Annales UMCS Informatica AI 4 (2006) 72-85

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Specification of dependency areas in UML designs

Anna Derezińska*

Institute of Computer Science, Warsaw University of Technology,
Nowowiejska 15/19, 00-665 Warsaw, Poland

Abstract

A concept of dependency areas can help in tracing an impact of artifacts of a project (requirements, elements of the UML design, extracts of the code) and assist in their evolution. The dependency area of an element of a UML design is a part of the design that is highly influenced by the given initial element. Dependency areas are identified using sets of propagation rules and strategies. Selection strategies control application of many, possible rules. Bounding strategies limit the number of elements assigned to the areas. This paper is devoted to the specification of the rules and strategies. They are specified using an extended UML meta-model and expressions in the Object Constraint Language (OCL).

1. Introduction

The Unified Modeling Language (UML) [1] is intended as a language to be used for model-driven development. A model gives the ability to consistently show different views of the same design. The views are expressed by the relevant diagrams of the UML. As the UML is semantically rich, we can widely describe the system that will be developed, but we cannot guarantee the consistency of the designed model.

There are three natures of checking the rightness of user diagrams, namely completeness, consistency and correctness [2-4]. The completeness states whether the user requirements are completely reflected on diagrams of a model. The consistency is responsible for checking whether the diagrams are coherently designed with only one requirement. It encompasses so-called vertical consistency that confirms the relation between corresponding models of different abstraction levels (inter-model consistency). Finally, the correctness decides whether the user diagrams are compliant to the syntactic and semantic rules of the UML standard. Within a model it is also named horizontal or intra-model consistency. In some cases it is impossible to conclude whether diagrams are inconsistent or incomplete.

* *E-mail address*: A.Derezinska@ii.pw.edu.pl

The industrial projects are often incomplete [5]. They comprise mostly class diagrams. Their state diagrams are assigned only for selected classes. Such projects can have redundant or contradictory information. Traceability information can be partially missing or not up to dated. The development tools for UML models can check the syntax of the diagrams, but the consistency verification is still unsatisfactory. A systematic modeling strategy, e.g. RUP (*Rational Unified Process*) [6], can recommend an appropriate structure of a model. However, its successful application depends on the effort and experience of a developer of the model.

Addressing the problems of comprehension and evolution of UML designs, a framework for the identification of dependency areas was introduced [7]. The dependency area of an initial element is a part of the model consisting of elements that are highly related to, and influenced by, this element (or a set of initial elements).

The identification of the dependency areas is not a final goal, but one step of a model-driven development. Dependency between parts of design artifacts is an issue of very high practical relevance. The approach of dependency areas can be beneficially applied in the following domains:

- Support understanding of a design.
- Ensure that requirements are correctly implemented.
- Determine the effects of parts of the specification.
- Monitor the changes in the design.
- Assist in testing – assuring that an adequate part of the design was covered by the tests.

Based on the dependency areas, defined in a design by a prototype tool, we selected the appropriate parts of the code generated from the design [7]. The automatically selected code was used in the further evaluation of the program dependability.

Dependency areas are identified using the sets of propagation rules and strategies. Selection strategies control application of many, possible rules. Bounding strategies limit the number of elements assigned to the areas.

The approach deals with imperfect designs, incomplete or inconsistent, in a tool-supportable way. Conflicting and ambiguous situations are resolved according to given strategies. The usage of the rules can be restricted by the strategies. In the strategies, the consistency with the UML specification is preferred. In some ambiguous cases the strategies are based on heuristics that try to reveal the intensions of a developer.

The focus of this paper is on the precise definition of the rules and strategies using a UML meta-model combined with a textual specification written in the Object Constraint Language (OCL) [8]. The OCL, a generic query language is a part of the current standard of modeling, the UML [1].

The paper is structured as follows. Section 2 presents an overlook of a concept of dependency areas and its framework. The specification of propagation rules and strategies is explained in the next sections. Section 5 gives basic information about experiments and a tool support. Remarks about related work and conclusions finish the paper.

2. Dependency areas

A concept of dependency areas is a successor of the idea of a dependency region [7]. It was later revised and specified as an extension of the UML meta-model [9]. The term area replaced the former term region, because the notion of a region is used in UML 2.0 for structuralizing of state machines [1].

An outlook of the idea of dependency areas is given below. The framework for the identification of dependency areas is briefly reminded in Section 2.2.

2.1. A concept of dependency areas

The ***potential dependency area*** of an initial element a – **PDA(*a*)** – is the set of all possible model elements accessible from x through the relations available in the design: traceability, dependency, containment, etc.

The ***dependency area*** of an initial element a – **DA(*a*)** – is a certain subset of PDA(a). It is obtained by reducing the potential dependency area according to given strategies.

One can also define the dependency area of a given set of initial elements.

The identification of the dependency area of a given initial element is based on the three general strategies:

1. Propagation of relations.
2. Selection of relations.
3. Bounding potential dependency areas.

The first strategy explores different relations between elements of a design. It decides about assigning elements to potential dependency areas. The propagation strategy can be expressed by a set of propagation rules. A propagation rule states that if an element x of a design belongs to a current PDA(a) and other pre-conditions of the rule are satisfied then the selected elements of this design are also included in the PDA(a).

Different traceability relations can be identified for an element of a design. Therefore, the element can satisfy the pre-conditions of different propagation rules. Moreover, some of the relations can be incomplete, ambiguous, or even contradictory. According to a selection strategy (2.), we choose which of the propagation rules should be applied in the defined cases.

Many elements of a design can be assigned to a certain PDA(a). In an extreme case, even all elements of the design can belong to the same PDA(a). Such an area will not be useful for the applications of dependency areas,

mentioned in the previous section. Therefore, a potential dependency area can be reduced according to a given bounding strategy (3.).

The idea of dependency areas was adopted for UML designs [7,9]. It was especially aimed at imperfect designs. A given UML design could be incomplete or the information from different diagrams is inconsistent. Therefore, the selection and bounding strategies are based on some heuristics that presume the intentions of a design developer. Using these strategies the identification of dependency areas can be performed automatically. The identification of the dependency area of an element a – **DA(*a*)** (i.e. its quality) can be verified by the following conditions:

- all elements of DA(a) are directly or indirectly dependent on the element a assuming the defined types of dependency,
- no element of DA(a) is superfluous – i.e. not substantially dependent on the element a.

2.2. Meta-model of dependency areas

A framework for identification of dependency areas in UML designs was defined using a conceptual UML model [9]. The model can be interpreted on a meta-model level – as an extension of the UML meta-model [1]. The meta-model for identification of dependency areas will be called briefly DA meta-model. A small part of the DA meta-model is shown in Fig. 1.

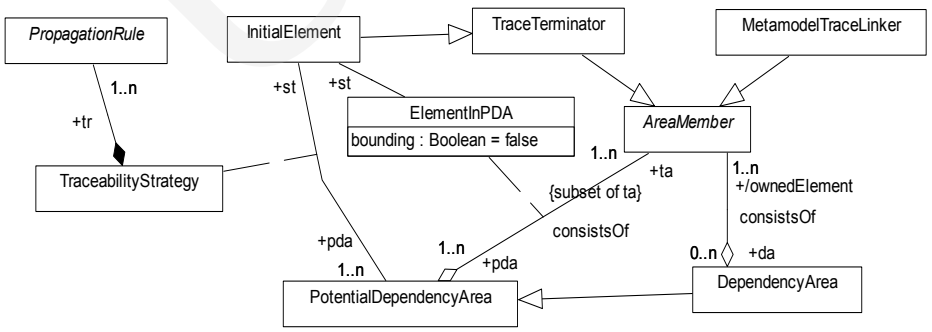

Fig. 1. A part of the DA meta-model – dependency areas

A potential dependency area is defined for a given element of the type *InitialElement*. The identification of potential dependency areas assigned to the initial element is controlled by *TraceabilityStrategies.* A traceability strategy consists of a set of *PropagationRules*. The class *DependencyArea* is a specialization of the class *PotentialDependencyArea.*

Any area consists of elements of the type *AreaMember*. The abstract type *AreaMember* combines two concrete types of elements *TraceTerminator* and *MetamodelTraceLinker*. These elements are derived from various elements of the UML meta-model.

3. Propagation rules

The strategies of the framework are defined using sets of rules. The rules are specified with the notation of the Object Constraint Language (OCL) [8], which is a part of the UML standard. In this paper, we focus on the way of specification of the rules and strategies defined for the identification of dependency areas. The approach will be illustrated by simple examples. The complete mapping of the DA meta-model to the whole UML specification will be not given, for the brevity reasons.

A propagation strategy consists of propagation rules. A propagation rule can determine that different elements of the design are included to a current *PDA(a)*. These elements can be for example: properties of a class that belongs to the *PDA(a)*, a use case included in another use case from the *PDA(a),* a whole diagram assigned to an element belonging to the *PDA(a)*. The above examples correspond to the relations within a UML model. Other propagation rules are defined also using different relations between the models (trace, access, usage etc.).

This section explains how the propagation rules are specified. First, a simple example of an OCL specification of a rule is shown. Next, the same example is defined more precisely in the context of the DA meta-model.

3.1. Specification of propagation rules in the OCL

Propagation rules are defined as pre and post-conditions of an operation. The conditions are written in the OCL. The following example explains an idea of a propagation rule. The reference to the UML meta-model is illustrated by an appropriate extract from the meta-model diagram [1].

The example considers a relation of the generalization between two elements of a model. Let us assume that an element *x* belongs to a certain PDA*(a)*. The element *x* is connected with the generalization *y* to its base element *z*. The assignment of the specific element (here *x*) to PDA*(a)* implies the assignment of a more general element (here z) to the same area. The considered element *x* can be for example a use case, a class or a signal. The rule R1 is based on the part of the UML meta-model shown in Fig. 2. The generalization *z* has its corresponding class in the meta-model, similarly to the elements *x* and y. Therefore, the generalization *z* is also included in the same potential dependency area PDA*(a)*.

The pre- and post-conditions are interpreted using the combined domain of elements from the conceptual model and the UML meta-model. The OCL expression *tr.st.pda.ta* denotes the set of elements comprised in a potential dependency area of an initial element. It is specified by the names of the roles from the classes of the meta-model (Fig. 1). The navigation from the *PropagationRule* approaches the *TraceabilityStrategy* (the role *tr),* then the *InitialElement* (the role *st*) and its *PotentialDependencyArea* (*pda*), and finally the elements of this area (*ta*). The standard OCL operation *A->includes(a)* denotes that an element *a* is contained in the set defined by the expression *A*.

```
Rule  R1     pre: tr.st.pda.ta -> includes(x) and
             x.oclIsKindOf(Classifier) and
             x.generalization -> includes(z) and
             z.general = y
       post:tr.st.pda.ta -> includes(y) and
             tr.st.pda.ta -> includes(z)
```

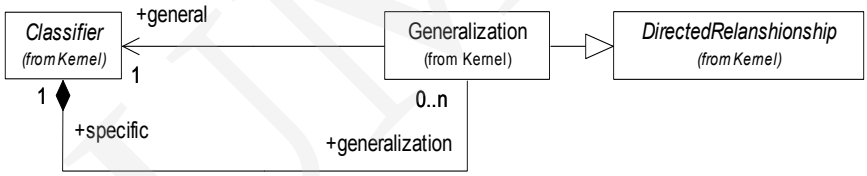

Fig. 2. A part of the UML meta-model – a relation of the generalization

3.2. Propagation rules in the DA meta-model

An abstract class of the type *PropagationRule* is the base class of all propagation rules in the DA meta-model (Fig. 3). Each propagation rule has its identifier and a priority. Pre- and post-conditions of a rule are specified in the context of its operation *ruleBody()*. The operation takes as a parameter an element of the type *TraceTerminator*. Any *TraceTerminator* is either the initial element of a potential dependency area or an element already included in the current area. A *TraceTerminator* represents an element from the considered UML design (the application model). Therefore, it is derived from a certain element of the UML meta-model.

In the DA meta-model a group of special classes represents a set of propagation rules. The basic idea of these classes will be explained with an example. It considers the same relation of the generalization between classifiers (Fig. 2). A part of the DA meta-model specifying this relation is shown in Fig. 3. A class of the type *ClassifierPropagationRule* is responsible for the realization

of all rules devoted to classifiers. This fact is defined in the pre-condition of the operation of this class.

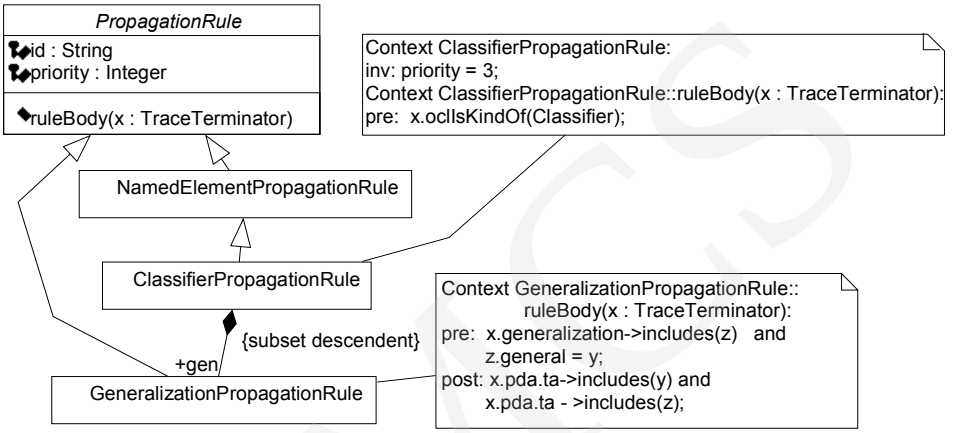

Fig. 3. A part of the DA meta-model – a generalization of classifiers

Consequently, analyzing a generalization of a classifier, the pre- and post-conditions from the both detailed rules, the *ClassifierPropagationRule* and the *GeneralizationPropagationRule,* should be satisfied. The logical conjunction of the conditions defined in these rules is equivalent to the conditions presented in the Rule R1 in the previous section.

Many other propagation rules are specified in the similar way in the DA meta-model.

4. Specification of the strategies

Specifying the selection and bounding strategies, we use a notion of the propagation rules written in the OCL. The strategies are based on heuristics that presume the intentions of the developer. The application of the rules is controlled by their position in the DA meta-model, by their priorities and additional attributes of the strategies.

4.1. Selection strategy

Many propagation rules can be applied to a given element (*TraceTerminator*) belonging to a potential dependency area. The application of propagation rules is controlled by selection strategies. These strategies are responsible also for the resolving of ambiguous situations. A selection strategy defines a hierarchy of applicable propagation rules. The strategy takes into account the priorities of the propagation rules and additional selection priorities, if necessary to decide among the rules of the same priority.

In the DA meta-model the propagation rules create a hierarchy. This hierarchy corresponds to the inheritance hierarchy of the base types of *TraceTerminators*, i.e. the inheritance hierarchy of elements in the UML meta-model. For example, a class of the type *ClassifierPropagationRule* is derived from the *NamedElementPropagationRule* (Fig. 3), because the class *Classifier* derives from the class *NamedElement* in the UML specification. Other propagation rules that are derived from the *ClassifierPropagationRule* exist also in the DA meta-model, e.g. a rule of a use case or a rule of a class.

A given *TraceTerminator* can satisfy pre-conditions of many propagation rules in its inheritance path of the hierarchy. In the first step, the most specific propagation rule that corresponds to a given *TraceTerminator* type will be taken into account. After considering this rule and the rules aggregated to it, its base rule, defined for a more general *TraceTerminator*, can be evaluated. For example, a *TraceTerminator* is at first recognized as a use case and later as a classifier.

Furthermore, a certain propagation rule PR corresponding to a given type of a *TraceTerminator* can have more detailed propagation rules that are aggregated to PR. It can have more than one such rule. These rules have priorities of the same or different values. At first, the rules with the highest priority are applied. Among the rules with the same priority all of them are used or only a subset can be chosen. The application of these rules is controlled by appropriate attributes of the selection strategy.

As an example, the UML dependency relation between two elements of a design is defined by the rule R2 given below. It is one of the rules defined for a *TraceTerminator* that is the *NamedElement* from the UML meta-model. A similar rule is also defined for another *TraceTerminator* – the class. According to the selection strategy and its hierarchy, this similar rule will be applied first than the rule R2.

```
Rule R2     pre: tr.st.pda.ta -> includes(x) and
            x.oclIsKindOf(NamedElement) and
            x.suplierDependency-> includes(z) and
            z.stereotype.name->includes('derive')
            and z.client->includes(y)
      post:tr.st.pda.ta -> includes(y) and
            tr.st.pda.ta -> includes(z)
```

In the rule R2, the dependency relation is specified with the stereotype «derive». Another rule can describe the same dependency relation but without any stereotype. This rule has a lower priority than the rule R2.

In some designs, no direct relations between elements can be found and/or no meaningful stereotypes are present. In this case, the rules based on an identity or a similarity of the names of appropriate elements can be used. For example,

a name of a use case is equal to a name of a collaboration. Such rules have lower priorities than the rules discussed above.

4.2. Bounding strategy

Bounding strategies limit a number of elements included in a potential dependency area. According to given heuristics, they try to select only those elements which are directly pointed out or intently used by a developer. Instead of identifying potential dependency areas and further excluding the assigned elements, a bounding strategy can be specified as a specific selection strategy. Bounding strategies use the similar notion of propagation rules. Within a bounding strategy the application of the rules can be restricted. The potential dependency area PDA(*a*) finally obtained according to the selection and bounding strategies is the resulting dependency area DA(*a*) identified for the initial element *a*.

Bounding strategies are based on heuristics. One of them assumes that the information included in the behavioral diagrams of a design is superior to that from the structural diagrams. A simple example will illustrate this approach and its specification.

There are two diagrams concerning a class *C* in a UML design. The class *C* is defined with its operations *O1* and *O2* in a class diagram - structural diagram (Fig. 4a). The second one is a behavioral diagram assigned to the class *C* (Fig. 4b).

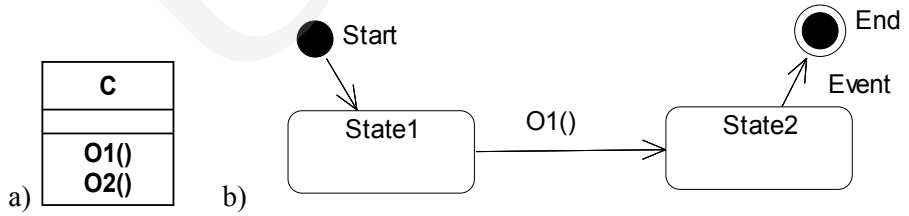

Fig. 4. a) A structural diagram – class C b) A behavioral diagram – state machine of the class C

The propagation rule R3 given below takes into account the information derived from a class diagram. According to the rule R3, if the class *C* belongs to a certain PDA(*a*) then both its operations *O1* and *O2* would be assigned to the same PDA(*a*).

```
Rule R3     pre: tr.st.pda.ta -> includes(x) and
            x.oclIsKindOf(Class) and
            x.ownedOperation->includes(y)
      post:tr.st.pda.ta -> includes(y)
```

Other rules deal with the information defined in the behavioral diagrams. The class *C* is a classifier that has its behavior specification expressed by a state

machine. One of transitions of this state machine is triggered by the operation *O1*. If the class *C* belongs to the PDA(*a*) a sequence of propagation rules can be performed. The last rule of this sequence is the rule R4 shown below. Appling this rule, the operation *O1* will be assigned to the PDA(*a*).

```
Rule R4     pre: tr.st.pda.ta -> includes(x) and
                 x.oclIsKindOf(transition) and
                 x.trigger.operation->includes(y)
       post:tr.st.pda.ta -> includes(y)
```

Without using any bounding strategy, both operations *O1* and *O2* are contained in the considered PDA(*a*). However, the operation *O2* is present only in the structural diagram. It is not used in any behavioral diagram. In this design, only the operation *O1* expresses an activity of the class C. Therefore, after using the bounding strategy, the operation *O1* will be included in the PDA(*a*) and the operation *O2* will be not included. The developer of the design can be warned that the operation *O2* could be missing in behavioral diagrams due to incompleteness or inconsistency of the design.

The bounding strategy is realized by the modification of the propagation rules and the restricted order of their application. As an example, the modified parts of the rules R3 and R4 are shown below. The inclusion of an element to a given area is controlled by a *bounding* attribute defined for the bounding strategy in the DA meta-model (Fig. 1). The bounding strategy assures also that the rule R4 will be applied before the rule R3.

```
Rule R3     (The modified pre-condition)
       pre:tr.st.pda.ta -> includes(x) and
                 x.oclIsKindOf(Class) and
                 x.ownedOperation->includes(y) and
                 x.ownedOperation-> forAll (p|
                 p.st.pda.elementInPDA.bounding = false)
Rule R4     (The modified post-condition)
       post:tr.st.pda.ta -> exist (p| p = y and
                 p.st.pda.elementInPDA.bounding)
```

In another example, a design contains only one diagram dealing with the class *C*, namely a class diagram (Fig. 4a). In this case no information about the behavior can be used. The rule R4 is not applied and no bounding will be provided. According to the modified rule R3 both operations *O1* and *O2* are included in the PDA(*a*).

Summing up, the following heuristics is specified by the rules discussed above. If a class has no information about its behavior all its operations, given in structural diagrams, are assigned to a PDA(*a*). Otherwise, the PDA(*a*) comprises only these operations that were specified in the behavioral diagrams. The

bounding strategy deals in the similar way with other members of a class and with other classifiers.

5. Experimental evaluation of dependency areas

The proposed strategies were empirically verified on several small UML projects. Dependency areas identified automatically by a prototype tool [7] were compared with the corresponding parts of the model selected manually by developers of the projects. In some projects the dependency areas generated by the tool contained fewer elements than those anticipated by the developers. In an extreme case, the dependency area of a given use case was even the empty set. The analysis of such projects revealed the lack of traceability notions in them and/or a disordered structure of these projects. In general a dependency area can help in pointing out an ill-quality of a project.

In other projects, evaluated in these experiments, the elements automatically assigned to the dependency areas were consistent with the expectations of the developer. These areas contained all elements responsible for the realization of initial use cases. At the same time, no superfluous elements were assigned to these areas.

The analysis of the adequacy of dependency areas identified by the tool and the evaluation of the quality of a project could be supported by quantitative measures. Identification of a dependency area that has no elements or a relatively small number of them can be a warning of an incomplete design. On the other hand, a dependency area containing many, or even all, elements can indicate a project that is coupled too strongly. The analysis of a project can be also supported by other metrics, based on the number and kinds of the rules used during the identification of dependency areas. The detailed definition of all these metrics and their interpretation is beyond the scope of this paper.

The experiments, mentioned above, were performed using the prototype tool for **I**dentification of **D**ependency **A**rea – so-called IDA. Only the elements belonging to class diagrams were taken into account during identification of dependency areas in a UML model. Next, these dependency areas were used for selection of the corresponding fragments of the C++ code generated from the model. IDA traced the changes of requirements managed by IBM-Rational RequisitePro [10]. The UML models were analyzed in cooperation with the UML designer IBM-Rational Rose [10]. IDA recognized use cases assigned to the changed requirements and identified their dependency areas. All elements belonging to the dependency areas were marked with additional stereotypes, modifying the given model. A dependency area of any initial element (here a use case) was denoted with its own stereotype. In this way one element could have been assigned to many different dependency areas, eg. DA*(a)* and DA*(b)* for $a \neq b$. Dependency areas of the sets of initial elements were also identified, if required.

After analyzing the results of the experiments, we started developing an improved version of the tool. It is based on the given meta-model and the specification of pre- and post-conditions in the OCL. A new version of IDA identifies dependency areas using information from all types of UML diagrams. A dependency area can contain any element of the design. The tool supports UML 2.0.

6. Related work

Traceability from software artifacts (requirements, models) through to the program code supports understanding the code, and how the former implement the latter [11-13]. Many traceability approaches address a granularity level of diagrams, not interfering into the UML details. In [13], the elements of a UML model are integrated with textual specifications of requirements, but the approach does not explore dependencies within the model. In [14] uncertain relations between requirements and elements of a UML model are resolved by a statistical evaluation of the developers' decisions.

Our work relates also to the field of consistency of UML models. The consistency problems in UML designs were extensively studied in many papers [2-5,15,16]. Nevertheless, many existing approaches, especially formally-based ones, require complete traceability to guide consistency checking. Egyed [16] maintains the consistency of UML class diagrams during refinement and takes into account partial lack of traceability information. Hypothesizing that at least one of the potentially many choices ought to be consistent, the maximum flow algorithm finds the unique correct interpretation and inconsistent traces are removed.

OCL is primarily used for two purposes, for a precise description of UML models and for the specification of the UML with the relation to its meta-model [1,8]. It was successfully applied in different UML-based specifications [2-4,15,17]. There are some tools that offer verification of OCL constraints or animation of UML/OCL specifications [4,17]. However, the widespread adoption of the OCL by industry is still limited due to the insufficient support and integration with other CASE tools.

7. Conclusions

This paper presents the specification of dependency areas identified in the UML designs. The approach is based on the strategies and propagation rules. The rules are precisely defined using pre- and post-conditions denoted in the Object Constraint Language (OCL) [8]. The OCL expressions are interpreted in a meta-model (DA meta-model). It extends the UML meta-model [1] with a conceptual model of dependency areas. The rules and strategies take into account real-world designs, possibly incomplete or inconsistent.

The concept of dependency areas provides a basis for controlling changes within a design, as well as an impact of requirements on the design and the generated code. It can support automation of project management and maintenance.

Currently, the work is focused on developing a new version of IDA, a tool that can support experiments with more comprehensive projects. Furthermore, an integration of the tool with other CASE-tools is needed.

Our future work is dedicated also to the construction of certain metrics. The metrics are based on the size of dependency areas and kinds of the rules used for the identification of the dependency areas. These metrics help in assessing a quality of dependency areas identified automatically by the tool and a quality of the current stage of the design. The new tool should support the calculation of the metrics.

References

[1] *Object Management Group, UML 2.0 Superstructure Specification*, formal/05-07-04, www.uml.org.

[2] Ha L-K., Kang B-W., *Meta-Validation of UML Structural Diagrams and Behavioral Diagrams with Consistency Rules*, Proc. of IEEE Pacific Rim Conf on Communications, Computers and Signal Processing, PACRIM, (2) (2003) 679.

[3] Kuzniarz L. et al. (eds.), *2nd Intern. Workshop on Consistency Problems in UML-based Software Development, <<UML>> 2003*, San Francisco, (2003).

[4] Kuzniarz L. et al. (eds.), *3rd Intern. Workshop on Consistency Problems in UML-based Software Development, <<UML>> 2004*, Lisbon, (2004).

[5] Lange C.F.J., Chaudron M.R.V., *An Empirical Assessment of Completeness in UML Designs*, Proceedings of "EASE International Conference on Empirical Assessment in Software Engineering", Edinburgh, Scotland, (2004).

[6] Krutchen P., *The Rational Unified Process an Introduction*, Addison-Wesley, (2000).

[7] Derezińska A., *Reasoning about Traceability in Imperfect UML Projects*, Foundations of Computing and Decision Sciences, (29)1-2 (2004) 43.

[8] Warmer J., Kleppe A., *The Object Constraint Language: Getting Your Models ready for MDA*, Addison Wesley (2003).

[9] Derezińska A., Bluemke I., *A Framework for Identification of Dependency Areas in UML Designs*, Proc. of IASTED Inter. Conf. on Software Engineering and Application (SEA'05), Phoenix, USA, (2005) 177.

[10] Rational Rose, Rational Requisite Pro, www-306.ibm.com/software/rational/

[11] Alves-Foss J., Conte de Leon D., Oman P., *Experiments in the Use of XML to Enhance Traceability Between Object-Oriented Design Specifications and Source Code*, Proc. of the 35th Hawaii Int.. Conf. on System Sciences, HICSS-35'02 (2002).

[12] Egyed A., *A Scenario-Driven Approach to Trace Dependency Analysis*, IEEE Trans. on Software Engineering, (29)2 (2003) 116.

[13] Letelier P., *A Framework for Requirements Traceability in UML-based Projects*, Proc. of 1st Int. Workshop on Traceability in Emerging Forms of Soft. Eng. by IEEE Conf. on Automated Software Engineering (ASE 02), Sept. 28, Edinburg, (2002).

[14] Spanoudakis G. et al., *Rule-based Generation of Requirements Traceability Relations*, J. Systems and Software, (72)2 (2004) 105.

[15] Briand L. C., Labiche Y., O'Sullivan L., *Impact Analysis and Change Management of UML Models*, Proc. of International Conference on Software Maintenance, Amsterdam, The Netherlands, IEEE CS Press (2003) 256.

[16] Egyed A., *Consistent Adaptation and Evolution of Class Diagrams during Refinement*, Proceedings of the 7th International Conference on Fundamental Approaches to Software Engineering (FASE), Barcelona, Spain, March (2004) 37.
[17] Correa A.L. Werner C.M.L., *Precise Specification and Validation of Transactional Business Software*, Proc. of the 12th IEEE Inter. Requirements Eng. Conf. (RE'04) (2004).