ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2007, Vol. 36, No. 3

CHECKING OF CONCEPTUAL MODELS WITH INTEGRITY CONSTRAINTS

Elita Pakalnickiene, Lina Nemuraite

Kaunas University of Technology
Studentu 50-308, LT 51368 Kaunas, Lithuania

Abstract. Due to the raising level of abstraction in information systems development many activities of this process are migrating to its early phases. The same is true for testing – modern CASE tools are undertaking validation of software models. In this paper the methodology for checking of conceptual models is proposed as the step-wise process during which model elements including integrity constraints are progressively checked for their adequacy to values of objects, their relationships and constraints of the corresponding problem domain. The checking process is associated with the particular methodology for development of ordered and precise conceptual models (OPCM), which brings improvements to their quality: conformity to normal forms and ontological foundations, and to the observed reality. The rules for checking of integrity constraints are proposed on the base of taxonomy created in the result of analysis of the most promising methods for conceptual modelling.

Keywords: checking, conceptual model, integrity constraint, ordering, entity, UML, OCL.

1. Introduction[1]

Validation and verification of UML class diagrams constrained by OCL invariants is still an open question of research and the topic of the great interest. It becomes more and more important because of increasing complexity of today software systems and highly competitive software market requirement to develop software systems of high quality at a reasonable cost and time. The software quality can be significantly improved by integration of checking of conceptual models into the development process. We sustain the idea that model and its constraints should be validated and verified before the start of its implementation, because many design mistakes and implementation faults can thus be avoided [1]. Especially for critical systems, identification of errors in modelling can prevent from failures that may result in serious damages. Easy-to-use checking technique and tool support for them are the real necessities. We have a great number of CASE tools that facilitate modelling, documentation and even code generation, but there are rare cases of support for checking conceptual models during design. The advanced CASE tools are already supporting augmenting of conceptual models with constraints that are intended to ensure correctness of data in the implementation in database or program code, but a few of them offer their consistency support and checking.

In this paper, we present ideas for checking of Ordered and Precise Conceptual Model (OPCM), which principles were presented in [2]. The *preciseness* means that conceptual model (represented by UML class diagram for entities − persistent objects whose states are stored in database) is complemented with integrity constraints (OCL invariants), and it is capable to precisely describe states of the problem domain under consideration. The *orderliness* means that the partial order relation exists between entities, which are arranged on `n` levels where entities on the level `i` are dependent on entities of the levels `j<i,i=2,…, n; j=1,…n-1;` entities of level `1` are independent. The OPCM development process consists of three steps: creation of ordered conceptual model; adding integrity constraints; and checking constraints with instances of domain objects. The goal of this paper is focused to the third step, when conceptual models are investigated for ensuring constraints (invariants) that must be satisfied in every state of the system, abstracting from state transition constraints that should be satisfied for application-specific state transitions. Checking of OPCM adhere the same principles as its construction: the sequential procedure starting from the top level. However, elements brought for checking are of the finer granularity as every constraint is checked separately: constraints on model elements (attribute, entity, relationship) and their groups, and, finally, on the overall schema. The gradual

[1] This work is supported by High Technology Development Program Project "Business Rules Solutions for Information Systems Development (VeTIS)" Reg.No. B-07042

checking may be integrated with the creation of model; however, it is not a good practise to elaborate precisely a part of a model without viewing a sketch of its whole.

In [2] paper we have demonstrated that the instituted ordering in the development of conceptual models has many advantages: it results in an ordered, well-formed, easy readable model that conforms to normal forms and ontological foundations of conceptual models. Extending of ordered conceptual model with integrity constraints makes model precise and capable to ensure that domain semantics is rightly expressed [3]. In this paper we propose the process for checking of conceptual model supporting its conformance to the observed reality and assurance that model is syntactically, type- and semantically correct. In overall, the proposed methodology consisting of earlier mentioned three steps brings significantly more quality to conceptual models.

The rest of the paper is organized as follows. In sections 2 and 3 the existing concepts and techniques for verifying, validating and testing of data models with integrity constrains are investigated in relation with the proposed "checking" concept and methodology. In section 4 rules for checking integrity at the model and meta-model level are described. In section 5 the process for checking of ordered and precise conceptual models is presented. Finally, section 6 concludes paper and discusses the future work.

2. What is checking of conceptual model?

Traditionally, the database design process may be defined as made up of sequences of schema transformations between conceptual, logical and physical models [4], where logical design includes transformations for schema simplification, optimization and translation into structures compliant to database management systems. It is worth to mention that several equivalent representations of the same problem domain may exist on every of these levels as several logical models may be obtained from the same conceptual model, and vice versa [5]. Hence, several paths are available for going from the higher to the lower levels (Figure 1). Though it is impossible to discover the best and unique representation of the particular problem domain, all representations on different abstraction levels should adequately describe the same objects of the real world and their semantics. Another important requirement is that these transformations should not lose information between layers. Consequently, the base criterion of the quality of conceptual model should be its capability to represent precisely semantics of the real world, and the purpose of checking is to examine this.

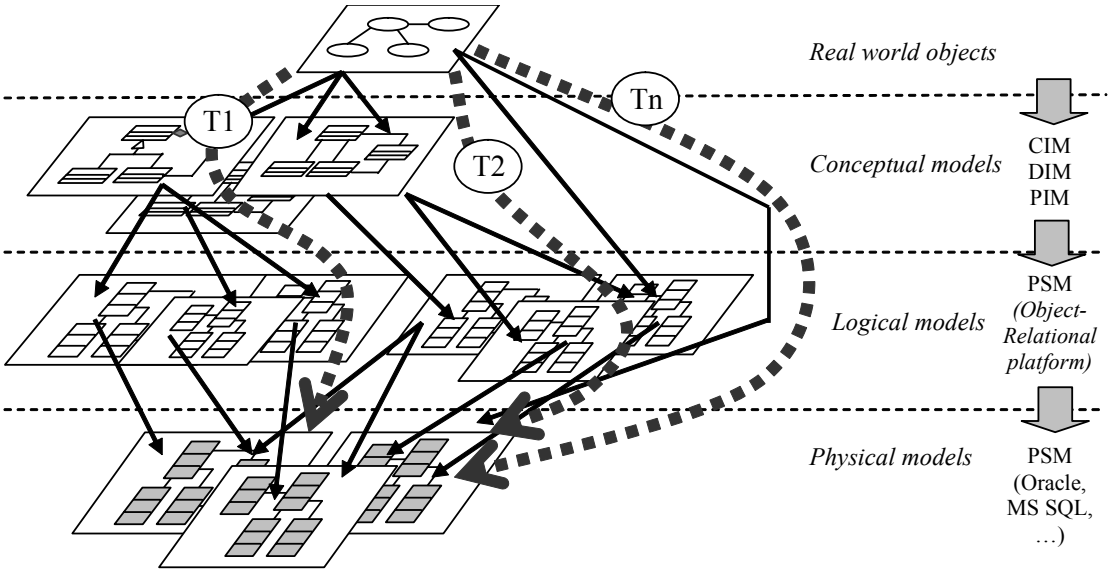

Figure 1. Transformations from the real world to a physical database

For constructing and checking the conceptual model we should have some set of real life instances and known set of actual business rules that we want to implement in our software system. These instances and rules may be obtained from the expert of the problem domain or constructed specially for this purpose. Furthermore, we argue as in [6] that checking of conceptual model should combine validation and verification. In practice, terms "validation" and "verification" are often incorrectly used as interchangeable. Verification and validation definitions used in this paper are adopted from the 1998 AIAA Guide [7], though the similar definitions are met in other sources as well:

- Verification is the process of determining that a model implementation accurately represents the designer's conceptual description of the model and the solution to the model.

- Validation is the process of determining the degree to which the model is an accurate representation of the real world from the perspective of the intended uses of the model.

In short, verification addresses the question "Have we built the model right?", and validation considers the question "Have we built the right model?". These two steps must be taken if we have ascertained whether the model implements the assumptions correctly (model verification) and whether the assumptions which have been made are reasonable with respect to the real system (model validation).

Verification and validation do not replace or include testing, but may be used to determine if testing has been performed correctly [8]. The goal of verification and validation of conceptual models is to assure model correctness and to test conformance of model semantics to the real world from the perspective of the intended uses of the model. Therefore verification and validation cannot prove that a model is correct and accurate for all possible conditions and scenarios; the verification and validation process is completed when the required sufficiency is reached.

In our proposed methodology we define the concept "checking of conceptual model" as a process that determines if the conceptual model is syntactically and type- correct, well formed, and adequate to the observed reality. The syntax check verifies a specification against the grammar of the specification language (in our case, UML and OCL). The type-check makes sure that every OCL expression is described correctly using only types that exist in the conceptual model. Well-formedness check examines if rules of the conceptual data model (defined similarly as UML well-formedness rules) are satisfied. The adequacy check considers if model is one enabling to correctly represent all feasible states of problem domain (i.e. sets of data objects of problem domain), and forbidding to represent unfeasible states.

Different verification and validation techniques address slightly different quality criteria. Structured reviews assure correctness arguments such as completeness, robustness, and optimality of design decisions. Cook and Skinner [9] consider the following six correctness arguments: validity, traceability, optimality, robustness, well formedness, and consistency. In overall, we give priority for the following elements of quality criterion that are covering the most important concerns raised with respect to the quality of conceptual models:

- *Well-formedness* is mainly concerned with a correct use of notations to describe conceptual models and satisfy additional rules (specified as OCL constraints). The part of well-formedness rules is defined in UML meta-model specification, but there are additional rules for precise conceptual data models. For example, it is required for marking the primary identifier for every entity; otherwise an independent identifier is accepted as the primary identifier by default. Such requirements are not raised in "imprecise" conceptual modelling techniques.
- *Robustness* deals with handling of abnormal or exceptional situations. It deals with questions that should focus on detecting omissions and gaps in the model: what are the normal conditions under which the system operates? What are the exceptional and abnormal conditions related to the system operation? Are they handled correctly? Robustness is addressed during creation of OPCM when use cases and their steps are analyzed and all objects required as inputs and outputs are identified. Robustness is checked by applying model to describe collections of correct and incorrect instances of data objects.
- *Adequacy* deals with checking of obtained artefacts to ensure their conformance to data of the observed reality. This kind of checking requires the sufficient amount of model instances enabling to evaluate model capability to correctly represent them. The model instance is a snapshot of the state of the problem domain at the particular moment of time (a particular set of object instances and links between them); it is presented with object diagram. Instances of models can be complete, incomplete and inconsistent. A *complete model instance* is one in which at least one instance of each model element is included. *Incomplete model instance* includes instances of part of model elements, but in either case it must be consistent. A *consistent model instance* represents the consistent state in which all constraints intended and specified by designer are obeyed [10]. It is enough to obtain one complete model instance to evaluate the feasibility of the conceptual model, but it is possible to determine its suitability for various situations only having a sufficient amount of model instances. In general, there is a problem of having this set of required model instances. One possibility is to rely on assumption that the expert of the problem domain can provide it. The other choice is to use empirical methods for creation of model instances that are offered by methods for testing data models or databases. Such methods are described, for example, in [11], implemented by USE tool, Alloy Analyzer, AGENDA etc.

In our methodology, the concept "checking of conceptual model" is used, which includes verification and validation, and is very similar to testing, because we are using "checking cases" – sets of domain instances for validation of models; we are making checking plans where sufficient number of "successful" and "unsuccessful" cases are introduced; we are using "checking units" – model elements, which are combined into larger units; model walkthroughs, inspections and other testing techniques should be used before checking, and so on. Moreover, the goals of checking are the same as of testing: to assure that

conceptual model operates correctly according to its purpose. The same process and instances of problem domain may be used for checking conceptual model and testing its implementation in a database or program code.

However, we are not willing to relate to "testing" because testing is defined as "assurance that software performs correctly" and term "model testing" is not defined anywhere. "Model-based testing", a known term, has a different meaning referring to software testing where test cases are automatically generated on the base of models and usually is used for conformance testing [12].

"Incremental testing" should be mentioned here also as it can lead to confusion in relation with our method. The method presented in [13] is associated with increasing effectiveness of methods, implemented in database for checking constraints at run-time, by checking only part of constraints, which are potential for causing violations of database states. Our method is "incremental" in another sense as it is making checking of part of constraints starting from the top level. However, the purpose of our method is to check the conceptual model in the requirement analysis phase. Also, "checking of conceptual model" should not be confused with "model checking" – state machine-based verification techniques.

3. Related work

Many of different techniques have been developed for model checking, but we would like to focus on approaches that analyze UML class diagrams with OCL invariants and are supported by tools. Currently there are only a few tools specifically designed for analyzing UML models and OCL constraints. Probably, this is mostly due to the lack of a precise standardized formal semantics of UML and OCL. A well-defined semantics is a prerequisite for building tools offering sophisticated analysis features [1].

One of the first tools dedicated to check UML class models together with OCL invariants is USE (UML Specification Environment) tool [14]. The tool takes UML class diagram with OCL expressions and makes them machine-analyzable. The model is verified performing syntax, type and semantic checking. Validation is performed with user-provided test cases by means of object diagrams. The tool enables creating and changing object diagrams and automatically gives responses about their validity against the invariants and pre- and post conditions specified in the model. USE system has snapshot generator that allows manipulation in more flexible way not only by creating and destroying objects, inserting and removing links between objects and setting attributes values of objects.

This approach requires less efforts from designer since models can be directly used as input for analysis, but creating of procedures for automation of snapshot generation requires the special training in ASSL (A Snapshot Sequence Language); manual production of test cases is the time-consuming work that has to be done by domain expert. The technique is quite intuitive and no rules are defined for selecting data structures for verification and validation.

Another approach for automating analysis of UML class diagram is to use formal specifications with precise semantics based on Alloy (a formal object-oriented modelling language founded on first-order logic) [15]. In ref. [16] rules are presented for mapping between UML&OCL and Alloy elements. Alloy models are analyzed with the Alloy Analyzer, which allows automatic generation of all valid snapshots and counterexamples within a given scope, satisfying model constraints. The scope for validation can be gradually increased by the user enlarging the number of elements for each basic type. Besides, this analysis supports checking of assertions by searching a snapshot that refuses the asserted property.

The validation processes in USE and Alloy Analyzer differ in that USE tool offers the evaluation of user-provided snapshots of models while Alloy Analyzer offers the automatic simulation by searching for instances satisfying given constraints. However, Alloy is a light-weight formal language and is not able to reflect all semantics of UML and OCL. So the priority should be given for tools like USE.

In the sources [17], [18] an approach for testing database applications is developed and AGENDA tool-set is implemented to facilitate this approach. AGENDA performs testing at physical database schema level and takes as input the database schema of the database on which the application runs, the application source code, and 'sample-value files' containing suggested values for attributes. The tester can select test heuristics and provide information about expected behaviour of test cases. Differently from USE and Alloy Analyzer, AGENDA performs validation of model implementation populating the database with meaningful data satisfying constraints (currently AGENDA can handle just uniqueness, not-null and referential constraints, and semantic constraints involving simple expressions). Generating inputs to the application and executing the application for those inputs AGENDA performs checking if the resulting database state and application output is valid according to the expected behaviour indicated by the tester.

Integration of informal and formal methods is one of the corner stones of the KeY approach [19], [20]. The KeY tool provides automatic support for creating formal specifications. It has a uniform user interface for modelling, specification, implementation, and verification of software and may be used for the entire development process. Formal specifications can be introduced and verified incrementally. However, the KeY tool is devoted for checking Java implementation and does not have capabilities for validating models.

The most promising methodologies for checking UML models are implemented in OCLE [21] and

MagicDraw tools. Differently from analyzed earlier, they allow users to define rules at model and meta-model level and use them for validating chosen models. In OCLE, OCL specifications are defined in separate files; in MagicDraw – in stereotyped validation packages. These tools will analyze the loaded model and evaluate the corresponding OCL expressions on each model element. Thus, if a rule is defined for a class in the model, it will be evaluated for all its instances and all the instances of any of its child classes. If a rule is defined in the context of meta-class Class, it will be evaluated for each class and association class in the model. If such a rule evaluates to false or if it cannot be evaluated for some reason, the model is considered faulty – either the model contains errors or it is not adequate.

OCLE also helps in debugging models and OCL specifications. Both tools perform syntax and type checking – OCLE during typing OCL expression, and MagicDraw – during validation (only partial syntax checking is done in MagicDraw during writing expressions). Besides the capabilities to create class instances provided by the model browser and snapshot diagrams, OCLE allows to import instances from files generated by other tools. Currently, two state-of-the-art tools are supported: the USE open-source project and the ModelRUN commercial tool. This functionality is very useful for evaluation of business constraints, because it requires concrete objects (instances) to set the context for certain rules.

To reveal the presence of errors in the OPCM and to check its conformance to the observed reality, we will embrace verification and validation capabilities offered by MagicDraw and check models using the representative collection of domain objects. The main difference of our proposal from aforementioned methods is that we institute the order of checking by adhering the same principles as model construction: the sequential procedure starting from model elements of the top level. Validation of model elements against valid states of problem domain described by object diagrams containing instances of validated elements and elements under validation is performed in a gradual way, by the order arising from a problem domain. Object diagrams with validated instances are incrementally complemented with new instances for validation in a natural way understandable for users. In addition, the gradual checking has performance advantages against other model verification and validation techniques. Another important point of the proposed method for checking conceptual models is in that we are giving taxonomy and recommendations what types of integrity constraints and under what circumstances should be analyzed and applied [22], [23].

4. Rules for Checking Conceptual Models

In our previous work [22] the taxonomy of integrity constraints relevant for making semantically meaningful model was proposed on the base of analysis of types of constraints adressed in the most promising conceptual modelling methods (ER, Extended ER (EER), UML, eXexutable UML (xUML), and ORM). We have applied these types of constraints for creation of ordered and precise conceptual models. The capabilities of UML to accurately express all important types of constraints in terms of UML metamodel and its extension mechanisms of stereotypes, tagged values and constraints were presented in the paper [23]. There are alternative options for representation of constraints using UML: natural language or OCL expressions in notes, but stereotypes are useful as patterns not only for discovering constraints, but also for succeeding generation of implementation code as properties and constraints of stereotypes may be mapped to the functionality of database management systems. Hence, there are several possibilities to define integrity constraints in UML:

- Specify constraints on model elements;
- Specify constraints on model elements using constraint patterns;
- Specify constraints on stereotypes;
- Specify constraints on elements of UML meta-model.

The first possibility is the simplest one, but in such case every modeller must directly create constraints on each element requiring for constraints. Specification of constraints using patterns defined for stereotypes of constraints, releases modellers from repetitive efforts for defining typical constraints. Predefined constraint patterns can be instantiated for constrained elements without having deep knowledge about OCL syntax. Specification of constraints on stereotypes is even more effective way because stereotypes may be reused. Applying stereotype for model element the modeller wouldn't have to worry about writing constraints because they will be defined in advance on meta-model elements. However, the possibilities to validate constraints on stereotypes are much more restrictive in current implementations of CASE tools. Only simple constraints may be validated in the meantime. And there are some kinds of constraints that should not be specified using stereotypes, for example, association between entities A and B with multiplicities "0..*" and "1" means that instances of class A always must have values for an attribute referring to class B. Seeking for reusability these constraints should be specified on elements of UML meta-model. By all means, modelling of domain-specific integrity constraints may require for specifying constraints inherent only for that domain. For example, such are constraints on derived values. However, stereotypes and patterns are capable to considerably reducing of these efforts.

Let's consider rules for checking constraints specified directly on model elements, stereotypes and meta-model elements using example in Figure 2. Here the part of OPCM-type model is given with

constraints of different complexity represented by appropriate stereotypes.

Primary identifier constraint. The stereotype `<<P>>` is used for representation of primary identifiers (analogous to primary keys in relational databases) for data objects. An attribute or a group of attributes with stereotype `<<P>>` comprise the primary identifier for unique identification of instances of the class. It requires that the identifying attribute or the group of attributes always should have values and these values should be unique.

The model-level constraint for `Department` primary identifier denoted with stereotype `<<P>>` is simply expressed as OCL invariant:

```
Context Department inv unique:
   self.allInstances()->
         isUnique(d:Department|d.code)
```

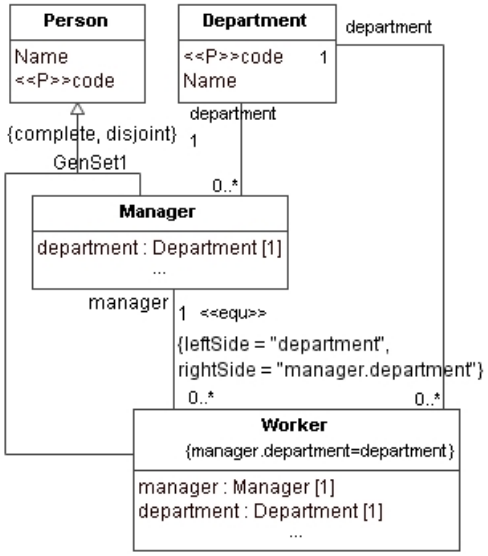

Figure 2. Example of stereotypes for representation of integrity constraints

However, such simple constraints must be specified for every attribute having the stereotype `<<P>>`, while the same constraint specified for stereotype will check the uniqueness of slots of all attributes having the stereotype `<<P>>` (Figure 3):

```
context P inv meta_unique:
   self.classifier.allInstances().slot->
      select(
      s|s.definingFeature.name=self.name)->
      isUnique(s|s.value)
```

Mandatory association constraint. Mandatory constraint on an attribute representing link to the associated object (association end) is used to indicate that this attribute must have a value. In UML mandatory constraint on association is denoted by multiplicities "`1`" or "`1..*`". The model-level mandatory constraint on the property (attribute) representing association end is defined by a simple invariant expression:

```
Context Manager inv mandatory_association:
   not self.department.oclIsUndefined()
```

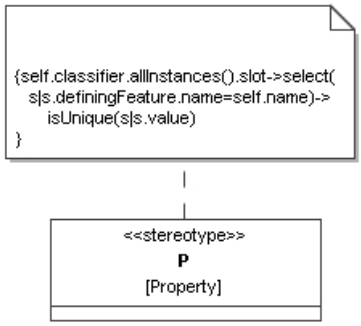

Figure 3. Primary identifier constraint specified on stereotype

The corresponding meta-model level constraint will check all properties representing association ends and having multiplicities "`1`" or "`1..*`":

```
context Property inv meta_mandat_assoc:
 if not self.association.oclIsUndefined()
  and self.association.lowerValue=1 then
  let r:String=self.association.name in
 self.classifier.allInstances().slot->
  select(s|s.definingFeature.name =
    self.name)->forAll
  (p|p.concat('.').concat(r).value=p.value)
    else false endif
```

Fragment of UML meta-model on which this constraint was defined is represented in Figure 4, where `upperValue` and `lowerValue` elements are inherited by `StructuralFeature` element from `MultiplicityElement`:

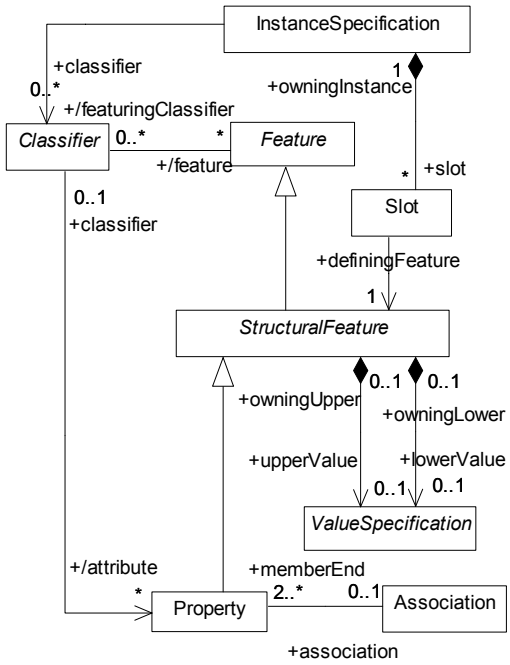

Figure 4. UML meta-model elements on which constraint for mandatory association is specified

Generalization constraints. Generalization in UML may have several generalization sets, where every set means the particular specialization of the same super-type. For example, all animals may have generalization set `G1`, in which they are specialized to flying and cursorial according to their motion, and generalization set `G2`, in which they are specialized to mammals, reptilians, etc., according to their feed. Generalization constraints are defined on generalization sets. `{complete}` means that a set of instances of the super-type in a given generalization set is fully covered by instances of its subtypes. `{disjoint}` constraint means that sets of instances of subtypes in a given generalization set do not overlap. In general, four types of generalization are possible: `complete, disjoint; incomplete, disjoint; complete, overlapping; incomplete, overlapping` (default is `{incomplete, disjoint}`). However, such constraints are not enforced in UML CASE tools. `{incomplete}` and `{overlapping}` generalizations do not require constraints. For ensuring `{complete}` and `{disjoint}` constraints on generalization relationship in Figure 2 having generalization set `GenSet1` with `{complete}` and `{disjoint}` constraints, the model-level checking rules may be specified:

```
context Person inv complete:
not  self.oclAsType(Manager).oclIsUndefined()
or
not self.oclAsType(Worker).oclIsUndefined()
context Person inv disjoint:
if                                          (not
self.oclAsType(Manager).oclIsUndefined())
then
self.oclAsType(Worker).oclIsUndefined() else
not  (self.oclAsType(Worker).oclIsUndefined())
endif
```

As it can be seen, the dependence to a generalization set is not reflected on model-level generalization constraints. Reusable `{complete}` and `{disjoint}` constraints should be defined on meta-model level (Figure 5); it is considerable to define them on UML meta-model element `GeneralizationSet`, because the constraint expression is the simplest in this case.

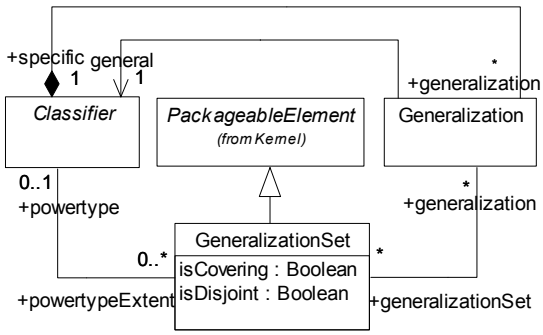

Figure 5. UML meta-model elements on which generalization constraints are defined

```
context GeneralizationSet inv meta_complete:
   if self.isCovering = true then
self.generalization.specific->
   collect(p|p.allInstances())->includesAll
(self.generalization.general.allInstances())
   else false endif
context GeneralizationSet inv meta_disjoint:
if self.isDisjoint = true then
   self.generalization.specific->
collect(p|p.allInstances()).asSet()->size()=
self.generalization.general.allInstances()->
size())else false endif
```

Equal set constraint on path of relationships. Equal set and subset constraints on path of relationships comprising loops are the most complicated constraints and are not considered in most popular conceptual modelling methods [24]. These constraints indicate that not all instances of object type can participate in appropriate relationship but just instances participating in set of constrained relationships. For example, in Figure 2 the equal set constraint denoted by stereotype `<<equ>>` is defined for property of class `Worker` representing association with `Manager` and means that the set of instances selected by traversing a loop in one direction (`Worker.Department`) must be the same as the set of instances selected by traversing the loop in the opposite direction (`Worker.Manager.Department`), according to the rules and policies of the domain. Model-level equality constraint on relationship loop:

```
context Worker inv loop:
   department = manager.department
```

For definition of this constraint on meta-model level, the property of stereotype (tag) should be used. Like a class, a stereotype may have properties, which may be referred to as tag definitions (Figure 6). When a stereotype is applied to the model element, the properties of stereotype are referred to as tags, and the values of these properties are referred to as tagged values. For example, in Figure 2 the aforementioned equal set constraint is marked with stereotype `<<equ>>` having tags "`left_side`" and "`right_side`" of type `String` used for the definition of the loop: `department = manager.department`. For defining the equal set constraints on the meta-model level the following constraint on stereotype `<<equ>>` should be used:

```
context Property inv meta_loop:
   if self.classifier.allInstances()->
      forAll(p|p.concat(left_side).value=
      p.concat(right_side).value)
   then true else false endif
```

In the similar way meta-model-level constraints and/or constraints on stereotypes were created for all integrity constraints from taxonomy [22], [23], comprising the `<<OPCM profile>>` that was proposed

for development and checking of conceptual models. Besides aforementioned constraints, we have considered constraints on values of attributes; disjunctive mandatory, coexistence, exclusion, subset, uniqueness constraints restricting groups of optional attributes or relationships; acyclic, ireflexive, symmetric, intransitive, antisymetric constraints for reflexive relationship, and others. This profile may be applied on the top of existing facilities of UML CASE tools, for example, MagicDraw. MagicDraw tool provides validation profile for validating UML models: stereotypes `<<validationRule>>` for turning constraints into validation rules and supplementing them with error messages; stereotype `<<validationSuite>>` for creation of validation rule sets, and supports automatic validation of sets of constraints against UML models and instances. There are even validation rule suites including a part of UML meta-model constraints, however, these rule suites are still incomplete with regards to UML models, and they are rather unforeseen for checking of conceptual models.

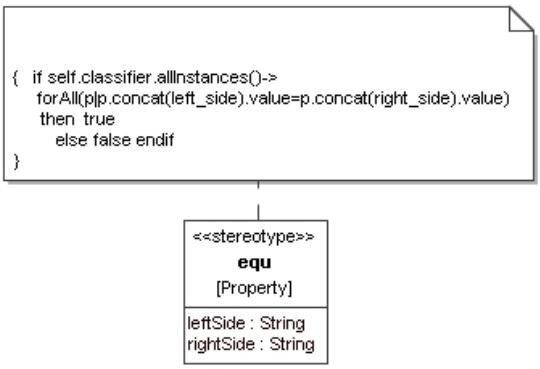

Figure 6. Stereotype with tag definitions and constraint

Using MagicDraw tool, OPCM rule sets are applied to conceptual models complemented with object diagrams presented by the domain expert. The domain expert should create object diagrams that represent adequate and not-adequate states of problem domain. MagicDraw tool, using these rule sets, will check objects diagrams helping to find out if there are reasonable object diagrams that do not satisfy checking rules, or if there are undesirable system states that satisfy them. The first case may indicate that constraints are too strong or the model is not adequate in general. The second case may indicate possibility that constraints may be too weak. Therefore, in both cases the model must be revised, e.g., by relaxing or making more restrictive constraints. The revised model having refined object types, relationships and constraints is checked again until an appropriate assumption is reached about the correctness of the model with respect to the analyzed states of the problem domain represented in object diagrams.

5. Checking Process

The methodology for creating conceptual models presented in [2] institutes the sequence of analysis steps and results in the ordered conceptual schema, where classes are arranged to levels starting from independent (top level) ones. Classes on the next levels are dependent on the classes of the higher levels. For example, the fragment of conceptual model presented in Figure 2 is ordered because partial order relationship exists between its elements, which are arranged on three levels: `Department` and `Person` are independent entities, `Manager` on the second level is dependent on these independent entities, and the third level contains the `Worker` entity dependent on `Manager` from the second level and `Department` from the first level.

The algorithm for checking of ordered and precise conceptual models (OPCM) is presented in Figure 7. It is based on the iterative process similar to its construction. The procedure starts from checking the object diagram composed of objects of the top level elements. In the next steps objects are gradually added for entities of lower levels dependent on already analyzed ones.

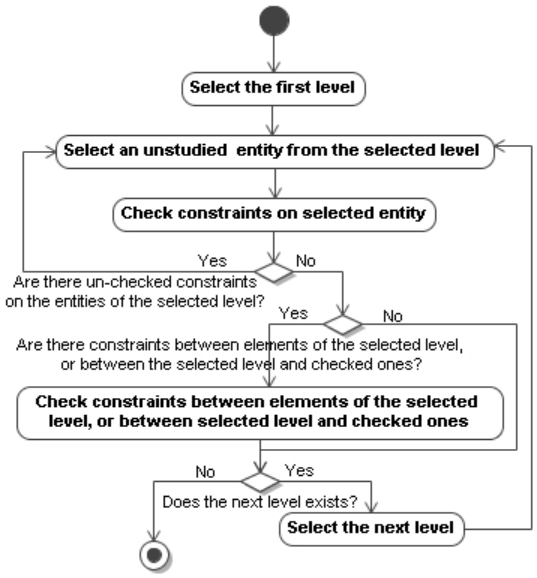

Figure 7. The algorithm for checking of ordered and precise conceptual models

Firstly, constraints on independent elements and associations between them are checked. For this purpose an object diagram must be created with desirable and undesirable objects of independent entities. Checking of OPCM, presented in Figure 2, will start from the object diagram shown in Figure 8 having valid and invalid objects of entity `Department`. The result of MagicDraw validation of entity `Department` is presented in Figure 9 where invalid objects violating primary identifier constraint on department code or mandatory constraint on department name are presented in the red rectangular shapes with thickened

borders. Continuing procedure the accepted elements are included as valid and not analyzable in the next steps of checking process. In our case checked `Department` object with department code "1" satisfies all requirements of problem domain and will be used in further checking. In the same way the entity `Person` is checked and two different objects satisfying all constraints on `Person` are picked for further checking of generalization subtypes.

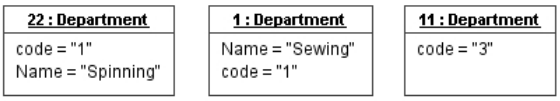

Figure 8. The object diagram for checking the entity "`Department`"

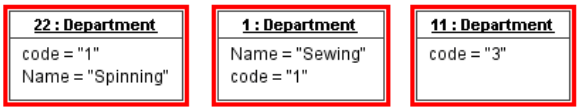

Figure 9. The resulting object diagram after the first step

The next step of the algorithm validates objects of entities dependent upon checked entities and associations between entities of currently analyzed level and the previous one. These steps are repeated till any unstudied level or entity exists. In our example, during the second step of the algorithm we have to check if model accepts only objects of `Manager` entity (subtyped from `Person`) related with the existing department and not violating primary identifier constraint on the attribute `code`. Finally, the `Worker` entity is checked using object diagram presented in Figure 10 where valid object of `Person` entity sub-typed into `Worker` object is rejected because it is associated with the `Manager` object having link with a different department than `Worker` object (and violates the equality constraint). The generalization constraints also are checked; they are satisfied in the current object diagram because the manager and the worker are different persons.

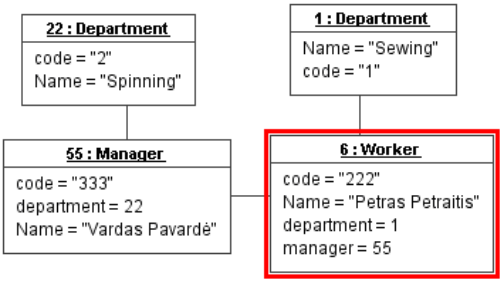

Figure 10. Object diagram for checking Worker entity

In our little example at least three instances (of `Department`, `Manager` and `Worker`) are needed for checking; they comprise one complete instance of the overall model (Figure 11). One complete instance of a model obeying all integrity constraints can confirm the feasibility of that model (the absence of syntax and type errors, well-formedness and adequacy to problem domain). However, much more instances are required to check the relevance of models for all intended scenarios of their usage.

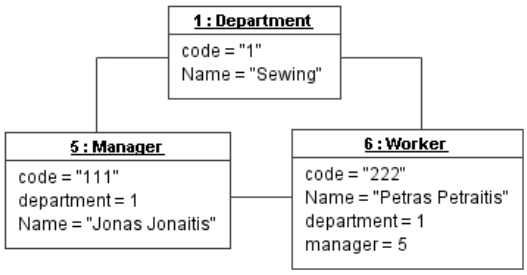

Figure 11. The complete instance tested for the adequacy to the problem domain

In [2] it is shown that the same structure of complete instances of conceptual model would be obtained applying methods of Formal Concept Analysis (FCA) [25], [26].

6. Conclusion and Further Work

Many activities of information systems development including checking of their correctness and adequacy to the problem domain may be made on the conceptual level. Currently it seems that checking of conceptual models is the much more difficult problem than their development and cannot be done without automation. The real possibilities to check conceptual models already exist in advanced UML&OCL CASE tools like OCLE and MagicDraw, but comprehensive methods offering practical checking rules and checking processes are not elaborated. In the current paper we have discussed the rules required for checking of conceptual data models and investigated the possibilities to define them using elements of such models, stereotypes and meta-model elements.

We have proposed reusable checking rules for taxonomical integrity constraints inherent for precise conceptual models, and tried the incremental checking process executing it on the top of facilities of existing CASE tool. The orderliness of conceptual models, as well as processes of their development and checking, have demonstrated their additional quality and performance advantages and may be easily applied by modellers without training in formal methods. However, the problem of generating instances for investigation of the sufficient set of cases of intended usages of models still remains under the responsibility of domain experts. This task may be only partially automated – as well as creation of conceptual models.

Our future work is addressed to looking for possibilities to using the OPCM profile, containing introduced stereotypes and integrity constraints, in generation of the full-fledged database schema when the consistent conceptual model created and checked

using OPCM profile should be automatically transformed into statements of structured query language and/or program code containing data structures and constraints, therefore reducing time and avoiding additional errors in model implementation.

References

[1] **M. Richters, M. Gogolla.** Validating UML Models and OCL Constraints. *Proc.* 3*rd International Conference on the Unified Modeling Language* (*UML*), 2000.

[2] **E. Pakalnickiene, L. Nemuraite, B. Paradauskas.** The Orderliness and Precision in Conceptual Modelling. 11*th Panhellenic Conference on Informatics* (*PCI* 2007), 18-20 *May* 2007, *Patras, Greece*, ISBN: 978-960-89784-1-6, 341-350.

[3] **T. Halpin.** UML Data Models from an ORM Perspective: Part 1 – 10. *Journal of Conceptual Modeling*, 2003, *Available at: http://www.orm.net/uml_orm.html.*

[4] **J.L. Hainaut, C. Tonneau, M. Joris, M. Chandelon.** Transformation-based Database Reverse Engineering. *Proceedings of the* 12*th international conference on entity relationship approach held in Texas, USA, December*, 1993, *edited by R. Elmasri, V. Kouramajian, and B. Thalheim*, 364–375.

[5] **M. Gogolla, M. Richters**. Expressing UML class diagrams properties with OCL. *Lecture Notes in Computer Science*, *Vol*. 2263, 2002, 85-114.

[6] **J. Dietrich, A. Paschke.** On the Test-Driven Development and Validation of Business Rules. 4*th International Conference on Information Systems Technology and its Applications* (*ISTA* 2005), *New Zealand, May* 2005, 31-48.

[7] American Institute of Aeronautics and Astronautics, Guide for the Verification and Validation of Computational Fluid Dynamics Simulations. *AIAA-G*-077-1998, *Reston, VA*, 1998.

[8] **I. Traore, D.B. Aredo.** Enhancing Structured Review with Model-Based Verification. *Software Engineering, IEEE Transactions on Vol.*30, *Issue* 11, *Nov*. 2004, 736 – 753.

[9] **A.D. Cook, M.J. Skinner.** How to Perform Credible Verification, Validation, and Accreditation for Modeling and Simulation. *The Journal of Defense Software Engineering, May* 2005, *http://www.stsc.hill.af.mil/crosstalk/2005/05/0505Cook.html*.

[10] **H. Garcia-Molina, J.D. Ullman, J. Widom.** Database System Implementation. *Prentice-Hall: Englewood Cliffs, NJ*, 2000.

[11] **I. Burnstein.** Practical Software testing. *Springer*, 2002.

[12] **I.K. El-Far, J.A. Whittaker.** Model-based Software Testing. *Encyclopedia on Software Engineering* (*edited by J.J. Marciniak*), *Wiley*, 2001.

[13] **J. Cabot, E. Teniente.** Incremental Evaluation of OCL Constraints. *Proc.* 18*th Int. Conf. on Advanced Information Systems Engineering* (*CAiSE*'06), *LNCS*, *Vol*. 4001, 2006, 81-95.

[14] **P. Ziemann, M. Gogolla.** Validating OCL Specifications with the USE Tool – An Example Based on the BART Case Study. *Electronic Notes in Theoretical Computer Science, Vol.*80, *August* 2003, 157-169.

[15] **C. Wallace.** Using Alloy in process modeling. *Information and software technology*, ISSN 0950-5849, *Vol*.45, *No*.15, 2003, 1031-1043

[16] **T. Massoni, R. Gheyi, P. Borba.** A UML Class Diagram Analyzer. *Proceedings of the Third International Workshop on Critical Systems Development with UML*, 2004.

[17] **D. Chays, Y. Deng.** Demonstration of AGENDA tool set for testing relational database applications. *Proceedings of the* 25*th International Conference on Software Engineering, May* 03-10, 2003, *Portland, Oregon*.

[18] **D. Chays, Y. Deng, G.P. Frankl, S. Dan, I.F. Vokolos, J.E. Weyuker.** An AGENDA for testing relational database applications. *Journal of Software Testing, Verification and Reliability, Mar*. 2004.

[19] **W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski.** The KeY Tool. Software and Systems Modeling. *Springer*, 2005.

[20] **B. Beckert, U. Keller, H.P. Schmitt.** Translating the Object Constraint Language into the Java Modelling Language. *Proceedings of the* 2004 *ACM symposium on Applied computing*, ISBN:1-58113-812-1, *Cyprus*, 2004, 1531 – 1535.

[21] Object Constraint Language Environment (OCLE). *http://lci.cs.ubbcluj.ro/ocle/index.htm*.

[22] **E. Miliauskaite, L. Nemuraite L.** Taxonomy of integrity constraints in conceptual models. *P.Isaias et all*. (*Eds*.): *Proceedings of the IADIS Virtual Multi Conference On Computer Science and Information Systems, IADIS Press*, ISBN: 972-8939-00-0, 2005, 247-254.

[23] **E. Miliauskaite, L. Nemuraite.** Representation of integrity constraints in conceptual models. *Information technology and control, Kauno technologijos universitetas*, ISSN 1392-124X. *Vol*. 34-4, 2005, 355-365.

[24] **L. Starr.** Executable UML. How to build class models. *Prentice Hall, Upper Saddle River*, 2002.

[25] **H.A. Priestley.** Ordered Sets and Complete Lattices. *International Summer School and Workshop: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Lecture Notes in Computer Science*, *Vol*. 2297/2002, *Springer Berlin / Heidelberg Publishers*, 2002, 21-78.

[26] **R. Godin, P. Valtchev.** Formal concept analysis-based class hierarchy design in object-oriented software development. *Lecture Notes in Computer Science, Springer Berlin/Heidelberg Publishers*, ISBN 978-3-540-27891-7, 2005, 304-323.

Received July 2007.