@orig* and *@copy*). The type of these arguments becomes known only during the generation process. The given procedure call contains one more argument – the reference to type itself. This last argument is a generation-time argument, which is not included in the generated invocation (it has no sense in that context). Yet for the generation of the procedure *copyProperties*, which has to perform copying of all attributes of arbitrary class, such a parameter could be of high value to define an appropriate generation time loop (traversing the attributes).

The exact kind of procedure parameters is visible in its declaration. There are three types of parameters that can be declared in a Template MOLA procedure – *template*, *generation* and *type* parameters. Template parameters are created in a generated procedure. Generation parameters are used in the generation time and are not created in a generated procedure. Appropriate arguments must be passed in call statements for the template and generation parameters. The type parameters are also used in generation time, but they are inferred from other parameters instead of passing them explicitly. Since the types of parameters in MOLA are described using class *Kernel::Type*, *type* parameters may refer to instances of *Kernel::Type* (*Class*, *PrimitiveType* or *Enumeration*) only.

We have already given an insight into template expressions used in Template MOLA; however, the example does not cover all possible use cases. Therefore, a short summary on template expressions follows. The most common elements where template expressions appear are class elements within a template rule. A template expression can be used to specify the class of the class element. In this case, the template expression must be a reference to *Kernel::Class* instance. If template expressions are used to specify the name of the class element, constraint or expressions in assignment, a string expression is used for this purpose. These expressions may contain generation time variables, parameters and attribute specifications, but not template element references. References to instances of appropriate classes can be used to specify the attribute to be assigned in a class element (a reference to *Kernel::Property*) and the association of an association link (reference to *Kernel::Association*). Template expressions can also be used in template text statements and in call statements to specify arguments which conform to template parameters of the called procedure.

The usage of template procedures in general is more widely discussed in Section 4. On the whole, the idea of generating template procedures in Template MOLA and providing appropriate naming conventions for them is based on principles similar to those in OOP languages such as C++ and Java, also containing some template mechanisms.

2.3 Template MOLA Compared to MOLA as a HOT

A question may arise for the reader, why is transformation synthesis in Template MOLA better than in traditional MOLA? Writing higher-order transformations for

transformation synthesis directly in MOLA requires to define creation of all MOLA metamodel elements explicitly (i.e., according to the abstract syntax of MOLA). To create one rule, we have to create the rule, all its class elements, all association links, all their sub-elements, and to map them to appropriate types from the metamodel of this transformation. Fig. 4 demonstrates a transformation for creation of one rule using traditional MOLA as a HOT language. Creation of the same rule in Template MOLA was demonstrated in Fig. 2.

It is easy to see that the code for creation of this rule in Template MOLA is significantly more readable than in traditional MOLA. Firstly, the size of the rule creation pattern differs significantly. Note that in this example we considered creation of a very simple rule. For more complicated rules, the difference is even more significant. The same situation holds for loops since they mainly consist of rules.

The same issue of complexity arises in regard to other transformation languages also usable for HOT tasks.

Template MOLA allows to implement the same HOT tasks with much less effort and with smaller amount of errors since the structure of the resulting MOLA statements is clearly visible already in the templates.

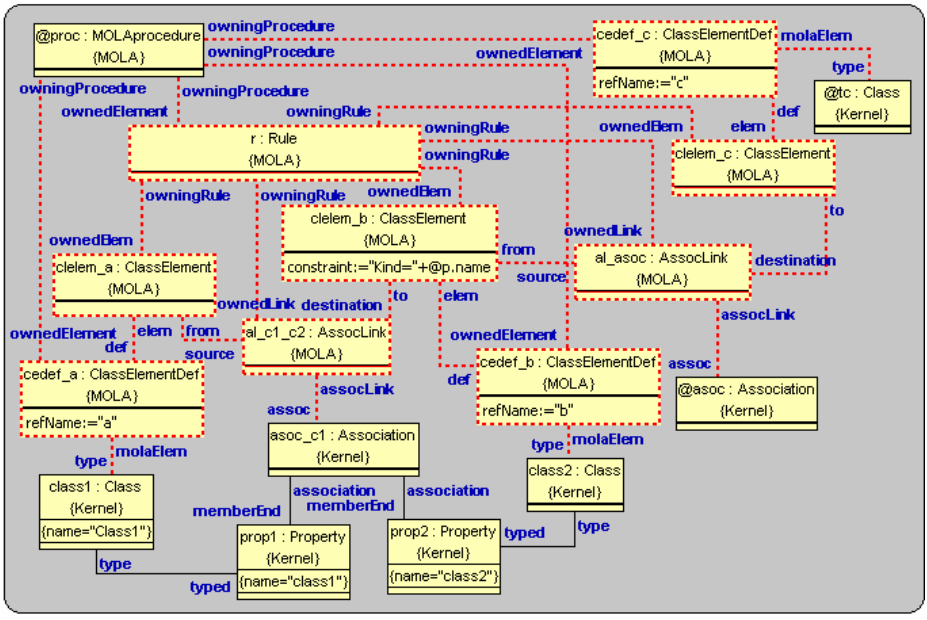


Fig. 4. Creation of the rule from Fig. 2 using MOLA as a HOT

3 Metamodelling Issues

As in any other transformation language, transformations in MOLA are based on the appropriate metamodel definition, frequently containing the source and target part. The definition of a metamodel for Template MOLA is more complicated because the relevant HOT level features for defining the generation logic have to be supported. At

the same time, the use of template statements requires that appropriate parts are present in the metamodel.

In order to have a deeper understanding of metamodelling issues in Template MOLA, we start with the comparison to the metamodel structure required for defining a traditional HOT in MOLA for synthesis of a MOLA transformation (an example of which was shown in Sub-section 2.3). Fig. 5 shows this metamodel structure. The source of the HOT is the source model (a mapping definition or something similar) corresponding to the source metamodel. The HOT must create a complete MOLA transformation definition consisting of a specific metamodel for this transformation (frequently containing the source and target parts) and the proper transformation (a set of MOLA procedures). Similarly, at the metamodel level, the definition of HOT is based on two metamodel parts that serve as a target metamodel for this HOT. Firstly, there are MOLA metamodelling facilities named MOLA MOF MM (the *Kernel* package mentioned in 2.1). Secondly, the MOLA procedure metamodel (MOLA MM) is required.

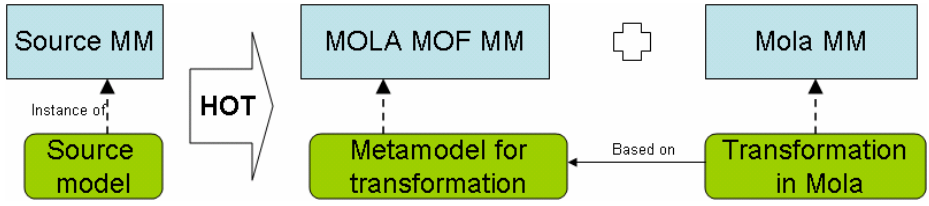


Fig. 5. Models to be used if higher order transformations are written in MOLA

A typical application of HOTs in general and Template MOLA in particular is the generation of transformations from mappings for metamodel-based graphical DSL tool building. The tool building platforms that really require it are METAclipse [10] and ViatraDSM [11]. However, the basic ideas can also be demonstrated in the popular Graphical Modeling Framework (GMF) [12] in Eclipse (we assume for a moment that transformations are generated in MOLA instead of Java for all actions). Fig. 6 illustrates the specialization of the metamodelling situation in Fig. 5, when MOLA transformations are generated by HOT for a DSL tool – i.e., we assume that the GMF generator is implemented as a HOT instead of being written in Java. The source metamodel now consists of several parts with different roles. A definition of DSL normally is based on the relevant domain metamodel (abstract syntax) using, in turn, a version of MOF as a metamodel (in particular, the MOLA MOF could be used in such a role). Another part of the metamodel used by GMF and similar platforms is the presentation type metamodel (named graphical definition metamodel in GMF) and the mapping metamodel. Together they provide the means for graphical syntax definition of a diagram and mapping definition from domain metamodel classes to presentation types in the diagram (by these means instances of the classes must be visualized). The generated transformations in runtime should use the same domain metamodel; therefore, this metamodel must be copied by the HOT to the generated transformation. There also is a constant part of the metamodel – the presentation metamodel (named notation metamodel in GMF) – which defines possible diagram elements at runtime. This constant part also should be created by the HOT. One of the tasks the generated

transformation should do in runtime is to create a visual diagram element for a new domain class instance (according to the defined mapping). Thus, two important special features have appeared in this application: the use of the domain metamodel in two different roles (part of the HOT source and part of the created transformation metamodel), and the constant (independent of the source) presentation metamodel is included in the created transformation. In fact, the reuse of part of the HOT source as a variable part of the metamodel for the created transformation is quite typical when transformations are generated by HOTs from mappings.

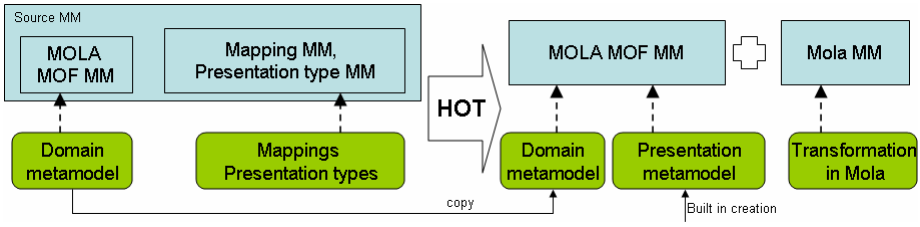


Fig. 6. Models in case MOLA is used as a HOT for tool building

Now we can show what the differences in metamodel structure are if Template MOLA is used instead of a standard HOT approach for the same tasks. Fig. 7 shows the general transformation synthesis by Template MOLA (an analogue of Fig. 5). The "runtime" metamodel for the generated transformation (more precisely, its variable part), as a rule, must also be provided as an input to the Template MOLA-based HOT implementation. This situation could certainly occur in the general case of Fig. 5, but in Fig. 7 this situation is clearly syntactically visible. It is due to the necessity to use template expressions for accessing classes of this variable metamodel part in template rules in a generic way (see Fig. 3). A typical example of such variable part is the domain metamodel for DSL definition (see Fig. 6). What is more different from Fig. 5 is the necessity to provide the constant part of this "runtime" metamodel for the definition of Template MOLA-based HOT. This is due to the fact that classes of this constant part are used to define "constant" class elements in template rules. Therefore, these classes must be defined before the definition of Template MOLA rules. Although this constant part of the metamodel is clearly an instance of MOLA MOF metamodel, in order to be referenced in "constant" Template MOLA elements, it must be provided alongside the MOLA MOF metamodel itself. Metamodel packages included in a complete transformation definition in Template MOLA belong to two adjacent metalevels. However, it is not confusing since the usage of their elements is

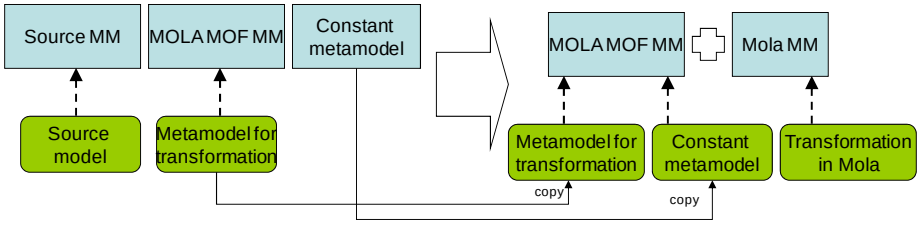


Fig. 7. Metamodels and models used for defining transformations in Template MOLA

clearly distinguished. Note that there may also be elements of another kind in the variable part, but we do not discuss this situation here as it is not typical of our applications.

Finally, we analyse the application-to–metamodel-based tool building in Template MOLA (Fig. 8). The main difference from Fig. 6 is that the presentation metamodel plays the role of the constant part of the metamodel for transformation. Therefore, it must be provided before the definition of Template MOLA. Note that classes for mappings and presentation types can only be used in the generation (non-template) rules and loops of Template MOLA (they play the role of the source metamodel). The domain metamodel is clearly the variable part of the metamodel for transformation. An example of this kind of application is presented in Sub-section 4.1.

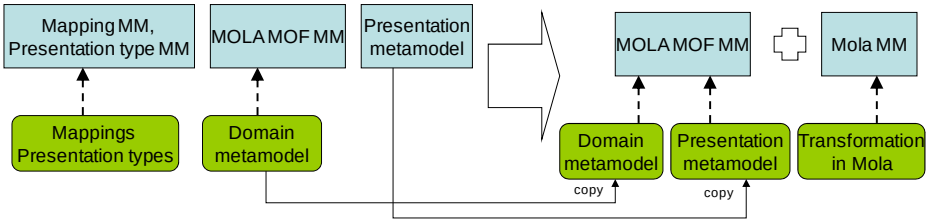


Fig. 8. Metamodels and models used to define transformations in Template MOLA for tool building

Now let us remark on the permitted use of metamodel elements in Template MOLA constructs. Source metamodel elements can be used directly only in generation (non-template) statements of Template MOLA. They can also be used inside template expressions in template statements. Elements of the variable part of the metamodel for transformation (the “runtime” metamodel) can be referenced via corresponding classes of the MOLA MOF in generation statements as well. The same elements can be referenced in template statements only via template expressions for types. The elements of the constant part of the metamodel for transformation can only be used in “constant” class elements in template rules.

4 Template MOLA Examples

In this section, we will demonstrate Template MOLA constructs on examples. In [3] two types of transformation synthesis are considered. We will present an example of each type.

The first is mapping implementation. It means we have some kind of a mapping model to describe dependencies between models. We can generate transformations to implement these mappings from this mapping model. In Sub-section 4.1, a mapping implementation example from the field of tool building will be demonstrated.

The second type is transformation creation for generic metamodels. In this case transformations for a concrete metamodel can be generated from generic transformations. In Sub-section 4.2, we demonstrate how Template MOLA could be applied to such use cases. An example of instance cloning is presented.

4.1 Transformation Synthesis from Mappings

In this sub-section, a simplified example of tool building is presented. Graphical domain-specific languages (DSL) are widely used nowadays. Several tool building environments have been introduced to support tool building for graphical DSLs, for example, GMF [12], MS DSL [13], GrTP [14], METAclipse [10]. In GMF a domain metamodel for DSL is defined in the first step. Then presentation types (in GMF terminology the graphical definition models) and tooling models are defined. Presentation types describe different graphical elements used in the graphical syntax of the language. The tooling models describe palette elements. Then a mapping model that links all these models together is defined. These models are used to generate the JAVA source of the DSL tool. This Java source precisely defines the tool behaviour. Alternatively, a DSL tool-building environment can be transformation-based, i.e., transformations are used to describe the tool behaviour, as it is, for example, in METAclipse. However, there are approaches that combine mappings and transformations [15]. In this case, mappings are used to generate transformations. Since transformation synthesis is needed there, it is a perfect opportunity for application of Template MOLA.

In this section, we use a specific task from the tool-building field as an example. We assume that we have instances of some graphical DSL in abstract syntax (a domain model), and we want to generate the corresponding visualisation (instances of the presentation metamodel). We can certainly write manuallya MOLA transformation, solving the task for this concrete DSL.

In our tool building environment we have means for domain metamodel definition as well as for mapping and presentation type definition; therefore, visualisation transformation for each DSL can be created in a generic way. It means we can build a generic transformation in Template MOLA from which the transformation for visualisation creation in a concrete DSL can be generated automatically.

To write the transformation, we need the corresponding metamodels (built according to the general schema in Fig. 8). A simplified metamodel version is used in this example. The domain metamodel is defined using a small subset of UML (see the upper left side of Fig. 9). Presentation types and a mapping metamodel are also needed. Instances of this metamodel are used as input in the generation time. Here we present a very simple integrated mapping and presentation type metamodel where minimal information on the intended graphical form is included directly in the mapping definition (see Fig. 9, upper right side). Instances of a domain class can be visualised as a box (*ClassToBox*) or as a line (*ClassToLine*). If the class is visualised as a box it may contain several text fields. In these fields, values of some class properties are usually displayed (*PropertyToField*).

During the visualization of classes, the generated transformation has to create instances of a fixed presentation metamodel supported by the tool (see the lower part of Fig. 9). These instances appear only in generated transformations. Therefore, the presentation metamodel is the *constant* part of the metamodel for the generated transformation (compare to Fig. 7 and 8). It describes a graph diagram with *Nodes* and *Edges*. There are *CompositeNodes* containing other *Nodes* and *Labels* for text visualization.

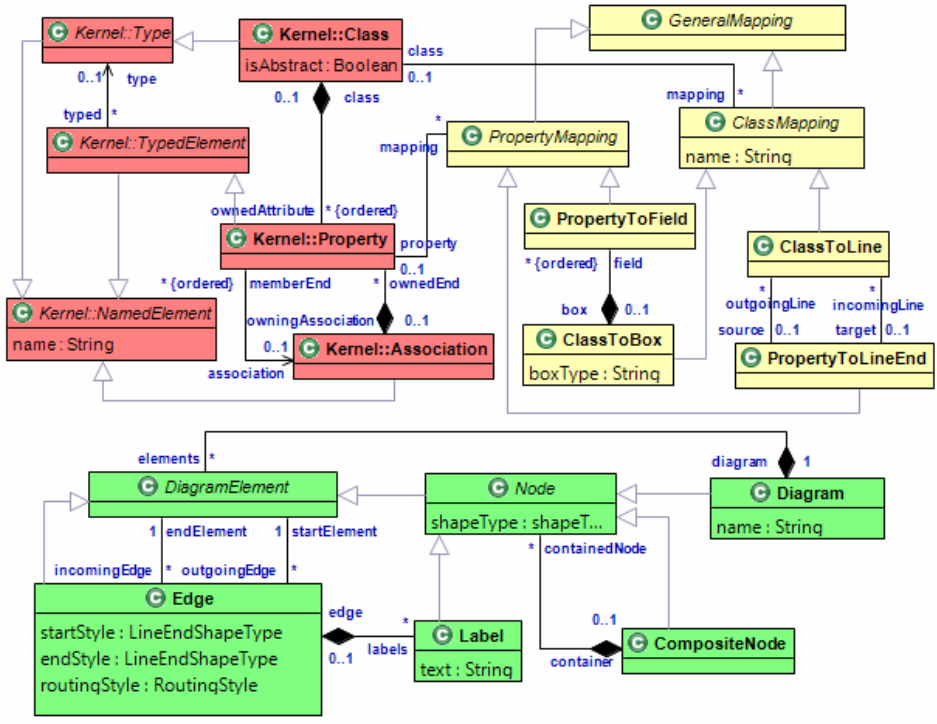


Fig. 9. A simplified domain (upper left side), mapping (upper right side) and presentation (lower part) metamodel

When metamodels and their roles are specified, we can move on to transformation definition in Template MOLA (see Fig. 10). We remind that the proper input for this generation transformation is a specific domain metamodel and a related mapping model. The transformation starts with the loop iterating through all instances of class to box mapping. This loop is a generation loop and is executed in the generation time. As a result, a traditional MOLA procedure is built, containing a loop for each such mapping instance (generated from the template loop which constitutes the body of the generation time loop). The generated loops simply follow each other linked by control flows. The template loop contains the loop variable with the name being generated. The loop variable name is the concatenation of letter *"i"* and the name of appropriate class given by template expression <%@c.name%>. The type of the loop variable is defined by the template expression <%@c:Class%>. In each generated loop the type (@c) is replaced with the concrete domain class corresponding to the mapping instance this loop is generated from. In each loop the value assigned to *shapeType* attribute is explicitly defined. This value is calculated in generation time using the corresponding mapping data (the template expression <%@cm.boxType%> directly references the *boxType* attribute of the current mapping instance). Now in runtime each generated loop iterates over all instances of the corresponding domain class and creates a box for each of them.

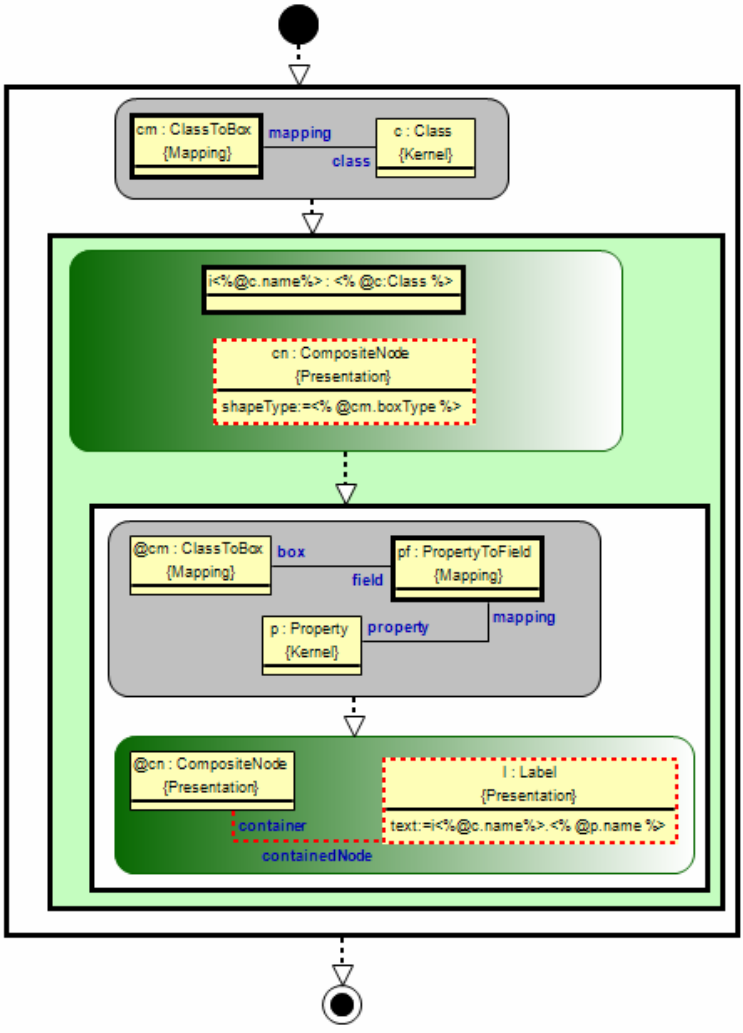


Fig. 10. Mapping implementation for tool building in Template MOLA

We must also generate transformations to create fields and set their values. Therefore, a rule for processing each field has to be generated in the loop body. To ensure this, in the template loop a generation time loop is included. This loop checks which field mappings are included into the given class mapping. For each such field, a rule is created. This rule adds a label to the box and sets its value. To set the value of the label, the relevant property value of the runtime instance should be used. To access this property, the template expression *<%@p.name%>* is used within the assignment in the template rule. During generation the generation time loop ensures that the template expression is replaced with the relevant property each time. It is not difficult to see that the generated sequence of rules will do exactly the required label creation. The structure of the generated procedure is shown in Fig. 11.

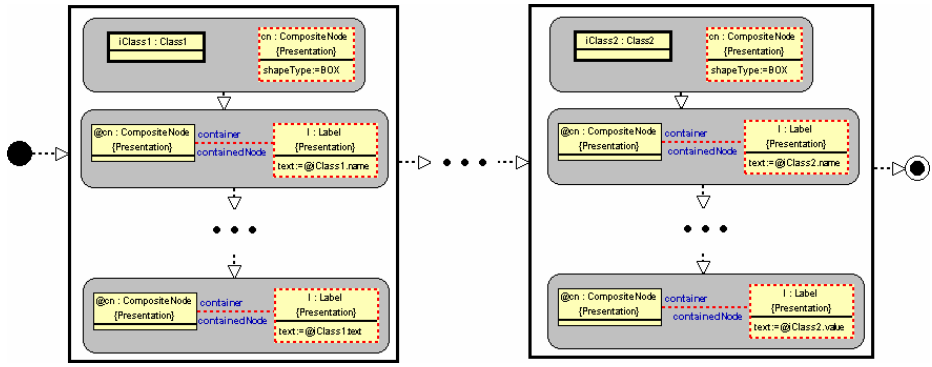


Fig. 11. A MOLA procedure generated for Fig. 10

4.2 Transformations for Generic Metamodels

Template MOLA can be used to write transformations for generic metamodels (the metamodel is unknown at the time of writing). For example, we can write a generic instance cloning procedure. More precisely, in Template MOLA we can write an instance cloning generator, then execute it for a concrete metamodel and run the generated traditional MOLA to clone instances of this metamodel.

Such approach can be used to create reusable transformation libraries. Model transformation reuse has been considered an important topic [16]. One of the obstacles is the complete dependency of transformation definition on the used metamodel. Generic transformations (transformation generators) in Template MOLA could be used to create a reusable library of common metamodel independent algorithms for model processing.

This approach is less important if the transformation language contains features for work with several meta-levels at a time. However, it is useful for transformation languages like MOLA (and most of others that include the OMG standard MOF QVT [17]), which have no support for work with different meta-levels.

Generic Template MOLA procedures can be combined with traditional MOLA. The analogy with C++ templates and Java generics is used here. For example, it is also possible to write such a template based cloning procedure in C++:

```
template <class T> void Clone (T orig, T& copy) {...}.
```

In C++ this template procedure can be called with parameters of a concrete type. To process this template procedure, the preprocessor generates an instance of this procedure for every type it is called with. The same idea is used to combine MOLA with Template MOLA. This feature is required if we want to invoke reusable transformations from a transformation library.

Calls to template procedures can be used in ordinary MOLA transformations. In Fig. 12 calls to the template procedure *Clone* are demonstrated. The same preprocessor technology is kept when combining MOLA with Template MOLA as in C++ when generating procedures for each type they are called with.

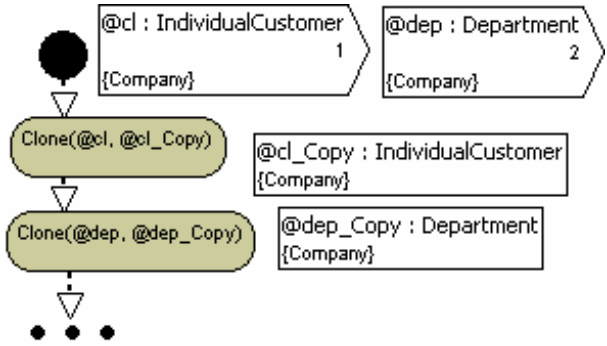


Fig. 12. An example of combining traditional MOLA with Template MOLA. A MOLA procedure calling a template procedure *Clone* from Fig. 13 is shown

Since several MOLA procedures should be generated from one template procedure, the procedure names should be generated too (several procedures with the same name are not allowed in MOLA). For a template procedure, it is possible to define an expression of how procedure name should be generated exactly, but default naming conventions are also provided. One of the preprocessor tasks in combining MOLA and Template MOLA is to replace calls to template procedure with calls to appropriate generated procedures.

Fig. 13 demonstrates the content of the template procedure *Clone*. It contains two template parameters. It means that two parameters will be created in the generated procedure. Instead of type, these parameters contain the template expression *<%@ type:Class%>*. This template expression is evaluated in generation time and replaced with the appropriate values in generated procedures. This procedure contains one more kind of parameter – a *type parameter* (parameter *@type*). This parameter has an analogy to C++ code, where a template parameter *T* was explicitly defined in procedure definition. In the same way as in C++, the value of the parameter is not defined in call but it is inferred from other parameters. Note that this type of parameter is used for this type of transformations only (transformations for generic metamodels) and is not required for typical HOT use cases. Since this template procedure is invoked from ordinary MOLA, the referenced metamodel must be MOLA MOF itself (the *Kernel* package).

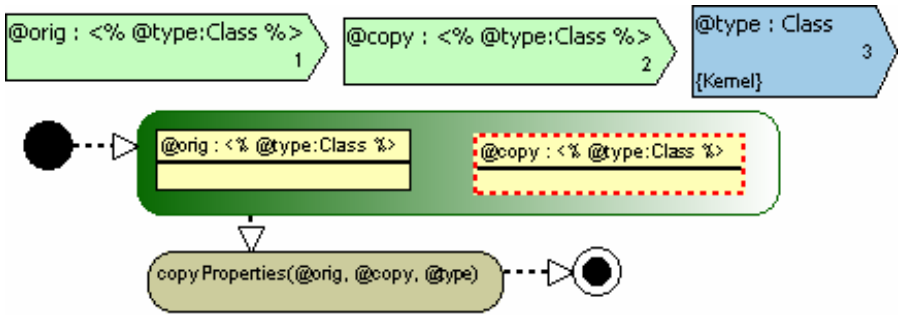


Fig. 13. The *Clone* procedure

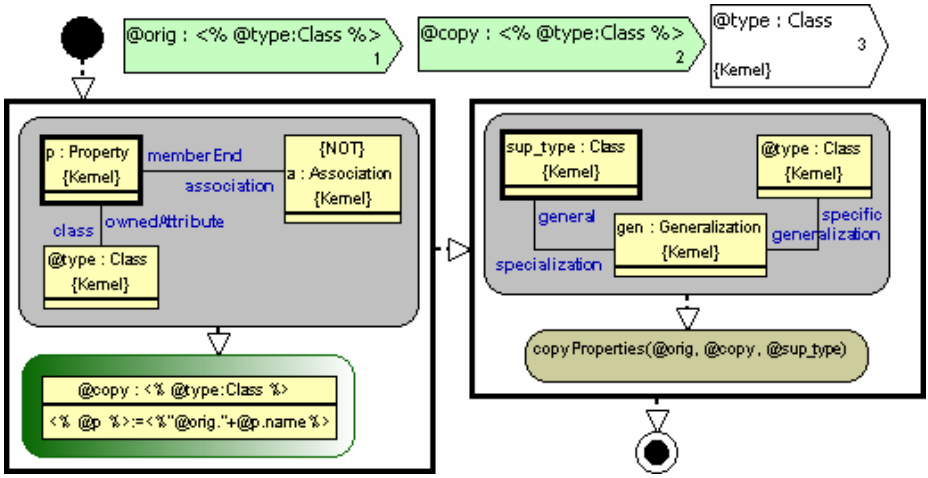


Fig. 14. The *copyProperties* procedure

In the *Clone* procedure one rule and one call is generated. In the rule, the template expressions (which specify types of class elements) are replaced with their generation time values in the same way as in template parameters. The call statement contains one type parameter and two template parameters. The template parameters are kept in the generated call. Actually, instead of a call to the template procedure, a call to the appropriate instance of procedure generated from template procedure is created (taking into account the name generation).

The template procedure in Fig. 14 generates the procedure to copy instance properties. It contains two template parameters and one generation time parameter. The generated procedure will have two parameters created from template parameters. Generation time parameters are only used in generation time.

The *copyProperties* procedure contains two generation time loops. The first loop (on the left in Fig. 14) iterates through all direct attributes of the class. For each attribute, it generates a rule containing a class element with assignment in it. The value of the same attribute in the instance *orig* is assigned to this attribute. In the generated class element, all template expressions are replaced with their values. Template expressions are used for the class element type, for the attribute to be assigned and for the assigned expression. Here is a remark on template expression syntax: the left hand side of the assignment must be an attribute reference in MOLA. Formally, both the notation *@p* (the reference to the attribute) and *@p.name* (a string expression equal to the attribute name) could be used here. Our choice is *@p* since it expresses more directly that the left hand side is a reference (it is preferred for implementation as well).

The second loop (on the right in Fig. 14) iterates trough all immediate superclasses of this class. For each superclass, it generates a call to a procedure that copies direct attributes of this superclass. In this way, using recursion in Template MOLA, values of all attributes are finally copied. It should be noted that the generated MOLA procedures are not recursive due to the fact that procedure names are generated when several MOLA procedures are created from one template procedure. Fig. 16 and 17 explain this situation in an example.

Now let us consider MOLA procedures generated from the Clone algorithm described above using Template MOLA. We will demonstrate the generated result for the first call of the *Clone* procedure in Fig. 12. The type of the instance to be cloned is *Company::IndividualCustomer*. The metamodel for this fragment is described in Fig. 15 (the package containing the fragment is assumed to be *Company*). This could be a simplified metamodel describing the information processed by a company. Fig. 16 presents the code generated form the template procedure *Clone*. The type parameter value is the type of the instance the call statement was invoked with. In this case, it is the class *Company::IndividualCustomer*. In the generated code, the type parameter *@type* is replaced with this class. The procedure call is replaced with a call to the generated procedure with appropriate types. Note that procedure names are generated in Template MOLA as well (according to default name generation rules, which can be modified if required). The procedure name here will be appended by the class name from the type parameter. The procedure name generation is necessary because the generated procedure code depends on the type (or generation) parameter value (as shown in Fig. 17). The type parameter itself is not included in the generated code.

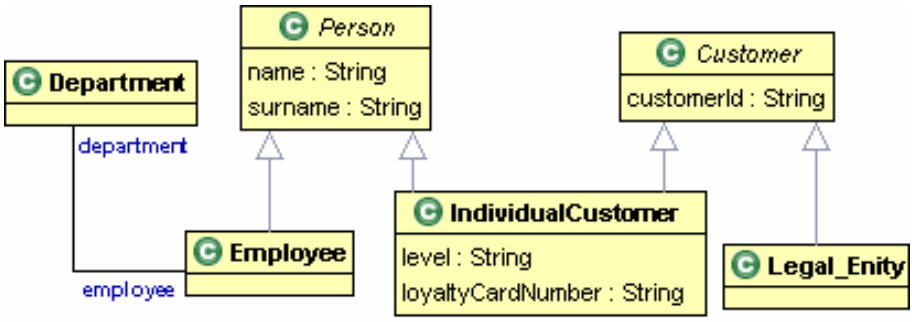


Fig. 15. A metamodel example describing information processed by a company. The class *IndividualCustomer* is used to describe the generated code in Fig. 16 and 17

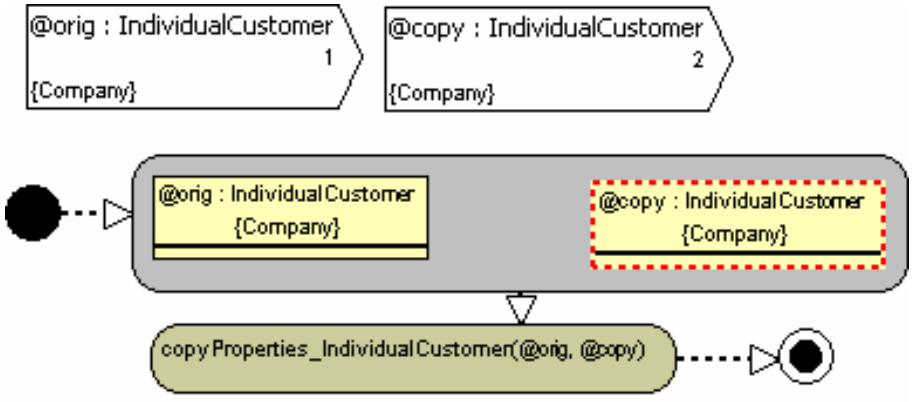


Fig. 16. A MOLA procedure generated from the template procedure *Clone*

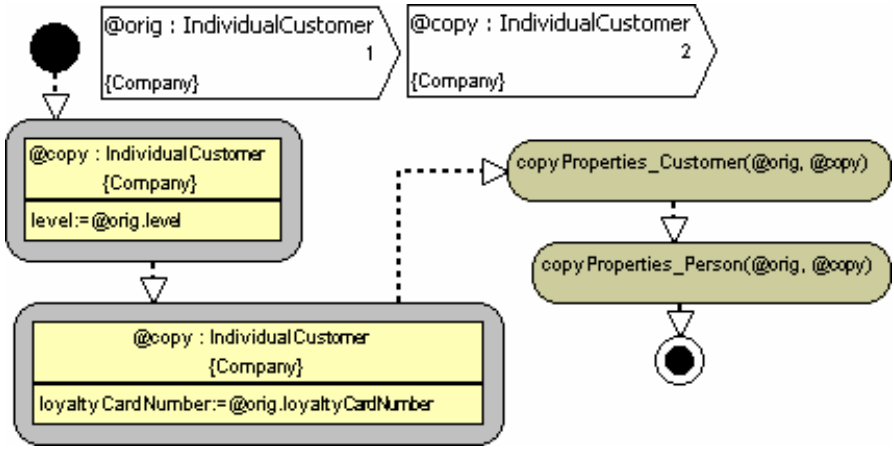


Fig. 17. A MOLA procedure generated from the template procedure *copyProperties*

Fig. 17 presents the structure of a MOLA procedure generated from the *copyProperties* procedure in Fig. 14 when the class specified by the generation time parameter is *Company::IndividualCustomer* (i.e., it is the procedure *copyProperties_IndividualCustomer*). The left side shows two of the generated rules for assigning direct attribute values of the *IndividualCustomer* class (to attributes *level* and *loyaltyCardNumber*). Attribute assignments are followed by calls to copy procedures generated for the superclasses of *IndividualCustomer* (calls for superclasses *Person* and *Customer* are shown). Note that the generated names of the procedures include the class name from the type parameter: thus, there is no recursion in the generated code.

In this example the generated MOLA source is a kind of *spaghetti code*. However, it would be sufficient to have one class element containing assignments for each property. Yet in the current version of Template MOLA, there are no facilities for creation of a variable number of assignments in one class element. This is an open avenue for further research.

5 Implementation Principles

To implement Template MOLA, we have to consider two aspects – editing and processing of Template MOLA.

The Template MOLA Editor was built in a METAclipse framework using the MOLA Editor as a basis. Model transformations which implement the traditional MOLA language within a METAclipse framework have been extended to support the desired functionality in the new editor. Since Template MOLA reuses the syntax from the traditional MOLA language, many of the MOLA procedures implementing the editing actions can be reused. The template elements can be regarded as subclasses of their related “regular” elements, thus inheriting all their required editing behaviour. A template text statement, for example, is almost equivalent to the traditional text statement from the editor’s point of view. New and unique functionality can be easily included where appropriate. So even though a substantial number of new diagram

elements have been introduced, the volume of the code has not grown proportionally, but much less than that. In addition, the subclassing approach eliminates any need for non-trivial migration when converting pure MOLA transformation models to Template MOLA transformation models.

Another aspect is the execution of Template MOLA. Several solutions were considered, including an interpreter and a Template MOLA preprocessor. The chosen solution was to use the preprocessor that converts Template MOLA to traditional MOLA with later reuse of the MOLA compiler to obtain transformations for generation. This approach is similar to preprocessing of macros in C++ environments. The preprocessor replaces Template MOLA statements with traditional MOLA rules which create corresponding instances of MOLA statements. For example, the template rule in Fig. 2 is replaced with the MOLA rule in Fig. 4. The newly-created MOLA transformation is compiled using the compiler of the traditional MOLA language. Finally, the obtained transformation is used as a HOT. Evaluation has shown that the preprocessor solution requires less effort to be implemented.

Another issue for consideration is the readability of MOLA sources generated using Template MOLA. The easiest solution is to create transformations using only the abstract syntax of MOLA. Abstract syntax is enough if we want to execute these transformations without manual extension. However, to obtain concrete graphical syntax for generated transformations, an abstract-to-concrete syntax transformation and an automatic diagram layout generator must be used. Note that transformations in Template MOLA actually contain some layout information for MOLA procedures to be generated. For example, the layout of elements in a template rule could be reused in the generated transformation. However, this issue requires further research.

6 Related Work

The necessity to use Higher-Order Transformations (HOTs) to support many MDD-related tasks was already discussed in the introduction. A comprehensive survey of HOT applications is presented in [3]. Although [3] shows that the classical HOT approach to synthesis of transformations is applicable in practice, it is not always the best solution. Sub-section 2.3 demonstrates how complicated it is to describe creation of a simple MOLA rule directly in MOLA. Creation of transformations in ATL [1] using ATL as a HOT discussed in [3] frequently is similarly difficult. In transformation languages such as Viatra [18], where the metamodelling facilities support simultaneous work at various meta-levels, the usage of HOTs is not required for work with generic metamodels. However, they do not solve transformation synthesis from mappings. In most of other transformation languages, transformation synthesis is even more important.

Therefore, a graphical template language-based solution for transformation synthesis was proposed in this paper. To a great extent, this solution has been inspired by textual template-based model-to-text languages – [5, 6, 7, 8] and many others.

The idea of using a graphical template language for transformation synthesis is new, as far as we know. The comprehensive survey in [19] of various features used for model transformation definition briefly mentions the template-based approach for model-to-model transformations as well. However, the only reference in [20] mentioned

as related to this approach is of templates applied for a very specific task of how to select prefabricated fragments of a target model on the basis of the existence of appropriate elements in the source model.

One more recent approach in transformation development somewhat similar to the approach described in this paper is the use of concrete graphical syntax to define a graph transformation [21, 22]. A graph transformation is defined from the graphical representation of the source model to the graphical representation of the target model. However, the approach is limited and there is no clear application of these ideas to the HOT-related tasks discussed in this paper.

7 Conclusions

A new graphical template based language Template MOLA for MOLA transformation synthesis is proposed in this paper. This language leverages the advantage of template-based model-to-text languages – easy specification of language elements to be generated – on to graphical languages. The graphical template statements of Template MOLA – template rules and template loops – are transferred to the new transformation to be generated. They can contain variable elements – template expressions replaced in the generation process. The generation process itself, which depends on the input model, is defined by means of generation statements – ordinary MOLA statements included in Template MOLA. These generation statements are executed in a standard way during the generation process.

It is shown that it is much easier to specify a transformation synthesis task in Template MOLA than to specify the same task in a traditional HOT style (using MOLA as a HOT).

Several application areas for Template MOLA arise, firstly, metamodel-based tool building for graphical DSLs. More precisely, it is the generation of transformations that determine the tool behaviour according to mappings that define the tool functionality in a static way (as, for example, in GMF). Some research on that has already begun. This paper also provides a small example.

A related application could be generation of transformations from a more general kind of mappings between models. This is the area where HOTs are widely used, especially in ATL.

Another important application is the building of transformations for unknown metamodels. This way, reusable transformation libraries for performing typical model processing tasks could be created. Then transformations from such libraries could be used in ordinary MOLA transformations for a specific metamodel. A very simple example from this area is also provided in this paper.

A future research direction could be to extend Template MOLA for defining templates in other graphical languages, for example, UML activity diagrams. The corresponding template statements then would be defined by the graphical syntax of the generated language. Generation statements controlling the generation process certainly would remain in MOLA. For example, various process generators could be built. This requires more research because implementation could turn out to be more complicated than for Template MOLA.

Acknowledgments. The authors would like to thank Oskars Vilitis for assistance in Template MOLA editor development and Karlis Cerans for valuable comments.

References

1. F. Jouault, I. Kurtev. *Transforming Models with ATL*. Satellite events at the MoDELS 2005 Conference. 2006, pp. 128–138.
2. M. Didonet Del Fabro, J. Bezivin, F. Jouault, E. Breton, G. Gueltas. AMW: a generic model weaver. *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles*, 2005.
3. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bezivin. On the use of higher-order model transformations. *ECMDA-FA 2009*. LNCS, Vol. 5562, Springer-Verlag, 2009, pp. 18–33.
4. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. *Proceedings of MDAFA 2004*, Springer LNCS, Vol. 3599, 2005, pp. 62–76.
5. Eclipse, JET. Available: http://www.eclipse.org/modeling/m2t/?project=jet.
6. OMG, MOF Model to Text Transformation Language, v1.0. OMG Document Number: formal/2008-01-16. Available: http://www.omg.org/docs/formal/08-01-16.pdf.
7. Eclipse, Xpand. Available: http://www.eclipse.org/modeling/m2t/?project=xpand.
8. L. M. Rose, R. F. Paige, D. S. Kolovos, F.A.C. Polack. The Epsilon Generation Language. *Proceedings of ECMDA-FA 2008*. Berlin, Germany, 2008.
9. UL IMCS, MOLA pages. Available: http://mola.mii.lu.lv/.
10. A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, J. Barzdins. Building Tools by Model Transformations in Eclipse. *Proceedings of DSM'07 Workshop of OOPSLA 2007*, Montreal, Canada: Jyvaskyla University Printing House, 2007, pp. 194–207.
11. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. *Proceedings of Workshop on Domain-Specific Program Development (DSPD), ECOOP 2006*. France, 2006.
12. Eclipse, Graphical Modeling Framework (GMF). Available: http://www.eclipse.org/modeling/gmf.
13. S. Cook, G. Jones, S. Kent, A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
14. J. Barzdins, A. Zarins, K. Cerans et al. GrTP: Transformation-Based Graphical Tool Building Platform. *Proceedings of Workshop on MDDAUI, MODELS 2007*. Nashville, USA, 2007.
15. E. Kalnina, A. Kalnins. DSL tool development with transformations and static mappings. In: M. R. V. Chaudron (ed.), *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France. Reports and Revised Selected Papers.* LNCS, Programming and Software Engineering, Vol. 5421, 2009, pp. 356–370.
16. J. S. Cuadrado, J. G. Molina. Approaches for Model Transformation Reuse: Factorization and Composition. *Proceedings of ICMT 2008*. LNCS, Vol. 5063. Zürich, Switzerland, 2008, pp. 168–182.
17. MOF QVT Final Adopted Specification, OMG, Document Number: ptc/08-04-03, 2008.
18. Visual Automated Model Transformations (VIATRA2), GMT subproject. Budapest University of Technology and Economics. Available: http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html.
19. K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. IBM *Systems Journal*, v. 45 no. 3, July 2006, pp. 621–645.
20. K. Czarnecki, M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Tallinn, Estonia, 2005, pp. 422–437.
21. R. Grønmo, B. Møller-Pedersen, G. K. Olsen. Comparison of Three Model Transformation Languages. *ECMDA-FA 2009*. LNCS, Vol. 5562, 2009. Springer-Verlag, pp. 2–17.
22. J. de Lara, H. Vangheluwe. AToM: a Tool for Multi-formalism and Meta-modelling. *FASE 2002*. LNCS, Vol. 2306, 2002. Springer-Verlag, pp. 174–188.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 99–117 P.

Mapping between Relational Databases and OWL Ontologies: an Example

Guntars Bumans
Department of Computing, University of Latvia
Raina bulv. 19, Riga, LV-1586, Latvia
guntars.bumans@gmail.com

This paper shows how relational databases can be used to define a bridging mechanism between relational database and OWL ontology. We demonstrate on a simple yet completely elaborated example how mapping information stored in relational tables can be processed using SQL to generate RDF triples for OWL class and property instances. This technology provides the means to use relational database as a powerful tool to transform relational data to Semantic Web layer.

Keywords: relational databases, RDF triples, mappings.

1 Introduction

There are several studies or tools allowing mapping relational databases (RDBs) to RDF schema or OWL ontologies. Some of the most notable approaches of this kind are R2O [1], D2RQ [2], Virtuoso RDF Views [3, 4] and DartGrid [5]. There is W3C RDB2RDF Incubator Group [6] related to standardization of RDB to RDF mappings, the group has published its survey of mapping RDBs to RDF [7].

R2O [1] approach defines declarative and extensible language (in xml) to describe mapping between given RDB and an OWL ontology or RDFS schema so that tools can process this mapping and generate triples that correspond to source RDB data. D2RQ [2] technology is another bridging technology where one can use SQL to describe the mapping information. This language is closer to SQL level and is not as declarative as R2O. Both D2RQ [2] and Virtuoso RDF Views [3, 4] allow retrieving instance data from RDB on-the-fly during the execution of SPARQL queries over the RDF data store.

The aim of this paper is to demonstrate a very simple standard SQL-based RDB to RDF/OWL mapping approach that is based on defining correspondence between the tables of the database and the classes of the ontology, as well as between table fields/links in the database and datatype/object properties in the ontology (with possible addition of filters and linked tables in the mapping definition), and later automatically generating SQL statements that generate the RDF triples that correspond to the source database data.

Our work setting for RDB to RDF/OWL translation involves the assumption that both the database and the ontology (or RDF schema) are given. The translation is not meant to be on-the-fly because huge amount of data can be involved. This corresponds to the practical database semantic re-engineering task, as advocated in [8, 9, 10, 11] in the setting of Latvian medical research databases.

It should be possible to translate the mappings specified here into other RDB-to-RDF/OWL mapping formalisms (e.g., R2O [1], D2RQ [2], Virtuoso RDF Views [3]), thus obtaining alternative implementations of these mappings.

There is at least a conceptual possibility to create the mapping between the source RDB schema and target OWL ontology by means of model transformations described in some transformation language (e.g. MOF QVT [12], ATL [13], or MOLA [14]); however, typically these translations are not supported on data in RDB or RDF formats and require the use of an intermediate format (a so-called model repository, such as EMF [15]) which may not be feasible for large data sets.

Our approach is to go for direct translation of RDB-stored data into (conceptual) OWL ontology format that can be executed on the level of DBMS.

In Section 2 of this paper we describe the mapping method and provide the table structure for storing mapping data. Section 3 introduces the demonstration example. Section 4 describes and demonstrates the instance generation process for OWL classes, OWL datatype properties and OWL object properties. Section 5 concludes the paper.

2 A Mapping Schema

We propose a bridging mechanism between relational databases and OWL ontologies. We assume that the ontology and the database have been developed separately. Most often the database is of legacy type but the ontology reflects the semantic concerns regarding the data contents. Our approach is to make a mapping between these structures and store the mapping in meta-level relational schema (we are working towards mapping specification language that is suitable for the end user, which however is beyond the scope of this paper). This approach allows us to use relational database engine to process mapping information and generate SQL sentences that, when executed, will create RDF/OWL-formatted data (RDF triples) describing instances of OWL classes and OWL datatype and OWL object properties that correspond to the source RDB data.

In the simplest case, an OWL class corresponds to a RDB table, an OWL datatype property corresponds to a table field, and an OWL object property corresponds to a foreign key. In real life examples the mappings are not so straightforward.

For example, an OWL class *Person* could be a domain for OWL datatype property *personAddress*. But the corresponding database table *persons* could have a foreign key reference to some other table having address information. To complicate things even more, one property of type *xsd:string* can correspond to a combination of columns spread over many tables in the database (e.g., country, city, street information stored in separate tables). Other possible causes of direct mapping impossibility are subclass relation in the ontology, the use of many to many relations, the non-existence of “natural” foreign keys in RDB. Often databases are normalized and their structure is optimized out of performance concerns thus hiding true conceptual meaning. To deal with all this complexity, we introduce mapping schema (Fig. 1). We will call it *mapping DB*. Source database (legacy type) will be denoted by *source DB* in this paper.

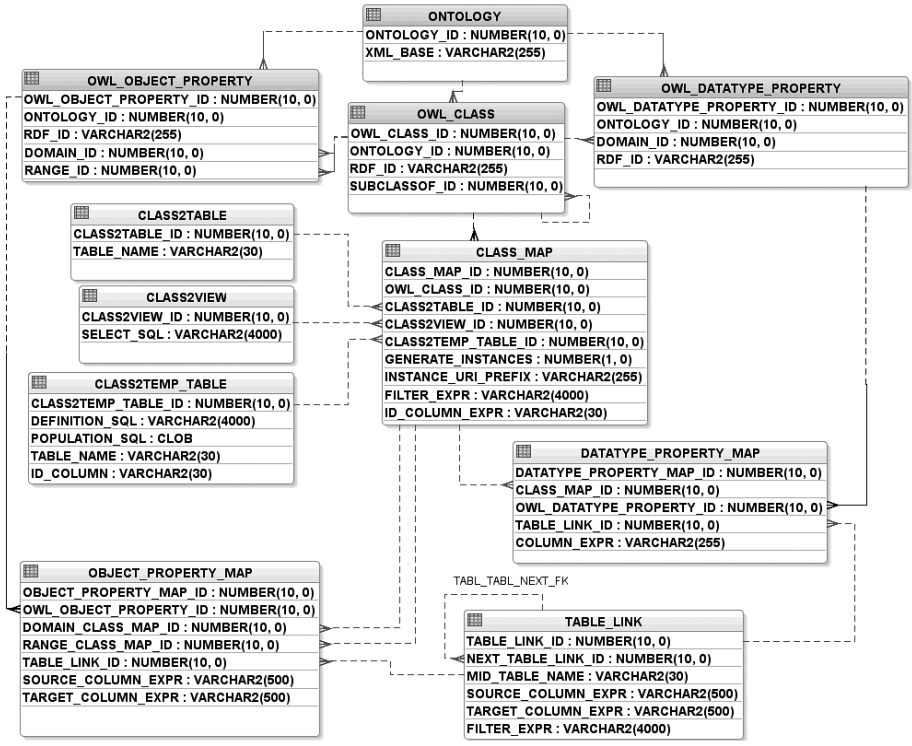


Fig. 1. A mapping schema between OWL ontology and relational database

The OWL ontology basic information (classes and properties) is stored in tables *ontology*, *owl_class*, *owl_object_property* and *owl_datatype_property*. For each OWL class the table *class_map* defines the connections to RDB objects that can be a table (a real, existing table in a source database), a view (a view can be defined inline using column *class2view.select_sql*), or a temporary table (*class2temp_table*). In typical cases only real existing source database tables are referenced: however, the view and temporary table techniques are available to handle certain advanced mapping situations. For example, the temporary table mechanism allows to create a new table in triple generation process (*definition_sql* column) and fill it with data (*populate_sql).* This mechanism can be used both for complex data-specific mappings (e.g. splitting a comma-separated list into independent element data values), and for providing an auto-generated identity column as a resource for OWL class instance URI generation. To simplify the further description, we will assume that mapping is to real database tables *(class2table)*. This way we do not lose the functionality that is required in the forthcoming example that does not use either inline defined views or temporary tables. From the perspective of transformation process, mapping specified in *class2table* can be also directed to named view in the database. Both tables and named views have columns and no other table specific information is used.

Each row in the *class_map* table contains a further filter possibility on the referenced RDB object (the *filter_expr* column), as well as the specification for

target OWL class instance URI generation on the basis of the data contained in the referenced RDB object. The URI of an OWL class instance is defined based on a record in the *class_map* table, by concatenating the *ontology.xml_base* value with *instance_uri_prefix* column value (typically holding the name of the referenced RDB object), followed by the contents of *id_column_expr* (column or column expression) evaluated in the referenced RDB object in *source DB*. The *generate_instances* column specifies whether the OWL class instances indeed have to be generated by this *class_map* specification. If the instances are not to be generated, the *class map* can still be used as a reference point in further *object_property_map* and *datatype_property_map* definitions.

Table *object_property_map* holds specifications for instance generations for OWL object properties. Each record in this table is based on a *class_map* record for both domain and range of the object property, as it describes or references the rules how the triples connecting the domain and range instances by this object property are formed. In the simplest case, a column expression is specified for both DB objects (tables) corresponding to the domain and range class maps (in *source_column_expr* and *target_column_expr*, respectively), and the values of these expressions are required to coincide for a triple to be created. If DB objects (tables) for domain and range class maps cannot be joined directly based on column expressions but they can be joined through some intermediate table joins then these middle joins are specified by rows in *table_link* table (more on this in Sub-section 4.3).

The table *datatype_property_map* holds specifications for instance generation for OWL datatype properties. Each record in this table is based on a *class_map* record for domain of the datatype property. The record maps domain to DB table (or view) and specifies how to generate subject part of generated triples: filtering, URI creation is done the same way as for OWL instance generation. Column *column_expr* specifies how to generate object part of triples (value for range). In the simplest case it is evaluated in the table specified for domain class map. If the value is to be taken from some other table then *table_link* row can be used to specify join to that table. Example: *Person* table is domain table but range for the property is *address* that is stored in *Address* table to which *Person* has foreign key.

A *mapping DB* can hold information of more than one OWL ontology to database mapping. For each such mapping there should be a separate row in *ontology* table and foreign key to it from *owl_class*, *owl_datatype_property* and *owl_object_property* table.

3 A Mapping Example

To better explain our proposed approach, we will use a simple example taken from [8] in Fig. 2. and 3. Below are a sample database schema and a corresponding ontology (OWL class *Thing* is omitted for simplicity).

For example, the classes *Student* and *Course* in this sample ontology have corresponding tables *student* and *course* in the sample database. To get instance data for OWL object property *takes* the table link path is needed: tables *student* and *registration* joined on *student_id* and *registration* and *course* joined on *course_id*.

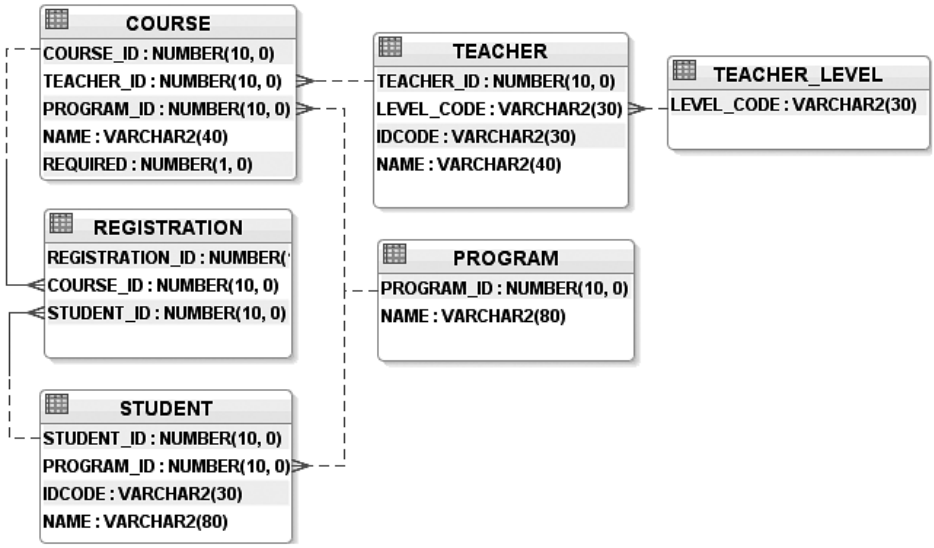


Fig. 2. A sample relational database schema

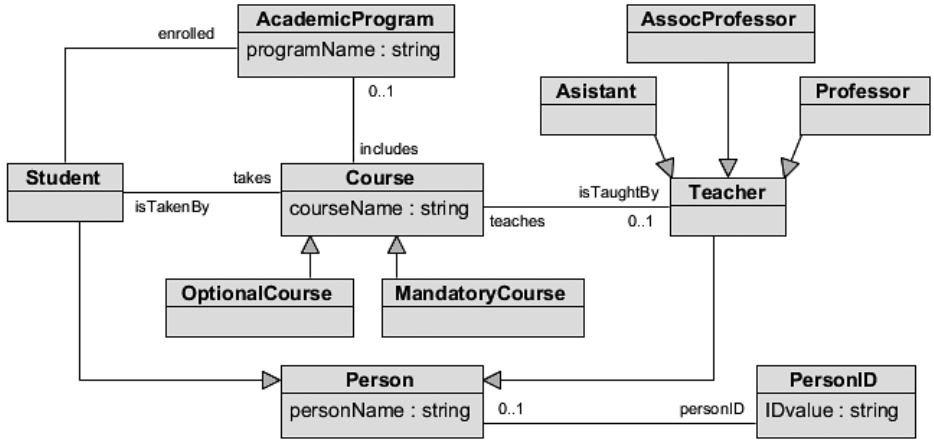


Fig. 3. Sample ontology

Class *personID* instances are populated from s*tudent* and *teacher* tables (*idcode* column).

Classes *Asistant*, *AssocProfessor* and *Professor* all get instance data from one table *teacher* but each has a different filtering. It means that string data 'level_code=…' must be written in *class_map.filter_expr* in corresponding rows.

In the following tables we show the actual data tables of our sample database. This specific data set will be used as an example.

Table 1

Table *program* data

program_id	name
1	Computer Science
2	Computer Engeneering

Table 2

Table *teacher_level* data

level_code
Assistant
Associate Professor
Professor

Table 3

Table *course* data

course_id	name	program_id	teacher_id	Required
1	Programming Basics	2	3	0
2	Semantic Web	1	1	1
3	Computer Networks	2	2	1
4	Quantum Computations	1	2	0

Table 4

Table *student* data

student_id	name	idcode	program_id
1	Dave	123456789	1
2	Eve	987654321	2
3	Charlie	555555555	1
4	Ivan	345453432	2

Table 5

Table *teacher* data

teacher_id	name	idcode	level_code
1	Alice	999999999	Professor
2	Bob	777777777	Professor
3	Charlie	555555555	Assistant

Table 6

Table *registration* data

registration_id	student_id	course_id
1	1	2
2	1	4
3	2	1
4	2	3
5	3	2

Mapping data between these two models inserted in our *mapping DB* is shown in Sub-sections 4.1–4.3 in appropriate places when describing instance generation methods. For our mapping example we have *owl_ontology.ontology_id=1*.

The basic information from OWL ontology (classes, datatype and object properties) is encoded in tables *ontology, owl_class, owl_datatype_property* and *owl_object_property* in an obvious way.

4 OWL Instance Generation

In this section, we describe the instance generation process. it is done by SQL select statements executed in *mapping DB* to generate another SQL select statement that in turn generates RDF triples when executed in *source DB.* In special cases when *table_link* is to be used in generation process more than once procedural language (java + jdbc) can be applied. However, that is not the case in the example discussed in this paper.

4.1 OWL Class Instance Generation

Link between tables *owl_class* and *class_map* is used to generate RDF triples for OWL class instances. Unique URI for these instances are formed concatenating fields *ontology.xl_base, owl_class.rdf_id* and value derived from evaluating the expression described in *id_column_expr* in *source DB.* To generate triples for OWL class instances that are based on real tables (having foreign key from *class_map* to *class2table*) we need to create SQL select statement based on tables *ontology, owl_class, class_map* and *class2table.* If source data comes from database view or temporary table then query needs to be modified (*class2view/class2temp_table* instead of *class2table*).

OWL class mappings are listed in the next table. In the example only the mappings to database tables are used. Data from tables *class_map* and referenced tables *owl_class, class2table* is listed below.

Table 7

OWL class mappings to database tables

class_map_id	OWL class (rdf_id)	table_name	filter_expr	id_column_expr	instance_uri_prefix	generate_instances
1	Teacher	teacher		teacher_id	Teacher	0
2	Assistant	teacher	level_code= 'Assistant'	teacher_id	Teacher	1
3	AssocProfessor	teacher	level_code= 'Associate Professor'	teacher_id	Teacher	1
4	Professor	teacher	level_code= 'Professor'	teacher_id	Teacher	1
5	Student	student		student_id	Student	1
6	Course	course		course_id	Course	0
7	MandatoryCourse	course	required=1	course_id	Course	1
8	OptionalCourse	course	required=0	course_id	Course	1
9	PersonID	teacher		idcode	PersonID	1
10	PersonID	student		idcode	PersonID	1
11	AcademicProgram	program		program_id	Program	1

Most of the class mappings are used for the real OWL class instance generation. There are, however, a few class mappings that are not used in the class instance generation, but which will be further referenced in datatype property mappings.

With an SQL statement it is possible to generate another SQL statement which, when executed in *sample DB*, would generate instance RDF triples. Executing script *OWL_instance_gen.sql* (see Appendix for code) against our sample data, we obtain row set with generated SQL statements, one of which is:

```
  SELECT '<http://lumii.lv/ex#Course'
 || course.course_id || '>' as subject,
 '<type>' as predicate,
 '<lumii#MandatoryCourse>' as object
FROM course
WHERE required=1
```

Executing all generated statements in our sample *source DB* we obtain the following triples, with duplicates removed. The duplicates in the example come from the fact that one *teacher* table row and one *student* table row have the same *idcode* value (the same person being student and teacher at the same time). In Table 8 we use the prefix "lumii" to denote "http://lumii.lv/ex", and the predicate notation "type" to stand for http://www.w3.org/1999/02/22-rdf-syntax-ns#type.

Table 8

Generated OWL class instance RDF triples

Subject	Predicate	Object
< lumii #Course1>	<type>	<lumii#OptionalCourse>
< lumii #Course2>	<type>	<lumii#MandatoryCourse>
< lumii #Course3>	<type>	<lumii#MandatoryCourse>
<lumii#Course4>	<type>	<lumii#OptionalCourse>
<lumii#PersonID123456789>	<type>	<lumii#PersonID>
<lumii#PersonID345453432>	<type>	<lumii#PersonID>
<lumii#PersonID555555555>	<type>	<lumii#PersonID>
<lumii#PersonID777777777>	<type>	<lumii#PersonID>
<lumii#PersonID987654321>	<type>	<lumii#PersonID>
<lumii#PersonID999999999>	<type>	<lumii#PersonID>
<lumii#Program1>	<type>	<lumii#AcademicProgram>
<lumii#Program2>	<type>	<lumii#AcademicProgram>
<lumii#Student1>	<type>	<lumii#Student>
<lumii#Student2>	<type>	<lumii#Student>
<lumii#Student3>	<type>	<lumii#Student>
<lumii#Student4>	<type>	<lumii#Student>
<lumii#Teacher1>	<type>	<lumii#Professor>
<lumii#Teacher2>	<type>	<lumii#Professor>
<lumii#Teacher3>	<type>	<lumii#Assistant>

4.2 OWL Datatype Property Instance Generation

Table *datatype_property_map* allows to specify for each *owl_datatype_propery* several possible value generation mappings, each based on some *class_map*. This linking allows to obtain domain instance URIs for an OWL datatype property. The property range values are obtained from table columns or expressions thereof. In the simplest case when range is to be mapped to column in the same table specified through *datatype_property_map*→*class_map*, we use column *column_expr.* If the property range is mapped to table column (or column expression) in a linked table, we specify the link expression in table *table_link* (the *source_column_expr, target_column_expr* and *mid_table_name* columns encode the linking (table join) conditions). There is no *table_link* usage for OWL datatype property generation in the example.

Table 9 represents data in table *datatype_property_map* and referenced tables *class_map* and *owl_datatype_property* in case when no table link is used (no *table_link* table usage). One can compare the first column in Table 7 and Table 9 below. For example, property *personName* is linked to *class_map_id*=1 and *class_map_id*=5 that correspond to class maps for OWL classes *Teacher* and *Student*. Instances are not directly generated for *Teacher* class *(generate_instances=0)*. Class instances are generated for subclasses *Professor, AsocProfessor* and *Asistant* classes. As *instance_uri_prefix, table_name* and *id_column_expr* have the same value in the class map for superclass (*Teacher* in this case), it enables correct generation of the subject part of triples for OWL datatype properties. There is no need to make class map for each subclass. As to correctness of the mapping, the class map to super class should have the same filtering as union of all subclasses. In the case of *Teacher* it has no filter (*filter_expr* is empty for *class_map_id*=1) but filters for sub-class maps *(class_map_id:2,3,4)* are *level_code='Assistant', level_code='Associate Professor'* and *level_code='Professor'*. All these together produce all *teacher* rows and *Teacher* class map with no filtering corresponding to the same row set.

Table 9

OWL datatype property class mappings to database table column expressions (data from tables *datatype_property_map* and referenced *class_map, class2table* and *owl_datatype_property*)

class_map_id	OWL_datatype_property	table_name	column_expr	filter_expr
6	courseName	Course	name	
11	programName	Program	name	
1	personName	Teacher	name	
5	personName	Student	name	
9	IDValue	Teacher	idcode	
10	IDValue	Student	idcode	

Executing script *generate_sql4datatype_props.sql* (see Appendix for the code) against our sample data, we obtain row set with generated SQL statements, one of which is:

```
  SELECT
 '<lumii#optionalCourse'
 || course.course_id || '>' as subject,
 '<lumii#courseName>' as predicate,
 name as object
FROM course
WHERE required=0
```

Executing all generated statements in our sample *source DB*, we obtain the following triples, duplicates removed (note the abbreviations, as in Table 8).

Table 10

Generated OWL datatype property instance RDF triples

Subject	Predicate	Object
<lumii#Course1>	<lumii#courseName>	Programming Basics
<lumii#Course2>	<lumii#courseName>	Semantic Web
<lumii#Course3>	<lumii#courseName>	Computer Networks
<lumii#Course4>	<lumii#courseName>	Quantum Computations
<lumii#PersonID123456789>	<lumii#IDValue>	123456789
<lumii#PersonID345453432>	<lumii#IDValue>	345453432
<lumii#PersonID555555555>	<lumii#IDValue>	555555555
<lumii#PersonID777777777>	<lumii#IDValue>	777777777
<lumii#PersonID987654321>	<lumii#IDValue>	987654321
<lumii#PersonID999999999>	<lumii#IDValue>	999999999
<lumii#Student1>	<lumii#personName>	Dave
<lumii#Student2>	<lumii#personName>	Eve
<lumii#Student3>	<lumii#personName>	Charlie
<lumii#Student4>	<lumii#personName>	Ivan
<lumii#Teacher1>	<lumii#personName>	Alice
<lumii#Teacher2>	<lumii#personName>	Bob
<lumii#Teacher3>	<lumii#personName>	Charlie
<lumii#Program1>	<lumii#programName>	Computer Science
<lumii#Program2>	<lumii#programName>	Computer Engeneering

4.3 OWL Object Property Instance Generation

Rows in *object_property_map* specify how to generate instances for OWL object properties. The references to *class_map* through foreign keys *domain_class_map* and *range_class_map* determine *source DB* tables for subject and object of generated triples for property instances. The *class_map* rows in column *filter_expr* determine filtering on these tables. These tables are joined by column expressions specified in *source_column_expr* and *target_column_expr.* They are joined directly or by using one or more intermediate table joining steps. The latter require usage of one or more rows in *table_link.*

First we shall discuss direct joining of domain table to range table without *table_link.* An OWL datatype property can have several mappings – several rows in *object_property_map.* In this case the triple generation will process all of them. In the process of

triple generation for OWL object properties (also for OWL datatype properties) the *class_map* column *generate_instances* is not used. This field is only for OWL class instance generation. URI for subject and object part of triple are determined by concatenation of *ontology.xml_base, class_map.instance_uri_prefix* and evaluation of *id_column_expr* in *source DB.* URI of predicate part of generated triples are determined by concatenation of *ontology.xml_base* and *owl_object_property.rdf_id*. Table 11 represents data from *object_property_map*, and referenced *owl_object_property*, as well as two *class_map* rows for subject and object and corresponding *class2table* rows. See Table 7 for more details on referenced *class_map* rows.

Table 11

Owl object property mappings to database tables pairs for domain and range

class_ map_id (domain)	class_ map_id (range)	object_ property	table_ name (domain)	table_ name (range)	source_ col_expr	target_ col_expr
11	6	includes	program	course	program_id	program_id
5	10	personID	student	student	student_id	student_id
1	9	personID	teacher	teacher	teacher_id	teacher_id
5	11	enrolled	student	program	program_id	program_id
1	6	teaches	teacher	course	teacher_id	teacher_id

Reading the data, we can see that OWL object properties generally map to table pairs corresponding to domain and range class pair. *PersonID* object property is an exception because it has *Person* class as domain and *PersonID* class as range and both these classes have mappings to 2 tables: *student* and *teacher.* For this property two mappings exist (*object_property_map* rows), one of which maps *student* table for domain to *student* table for range. The mapping is based on *student_id* column (*source_column_expr*, *target_column_expr*). The second row maps *teacher* table to *teacher* table based on *teacher_id* column in a similar way.

To generate RDF triples for OWL object property instances the data represented in Table 11 above can be used. A framework of SQL for main information retrieval for generation process is as follows.

```
  SELECT
 <domain_table>_1.<domain_class_map_id→class_map.id_column_expr>,
 <range_table>_2.<range_class_map_id→class_map.id_column_expr>
FROM <domain_table> AS <domain_table>_1
 INNER JOIN <range_table> AS <range_table>_2
 ON <domain_table>_1.<domain_column_expr>
 = <range_table>_2.<range_column_expr>
```

The suffixes _1 and _2 are added here to prevent name collision. For example, in the case of mapping for *PersonID* property (for *student*) query joins *student* table to itself because *object_property_map* table specifies two tables via *domain_class_map* and *range_class_map* although the tables are the same.

```
  SELECT student_1.student_id, student_1.program_id
FROM student AS student_1
INNER JOIN student AS student_2
ON student_1.student_id = student_2.student_id
```

For *enrolled* property the query is

```
  SELECT student_1.student_id, program_2.program_id
FROM student AS student_1
INNER JOIN program AS program_2
ON student_1.program_id = program_2.program_id
```

An SQL script for OWL object property instance generation can be defined in a similar way as it was done for OWL class and OWL datatype property instance generation.

Executing script *generate_sql4object_props.sql* (see Appendix for code) against our sample data, we produced row set with generated SQL statements, one of which was:

```
  SELECT
 '<lumii#Program' || program_1.program_id || '>' as subject,
 '<lumii#includes>' as predicate,
 '<lumii#Course' || course_2.course_id || '>' as object
FROM program program_1 INNER JOIN course course_2
 ON program_1.program_id = course_2.program_id
WHERE 1=1 AND 1=1
```

Executing all generated statements in our sample *source DB*, we produced the following triples (note the abbreviations, as in Table 8).

Table 12

Generated OWL object property instance RDF triples

Subject	Predicate	Object
<lumii#Student1>	<lumii#enrolled>	<lumii#Program1>
<lumii#Student2>	<lumii#enrolled>	<lumii#Program2>
<lumii#Student3>	<lumii#enrolled>	<lumii#Program1>
<lumii#Student4>	<lumii#enrolled>	<lumii#Program2>
<lumii#Program1>	<lumii#includes>	<lumii#Course4>
<lumii#Program1>	<lumii#includes>	<lumii#Course2>
<lumii#Program2>	<lumii#includes>	<lumii#Course1>
<lumii#Program2>	<lumii#includes>	<lumii#Course3>
<lumii#Student1>	<lumii#personID>	<lumii#PersonID123456789>
<lumii#Student2>	<lumii#personID>	<lumii#PersonID987654321>
<lumii#Student3>	<lumii#personID>	<lumii#PersonID555555555>
<lumii#Student4>	<lumii#personID>	<lumii#PersonID345453432>
<lumii#Teacher1>	<lumii#personID>	<lumii#PersonID999999999>
<lumii#Teacher2>	<lumii#personID>	<lumii#PersonID777777777>
<lumii#Teacher3>	<lumii#personID>	<lumii#PersonID555555555>
<lumii#Teacher1>	<lumii#teaches>	<lumii#Course2>
<lumii#Teacher2>	<lumii#teaches>	<lumii#Course3>
<lumii#Teacher2>	<lumii#teaches>	<lumii#Course4>
<lumii#Teacher3>	<lumii#teaches>	<lumii#Course1>

Now we shall discuss the table link usage. It is required for instance generation of OWL object property *takes* which is between *Student* and *Course* OWL classes and requires to join tables *student* and *course* through *registration*. Table *object_property_*

map links to *class_map* two rows for subject and object through *domain_class_map_id* and *range_class_map_id* foreign keys. That produces pair of two relations (tables). To join these tables *source_column_expr* and *target_column_expr* are used. If these tables, cannot be joined directly, then *table_link* table is to be used. It stores information about middle steps in table traversing. To support joining table *t1* with *t2* through middle table, the *table_link* columns has these meanings:

mid_table_name- table name in the middle step,

source_column_expr- joins <*mid_table_name*> table to *t1* by this column expr.,

target_column_expr- joins <*mid_table_name*> table to *t2* by this column expr.,

filter_expr- additional filter expression on table <*mid_table_name*>,

next_table_link_id- foreign key to the same table to implement more intermediate steps if needed (t1→mid_table_1→mid_table_2 → … →mid_table_n→t2).

Table 13 and Table 14 represent OWL object property mapping data for properties that need table links (*object_property_map.table_link* is not null). Data comes from tables *owl_object_property, object_property_map* as well as their referenced table rows. the corresponding *table_link* data follows. *Filter_expr* is not used in the example.

Table 13

Owl object property mappings to database tables pairs for domain and range when table link is used

class_ map_id (domain)	class_ map_id (range)	object_ property	table_ name (domain)	table_ name (range)	source_ column_ expr	target_ column_ expr
5	6	takes	student	Course	student_id	course_id

Table 14

***Table_link* table data**

mid_table_name	source_column_expr	target_column_expr	next_table_link_id
registration	student_id	course_id	

The join condition is:

```
  <domain_table>.<source_column_expr>=
<mid_table_name>.<table_link.source_column_expr>
AND
<mid_table_name>.<table_link.target_column_expr>=
<range_table>.<target_column_expr>
```

In this case the exact condition is:

```
  student.student_id=registration.student_id
AND
registration.course_id=course.course_id
```

Executing script *generate_sql4object_props_table_links.sql* (see Appendix for the code) against our sample data, we obtain row set with generated SQL statements, one of which is:

```
  SELECT
'<lumii#Student' || student_1.student_id || '>' as subject,
'<lumii#takes>' as predicate,
'<lumii#Course' || course_2.course_id || '>' as object
 FROM student student_1
INNER JOIN registration registration_3
 ON student_1.student_id = registration_3.student_id
INNER JOIN course course_2
 ON registration_3.course_id = course_2.course_id
WHERE 1=1 AND 1=1 AND 1=1 AND 1=1
```

Executing it in sample *source DB* we get the following triples.

Table 15

Generated OWL object property instance RDF triples when *table_link* table used

Subject	Predicate	Object
<lumii#Student1>	<lumii#takes>	<lumii#Course2>
<lumii#Student2>	<lumii#takes>	<lumii#Course4>
<lumii#Student3>	<lumii#takes>	<lumii#Course1>
<lumii#Student4>	<lumii#takes>	<lumii#Course3>
<lumii#Student5>	<lumii#takes>	<lumii#Course2>

4.4 The result of RDF Triple Generation

When all generated SQLs were executed in our example database, we produced the following triple set, essentially being data export from original relational database to RDF format for target OWL ontology. Following the data is a union of data in Table 8, 10, 12 and 15 with shorthands “lumii” and “type” expanded.

```
<http://lumii.lv/ex#Course1>  <http://lumii.lv/ex#courseName>  Programming Basics
<http://lumii.lv/ex#Course2>  <http://lumii.lv/ex#courseName>  Semantic Web
<http://lumii.lv/ex#Course3>  <http://lumii.lv/ex#courseName>  Computer Networks
<http://lumii.lv/ex#Course4>  <http://lumii.lv/ex#courseName>  Quantum Computations
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program1>
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program2>
<http://lumii.lv/ex#Student3> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program1>
<http://lumii.lv/ex#Student4> <http://lumii.lv/ex#enrolled> <http://lumii.lv/ex#Program2>
<http://lumii.lv/ex#PersonID123456789><http://lumii.lv/ex#IDValue> 123456789
<http://lumii.lv/ex#PersonID345453432><http://lumii.lv/ex#IDValue> 345453432
<http://lumii.lv/ex#PersonID555555555><http://lumii.lv/ex#IDValue> 555555555
<http://lumii.lv/ex#PersonID777777777><http://lumii.lv/ex#IDValue> 777777777
<http://lumii.lv/ex#PersonID987654321><http://lumii.lv/ex#IDValue> 987654321
<http://lumii.lv/ex#PersonID999999999><http://lumii.lv/ex#IDValue> 999999999
<http://lumii.lv/ex#Program1> <http://lumii.lv/ex#includes> <http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Program1> <http://lumii.lv/ex#includes> <http://lumii.lv/ex#Course4>
<http://lumii.lv/ex#Program2> <http://lumii.lv/ex#includes> <http://lumii.lv/ex#Course3>
<http://lumii.lv/ex#Program2> <http://lumii.lv/ex#includes> <http://lumii.lv/ex#Course1>
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID123456789>
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID987654321>
<http://lumii.lv/ex#Student3> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID555555555>
<http://lumii.lv/ex#Student4> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID345453432>
<http://lumii.lv/ex#Teacher1> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID999999999>
<http://lumii.lv/ex#Teacher2> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID777777777>
<http://lumii.lv/ex#Teacher3> <http://lumii.lv/ex#personID>  <http://lumii.lv/ex#PersonID555555555>
<http://lumii.lv/ex#Student1> <http://lumii.lv/ex#personName> Dave
<http://lumii.lv/ex#Student2> <http://lumii.lv/ex#personName> Eve
```

<http://lumii.lv/ex#Student3>	<http://lumii.lv/ex#personName>	Charlie
<http://lumii.lv/ex#Student4>	<http://lumii.lv/ex#personName>	Ivan
<http://lumii.lv/ex#Teacher1>	<http://lumii.lv/ex#personName>	Alice
<http://lumii.lv/ex#Teacher2>	<http://lumii.lv/ex#personName>	Bob
<http://lumii.lv/ex#Teacher3>	<http://lumii.lv/ex#personName>	Charlie
<http://lumii.lv/ex#Program1>	<http://lumii.lv/ex#programName>	Computer Science
<http://lumii.lv/ex#Program2>	<http://lumii.lv/ex#programName>	Computer Engeneering
<http://lumii.lv/ex#Student1>	<http://lumii.lv/ex#takes>	<http://lumii.lv/ex#Course4>
<http://lumii.lv/ex#Student1>	<http://lumii.lv/ex#takes>	<http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Student2>	<http://lumii.lv/ex#takes>	<http://lumii.lv/ex#Course3>
<http://lumii.lv/ex#Student2>	<http://lumii.lv/ex#takes>	<http://lumii.lv/ex#Course1>
<http://lumii.lv/ex#Student3>	<http://lumii.lv/ex#takes>	<http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Teacher1>	<http://lumii.lv/ex#teaches>	<http://lumii.lv/ex#Course2>
<http://lumii.lv/ex#Teacher2>	<http://lumii.lv/ex#teaches>	<http://lumii.lv/ex#Course3>
<http://lumii.lv/ex#Teacher2>	<http://lumii.lv/ex#teaches>	<http://lumii.lv/ex#Course4>
<http://lumii.lv/ex#Teacher3>	<http://lumii.lv/ex#teaches>	<http://lumii.lv/ex#Course1>
<http://lumii.lv/ex#Course1>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#OptionalCourse>
<http://lumii.lv/ex#Course2>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#MandatoryCourse>
<http://lumii.lv/ex#Course3>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#MandatoryCourse>
<http://lumii.lv/ex#Course4>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#OptionalCourse>
<http://lumii.lv/ex#PersonID123456789>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID345453432>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID555555555>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID777777777>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID987654321>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#PersonID999999999>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#PersonID>
<http://lumii.lv/ex#Program1>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#AcademicProgram>
<http://lumii.lv/ex#Program2>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#AcademicProgram>
<http://lumii.lv/ex#Student1>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Student2>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Student3>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Student4>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Student>
<http://lumii.lv/ex#Teacher1>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Professor>
<http://lumii.lv/ex#Teacher2>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Professor>
<http://lumii.lv/ex#Teacher3>	<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>	<http://lumii.lv/ex#Assistant>

We note that the RDF triple instances obtained here are the same, as obtained in [8] by a manual translation SQL definition, or by D2RQ [2] mapping definition approach. In our case the manual user work needed for the triple creation consists of filling in the appropriate data in the tables *class_map*, *object_property_map* and *datatype_property_map*, as well as *table_link* (see the data in Tables 7, 9, 11, 13, 14). If compared to D2RQ solution of the same instance generation problem, as provided in [8], we note that we have provided a more compact representation of input data by the user since a D2RQ mapping cannot be made aware of the subclass relation in the target ontology. In the example containing a 6-fold specification of instance generation for object property 'teaches' (3 subclasses of 'Teacher' times 2 subclasses of 'Course') a tremendous increase of data volume occurs in case of ontologies with large subclass hierarchies (the instance generation for object property 'teaches' is defined here as a single row in Table 10). If compared to D2RQ [2] approach our method requires no custom SQL writing for mapping definitions, except to define custom views in *class2view* which has not been necessary in our example of *source DB.*

5 Conclusions

In this paper we have demonstrated an example of how relational database itself can be used to create mapping between a source relational database (legacy type) and

target OWL ontology and to generate RDF triples for instance data. The work is still in progress, which means new use cases are studied and the mapping schema is being continuously improved. Next step in our research is to study possibilities for SPARQL to SQL translation in correspondence to the defined mapping.

We plan to apply the current functionality to transform relational data to RDF format in real life medical database [9, 10]. Although our RDB to OWL mapping specification format and implementation can be used together with different end-user mapping specification languages, we are working to define a language that would allow defining the correspondence between target ontology and its corresponding RDB schema elements in a user friendly way.

I would like to thank Karlis Cerans at the Institue of Mathematics and Computer Science, the University of Latvia, for his support and assistance.

References

1. J. Barrasa, A. Gómez-Pérez. Upgrading relational legacy data to the semantic web. In: *Proc. of the 15th International World Wide Web Conference (WWW 2006)*, Edinburgh, United Kingdom, 23–26 May 2006, pp. 1069–1070.
2. D2RQ Platform. Available: http://www4.wiwiss.fu-berlin.de/bizer/D2RQ/spec/.
3. C. Blakeley. RDF Views of SQL Data (Declarative SQL Schema to RDF Mapping). OpenLink Software, 2007.
4. OpenLink Virtuoso Platform. Automated Generation of RDF Views over Relational Data Sources. Available: http://docs.openlinksw.com/virtuoso/rdfrdfviewgnr.html.
5. W. Hu, Y. Qu. Discovering Simple Mappings between Relational Database Schemas and Ontologies. In: *Proc. of the 6th International Semantic Web Conference (ISWC 2007), 2nd Asian Semantic Web Conference (ASWC 2007)*, Busan, Korea, 11–15 November 2007, LNCS, 4825, pp. 225–238.
6. http://www.w3.org/2005/Incubator/rdb2rdf/.
7. http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf.
8. G. Barzdins, J. Barzdins, K. Cerans. From Databases to Ontologies. Semantic Web Engineering in the Knowledge Society. In: J. Cardoso, M. Lytras (eds.), *IGI Global*, 2008, pp. 242–266. ISBN: 978-1-60566-112-4
9. G. Barzdins, S. Rikacovs, M. Veilande, M. Zviedris. Ontological Re-engineering of Medical Databases. *Proceedings of the Latvian Academy of Sciences*, Section B, Vol. 63, No. 4/5 (663/664), 2009, pp. 20–30.
10. G. Barzdins, E. Liepins, M. Veilande, M. Zviedris. Semantic Latvia Approach in the Medical Domain. In: H. M. Haav, A. Kalja, *Proceedings of the 8th International Baltic Conference on Databases and Information Systems*. Tallinn University of Technology Press, 2008, pp. 89–102.
11. J. Barzdins, G. Barzdins, R. Balodis, K. Cerans et al. Towards Semantic Latvia. In: *Proccedings of the 7th International Baltic Conference on Databases and Information Systems*, 2006, pp. 203–218.
12. Object Management Group MOF QVT Final Adopted Specification. Available: http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf.
13. ATLAS Model Transformation Language. Available: http://www.eclipse.org/m2m/atl/.
14. MOLA resources. Available: http://mola.mii.lu.lv/.
15. Eclipse Modeling Framework Project (EMF). Available: http://www.eclipse.org/modeling/emf/.

Appendix

We provide listings of SQL scripts which, when executed in *mapping DB*, generate SQL scripts which, in turn, when executed in *source DB*, generate RDF triples for instances of OWL classes, OWL datatype properties, and OWL object properties.

They are not easily readable as two SQL levels are mixed. They show that mere SQL statement can do the task. They generate SQL statements using string concatenation function || as it is in Oracle DB. They also use Oracle DB functions NVL (null value replacement), NVL2 (return value depends on parameter being null or not null). It is easy to rewrite these SQLs for another DB if needed.

SQL script ***OWL_instance_gen.sql*** that generates SQL statement for RDF triple generation for OWL class instances

```
  SELECT 'SELECT '
 || '''<' || o.xml_base || cm.instance_uri_prefix || ''''
 || ' || '     || c2t.table_name
 || '.' || cm.id_column_expr || ' || ''>'' as subject'
 || ',''' || '< http://www.w3.org/1999/02/22-rdf-syntax-ns#type >''
 as predicate'
 || ',''' || '<' ||o.xml_base || c.rdf_id || '>'' as object'
 || ' FROM ' || c2t.table_name
 || NVL2(cm.filter_expr,' WHERE ', '') || cm.filter_expr as sql4rdf
FROM ontology o,       owl_class c, class_map cm, class2table c2t
WHERE o.ontology_id = c.ontology_id AND
 c.owl_class_id = cm.owl_class_id AND
 cm.class2table_id = c2t.class2table_id AND
 o.ontology_id=1 AND   cm.generate_instances=1
```

SQL script ***generate_sql4datatype_props.sql*** that generates SQL statements for RDF triple generation for OWL datatype property instances

```
  SELECT 'SELECT '
 || '''<' || o.xml_base || cm.instance_uri_prefix || ''''
 || ' || ' || c2t.table_name
 || '.' || cm.id_column_expr || ' || ''>'' as subject'
 || ',''' || '<' || o.xml_base || dp.rdf_id || '>'' as predicate'
 || ',' || dpm.column_expr || ' as object'
 || ' FROM ' || c2t.table_name
 || NVL2(cm.filter_expr,' WHERE ', '') || cm.filter_expr
FROM
 owl_datatype_property dp, datatype_property_map dpm,
 class_map cm, class2table c2t, ontology o
WHERE dp.owl_datatype_property_id=dpm.owl_datatype_property_id AND
 dpm.class_map_id = cm.class_map_id AND
 cm.class2table_id = c2t.class2table_id AND
 dp.ontology_id = o.ontology_id AND o.ontology_id=1
```

SQL script ***generate_sql4object_props.sql*** that generates SQL statements for RDF triple generation for OWL object property instances without intermediate table link usage

```
  SELECT
 'SELECT '
 || '''<' || o.xml_base || cm_domain.instance_uri_prefix || ''''
 || ' || ' || c2t_domain.table_name || '_1'
 || '.' || cm_domain.id_column_expr || ' || ''>'' as subject'
 || ',''' || '<'|| o.xml_base || op.rdf_id || '>'' as predicate'
 || ','
```

```
   || '''<' || o.xml_base || cm_range.instance_uri_prefix || ''''
 || ' || ' || c2t_range.table_name || '_2'
 || '.' || cm_range.id_column_expr || ' || ''>'' as object'

   || ' FROM '
 || c2t_domain.table_name || ' ' || c2t_domain.table_name || '_1'
 || ' INNER JOIN '
 || c2t_range.table_name || ' ' || c2t_range.table_name || '_2'
 || ' ON ' || c2t_domain.table_name
 || '_1.' || opm.source_column_expr
 || ' = ' || c2t_range.table_name || '_2.' || opm.target_column_expr
 || ' WHERE ' || NVL(cm_domain.filter_expr ,' 1=1 ')
 || 'AND ' || NVL(cm_range.filter_expr , ' 1=1 ')
 AS generated_SQL
FROM
 owl_object_property op,
 ontology o,
 object_property_map opm,
 class_map cm_domain,
 class_map cm_range,
 class2table c2t_domain,
 class2table c2t_range
WHERE
 op.ontology_id=o.ontology_id AND
 op.owl_object_property_id=opm.owl_object_property_id AND
 opm.domain_class_map_id =cm_domain.class_map_id AND
 opm.range_class_map_id =cm_range.class_map_id AND
 cm_domain.class2table_id=c2t_domain.class2table_id AND
 cm_range.class2table_id=c2t_range.class2table_id AND
 opm.table_link_id IS NULL AND op.ontology_id=1
ORDER BY 1
```

SQL script ***generate_sql4object_props_table_links.sql*** that generates SQL statements for RDF triple generation for OWL object property instances with one intermediate table link usage

```
  SELECT
 'SELECT '
 || '''<' || o.xml_base
 || cm_domain.instance_uri_prefix || ''''
 || ' || ' || c2t_domain.table_name || '_1'
 || '.' || cm_domain.id_column_expr || ' || ''>'' as subject'

 || ','''' || '<'|| o.xml_base || op.rdf_id || '>'' as predicate'

 || ',' || '''<' || o.xml_base
 || cm_range.instance_uri_prefix || ''''
 || ' || ' || c2t_range.table_name || '_2'
 || '.' || cm_range.id_column_expr || ' || ''>'' as object'

 || ' FROM '
 || c2t_domain.table_name || ' ' || c2t_domain.table_name || '_1'
 || ' INNER JOIN '
 || tl.mid_table_name || ' ' || tl.mid_table_name || '_3'
 || ' ON ' || c2t_domain.table_name || '_1. '
```

```
 || opm.source_column_expr
 || ' = ' || tl.mid_table_name || '_3.' || tl.source_column_expr
 || ' INNER JOIN '
 || c2t_range.table_name || ' ' || c2t_range.table_name || '_2'
 || ' ON ' || tl.mid_table_name || '_3.' || tl.target_column_expr
 || ' = ' || c2t_range.table_name || '_2.' || opm.target_column_expr
 || ' WHERE ' || NVL(cm_domain.filter_expr ,' 1=1 ')
 || 'AND ' || NVL(cm_range.filter_expr , ' 1=1 ')
 || 'AND ' || NVL(tl.filter_expr , ' 1=1 ')
 || 'AND ' || NVL(tl.filter_expr , ' 1=1 ')
 AS generated_SQL
FROM
 owl_object_property op,      ontology o,
 object_property_map opm,     class_map cm_domain,
 class_map cm_range,                        class2table c2t_domain,
 class2table c2t_range,                     table_link tl
WHERE
 op.ontology_id=o.ontology_id AND
 op.owl_object_property_id=opm.owl_object_property_id AND
 opm.domain_class_map_id =cm_domain.class_map_id AND
 opm.range_class_map_id =cm_range.class_map_id AND
 cm_domain.class2table_id=c2t_domain.class2table_id AND
 cm_range.class2table_id=c2t_range.class2table_id AND
 opm.table_link_id=tl.table_link_id AND
 opm.table_link_id IS NOT NULL AND op.ontology_id=1
```

TOOLS AND TECHNIQUES FOR MODEL-DRIVEN DEVELOPMENT

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 121–138 P.

An MDE-Based Graphical Tool Building Framework

Janis Barzdins, Karlis Cerans, Sergejs Kozlovics, Lelde Lace,
Renars Liepins, Edgars Rencis, Arturs Sprogis, Andris Zarins
Institute of Mathematics and Computer Science
University of Latvia, Raina bulv. 29, Riga, LV-1459, Latvia
{Janis.Barzdins, Karlis.Cerans, Sergejs.Kozlovics, Lelde.Lace,
Renars.Liepins, Edgars.Rencis, Arturs.Sprogis, Andris.Zarins}@lumii.lv

In this paper, an MDE-based approach to tool building is described. It is based on a core tool definition metamodel and an interpreter of this metamodel. Besides, an extension of the core metamodel is proposed, allowing for tool-specific model transformations to enrich the behavior of the universal interpreter. As a result, a novel wide-profile tool building platform is obtained. The visualization component of the platform is based on an original high-performance graphical diagram presentation engine which embodies advanced graph drawing algorithms.

Keywords: tool definition metamodel, tool building platform, model transformations.

1 Introduction

In this paper, we present an MDE-based interpretive approach to domain-specific tool (DST) building on the basis of a simple yet flexible and powerful tool definition metamodel (TDMM) that fully specifies a DST as its instance, and the interpreting engine of this metamodel.

The idea of providing explicit metamodeling foundations for the meta-tools themselves has not been central to many powerful developments in DST building area, including MetaEdit+ [1, 2], Pounamu/Marama [3, 4], ViatraDSM [5], Tiger [6], and METAclipse [7]. These tool-building frameworks generally offer some configuration facilities that allow us to define a DST in a user-friendly way (for instance, Pounamu [3] offers a shape designer, metamodel designer, event handler designer and view designer, MetaEdit+ [1] offers Object, Relationship, Role, Port, Graph and Property tools). The result of the configuration process, however, is typically stored in some format that is not revealed to the tool user and that is later compiled or interpreted to obtain a DST.

Our approach advocates opening the tool runtime structures to the end user in the form of a simple metamodel that specifies the DST as its instance. The organization of the DST definition and runtime structures in the form of a simple metamodel, in addition to its theoretical appeal (applying MDE principles to meta-tools supporting MDE-based development themselves), allows for possibilities of basic tool behavior extension by (high-level) model transformations that we ascribe to certain well-defined extension points in the tool definition metamodel and that are handled by the tool metamodel interpreting engine. These transformations can be used for, e.g., domain

model synchronization, constraints, dynamic content in the tool, advanced presentation behavior, as well as integration with other data engines.

Some further benefits of the metamodel-based tool data structure include model migration possibilities among tool and meta-tool versions just by model transformations, as well as easy external access to model repository.

The idea of DST definition by a metamodel is already successfully implemented, for instance, in the Eclipse GMF framework (GMF) [8] (Microsoft DSL Tools (DSLT) [9] also follow a related approach). In GMF, a tool definition consists of instances that correspond to domain, graphical definition, tooling and mapping metamodels, and the tool itself is obtained by compiling these instances into a Java code. The main difference of our approach from that of GMF or DSLT is that we aim for greater flexibility of MDE-level constructs in tool definition by following the tool model interpretation approach (instead of compilation into JAVA or C#). On the basis of this approach, we are able to offer the possibility to extend tool behavior by means of model transformations that can be attached to certain well-defined points in the tool definition metamodel. In GMF or DSLT, the tool behavior extension is possible by adding code to the JAVA or C# classes generated for the tool by the framework. This task may be feasible; however, it requires rather profound expertise in the internal program-level structure of classes and methods generated by the corresponding framework. Our approach provides an alternative to GMF and DSLT by allowing us to create the extensions in model-level rather than program-level terms.

We structure the presentation of the TDMM into core and extended versions, where the core metamodel allows for basic tool behavior description and the extended TDMM allows for model transformation incorporation. In Core TDMM, we focus on tools defined directly in terms of their graphical presentation (there are applications where this is sufficient). The handling of domain model, if that were necessary, is delegated to model transformations (allowed by the extended TDMM) that can perform the task (see, e.g., [7] for comparison of static mapping and model transformation approaches in modeling tools).

The TDMM is defined to contain both the tool definition and tool runtime instances at the same metamodeling abstraction level. This is achieved using a structure that resembles an adaptive object-model [10] element type pattern. A theoretical note: this design allows for easy implementation of dynamic tool model reconfiguration in parallel with particular model creation by the tool, as advocated, for instance, in [3] (in practice the modeling power of the platform is restricted in its “end-user” versions and user model migration between tool versions (as well as between platform versions) is achieved by model transformations).

The TDMM is also defined as an extension of a more general graph diagramming metamodel (GDMM) [11], and its implementation is provided by universal (platform-level) model transformations associated with the events (instances of Event class) defined in GDMM that interpret the specific TDMM instance. To make our presentation complete, we also review GDMM in this paper. An earlier authors’ work with much more limited tool definition possibilities and without separating GDMM from the tool definition metamodel has been reported in [12].

The rest of this paper is organized as follows. Section 2 describes the graph diagramming metamodel and engine (GDE), explaining what is used as the basis for

the tool-building platform. The communication mechanism between the graph diagram presentation engine and the "business logic engine" that is behind any specific graph diagramming tool and is typically implemented by model transformations (on the basis of GDMM) is also outlined here.

Section 3 presents the core tool definition metamodel, including its full abstract syntax and explanation of its semantics. This metamodel is general enough to allow for a definition of a broad class of DST, including the EMOF [13] class diagram and UML 2.0 activity diagram editors; yet it is simple enough for its abstract syntax, together with the relevant parts of GDMM, to be presented on a single page. We also outline principles of implementing the core tool building platform on the basis of GDE.

In Section 4, we extend the core tool definition metamodel to allow for MDE-based extension mechanism to the platform. It is a widely accepted fact that extensions are among the most complicated problems every meta-tool faces. The extension mechanism we propose is not hidden in the depths of implementation; instead, it is elevated to the level of the metamodel. The core tool definition metamodel together with extensions is sufficient to build efficiently, e.g., a full UML 2.0 class diagram editor with full support of attributes, stereotypes and tagged values, as well as other DST editors of comparable complexity. The high-level extensibility mechanism based on model transformations allows us to achieve such tool features that are beyond the scope of usual DSTs.

2 The Graph Diagramming Metamodel and Engine

The tool definition metamodel together with its interpreter – the tool building platform – are based on basic presentation services whose interface is described by metamodels. One of the most important such services is that of graph diagramming. It is defined by means of a graph diagramming metamodel (GDMM) and implemented by a graph diagramming engine (GDE). Another service for which we also have a metamodel and a corresponding engine is that of property editors. The property editor metamodel and engine are used in our implementation of the tool building platform; however, they are not of primary importance in explaining its semantics. Therefore, they are not considered in detail here.

The aim of GDMM is to describe the graph diagramming functionality that can be offered by GDE and that is common to a wide range of graphical diagramming tasks that may go beyond any particular DST, or even the task of DST building in general. Since providing appropriate abstractions in GDMM can considerably ease the tool definition process on the basis of GDE, the emphasis in the design of GDMM has been on properly separating our concerns into "purely graphical" tasks that are to be handled by the GDE itself, and tasks involving "logic" of the tools using GDE.

GDMM (Fig. 1) is built around the classes for visual elements of the presentation, namely *GraphDiagram*, *Element*, *Box*, *Line*, and *Port* together with *Compartment* corresponding to text fields placed in boxes and attached to lines and ports (note that the start and end of lines can be attached to any elements, not just boxes). Instances of these classes are diagrams and elements created by the user. Every element, compartment and graph diagram has its style (see classes *ElemStyle*, *CompartStyle* and *GraphDiagramStyle*). The metamodel allows for every element to specify its default

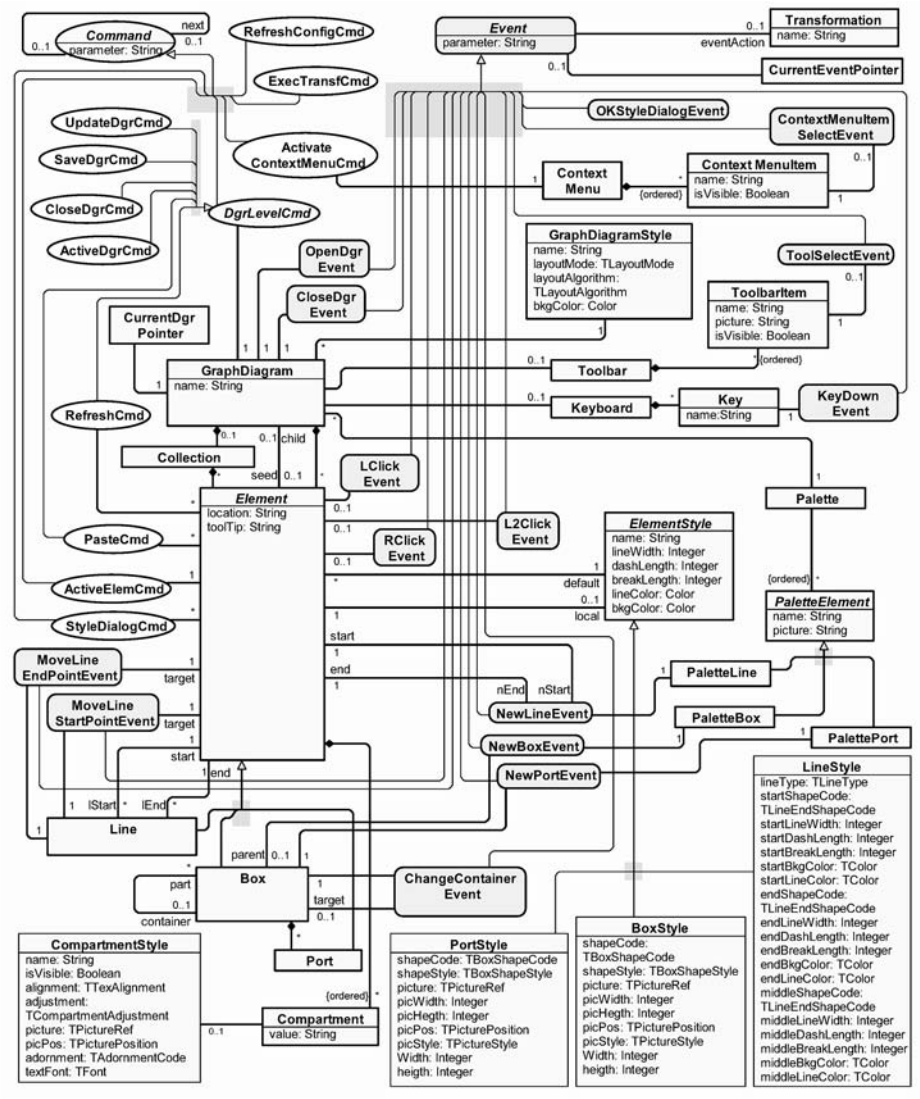


Fig. 1. The graph diagramming metamodel

style and local style (the diagramming engine uses the local style if it is defined; otherwise the default style is used). The *Collection* class contains a single item that is linked to currently active (selected) elements in the diagram. The *seed*/*child* link between *Element* and *GraphDiagram* permits specifying an element to be a seed for a diagram (typically, not the diagram the element is in), thus providing means for building diagram hierarchies.

Besides the classes of visual elements, GDMM also contains classes describing the tool's environment (*Palette*, *Toolbar* and *Keyboard* classes with corresponding elements). Instances of these classes are typically created at the tool creation time and

do not change during the work with a tool. A context menu (*ContextMenu* class) can also be specified to be opened in response to the tool's request.

There is an *Event* class in GDMM whose singleton subclasses correspond to the actions the user may perform on a particular diagram (the event classes are represented as rounded rectangles), and that are understood by GDE. Upon observing a current event, GDE invokes the event's *eventAction* transformation responsible for particular tool's "business logic" in response to this event. The *Command* class describes the requests (commands) that the tool transformations can issue for GDE. There may be several commands issued by a single tool transformation. Command classes are denoted as ellipses in Fig. 1.

For instance, the creation of a new box in a graph diagram starts by the user clicking the tool-triggering GDE to set *CurrentEventPointer* to the only instance of *NewBoxEvent* (the *parent* link from the event is set if the new box is to be created inside another box). The event's transformation then may, for instance, create a new element of the *Box* class (or it may do some extra/other action depending on the tool's specific logic). Then it creates an instance of *UpdateDgrCmd* and transfers the control back to GDE that processes the command by updating the diagram so that the newly-created box becomes visible.

The semantics of some further *Command* subclasses is explained as follows. The *ActiveDgrCmd* sets the editor's focus on the particular diagram, *RefreshCmd* refreshes the specified elements in the diagram, *PasteCmd* computes coordinates of elements pasted into the diagram model, *RefreshConfigCmd* rebuilds toolbars and palettes, *ActivateContextMenuCmd* opens a context menu (depending on the collection of elements pointed to by the *Collection* element), *StyleDialogCmd* opens the style dialogue of elements, *ExecTransfCmd* is used for calling back transformations. The other commands and events should be mostly self-explanatory.

Although most of user activities in a tool trigger setting of the current event and invoke some transformation, there are actions that are performed solely by GDE (e.g., undo/redo, zoom, export to HTML, print diagrams, etc). The toolbar items responsible for these actions do not have associated *ToolSelectEvent*s to be triggered when the user selects the toolbar item. The context menu item that is handled directly by GDE is "Symbol Style". GDE is also responsible for handling element coordinates (the coordinates can be abstracted away while writing tool defining transformations).

The implementation of GDE has been a considerable programming task of several person years. The relatively simple diagram structure has allowed us to implement advanced graph drawing capabilities [14, 15] in GDE, which support diagram initial layout application as well as serve the interactive diagram editing process. The definition of GDE interface in the form of GDMM allows for reuse of its graph diagramming capabilities in various MDE-related tasks, among them, meta-tool creation. The architecture of GDE is described in more detail in [11, 16].

3 The Tool Definition Metamodel: the Core

In this section, we describe the syntax and semantics of the core tool definition metamodel (Core TDMM) that can have (simple) modeling tools as its instances. The aim of Core TDMM is to provide basic means for DST definition on the level of graphical

presentation. There is a wide range of applications where the graphical presentation view on the modeled system is sufficient since this is the view of the system directly perceived by the user. The other views on the system if necessary can be obtained by model transformations that work either offline, performing export and import tasks, or synchronously, using tool behavior extension points, as described in Section 4.

Core TDMM (Fig. 2) is built around the concepts of *GraphDiagramType*, *ElementType* and *CompartmentType*, providing type (or pattern) information for graph diagrams, elements and compartments that are specified in the graph diagramming metamodel (GDMM) and that may appear in the particular tool's visual editor. Therefore, Core TDMM is described as an extension of GDMM. Fig. 2 describes the classes of Core TDMM as well as a selection of relevant classes of GDMM (in two grey rectangles).

The containment hierarchy *Tool* → *GraphDiagramType* → *ElementType* → *CompartmentType* (via *base* link) forms the backbone of TDMM. Every tool can serve several graph diagram types (one of these being the *first* diagram type in the tool). Every graph diagram type contains several element types (instances of *ElementType*), each of them being either a box type (e.g., an Action in the activity diagram), a line type (e.g., a Flow), or a port type (e.g., a Pin). Every element type has an ordered collection of *CompartmentType* instances attached via its *base* link. These instances form the list of types of compartments of the diagram elements of the particular element type.

We notice the resemblance of relations between graph diagram and graph diagram type, element and element type, and compartment and compartment type to adaptive object model [10] patterns.

The element type specification (*ElementType* class and its subclasses) allows to describe inclusion possibility between boxes of different types (*partType*/*containerType* relation), attachments of ports to boxes, the box type multiplicity constraints (e.g. 0..1 boxes of certain type in a diagram), as well as line type connectivity rules (the element type pairs for which connection by a line of a certain type is possible are specified by *LineSubtype* class instances).

The *CompartmentType* class is divided into subclasses according to the multiplicity of the type's compartments in the elements as well as the possibilities to work with them in the property editor. Table 1 summarizes these subclasses.

Table 1

Compartment type subclasses

FieldType	Single input field.
MultiLine FieldType	Multi-line input field, with each line corresponding to a compartment. The empty field corresponds to no compartments of this type in the element.
LabelType	Non-editable label. Used, for instance, in the property editor to show element names.
CheckBox Type	Check box. The attribute *displayValue* defines the value shown in the diagram when the user has selected the corresponding value. For instance, in a class diagram, when an attribute is derived (the corresponding check box is selected, activating a *CheckBoxItem* with value **true**), it should be displayed in diagram as "/".
ComboBox Type	Combo box. The user can choose among certain values predefined as *ComboChoiceItems*.

ComplexType	Compartment consisting of several sub-compartments. It can be entered either directly (e.g., as a string "attr:Integer=5"), or in a separate window, where values for sub-compartments (e.g., "attr" for the name, "Integer" for the type, and "5" for the default value) can be entered. The compartment's value is obtained by concatenating the values of its sub-compartments supplemented with the corresponding prefixes (like ":" and "=") and suffixes. See *displayPrefix* and *displaySuffix* attributes in *CompartmentType*. Note that to support the compartment hierarchy persistence beyond the element's editing time as well, we have introduced a *subCompartment* link from GDMM's *Compartment* class to itself in TDMM.
MultiLine ComplexType	Multi-line input field, where each line is a compartment of *ComplexType*.

In TDMM, there are diagram, element and compartment styles from GDMM connected to diagram, element and compartment types, determining how the diagrams, elements and compartments of the corresponding types are visualized (see Fig. 1 for style attributes). Apart from specifying the default style for diagram, element and compartment types, TDMM allows for the so-called optional styles of element and compartment types that can be triggered to become effective for a particular element/compartment, selecting a certain choice item in (possibly another) compartment of *CheckBoxType* or *ComboBoxType* (the links *elemStyleByItem* or *compartStyleByItem* from the *ChoiceItem* to the particular style instance are used). A classical application of this feature is putting or canceling the formatting of the class name compartment in italics depending on the value of class attribute isAbstract; however, this feature is much more useful.

Another form of dynamics supported by Core TDMM is adding compartments of new types to the elements depending on some compartment's value selected in a combo-box. This dynamics is implemented by defining instances of *DynamicCompartTypes* class as well as setting their dependencies from their triggering combo-box choice items, the position where the new compartments go, as well as the list of new compartment types themselves. This dynamics may be useful, for instance, in implementation of tagged values associated with stereotypes.

In TDMM we extend the GDMM *Compartment* class by the *inputValue* attribute, so that every compartment has both *inputValue* and *value* attributes. The *value* attribute to be displayed in the diagram is obtained from *inputValue* by prefixing it with compartment type's *displayPrefix* and suffixing it with *displaySuffix* (an example of this construction is putting double angle brackets around the stereotype name).

Besides the element and compartment types, every graph diagram type can have an associated toolbar consisting of toolbar elements. We consider only pre-defined (core) toolbar elements whose implementation is provided by GDE in Core TDMM.

The graph diagram type has an associated palette to be shown with particular diagrams. Each of the palette elements are connected to one or more (in case of ports or lines) element types. This connection determines the type of element being created when a palette element is activated. If several line or port types are connected to one palette element (for instance, in class diagrams it may be convenient to use the same palette element for creating associations and links), the type of element is determined by the context of the corresponding *NewLineEvent* or *NewPortEvent*. If there is more than one possible alternative, the list of options is presented to the user.

The context menus (*ContextMenu* instances) can be ascribed to element types as well as to graph diagram types. There may be different context menus for the same diagram depending on the existence of selected elements in the diagram; therefore, there are two associations – *contextCollection* and *contextEmpty* – from *GraphDiagramType* to *ContextMenu*. In Core TDMM we consider only items implemented by GDE (symbol style), or that are provided a universal implementation on the level of tool definition platform ("properties", "copy", "cut", "paste", "delete", "refine").

Similarly, we include a keyboard with universal keys in Core TDMM, allowing for standard editor functionality (e.g., Ctrl+C for "copy", Ctrl+V for "paste", etc), or serving as shortcuts for GDE services (e.g., Ctrl+> for "zoom in" etc).

Implementation of the tool definition framework is achieved by developing an interpreter that, relying on the existing implementation of GDE (Section 2), interprets a particular instance of TDMM in the way the corresponding tool reacts from the end user's point of view.

Regarding semantics of Core TDMM and its interpreter, we note that *LClickEvent* does not invoke a transformation, *RClickEvent* prepares and opens context menu (via *ActivateContextMenuCmd*), and *L2ClickEvent* opens a property dialog.

The interpreter also uses a property dialog engine (PDE) with a metamodel-based interface (the property dialog metamodel, PDMM). This architecture allows the interpreter to be written as a collection of model transformations. The transformations have been created for all events of GDE, and they are responsible for the "business logic" of the tool that corresponds to the semantics of Core TDMM, outlined here. We have used the model transformation language L0+ [17] for our implementation; however, other "higher-level" transformation languages could have been used as well (e.g., the graphical model transformation language MOLA [18]).

An alternative approach to particular tool definition could be to write the transformations implementing the behavior of the tool directly against the events of GDMM. The possibility remains to replace some of the platform-defined transformations by tool-specific transformations (for instance, one may replace the "properties" transformation by "refine" transformation (navigate from the seed to the child) as a response to *L2ClickEvent* for some specific element types). Our approach to introducing tool-specific behavior (explained in Section 4), however, is via a mechanism of extending universal platform-level transformations instead of replacing them, so that the functionality present in the platform-level interpreter is efficiently retained.

As to the expressiveness of the proposed metamodel, a very wide range of graphical tools (inter alia an editor for EMOF [13] class diagrams and UML activity diagrams) can be defined as its instances.

We note that many popular and powerful meta-tools (see, for instance, MetaEdit [2, 3]) do not present an explicit tool definition metamodel, but explain the tool behavior by means of some configuration facilities for the end user instead. Some meta-tools provide the possibility to use more powerful constraints in some constraint definition language. However, if we want to offer a really dynamic behavior, we have to do serious programming and to understand the implementation of the particular meta-tool thoroughly. In our approach, all information relevant to DST building and running is captured as an instance of an expressive yet sufficiently simple metamodel (Fig. 2), thus providing sufficiently easy means for tool functionality extensions. These extension opportunities are described in the next section.

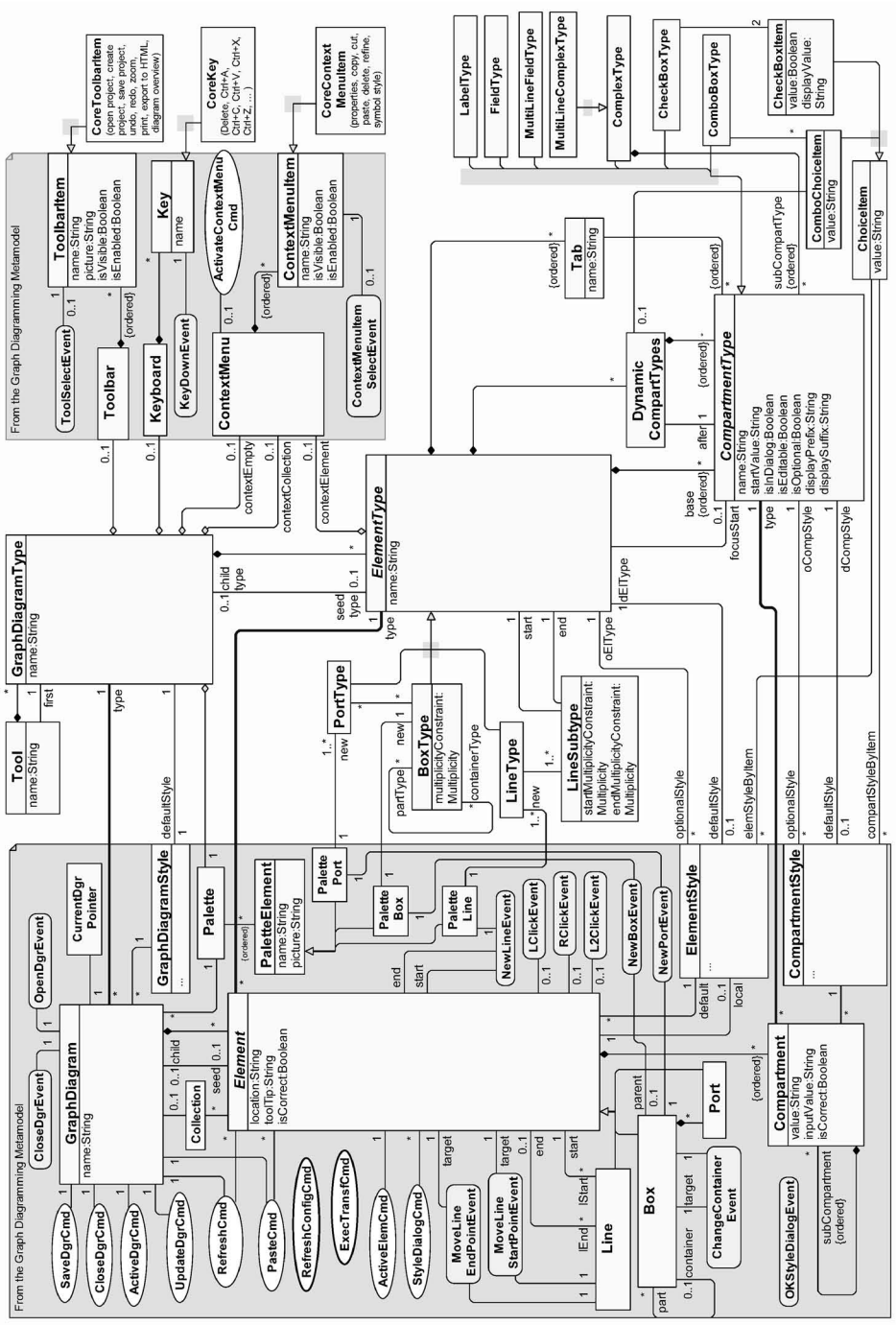


Fig. 2. The tool definition metamodel

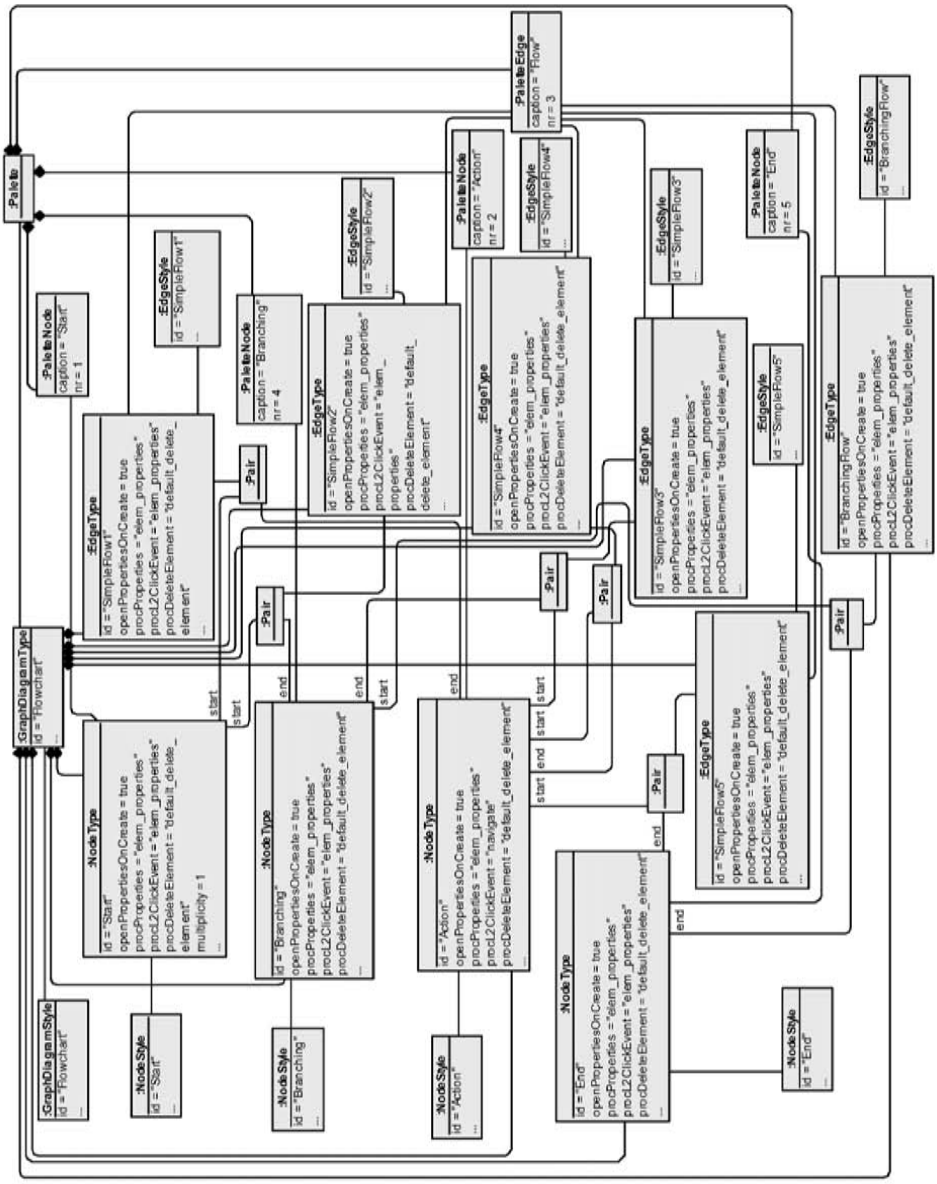


Fig. 3. Graph diagram type "Flowchart" and its context

3.1 Instantiation of the Tool Definition Metamodel

In this sub-section, an example is given in a form of a simple flowchart editor, which is an instance of the tool definition metamodel. Since the instance graph turned out to be quite large and thus unreadable for humans, it has been divided into three parts here. The first part (Fig. 3) contains the top level type information – instance of *GraphDiagramType* representing the flowchart diagram type – and its context. Here, any Flowchart diagram

is to consist of four *NodeTypes* – Start, End, Action, and Branching. For each of them, the most important attributes are given. For example, for the Start element, it is said that only one element of this type is allowed in a flowchart (see attribute "multiplicity"). Also, some transformation names can be seen, e.g., a transformation "navigate" is to be called when user double-clicks an Action, while a transformation "elem_properties" is to be called when user double-clicks a node of some other type. Next, several *EdgeTypes* exist in order to offer an opportunity to draw a line between nodes of different types. However, all these edge types are connected to one *PaletteEdge* called "Flow"; thus, the user is not responsible for picking the right palette element for different flow types – they all look alike from the user's point of view. Finally, *NodeStyle* and *EdgeStyle* instances are present as well. Due to the large number, all style attributes are not listed here.

The second part of the instance graph contains detailed information about the four node types sketched in the first part (Fig. 4 and 5). For every node type, a *PropertyDiagram* and a *PopUpDiagram* is depicted. The property diagram is a way to specify the property dialog window to be opened when the user, for example, double-clicks some element. Here, property diagrams of Action and Branching node types each consist of one *PropertyRow* being a simple text field (see attribute "rowType") for entering and altering the values of the respective compartments (of *CompartmentTypes* "Expression" and "Condition", respectively). The pop-up diagram contains *PopUpElements* to be shown

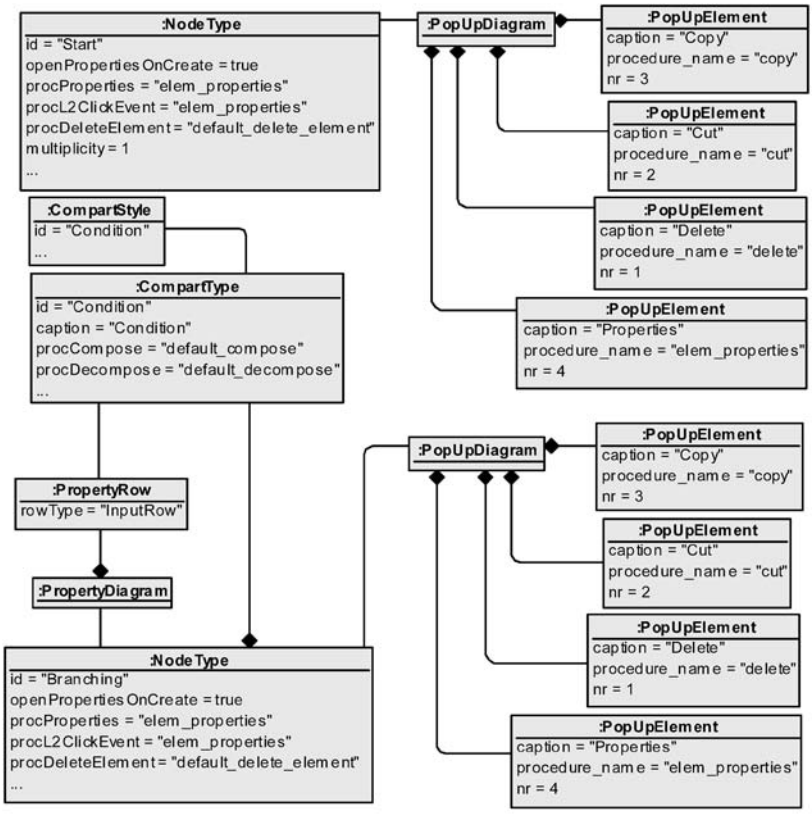


Fig. 4. Flowchart node types "Start" and "Branching" and their context

when the user, for example, clicks with the right mouse button on some element. Here, each pop-up menu contains four elements for standard actions “copy”, “cut”, “delete”, and “properties”. For each pop-up element, a calling transformation name is specified with the attribute “procedure_name”.

The third part of the instance graph contains detailed information on the edge types sketched in the first part (Fig. 6). The information to be specified for an edge type is approx. the same that needs to be specified for a node type. Thus, instances shown here are quite alike to those shown in Fig. 4 and 5.

4 The Tool Definition Metamodel: Extensions

The implementation of Core TDMM, as described in Section 3, attached a fixed universal model transformation to every event of the presentation engine (GDE). However, there may be situations in advanced tool building when such standard universal functionality is not sufficient and a tool-specific behavior is required. For instance, there may be a need to synchronize the contents of the graphical editor with data in some other source (e.g., a domain model), or there may be some further restrictions or constraints to be observed regarding elements and values that can be introduced during the diagram editing process.

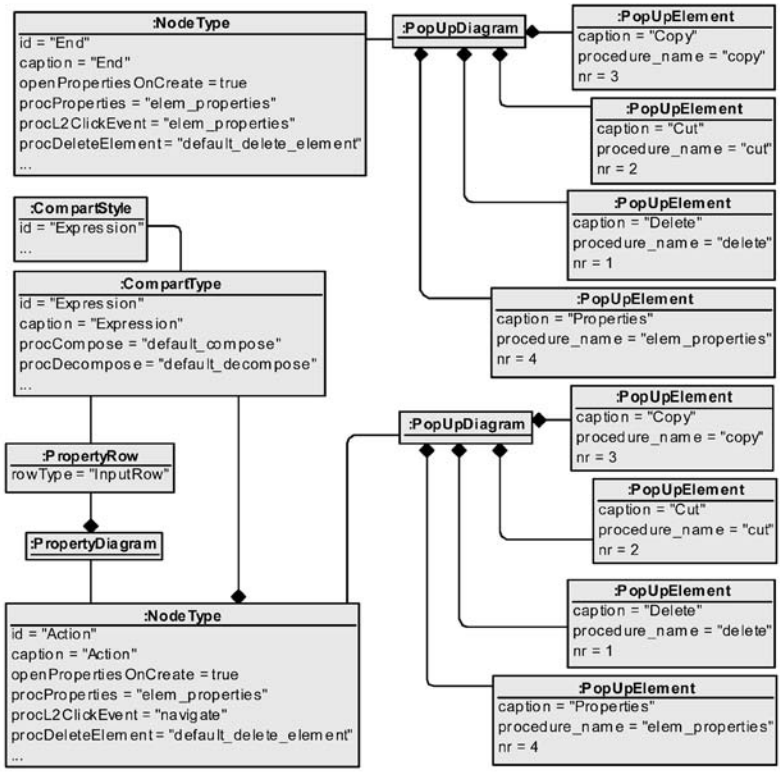


Fig. 5. Flowchart node types “End” and “Action” and their context

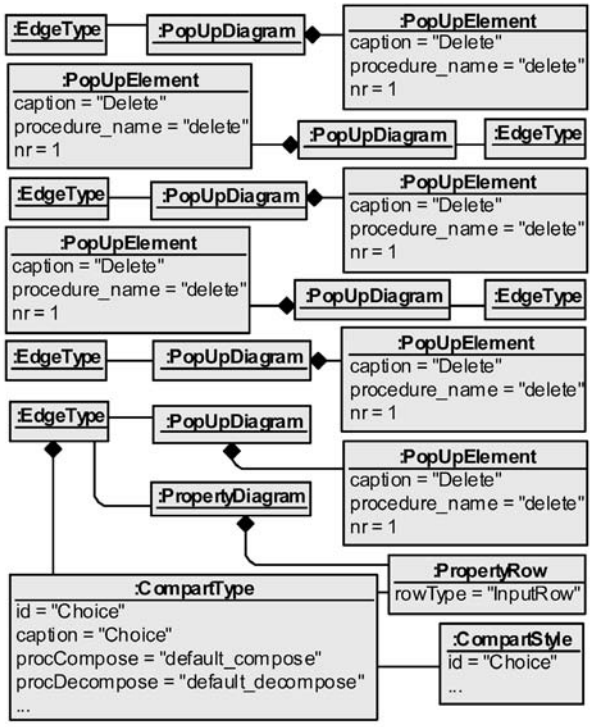


Fig. 6. Flowchart edge types and their context

Since the tool to be defined by the tool definition platform conforms to the given tool definition metamodel, in principle, it is possible to allow the tool builder to write his/her own model transformations for handling certain events instead of the transformations in-built in the platform (these transformations can work in terms of the tool definition metamodel). However, our approach to the tool functionality extension is more refined in that we allow the tool builder willing to introduce the extended functionality to rely on the basic work done by the transformations implementing the platform nevertheless. This is achieved by extending Core TDMM with classes *XElemType* and *XCompartType* that are subclasses of *ElemType* and *CompartType,* respectively (Fig. 7). These classes contain attributes that correspond to certain call points at which the platform-level event processing transformation (which is to be adopted to respect these call points) may give control over to an external tool-specific transformation.

The extended tool definition metamodel also contains classes *AdvancedKey*, *AdvancedContextMenuItem* and *AdvancedToolBarItem* that provide the tool constructor with more points where the tool-specific transformations can be attached.

In the remaining paper, we explain the semantics of particular call points – their placement in the tool interpretation process. We claim that this explanation, together with understanding of the tool definition metamodel, is sufficient to efficiently use the call point mechanism in advanced DST building. This is in sharp contrast with the amount of platform-specific implementation details required for developing advanced tools, for instance, in the Eclipse GMF platform [8].

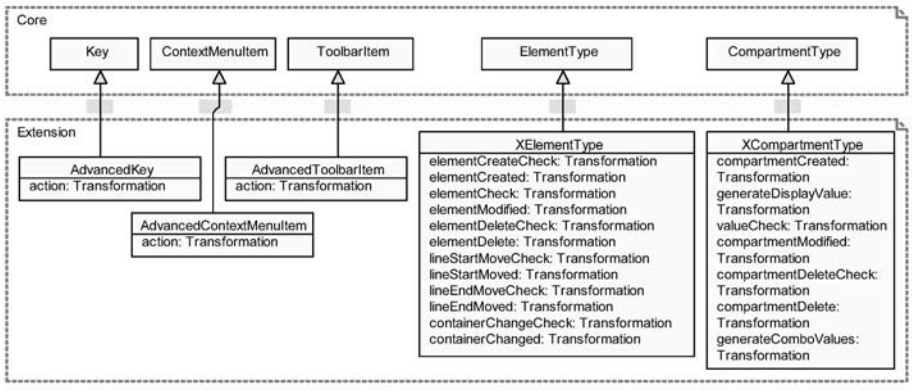


Fig. 7. The tool definition metamodel: extensions

Table 2 summarizes the call points in the *XElementType* class that arise in connection with element creation, content modification and deletion (if not specified otherwise, each transformation accepts a corresponding instance E:*Element* as its only argument; the call points are designed to have transformations that either do or do not have a (Boolean) return value).

Table 2

Call points in *XElementType*

elementCreateCheck : Boolean	Called before creating an element (an instance of the *Element* class). If the function returns **false**, the element creation process is canceled. Recommended for initial correctness constraints (e.g., whether a new element of the given type is possible in the diagram).
elementCreated	Called after creating the element, after *elementCreateCheck*, before adding compartments.
elementCheck : Boolean	Called upon completing value change of the element's compartments. The result of the function is recorded in the element's *isCorrect* attribute. The user is notified if the transformation returns **false**.
elementModified	Called upon completing value change of the element's compartments, after *elementCheck*.
elementDeleteCheck : Boolean	Called upon the user's request to delete an element, after the system's own checks for the possibility to delete are completed. If the return value is false, the "delete" action is canceled.
elementDelete	Called upon the user's request to delete an element, after *elementDeleteCheck*, before (unconditional) deleting of the element.
lineStartMoveCheck (E, OLDSTART, NEWSTART: *Element*): Boolean	Called upon the user's request to move the line's start point, after the system's own checks for the possibility of action are completed. If the procedure returns **false**, the action is canceled.
lineStartMoved (E, OLDSTART, NEWSTART: *Element*)	Called after the line's start point has been moved.
lineEndMoveCheck (E, OLDEND, NEWEND: *Element*): Boolean	Called upon the user's request to move the line's end point, after the system's own checks for the possibility of action are completed. If the procedure returns **false**, the action is canceled.

lineEndMoved (E, OldEnd, NewEnd: *Element*)	Called after the line's end point has been moved.
containerChangeCheck (E:*Element*,OC:[*Element*], NC:[*Element*]) : Boolean	Called upon the user's request to change the element's container (e.g., to move a box in or out another box, or from one containing box to another), after the system's own checks for the possibility of action are completed. [*Element*] denotes optional argument of type *Element*. If the procedure returns **false**, the action is canceled.
containerChanged(E:*Element*,OC:[*Element*], NC:[*Element*])	Called after the element's container has been changed.

Note. Moving a line's start or end point or changing a container do not invoke initial deletion and further creation of elements; therefore, the corresponding call points for element deletion and element and compartment creation are not activated.

Table 3 summarizes the call points in the *XCompartmentType* class (each transformation accepts a corresponding instance C:*Compartment* as its argument).

Table 3

Call points in *XCompartmentType*

compartmentCreated	Called after creating compartment and setting its context (link to the element or containing compartment), before setting up the compartment's value and processing sub-compartments.
generateDisplayValue	If specified, is used instead of the Core mechanisms for generating the compartment's value (as seen in the diagram) from an input value (as entered in the property editor). Called after the input value of the compartment is prepared (e.g., in the property editor).
valueCheck: Boolean	Called upon completing a value change of the compartment, after *generateDisplayValue*. The result of the procedure is recorded in the compartment's *isCorrect* attribute. The user is notified if the transformation returns **false**.
compartmentModified	Called upon completing a value change of the compartment, after *valueCheck*.
compartmentDeleteCheck : Boolean	Called upon the user's request to delete the compartment, after the system's own checks for possibility to delete are completed. If the procedure returns **false**, the action is canceled.
compartmentDelete	Called upon the user's request to delete the compartment, after *compartmentDeleteCheck*, before (unconditional) deleting of the compartment.
generateComboValues	Procedure for dynamic generation of values in the compartment's combo box in the property editor. If unspecified, the combo box is filled up by means specified in the Core.

Note. The *compartmentDeleteCheck* and *compartmentDelete* transformations are not called when deleting a whole element.

Note 2. The tool-specific transformations inserted at the call points are not automatically invoked in case of the user's own manipulation of the model contents behind the platform's event-processing transformations.

The introduced tool extension mechanism, albeit simple, is sufficient for a large range of tasks arising in DST building. We mention some of them here:

- synchronization with an abstract user-defined domain model,
- constraints of potentially arbitrary logical complexity,
- dynamic contents in the tool (e.g., drop-down values in a combo-box),
- advanced dependencies in the tool's presentation behavior,
- integration with other data engines (e.g., data from relational databases, provided the data access interface is created).

Synchronization of the contents of the model with a user-defined domain model can be performed by transformations *elementCreated*, *elementModified* and *elementDelete*, as well as *compartmentCreated*, *compartmentModified* and *compartmentDelete* that provide the tool builder the points at which a corresponding action can be defined in the domain model (e.g., creating, modifying or deleting a structure corresponding to an element or compartment on the presentation level). If necessary, the *lineStartMoved*, *lineEndMoved* and *containerChanged* transformations can also be used for this purpose.

The constraints can be implemented in the tool by the transformations *elementCreateCheck*, *elementCheck*, *elementDeleteCheck*, *lineStartMoveCheck*, *lineEndMoveCheck*, *containerChangeCheck*, *compartmentDeleteCheck* and *valueCheck*. All these transformations, except *elementCheck* and *valueCheck*, cancel the action initiated by the user in case of returning **false**. The result of *elementCheck* and *valueCheck* transformations is placed in the element's or compartment's attribute *isCorrect*, and the user is notified to take a correcting action in the case if the result had been false. Note that both the structure of the model created in the editor (the presentation) and the tool-specific domain model information can be accessed by the procedures implementing the constraints.

Since the DST conforms to the (extended) tool definition metamodel (is an instance of this metamodel), the transformations attached to the call points as well as the event-processing transformations defined by the user (in case of *AdvancedKey*, *AdvancedContextMenuItem* and *AdvancedToolBarItem*) can be defined, in principle, in any high-level model transformation language. This means that we have reached a point when an advanced DST including user-defined extensions can be fully implemented within an MDE framework without the need to resort to structures and constructs typical of programming languages. With the extension mechanism, programmers are free to add a dynamic behavior to the tool being created without putting in too much effort. The simplest example is perhaps generation of combo box items dynamically – if needed, the transformation *generateDisplayValue* can do the job.

Furthermore, the definition of the call points in the tool interpretation process hides the details of the tool interpretation process from the user (it allows the user to seamlessly re-use the implemented process). It allows the user to focus just on adding the tool-specific advanced functionality and rely on the fact that transformations will be called at the right time and place. The only requirement for the tool builder (the writer of extension transformations) is not to introduce inconsistencies in the metamodel depicted in Fig. 2.

5 Conclusions

In this paper, we have presented a universal tool definition metamodel with an extension mechanism that allows us to construct advanced DSTs while staying within the MDE framework. The static part of the tool has to be first defined as an instance of the (extended) tool definition metamodel, and then the model transformations for tool-specific operations as well as for defined call points can be provided.

The definition of the static part of the tool can be performed by a model transformation, setting up the appropriate instances necessary for the work of the tool (these include instances of *ElementType*, *CompartmentType*, as well as *ElementStyle* and *CompartmentStyle* and their related classes). Nevertheless, our implementation of the platform also provides a configurator (as most of DST building platforms do) that can be used to set up the tool's instances in a user-friendly way. All our test cases (including the UML 2.0 class diagram editor with a full support of attributes, stereotypes and tagged values) and practical applications of the platform (including several document flow and workflow modeling systems, e.g. [19]), have been successfully created using the configurator and providing the specific transformations at suitable extension points, where necessary.

We note that for a large range of tools, most of the tool functionality fits into Core TDMM and that the transformations at the call points tend to be rather small in size. We usually call them "mini-transformations"; however, we also recognize the potential of using more powerful transformations.

The MDE-based platform has allowed for building of modeling tools that are integrally incorporated into larger business information infrastructure where the graphical modeling of processes within a DST is coupled with the organization's actual data residing, for instance, in a relational database (e.g., a transformation looking up values for a combo box drop-down list can be easily redirected to an external data source, or a copy of the system model can be easily transferred to a database where further analysis of it can be enabled, etc).

The tool architecture allows for both accessing the external data from the tool's environment (provided suitable adapters for external data are created; we have elaborated on such architecture in [16]) and accessing the tool's repository from an external application. The easy external access to the graphical contents of the tool's model has proved useful, for instance, for visualizing feedback in the model from actually implemented systems.

In practice we also noticed that model migration between the tool and platform versions by model transformations works seamlessly from the end user's point of view.

We are looking forward to new applications of our platform within the area of integrating modeling tools within larger information infrastructures as we believe in the MDE-based approach we have chosen as the basis for DST building.

6 References

1. MetaEdit+ Workbench User's Guide, Version 4.5. Available: http://www.metacase.com/support/45/manuals/mwb/Mw.html, 2009.
2. S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008, 448 p.

3. Z. Nianping, J. Grundy, J. Hosking. *Pounamu: a meta-tool for multi-view visual language environment construction.* 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC '04), 30 September 2004, pp. 254–256.
4. J. Grundy, J. Hosking, J. Huh, K. Na-Liu Li. *Marama: an Eclipse Meta-Toolset for Generating Multi-View Environments.* ICSE '08, May 10–18, Leipzig, Germany. 2008.
5. I. Rath, D. Varro. Challenges for advanced domain-specific modeling frameworks. *Proc. of Workshop on Domain-Specific Program Development (DSPD).* ECOOP 2006, France.
6. C. Ermel, K. Ehrig, G. Taentzer, E. Weiss. Object-Oriented and Rule-Based Design of Visual Languages Using Tiger. *Proceedings of GraBaTs '06,* 2006, p. 12.
7. A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, J. Barzdins. Building Tools by Model Transformations in Eclipse. *Proceedings of DSM '07 Workshop of OOPSLA 2007, Montreal, Canada.* Jyvaskyla University Printing House, 2007, pp. 194–207.
8. Graphical Modeling Framework (GMF, Eclipse Modeling Subproject). Available: http://www.eclipse.org/gmf/.
9. S. Cook, G. Jones, S. Kent, A. C. Wills. Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley, 2007, 524 p.
10. J. W. Yoder, F. Balaguer, R. Johnson. Architecture and Design of Adaptive Object-Models. ACM SIGPLAN Notices, Vol. 36, 2001, pp. 50–60.
11. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proc. of Workshop on Model Driven Development of Advanced User Interfaces (IUI 2009).* Florida, USA.
12. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation-Based Graphical Tool Building Platform. *Proc. of MODELS 2007 Workshop on Model-Driven Development of Advanced User Interfaces (MDDAUI 2007).* Nashville, USA.
13. Meta-Object Facility (MOF). Available: http://www.omg.org/mof/.
14. K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proc. of Latvian Academy of Sciences*, Section B, Vol. 55, No. 1, 2001, pp. 43–51.
15. P. Kikusts, P. Rucevskis. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Proc. of Graph Drawing '95*, Lecture Notes in Computer Science, Vol. 1027, 1996, pp. 361–364.
16. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation- Driven Architecture. *Proceedings of DSM '08 Workshop of OOPSLA 2008*, Nashville, USA, pp. 60–63.
17. J. Barzdins, A. Kalnins, E. Rencis, S. Rikacovs. Model Transformation Languages and Their Implementation by Bootstrapping Method. *Pillars of Computer Science*, Lecture Notes in Computer Science, Vol. 4800. Springer-Verlag, 2008, pp. 130–145.
18. A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA. *Proceedings of MDAFA 2004*, Lecture Notes in Computer Science, Vol. 3599. Springer-Verlag, 2005, pp. 62–76.
19. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. Domain-Specific Languages for Business Process Management: a Case Study. *Proceedings of DSM '09 Workshop of OOPSLA 2009*, Orlando, Florida, USA, pp. 34–40.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 139–149 P.

A Graph Diagram Engine for the Transformation-Driven Architecture

Janis Barzdins, Karlis Cerans, Sergejs Kozlovics, Edgars Rencis, Andris Zarins
Institute of Mathematics and Computer Science, University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
{janis.barzdins, karlis.cerans, sergejs.kozlovics, edgars.rencis, andris.zarins}@lumii.lv

The transformation driven architecture (TDA) is a system building (in particular, tool building) approach that is based on model transformations, interface metamodels with corresponding engines, and event/command mechanism. This paper describes a metamodel and the corresponding engine for graph diagram presentations within TDA. The facilities of the metamodel and the engine include static diagram presentations, as well as graph diagram animations.

Keywords: transformation-driven architecture, model transformations, metamodels, graph diagrams, diagram animation, modeling tools.

1 Introduction

The increasing variety of metamodel-based tools such as MetaEdit [1], Eclipse GMF [2], Microsoft DSL Tools [3], DiaGen/DiaMeta [4] and METAclipse [5] has lead to study of principles behind tool architecture. Metamodel-based tools allow domain data to be represented in a graphical form according to some (perhaps, implicit) presentation metamodel. In [6] we have developed an approach called the *Transformation-Driven Architecture* (TDA), where not just one, but several presentation metamodels are allowed. The link between domain and presentation models within a modeling tool is established by means of model transformations.

Since a presentation model is not yet the end interface that can be presented to the user, some engine is needed to construct the corresponding diagram itself from the instance of the presentation metamodel. Presentation engines form an essential part of the TDA.

Developing a presentation engine and the corresponding metamodel may be a non-trivial task yet when implemented, the corresponding engine can be reused in several tools built upon the TDA.

In this paper a metamodel for graph diagram presentations within TDA and the corresponding engine for drawing/editing graph diagrams is presented. The metamodel along with the engine is a further development based on previous authors' work [7] by fully elaborating the metamodel and putting it within the context of TDA. The graph diagram animation facilities are also newly sketched here.

The paper is organized as follows. The next section lists some ideas of the TDA and explains how the proposed Graph Diagram Engine can be integrated within the TDA Framework. In Sect. 3 the Graph Diagram Metamodel and the Graph Diagram Engine are explained. Sect. 4 presents a way of implementing animation mechanism for graph diagrams. Finally, Sect. 5 concludes the paper.

The short version of the concepts presented in this paper is published in [8]. This is an extended version of [8] and can be presented as a technical report as well.

2 The Essence of the Transformation-Driven Architecture

The Transformation-Driven Architecture [6] is a metamodel-based system (in particular, tool) building approach, where the system metamodel consists of one or more presentation metamodels served by the corresponding engines and the (optional) Domain Metamodel. There is also the Core Metamodel (fixed) with the corresponding Head Engine. Model transformations are used for linking instances of the mentioned metamodels (see Fig. 1).

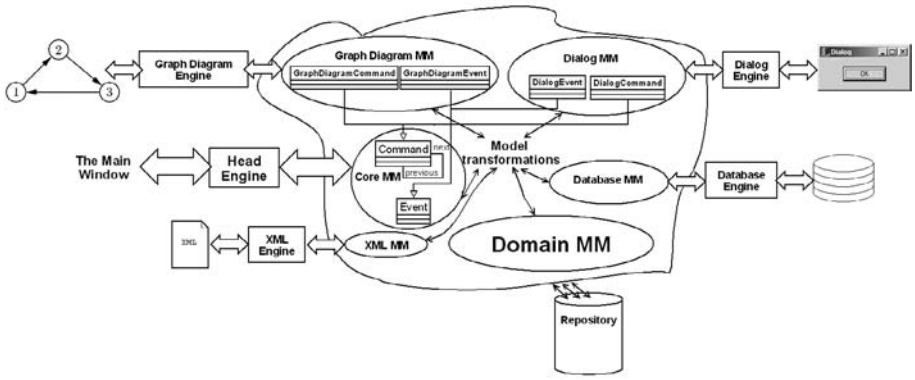


Fig. 1. Metamodels and engines in transformation-driven architecture

There is an *Event* class in the metamodel whose singleton subclasses correspond to the actions the user may perform on a particular diagram and that are understood by a number of engines. Upon observing a current event, engine invokes the event's transformation that is responsible for concrete tool's "business logic" in response to this event. The *Command* class describes the requests (commands) that the tool transformations can issue for an engine. There may be several commands issued by a single tool transformation.

The Head Engine is a special engine, whose role is to provide services for transformations as well as for presentation engines. For instance, when in a presentation engine a user event (such as a mouse click) occurs, the Head Engine may be asked to call the corresponding transformation for handling this event. A transformation may give commands to presentation engines. The Core Metamodel contains classes *Event* and *Command*, and the Head Engine is used as an event/command manager.

TDA has its own framework that comes with the built-in Head Engine (serving the Core Metamodel) and a number of predefined pluggable engines (the Graph Diagram Engine is one of them). Other presentation engines may also be written and plugged-in, as needed. The TDA framework is common to all the tools built upon the TDA. The framework is brought to life by means of transformations. One can choose between writing different transformations for different tools and writing one configurable transformation covering several tools.

3 Graph Diagram Metamodel and Graph Diagram Engine

In the course of time, the graph diagram metamodel has been evolving and providing more and more new facilities. As a result, the physical amount of metamodeling elements (classes, attributes, associations) has significantly increased and representing the whole metamodel visually is a tricky thing to do now. Therefore, in this section, the whole graph diagram metamodel is divided in several parts and each part is discussed in a separate subsection. From here on – if the role name for some association is not mentioned in a metamodel, it is assumed to be default, i.e., the same as the class name with the first letter in lower case.

The graph diagram engine is responsible for visualizing instances of the Graph Diagram Metamodel. The engine is developed on the basis of graphical engines for GRADE tools family [9]. The engine relies on advanced graph layout algorithms [10, 11] as well as effective internal diagram representation structures allowing to handle the visualization tasks efficiently even for large diagrams.

The purpose of the Graph Diagram Metamodel is to describe the graph diagramming functionality that can be offered by the Graph Diagram Engine and that is common to a wide range of graphical diagramming tasks that may go beyond any particular domain specific tool, or even the task of domain specific tool building in general. Since providing appropriate abstractions in the Graph Diagram Metamodel can considerably ease the tool definition process on the basis of the Graph Diagram Engine, the design emphasis of the Graph Diagram Metamodel has been on properly separating concerns between "purely graphical" tasks that are to be handled by the Graph Diagram Engine itself and tasks involving "logic" of tools using the engine.

3.1 The Kernel of the Graph Diagram Metamodel

The visual elements of the presentation (see Fig. 2) correspond to the classes *GraphDiagram*, *Element* and *Compartment*. Every graph diagram can consist of elements of several distinct types – *Node*, *Edge*, *Port*, *FreeBox* or *FreeLine*. A port is a

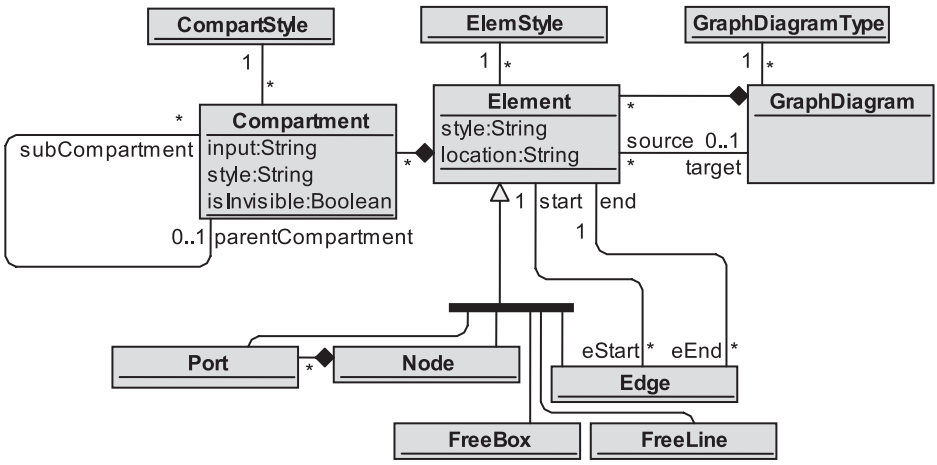


Fig. 2. The kernel of the graph diagram metamodel

small box that can not exist on its own but is instead attached to a *Node*. An edge always contains exactly one start element and one end element as noted by associations *start* and *end*. Free boxes and lines denote visual objects having no layout constraints to be satisfied by the graph diagram engine. Compartments correspond to text fields that may be placed inside nodes or attached to edges and ports. The value of the field is stored in the *input* attribute, and the compartment itself can be made invisible by changing the value of its attribute *isInvisible*.

Instances of the classes mentioned above are diagrams and graphical elements created by the user. Every element and compartment has exactly one style (see classes *ElemStyle* and *CompartStyle*) denoting the visual appearance of the element (or compartment). Instances of classes *ElemStyle* and *CompartStyle* store the default styles of elements and compartments, while the actual style is coded as a string and stored in the *style* attribute of classes *Element* and *Compartment*. Graph diagram engine generates the style string at element or compartment creation time accordingly to the style instance attached to it. It is allowed to change the actual style at runtime (by changing the *style* attribute) while the default style remains unmodified. Likewise, the *location* attribute of *Element* is generated by the graph diagram engine.

In the case of *GraphDiagram*, the class *GraphDiagramType* is attached to it containing both type and style information for the graph diagram. For classes *Element* and *Compartment*, the type information is separated from the style information by making classes *ElemType* and *CompartType* separately from classes *ElemStyle* and *CompartStyle*. The type information goes beyond the scope of this paper and thus will not be discussed in more detail here (see [12] for more details).

Navigation among diagrams can be made according to the metamodel by using the “source – target” association between *Element* and *GraphDiagram*. The other type of hierarchy is the compartment containing hierarchy implemented by the “parentCompartment – subCompartment” association.

3.2 *GraphDiagram* and Its Context

As was stated before, *GraphDiagramType* contains style information for the diagram. This information is put in attributes of the class *GraphDiagramType* (see Fig. 3). When a diagram is being made, one can copy the values of attributes to the attributes of the particular *GraphDiagram*, thus giving it the default style. These values can, however, be changed to assign an individual style to a diagram. The meaning of the style attributes is explained in the next paragraph.

First, a diagram can have a *caption* that will be seen at the title of the diagram window. Next, diagrams background color is coded in *bkgColor* and *layoutMode* and *layoutAlgorithm* imply layout information, for example whether the layout mode is automatic, semi-automatic or completely manual. Value of this attribute is coded as integer 0, 1 or 2, respectively. Finally, *screenZoom* and *printZoom* are responsible for the scale of the diagram.

Next, a set of active elements can be found in a graph diagram. Therefore, a class *Collection* is present here. The active diagram itself can be found following the link from the only instance of the singleton class *CurrentDgrPointer*.

Every *GraphDiagram* has its context defined by classes *Palette*, *PopUpDiagram* and *KeyboardShortcut* and is attached to the diagram through *GraphDiagramType*.

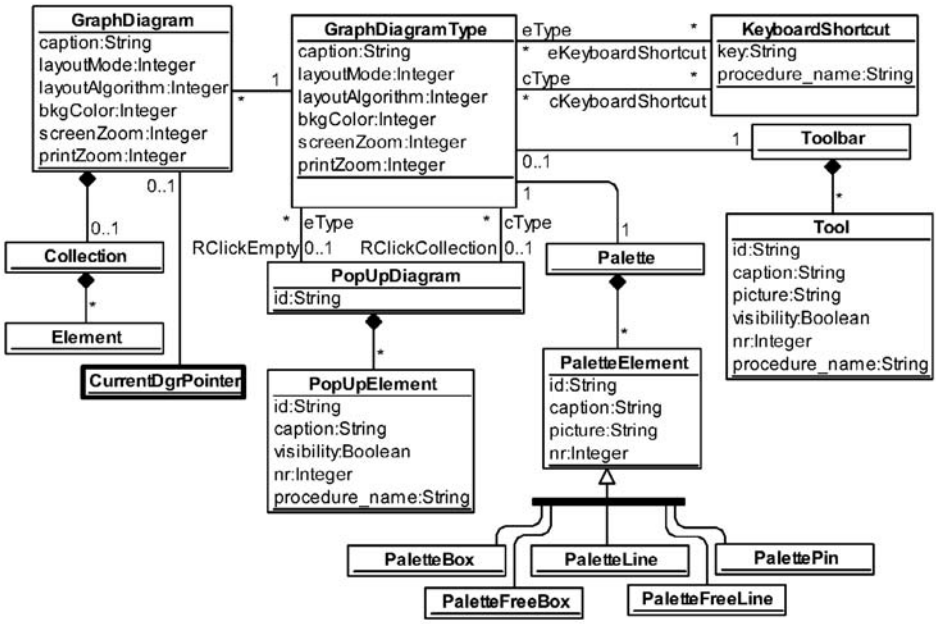


Fig. 3. GraphDiagram and its context

Palette consists of *PaletteElement*s, each of them being a line, a box, a port, a freeline or a free box. Apart from *id* and *caption*, every palette element can have a *picture* (a path to some graphical image) and an *nr* denoting the sequence in the palette.

*Toolbar*s consisting of *Tools* can also be associated with the *GraphDiagramType*. When the graph diagram is being activated, the corresponding toolbars are made visible. Like palette elements, tools can also have an *id*, a *caption*, a *picture* and an *nr*. Moreover, tools can be made invisible by setting the value of the attribute *visibility* to false. The attribute *procedure_name* must contain the name of an existing procedure to be called whenever the user presses the tool in the toolbar. It is assumed that a procedure with such a name can be found in the default dynamic link library provided in the tool (*main.dll*). If the procedure is contained in other dynamic link library than *main.dll*, the library name must be specified as well (following the syntax "<dllName>#<procedureName>").

The metamodel allows the user to specify a *PopUpDiagram* consisting of *PopUpElements*. Usually this kind of menu is activated when the user clicks the right mouse button. Depending on the context, two types of *PopUpDiagrams* can exist – one for the right click in an empty spot of the diagram, and another for the right click on a set of selected elements. Therefore, two associations between classes *GraphDiagramType* and *PopUpDiagram* exist. As it was done before for tools, a calling *procedure_name* must be specified here as well.

Finally, *KeyboardShortcuts* can be added to *GraphDiagramType* providing a possibility to perform some actions using a keyboard. Shortcuts can be specified for both cases – when a set of elements is or is not selected there. For every shortcut, a *key* and a calling *procedure_name* must be specified.

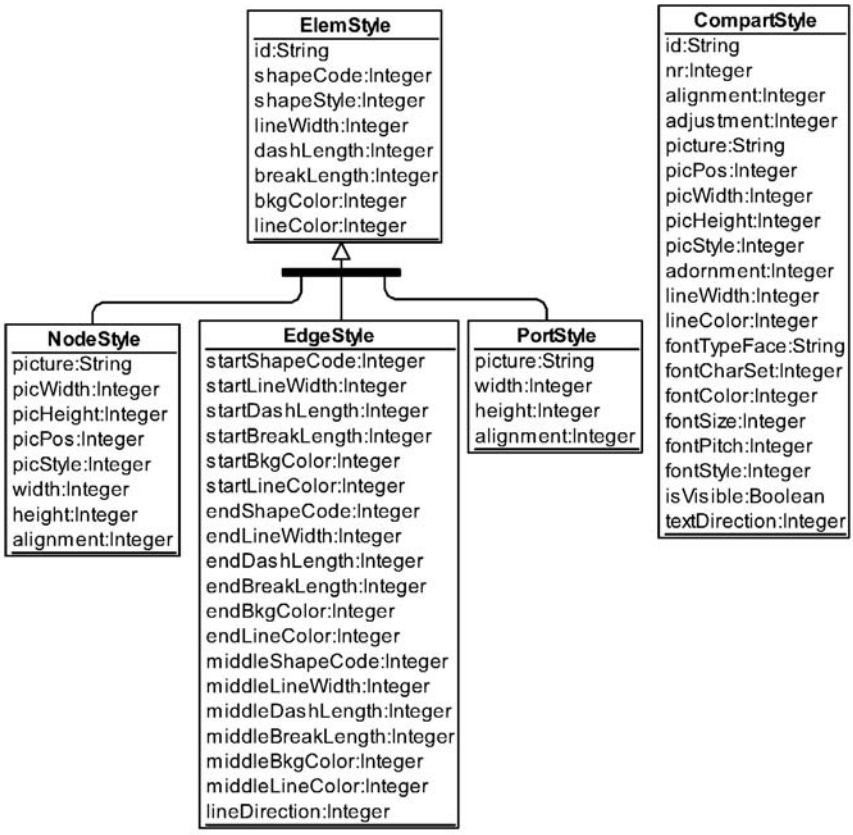


Fig. 4. Element and compartment styles

3.3 Element and Compartment Styles

As mentioned above, instances of classes ElemStyle and CompartStyle contain the default style information for elements and compartments, respectively. The style is a set of several style attributes that can be seen in Fig. 4.

Most of the Element style attribute depend on the particular Element subclass, and thus ElemStyle is divided in three subclasses as well. However, some attributes are generic enough to be attached directly to the superclass. These are *id*, *shapeCode*, *shapeStyle*, *lineWidth*, *dashLength*, *breakLength*, *bkgColor* and *lineColor*.

3.4 Events and Commands

The Graph Diagram Metamodel defines engine-specific events and commands that are subclasses of *Event* and *Command* (see Fig. 5 and 6, events and commands are white classes). Every event and every command during tool runtime is placed within the context defined by the metamodel. For example, the *NewBoxEvent* is attached to the *PaletteBox* with which it is being created, and the *Box* in which it is being put (see associations from class *NewBoxEvent*). All the events together with their context can be seen in Fig. 5, while Fig. 6. represents the commands.

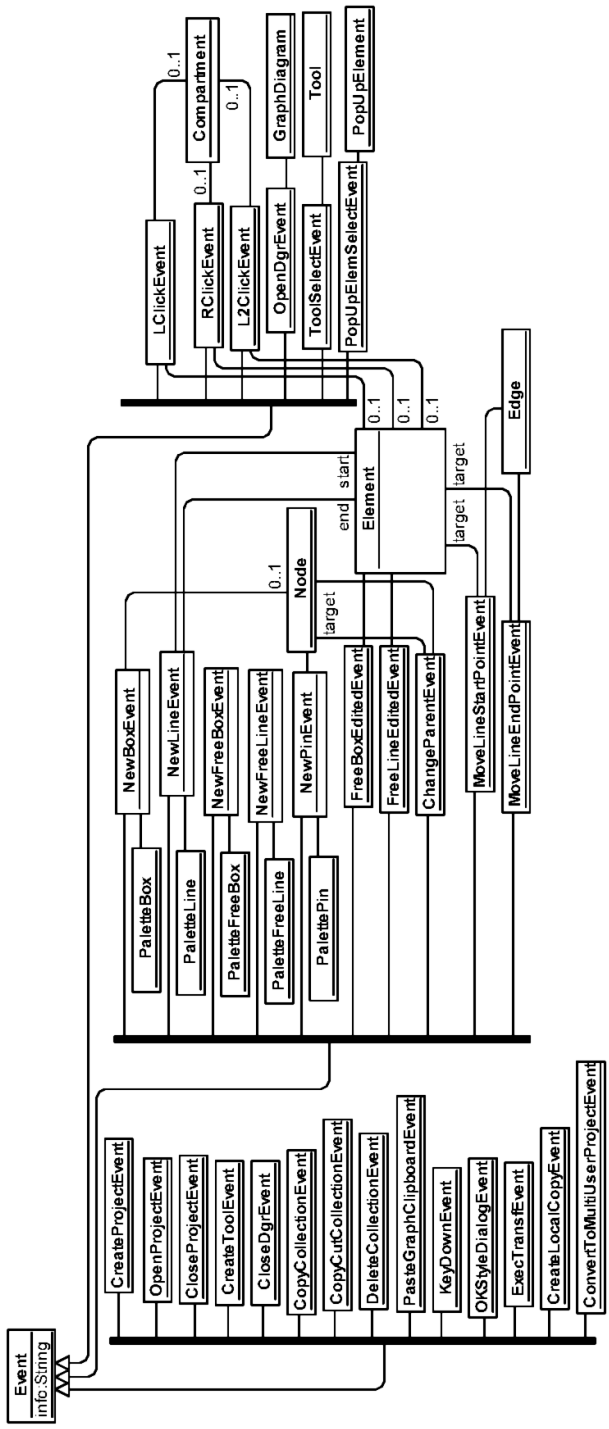


Fig. 5. Events and their context

The meaning of events and commands is mostly inferable from their names. For some events and commands, an additional attribute *info* is needed, i.e., the code of the pressed key is stored in that attribute in the case of *KeyDownEvent*. The multiplicities of

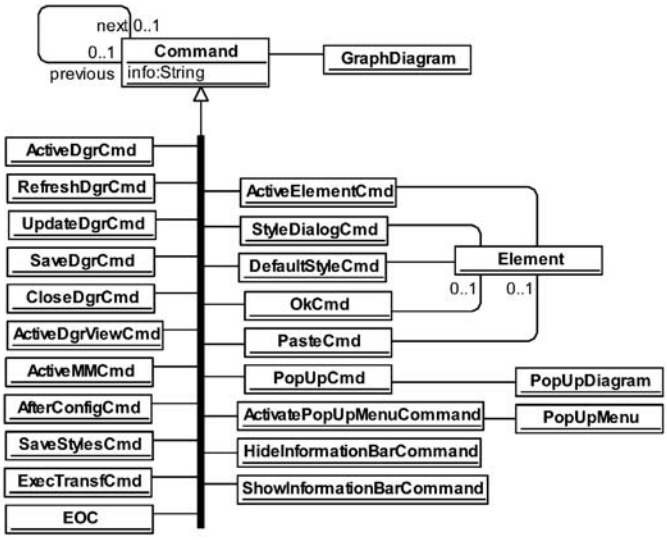


Fig. 6. Commands and their context

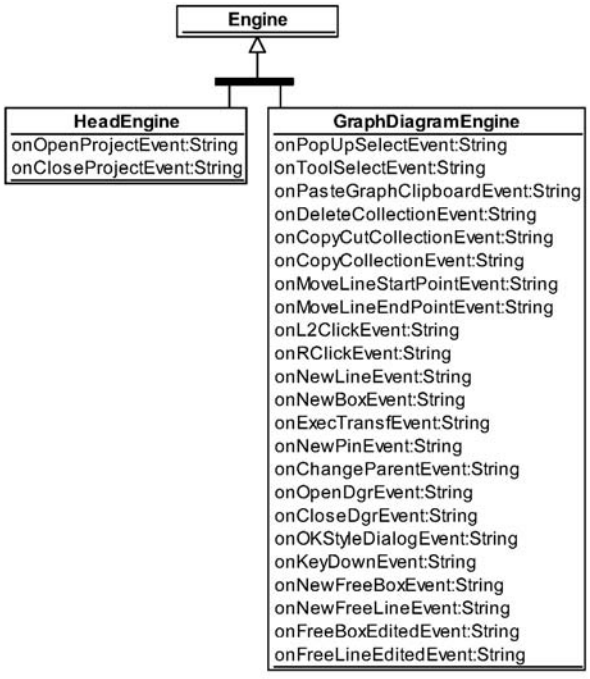


Fig. 7. Engine-specific classes

roles is omitted in figures due to the similarities – the multiplicity is always "0..1" at the event side of an association, and "1" at the other side (if some role does not match the criteria, its multiplicity is noted separately).

The singleton class *GraphDiagramEngine* contains attributes that correspond to engine's events (see Fig. 7). In the beginning a transformation can assign values for these attributes, each value representing the name of the transformation that has to be called when the particular event occurs. In TDA, other singleton subclasses for other engines exist there as well. In Fig. 7, class *HeadEngine* is represented together with its attributes for its two events – *OpenProjectEvent* and *CloseProjectEvent*.

4 Graph Diagram Animation

Although there are several approaches for metamodel-based handling of dynamic multimedia objects that include animations (see, for instance, [13]), our goal here is more specific — to provide simple animation facilities for graph diagrams explained in Section 3. Complex interactive animations (such as animations that can be created in *Microsoft Silverlight* [14] or *Adobe Flash* [15]) are beyond the scope of this approach.

In Fig. 8 we extend the metamodel of graph diagrams by classes for describing graph diagram animations. The animation of graph diagrams is based on the concept of token that is associated with some element (box or line) in a graph diagram. Tokens do

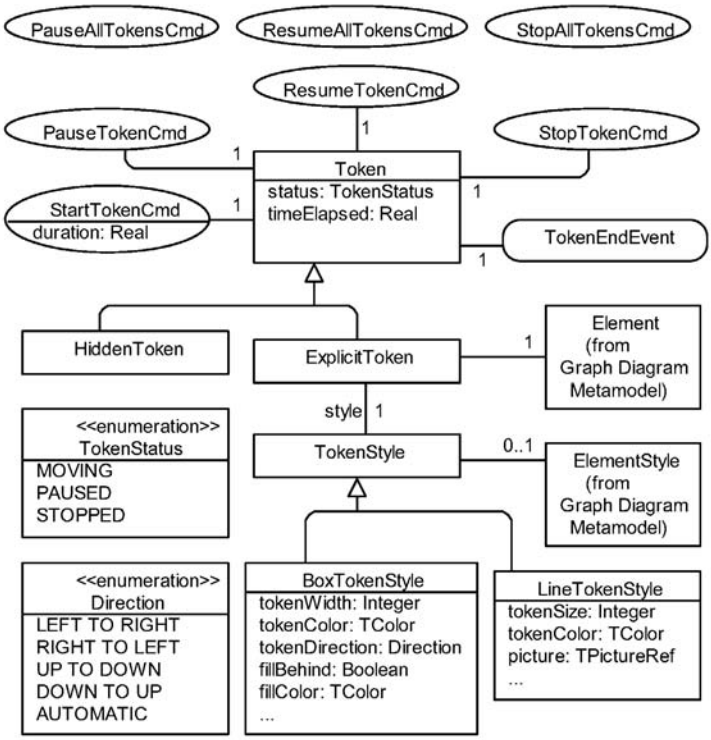


Fig. 8. Adding animation capabilities to the Graph Diagram Metamodel

not imply any semantics, they are used only for managing the animation process. The semantics is up to transformations.

A token is started by *StartTokenCmd* that also specifies its *duration* (how long the token "lives"). There are also commands for starting, stopping, pausing and resuming a token in the diagram, as well as pausing, resuming and stopping all tokens in the diagram. The "end of life" of a token is determined by the presentation engine – at that time it creates a corresponding *EndTokenEvent*. There can be several tokens living concurrently in the diagram.

An *explicit token* is able to simulate the activity of the associated element for the given duration. The visual effect of the simulation is determined by *TokenStyle* instance associated with the token. If an *ElementStyle* instance is associated with the token style, then the animation consists of changing the element style for the token's lifetime. Other options of animation consist of moving a bullet of certain size or some image along the line in the diagram, or animating a box by a line moving across it in certain direction, with or without leaving the trailing part in the specified color. In the case of *AUTOMATIC* direction, the actual line flowing direction is determined by the presentation engine on the basis of the placement of the actual outgoing line from the box. A *hidden token* does not animate any element, it just "lives" for the specified amount of time. Hidden tokens can be useful, e.g., for accounting the global animation time, or for creating certain breakpoints during the animation when the control is transferred to transformations for some semantic actions.

The implementation of animation facilities in our graph diagram engine is currently under development.

5 Conclusions

The Graph Diagram Engine has been successfully implemented in a recent version of transformation-based tool building platform GrTP [7]. The GrTP tool is now being transformed to the TDA framework, which should become publicly available soon. At the moment, the TDA framework consists of two predefined engines (one of them is the Graph Diagram Engine and the other is the Dialog Engine), and the interaction between these engines and model transformations performed by means of commands and events is working quite well. We are working on ameliorating the TDA framework and its engines. One of the research topics here is adding advanced graph diagram layout capabilities to the Graph Diagram Engine. We are also working on implementing diagram animations within the Graph Diagram Engine for TDA.

Several diagram editors (such as class diagram editor and activity diagram editor) have been successfully built using the Graph Diagram Engine. This engine has also been used in [16] and [17]. We are looking forward for applying the TDA and its engines in the Semantic Web domain.

Acknowledgments.

The authors would like to thank (alphabetically) A. Kalnins, L. Lace, R. Liepins, A. Sprogis, R. Zarits and M. Zviedris for their efforts in implementing the concepts presented in this paper.

References

1. MetaEdit+. Available: http://www.metacase.com.
2. A. Shatalin and A. Tikhomirov. *Graphical Modeling Framework Architecture Overview.* Eclipse Modeling Symposium, 2006.
3. S. Cook, G. Jones, S. Kent, A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley, 2007.
4. DiaGen/DiaMeta. Available: http://www.unibw.de/inf2/DiaGen.
5. A. Kalnins, O. Vilitis, E. Celms, E. Kalnina, A. Sostaks, J. Barzdins. Building Tools by Model Transformations in Eclipse. *Proceedings of DSM'07 Workshop of OOPSLA 2007*, Montreal, Canada: Jyvaskyla University Printing House, 2007, pp. 194–207.
6. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. *Proceedings of DSM'08 Workshop of OOPSLA 2008*. Nashville, USA, 2008, pp. 60–63.
7. J. Barzdins, A. Zarins, K. Cerans, A. Kalnins, E. Rencis, L. Lace, R. Liepins, A. Sprogis. GrTP: Transformation-Based Graphical Tool Building Platform. *Proceedings of MDDAUI Workshop of MoDELS 2007*. Nashville, USA, 2007.
8. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proceedings of MDDAUI'09 Workshop of International Conference on Intelligent User Interfaces 2009*, Sanibel Island, Florida, USA, 2009, pp. 29–32.
9. GRADE tools. Available: http://www.gradetools.com.
10. P. Kikusts, P. Rucevskis. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Proceedings of Graph Drawing '95*, LNCS, vol. 1027, Springer-Verlag, 1996, pp. 361–364.
11. K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proceedings of Latvian Academy of Sciences*, Section B, vol. 55, no. 1, 2001, pp. 43–51.
12. J. Barzdins, K. Cerans, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, Z. Zarins. An MDE-Based Graphical Tool Building Framework. This publication, pp 121–139.
13. A. Pleuss, A. Vitzthum, H. Hussmann. Integrating Heterogeneous Tools into Model-Centric Development of Interactive Application. *MoDELS 2007*, LNCS, vol. 4735, Springer-Verlag, 2007, pp. 241–355.
14. Silverlight Animation Overwiew. *MSDN*, Microsoft Corp. Available: http://msdn.microsoft.com/en-us/library/cc189019(vs.95).aspx.
15. Adobe Flash. Available: http://www.adobe.com/products/flash.
16. G. Barzdins, E. Liepins, M. Veilande, M. Zviedris. Semantic Latvia Approach in the Medical Domain. In: H-M. Haav, A. Kalja (eds.), *Proceedings of the 8th International Baltic Conference (Baltic DB & IS2008)*. June 2–5, Tallin, Estonia. Tallinn University of Technology Press, 2008, pp. 89–102.
17. J. Barzdins, K. Cerans, A. Kalnins, M. Grasmanis, S. Kozlovics, L. Lace, R. Liepins, E. Rencis, A. Sprogis, A. Zarins. Domain-Specific Languages for Business Process Management: a Case Study. *Proceedings of DSM'09 Workshop of OOPSLA 2009*. Orlando, Florida, USA, 2009, pp. 34–40.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 151–170 P.

A Dialog Engine Metamodel for the Transformation-Driven Architecture[1]

Sergejs Kozlovics
University of Latvia, Faculty of Computing, Raina bulv. 19, Riga, LV-1586, Latvia
Institute of Mathematics and Computer Science, University of Latvia
Raina bulv. 29, Riga, LV-1459, Latvia
sergejs.kozlovics@lumii.lv

Many metamodel-based tools provide only limited features for specifying dialog windows by means of the metamodel. Is there a way of specifying complex dialogs while still using the metamodel-based ap-proach? The metamodel proposed in this paper permits specifying rather complex dialogs in a simple and intuitive way. It has been successfully used within the context of the transformation-driven architecture (TDA).

Keywords: dialogs, GUI, dialog engine, TDA, transformation-driven architecture.

1 Introduction

It is hard to imagine a graphical tool without a graphical user interface (GUI). Classical dialog boxes with input fields and buttons are well-known and accustomed. We will not invent the wheel in this paper; we concentrate on already familiar dialog windows. We present a simple yet expressive metamodel for describing such dialogs and comment on the corresponding engine for handling dialogs specified by means of that metamodel. Since this is a metamodel, it may be used in the world of model transformations. Our metamodel has already been successfully used with the transformation-driven architecture [1], which is a system-building approach that incorporates model transformations and metamodels with their engines.

The above mentioned dialog metamodel is the main contribution of this paper. The metamodel has been developed in such a way that, given an instance of it, the dialog engine is able to automatically create the real dialog box at runtime and to show it to the user. One of the important features of the metamodel is the possibility to specify the layout of dialog elements. If we sketch a dialog box on a sheet of paper, we usually do not bother about exact coordinates, but we think about the layout and grouping of components and aesthetics. The same kind of layout information is expected in instances of the proposed metamodel.

One may be interested whether the metamodel uses exact sizes and/or co-ordinates for components. A dilemma arises: on the one hand, exact coordinates may guide the dialog engine on the desired sizes of components in case the components with the default (or in some way calculated) sizes are not aesthetic. On the other hand, the system font

[1] This research is partially supported by European Social Fund.

and depth-per-inch (DPI) settings may differ from one computer to another; thus, it is preferable to avoid exact sizes and coordinates. The features of our metamodel may help to deal with this dilemma.

– The metamodel permits specifying absolute sizes, including minimal, preferred and maximal. Yet all these sizes are optional, and when they are not specified, the dialog engine selects the values itself. These values are suitable for the specific platform and widget toolkit to provide nice look and feel and to allow resizable components to be resized. When applicable, the DPI settings and the size of the font used are taken into account.
– The metamodel also permits specifying relative sizes of components. One may require that the input field has to be two times wider than the button B or that the aspect ratio of the dialog form should be 4 : 3.

Another question that arises is whether the proposed metamodel is bound to some specific widget toolkit. Basic components such as buttons, input fields and check boxes can be found in a wide variety of widget toolkits. We include these basic components as well as other more complex but also popular components like the table and the tree in the metamodel. There should be no problem to use those toolkits for handling instances of our metamodel. The metamodel certainly can be augmented to support other components as well. We will show how that can be done in this paper.

We try to explain the semantics of the proposed metamodel by means of graphical images in this paper, which is an interesting feature, but textual explanations are also used.

Our metamodel utilizes the event/command mechanism of the transformation-driven architecture. We start with a brief explanation of TDA (Section 2) before presenting and explaining the dialog metamodel with its semantics (Section 3). Then we explain how additional components may be included in the metamodel and present two non-trivial metamodels for the tree component and the table component (Section 4). Before the conclusion, we reference some related work (Sections 5–7).

2 The Transformation-Driven Architecture as the Context

The transformation-driven architecture (TDA) [1] is an approach to building systems in general and tools in particular (Fig. 1). The idea of the TDA is very simple. There is a system metamodel (the largest bubble in Fig. 1) which merges interface metamodels and the core metamodel and optional domain meta-model. Interface metamodels are processed by the corresponding engines. While processing instances of interface metamodels, engines, for example, may create a graphical presentation of the data represented by these instances. The data are stored in the Repository, which is accessed through the Repository Proxy that provides the UNDO/REDO functionality. All this is brought to life by model transformations.

Transformations can communicate with engines by means of the event/command mechanism. The Core Metamodel served by the Head Engine provides classes *Event* and *Command* as well as the class *Engine*. We will not describe the Core Metamodel in detail. However, let us look at the design pattern which uses the above mentioned classes to define events and commands in engine metamodels.

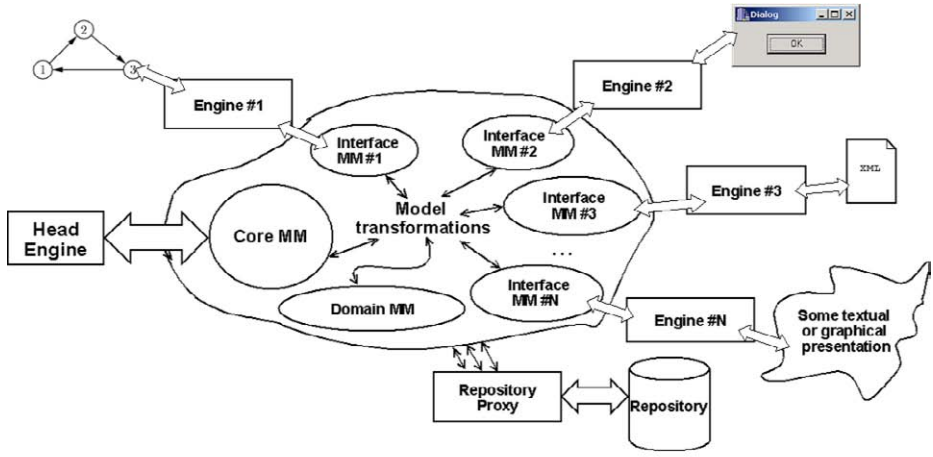


Fig. 1. The essence of transformation-driven architecture

When some event (e.g., a mouse click) occurs in the engine, it may be useful to notify the transformation about that event. For all such events, the engine's metamodel should contain some class derived from the Event class (Fig. 2 (a)). When an event occurs, the engine should create an instance of the corresponding *Event* subclass, set its attributes (if needed), create links from the event object to other object(s) (if needed), and ask the Head Engine to call the corresponding transformation for handling that event.

However, we may wonder how does the Head Engine know what transformation to call? If the event has one or more objects associated with it (the context), one of these objects (event source) may have an attribute called *on<EventName>*, whose value would be the name of the transformation to be called. If the event does not have a context, this attribute may be defined in the corresponding engine's class (Fig. 2 (b)). The values for the *on<EventName>* attributes are supposed to be specified by transformations and/or by the trans-formation programmer to handle the required events in suitable way.

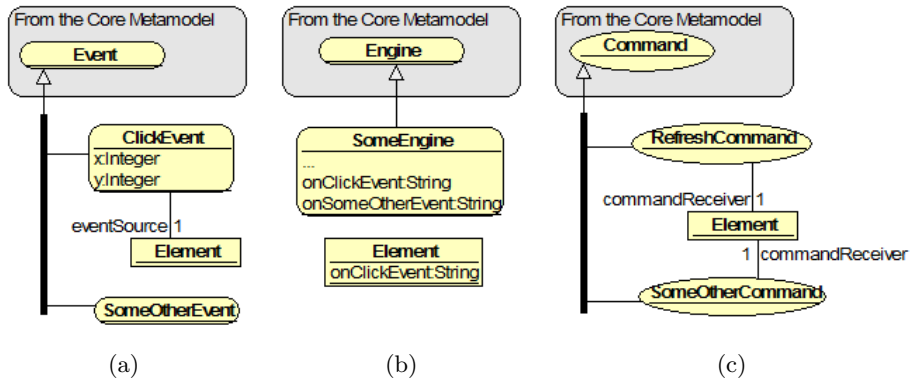


Fig. 2. (a) Defining events as *Event* subclasses; (b) defining attributes to specify event handling transformations; (c) defining commands as *Command* subclasses

If more than one class contains the *on<EventName>* attribute, the search order for finding the valid transformation name should be defined. As shown in Fig. 2 (b), the *onClickEvent* attribute of the event source element has to be checked first. If the value is not set, the attribute of the *SomeEngine* instance should be checked. This permits handling the *ClickEvent* for different elements differently by specifying transformations in the *Elements onClickEvent* attributes, while also permitting handling the *ClickEvent* by the same transformation for all elements by specifying the value for *onClickEvent* in the *SomeEngine* instance and leaving such attributes empty for the elements.

While events serve as a communication bridge in the direction from engines to transformations, commands serve communication in the opposite direction. Commands are derived from the *Command* class in the Core Metamodel (Fig. 2 (c)). The transformation may create command instances (the context may also be specified), and leave them in the repository. When the transformation finishes, the commands are being sent to the corresponding engines. What exactly has to be called may be considered internal information; thus, there are no attributes in the metamodel for storing the name of the function to execute commands.

Having in mind the approach to defining events and commands just described, let us take a look at the dialog metamodel.

Dialog metamodel instances are usually created by transformations. The data may be collected from the domain model or from interface models of engines and then presented as a dialog window. After the dialog window is closed, the transformation brings the data entered or modified by the user from the dialog window to the corresponding places in the system model. The transformation may also handle inputs in the dialog window, not waiting for the dialog window to be closed.

3 The Dialog Engine Metamodel and Its Semantics

3.1 At the First Glance

Perhaps the main notion in the Dialog Engine Metamodel (Fig. 3) is the notion of the component (see the abstract class *Component*). Components are graphical elements such as the *Label, CheckBox, InputField,* and so on (see direct *Component* subclasses on the left in Fig. 3).

Another important notion in the metamodel is the notion of the container (see the abstract class *Container*). Containers are special components which may contain other components (see the generalization and the composition between the *Component* and the *Container*). Familiar containers that can be found in the metamodel are the *Form, GroupBox, TabContainer,* and *Tab.* Such containers are used to group components visually. However, there are other containers in the rounded rectangle on the right. The types of those containers specify how the components are laid out inside them. This will be discussed in more detail below.

[1] In order not to overload Fig. 3, we do not show generalizations for subclasses of *DialogEngineCommand* and *DialogEngineEvent.* Instead, we use rounded rectangles and ellipses to show which classes are events and which are commands.

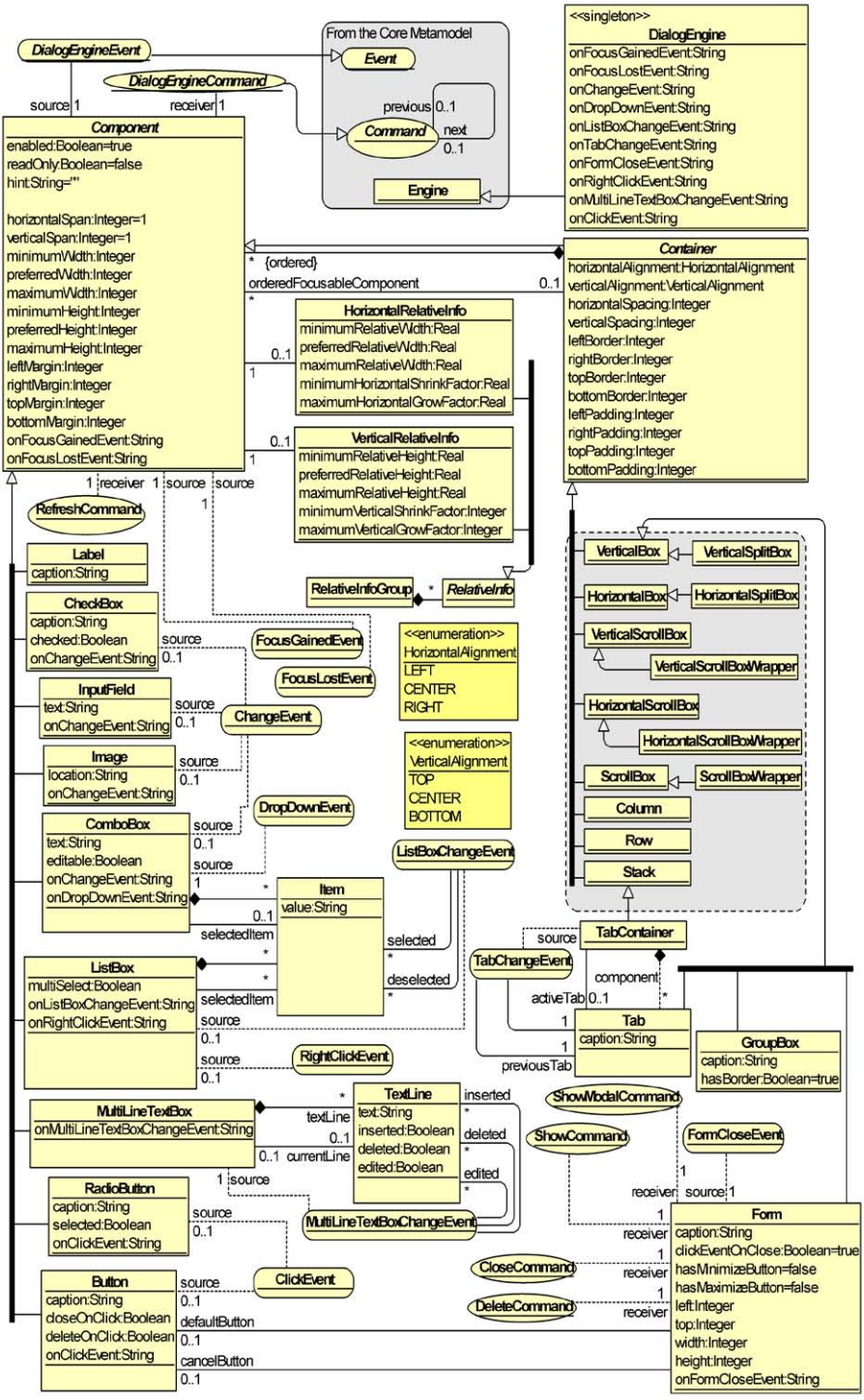


Fig. 3. The Dialog Engine Metamodel

There are certain events and commands that may be assigned to components. As we can see at the top of Fig. 3, the Dialog Engine Metamodel follows the design pattern mentioned in Sect. 2. Each *DialogEngineCommand* and *DialogEngineEvent*[1] has a context which must include a component to which the command/event refers to; see roles *source* and *receiver* of the class *Component.* For instance, a *ClickEvent* may be linked to a *Button* or to a *RadioButton.*

The class *DialogEngine* contains *on<EventName>* attributes for specifying event handling transformations. Note that components also have such attributes for the events they may refer to; see, for example, the *Button*'s *onClick-Event* attribute.

After the first glance at the metamodel, let us look closer at the basic components and the layout specification.

3.2 The Basic Components with Their Events and Commands

We start from the features common to all components.

There is a *readOnly* attribute in the *Component* class. Different components may implement their semantics differently, but the main meaning is that if *readOnly*= **true**, the user is not able to change the value of the component.

The *readOnly* attribute is recursive, i.e., its value applies to the component itself, and, in case the component is a container, to the child components, and so on. In case another value is specified for some child/descendant, that new value is propagated recursively. Such recursive semantics of the *readOnly* attribute may be useful when a non-editable form has to be shown, for instance, in read-only mode, or when the user does not have access to change the values in the form. It suffices to specify *readOnly*= **true** only for the *Form* instance, leaving *readOnly* undefined for all other components in that form.

The *Component* class also has two events: the *FocusGainedEvent* and the *FocusLostEvent.* These events occur when the component gains or loses the input focus. Not all components may produce such events (e.g., the *Label* does not).

Furthermore, a *RefreshCommand* may be associated with the component. The semantics of this command is to re-read value(s) for the component from the repository.[2] If the component is a container, then, besides refreshing the component itself, the *RefreshCommand* reloads its descendants — one may think of it as if the previous descendants have been deleted, and the current descendants have been added.

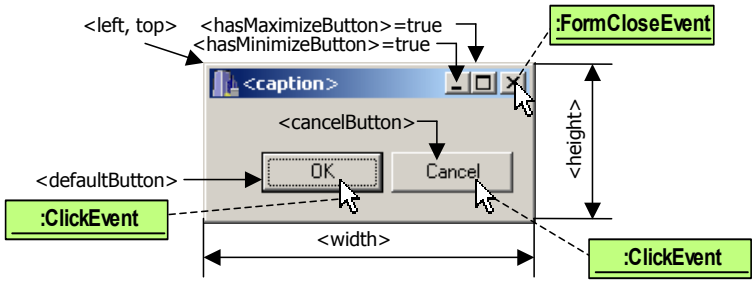
Now we are going to look at the semantics of basic dialog components. Let us explain the semantics graphically. We use the following notation: the names of properties (attributes or roles) in angular brackets and the arrow points to the graphical presentation of the value of that property (the *<caption>*, *<text>* and *<value>* attributes are present at places without a pointing arrow; one or more asterisks may be added to denote that the values correspond to different instances). Events are shown as instances in rectangles, and the lines point to user actions that generate these events.[3]

Figures 4 and 5 illustrate the semantics of the visual components that can be found in the metamodel.

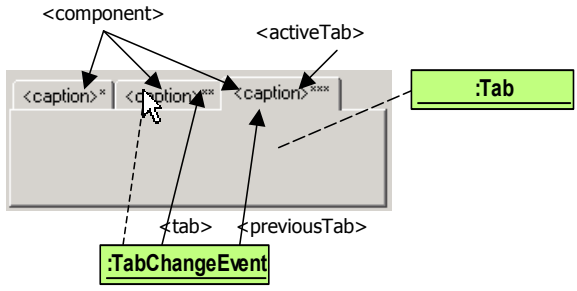
---

[2] Internally this action may be performed either on the same graphical control or on a new control that replaces the previous one.

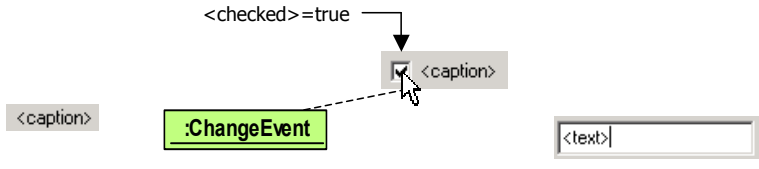
[3] We show only mouse actions.



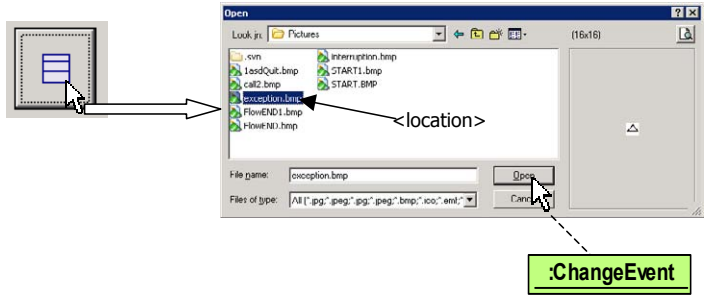
(a)



(b)



(c) (d) (e)



(f)

Fig. 4. The semantics for the (a) *Form*, (b) *TabContainer* and Tab, (c) *Label*, (d) *CheckBox*, (e) *InputField* and (f) *Image* classes

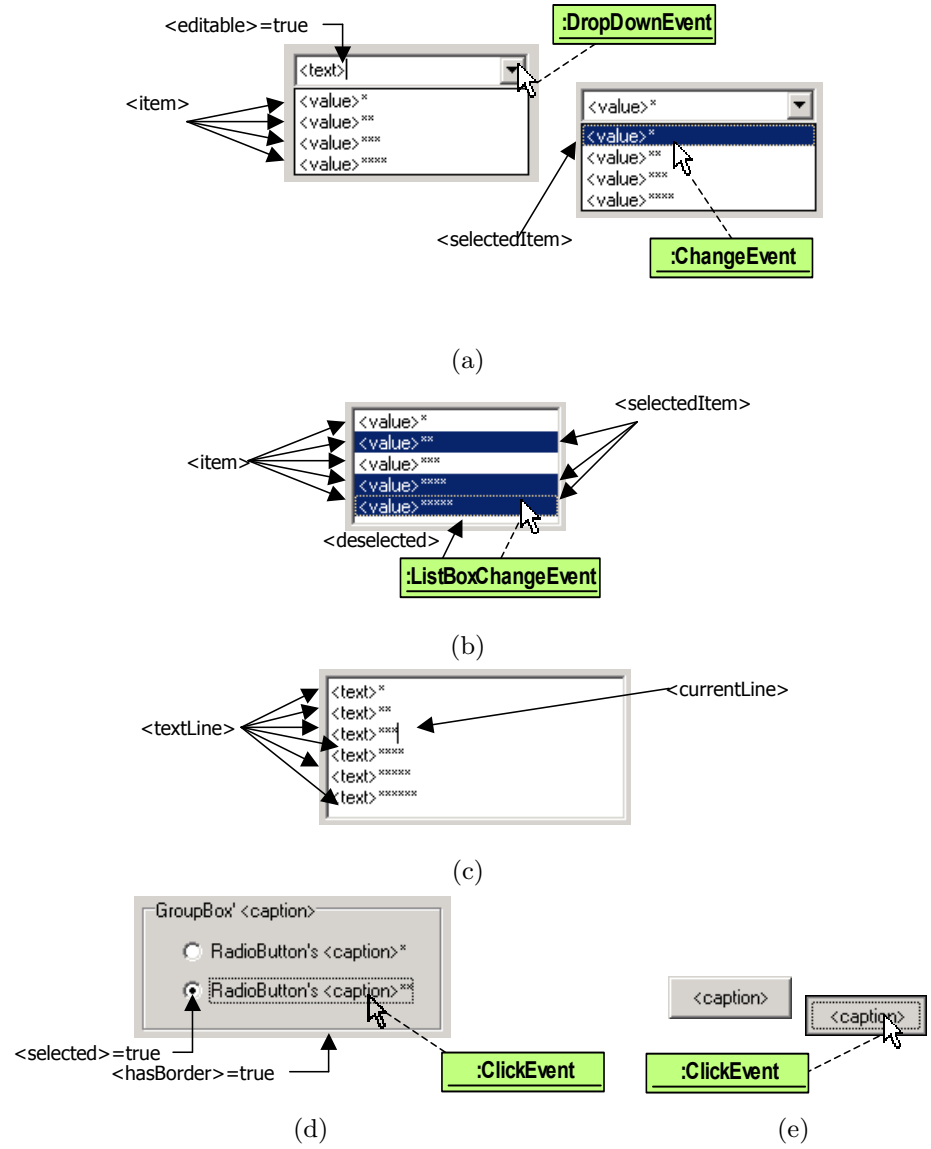


Fig. 5. The semantics for the (a) *ComboBox*, (b) *ListBox*, (c) *MultiLineTextBox*, (d) *GroupBox* and *RadioButton*, and (e) *Button* classes

Let us explain some aspects not explicitly shown in Fig. 4 and 5.

– The semantics for the event produced when the user clicks the “close” ("X") button of the form (Fig. 4 (a)) is as follows: if <*clickEventOnClose*> = **false** or there is neither the “cancel”, nor “default” button[4], the *FormCloseEvent* is generated. Otherwise, a *ClickEvent* for the “cancel” button is generated. In case there is no “cancel” button, a *ClickEvent* is attached to the “default” button (this is useful when the form contains only one "OK" button).

– The *MultiLineTextBox* (Fig. 5 (c)) continues to be linked to those *TextLine*s which have already been deleted from the screen and have *deleted* = **true**. This may be useful when some actions have to be performed after some *TextLine*s have been deleted, but these *TextLine*s are linked to other objects.

– *RadioButtons* (Fig. 5 (d)) usually are grouped together. Only one of the radio buttons in the group may be selected at the same time. We assume that the group consists of the radio buttons that are in the same visible container.[5]

– In case *readOnly* = **true**[6], the values of the following properties are blocked and the user is not able to change them as normally (when *readOnly* = **false)**.

Component	Property
CheckBox	*checked*
InputField	*text*
ComboBox	*text* and *selectedItem*
ListBox	*selectedItem*
MultiLineTextBox	*textLine* (also the *text* property of *TextLines)*
RadioButton	*selected*

– There are the following commands for the *Form:*

- *ShowCommand* is used to show the modeless form; after the command is executed, the form remains on the screen;
- *ShowModalCommand* is used to show the modal form; the command is being executed until the form is closed (see *CloseCommand* below);
- *CloseCommand* is used to close the dialog window;
- *DeleteCommand* is used to cascade delete the form with its containers and components from the repository.

3.3 Laying Out the Components

In this sub-section, we first describe the classes contained within the rounded rectangle in Fig. 3. Then we describe how absolute and relative sizes may be specified.

Container types. When imaging a dialog box, we assume that all begins with the form which is the top-level (root) container. This container can be logically divided into several parts or cells. Each cell may be divided again, and so on, recursively. Some

[4] The “default” button is the button which is automatically clicked when the user presses the "Enter" key; the “cancel” button is automatically clicked when the user presses the "Esc" key.

[5] Invisible containers which are "skipped" when forming a group of radio buttons are *VerticalBox*, *HorizontalBox*, *Column*, *Row*, *Stack*, see Sect. 3.3.

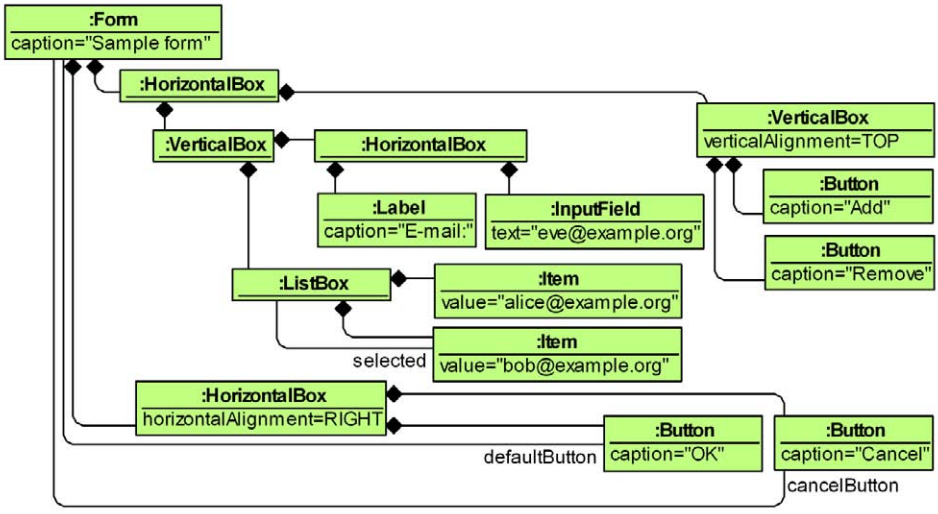
[6] The attribute *readOnly* is defined for all components in the common superclass *Component.*

cells are occupied by visible components or containers, while other are simple invisible "frames" or "borders".

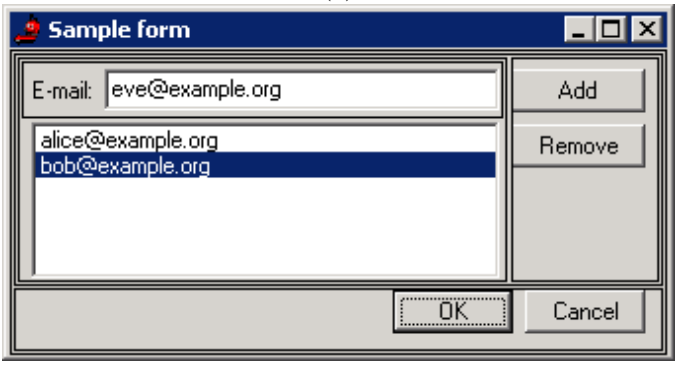
Each cell has its width and height. However, if the cell is occupied by a (visible) scrollable container, there are also interior width and interior height that correspond to the scrollable area where the child components are placed.

In the metamodel, the notion of the cell is represented by the *Container* class. The way the cell (or the container) is divided into other cells determines the type of the container.

Two obvious ways of dividing a cell is dividing it into horizontal and vertical boxes. In the first case horizontal child cells are placed vertically one on another; thus, we call the parent cell the *VerticalBox*. In the second case child cells are placed horizontally from left to right; thus, we call the parent cell the *HorizontalBox*. Fig. 6 shows a sample



(a)



(b)

Fig. 6. (a) An instance of the Dialog Engine Metamodel for the sample form; (b) the sample form: the rectangles (in reality invisible, but shown here) outline horizontal and vertical boxes

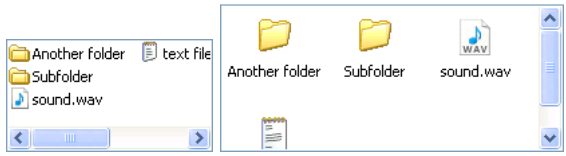


Fig. 7. Examples of (a) a horizontally scrollable box and (b) a vertically scrollable box

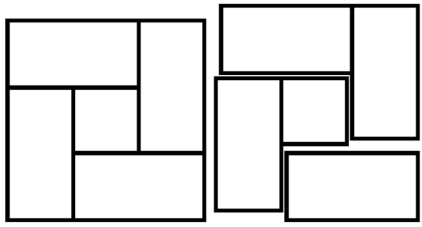


Fig. 8. (a) An example of five containers that cannot be laid out using horizontal and vertical boxes only; (b) the arrangement of the same five containers using rows – the first row contains two components: the first one spans two columns (horizontally) and the second spans two rows (vertically). The first component of the second row spans two rows; neither rows nor columns are spanned by the second component (this is the default behavior). The third row has only one component that spans two columns.

form which uses horizontal and vertical boxes for the layout.[7] The form itself is a vertical box as well. A container of some other type may be placed on the form when needed, thus converting the form from a vertical box to the other container type.

However, one of the scrollbars adds the possibility for the children to move to the second line (in a vertically scrollable box, *VerticalScrollBox*, Fig. 7 (a)) or to the second column (in a horizontally scrollable box, *HorizontalScrollBox*, Fig. 7 (b)) and so on. In the case of the horizontally scrollable box, the children are laid out as text in newspaper columns.

When there are both horizontal and vertical scrollbars, we call such a container the *ScrollBoxContainer.* It is similar to the *VerticalBox*, but in case the internal area of the *ScrollBoxContainer* is scrolled out of the visible area, one or both scrollbars appear, and the internal area can be scrolled. We use one scroll box container (vertical) instead of two (horizontal and vertical) since a horizontal box may be put inside the scroll box when needed.

The container types mentioned above, however, do not permit creating such structures as in Fig. 8 (a). Moreover, they do not provide a way of creating a grid-like structure to be able to align components to grid. Thus, we add the following two container types: the *Row* and the *Column.* Rows and columns are for creating a grid-like layout; they are not for scrolling, but they may be put into a scroll box if needed.

[7] This figure also shows an example of how to specify dialog boxes by means of the metamodel from Fig. 3.

We do not add the container for representing the table. We assume that neighboring rows (or columns) form the necessary grid-like structure, i.e., the compo-nents of neighboring rows (or columns) are aligned to grid. Thus, their parent container may be considered to be a container representing the table.

However, having a grid, we should allow the components to be able to span several rows and/or columns (see attributes *horizontalSpan* and *verticalSpan* of the *Component* class). This permits creating layout structures such as in Fig. 8.

One more container type is needed to implement the tabs. Since tabs occupy the same space, we think that the components are put one over another like cards. We introduce the *Stack* container, where all the children occupy the same space.

Table 1 summarizes the types of the containers we described and tells which containers are visible and which are invisible. In case a visible analog for an invisible container is required, a *GroupBox* can be used as a visible parent of the invisible container.

Table 1

Container types

Invisible container types	Visible container types
VerticalBox	*VerticalSplitBox*
HirizontalBox	*HorizontalSpllitBox*
Column	*VerticalScrollBox*
Row	*VerticalScrollBoxWrapper*
Stack	*HorizontalScrollBox*
	HorizontalScrollBoxWrapper
	ScrollBox
	ScrollBoxWrapper

We find that the container types listed in Table 1 permit laying out components in many ways and cover not only simple layouts, but also layouts that are complex enough.

Absolute Sizes. The meaning for the six attributes from *minimumWidth* to *maximumHeight* of the *Component* class is obvious. The *maximumWidth* and *maximumHeight* values are allowed to increase (minimally) to satisfy other constraints. Thus, if a component has *maximumWidth* = 0, then its real width would be as small as possible.

The *Container*'s attributes *horizontalAlignment* and *verticalAlignment* refer to the children. If a child is resizable, it is attached to the border of the parent container. However, if the child reaches its maximum width (or height), it is aligned according to the *horizontalAlignment* value (or *verticalAlignment* value). If there are several children in a horizontal box, *horizontalAlignment* refers to all of them as one component. The same is true for the vertical box and the *verticalAlignment* attribute.

The meaning of attributes for specifying margins (in the *Component* class) as well as for specifying borders, padding and spacing (in *Container* class) is revealed in Fig. 9. The margins specify the extra space outside the component (i.e., this space is not considered to be part of component's width or height). The borders in the Container class

specify the size of the border (e.g., bevel). These values are parts of the component's width and height. Padding is like a margin but inside the area it is bounded by the border. In non-scrollable containers the notions of padding and border are the same. However, in scrollable containers, the border is outside the scrollable area, while the padding is inside.

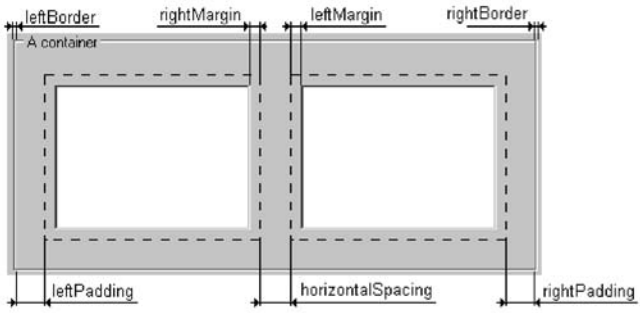


Fig. 9. An example illustrating what values for margins, borders, paddings and spacings mean

Relative sizes. Relative dimensions are related to the notion of the relative information group (see class *RelativeInfoGroup).* The group consists of widths and/or heights which relatively depend on each other. For example, we may group the widths of three components to specify the ratio for the widths as 2 : 3 : 4. There is no need for a particular width or height of some component to be in several groups, since in this case the groups depend on each other and may be replaced by one group by adjusting the ratio.

To specify the relative width ratio 2 : 3 : 4 between three components, we attach a *HorizontalRelativeInfo* instance to each of these components and set the values of the *preferredRelativeWidth* to 2, 3 and 4, respectively. Finally, the three *HorizontalRelativeInfo* instances are linked to the *RelativeInfoGroup* instance to form a group. The relative heights are specified in the same way.

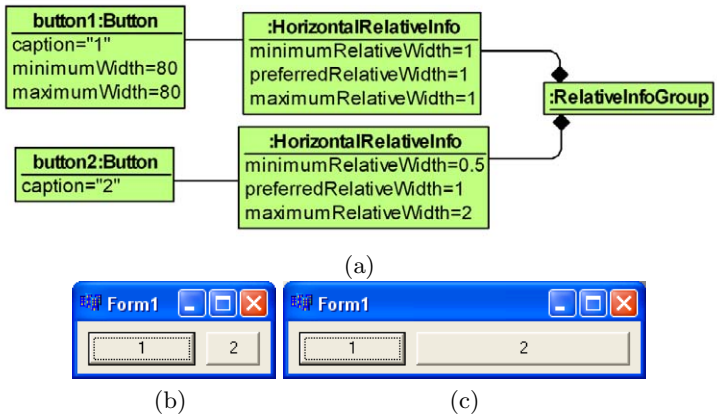


Fig. 10. An instance (a) demonstrating the usage of minimum and maximum relative sizes; the minimum (b) and the maximum (c) sizes of Button 2

The minimum and maximum relative width and height are useful in resizing. An example is given in Fig. 10. Button 1 is not resizable, and the width of Button 2 is preferred to be the same as of Button 1. However, if the preferred ratio cannot be achieved, Button 2 is allowed to be up to 2 times wider or shorter than Button 1.

4 Adding Components: the Tree and the Table

In order to add a component to the Dialog Engine, two steps have to be performed.

1 Developing a metamodel for the component for which the component class is a direct or indirect subclass of *Component.*

2 Implementing a certain interface for the new component so that the Dialog Engine could use the new component.

Let us see the examples of the first step for the *Tree* and the *VerticalTable* components. We outline the interface that has to be implemented to be able to use the new components within the dialog engine.

4.1 The *Tree* and the *VerticalTable* Metamodels

Figure 11 and 12 depict the syntax and semantics for the *Tree* and *VerticalTable* components. Note that these components are descendants of the *Component* class of the main dialog engine metamodel and events are descendants of the *DialogEngineEvent* (and thus also of *Event)* class. Some comments on the *Tree* metamodel follow.

– The *TreeNodeSelectEvent* may have a *previous* link which denotes which tree node was selected last.

– The *TreeNodeMoveEvent* occurs only when *movableNodes* attribute value of the *Tree* instance is **true.** The event is produced when the user drags one node over another (non-descendant) node (as in Fig. 11 (b)) or before or after some (non-descendant) node (in this case the position is shown as a line before or after the node). The links *previousParent, previousSiblingBefore* and *previousSiblingAfter* are created when necessary.

Some comments on the *VerticalTable* metamodel follow.

– The *lazyLoadRows* attribute means that the table rows should not be loaded all at once. Only visible rows have to be loaded first. Then, when the user scrolls the table, other rows may be loaded. Although this may speed up the table, not all cells are taken into consideration when preferred widths for the columns are calculated.

– The *insertButtonCaption* and *deleteButtonCaption* are useful only for non-read-only tables (i.e., when the superclass *Component* attribute *readOnly* value = **false**). These are captions for the buttons to insert and delete rows.

– Similarly to the *MultiLineTextBox TextLine*s, the *VerticalTableRow* also has the *inserted, edited* and *deleted* attributes. The rows are not deleted from the repository automatically, but only marked by setting *deleted* = **true**.

– The *hasCells* association is derived since it may be calculated. The order of *VerticalTableCell*s in a *VerticalTableRow* corresponds to the order of the *VerticalTableColumnType* of the *VerticalTable.*

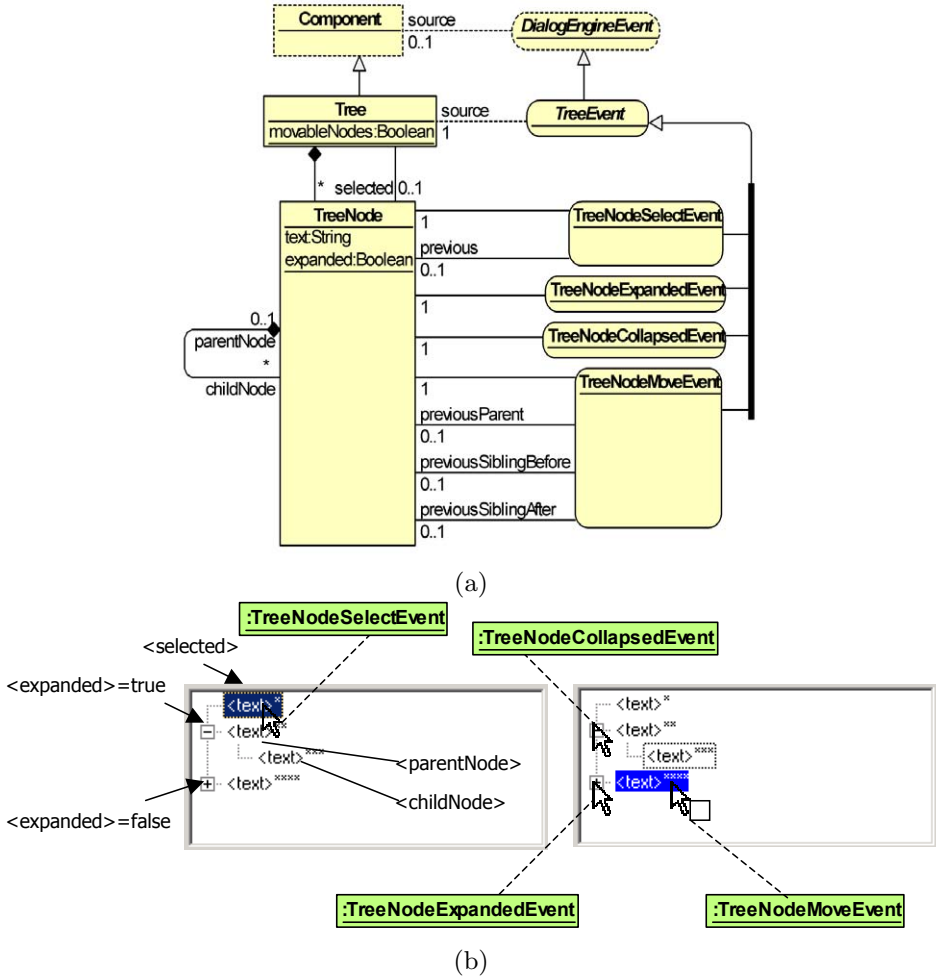
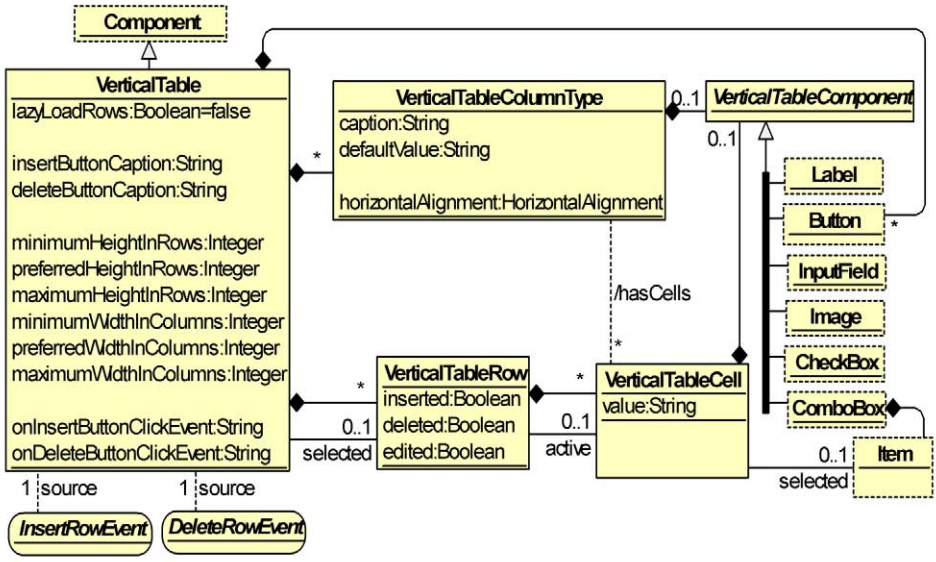
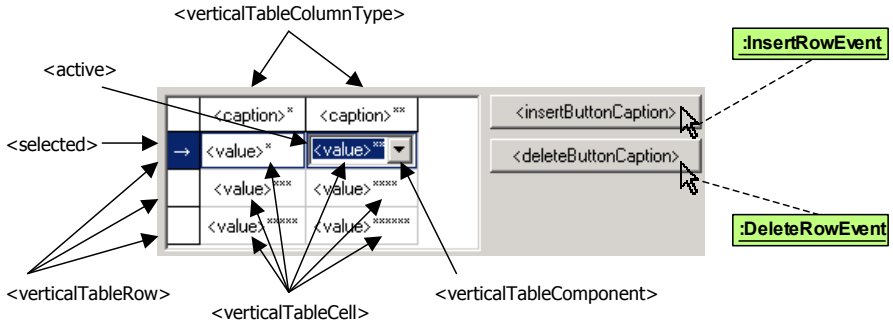


Fig. 11. (a) A metamodel and (b) its semantics for the Tree component

– The *defaultValue* attribute values are used for the corresponding cells when a new row is added.
– The *VerticalTableComponent* can be linked either to a *VerticalTableColumnType* or to a *VerticalTableCell*. In case there is no component linked to a cell, it is considered that the cell is occupied by the component linked to the corresponding *VerticalTableColumnType*. This component is also used as a default component for new rows.
– Since a component linked to the *VerticalTableColumnType* may correspond to several cells in the same column, the input value of that component should be taken from the *value* attribute of the cells. For the *ComboBox*, there is also a *selected* link from the cell to the item that should be used instead of the original *selectedItem* link from the *CombobBox* to the *Item*.



(a)



(b)

Fig. 12. (a) A metamodel and (b) its semantics for the *VerticalTable* component

4.2 The Communication Interface between Components and the Dialog Engine

In addition to the type *Component* that corresponds to the *Component* class from the Dialog Engine Metamodel, we use the following types:

– *Node*, used for internal description of a component including layout information[8];

– *Handle*, used for window handles that can be used by the dialog engine and by implementations of additional components;

[8] We will not describe that structure in detail here.

– *Control*, used as a pointer to a control (for example, to an object of the class implementing the component). The dialog engine passes this pointer back to the implementation of the component, for example, to lay out the control.

The interface is as follows.

Control, Handle – LOAD(*Component* component, *Node* node)
Called by the dialog engine when the given component has to be loaded from the repository and the corresponding graphical control created. The layout information stored in node is already loaded by the dialog engine, but that information may be adjusted here. Returns the control and the corresponding window handle. (The window handle is used to specify parent container for that window. Since the dialog engine is not related to the internal implementation of the control and obtaining of the handle, the LOAD function has to return that handle specifically.)

AFTERSHOW(*Control* control, *Component* component) // optional
Called by the dialog engine before the form becomes visible. May be useful if some initialization has to be performed when the control becomes visible.

BEFOREHIDE(*Control* control) // optional
Called by the dialog engine after the form becomes invisible. May be useful if some finalization has to be performed while the control is still visible.

FREE(*Control* control)
Called when the control should be deleted.

LAYOUT(*Control* control, *Integer* left, *Integer* top, *Integer* width, *Integer* height, *Integer* interiorWidth, *Integer* interiorHeight)
Called when the control has to be laid out. The (left; top) coordinates denote the position relative to the parent container. The interiorWidth and interiorHeight are only needed for scrollable controls.

Boolean, Control, Handle — LOADCHILD(*Control* parent, *Component* child, *Node* childNode) // optional
Called for containers when the child component has to be loaded. If the child component has been loaded, LOADCHILD should return TRUE as well as a control and a handle for the child component (like in the LOAD function). Otherwise, if the child has not been loaded or has to be skipped, the function should return FALSE as the first value.
This function can, for example, manage the situation when the children components have to be of certain type(s) only, or when some additional steps have to be performed to attach the children to the control.
If not implemented, then the LOAD(child, childNode) is called by the dialog engine and the returned control and handle are used.

5 Related Work

There are several ways of specifying dialog boxes. One is to use graphical designers that can be stand-alone programs (like GLADE [3]), or incor-porated into the IDE (Integrated Design Environment) such as Borland C++ Builder, Microsoft Visual Studio and Java NetBeans. Such designers are usually developed for a specific widget toolkit or library (e.g., GLADE is developed for GTK+ library, Borland C++ Builder uses VCL (Visual Component Library), Microsoft Visual Studio uses Windows::Forms library, and Java NetBeans uses Swing library).

There are also user interface (UI) libraries that do not have designers. In this case dialog boxes are specified in the program code that uses the routines of the particular library. Of course, such a code can also be written for the libraries that do have graphical designers.

Another way for specifying dialogs is using textual languages. HTML (Hyper Text Markup Language) is an example of such language since it allows graphical user interface components to be placed on the web pages. Other examples include User Interface Markup Language (UIML) [4] and UsiXML [5].

Along with the specification, there is the problem of laying out the dialog components. Many toolkits permit specifying absolute coordinates (like coordinates of the left-top corner) and dimensions (i.e., width and height) for each component. Several tools avoid specifying coordinates by using tables (HTML), boxes (GLADE) or other mechanisms (Java NetBeans UI designer uses horizontal and vertical groups to lay out the components by means of the *GroupLayout* manager [6]).

Java *Swing* library contains several layout managers for laying out and resiz-ing GUI components [7]. A layout manager is associated with a container, so the elements inside that container are laid out depending on the layout manager.

As of specifying resizable components, some tools (Borland C++ Builder and Delphi, Microsoft Visual Studio) permit to set up *anchors*, i.e., to fix the distance between the component and one or several window borders.

Thus, when the window is resized, the component is relocated or resized to keep these distances constant. This is useful when there is one large component that has to be resized along with the window. However, if several components have to be resized simultaneously, anchors may be a not-so-good solution, as can be seen in Fig. 13: when the form is resized, the buttons overlap.

(a) (b)

Fig. 13. (a) A form with two buttons where left and right anchors are set; (b) the form after resizing

The Windows Presentation Foundation (WPF) [8] is a platform allowing to build rich user interfaces in Windows applications. WPF uses *panels* to lay out child components (panels are similar to our containers). Also, WPF uses alignments (similar to our *horizontalAlignment* and *verticalAlignment* properties), padding (similar to

our borders and padding) and margins (similar to our margins). The difference is in stretching the components: we use maximal sizes to bound streching, while WPF uses special alignment constant "Stretch".

An interesting idea for specifying both absolute and relative sizes is based on the usage of linear constraints [9].[9] Using the constraints permits specifying the layout and behaviour of components in a more flexible way. However, defining the constraints explicitly by means of equations and inequalities is not a natural way to specify the properties of UI components. Moreover, the question arises, what to do if the constraints are unsatisfiable. UI may be generated at runtime, and the components should be laid out despite inconsistent constraints. As it was told before, we solve this problem by allowing the maximum sizes to be increased when needed.[10]

Several web-based techniques with multiple opportunities for creating user interfaces are available for developers today. AJAX is an approach where several web technologies are used to provide fast responses to the user [10]. If the client-side AJAX engine can handle the user request by its own, it does so. Otherwise, a request (usually, asynchronous) to the server is performed. Google Web Toolkit (GWT) [11] is an AJAX-type framework, which provides solutions to many AJAX problems. With GWT, web-based applications are developed in Java. However, at runtime, web-based technologies such as JavaScript and HTML are used.

Microsoft Silverlight [12] and Adobe Flash [13] are two other platforms for providing enhanced user interface experience including interactivity and anima-tions.

The XForms XML format can be used for specifying user interfaces along with data processing on the client's side. XForms is "the next generation of forms technology for the world wide web" [14]. The Apogee project [15] is aimed to provide the XForms engine (and other features) for the Eclipse environment [16].

6 Conclusion

The dialog engine that uses the proposed dialog engine metamodel has been successfully implemented, although with minor differences, in the graphical tool-building platform GrTP [2] that now uses the principles of the transformation-driven architecture [1]. The implementation of the layout of graphical dialog components is based on the quadratic optimization technique [17].

The GrTP tool contains the universal transformation which allows for common functionality to create graphical modeling tools. Universal transformation can create tool-specific dialog boxes on the fly. However, such generated dialogs do not use all the opportunities the Dialog Engine provides. For example, dialog boxes have the same row-by-row layout. To be able to adjust generated dialog windows, a graphical dialog designer may be added to GrTP.

Acknowledgments. The author would like to thank J. Barzdins and K. Freivalds for valuable personal conversations concerning the topic. Thanks also to A. Sostaks,

[9] Java also allows for creating layout managers that support constraints.

[10] A maximum size bound can also be set for the components in order not to allow them to become very large.

R. Liepins and L. Lace who gave their comments concerning graphical semantics for components. Thanks to others who directly or indirectly helped the author to get the work done.

References

1. J. Barzdins, S. Kozlovics, E. Rencis. The Transformation-Driven Architecture. *Proceedings of DSM'08 Workshop of OOPSLA 2008.* Nashville, USA, 2008, pp. 60–63.
2. J. Barzdins et al. GrTP: Transformation-Based Graphical Tool Building Platform. *Proceedings of MODELS 2007.* Nashville, Tennessee, USA: MDDAUI, 2007.
3. Glade – A User Interface Designer. Available: http://glade.gnome.org.
4. M. Abrams, J. Helms (Eds.) *User Interface Markup Language (UIML) Specification.* Working Draft 3.1. 1994. Available: http://www.oasis-open.org/committees/ download.php/5937/uiml-core-3.l-draft-0l-200403ll.pdf. See also: http://www.oasis-open.org/committees/documents.php?wg\abbrev= uiml.
5. UsiXML, USer Interface eXtensible Markup Language. V1.8. February, 2007. Available: http://www.usixml.org/index.php5?mod=download\&file=usixml-doc/UsiXML_vL8.0-Documentation.pdf.
6. T. Pavek. Get To Know GroupLayout. *NetBeans Magazine*, No. 1, pp. 58–66. Available: http://www.netbeans.org/download/magazine/01/nb01\group\layout.pdf.
7. Laying Out Components Within a Container. Available: http://java.sun.com/docs/books/tutorial/uiswing/layout/
8. Windows Presentation Foundation. Available: http://msdn.microsoft.com/en-us/library/ms754130.aspx
9. C. Lutteroth, G. Weber. User Interface Layout with Ordinal and Linear Constraints. *Proceedings of the 7th Australasian User Interface Conference*, Vol. 50, pp. 53–60.
10. J. J. Garrett. Ajax: A New Approach to Web Applications. Available: AdaptivePath.com, http://www.adaptivepath.com/ideas/essays/archives/000385.php.
11. Google Web Toolkit homepage. Available: http://code.google.com/webtoolkit/.
12. The Official Microsoft Silverlight Site. Available: http://www.silverlight.net/.
13. Adobe Flash. Available: http://www.adobe.com/products/flash.
14. W3C: The Forms Working Group. Available: http://www.w3.org/MarkUp/Forms/.
15. The Apogee Project. Available: http://www.eclipse.org/apogee/.
16. Eclipse homepage. Available: http://www.eclipse.org.
17. K. Freivalds, P. Kikusts. Optimum Layout Adjustment Supporting Ordering Constraints in Graph-Like Diagram Drawing. *Proceedings of the Latvian Academy of Sciences*, Section B, Vol. 55, No. 1.

DOMAIN-SPECIFIC LANGUAGES AND TOOLS

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 173–192 P.

The Configurator in DSL Tool Building

Arturs Sprogis
Institute of Mathematics and Computer Science, University of Latvia
Raina bulv. 29, Riga, LV-1459, Latvia
Arturs.Sprogis@lumii.lv

This paper describes the Configurator which provides ability to create graphical tools for different domain-specific languages (DSLs) quickly and convieniently. To define different DSLs by the Configurator, a TDA graphical tool building platform and its main component – a *Tool Definition Metamodel*, is used. By using this technology, a specific graphical tool is built as an instance of the *Tool Definition Metamodel,* the main task of the Configurator being creation of new *Tool Definition Metamodel* instances. The basic idea behind the Configurator is to create the instances graphically and add its properties through dialog windows. New universal graphical language and transformations converting universal language elements into *Tool Definition Metamodel* instances was developed as a materialization of this idea.

Keywords: DSL, graphical tool building, metamodel, model transformation.

1 Introduction

Many different formal models are used to describe complex information structure and each model is expressed in a particular language. DSLs [1, 2] are typically the choice because they are specially created to solve problems in one specific domain using well-known concepts for domain experts in contrary to universal languages which solve problems in many domains simultaneously. The main advantage of DSLs is that they allow thinking in a higher level of abstraction; however, their application is restricted by the lack of corresponding tools. Programming every single tool from scratch is time-consuming and takes a lot of effort. Therefore, more advanced methods implementing DSL tools are necessary.

Currently the leading DSL tool definition frameworks are MetaEdit+ [3, 4, 5], Eclipse GMF [6] and Microsoft DSL Tools [7]. DSL tools in MetaEdit+ are defined by GOPPRR [3] (Graph, Object, Property, Port, Relationship, Role) modelling language. All concepts are defined independently from each other but their relationships are specified later when all concepts have been defined. Although in MetaEdit+ new DSL tools are made easily by defining language concepts graphically, the main disadvantage is that it is impossible to change the default behaviour with additional code.

Eclipse GMF and Microsoft DSL Tools use code generation approach. DSL tools are created by compiling generated code and if any changes are necessary, the generated code has to be altered. This requires DSL developers to have advanced knowledge in generated code and in the programming language used in code generation. In addition, one of the main disadvantages of Eclipse GMF is that a user interface is hard to understand, whereas Microsoft DSL Tools is a commercial product and it may only be used together with Microsoft Visual Studio.

In this paper, a new approach of defining DSL tools is presented incorporating both – an easy-to-use graphical interface for typical use cases and a programmatic approach for more specific cases. This idea is implemented in the Configurator, allowing tool builders to define DSL tools with greater flexibility.

Chapter 2 is an overview of the Configurator. An overview of the graphical platform used to build the Configurator is presented in Chapter 3. The implementation of the Configurator and an example illustrating the use of the Configurator is described in Chapter 4.

2 An Overview of the Configurator

Each DSL consists of a number of graphical concepts. One of the basic principles used in the Configurator is to define each concept graphically by defining concept prototypes. Thus, it is necessary for the Configurator's DSL to define other DSLs in the same way OMG defines UML [8] by using Meta-Object Facility (MOF) [9].

DSLs are implemented as graph diagrams; therefore, the Configurator's DSL consists of three main concepts – box, line and property. Box describes nodes, line describes edges and property describes compartments added to node or edge in graph diagrams. Thus, prototypes are expressed in terms of these three concepts. However, each concept has its own behaviour, notation and constraints distinguishing it from other concepts and these features are specified by complex dialog windows. Thus, graphical concepts together with dialog windows make the Configurator's DSL.

The Configurator is implemented using the TDA [10, 11, 12] graphical tool-building platform. The TDA platform consists of engines and metamodels. Every engine has its own corresponding metamodel. For example, *Presentation Engine* uses *Presentation Metamodel* to depict diagrams, whereas *Dialog Engine* uses *Dialog Metamodel* to show dialog windows to end users. Most important of those are the *Universal Interpreter* and the *Tool Definition Metamodel*. The *Universal Interpreter* is a universal transformation interpreting the *Tool Definition Metamodel* to provide working DSL tools. The basic idea of the Configurator is that it defines instances of the *Tool Definition Metamodel* and the *Universal Interpreter* does the rest of the work in cooperation with the *Presentation Engine* and the *Dialog Engine*. The main task in TDA platform which is accomplished by the Configurator is the creation of the the *Tool Definition Metamodel* instances.

Although the Configurator is a tool that defines other DSL tools, the Configurator itself is implemented as a DSL tool in TDA platform using the *bootstrapping* method. The Configurator's *Tool Definition Metamodel* instances are created as a software code and interpreted by the *Universal Interpreter* afterwards. An important thing in the TDA platform is the *Extension Point* mechanism. The *Extension Point* mechanism allows a tool builder to create his own transformations or even programmes and then stores them in the *Tool Definition Metamodel* instances, in this way defining a self-contained tool. The *Extension Point* transformations are called by the *Universal Interpreter* in certain situations. Therefore, very complex tools can be made including the Configurator itself, which maps the Configurator's DSL individuals to the *Tool Definition Metamodel* instances.

3 The TDA Platform

The Configurator is implemented in the TDA platform as a DSL tool; therefore, the TDA platform will be explained in more detail. The TDA platform consists of engines and related metamodels. Every engine accomplishes its functions by interpreting corresponding metamodel. The most important components are the *Universal Interpreter* and the *Tool Definition Metamodel*. The *Tool Definition Metamodel* defines DSLs and the *Universal Interpreter* implements them. The *Universal Interpreter* consists of two kinds of transformations – *Universal Transformations* and *Specific Transformations* executed in specific situations. The main transformation is the *Universal Transformation*, which provides the end users with working DSL tools by interpreting static part of the *Tool Definition Metamodel*.

A command and event mechanism is used to provide a communication among multiple engines. Each event corresponds to the end user's action. As an example – a double click on element corresponds to the event, commands correspond to an order for the engine, for instance, an order for the *Presentation Engine* to redraw all the elements in the diagram. Thus, the communication is organized in such a way that if the end-user does something in the diagram, the *Presentation Engine* receives this action. Then the *Presentation Engine* classifies the action and creates a new event for the transformation. At this moment, the control is assigned to the main transformation that decides which transformation is called to process the event. When the event is processed, the control is passed back to the *Presentation Engine* and a command is created if any assistance by *Presentation Engine* is necessary.

3.1 The Presentation Metamodel

The *Presentation Engine* interprets the *Presentation Metamodel* that results in diagrams seen by end users. Diagrams and their elements are presented as graphs and therefore the *Presentation Metamodel* is very similar to the graph metamodel. In Fig. 1, the kernel of the *Presentation Metamodel* is presented.

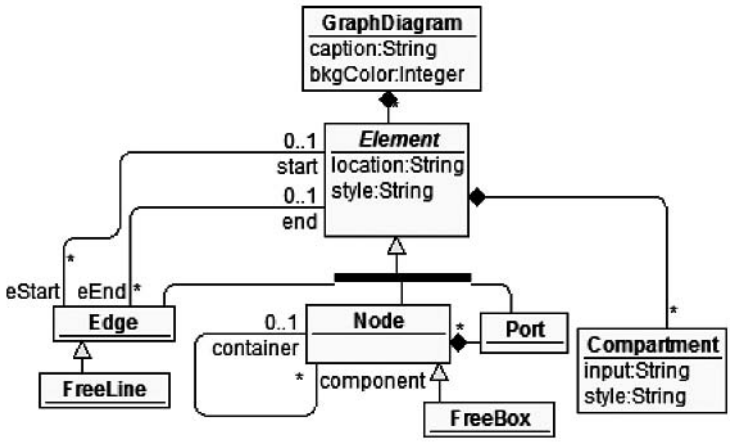


Fig. 1. The kernel of the *Presentation Metamodel*

In the *Presentation Metamodel* every diagram is a graph with name represented by the class *GraphDiagram*. Each diagram contains some elements represented by *Element*. *Element* is an abstract class and therefore real diagram elements are described by its subclasses *Node*, *Edge*, *Port*, *FreeLine* and *FreeBox*. These elements have two attributes – *location* and *style*. Attribute *style* describes how an individual element is visualized and *location* contains information about element position in the diagram and its size. Each element may have a number of attributes, which display the information entered by an end user and are represented by class *Compartment* containing attribute *input* to store entered value and *style* to represent the value.

However, every element must have its own default style and therefore the *Presentation Metamodel* is symmetrically extended with classes *GraphDiagramStyle*, *ElemStyle*, *EdgeStyle*, *NodeStyle* and *CompartStyle*. The extended *Presentation Metamodel* is presented in Fig. 2. Default styles are used when a new element is created, but they can be changed by an end user as well. Thus, when each element is created the default style value is stored in *style*, but if the individual style is changed afterwards, the value stored in *style* is overwritten.

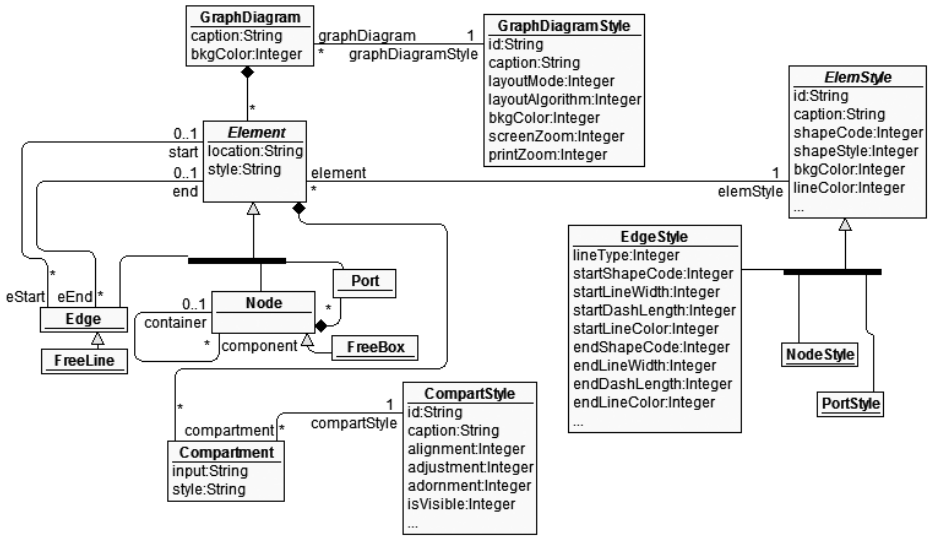


Fig. 2. The *Presentation Metamodel*

The next step is to extend the metamodel to support additional services. Classes *Palette*, *PaletteElement*, *PaletteNode*, *PaletteFreeBox*, *PaletteFreeLine*, *PalettePort* and *PaletteEdge* describe controls allowing to create new elements in diagrams. Classes *Toolbar* and *ToolbarElement* add a toolbar component and classes *PopUpDiagram* and *PopUpElement* add context menus.

To ensure the previously described event and command mechanism, *Event* and *Command* classes must be added to the metamodel as well. Each particular event and command is represented as a subclass of *Event* or *Command* (they are not presented in this paper). Class *Event* has exactly one instance at any given time, whereas

several *Command* instances can be linked by *previous-next* links simultaneously. Two additional classes *CurrentDgrPointer* and *Collection* indicate the state of the tool. *CurrentDgrPointer* indicates the active diagram; *Collection* indicates elements selected by an end user. In Fig. 3, a simplified metamodel is presented.

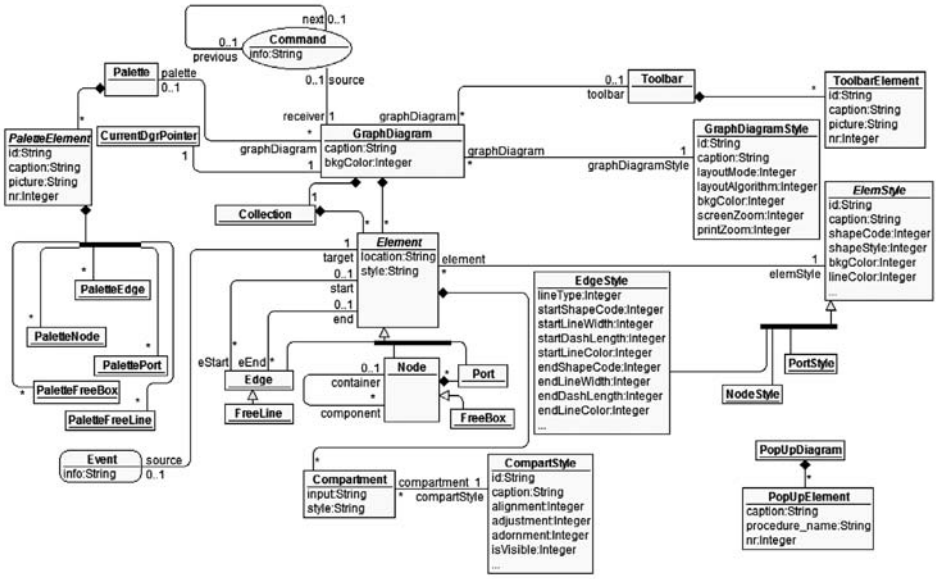


Fig. 3. A simplified *Presentation Metamodel*

3.2 The Structure of the Tool Definition Metamodel

The *Tool Definition Metamodel* is created as an extension of the *Presentation Metamodel* and its basic idea is to describe DSL's graphical elements, their behaviour, constraints, and the necessary information to automatically generate dialog windows. The main classes are *GraphDiagramType*, *ElemType*, *NodeType*, *EdgeType*, *PortType*, *CompartType* that are symmetric to the *Presentation Metamodel*. These classes store metainformation about each individual tool and are interpreted by the *Universal Interpreter* which processes all end user's actions in cooperation with other engines, for example, *Presentation Engine* and *Dialog Engine*. To create more powerful tools, types are complemented with a special kind of attributes starting with prefix "proc" in order to implement the *Extension Point* mechanism which allows adding specific transformations by tool developers to specify element behaviour in certain situations, for example, to fill dynamically drop-down menus.

However, types not only describe element behaviour, they describe the existing constraints as well. A composition relationship between *GraphDiagramType* and *ElemType* is a constraint, which determines a set of elements contained in the diagram, whereas composition between *ElemType* and *CompartType* defines attributes linked to the element. A class *Pair* determines which types of elements may be connected. Thus, associations *pair-start* and *pair-end* define which type of elements can serve as start

and end elements. Another constraint is the association *containerType-componentType* defining which type of *Boxes* may contain other *Boxes*. At the same time association *nodeType-portType* determines the type of *Box* that is enchained to a *Port*.

There are situations when some attribute values have to be entered independently from other attribute values and they have to be concatenated when shown in diagrams. This is implemented using composite attributes by adding associations *subCompartment-parentCompartment* and *subCompartType-parentCompartType*. A hierarchy of attributes is made in a way that only first level attributes are displayed to end users and attributes above hold their temporary values when processed by the *Universal Interpreter*. For instance, in a UML class diagram, *Object* name (full name) is made by concatenating three values – "individual_name", ":" and "class_name". There is one first level attribute showing the result of concatenation, for example, "John:Person", and three second level attributes holding values for each attribute – "John", ":" and "Person".

Classes *PropertyDiagram*, *PropertyTab*, *PropertyRow* represent components used by the *Universal Interpreter* to generate complete dialog windows automatically. Class *PropertyRow* has an attribute *rowType* determining the type of control used to enter attribute values. For example, the value "InputRow" specifies the text box control. To allow calling multiple dialog windows in several levels an association *calledPropertyRow-calledPropertyDiagram* is introduced. This feature is used if the *rowType* value, for example, "InputRow+Button" is chosen. As a result, a text box and a button are added, and by pressing the button, another dialog window is opened.

Sometimes it is necessary to dynamically change element and attribute styles. Element and attribute type has at least one corresponding style that is joined by the associations *elemType-elemStyle* or *compartType-compartStyle*. If there is more than one style, they must be switched in certain situations. It is implemented by adding a class *ChoiceItem* and associations *choiceItem-ElemStyleByChoiceItem* and *choiceItem-compartStyleByChoiceItem*. Each *ChoiceItem* instance holds a certain value and if this value is entered, the linked style is added. For instance, in UML class diagram, *Class* name's text has to be shown in normal or in italic based on whether the class is abstract or not. Thus, there must be a check box, which is checked if the class is abstract and not checked otherwise. According to the metamodel, there are two *ChoiceItem* instances holding values "True" and "False". A normal style is added to "False" value and an italic style is added to "True" value. Hence, when an end user checks or unchecks the check box, the *Class* name's style is changed accordingly. In Fig. 4, the complete *Tool Definition Metamodel* is present.

4 The Configurator

The implementation of the Configurator is predominantly based on the *Universal Interpreter* and the *Tool Definition Metamodel*. All the tools store two different kinds of instances in the *Tool Definition Metamodel*. One kind of instances defines the tool and is static as far as instances never change. The *Universal Interpreter* uses them to process end user actions. The second kind of instances is created dynamically and those correspond to the elements end users work with. End users may add, update, and delete them. The main problem is how to define static instances because they are individual for every single tool, whereas dynamic instances are processed equally in all tools.

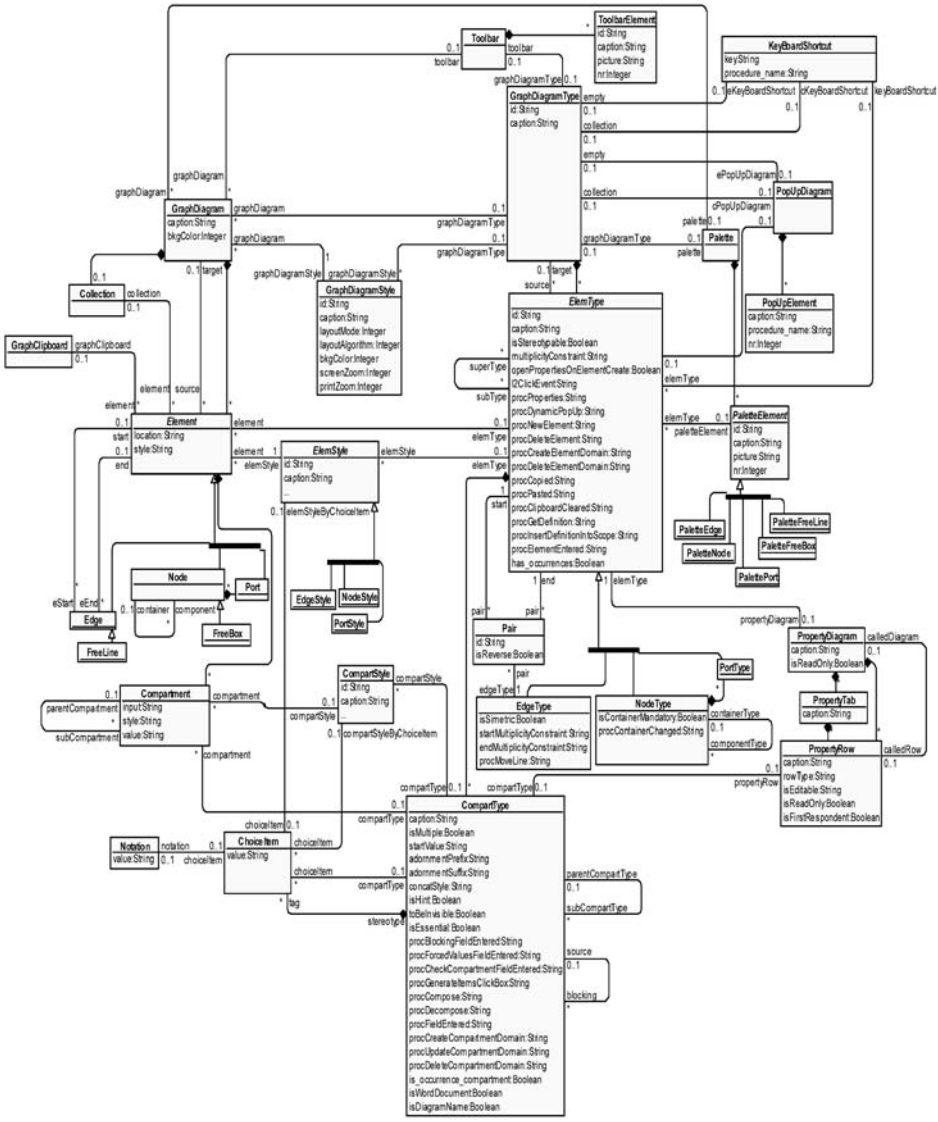


Fig. 4. The *Tool Definition Metamodel*

If static instances are “somehow” created, the working tool is obtained immediately. The naïve approach would be to create them manually but it causes several problems. Firstly, a number of instances soon grow very large. Secondly, there are many links and attribute values to be set and those can easily cause an error; therefore, this approach is significantly error-prone. Thirdly, a tool developer must know the *Tool Definition Metamodel* and attribute values expected by the *Presentation Engine*.

The Configurator is built to automate the creation of the *Tool Definition Metamodel* instances using the *bootstrapping* method. The basic idea is to implement the Configurator

as a DSL tool using the *Tool Definition Metamodel* and the *Universal Interpreter*. There are static instances defining the Configurator like any other tool in TDA but the new approach is to define static instances of DSL tools by dynamic instances using mapping from dynamic instances to static. The mapping is created using the *Extension Point* mechanism. The *Universal Interpreter* calls specific transformations in certain situations and this process consists of two steps. In the first step, the *Universal Interpreter* creates dynamic instances, which are seen by tool developers. Then, in the second step, a specific transformation which creates, deletes or updates static instances is called. The structure of DSL tool definition in the Configurator is presented in Fig. 5.

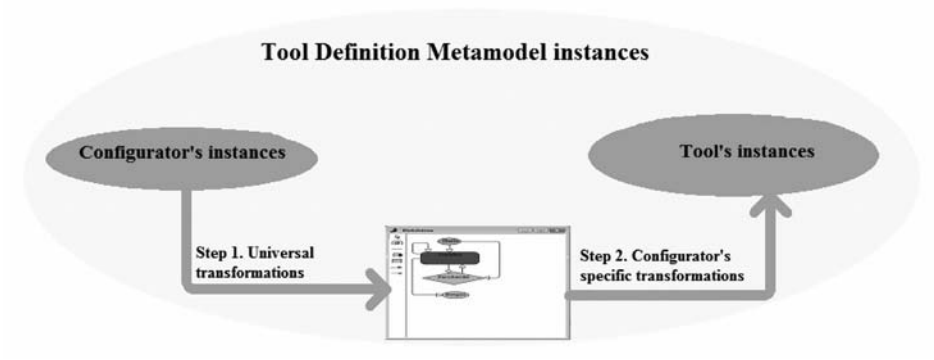


Fig. 5. The structure of DSL tool definition in the Configurator

The scaffolding must be added to the *Tool Definition Metamodel* to implement the Configurator according to the schema. The main purpose of scaffolding is to map dynamic instances to the static instances. There are three associations *presentation-target_type* added to the metamodel to identify an element type being defined by *GraphDiagram*, *Element* or *Compartment*. In Fig. 6, an extended *Tool Definition Metamodel* fragment is presented.

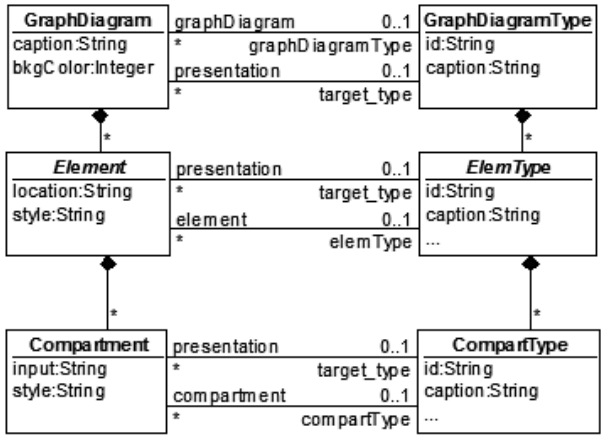


Fig. 6. An extended *Tool Definition Metamodel*

4.1 Implementing the Configurator

According to the TDA platform, static and dynamic instances are stored for each DSL tool. As far as the Configurator is implemented as a DSL tool, there are static instances defining the Configurator and those are presented in this sub-section.

The Configurator's DSL consists of two diagram types. One type of diagrams defines prototypes for diagram seeds, and lines illustrating dependencies between them. The second type of diagrams defines element prototypes. The first type of diagram is named *Specification Diagram* with three types of elements possible – *Seeds*, *Lines* and *Specializations*. *Seed* is an element which defines seed prototype; *Line* is an element which defines dependency prototype, and *Specialization* is a line used to indicate that a sub-element inherits incoming and outgoing lines and constraints from a super-element. *Tool Definition Metamodel* instances presented in Fig. 7 specify element types for *Seeds*, *Lines*, and *Specializations*. There are additional instances specifying context menus and corresponding palette buttons.

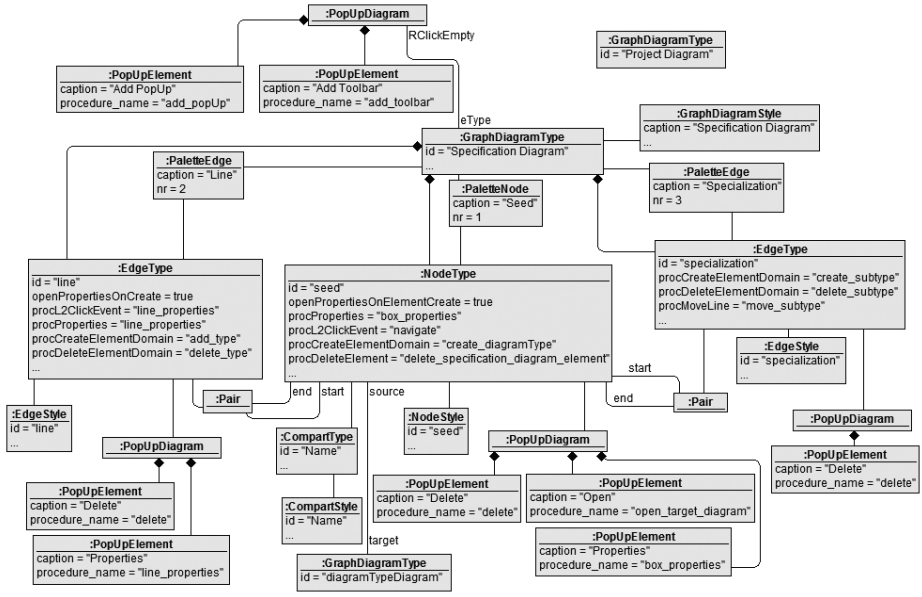


Fig. 7. The specification diagram defined in the *Tool Definition Metamodel*

The second diagram type defines element prototypes. There are six different element types available – *Box*, *Line*, *FreeBox*, *FreeLine*, *Port* and *Specialization*. *Box* and *Line* elements allow defining prototypes for boxes and lines; *FreeBox* allows defining boxes always remaining in background; *FreeLine* allows defining lines having no start and no end elements; *Port* allows defining small boxes which are always attached to some *Box*; *Specialization* is used for the same purpose as in *Specification Diagram*. In Fig. 8, a definition of prototype diagram and its elements with relevant context menus in the *Tool Definition Metamodel* is presented. *Line* and *Specialization* elements are allowed to connect all the elements except *Specialization.* For example, *Box* and *Box* elements are

allowed being connected by *Line* or *Specialization* element, but *Box* and *Specialization* elements are not allowed being connected by either *Line* or *Specialization*. Thus, all the possible pairs must be present in the metamodel to indicate which elements may be connected and which may not. there is a special *NodeType* instance with *id* value "superType" added as a super-type for all the elements, except *Specialization,* and two *Pair* instances, which connect "superType" instance and *Specialization*'s type instance, "superType" instance and *Line*'s type instance. Introduction of "superType" is needed to save the effort of making all the necessary pairs because the incoming and outgoing lines are inherited from the super-type.

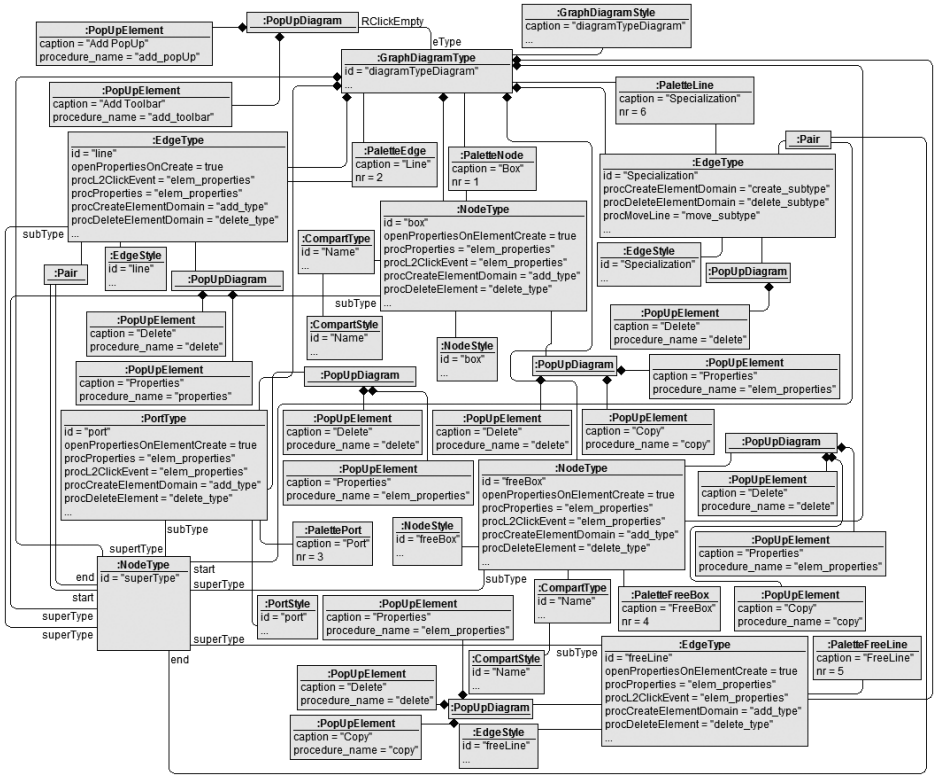


Fig. 8. Prototype diagram's definition in the *Tool Definition Metamodel*

4.2 The Configurator in Use

In Fig. 7 and 8, the *Tool Definition Metamodel* instances defining the Configurator are presented. If the tool is specified by static instances, the *Universal Interpreter* creates and processes dynamic instances. To illustrate how static instances are used in tool building, a simplified *Flowchart* editor is built consisting of the following symbols – *Start*, *End*, *Action*, *Branching*, *Simple Flow* and *Branching Flow*. In addition, *Action* symbol has a property *Expression*; *Branching* symbol has a property *Condition* and *Branching Flow* has a property *Choice*.

A new diagram type is defined in *Specification Diagram* by creating a new seed. A new *Node* instance is created and linked to *GraphDiagram*, *NodeType* and *NodeStyle* instances by the *Universal Interpreter*. The result is presented in Fig. 9.

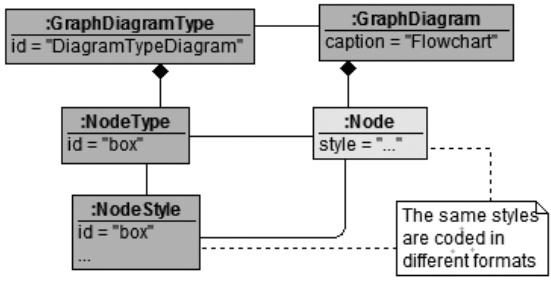


Fig. 9. Node instance created by the *Universal Interpreter*

When the *Universal Interpreter's* work is done, a specific *Extension Point* transformation from attribute *procCreateDomain* is called to create corresponding static instances for *Flowchart* editor and dynamic instances for the Configurator. It is achieved in two steps. In the first step, a new *GraphDiagram* instance is created for *Flowchart* element prototype definition. In the second step, new *NodeType*, *NodeStyle*, *GraphDiagramType* and *GraphDiagramStyle* instances are created. *NodeType* instance defines a seed type in *Project Diagram*, *NodeStyle* instance defines a seed style. *GraphDiagramType* instance defines a diagram type, which corresponds to *Flowchart* diagram, and *GraphDiagramStyle* instance defines its style. *Node* and *NodeType*, *GraphDiagram* and *GraphDiagramType* instances are linked by *presentation-target_type* to indicate which diagram type is defined; *Node* and *GraphDiagram* instances are linked by *source-target* to connect seed prototype and diagram, in which prototypes are defined. However, there is a necessity for additional instances of type *Palette* and *PaletteNode* to create a palette and a palette button in *Project Diagram*.

When a seed element is created, a property to store diagram's name must be added. This is achieved by adding *Compartment*, *CompartStyle*, *CompartType* to store and represent the end user values, but *PropertyDiagram* and *PropertyRow* instances are added to enter the value from dialog window. In Fig. 11, the instance diagram defining *Flowchart* seed is presented.

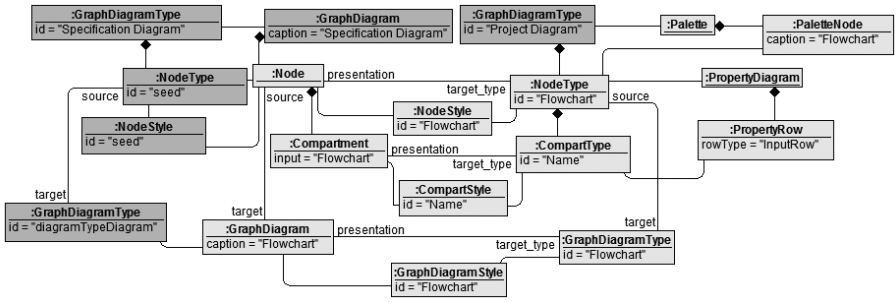


Fig. 10. Instance diagram after creating *Flowchart's* seed element

When a *Flowchart*'s seed element is defined, element prototypes must be defined. Element prototypes of *Box* type are added in the same way as seed prototype; therefore, only line definition is explained in more detail. Assuming that *Start* and *Action* elements are defined in the same manner as *Seed*, *Simple Flow* prototype is added in two steps. In the first step, *Edge* instance is created that links two *Node* instances which represent *Start* and *Action*. In the second step, *Flowchart*'s editor static instances *EdgeType*, *EdgeStyle*, *Pair* and *PaletteLine* are created. *EdgeType* and *Pair* instances define a *Simple Flow* element, which allows to connect *Start* and *Action* elements; *EdgeStyle* defines *Simple Flow*'s style and *PaletteLine* defines a palette button to create *Simple Flow* element. In Fig. 11, the instance diagram defining *Flow* is presented.

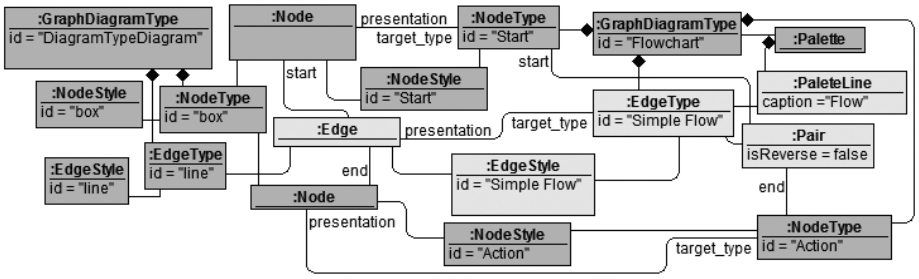


Fig. 11. Flow's definition

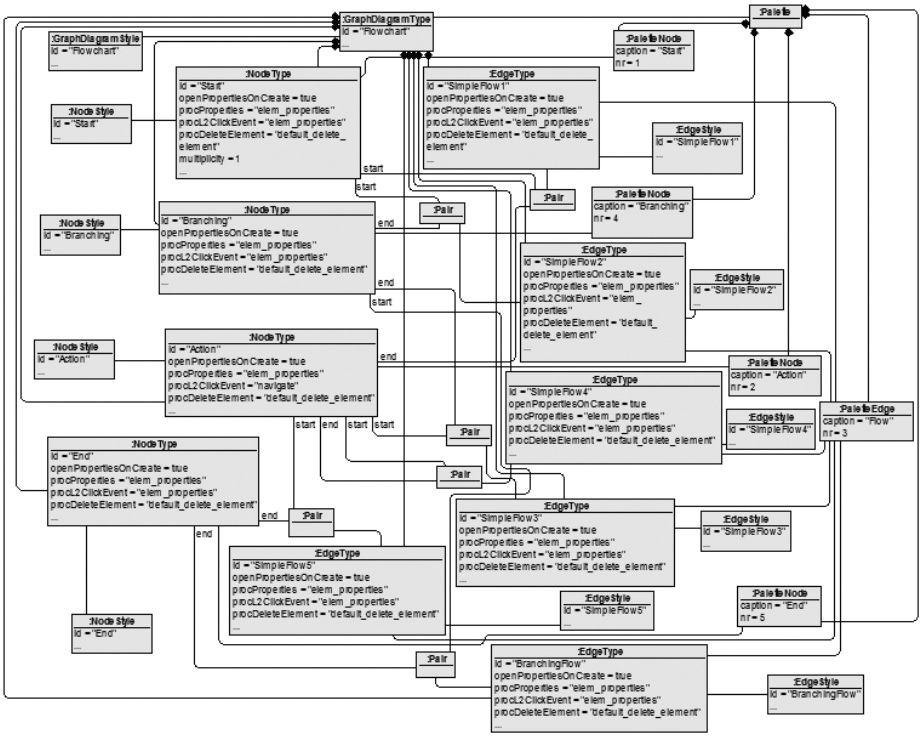


Fig. 12. The *Tool Definition Metamodel* instances

The entire *Tool Definition Metamodel* instance of the *Flowchart* editor presented in Fig. 12 can be obtained using the method described above.

4.3 Defining *Flowchart* Editor Using the Configurator

After discussing the Configurator's implementation and the way it creates the instances of the *Tool Definition Metamodel* above, we shall demonstrate the use of the Configurator from the tool builder's point of view by implementing the *Flowchart* editor.

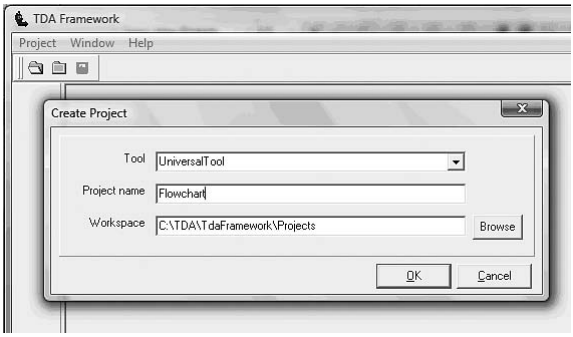


Fig. 13. A new tool definition window

When a new DSL tool is defined, a window to specify project details (Project→New project) is opened. It is presented in Fig. 13. A tool builder has to select value *UniversalTool* in the field *Tool*. Then he has to specify the name of the new tool in the field *Project name* and the project location in the field *Workspace*. When DSL developer presses *OK* button, *Project diagram* is opened. In general, *Project diagram* contains all the available diagram type seeds (elements that allow making diagrams), but that is not the case in the Configurator. When using the Configurator, *Project diagram* contains no diagram type at all, because the Configurator will define it later. New diagram types are defined in *Specification diagram*. Tool builder can navigate to *Specification diagram* by right-clicking and choosing *Specification diagram* from the context menu. A sample project diagram and the context menu are presented in Fig. 14.

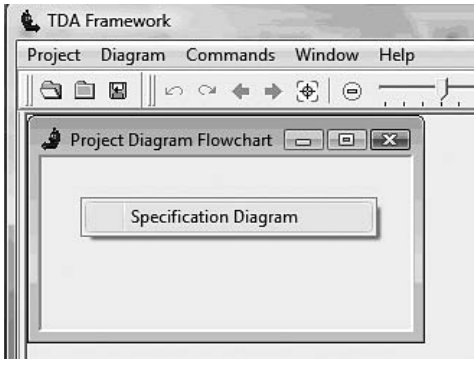


Fig. 14. A sample project diagram

In *Specification diagram*, new diagram types are defined using *Seed* element. A *Flowchart* diagram *Seed* has to be created by pressing *Seed* button in the palette. When *Flowchart* diagram type is defined, a *Flowchart* diagram is opened by double-clicking on the diagram *Seed*. In Fig. 15, *Flowchart* diagram *Seed* and diagram for element definitions is presented.

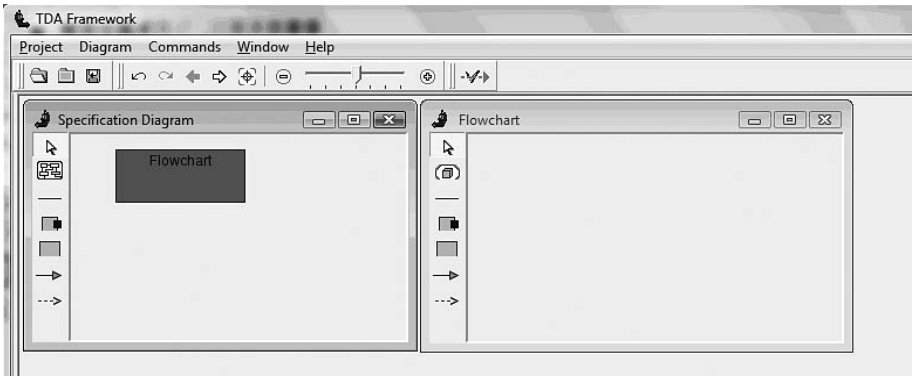


Fig. 15. Creating Flowchart Seed

When *Flowchart* definition diagram is opened, *Flowchart* elements can be defined by creating their prototypes. For instance, *Action* is defined by choosing *Box* button in the palette. A *Box* dialog window is displayed afterwards and tool builder is prompted to enter element values. In the field *Name* a value "Action" has to be entered which automatically renames a palette element name in the field *Palette Element Name*. In the field *Palette Element Nr*, a number for palette element in the palette has to be entered and in this case, the number is "2". In the field, *Icon Path* an icon's name for a palette element has to be specified. Context menu elements for *Action* have to be defined as well. Those are specified in the table *PopUpDiagram* and in this case, context menu items are default with corresponding default transformations added from the transformation library – *Delete, Cut, Copy* and *Properties*. It is possible to specify navigation target diagram in the field *Navigate To Diagram* by double-clicking on the element. If nothing is specified, no navigation is possible. However, in this particular case, a value "Flowchart" is specified meaning that double-clicking navigates the end user to one of the *Flowchart* diagrams. In Fig. 16, a window to enter *Action* values is presented.

When all the element values are entered, element properties have to be specified. It is done by pressing the button *AddChild*. In Fig. 17, a property dialog window is presented. Property value has to be entered in the field *Name*, and in this particular case, the value is "Expression". The visual control used to enter the property value has to be specified in the field *Row Type*, and in this case, the value is "InputRow", meaning the control to enter property values is a textbox.

When all the values are entered, element style has to be specified by pressing the button *Style*. In Fig. 18, the dialog window to enter element style is presented. In this dialog window, a tool builder has to specify values as box type, which may take one of the following values – rectangle, ellipse, round rectangle, etc; a default size, a colour, a border's colour, a border's width and some other visual features.

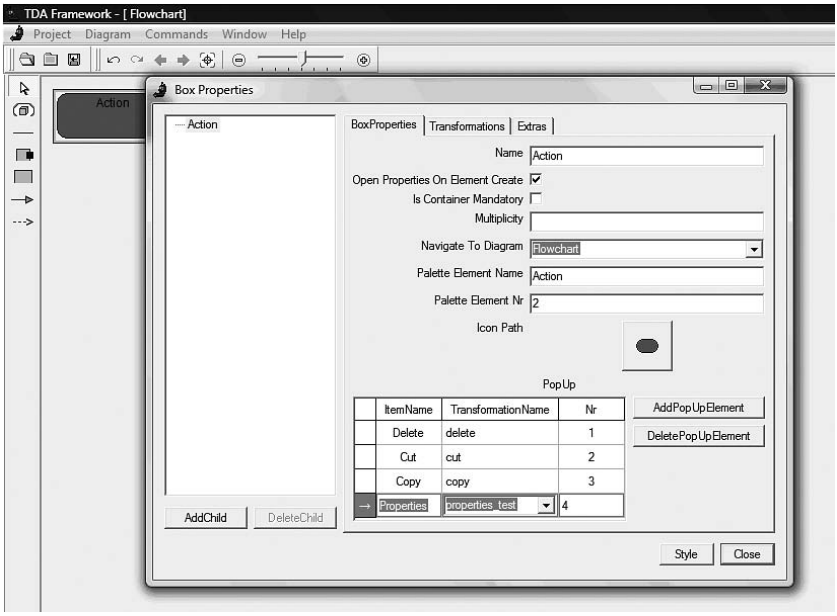


Fig. 16. Definition of an *Action* element

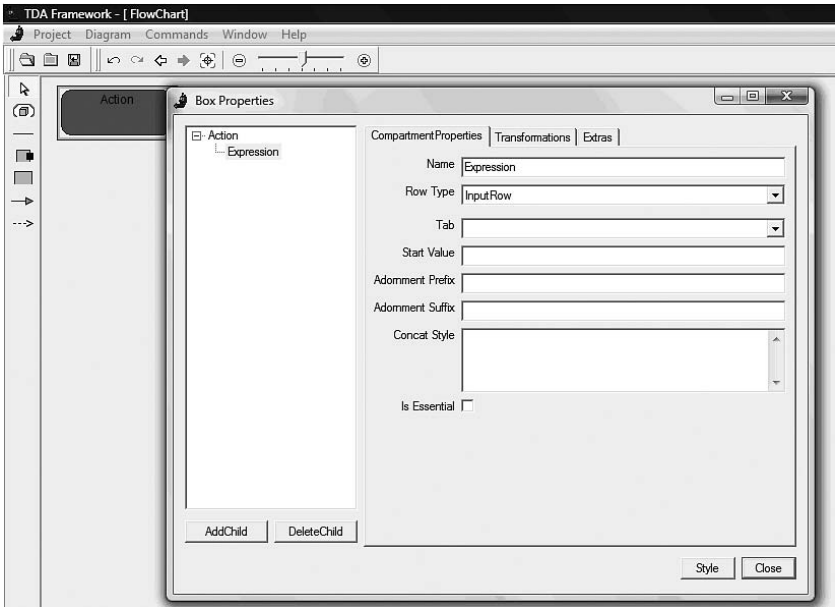


Fig. 17. Definition of the property "Expression"

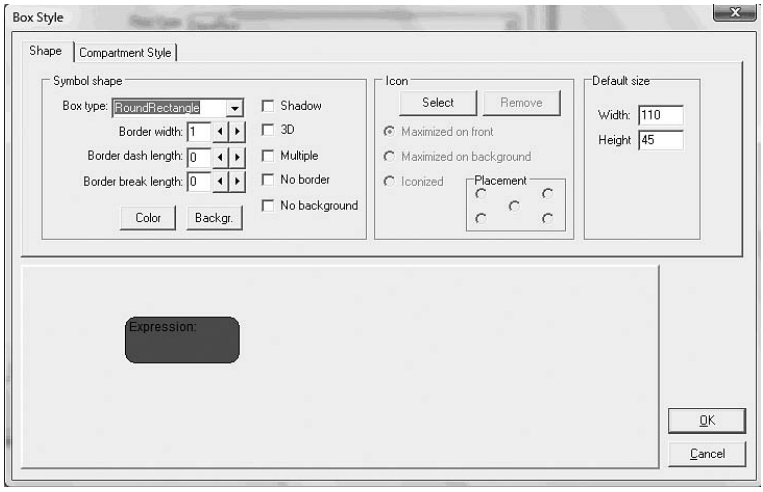


Fig. 18. A style definition window for *Box* prototype

In Fig. 19, the dialog window to enter properties style values is presented. The tool builder has to specify property values like text alignment, adjustment, font style, etc in this window.

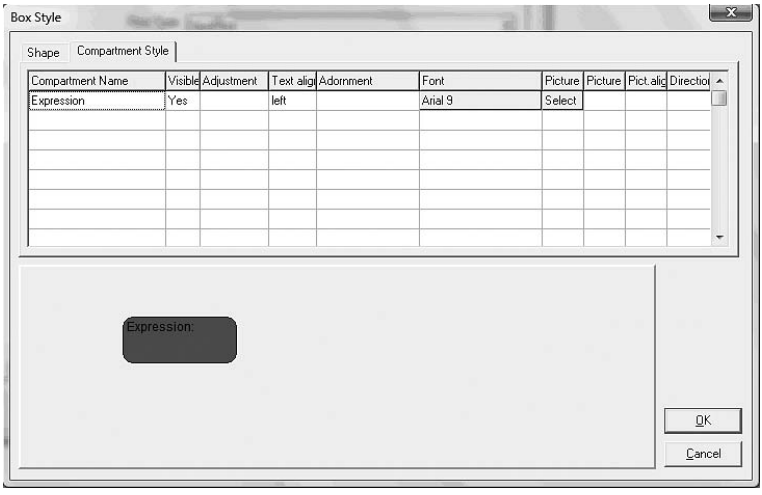


Fig. 19. A style definition window for properties

This is how concepts of *Box* type are defined in the Configurator. Other *Flowchart* concepts of *Box* type like *Start*, *End* and *Branching* symbols are defined using similar approach. In Fig. 20, all *Flowchart* prototypes of *Box* type are presented.

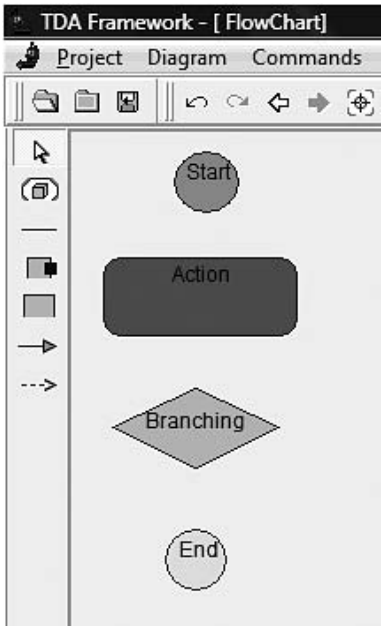


Fig. 20. Box prototypes for the *Flowchart* editor

The next step is to define prototypes of *Line* type. They are *Simple Flow* and *Branching Flow*. In the context of this example, the assumption is made that a *Simple Flow* is an element which may join *Start* and *Action*, *Start* and *Branching*, *Action* and *Action*, *Action* and *Branching*, *Action* and *End* symbols, whereas *Branching Flow* may join only *Branching* and *Action* symbols.

When a *Line* prototype for a *Simple Flow* is defined, all the mentioned cases have to be considered. One *Line* can join only two elements and wherefore there is a necessity for many new prototypes to consider all the *Simple Flow* cases. However, the tool user does not have to know all the technical constraints; therefore, an illusion must be created that there is only one *Simple Flow* element in the diagram. This is achieved by having a common palette button for all the different prototypes in diagram's palette and all the prototypes are made equal by their style and behaviour. In Fig. 21, an example is demonstrated of how prototype is defined for one of *Simple Flow* elements. The definition of elements with a *Line* type is very similar to the *Box* type definition; hence, in the field *Palette Element Name* a drop down menu is used to offer all the palette button names. If the name entered matches any item from the drop down menu, a new palette button is not created and prototype being defined is linked to an existing palette button. Otherwise, a new palette button is created. In the *Flowchart* case, all the *Simple Flow* prototypes are linked to the palette button *Flow*.

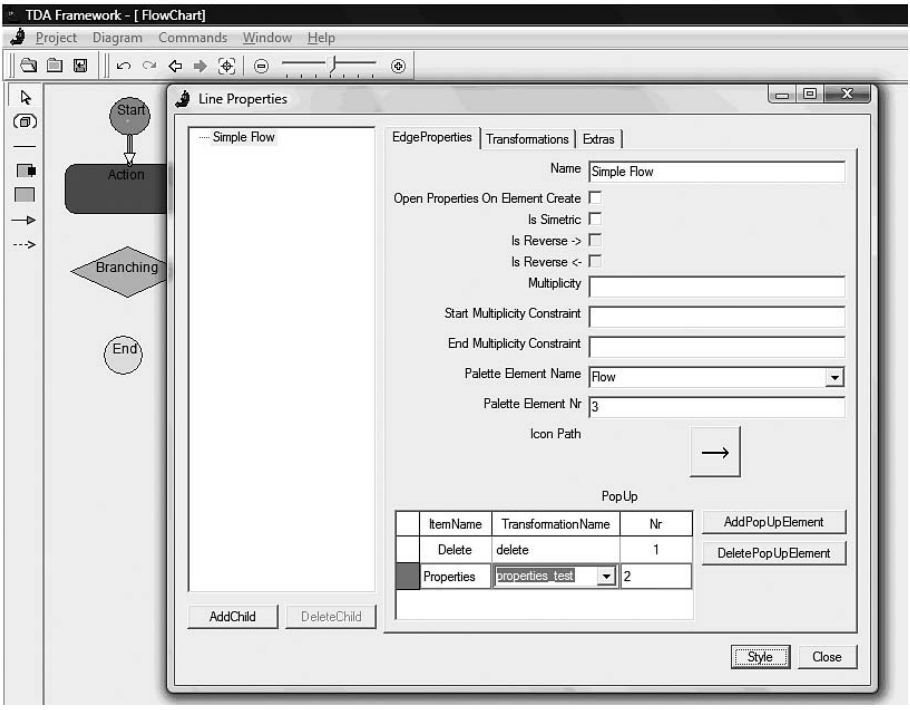


Fig. 21. Definition of a *Simple Flow* element

Branching Flow is defined almost in the same way as *Simple Flow*, except *Branching Flow* has a property *Choice* to enter values like – *Yes*, *No*, *True*, *False*, etc. In addition, *Branching Flow* is linked to the palette button *Flow*. Thus, all the *Lines* are created by only one palette button *Flow* and the decision which *Line* to choose in particular situation is made by the *Universal Interpreter*. In Fig. 22, a final definition of a *Flowchart* editor is presented.

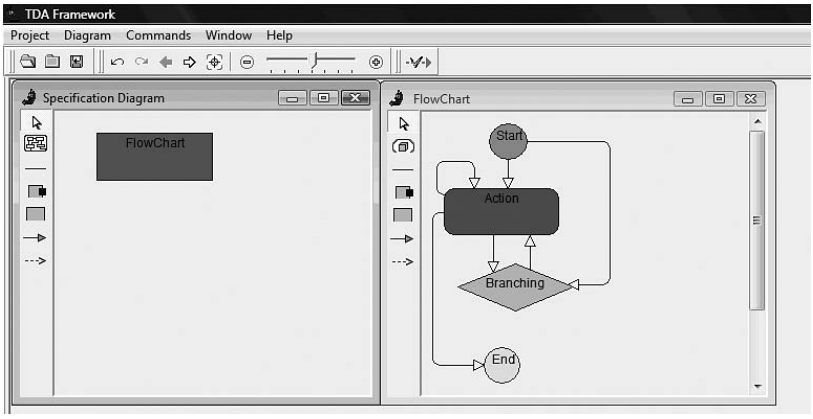


Fig. 22. Definition of a *Flowchart* editor

Yet, in Fig. 23, a working *Flowchart* editor is presented.

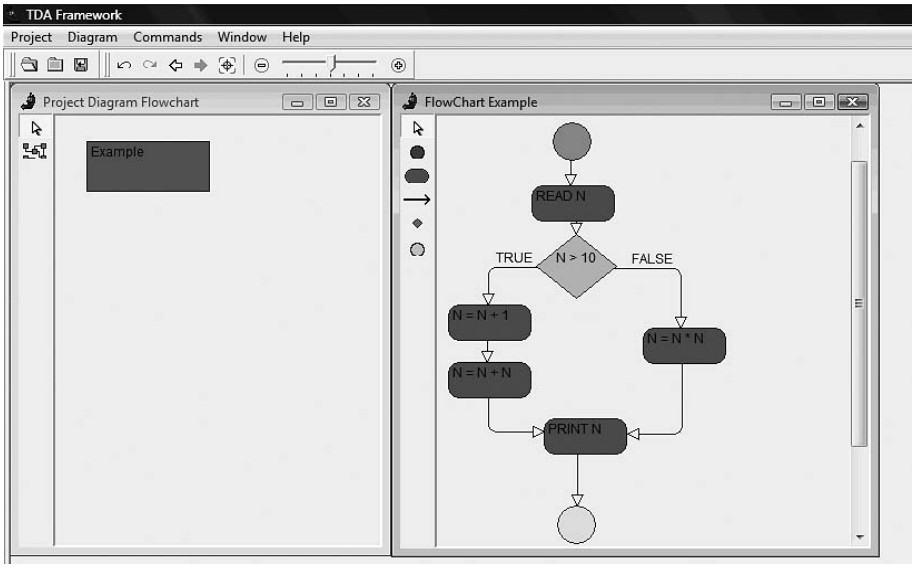


Fig. 23. A *Flowchart* editor in use

Conclusion and the Future Work

Currently the Configurator has enough functionality to implement many different DSL tools. For example, as far as the Configurator is a DSL tool, it is powerful enough to implement even such a complex tool as the Configurator itself. Real business tools are also implemented for Investment and Development Agency of Latvia and the State Social Insurance Agency. Although these tools where successfully implemented, several problems require further research – there is no multi-user mode to support multiple DSL tool developers, the graphical language is insufficiently self-descriptive and user-friendly, and incorporation of other software like MS Word, Database editors, etc in implemented tools is not completely satisfactory.

References

1. UML vs. Domain-Specific Languages. Available: http://www.methodsandtools.com/archive/archive.php?id=71.
2. Domain-Specific Language. Available: http://www.program-transformation.org/Transform/DomainSpecificLanguages.
3. MetaEdit+ Workbench User's Guide, Version 4.5. Available: http://www.metacase.com/support/45/manuals/mwb/Mw.html.
4. S. Kelly, J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008, p. 448.
5. Domain-Specific Modeling with MetaEdit+. Available: http://www.metacase.com/.
6. Graphical Modeling Framework (GMF, Eclipse Modeling subproject). Available: http://www.eclipse.org/gmf/.

7. S. Cook, G. Jones, S. Kent, A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools.* Addison-Wesley, 2007.
8. OMG modeling specification, UML 2.0 Superstructure and Infrastracture. Available: http://www.omg.org/docs/formal/07-02-05.pdf.
9. Meta-Object Facility (MOF). Available: http://www.omg.org/mof/.
10. J. Bārzdiņš, E. Rencis, S. Kozlovičs. *The Transformation-Driven Architecture.* The 8th OOPSLA Workshop on Domain-Specific Modeling, October 19–20, 2008, Nashville, TN.
11. J. Barzdins, K. Cerans, S. Kozlovics, E. Rencis, A. Zarins. A Graph Diagram Engine for the Transformation-Driven Architecture. *Proc. of the Workshop on Model-Driven Development of Advanced User Interfaces 2009*. Florida, USA: IUI, 2009.
12. J. Bārzdiņš, A. Zariņš, K. Čerāns, A. Kalniņš, E. Rencis, L. Lāce, R. Liepiņš, A. Sproģis. *GrTP: Transformation-Based Graphical Tool Building Platform.* The 10th International Conference on Model-Driven Engineering Languages and Systems, Models 2007, September 30–October 5, 2007, Nashville, TN.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 193–203 P.

The Concept of Automated Process Control

Ivo Oditis[1], **Janis Bicevskis**[2]
[1] Bank of Latvia, K. Valdemara 2a, Riga, Latvia
ivo.oditis@lais.lv
[2] University of Latvia, Raina bulv. 19, Riga, Latvia
Janis.Bicevskis@lu.lv

This paper describes research on control of heterogeneous information systems, which run as parallel interlinked processes. A formalized process control description language is proposed. It is a domain-specific language which provides opportunity to automate process execution control mechanism. The language separates two types of processes: base and supervisory processes. Supervisory processes require specific language elements for the control and synchronization of base processes. Also, the first concept of automated control mechanism is introduced. The proposed mechanism and process control definition language is developed as part of smart technology framework aiming at autonomous system concept developed by IBM.

Keywords: business process control, domain-specific languages.

Introduction

For many years computer scientists spent most of their work on research of software development technologies, while less effort was spent to make the use of the already developed software more convenient. In part this problem can be explained by software developers' concerns about software sales leaving software usage problems into users' hands. The complexity of the whole system is increased when one company or organization acquires software from more than one vendor and software is introduced with significant time span. This way a complex heterogeneous system environment is formed.

There are at least two groups among system users: those who are end users or users of the system's business functionality and system administrators whose responsibilities include system security and technical configuration of system and its environment. By its complexity the area of business system administration and control is comparable to network administration. There are numerous tools for network administration and monitoring in the market; however, the authors of this paper could find no acceptable solutions for heterogeneous system administration and their process execution control. It can be explained by the diversity of the systems and the nonstandard nature of legacy system communication.

Process control is a well-known problem. There have been many attempts to solve it in software history [1]. In the era of mainframes, process management was partly delegated to the operating system and Job Control Language. As a significant tool of this area, the SDL or Specification and Description Language must be mentioned. SDL is a specification language targeted at unambiguous specification and description of the

behavior of reactive and distributed systems. Originally focused on telecommunication systems, its current areas of application include process control and real-time applications in general; however, it requires a very detailed process description and is not suitable for high-level business process description. Therefore, authors introduce a new and simple domain-specific language for business process control.

In the first chapter, problems are identified and brief solutions are explained. The second chapter describes the architecture of the process control mechanism and the domain-specific language used for process control description.

1 Description of the Problem

1.1 Usage of Heterogeneous Systems

This publication is aimed to describe the usage of heterogeneous software in large companies, where many different software platforms are used. The formation of heterogeneous environment in long running large companies is unavoidable, if the necessary software is acquired gradually and the size and functions of the companies are changing over time.

The most serious problems are caused by distributed environment where many systems are running simultaneously on different platforms and communicating with each other. Typically, operators and administrators of each of these systems have to have specific management skills. As a rule, service staff have to follow if processes carried out by the systems are done correctly; and there is no process control built in systems. If one process is carried out by two or more independent systems, each system can control execution as far as it is in its scope, but the whole process typically is controlled manually. Therefore, the usage of systems depends on the qualification of the supporting staff and precise execution of operations by the staff.

Automated process control is proposed to solve the problems described above and to reduce the dependency of system usage on the subjective factor of the supporting staff. In brief, the proposed solution contains two components: a description of controlled processes and a mechanism for controlling process execution according to process description.

1.2 Smart Technology and Autonomous Systems

The proposed process control automation solution is based on the ideas of *smart technology* [2]. The idea of *smart technology* is to create software similar to a live organism, which can react adequately to unpredictable changes of living environment. Ideally software built according to the principles of smart technologies could adequately react to the changes of the external environment (changes of infrastructure, network throughput etc) as well as to the internal environment. Smart technologies provide a framework for software development. Using a common framework, smart technology could be included in systems without significant increase of software complexity.

The concept of smart technology includes external environment testing [3, 4], intelligent version updating [5], self-testing [6] and others. The concept of smart technology has similar goals as the concept of autonomous systems developed by IBM

in 2001 [7, 8, 9]. Both concepts aim at improving software intellect by adding a set of nonfunctional advantages – ability to adapt to external situation, self-renewing, self-optimizing and other advantages. The autonomous systems are built as universal and independent form properties of a specific system. As a rule, they function outside of a specific system and cooperate on the level of application interface.

The first results of smart technology implementations are available. There are two types of smart technology software developed and introduced in currently used systems: intelligent version updating software and software for external environment testing. The first is used in budget planning and the discharge control system FIBU. It is used in more than 400 state-owned organizations in the Republic of Latvia. There are more than 2 000 users of this software. The external environment testing software is used by the same system, FIBU, to solve the problem of many operating systems and other application versions. The same external environment testing software is planned to be used in the Bank of Latvia to manage numerous independently developed, but interrelated systems. The first results of the use of smart technology demonstrate their practical usefulness [10, 11].

This research is on smart technology as well: automated operation control of heterogeneous systems.

1.3 Process Control Automation as Part of Smart Technology Framework

The suggested automated process control concept can be identified as another extension of smart technology. It works over heterogeneous systems and semi-automates system process control.

The solution introduces two types of processes:

- *base processes* – simple processes containing no sub-processes;
- *supervisory processes* that contain, control and synchronize base processes.

Processes implemented by computer systems are mutable by their nature. Process modifications can be caused by changes in system infrastructure, by changes in process priorities or changes in organization structure. The suggested process control concept contains two components: the process control mechanism and process control description. The process control mechanism is running according to easily adjustable process description. It is useful to implement process control description by introducing a domain-specific language with elements particular for the control of base and supervisory processes.

We briefly describe the process control mechanism and process control definition language. Detailed implementation of the language and control mechanism is subject of further research.

Processes from a payment clearing system will be used in descriptions of the process control mechanism and process control definition language. Payment clearing systems provide a large volume of retail payment exchange and settlement (clearing) among banks (system participants). Typically, a clearing process is organized in four steps: systems receive retail payment batches from participants, calculate each participant's position (difference of payments' totals sent and received by participant), settle positions in the system where participant accounts are kept and deliver payments to participants,

payment receivers. This is called a "clearing cycle". Systems with one or more clearing cycles per day exist. A system with one clearing cycle will be used in our examples.

Using the clearing system, base and main processes can be identified. Receiving payments from one clearing participant can be described as a base process, and the whole clearing cycle can be identified as a supervisory process containing and controlling a set of base processes (each participant's payment collection).

2 Automated Process Control

Automated process or system operation control mechanisms check if the described system processes are running according to the process descriptions. The process sequence and operation timing is checked. If a discrepancy between the description and ongoing processes is detected, the control mechanism sends information to system support stuff. Two types of information can be identified: timely warnings (the system tries to identify potential problems) and information on the detected errors.

The control mechanism's main task is to continuously verify whether the process flow is correct, incoming and outgoing data is coherent, all process steps are done and whether all of the steps are done timely. The control mechanism does not test the system under control nor does it test the quality of data produced by the system.

Another important component of the process control mechanism is process trace recorder. Process traces could be useful not only to identify possible causes when a problem has occurred, but also could provide substantial statistical information on a typical system workload and bottlenecks. Analysis of system traces could provide early warnings about changes in process execution times.

Three collaboration types are possible between the control mechanism and systems under control:

- the system under control is interpreted as a black box from the control mechanism perspective, and all information on the process flow is taken from system external interfaces;
- the system under control sends information to the control mechanism on process execution;
- the system under control requests information from the control mechanism on process execution.

2.1 The Architecture of the Process Control Mechanism

A significant number of systems are distributed over more than one server. Thus, one of the most important requirements for the architecture of the system control mechanism is the possibility to control widely distributed systems.

The control mechanism offered by the authors contains two main components: *Central Hub* and *Agents*. Agents are software modules which trace different events of the systems implementing the process under control. For example, a new file or file modification could be one type of event handled by agents. When agent detects event related to the process under control, it sends event notification to the central in order to check if the process is running according to process description and if the timing of the process is accurate.

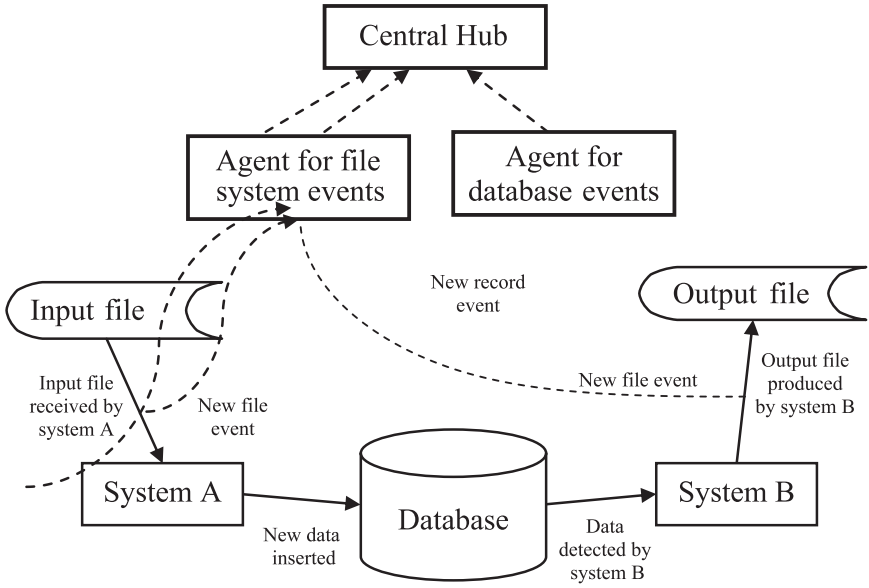


Fig. 1. The control mechanism contains two types of components: *Central Hub* and *Agents*

For instance (Fig. 1), one process could be provided by two systems: A and B. *System A* takes an input file, processes its contents and inserts new data in a database. *System B* takes the data from the database and produces another output file. *Central Hub* is controlling the whole process by using two event agents: one agent provides file system events, another – database events. When input file is received, the file system event agent receives "new file" event and passes it to *Central Hub.* After *System A* inserts data into database, "new record" events are handled by the second agent and sent to *Central Hub* for processing. When *System B* creates an output file one more "new file" event is handled by the first agent and passed to *Central Hub.*

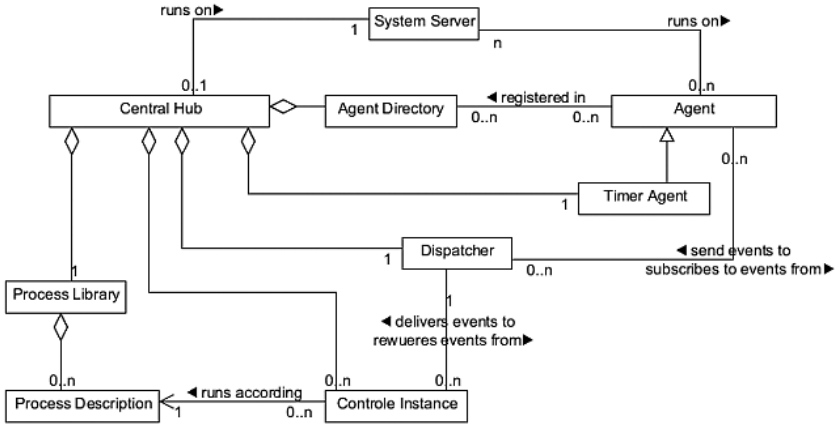


Fig. 2. A UML class diagram of the process control mechanism components

Central Hub (Fig. 2 represents components of the process control mechanism) contains five modules: *Process Library*, *Controller Instances*, event *Dispatcher*, *Agent Directory* and *Timer Agent*. *Process Library* contains all of *Process Description* the *Central Hub* has to look after. When a new process is started, *Central Hub* takes the *Process Description* from *Process Library* and creates new *Controller Instance* to control the process flow. *Controller Instance* analyses *Process Description* and receives events from *Dispatcher* the process could generate. Data *Dispatcher* subscribes to the appropriate type of event from appropriate *Agent* according to the required event types and *Agent Directory*. When *Agent* handles the requested event, it sends it to the *Central Hub's Dispatcher* where *Controller Instance* which requested the event is identified. *Control Instance* processes each event and checks process state according to *Process Description*. If events are fired in inappropriate order, *Controller Instance* sends error messages or warnings to the person in charge according to *Process Description* notification rules. There is a specific type of *Agent*, the *Timer Agent*, hosted in *Central Hub*. It provides timer events to *Controller Instances*. *Control Hub* is hosted on one server; however, there may be more than one *Agent* hosted inside a network on many servers. Many servers may host many *Agents*, but only one of each type. Thus, the control mechanism can be applied to a widely distributed system.

2.2 The Process Control Description Language ProCDeL

The domain-specific language ProCDeL is introduced by authors for description of controlled processes. The language was developed with two main criteria in mind:

- it must be easy to use by various types of users (from system administrators to skillful end users);
- the language should be used for rather complex process description.

The first criterion sets a requirement for the process description language to have both graphical and textual notation so that processes could be represented as graphs or scripts.

The concept of the process control definition language is similar to BiLingva [12], where a typical state chart diagram (contains state and connection elements) is supplemented with action elements. The process control definition language ProCDeL contains three types of elements: states, events (connections in state charts) and flow control elements (actions in BiLingva). Process flow control elements allow to describe parallel process execution, loops and control over other processes.

2.2.1 An Example of ProCDeL Usage

Let us demonstrate the language elements by example. Fig. 3 describes an electronic clearing payment system. The process starts with event *Clearing day opened*. It must be done no later than at 8:30 in the morning. After the day has been opened, the system starts to process incoming client payment files. Those are processed in parallel. At 14:00 file reception must be stopped and payments may be settled. After it is done, all payments are delivered to recipients.

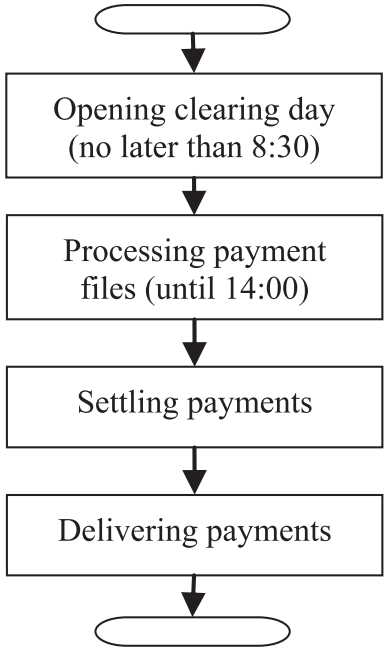


Fig. 3. The clearing process workflow

This process can be described in the process description language as a graph (Fig. 4) or as a textual version. The first event that the control mechanism has to detect is the beginning of the clearing day. This can be done by checking the database for new day event.

Next step in the clearing process workflow (Fig. 3) is file processing. There is another process description named *ReceiveIncomingFiles* made according to this workflow step. It describes file processing for one clearing participant. All clearing participants must be identified before process *ReceiveIncomingFiles* is started. It is done by flow control operation *ControlData (Banks, DBData.Procedure, [CLEAR_GET_PARTICIPANTS]).* After all participants are found, next control flow operation is executed to load file processing processes for each participant.

When all files are received, file reception must be closed. It is identified by state *FILE_RECEPTION_CLOSED* in the process description graph. This state can be reached when all file processing processes are finished. There is one more control added to this state – time 14:00. It means that the control mechanism must check if the state is reached by 14:00.

There are two more events and states following *FILE_RECEPTION_CLOSED*. The first event detects when all of payments are settled. This event corresponds to workflow step *Settling payments*. On this event, the process moves to the next state *PAYMENTS_SETTLED*. There is no time control for this state. The last event in this process description is the event that identifies the end of payment delivery. This event leads the process into the ending state with time control 14:30.

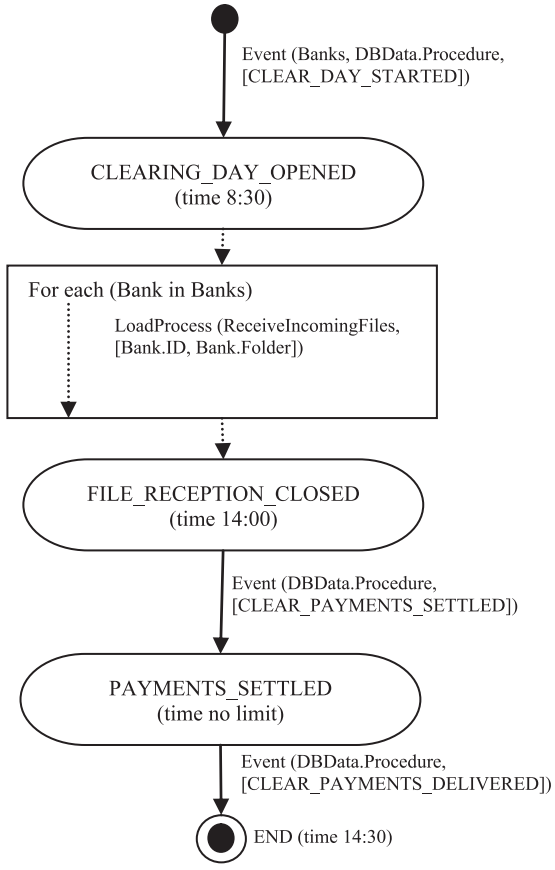


Fig. 4. The clearing process described in the process definition language

All of the processes mentioned above can be described in a textual form using the same language.

Clearing process description in a textual form in ProCDeL

```
process PaymentDay{
    event (Banks, DBData.Procedure,
          [CLEAR_DAY_STARTED]);
    state CLEARING_DAY_OPENED (time 8:30);
    forEach (Bank in Banks) {
          loadProcess (ReceiveIncomingFiles, [Bank.ID,
                                              Bank.Folder]);
    }
    state FILE_RECEPTION_CLOSED (time 14:00);
    event (DBData.Procedure, [CLEAR_PAYMENTS_SETTLED]);
    state PAYMENTS_SETTLED (time no limit);
```

```
      event (DBData.Procedure,
            [CLEAR_PAYMENTS_DELIVERED]);
      state END (time 14:30);
}
```

The example shows just one kind of event (DBData.Procedure – event occurs if the database procedure returns any data); however, there are no limitations to event types in the process description language. As many events as event agents implemented in the control mechanism can be used: for instance, file system event agents, database event agents, e-mail agents etc.

Each event's occurrence returns results that can be used in other events. For instance (Fig. 3), first event in the process *PaymentDay* is type of *DBData.Process* and it calls database procedure *CLEAR_DAY_STARTED*. It returns all of clearing participants and those are loaded in the variable *Banks*. Later the variable *Banks* may be used in other events or control flow statements as an argument. The variable *Banks* was used in the process *PaymentDay* to define *forEach* statement (loop over all list items).

Discussions are still ongoing on,how to describe reporting issues in the process description language. When the control mechanism detects improper process execution according to the process definition, it must send some alarms to the person who is in charge. There could be a rather simple process control with just one type of alarm (for instance, error messages) and one recipient. However, many complex processes running over more than one system could have errors, warnings and notifications with various recipients. Thus, the process description language must have rather flexible control flow expressions to add different types of notifications. These problems will be solved in future developments of the language.

2.2.2 Elements of ProCDeL

Three types of elements are utilized in the process description language: states, events and flow control elements (Fig. 5).

Process description (Fig. 5) has three attributes: process name, schedule for when process may be running and the number of process instances allowed to be running in parallel.

The language introduces three types of process states: *Beginning*, *Ending* and *Intermediate State*, the last two of which are time-controlled states. It means that time control can be done by reaching these states. Time control allows for two types of limits: absolute time (for example, state must be reached by 12:45) and relative distance from other states. The distance may be set in seconds, minutes, hours and days, depending on process specifics. Intermediate states may be identified by unique ids used to specify the acceptable distance between states.

States are connected by *Events*. Each event has event type, arguments and optional event id. Event id may be used in other events or control flow elements to refer to the results returned by the event.

Last group of the elements is *Flow Control* elements. *Cycle* allows to define iterations in the list of items returned as a result of some event. The body of *Cycle* may contain other *States*, *Events* or even *Flow Control* elements. *Load Control* is provided for loading sub-process controls. Those sub-process controls may run synchronously or asynchronously.

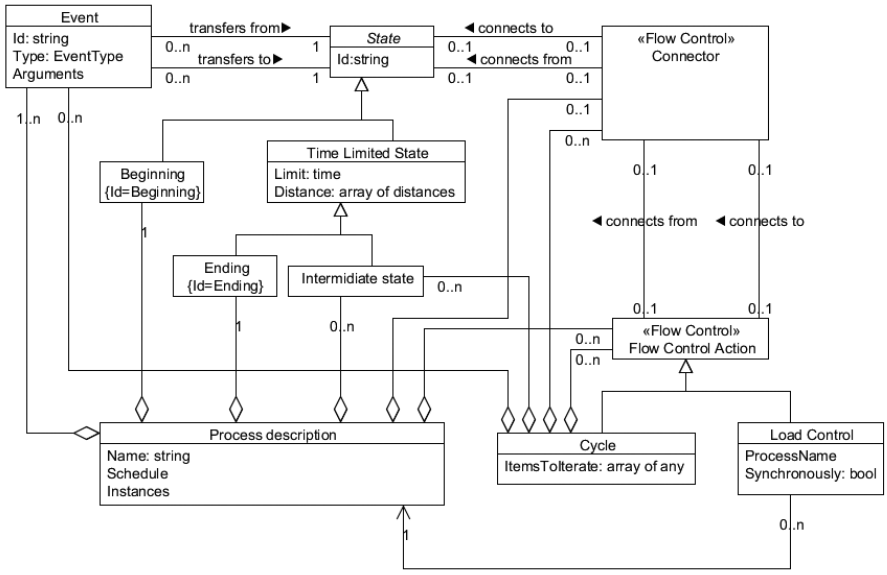


Fig. 5. Elements of the process description language

Conclusion

A new component of a smart technology framework – process control – is being researched. From the process description perspective, the ProCDeL language is ready for the first prototype of the process control mechanism implementation. However, the language must be supplemented with error and warning elements to offer broad potential of information distribution on incorrect and correct process flows. For instance, there can be a process state with two time limits in one process description: when the first limit is reached, the system operators are warned about a possible problem and, when the second limit is reached, error messages are sent.

After the language is supplemented, the first prototype of the process control mechanism will be developed. Most likely this step will make some further changes in the language to make it more usable. The process control mechanism itself is a wide avenue of future research as it is distributed in real time systems. The authors have identified two groups of problems in the process control mechanism:

- problems concerning correct interpretation of event flow by *Central Hub* of the control mechanism;
- technical problems to implement the control mechanism as a reliable distributed real-time system.

The first group of problems is concerned with the algorithms of process control. For instance, there must be an algorithm of how to identify right process control instance if two of the instances from the control instance pool have subscribed for *NewFile* event from the same network resource and one event has arrived. One of the solutions is to move both of processes one step further and keep in mind that one of them could be

rolled back. There are other problems concerning event interpretation in this problem group.

The other problem group of mechanism implementation contains more technical problems: heart beat mechanism implementation to determine if all the agents are up and running, time synchronization and tracking of the order of nearly simultaneous events, and other technical problems. Most of these problems are not unique for the process control mechanism and there are solutions in the distributed server system world.

This paper only introduces the concept of automated process control. Further research on concept prototype implementation will be done.

References

1. J. A. Bergstra, P. Klint. The discrete time TOOLBUS – a software coordination architecture. *Science of Computer Programming*, 31, 1998, pp. 205–229.
2. Z. Bičevska, J. Bičevskis. Smart Technologies in Software Life Cycle. In: J. Münch, P. Abrahamsson (eds.), *Product-Focused Software Process Improvement.* 8th International Conference, PROFES 2007, Riga, Latvia, July 2–4, 2007, LNCS, vol. 4589. Berlin/Heidelberg: Springer-Verlag, 2007, pp. 262–272.
3. K. Rauhvargers, J. Bicevskis. Environment Testing Enabled Software – a Step Towards Execution Context Awareness. In: H.-M. Haav, A. Kalja (eds.), *Databases and Information Systems, Selected Papers from the 8th International Baltic Conference*, vol. 187. IOS Press, 2009, pp. 169–179.
4. K. Rauhvargers. On the Implementation of a Meta-Data Driven Self Testing Model. In: T. Hruška, L. Madeyski, M. Ochodek (eds.), *Software Engineering Techniques in Progress*, Brno, Czech Republic, 2008, pp. 153–166.
5. Z. Bičevska, J. Bičevskis. Applying Smart Technologies in Software Development: Automated Version Updating. In: *Scientific Papers of the University of Latvia, Computer Science and Information Technologies*, vol .733, 2008, pp. 24–37. ISSN 1407-2157.
6. E. Diebelis, V. Takeris, J. Bičevskis. Self-testing – new approach to software quality assurance. In: *Proceedings of the 13th East-European Conference on Advances in Databases and Information Systems (ADBIS 2009)*. Riga, Latvia, 7–10 September, 2009, pp. 62–77.
7. A. G. Ganek, T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, vol. 42, no. 1, 2003, pp. 5–18.
8. R. Sterritt, D. Bustard. Towards an autonomic computing environment. *Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA 2003)*, 2003, pp. 694–698.
9. S. Lightstone. Foundations of Autonomic Computing Development. *Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, 2007, pp. 163–171.
10. Z. Bicevska. Applying Smart Technologies: Evaluation of Effectiveness. *Conference Proceedings of the 2nd International Multi-Conference on Engineering and Technological Innovation (IMETI 2009)*, Orlando, Florida, USA, July 10–13, 2009.
11. Z. Bičevska, J. Bičevskis. Applying Self-Testing: Advantages and Limitations. In: H.-M. Haav, A. Kalja (eds.), *Databases and Information Systems, Selected Papers from the 8th International Baltic Conference*, vol. 187, IOS Press, 2009, pp. 192–202.
12. J. Ceriņa-Bērziņa, J. Bičevskis, Ģ. Karnītis. Information Systems Development Based on Visual Domain-Specific Language BiLingva. Accepted for publication in the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2009), Krakow, Poland, October 12–14, 2009.

MATHEMATICAL FOUNDATIONS

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 207–220 P.

A Modified Spline Interpolation Method for Function Reconstruction from Its Zero-Crossings

Viktorija Solovjova
University of Latvia, 19 Raina bulv., LV-1459, Riga, Latvia
viktorija.solovjova@gmail.com
http://www.lu.lv/df

There are different algorithms for reconstruction of a one-dimensional function from its zero-crossings. However, none of them is stable and computable in real time. Methods for one-dimensional function reconstruction from its generalized zero-crossings based on cubic spline interpolation are introduced in this paper. The main goal is to build an algorithm which is able to reconstruct a smooth function from the generalized zero-crossings as close to the original function as possible. The three main advantages of the approaches described in this paper are reliability, stability, and linear time processing.

Keywords: cubic spline interpolation, one-dimensional function reconstruction, zero-crossings.

1 Introduction

In this paper we are going to explore the task of one argument function reconstruction from its zero-crossings. In literature, a one-dimensional function is often called *one-dimensional signal*. The only argument of the function most often represents time or space. Compression and further reconstruction of a one dimensional function is required in optics, acoustics, crystallography, vision, and many other fields [1]. The literature on this problem describes different approaches to function reconstruction from its zero-crossings - see [1,2,3,4,5].
Venkatesh [1] has attempted to use generalised Hermite polynomials for reconstruction of an unknown function from zero-crossing information. He has also introduced a computational implementation of the algorithm. However, as the author states, it is very unlikely that a real-time solution to a practical reconstruction problem can be achieved with the algorithm. Thus, these results are interesting, but unfortunately unusable in practice.
Boufounos and Baraniuk [2] have assumed that the input data are a sparse signal, and formulated the reconstruction problem as minimization of sparsity. The authors state that the presented algorithm converges in typical cases and produces the correct solution with a very high probability. Yet the method is unstable, and return of a correct answer is not guaranteed.

Mallat [4] proposed to use wavelet zero-crossing representation as a complete signal description. He introduced a reconstruction algorithm based on alternate projections and very accurate reconstruction results were obtained. Later Mallat and Zhong [6] introduced the wavelet maxima representation as an alternative to the wavelet zero-crossing representation. Quite accurate reconstruction results were also demonstrated. Unfortunately, both approaches are unstable, i.e., is the algorithms are not guaranteed to process any kind of input data and return an acceptable answer in real time.

We introduce an algorithm based on the cubic spline interpolation method for function reconstruction. We also explore its several modifications, ranging from the basic, when a small amount of data are extracted from the original function, to more precise and efficient approaches.

In existing reconstruction algorithms, only x values are stored for zero-crossings of the function. In contrast, we extract function value $f(x)$ and sometimes its derivative $f'(x)$ at these points. Zero-crossings usually stand for the points, where the value of the function is 0. We extend this notion. In our interpretation, zero-crossings stand for the points where the value of the function $f(x)$ or the value of its derivative of any order $f^{(n)}(x)$ is 0. Obviously, these extensions give us high reconstruction precision potential.

2 Notations and Definitions

We are going to denote derivatives of the function $f(x)$ in the following way:

$$\begin{array}{ll} f^{(1)}(x) = f'(x) & \text{– first-order derivative of the function } f(x); \\ f^{(2)}(x) = f''(x) & \text{– second-order derivative of the function } f(x); \\ \ldots & \\ f^{(n)}(x) & \text{– n-order derivative of the function } f(x). \end{array} \tag{1}$$

We will call points of one-argument function f at which the function value or its derivative of any order is equal to 0 **zero-crossings**.

$$\text{Zero-crossings of } f(x) = \{x \mid f(x) = 0 \text{ or } f^{(n)}(x) = 0,\ n = 1, 2, \ldots\} \tag{2}$$

Moreover, we will call points at which the first-order derivative of the function is equal to 0 **first-order zero-crossings**, and points at which the second-order derivative of the function is 0 - **second-order zero-crossings**, and so on:

$$\begin{array}{ll} \{x \mid f(x) = 0\} & \text{– zero-order zero-crossings of } f(x); \\ \{x \mid f'(x) = 0\} & \text{– first-order zero-crossings of } f(x), \text{ also local extrema}; \\ \{x \mid f''(x) = 0\} & \text{– second-order zero-crossings of } f(x); \\ \ldots & \\ \{x \mid f^{(n)}(x)\} & \text{– n-order zero-crossings of } f(x). \end{array} \tag{3}$$

3 Reconstruction with Modified Spline Interpolation

In this section we present algorithms for one argument function reconstruction from its zero-crossings. Three different approaches will be used – from the basic approach, trying to remember less information, to extended approaches when a larger amount of data are extracted from the original function. With all the approaches we remember end points data concerning and data on first-order zero-crossings. In extended methods, second-order zero-crossings are also considered. It is possible to exploit higher order zero-crossings and slightly improve the quality of reconstruction, but that influences the amount of zero-crossing data; therefore, this kind of extension will not be considered.

3.1 Obtaining Function Zero-Crossings

If the function is given analytically, then we can calculate its derivatives in a straightforward way. If the function f is discrete, i.e. given as a set of pairs (x, y), we will use numerical differentiation for obtaining the derivatives of the function [7,8]. We assume that points of the function f are distributed evenly, i.e. , the interval between any two adjacent points is a constant value. After calculating all the required function derivatives for each point x, we can compare adjacent values. If the sign of the derivative of the function is changing at the point x, it is considered a zero-crossing point.

$$\begin{gathered} x_i \; is \; a \; zero-crossing \; of \; order \; n \; \leftrightarrow \\ \leftrightarrow f^{(n)}(x_i) \cdot f^{(n)}(x_{i-1}) < 0 \; and \; |f^{(n)}(x_i)| < |f^{(n)}(x_{i-1})| \\ x_{i-1} \; is \; a \; zero-crossing \; of \; order \; n \; \leftrightarrow \\ \leftrightarrow f^{(n)}(x_i) \cdot f^{(n)}(x_{i-1}) < 0 \; and \; |f^{(n)}(x_i)| \geq |f^{(n)}(x_{i-1})| \end{gathered} \tag{4}$$

3.2 The Cubic Spline Interpolation Method

First of all, we briefly describe plain cubic spline interpolation method. For an explicit explanation of the cubic spline interpolation method, see [9]. The aim of the method is to find a cubic spline $S(x)$ which interpolates the given points $x_1, x_2, ..., x_n$ and the values of the original function at points $f(x_1), f(x_2), ..., f(x_n)$. The cubic spline $S(x)$ is defined as the combination of cubic polynomials $S_i(x)$.

$$S(x) = \begin{cases} S_1(x) = a_1x^3 + b_1x^2 + c_1x + d_1, & if \; x_1 \leq x \leq x_2 \\ S_2(x) = a_2x^3 + b_2x^2 + c_2x + d_2, & if \; x_2 \leq x \leq x_3 \\ ... \\ S_{n-1}(x) = a_{n-1}x^3 + b_{n-1}x^2 + c_{n-1}x + d_{n-1}(x), & if \; x_{n-1} \leq x \leq x_n \end{cases} \tag{5}$$

There are 4 unknown variables in each equation, so we have in total $4n - 4$ coefficients which we have to determine to specify the function $S(x)$. If the coefficients are chosen in a way that $S(x)$ interpolates n given points and $S(x)$, $S'(x)$

and $S''(x)$ are continuous at points $x_2, x_3, ..., x_{n-1}$, then we get an interpolating curve $S(x)$, which is called a *cubic spline*.

The following requirements are used to calculate the coefficients of the function $S(x)$ with the cubic spline interpolation method.

For the inner points x_i, $i = 2, ..., n-1$

S1.1. $S(x)$ interpolates the point $(x_i, f(x_i))$: $S_i(x_i) = f(x_i)$
S1.2. $S(x)$ is continuous at x_i: $S_{i-1}(x_i) = S_i(x_i)$
S1.3. $S'(x)$ is continuous at x_i: $S'_{i-1}(x_i) = S'_i(x_i)$
S1.4. $S''(x)$ is continuous at x_i: $S''_{i-1}(x_i) = S''_i(x_i)$

For the end points x_1 and x_n

S2.1. $S_1(x_1) = f(x_1)$ and $S_{n-1}(x_n) = f(x_n)$
S2.2. $S'_1(x_1) = f'(x_1)$ and $S'_{n-1}(x_n) = f'(x_n)$

The second requirement for the end points (S2.2) expresses the idea of the so-called *clamped* cubic spline, when function derivatives are known at the end points. Both end points give us 4 equations and each inner point gives 4 equations. Thus, we get $4n - 4$ equations in total, which is enough to calculate the unknown coefficients of the cubic spline $S(x)$.

The cubic spline interpolation method almost ideally suits our aim, except for one defect. The fact that all the given points (with the exception of the end points) are generalized zero-crossings of the function is not considered in the method. Let us see an example where all the inner points are local extrema (first-order zero-crossings). The method searches for a curve which interpolates the given points. We can get inappropriate results like in the example shown in Fig. 1. As algorithm does not take into account local extrema, the reconstructed function can be completely different; however, it interpolates the given points. That is why the cubic spline interpolation method cannot be used without changing it.

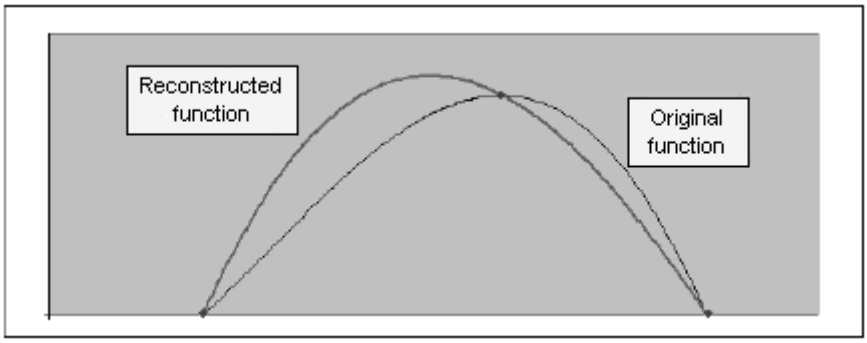


Fig. 1. Reconstruction with the plain spline interpolation method can cause inappropriate results

3.3 Function Reconstruction

Let us decide which points and which values at these points we are going to extract from the original function. We take first-order zero-crossings, i.e. local maxima and minima of the function. We remember x value and function value $f(x)$ at these points.

$$\{x_i, f(x_i) | f'(x_i) = 0, 1 \leq i < n\} \tag{6}$$

With the extended approaches, we also take second-order zero-crossings. For these points, we remember three values: x value, function value $f(x)$ and first derivative value $f'(x)$. Although not all values will be used in some approaches.

$$\{x_i, f(x_i), f'(x_i) | f''(x_i) = 0, 1 \leq i < n\} \tag{7}$$

As far as end points are concerned, it is enough to extract three values for each end point – x value, function value $f(x)$, and first derivative value $f'(x)$. Thus, we assume that we extract the following values.

$$x_1, f(x_1), f'(x_1), x_n, f(x_n), f'(x_n) \tag{8}$$

However constructing the algorithm we can get rid of some if they are redundant We now introduce three different approaches for function reconstruction from its zero-crossings.The first is a basic approach and only first-order zero-crossings are considered. The second-order zero-crossings are taken into account in the extended approaches also introduced.

Approach 1 (Basic). The following data of the local extrema of the original function and its end points were extracted from the original function:

$\{x_i, f(x_i) | f'(x_i) = 0, 1 \leq i < n\}$ – data of the local extrema,
$x_1, f(x_1), f'(x_1), x_n, f(x_n), f'(x_n)$ – data of the end points.

In the *basic approach* we define spline requirements in a slightly different way than in the plain cubic spline interpolation method. Instead of the requirement (S1.4) for the inner points in the plain cubic spline interpolation method that $S'(x)$ is continuous differentiable – $S''_i(x_i) = S''_{i-1}(x_i)$, we will require the first derivative at local extrema to be 0 (B1.3): thus, these points really are minima or maxima. Thus, the requirements we set for the polynomials are the following.

For the first-order zero-crossings (local extrema)

B1.1. $S(x)$ interpolates the point $(x_i, f(x_i))$: $S_i(x_i) = f(x_i)$
B1.2. $S(x)$ is continuous at x_i: $S_{i-1}(x_i) = S_i(x_i)$
B1.3. $S'(x)$ is 0 at x_i: $S'_i(x_i) = S'_{i-1}(x_i) = 0$

For the end points x_1 and x_n

B2.1. $S_1(x_1) = f(x_1)$ and $S_{n-1}(x_n) = f(x_n)$

B2.2. $S'_1(x_1) = f'(x_1)$ and $S'_{n-1}(x_n) = f'(x_n)$

Both end points give us 4 equations and each first-order zero-crossing point gives us 4 equations. Obviously, the number of first-order zero-crossings is $n-2$. Thus, the total count of equations is $4n-4$, which is the same as the count of unknown variables (cubic spline coefficients).
To obtain the coefficients of the polynomials, we use a linear equation solver. For a small amount of data, the Gaussian elimination method can be used. If the linear equation system is large, an iterative method for sparse linear systems [10] can be used to solve it.

Approach 2 (Extended). The following data of the first- and second-order zero-crossings of the original function and its end points were extracted from the original function:

$\{x_i, f(x_i) | f'(x_i) = 0, 1 \leq i < n\}$ – data of the local extrema,
$\{x_i | f''(x_i) = 0, 1 \leq i < n\}$ – data of the second-order zero-crossings,
$x_1, f(x_1), f'(x_1), x_n, f(x_n), f'(x_n)$ – data of the end points.

We use the same algorithm described in the *basic approach* but with different requirements. In addition we extract data about second-order zero-crossings. We search for a cubic spline $S(x)$ defined as a combination of cubic polynomials $S_i(x)$. For the second-order zero-crossings, we take all the same requirements as in the basic cubic spline interpolation algorithm, but the requirement (S1.1) for a precise function value at points $S(x_i) = f(x_i)$ is replaced with the requirement that second derivative of the function at these points is 0 (E2.3). We assume that function value at these points is not as essential as the information that the second derivative of the function is 0.

For the first-order zero-crossings (local extrema)

E1.1. $S(x)$ interpolates the point $(x_i, f(x_i))$: $S_i(x_i) = f(x_i)$
E1.2. $S(x)$ is continuous at x_i: $S_{i-1}(x_i) = S_i(x_i)$
E1.3. $S'(x)$ is 0 at x_i: $S'_i(x_i) = S'_{i-1}(x_i) = 0$

For the second-order zero-crossings

E2.1. $S(x)$ is continuous at x_i: $S_{i-1}(x_i) = S_i(x_i)$
E2.2. $S'(x)$ is continuous at x_i: $S'_i(x_i) = S'_{i-1}(x_i)$
E2.3. $S''(x)$ is continuous and equal to 0: $S''_i(x_i) = S''_{i-1}(x_i) = 0$

For the end points x_1 and x_n

E3.1. $S_1(x_1) = f(x_1)$ and $S_{n-1}(x_n) = f(x_n)$
E3.2. $S'_1(x_1) = f'(x_1)$ and $S'_{n-1}(x_n) = f'(x_n)$

Both end points give us 4 equations, each first- or second-order zero-crossing also gives us 4 equations. Thus, we get a system of $4n-4$ equations where the unknown variables are the coefficients of the spline polynomials. When the linear

equation system is solved, it gives us the coefficients of the spline and allows to define the function $S(x)$.

Approach 3 (Extended Precise). The following data of the first- and second-order zero-crossings of the original function and its end points were extracted from the original function:

$\{x_i, f(x_i) | f'(x_i) = 0, 1 \leq i < n\}$ – data of the local extrema,
$\{x_i, f(x_i), f'(x_i) | f''(x_i) = 0, 1 \leq i < n\}$ – data of the second-order zero-crossings,
$x_1, f(x_1), f'(x_1), x_n, f(x_n), f'(x_n)$ – data of the end points.

This approach is very similar to the previous approach. But here instead of requiring that the second derivative of the function $S''(x)$ is continuous and equal to 0 at the second-order zero-crossings (E2.3), we give the exact value of the function (P2.3) and the exact value of the first derivative of the function (P2.4) at these points.

For the first-order zero-crossings (local extrema)

P1.1. $S(x)$ interpolates the point $(x_i, f(x_i))$: $S_i(x_i) = f(x_i)$
P1.2. $S(x)$ is continuous at x_i: $S_{i-1}(x_i) = S_i(x_i)$
P1.3. $S'(x)$ is 0 at x_i: $S'_i(x_i) = S'_{i-1}(x_i) = 0$

For the second-order zero-crossings

P2.1. $S(x)$ is continuous at x_i: $S_{i-1}(x_i) = S_i(x_i)$
P2.2. $S'(x)$ is continuous at x_i: $S'_i(x_i) = S'_{i-1}(x_i)$
P2.3. $S(x)$ interpolates the point $(x_i, f(x_i))$: $S_i(x_i) = f(x_i)$
P2.4. $S'(x)$ interpolates the point $(x_i, f'(x_i))$: $S'_i(x_i) = f'(x_i)$

For the end points x_1 and x_n

P3.1. $S_1(x_1) = f(x_1)$ and $S_{n-1}(x_n) = f(x_n)$
P3.2. $S'_1(x_1) = f'(x_1)$ and $S'_{n-1}(x_n) = f'(x_n)$

Similarly as in the previous approaches, each inner point gives us 4 equations, but each end point gives 2. Thus, the total count of equations is $4n - 4$. Having solved this linear system, we get the values of the coefficients of the polynomials and hence the definition of the function we are looking for – $S(x)$.

3.4 Analysis of the Approaches

The *extended* and *extended precise* approaches are expected to be more precise as they demand more information about the original function. Nevertheless, now we see that actually the *basic approach* is more reliable because it guarantees that all the local extrema of the reconstructed function are exactly the same as the local extrema of the original function. The extended approaches cannot guarantee such accuracy.

Theorem 1. Let $f(x)$ be the original function and $S(x)$ – the function reconstructed from its zero-crossings with the *basic approach*. Then for $x \in [x_2, x_{n-1}]$, the local extrema of the function $S(x)$ are exactly the same as the local extrema of the original function $f(x)$, i.e., if $x \in [x_2, x_{n-1}]$, then

$$\begin{array}{l}(a)\ S'(x) = 0 \Leftrightarrow f'(x) = 0, \\ (b)\ S'(x) = 0 \Rightarrow S(x) = f(x).\end{array} \tag{9}$$

Proof. (a) The proof is acquired in two parts. Let us start with $f'(x) = 0 \Rightarrow S'(x) = 0$. As x is a first-order zero-crossing $x = x_i$ for some i, the linear equation system contains the following equation.

$$S'_i(x_i) = S'_{i-1}(x_i) = 0 \tag{10}$$

Then from Equation 10 and the definition of the cubic spline $S(x)$ it follows that $S'(x) = 0$.

Now let us prove the second part – that if $x \in [x_2, x_{n-1}]$, then $S'(x) = 0 \Rightarrow f'(x) = 0$. From the definition of the *basic approach* we know that all inner points are first-order zero-crossings:

$$f'(x_i) = S'_i(x_i) = 0 \text{ for } i = 2, ..., n-1. \tag{11}$$

This means that each function $S_i(x)$ for $i = 2, ..., n-1$ has two local extrema points – one at x_i and the other at x_{i+1}. Since all the functions $S_i(x)$ are cubic polynomials, they cannot contain more than 2 local extrema. This means that the only local extrema in each interval $[x_i, x_{i+1}]$ are x_i and x_{i+1}. All these intervals form the interval $[x_2, x_{n-1}]$. Thus, we can conclude that the only points where $S'(x) = 0$ are the points where $f'(x) = 0$ (the end points of the subintervals). Thus, for all $x \in [x_2, x_{n-1}] : S'(x) = 0 \Rightarrow f'(x) = 0$.

(b) We have proven that $S'(x) = 0 \Rightarrow f'(x) = 0$ in (a). Therefore, if $S'(x) = 0$, then x is a first-order zero-crossing of the original function - $x = x_i$ for some i, and the linear equation system contains the following equation.

$$S_i(x_i) = S_{i-1}(x_i) = f(x_i) \tag{12}$$

Then from Equation 12 and the definition of the cubic spline $S(x)$ it follows that $S(x_i) = f(x_i)$.

Theorem 1 states that in almost all of the domain (apart from two end subintervals), all the local extrema of the function reconstructed with the *basic approach* are exactly the same as the local extrema of the original function. Thus, the *basic approach* guarantees that the reconstructed function exactly reflects all the local extrema of the original function.

Now let us analyze the performance of the approaches. Both *basic* and *extended precise* approaches generate a large linear equation system. Instead, a set of linear equation systems of a constant size could be generated. For each first- and second-order zero-crossing and for the end points, we give a precise value of the function and a precise value of the derivative with both approaches.

$$\begin{array}{l}S_{i-1}(x_i) = S_i(x_i) = f(x_i) \\ S'_{i-1}(x_i) = S'_i(x_i) = f'(x_i)\end{array} \tag{13}$$

Thus, the coefficients of every two adjacent polynomials $S_{i-1}(x)$ and $S_i(x)$ do not depend on each other. That is why a separate linear equation system can be generated for each polynomial $S_i(x)$. Hence, each of these linear equation systems is of a constant size and can be computed in a constant time. It is obvious that the whole set of linear equation systems can be solved in linear time $O(n)$, where n is the count of the first- and second-order zero-crossings. Thus, there are linear time algorithms in both the *basic approach* and *extended precise approach*.

In the *extended approach*, the polynomials are connected to each other at the second-order zero-crossings. Nevertheless, the linear equation system can still be split into several small systems – one for each interval from one first-order zero-crossing to the next.

3.5 Results

In this sub-section we present the examples of function reconstruction using all three approaches. Different functions, starting with smooth analytically defined functions up to more complex functions, are reconstructed with the approaches described above. The reconstruction accuracy of each approach is discussed and compared to others.

$\boldsymbol{f(x) = (x-2)(x-9)(x-15)}$ The function is smooth and simple, it has only two local extrema points since it is a cubic polynomial. This function is interesting because it is defined in the same way as the function we get from reconstruction. That is why we can expect that the reconstruction quality will be very high. The results of the reconstruction are shown in Fig. 2. In figures below the original function is drawn black, reconstructed functions – gray, first-order zero-crossings (local extrema) are shown as black points, and second-order zero-crossings as gray points. The reconstruction quality in this case appeared to be ideal with each approach because of the fact that the data were extracted from the function of the same nature as the function we are trying to construct.

$\boldsymbol{f(x) = (2x-13)(\sin\frac{x}{5} + 2\cos\frac{x}{3} - 3\cos(\frac{x}{2} - 3))}$ Let us take a little more complex, however, also a smooth function. As shown in Fig. 3, there is a slight difference in the results of the approaches. The *basic approach* gives a slight deviation from the original function, but the *extended precise approach* – the best result. However, the difference between them is very small. Again all three methods give us good results.

$\boldsymbol{f(x) = e^{-x^2}}$ Let us take a Gaussian function whose nature is far from the nature of polynomial functions. The results of the three approaches are shown in Fig. 4. The difference between all three is remarkable. The *extended precise approach* gives a very good result in this case. The other two approaches also give

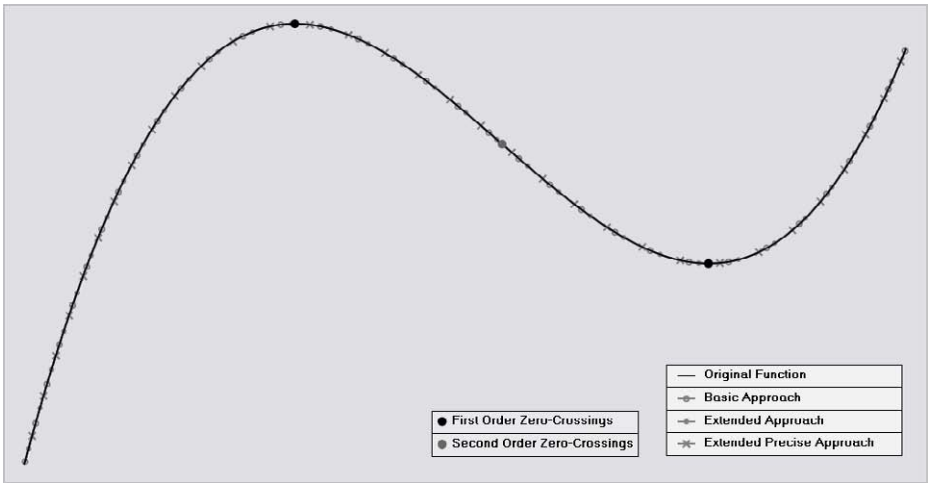


Fig. 2. Reconstruction of the function $f(x) = (x-2)(x-9)(x-15)$ at $x \in [0, 16]$ with the *basic approach*, *extended approach*, and *extended precise approach*

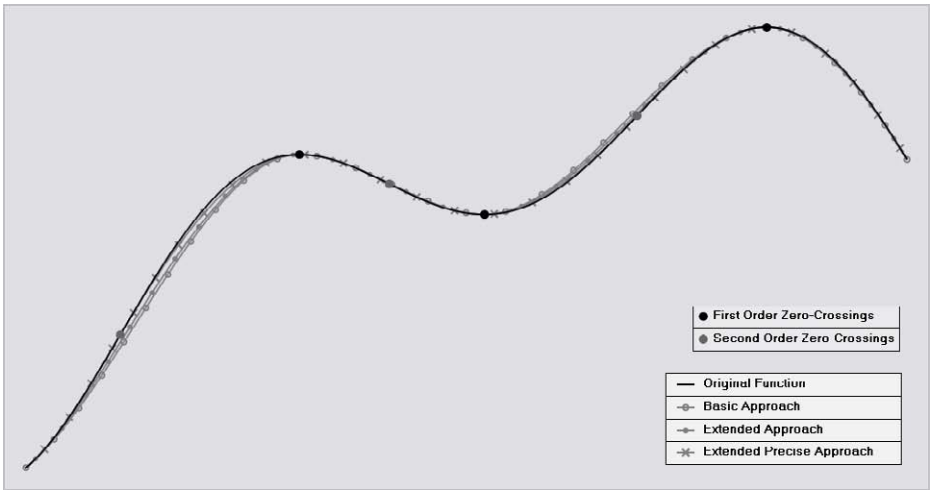


Fig. 3. Reconstruction of the function $f(x) = (2x-13)(\sin\frac{x}{5}+2\cos\frac{x}{3}-3\cos(\frac{x}{2}-3))$ at $x \in [0, 16]$ with the *basic approach*, *extended approach*, and *extended precise approach*

acceptable results; however, the reconstructed functions are a little smoothed out in comparison to the original function.

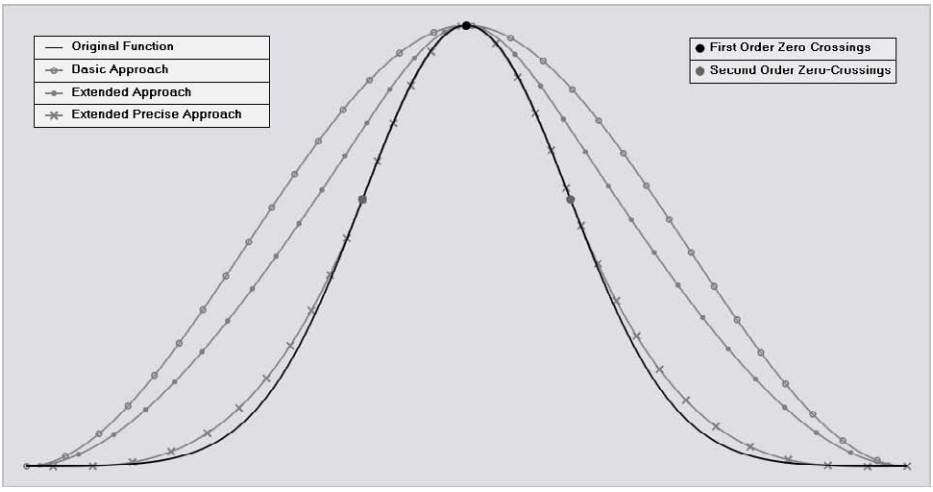


Fig. 4. Reconstruction of the function $f(x) = e^{-x^2}$ at $x \in [-3, 3]$ with the *basic approach*, *extended approach*, and *extended precise approach*

$\boldsymbol{f(x) = e^{-x^4}}$ We also take a modification of a Gaussian function with a flattened top. As we see in Fig. 5, all the approaches again give good results, yet the result of the *extended precise approach* is the most accurate.

A function with sharp angles. Let us also see the example of reconstruction of a broken-line function. This function has very little in common with polynomial functions; therefore, not very impressive results were expected. However, Fig. 6 shows that even such case is processed with a good precision. Again the *extended precise approach* gives the most accurate result. Nevertheless, the *extended approach* and the *basic approach* give almost the same result; thus, in this case the *extended approach* is not a significant improvement on the *basic approach*.

The examples above show that the quality of reconstruction is improving with an increasing amount of data stored. The most accuracy is achieved with the *extended precise approach*. However, we have seen in the analysis section that the *extended* approaches do not guarantee that all the local extrema will be identical to the local extrema of the original function. Therefore, we recommend to use the *basic approach* due to its reliability and stability.

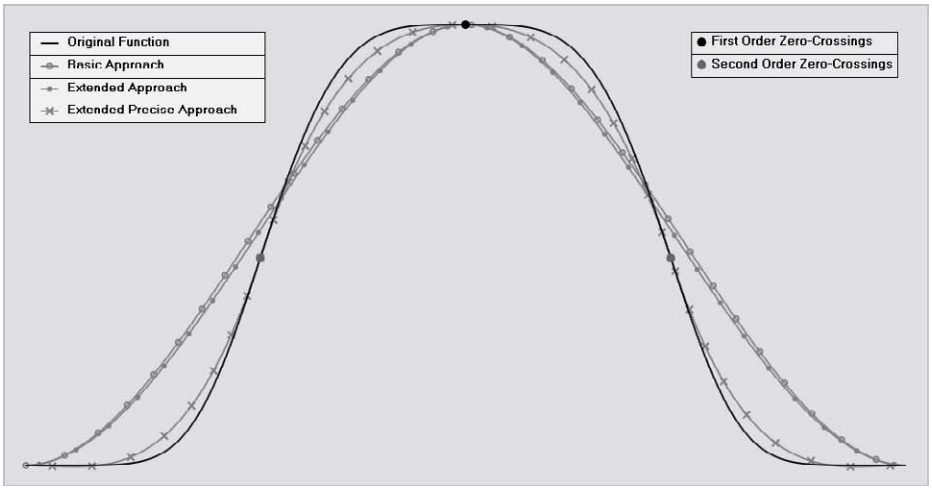


Fig. 5. Reconstruction of the function $f(x) = e^{-x^4}$ at $x \in [-2, 2]$ with the *basic approach*, *extended approach*, and *extended precise approach*

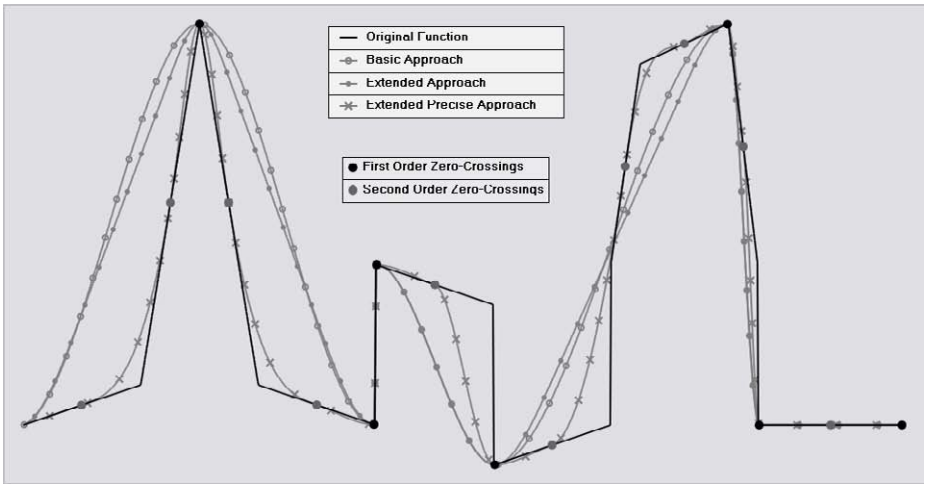


Fig. 6. Reconstruction of the broken line function with the *basic approach*, *extended approach*, and *extended precise approach*

4 Conclusion

We have introduced three approaches for function reconstruction from its zero-crossings. The most interesting among them is the *basic approach* since it is precise, stable, and computable in linear time. Previously known approaches focus on a particular type of functions, and the algorithms work well with this particular type of functions. However, in general, they are unstable and usually do not guarantee return of the result in real time.

Three main advantages of the *basic approach* can be pointed out. First, it guarantees return of the result in linear time. Second, the approach is precise, it perfectly reconstructs all the local extrema of the original function. Third, the approach is stable: it always gives the result even if the input data contain errors, i.e., the approach is tolerant to input data inaccuracy.

The presented approaches can also be extended for two-dimensional functions. Accurate reconstruction of two-dimensional functions from a relatively small set of data is useful in the area of image compression because any image can be represented as a two-argument function or a combination of several two-dimensional functions.

Reconstruction of two-dimensional functions from zero-crossings should be explored in further research. A wide range of research papers, for example, [11,12,13] are dedicated to two-dimensional function reconstruction from zero-crossings, actually from the so-called edges and ridges. The results of these methods are very good, yet the stability of the algorithms has not been proven.

The methods for reconstruction of one-dimensional functions presented in this paper appeared to be very efficient and practically useful. Therefore, we expect that these methods can most definitely be successfully extended for two-dimensional functions.

References

1. Y. V. Venkatesh. Hermite polynomials for signal reconstruction from zero-crossings. Part 1: One-dimensional signals. *IEE Proceedings I of Communications, Speech and Vision*, vol. 139 (6), 1992, pp. 587–596.
2. P. T. Boufounos, R. G. Baraniuk. Reconstructing sparse signals from their zero crossings. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2008)*, pp. 3361–3364.
3. Z. Berman, J. S. Baras. Properties of the multiscale maxima and zero-crossings representations. *IEEE Transactions on Signal Processing*, vol. 41, 1993, pp. 3216–3231.
4. S. Mallat. Zero-crossings of wavelet transform. *IEEE Transactions on Information Theory*, vol. 37, 1991, pp. 1019–1033.
5. R. Hummel, R. Moniot. Reconstructions from zero crossings in scale space. *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 37, 1989, pp. 2111–2130.
6. S. Mallat, S. Zhong. Characterization of signals from multiscale edges. *IEEE Transactions On Pattern Analysis and Machine Intelligence*, vol. 14 (7), 1992, pp. 710–732.

7. A. Iserles. A First Course in the Numerical Analysis of Differential Equation/ *Cambridge Texts in Applied Mathematics*, 2nd edition. Cambridge University Press, 2008.
8. J. H. Mathews, K. D. Fink. Numerical Methods Using Matlab, 4th edition. Prentice-Hall Pub., 2004.
9. G. D. Knott. Interpolating Cubic Splines. *Progress in Computer Science and Applied Logic (PCS)*. Boston: Birkhäuser, 1999.
10. Y. Saad. Iterative Methods for Sparse Linear Systems, 2d edition. Society for Industrial and Applied Mathematics, 2000.
11. J. H. Elder. Are Edges Incomplete? *International Journal of Computer Vision*, vol. 34 (2/3), 1999, pp. 97–122.
12. E. Barth, T. Caelli, C. Zetzsche. Image Encoding, Labeling and Reconstruction from Differential Geometry. *CVGIP: Graphical Models and Image Processing*, vol. 55 (6), 1993, pp. 428–446.
13. S. Carlsson. Sketch based coding of grey level images. *Signal Processing*, vol. 15, 1988, pp. 57–83.

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 221–225 P.

A Note on the Optimality of the Grover's Algorithm

Nikolajs Nahimovs, Alexander Rivosh*
Faculty of Computing, University of Latvia
nikolajs.nahimovs@lais.lv, aleksandrs.rivoss@lais.lv

Abstract. The Grover's algorithm is a quantum search algorithm solving the unstructured search problem in about $\frac{\pi}{4}\sqrt{N}$ queries. It is known to be optimal - no quantum algorithm can solve the problem in less than the number of steps proportional to $\sqrt{N}$ [3]. Moreover, for any number of queries up to about $\frac{\pi}{4}\sqrt{N}$, the Grover's algorithm ensures the maximal possible probability of finding the desired element [2].
However, it is still possible to reduce the average number of steps required to find the desired element by ending the computation earlier and repeating the algorithm if necessary. This fact was mentioned by Christof Zalka as a short remark on analysis of the Grover's algorithm [2]. Our article gives a detailed description of this simple fact.

1 Unstructured Search

Suppose we have a function

$$f : \{0,1\}^n \to \{0,1\}$$

given by a black box. The unstructured search problem is to find a string $x \in \{0,1\}^n$ such that $f(x) = 1$, or to conclude that no such x exists if f is identical to 0.

It is easy to see that a deterministic algorithm would need to make $N = 2^n$ queries to the blackbox in the worst case (to distinguish the case where f is identical to 0 from any of the cases where there is a single x for which $f(x) = 1$). It can be shown [4] that probabilistically we also need $\Omega(N)$ queries to solve the problem. In contrast, a quantum computer can solve the problem using $O(\sqrt{N})$ queries.

2 The Grover's Quantum Search Algorithm

The Grover's algorithm is a quantum search algorithm that solves the unstructured search problem in about $\frac{\pi}{4}\sqrt{N}$ queries. We will give a brief description of the algorithm. For a detailed description, see [1] or [4].

* Supported by Unversity of Latvia grant ZB01-100 and ESF.

The Grover's algorithm

1. Let X be an n-qubit quantum register with initial state $|0^n\rangle$. Perform $H^{\otimes n}$ on X.
2. For k times (k will be specified later), apply to the register X the transformation $G = DQ$, where D is a rotation about average [1] and Q is a query transformation $Q|x\rangle = (-1)^{f(x)}|x\rangle$.
3. Measure X and output the result.

For the purposes of analysis define two sets of strings:

$$A = \{x \in \{0,1\}^n : f(x) = 1\}$$

$$B = \{x \in \{0,1\}^n : f(x) = 0\}.$$

We will think of the set A as the set of "good" strings; the goal of the algorithm is to find one of these strings. The set B contains all the "bad" strings that do not satisfy the search criterion.

Let $a = |A|$ and $b = |B|$. Define states

$$|A\rangle = \frac{1}{\sqrt{a}} \sum_{x \in A} |x\rangle \quad \text{and} \quad |B\rangle = \frac{1}{\sqrt{b}} \sum_{x \in B} |x\rangle$$

which are both unit vectors and are orthogonal to one another.

The initial state of the register X is

$$|X\rangle = H^{\otimes n}|0^n\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle = \sqrt{\frac{a}{N}}|A\rangle + \sqrt{\frac{b}{N}}|B\rangle.$$

Calculations show that the transformation G changes states $|A\rangle$ and $|B\rangle$ as follows:

$$G|A\rangle = \left(1 - \frac{2a}{N}\right)|A\rangle - \frac{2\sqrt{ab}}{N}|B\rangle$$

$$G|B\rangle = \frac{2\sqrt{ab}}{N}|A\rangle - \left(1 - \frac{2b}{N}\right)|B\rangle.$$

As $\sqrt{\frac{a}{N}}^2 + \sqrt{\frac{b}{N}}^2 = 1$ there exists an angle θ that satisfies

$$\sin\theta = \sqrt{\frac{a}{N}} \quad \text{and} \quad \cos\theta = \sqrt{\frac{b}{N}}.$$

Using this notation, the initial register X state can be written as

$$|X\rangle = \sin\theta\ |A\rangle + \cos\theta\ |B\rangle$$

and the transformation G as

$$G\,|A\rangle = \cos 2\theta\ |A\rangle - \sin 2\theta\ |B\rangle$$

$$G\,|B\rangle = \sin 2\theta\ |A\rangle + \cos 2\theta\ |B\rangle$$

which is simply a rotation by angle 2θ in the space spanned by $|A\rangle$ and $|B\rangle$. T his implies that after k iterations of G, the state of X will be

$$\sin((2k+1)\theta)\ |A\rangle + \cos((2k+1)\theta)\ |B\rangle$$

The goal of the algorithm is to measure some element $x \in A$, so we would like the state of X to be as close to $|A\rangle$ as possible. If we want

$$\sin((2k+1)\theta) \approx 1$$

then

$$(2k+1)\theta \approx \frac{\pi}{2}$$

will suffice, so we should choose

$$k \approx \frac{\pi}{4\theta} - \frac{1}{2}\,.$$

Suppose $a = 1$. Then

$$\theta = \sin^{-1}\sqrt{\frac{1}{N}} \approx \frac{1}{\sqrt{N}}$$

so

$$k = \lfloor \frac{\pi}{4}\sqrt{N} \rfloor$$

is a reasonable choice for the algorithm.

In the general case the situation is more challenging. However, it can be shown that $O(\sqrt{\frac{N}{a}})$ queries are still enough to find an $x \in A$

[4].

3 Ending the Computation Earlier

Suppose we have an algorithm which gives a correct answer with some probability p. To obtain the correct answer, we need to repeat it $\frac{1}{p}$ times on the average. If the algorithm's running time is k, repeating it gives the average running time of $\frac{k}{p}$.

In the previous section we have shown that after k steps, the state of the Grover's algorithm is

$$\sin((2k+1)\theta)\ |A\rangle + \cos((2k+1)\theta)\ |B\rangle\,.$$

The amplitude of the correct answer grows proportionally to $\sin(2k\theta) \approx \sin(\frac{2k}{\sqrt{N}})$, therefore, the probability to obtain the correct answer grows proportionally to $\sin^2(\frac{2k}{\sqrt{N}})$. To get rid of N, let us scale k from $[0, \frac{\pi}{4}\sqrt{N}]$ to $[0, 1]$. That is, let the running time of the original algorithm be 1 and let k represent the fraction of steps completed by the algorithm. The probability to obtain the correct answer becomes $p = \sin^2(\frac{\pi k}{2})$.

Now, if we stop the computation at the moment k, the average running time of the algorithm will be

$$\frac{k}{p} = \frac{k}{\sin^2\left(\frac{\pi k}{2}\right)}.$$

If $k \in [0, 0.5)$ then

$$\sin^2\left(\frac{\pi k}{2}\right) < k$$

and

$$\frac{k}{p} = \frac{k}{\sin^2\left(\frac{\pi k}{2}\right)} > 1.$$

Therefore, the average running time is greater than in the original algorithm. If $k = 0.5$, then

$$\sin^2\left(\frac{\pi k}{2}\right) = 0.5$$

and

$$\frac{k}{p} = \frac{k}{\sin^2\left(\frac{\pi k}{2}\right)} = 1.$$

The average running time is the same as in the original algorithm. If $k \in (0.5, 1]$, then

$$\sin^2\left(\frac{\pi k}{2}\right) > k$$

and

$$\frac{k}{p} = \frac{k}{\sin^2\left(\frac{\pi k}{2}\right)} < 1.$$

Therefore, the average running time is less than in the original algorithm.

The optimal moment to end the computation is the minimum of the $\frac{k}{p}$ function.

$$\left(\frac{k}{p}\right)' = \left(\frac{k}{\sin^2\left(\frac{\pi k}{2}\right)}\right)' = \frac{\sin^2(\frac{\pi k}{2}) - k \cdot 2 \cdot \sin(\frac{\pi k}{2}) \cdot \cos(\frac{\pi k}{2}) \cdot \frac{\pi}{2}}{\sin^4(\frac{\pi k}{2})}$$

As $\sin(\frac{\pi k}{2}) \neq 0$,

$$\sin^2\left(\frac{\pi k}{2}\right) = 2 \cdot \sin\left(\frac{\pi k}{2}\right) \cdot \cos\left(\frac{\pi k}{2}\right) \cdot \frac{\pi k}{2}$$

or

$$\pi k = \tan\left(\frac{\pi k}{2}\right).$$

The equation has an infinite number of solutions. We are interested in a solution with $k \in (0.5, 1)$. Numeric calculation gives $k \approx 0.74202$ and the average running time $\frac{k}{p} \approx 0.87857$. i.e., is the average number of steps can be reduced by approximately 12.14%.

4 Conclusions

We have shown how to reduce the average number of the steps of the Grover's algorithm by approximately 12.14%. The same argument can be applied to a wide range of other quantum query algorithms, such as amplitude amplification, some variants of quantum walks, and NAND formula evaluation, etc: namely is to all algorithms that can be analysed similarly based on rotation from a "bad" to "good state".

References

1. Lov Grover
   A fast quantum mechanical algorithm for database search.
   Proceedings, 28th Annual ACM Symposium on the Theory of Computing (STOC), May 1996, pages 212-219
   also quant-ph/9605043
2. Christof Zalka
   Grovers quantum searching algorithm is optimal.
   quant-ph/9711070
3. C. Bennett et al.
   Strengths and Weaknesses of Quantum Computing.
   SIAM Journal on Computing (special issue on quantum computing) volume 26, number 5, pages 1510-1523.
   also quant-ph/9701001
4. John Watrous
   Quantum Computation.
   Lecture course "CPSC 519/619", University of Calgary, 2006.
   http://www.cs.uwaterloo.ca/ watrous/lecture-notes.html

SCIENTIFIC PAPERS, UNIVERSITY OF LATVIA, 2010. Vol. 756
COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES 227–247 P.

Quantum Algorithms for Computing the Boolean Function *AND* and Verifying Repetition Code

Alina Vasilieva[1]
Faculty of Computing, University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
Alina.Vasilieva@gmail.com

Quantum algorithms can be analyzed in a query model to compute Boolean functions. Function input is provided in a black box, and the aim is to compute the function value using as few queries to the black box as possible. In this paper we present two quantum algorithms. The first algorithm computes the Boolean function *AND* of two bits using one query with a probability p=4/5. It is also described how to extend this algorithm to compute $AND(f_1,f_2)$, where f_1 and f_2 are arbitrary Boolean functions. The second algorithm can be used for verification of the repetition code for error detection. A repetition code is an error detection scheme that repeats each bit of the original message r times. After a message with redundant bits is transmitted via a communication channel, it must be verified. The verification procedure can be interpreted as an application of a query algorithm, where input is a message to be checked. Classically, for an N-bit message, values of all N variables must be queried. We present an exact quantum algorithm that uses only N/2 queries in the case when r=2.

Keywords: quantum computing, quantum query algorithms, complexity theory, Boolean functions, algorithm design.

1 Introduction

Quantum computing is an exciting alternative way of computation based on the laws of quantum mechanics. This branch of computer science is developing rapidly; various computational models exist, and this is a study of one of them.

Let $f(x_1,x_2,...,x_N):\{0,1\}^N \to \{0,1\}$ be a Boolean function. We consider the black box model (also known as the query model), where a black box contains the input $X=(x_1,x_2,...,x_N)$ and can be accessed by querying x_i values. The goal is to compute the value of the function. The complexity of a query algorithm is measured by the number of questions it asks. The classical version of this model is known as *decision trees* [1]. This computational model is widely applicable in software engineering. For instance, a database can be considered a black box, and, to speed up application performance, the goal is to reduce the number of database queries.

Quantum query algorithms can solve certain problems faster than classical algorithms. The best known and at the same time the simplest exact quantum algorithm for a total Boolean function was designed for the *XOR* function with N/2

[1] This research is supported by the European Social Fund project No. 2009/0138/1DP/1.1.2.1.2/09/IPIA/VIAA/004, No. ESS2009/77.

questions versus N questions required by classical algorithm [2]. The quantum query model differs from the quantum circuit model [2, 3, 4], and algorithm construction techniques for this model are less developed. The problem of quantum query algorithm construction is very significant. Although there are many lower-bound and upper-bound estimations of quantum query algorithm complexity [2, 5, 6, 7], there are very few examples of original quantum query algorithms.

This paper consists of two parts, in which algorithm construction results for two different computational problems are presented.

In the first part of this paper, we consider computing the Boolean function *AND*. First, we demonstrate a bounded-error quantum query algorithm, which computes Boolean function $AND(x_1, x_2) = x_1 \wedge x_2$ with one query and probability $p = 4/5$. This is better than the best possible classical probabilistic algorithm, where the probability to obtain correct result is $p = 2/3$. Then we extend our approach and formulate a general method for computing a composite Boolean function $AND(f_1,f_2)$, where f_1 and f_2 are arbitrary Boolean functions. In particular, we explicitly show how an N-variable Boolean function $AND_N(x_1,...,x_N) = x_1 \wedge x_2 \wedge ... \wedge x_N$ can be computed by the quantum bounded-error algorithm with a probability p=4/5 using N/2 queries.

In the second part of this paper, we present an exact quantum query algorithm for resolving a specific problem. The task is to verify a codeword message that has been encoded using the *repetition code* for detecting errors [8] and has been transmitted across a communication channel. The considered repetition code simply duplicates each bit of the message. The verification procedure can be considered to be an application of a query algorithm, where the codeword to be checked is contained in a black box. To verify the message in the classical way, we would need to access all bits. That is, for a codeword of length N, all N queries to the black box would be required. We present an exact quantum query algorithm that requires only N/2 queries.

An exact algorithm always produces a correct answer with 100% probability. Another variation is to use a bounded-error model, where an error margin $\varepsilon < 1/2$ is allowed. It is well-known that in the bounded-error model, a large difference between classical and quantum computation is possible. The complexity gap between the best known classical algorithm and quantum algorithm can be exponential, as, for instance, in the case of the Shor's algorithm [9]. Another famous example is the Grover's search algorithm that achieves a quadratic speed-up [10]. However, in certain types of computer software, we cannot allow even a small probability of error, for example, in spacecraft, aircraft, or medical software. For this reason, the development of exact algorithms is extremely important.

Regarding exact quantum algorithms, the maximum speed-up achieved as of now is half the number of queries compared with a classical deterministic case[2] [11]. The major open question is: is it possible to reduce the number of queries by more than 50%? In this paper, we present an algorithm that achieves the borderline gap of N/2 versus N.

---

[2] Exact quantum algorithm with complexity $Q_E(f) < D(f)/2$ is not yet discovered for a total Boolean function. For partial Boolean functions this limitation can be exceeded. An excellent example is the Deutsch-Jozsa algorithm [12, 13].

2 Preliminaries

This section contains definitions and provides theoretical background on the subject. First, we describe classical decision trees and show how to compute a simple Boolean function in this model. Next, we provide a brief overview of the basics of quantum computing. Finally, we describe the quantum query model that is the subject of this paper.

2.1 Classical Decision Trees

The classical version of the query model is known as *decision trees* [1]. A black box contains the input $X = (x_1, x_2, ..., x_N)$ and can be accessed by querying x_i values. The algorithm must allow to determine the value of a function correctly for arbitrary input. The complexity of the algorithm is measured by the number of queries on the worst-case input. For more details, see the survey by Buhrman and de Wolf [1].

Definition 1 [1]. *The **deterministic complexity** of a function f, denoted by **D(f)**, is the maximum number of questions that must be asked on any input by a deterministic algorithm for f.*

Definition 2 [1]. *The **sensitivity** $s_x(f)$ **of f on input** $(x_1,x_2,...,x_N)$ is the number of variables x_i with the following property: $f(x_1,...,x_i,...,x_N) \neq f(x_1,...,1-x_i,...,x_N)$. The **sensitivity of f** is* $s(f) = \max_x s_x(f)$.

It has been proven that $D(f) \geq s(f)$ [1].

Figure 1 demonstrates a classical deterministic decision tree, which computes $MAJORITY_3(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. In this figure, circles represent queries, and rectangles represent output. It is easy to see that the third query is necessary if values of first two queried variables are different: $D(MAJORITY_3(x_1, x_2, x_3)) = 3$.

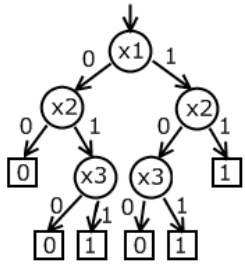


Fig. 1. Classical deterministic decision tree for computing $MAJORITY_3(x_1, x_2, x_3)$

As in many other models of computation, the power of randomization can be added to decision trees [1]. A *probabilistic decision tree* may contain internal nodes with a probabilistic branching, i.e., multiple arrows exiting from this node, each one labeled with a probability for algorithm to follow that way. The total sum of all probabilities assigned to arrows in a probabilistic branching is supposed not to exceed 1. The result

of a probabilistic decision tree is not determined by the input X with certainty anymore. Instead, there is a probability distribution over the set of leaves. The total probability to obtain a result $b \in \{0,1\}$ after the execution of an algorithm on certain input X equals the sum of probabilities for each leaf labeled with b to be reached. The total probability of an algorithm to produce the correct result is the probability on the worst-case input.

2.2 Quantum Computing

This section briefly outlines the basic notions of quantum computing that are necessary to define the computational model used in this paper. For more details, see the textbooks by Nielsen and Chuang [3] and Kaye et al [4].

An n-dimensional quantum pure state is a unit vector in a Hilbert space. Let $|0\rangle,|1\rangle,...,|n\text{-}1\rangle$ be an orthonormal basis for C^n. Then, any state can be expressed as $|\psi\rangle=\sum_{i=0}^{n-1} a_i |i\rangle$ for some $a_i \in \mathbb{C}$. Since the norm of $|\psi\rangle$ is 1, we have $\sum_{i=0}^{n-1} |a_i|^2 = 1$. States $|0\rangle,|1\rangle,\ldots,|n\text{-}1\rangle$ are called *basis states.* Any state of the form $\sum_{i=0}^{n-1} a_i |i\rangle$ is called a *superposition* of $|0\rangle,\ldots,|n\text{-}1\rangle$. The coefficient a_i is called an *amplitude* of $|i\rangle$.

The state of a system can be changed by applying *unitary transformation.* The unitary transformation U is a linear transformation on C^n that maps vectors of unit norm to vectors of unit norm. The *transpose* of a matrix A is denoted with $A_{ij}^T = A_{ji}$. We denote the *tensor product* of two matrices with $A \otimes B$.

The simplest case of quantum measurement is used in our model – the full measurement in the computation basis. Performing this measurement on a state $|\psi\rangle=a_0|0\rangle+\ldots a_{n\text{-}1}|n\text{-}1\rangle$ produces the outcome i with probability $|a_i|^2$. The measurement changes the state of the system to $|i\rangle$ and destroys the original state $|\psi\rangle$.

2.3 The Quantum Query Model

The quantum query model is also known as the quantum black box model. This model is the quantum counterpart of decision trees and is intended for computing Boolean functions. For a detailed description, see the survey by Ambainis [5] and textbooks by Kaye, Laflamme, Mosca [4], and de Wolf [2].

A quantum computation with T queries is a sequence of unitary transformations:

$$U_0 , Q_0 , U_1 , Q_1 , \ldots , U_{T-1} , Q_{T-1} , U_T \cdot$$

U_i's can be arbitrary unitary transformations that do not depend on input bits. Q_i's are query transformations. Computation starts in the initial state $\left|\vec{0}\right\rangle$. Then we apply U_0, $Q_0,\ldots, Q_{T\text{-}1}$, U_T and measure the final state.

We use the *ket* notation [3] to describe state vectors and algorithm structure:

$$\left| final \right\rangle = U_T \cdot Q_{T-1} \cdot \ldots \cdot Q_0 \cdot U_0 \cdot \left|\vec{0}\right\rangle .$$

We use the following definition of a query transformation: if input is a state $|\psi\rangle = \sum_i a_i |i\rangle$, then the output is $|\phi\rangle = \sum_i (-1)^{x_{k_i}} a_i |i\rangle$, where we can arbitrarily choose a variable assignment of x_{k_i} for each basis state $|i\rangle$. It is also allowed to skip variable assignment for any particular basis state, i.e. to set $x_{k_i} = 0$ for a particular $|i\rangle$.

Formally, any transformation must be defined as a unitary matrix. The following is a matrix representation of a quantum black box query.

$$Q = \begin{pmatrix} (-1)^{X_{k_1}} & 0 & \dots & 0 \\ 0 & (-1)^{X_{k_2}} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & (-1)^{X_{k_m}} \end{pmatrix}$$

Each quantum basis state corresponds to the algorithm's output. We assign a value of a function to each output. The probability of obtaining the result $j \in \{0,1\}$ after executing an algorithm on input *X* equals the sum of squared moduli of all amplitudes, which correspond to outputs with value *j*.

Definition 3 [1]. *A quantum query algorithm* ***computes f exactly*** *if the output equals f(x) with a probability* $p = 1$*, for all* $x \in \{0,1\}^N$*. Complexity is equal to the number of queries and is denoted with* $Q_E(f)$*.*

Definition 4 [1]. *A quantum query algorithm* ***computes f with bounded-error*** *if the output equals f(x) with probability* $p > 2/3$*, for all* $x \in \{0,1\}^n$*. Complexity is equal to the number of queries and is denoted with* $Q_p(f)$*.*

Quantum query algorithms can be conveniently represented in diagrams, and we will use this approach in this paper.

3 Quantum Query Algorithms for the Boolean Function *AND*

In this section, we present our results in constructing quantum query algorithms for a set of Boolean functions based on the *AND* Boolean operation. We consider bounded-error algorithms, which output a correct answer with some probability. Regarding computing a two-variable function $AND(x_1, x_2)$, the results were obtained as follows: using a method described in Section 2.2.1 of [14], it is possible to construct a bounded-error quantum algorithm for $AND(x_1, x_2)$ with one query and a probability $p = 2/3$. A better probability of correct answer for a one-query algorithm was obtained in [15] and it is $p = 3/4$. In [16] in a proof for Lemma 1, an algorithm for computing an arbitrary two-variable Boolean function is presented, whose probability

is p=11/14. The authors also claim to be able to prove that probability p=9/10 is optimal.

In this paper, we improve these results and show an algorithm which computes $AND(x_1, x_2)$ with one query and a probability $p = 4/5$. Moreover, we extend an algorithm to compute the *AND* of two functions.

This section is organized as follows: first, we discuss the classical complexity of the two-argument Boolean function $AND(x_1, x_2)$. Then we demonstrate a bounded-error quantum query algorithm that computes $AND(x_1, x_2)$ with a probability $p = 4/5$. Finally, we generalize our approach and present a method for constructing efficient quantum algorithms for computing a *composite function* $\overline{AND}_2[f_1, f_2]$, where f_1 and f_2 are Boolean functions.

Definition 5. *We define* $\overline{\boldsymbol{AND}}_{\boldsymbol{n}}$ ***construction*** *($n \in N$) as a composite Boolean function where arguments are arbitrary Boolean functions f_i and which is defined as*

$$\overline{AND}_n[f_1, f_2, ..., f_n](X) = 1 \quad \Leftrightarrow \quad \sum_{i=1}^{n} f_i(X_i) = n,$$

$X = X_1X_2...X_n$; X_i is input for i^{th} function[3]; f_i's are called base functions.

3.1 Classical Complexity of $AND(x_1, x_2)$

Classical deterministic complexity of the Boolean function $AND_2(x_1, x_2)$ is obviously equal to the number of variables: $D(AND_2) = 2$.

Next we will show that the best probability for a classical randomized decision tree to compute this function with one query is p = 2/3. The general form of the optimal randomized decision tree is shown in Figure 2.

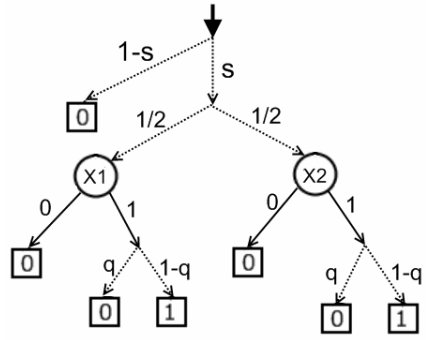


Fig. 2. The general form of the optimal randomized decision tree for computing $AND_2(x_1,x_2)$

[3] Variables may also overlap among inputs for different functions, i.e. for $X_i = (x_{i1},...,x_{in})$ and $X_j = (x_{j1}..x_{jm})$ there may be variables with the same indices.

We denote the probability to see the result $b \in \{0,1\}$ after executing the algorithm on input X with $\Pr("b" \mid X)$. The correct answer probability calculation:

1) $\Pr("0" \mid X = 00) = (1-s) + \frac{1}{2}s + \frac{1}{2}s = 1$,

2) $\Pr("0" \mid X = 01 \ \vee \ X = 10) = (1-s) + \frac{1}{2}s + \frac{1}{2}sq = 1 - \frac{1}{2}(s - sq)$,

3) $\Pr("1" \mid X = 11) = \frac{1}{2}s(1-q) + \frac{1}{2}s(1-q) = s - sq$.

We denote $(s - sq) = z$. Then the total probability of the correct answer is

$$p = min(\Pr("0"), \Pr("1")) = min(1 - \frac{1}{2}z, z).$$

The best probability is obtained when $\Pr("0") = \Pr("1")$.

$$1 - \frac{1}{2}z = z$$

$$z = \frac{2}{3}$$

Corollary 1. The *Boolean function* $AND_2(x_1, x_2)$ *can be computed by a randomized classical decision tree with one query with the maximum probability* p=2/3.

3.2 Quantum Query Algorithms for $AND(x_1, x_2)$

We start with a bounded-error quantum query algorithm for the simplest case of two-variable function $AND(x_1, x_2)$.

Theorem 1. *There exists a quantum query algorithm Q1 that computes the Boolean function* $AND(x_1, x_2)$ *with one quantum query and correct answer probability is* p=4/5: $Q_{4/5}(AND_2) = 1$.

Proof. The algorithm is presented in Fig. 3. Our algorithm uses 3-qubit quantum system. Each horizontal line corresponds to the amplitude of the basis state. Computation starts with the state $|\varphi\rangle = \left(\frac{2}{\sqrt{5}}, 0, 0, 0, \frac{1}{\sqrt{5}}, 0, 0, 0 \right)^T$ (we omit unitary transformation, which converts initial state $|\vec{0}\rangle = (1,0,0,..,0)^T$ into $|\varphi\rangle$). Two large rectangles correspond to the 8×8 unitary matrices U_0 and U_1. The vertical layer of circles specifies the queried variable order for the single query Q_0. Finally, eight small squares at the end of each horizontal line define the assigned function value for each basis state. The main idea is to assign the amplitude value $\alpha = 1/\sqrt{5}$ to the basis state $|100\rangle$ and leave it invariable until the end of the execution.

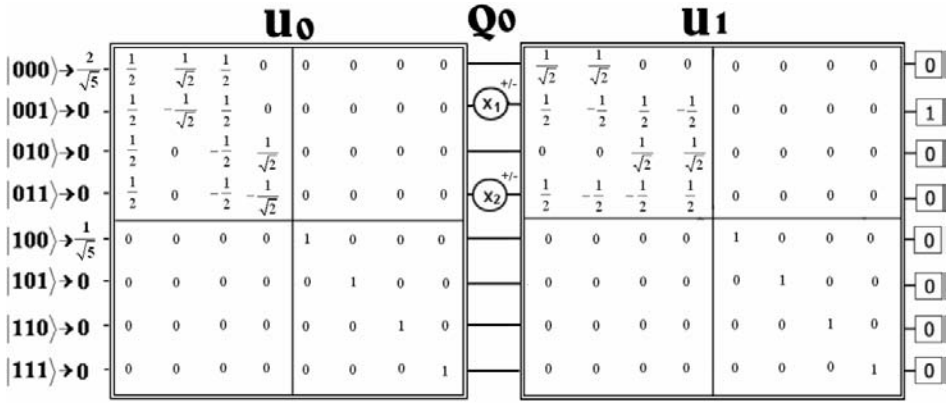


Fig. 3. Bounded-error quantum query algorithm Q1 for computing AND(x_1,x_2)

Quantum state after the first transformation U_0 becomes equal to

$$|\varphi_2\rangle = U_0|\varphi\rangle = U_0 \cdot \left(\frac{2}{\sqrt{5}},\ 0,\ 0,\ 0,\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T =$$

$$= \left(\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$$

Further evolution of the quantum system for each input X is shown in Table 1.

Table 1.

Quantum query algorithm Q1 computation process for AND(x_1,x_2)

X	$\lvert\varphi_3\rangle = Q_0U_0\lvert\varphi\rangle$	$\lvert\varphi_{FINAL}\rangle = U_1Q_0U_0\lvert\varphi\rangle$	p("1")
00	$\left(\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\left(\sqrt{\frac{2}{5}},\ 0,\ \sqrt{\frac{2}{5}},\ 0,\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	0
01	$\left(\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ -\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\left(\sqrt{\frac{2}{5}},\ \frac{1}{\sqrt{5}},\ 0,\ -\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\frac{1}{5}$
10	$\left(\frac{1}{\sqrt{5}},\ -\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\left(0,\ \frac{1}{\sqrt{5}},\ \sqrt{\frac{2}{5}},\ \frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\frac{1}{5}$
11	$\left(\frac{1}{\sqrt{5}},\ -\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ -\frac{1}{\sqrt{5}},\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\left(0,\ \frac{2}{\sqrt{5}},\ 0,\ 0,\ \frac{1}{\sqrt{5}},\ 0,\ 0,\ 0\right)^T$	$\frac{4}{5}$

3.3 Decomposing the $AND(x_1, x_2)$ Algorithm

This section is a transitional point to the generalized method for computing the construction $\overline{AND_2}$. Now we will reveal the internal details of the algorithm *Q1* that allow us to adapt its structure to compute a much wider set of Boolean functions.
The quite chaotic and asymmetric matrix U_0 actually is a product of two other matrices.

$$U_0 = U_0^B \cdot U_0^A = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix U_1, in turn, is a product of the following two matrices.

$$U_1 = U_1^B \cdot U_1^A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Detailed algorithm structure now looks as follows.

$$|\varphi\rangle \to U_0^A \ , \ U_0^B \ , \ Q_0 \ , \ U_1^A \ , \ U_1^B \to [Measure]$$

The final vector is calculated as $|\varphi_{FINAL}\rangle = U_1^B \cdot U_1^A \cdot Q_0 \cdot U_0^B \cdot U_0^A \cdot |\varphi\rangle$.

Now the most important point – the algorithm part represented by transformations U_0^B , Q_0 , U_1^A actually executes two instances of an exact quantum query algorithm for $f(x) = x$ in parallel. Fig. 4 and 5 graphically demonstrate this significant detail.

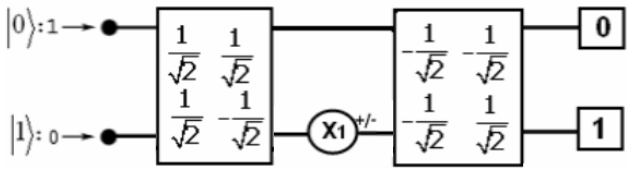


Fig. 4. An exact quantum query algorithm for computing f(x) = x

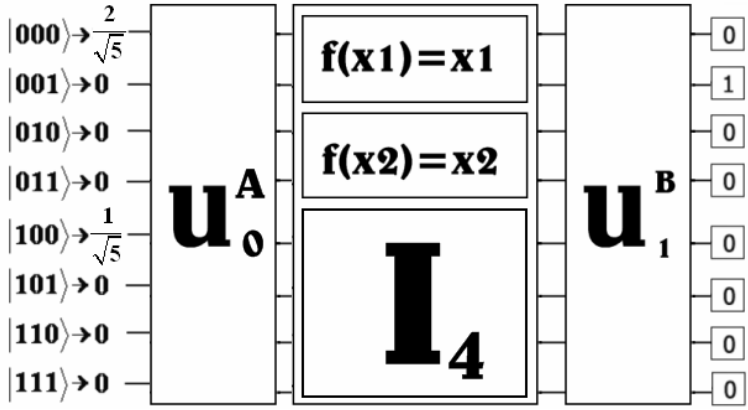


Fig. 5. A quantum algorithm for AND(x_1,x_2), revised

In other words, first of all quantum parallelism is employed to evaluate each variable. Then unitary transformation U_1^B is applied to correlate amplitude distribution in such a way that the resulting quantum algorithm computes $AND(x_1, x_2)$ with acceptable error probability.

In the next section we will generalize this approach to allow to use other Boolean functions as sub-routines.

3.4 A Method for Computing $\overline{AND_2}[f_1, f_2]$

It is possible to replace a sub-algorithm for $f(x) = x$ (in an algorithm construction demonstrated in the previous section) with any other quantum algorithm which satisfies specific properties. We define a class *EQQA+*, and our method is applicable to base algorithms that belong to this class.

Definition 6. *An exact quantum query algorithm belongs to the* ***class EQQA+*** *(positive exact quantum query algorithms) iff there is exactly one accepting basis state, and on any input for its amplitude* $\alpha \in C$ *only two values are possible before the final measurement: either* $\alpha = 0$ *or* $\alpha = 1$.

Theorem 2. *If there exist exact quantum query algorithms A1 and A2 for computing Boolean functions* $f_1(X_1)$ *and* $f_2(X_2)$ *that belong to the class EQQA+, then a composite Boolean function* $\overline{AND_2}[f_1, f_2]$ *can be computed with a probability p = 4/5 using* $\max(Q_E(A1), Q_E(A2))$ *queries to the black box.*

Proof. A general algorithm construction method for computing the Boolean function $\overline{AND_2}[f_1, f_2]$ is presented below. The main idea is to assign the amplitude value $\alpha = 1/\sqrt{5}$ to some fixed basis state and leave it invariable until the end of the execution.

A method for computing $\overline{AND}_2[f_1, f_2]$

Input. Two exact quantum query algorithms A1, A2 $\in EQQA+$ compute Boolean functions $f_1(X_1), f_2(X_2)$. We denote the dimension of Hilbert space utilized by the first algorithm with m_1 (the number of amplitudes), and by the second algorithm with m_2. We denote the positions of accepting outputs of A1 and A2 with acc_1 and acc_2.

Constructing steps

1. If $m_1 = m_2$, then utilize a quantum system with $4m_1$ amplitudes for a new algorithm. First $2m_1$ amplitudes will be used for the parallel execution of A1 and A2. Additional qubit is required to provide separate amplitude for storing the value of $1/\sqrt{5}$.
2. If $m_1 \neq m_2$ (without loss of generality assume that $m_1 > m_2$), then utilize a quantum system with $2m_1$ amplitudes for a new algorithm. First $(m_1 + m_2)$ amplitudes will be used for the parallel execution of A1 and A2. Use the first remaining free amplitude for storing the value of $1/\sqrt{5}$.
3. Combine unitary transformations and queries of A1 and A2 in the following way: $U_i = \begin{pmatrix} U_1 & O_{m_1 \times m_2} & O_{m_1 \times m_1} \\ O_{m_2 \times m_1} & U_2 & O_{m_2 \times m_1} \\ O_{m_1 \times m_1} & O_{m_1 \times m_2} & I_{m_1 - m_2} \end{pmatrix}$, here $O_{m_i \times m_j}$ are $m_i \times m_j$ zero-matrices, $I_{m_1 - m_2}$ is $(m_1 - m_2) \times (m_1 - m_2)$ identity matrix, U_i^1 and U_i^2 are either unitary transformations or query transformations of A1 and A2.
4. Start computation from the state
$$|\varphi\rangle = \left(\underbrace{\sqrt{2/5},\ 0,...,0}_{m_1}, \underbrace{\sqrt{2/5},\ 0,...,0,}_{m_2} \quad \underbrace{1/\sqrt{5},\ 0,..,0}_{remaining\ amplitudes} \right)^T .$$
5. Apply gates U_i. Before the final measurement apply an additional unitary gate.
$$U = (u_{ij}) = \begin{cases} 1, & \text{if } (i = j) \,\&\, (i \neq acc_1) \,\&\, (i \neq (m_1 + acc_2)) \\ 1/\sqrt{2}, & \text{if } (i = j = acc_1) \\ 1/\sqrt{2}, & \text{if } (i = acc_1) \,\&\, (j = (m_1 + acc_2)) \text{ OR } (i = (m_1 + acc_2)) \,\&\, (j = acc_1) \\ -1/\sqrt{2}, & \text{if } (i = j = (m_1 + acc_2)) \\ 0, & \text{otherwise} \end{cases}$$
6. Define as accepting output exactly one basis state $|acc_1\rangle$.

Output. A bounded-error QQA A for computing a function $F(X) = f_1(X_1) \wedge f_2(X_2)$ with a probability $p = 4/5$ and complexity $Q_{4/5}(A) = \max(Q_E(A1), Q_E(A2))$.

The most significant behavior of our method is that overall algorithm complexity does not exceed the greatest complexity of sub-algorithms. Additional queries are not required to compute a composite function. However, error probability is the cost for efficient computing.

A very important aspect is that we used a specific algorithm for the two-variable Boolean function $AND(x_1, x_2)$ as a base for the constructing method. If the correct answer probability for the $AND(x_1, x_2)$ algorithm, which would also use an algorithm for computing $f(X)=X$ as a sub-routine, will be improved to $p > 4/5$, then the probability of a general constructing method and all the further results of this section will be improved as well.

3.5 Class *EQQA*+

In this section, we show that *EQQA*+ class (see Definition 6) is wide enough to be taken into consideration. At the same time, approaches for constructing efficient instances of *EQQA*+ are worth to be examined in a separate paper.

3.5.1 Conversion of Classical Decision Trees into Quantum Query Algorithms

Given an arbitrary classical deterministic decision tree, it is possible to convert it into an exact quantum query algorithm which uses the same number of queries.

A classical query to the black box can be simulated with a quantum query algorithm construction presented in Fig. 6.

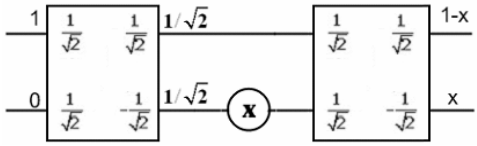


Fig. 6. Quantum query algorithm construction for simulating a classical query

After the second Hadamard gate we obtain $(1,0)^T$ if $x = 0$, or $(0,1)^T$ if $x = 1$.

Then we can continue to query other variables by logically splitting the algorithm flow into two separate threads and so on.

We demonstrate a complete example of converting a classical decision tree for computing $AND(x_1, x_2)$ into an exact quantum query algorithm. Fig. 7 shows a classical decision tree. Figure 8 shows the corresponding exact quantum query algorithm.

We would like to note that, although such conversion is possible, it is not optimal. For instance, it is well known that *XOR* can be computed in a quantum model using two times less queries than required in a classical model.

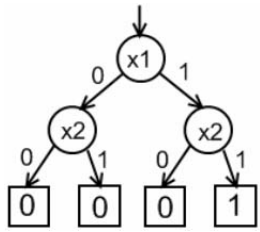


Fig. 7. A classical deterministic decision tree for $AND(x_1, x_2)$

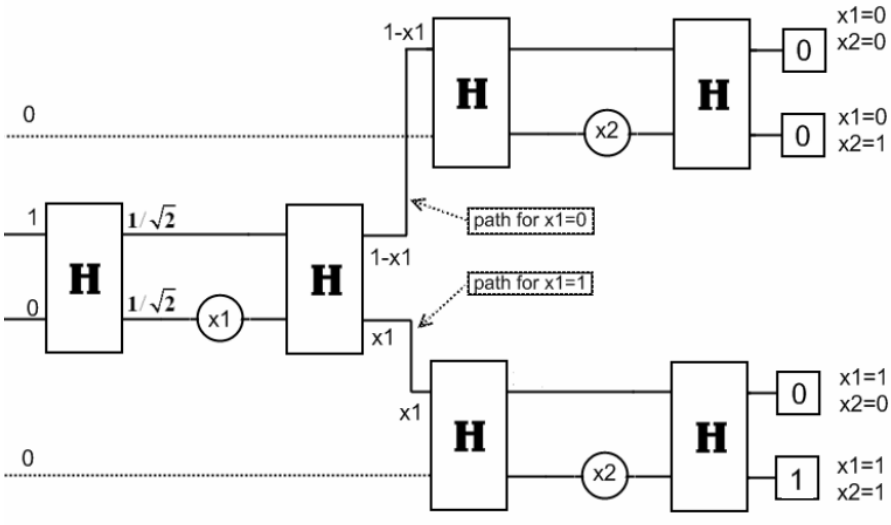


Fig. 8. An exact quantum query algorithm for $AND(x_1, x_2)$

If a deterministic decision tree has exactly one leaf with output value "1", then it obviously will be converted into an algorithm of class *EQQA+*.

It means that we can place into $\overline{AND_2}$ construction any Boolean function that has exactly one accepting input vector. Thus, our method is applicable to an infinite set of base functions.

Theorem 3. *For an infinite set of Boolean functions, quantum query algorithms can be constructed using a method described in Section 3.4. As a result, the following complexity gap can be achieved when computing the same function in quantum and classical deterministic models:* $Q_{4/5}(\overline{AND}_2(f_1, f_2)) = \frac{1}{2} \cdot D(\overline{AND}_2(f_1, f_2))$.

Proof. For any Boolean function f that has exactly one accepting vector, the sensitivity $s(f)$ and, consequently, the deterministic complexity $D(f)$ are equal to the number of variables. Suppose we have two such Boolean functions f_1 and f_2, with the same number of variables N, and wish to compute $\overline{AND_2}(f_1, f_2)$ [4]. Obviously, the

[4] We assume that variables do not overlap this time.

classical deterministic complexity of this function is $D(\overline{AND}_2(f_1, f_2)) = 2N$. For each function we can convert a deterministic algorithm into an exact quantum query algorithm of the class *EQQA+*, which will use the same *N* queries. Finally, we apply the method for constructing an algorithm for $\overline{AND}_2(f_1, f_2)$ which does not require additional queries: $Q_{4/5}(\overline{AND}_2(f_1, f_2)) = N = \frac{1}{2} \cdot D(\overline{AND}_2(f_1, f_2))$.

In the theorem above, classical deterministic and quantum bounded-error query complexity is compared. It would be interesting to compare classical probabilistic and quantum bounded-error complexity correlation for $\overline{AND}_2(f_1, f_2)$. As of today, we do not have such estimation yet.

Theorem 4. *The Boolean function $AND_N(X)$ ($N = 2k$, $k \in N$) can be computed by a bounded-error quantum query algorithm with a probability p = 4/5 using N/2 queries:* $Q_{4/5}(AND_N) = N/2$.

Proof. Boolean function $AND_N(X)$ can be represented as

$$AND_N = \overline{AND}_2(AND_{N/2}, AND_{N/2}).$$

It means that by applying our construction method it is possible to obtain an algorithm with complexity

$$Q_{4/5}(AND_N) = Q_E(AND_{N/2}).$$

The Boolean function $AND_{N/2}$ can be computed by a deterministic algorithm with complexity $D(AND_{N/2}) = N/2$, which has exactly one accepting output. It means that this deterministic algorithm can be converted into *EQQA+* class algorithm which uses the same *N*/2 number of queries.

$$Q_{4/5}(AND_N) = Q_E(AND_{N/2}) = N/2.$$

3.6 An Example of a Larger Separation: $D(f) = 6$ vs. $Q_{4/5}(f)=2$

We would like to demonstrate an example when quantum algorithm complexity can be over two times less than classical deterministic algorithm complexity. It is possible in cases when an exact quantum algorithm for a sub-function is better than the best possible deterministic algorithm for the same function.

An exact quantum query algorithm for $EQUALITY_3(X) = \neg(x_1 \oplus x_2) \wedge \neg(x_2 \oplus x_3)$ has been first presented in [17]. The algorithm is depicted in Fig. 12 and it uses only two quantum queries while classically all three queries are required. The algorithm belongs to the class *EQQA+* and can be used as a sub-algorithm for $\overline{AND}_2$ construction.

To evaluate deterministic complexity of $f = \overline{AND}_2[EQUALITY_3, EQUALITY_3]$, we use function sensitivity on any accepting input: $s(f) = 6 \Rightarrow D(f) = 6$.

A quantum bounded-error algorithm for $f = \overline{AND}_2[EQUALITY_3]$ constructed using our method will require only two queries: $Q_{4/5}(f) = 2$.

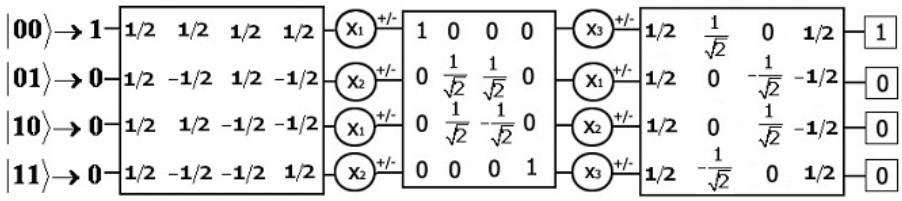


Fig. 9. An exact quantum query algorithm for EQUALITY$_3$

The same approach can be applied to any algorithm of class *EQQA+* that computes an *N*-variable Boolean function.

3.7 Repeated Application of a Method for Computing $\overline{AND}_2[f_1, f_2]$

The useful properties of the algorithm construction method described in Section 3.4 allow to apply this method repeatedly.

Theorem 5. *Let* $F_1 = AND_2[f_{11}, f_{12}]$ *and* $F_2 = AND_2[f_{21}, f_{22}]$ *be composite Boolean functions. Let Q1 and Q2 be bounded-error quantum query algorithms that have been constructed using a method for computing* $AND_2[f_1,f_2]$*, and that compute* F_1 *and* F_2 *with a probability* $p = 4/5$*. Then a bounded-error quantum query algorithm Q can be constructed to compute a composite Boolean function* $F = AND_2[F_1, F_2]$ *with a probability* $p = 16/25$.

Proof. We straightforwardly apply the method for computing $AND_2[f_1,f_2]$ to algorithms $Q1$ and $Q2$ instead of instances of QQA^{+1} class. As a result, the obtained complex algorithm computes $F = AND_2[F_1, F_2]$ with a probability $p = \frac{4}{5} \cdot \frac{4}{5} = \frac{16}{25}$.

As a consequence, we are able to compute a four-variable function $AND(x_1,\ldots,x_4)$ with a single quantum query with a probability p=16/25.

Next iteration produces quantum algorithms that compute functions like

$F = AND_2[AND_2[AND_2[f_1, f_2], AND_2[f_3, f_4]], AND_2[AND_2[f_5, f_6], AND_2[f_7, f_8]]]$

with a probability p=64/125, which is just slightly more than a half.

4 An Exact Quantum Query Algorithm for Verifying Repetition Code

In this section, we consider the second problem: verification of the codeword encoded by the repetition code for error detection. In the first sub-section, we introduce repetition codes and define a Boolean function for their verification. Secondly, we

show that classically, for an N-bit message, values of all N variables must be queried in order to detect an error. Finally, we present an exact quantum algorithm for N-bit codeword verification that uses only $N/2$ queries to the black box.

4.1 Error Detection and Repetition Codes

In this sub-section, we investigate a problem related to information transmission across a communication channel. The bit message is transmitted from a sender to a receiver. During that transfer, information may be corrupted. Because of the noise in a channel or adversary intervention, some bits may disappear, or may be reverted, or even added. Various schemes exist to detect errors during transmission. In any case, a verification step is required after transmission. The received codeword is checked using defined rules and, as a result, a conclusion is made as to whether errors are present.

We consider a repetition error detection scheme known as repetition codes. A repetition code is a (r, N) coding scheme that repeats each N-bit block r times [8].

An example

- Using a (3,1) repetition code, the message m = 101 is encoded as c = 111000111.
- Using a (2,2) repetition code, the message m = 1011 is encoded as c = 10101111.
- Using a (2,3) repetition code, m = 111000 is encoded as c = 111111000000.

Verification procedure for the repetition code is the following – we need to check if in each group of r consecutive blocks of size N all blocks are equal.

We start with verification of the (2,1) repetition code. The verification process can be expressed naturally as computing a Boolean function in a query model. We assume that the codeword to be checked is located in a black box. We define the Boolean function to be computed with the query algorithm as follows.

Definition 7. *The Boolean function* $VERIFY_N(X)$, *where* $N = 2k$, $X = (x_1, x_2, ..., x_{2k})$ *is defined to have a value of "1" iff variables are equal by pairs.*

$$VERIFY_{2k}(X) = \begin{cases} 1, \text{ if } (x_1 = x_2) \wedge (x_3 = x_4) \wedge (x_5 = x_6) \wedge \ldots \wedge (x_{2k-1} = x_{2k}) \\ 0, \text{ otherwise} \end{cases}$$

An example: the Boolean function $VERIFY_4(X)$ has the following accepting inputs:

$$\{0000, 0011, 1100, 1111\}.$$

4.2 Deterministic Complexity of $VERIFY_N$

Fig. 10 demonstrates a classical deterministic decision tree which computes $VERIFY_4(x_1, x_2, x_3, x_4)$. In this figure, circles represent queries, and rectangles represent output.

Theorem 6. $D(VERIFY_N) = N$ **.**

Proof. Check function sensitivity on any accepting input, for instance, on X = 1111..11. Inversion of any bit will invert the function value, because a pair of bits with different values will appear. $s(VERIFY_N) = N \quad \Rightarrow \quad D(VERIFY_N) = N$.

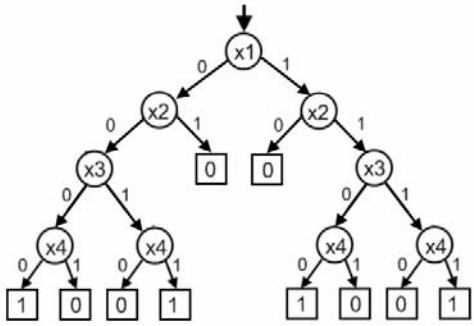


Fig. 10. A classical deterministic decision tree for computing $VERIFY_4(x_1, x_2, x_3, x_4)$

4.3 Computing the Function $VERIFY_N$ in a Quantum Query Model

Our approach to computing the Boolean function $VERIFY_N$ in a quantum query model is based on an exact quantum query algorithm for the *XOR* function.

Theorem 7. *There exists an exact quantum query algorithm that computes the Boolean function* $VERIFY_N(X)$ *using N/2 queries:* $Q_E(VERIFY_N) = N/2$.

Proof. Definition of the $VERIFY_N$ function can be re-formulated as follows.

$$VERIFY_{2k}(X) = \begin{cases} 1, \textit{ if } \neg(x_1 \oplus x_2) \wedge \neg(x_3 \oplus x_4) \wedge \neg(x_5 \oplus x_6) \wedge \ldots \wedge \neg(x_{2k\text{-}1} \oplus x_{2k}) \\ 0 \text{ , } \textit{otherwise} \end{cases}$$

An exact quantum algorithm for computing the Boolean function $f(x_1, x_2) = \neg(x_1 \oplus x_2)$ with one query is presented in Fig. 11. We compose an algorithm for $VERIFY_N$ using an algorithm for $f(x_1, x_2) = \neg(x_1 \oplus x_2)$ as building blocks.

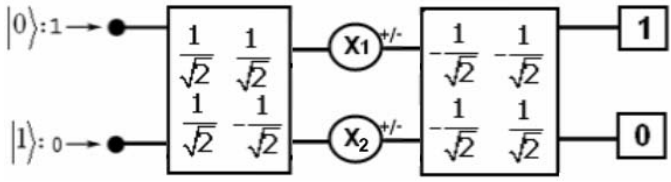


Fig. 11. An exact quantum query algorithm for computing $f(X) = \neg(x_1 \oplus x_2)$

First, we execute an algorithm for $f(x_1, x_2) = \neg(x_1 \oplus x_2)$ for variables x_1 and x_2. To the first output (which has "1" assigned, see Fig. 11), we concatenate the second instance of an algorithm for computing $f(x_1, x_2) = \neg(x_1 \oplus x_2)$. This time we execute it for variables x_3 and x_4. We continue this way until all variables of $VERIFY_N$ are queried. The algorithm has only one accepting output, which is the first output of the last sub-algorithm.

A schematic view of the described approach is depicted in Fig. 12. It is easy to see that the total number of queries is $N/2$. □

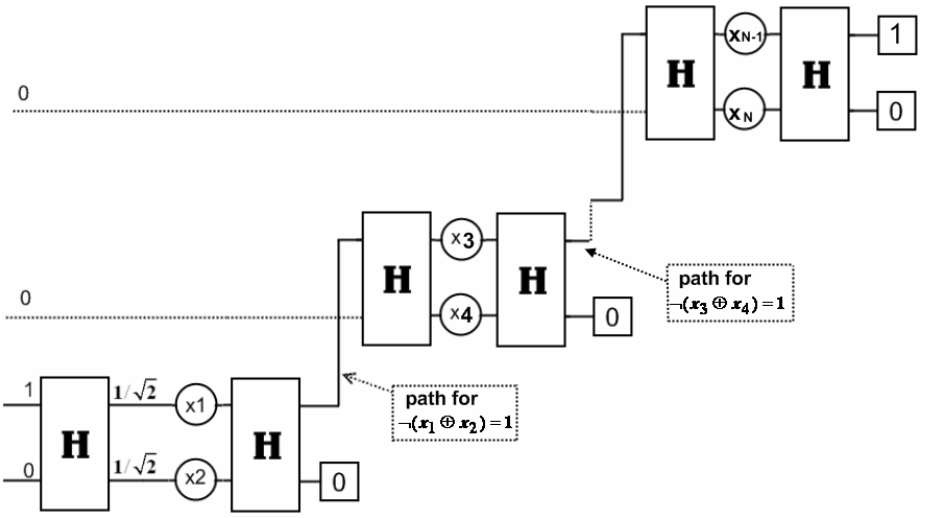


Fig. 12. An algorithm for computing the Boolean function $VERIFY_N$

4.4 Application to a String Equality Problem

The described approach can be adapted for solving such computational problem as testing whether two binary strings are equal. This is a well-known task, which can be used as a sub-routine in various algorithms.

A quantum algorithm for the Boolean function $VERIFY_N$ checks whether variables are equal by pairs, i.e. $(x_1 = x_2) \wedge (x_3 = x_4) \wedge \ldots \wedge (x_{N-1} = x_N)$. On the other hand, we can consider that this algorithm checks whether two binary strings,

$Y = x_1x_3x_5...x_{N-1}$ and $Z = x_2x_4x_6...x_N$, are equal. Therefore, the algorithm can be easily used not only to verify repetition codes, but also for checking equality of binary strings.

4.5 Verification of the (*r*,1) Repetition Code

Now, let us consider the (*r*,1) repetition code, where each bit is repeated *r* times during encoding. Verification procedure for a codeword encoded using such code consists of checking whether in each sequence of *r* bits all bits are equal.

The Boolean function $EQUALITY_r$ is defined as

$$EQUALITY_r(X) = \begin{cases} 1, \textit{ if } (x_1 = x_2) \wedge (x_3 = x_4) \wedge (x_5 = x_6) \wedge ... \wedge (x_{r-1} = x_r) \\ 0 \text{ , } \textit{otherwise} \end{cases}.$$

We define the Boolean function that corresponds to verification procedure as

$$VERIFY^r_{r\cdot N}(X) = \begin{cases} 1, \textit{ if } EQUALITY(x_1,...,x_r) \wedge EQUALITY(x_{r+1},...,x_{2r}) \wedge ... \\ \qquad\qquad ... \wedge EQUALITY(x_{(N-1)r+1},...,x_{Nr}) \\ 0 \text{ , } \textit{otherwise} \end{cases}.$$

Theorem 8. *Deterministic complexity of the Boolean function* $VERIFY^r_{r\cdot N}(X)$ *is equal to the number of variables:* $D(VERIFY^r_{r\cdot N}) = Nr$.

Proof. Again, we use function sensitivity on any accepting input. Inversion of any bit will invert the function value because a pair of bits with different values will appear. $s(VERIFY^r_{r\cdot N}) = Nr \quad \Rightarrow \quad D(VERIFY^r_{r\cdot N}) = Nr$.

Theorem 9. *There exists an exact quantum query algorithm that computes the Boolean function* $VERIFY^r_{r\cdot N}(X)$ *using* $N \cdot r - N$ *queries:*

$$Q_E(VERIFY^r_{r\cdot N}) = N(r-1).$$

Proof. Again, to speed up the verification procedure, we take into account the fact that *XOR* of two bits can be computed with one quantum query. The Boolean function $EQUALITY_r$ can be expressed using operations $\oplus$, $\wedge$ and $\neg$:

$$EQUALITY_r(X) = \neg(x_1 \oplus x_2) \wedge \neg(x_2 \oplus x_3) \wedge \neg(x_3 \oplus x_4) \wedge ... \wedge \neg(x_{r-1} \oplus x_r).$$

This logical formula contains (*r*-1) clauses each consisting of *XOR* of two bits. Using the approach described in the proof of Theorem 7, we can compose an exact quantum query algorithm which computes $EQUALITY_r$ using (*r*-1) quantum queries. This algorithm has one accepting output. Next we use an algorithm for $EQUALITY_r$ as a building block for composing an algorithm to compute $VERIFY^r_{r\cdot N}$. The resulting algorithm uses $N(r-1)$ queries to determine the value of the Boolean function $VERIFY^r_{r\cdot N}$ exactly.

5 Conclusion

In this paper, we presented quantum query algorithms for resolving two specific computational problems.

First, we considered computing the Boolean function *AND* in a bounded-error setting. We presented a quantum query algorithm that computes $AND(x_1, x_2)$ with one query and probability p = 4/5 while the optimal classical randomized algorithm can compute this function with a probability p=2/3 only. Then we extended our approach and formulated a general method for computing $\overline{AND}_2[f_1, f_2]$ composite construction with the same probability p=4/5 and number of queries equal to $\max(Q_E(f_1), Q_E(f_2))$. The suggested approach allows to build quantum algorithms for complex functions based on already known algorithms. A significant behavior is that the overall algorithm complexity does not increase; additional queries are not required to compute a composite function. However, error probability is the cost for efficient computing. We demonstrated that our method is applicable to a large set of Boolean functions. As a result, a complexity gap of $Q_{4/5}(f) = 1/2 \cdot D(f)$ can be achieved for an infinite set of Boolean functions. We also showed that this is not the lower bound for quantum algorithm complexity and examples where $Q_{4/5}(f) < 1/2 \cdot D(f)$ can be constructed as well.

In the second part of this paper, we considered verification of error detection codes. We have represented the verification procedure as an application of a query algorithm to an input codeword contained in a black box. We have represented an exact quantum query algorithm which allows to verify a codeword of length *N* using only *N*/2 queries to the black box. Our algorithm saves exactly half the number of queries comparing to the classical case. This result repeats the largest difference between classical and quantum algorithm complexity for a total Boolean function known today in this model.

We see many possibilities for future research in the area of quantum query algorithm design. The most significant open question still remains: is it possible to increase exact algorithm performance more than two times using quantum tools? Furthermore, there are many computational tasks waiting for an efficient solution in a quantum setting. Regarding the *AND* Boolean function, we would like to improve correct answer probability when computing a two-variable *AND* with one query. Regarding verification of repetition codes, we would like to be able to verify efficiently not only the (2,1) code, but also an arbitrary (*r*,*N*) code. Another fundamental goal is to develop a framework for building efficient *ad-hoc* quantum query algorithms for arbitrary Boolean functions.

Acknowledgments. I would like to thank my supervisor Rusins Freivalds for introducing me to quantum computation and for his constant support and advice.

This research is supported by the European Social Fund project No. 2009/0138/1DP/1.1.2.1.2/09/IPIA/VIAA/004, Nr. ESS2009/77.

References

1. H. Buhrman and R. de Wolf. Complexity Measures and Decision Tree Complexity: A Survey. *Theoretical Computer Science*, v. 288(1), 2002, pp. 21–43.
2. R. de Wolf. *Quantum Computing and Communication Complexity*. University of Amsterdam, 2001.
3. M. Nielsen, I. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.
4. P. Kaye, R. Laflamme, M. Mosca. *An Introduction to Quantum Computing*. Oxford, 2007.
5. A. Ambainis. Quantum query algorithms and lower bounds. (Survey article.) In: *Proceedings of FOTFS III, Trends on Logic*, vol. 23, 2004, pp. 15–32.
6. A. Ambainis and R. de Wolf. Average-case quantum query complexity. *Journal of Physics* A 34, 2001, pp. 6741–6754.
7. A. Ambainis. Polynomial degree vs. quantum query complexity. *Journal of Computer and System Sciences*, 72, 2006, pp. 220–238.
8. T. M. Cover, J. A. Thomas. *Elements of Information Theory.* Wiley-Interscience, 1991, pp. 209–212.
9. P. W. Shor. Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5), 1997, pp. 1484–1509.
10. L. Grover. A fast quantum mechanical algorithm for database search. In: *Proceedings of 28th STOC '96*, 1996, pp. 212–219.
11. A. Ambainis, personal communication, April 2009.
12. D. Deutsch, R. Jozsa. Rapid solutions of problems by quantum computation. *Proceedings of the Royal Society of London*, Vol. A 439, 1992, pp. 553–558.
13. R. Cleve, A. Ekert, C. Macchiavello, M. Mosca. Quantum algorithms revisited. *Proceedings of the Royal Society of London*, Vol. A 454, 1998, pp. 339–354.
14. L. Lāce. Quantum Query Algorithms. Doctoral Thesis. University of Latvia, 2008, pp. 42–43.
15. A. Vasilieva. Quantum Query Algorithms for AND and OR Boolean Functions, Logic and Theory of Algorithms. *Proceedings of the 4th Conference on Computability in Europe*, 2008, pp. 453–462.
16. I. Kerenidis, R. de Wolf. Exponential Lower Bound for 2-Query Locally Decodable Codes via a Quantum Argument. *Journal of Computer and System Sciences*, 2004, pp. 395–420.
17. A. Dubrovska. Quantum Query Algorithms for Certain Functions and General Algorithm Construction Techniques. *Quantum Information and Computation V, Proc. of SPIE*, vol. 6573. SPIE, Bellingham, WA, article 65730F, 2007.

LATVIJAS UNIVERSITĀTES RAKSTI
756. sējums, DATORZINĀTNE UN INFORMĀCIJAS TEHNOLOĢIJAS

Latvijas Universitātes Akadēmiskais apgāds
Baznīcas ielā 5, Rīgā, LV-1010
Tālrunis 67034535

Iespiests SIA «Latgales druka»