Enforcement of Conceptual Schema Quality Issues in Current Integrated Development Environments

David Aguilera, Cristina Gómez, and Antoni Olivé

BarcelonaTech – UPC
Department of Service and Information System Engineering
C/ Jordi Girona, 1-3, 08034, Barcelona, Catalunya, Spain
{daguilera,cristina,olive}@essi.upc.edu

Abstract. We believe that one of the most effective ways of increasing the quality of conceptual schemas in practice is by using an Integrated Development Environment (IDE) that enforces all relevant quality criteria. With this view, in this paper we analyze the support provided by current IDEs in the enforcement of quality criteria and we compare it with the one that could be provided given the current state of the art. We show that there is a large room for improvement. We introduce the idea of a unified catalog that would include all known quality criteria. We present an initial version of this catalog. We then evaluate the effectiveness of the additional support that could be provided by the current IDEs if they enforced all the quality criteria defined in the catalog. We focus on conceptual schemas written in UML/OCL, although our approach could be applied to other languages.

Keywords: conceptual modeling, quality, method engineering, UML, IDE.

1 Introduction

Conceptual schemas play a critical role in information systems (IS) development. In order to perform this role effectively, conceptual schemas should not have defects, and they should satisfy the quality criteria required by the methods used in their development [1–5]. In the literature, there are many proposals of quality criteria of conceptual schemas [6]. Most of them are relevant to all conceptual schemas, and others are relevant only to conceptual schemas developed in a particular language, method, organization or project.

A conceptual schema should satisfy all relevant quality criteria. This implies that during its development (or once it is finished) it should be checked that those quality criteria are satisfied, and that the appropriate actions are taken for those that are not. In principle, this checking can be done manually, but the time and effort it requires make it difficult to do it in practice. We believe that one of the most effective ways of increasing the quality of conceptual schemas is by using an Integrated Development Environment (IDE) that automatically enforces all

C. Salinesi, M.C. Norrie, and O. Pastor (Eds.): CAiSE 2013, LNCS 7908, pp. 626–640, 2013.

relevant quality criteria. Unfortunately, it is a fact that nowadays only a few quality properties have been integrated into the IDEs used by professionals and students, and thus enforced in the conceptual schemas they develop. Ideally, a conceptual modeling method should adopt all relevant quality criteria, and the method should be supported by an IDE that enforces those criteria, thus ensuring that the developed conceptual schemas have the quality level required by the method [1, 7].

In [8] we proposed a method for defining conceptual schema quality properties, based on the notion of conceptual schema quality issue. In essence, an issue is a condition, which may be an integrity constraint a schema must satisfy to be syntactically correct, a necessary condition for a schema to be satisfiable, a best practice defined as a condition that must be satisfied, and so on. We believe that having quality properties uniformly defined as quality issues eases their integration into IDEs and, therefore, their enforcement in conceptual schemas.

This paper aims at increasing the support given to conceptual modelers by current IDEs in the enforcement of the quality of conceptual schemas. The approach we have taken for achieving this objective is (1) analyzing the support provided by the current IDEs, (2) determining the support that an IDE could provide given the current state of the art—which would be formalized using our method—, (3) comparing the support currently provided with the one that could be provided, and (4) evaluating the effectiveness of the additional support in increasing conceptual schema quality.

We focus on conceptual schemas written in UML/OCL, although our approach could be applied to other languages. There are many IDEs based on UML/OCL[1], but we lack an assessment of the support given by them in the enforcement of conceptual schema quality. The first contribution of our work is a comprehensive review of the quality properties supported by the most widely used IDEs.

In our method [8], quality issues are defined by the engineers of conceptual modeling methods. In principle, all quality properties proposed in the literature could be defined in terms of our method, and they could therefore be included in a unified catalog. The set of issues included in the catalog, but not supported by a particular IDE, defines the ideal additional support that could be provided by that IDE in the enforcement of conceptual schema quality. The second contribution of our work is an initial version of that catalog, which has been made globally accessible.

We have evaluated the effectiveness of the additional support that IDEs would provide if they included all quality issues of the catalog. The last contribution of this paper is, as expected, the conclusion that including more quality issues in an IDE increases the quality of the developed conceptual schemas.

The structure of the paper is as follows. Section 2 briefly reviews the concept of conceptual schema quality issue that we presented in [8] and introduces a new classification of quality issues into five categories. Section 3 analyzes the support given by current UML-based IDEs in the enforcement of quality

[1] A comprehensive list can be found at
`http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools`

properties. Section 4 presents the catalog of quality issues we have developed so far and discusses how this catalog could be included in current IDEs. The catalog includes all quality issues enforced by current IDEs and many others. In Sect. 5 we evaluate the benefit that would be gained if current IDEs included all quality issues defined in the catalog. Section 6 summarizes the conclusions and points out future work.

2 Conceptual Schema Quality Issues

In this section, we review the concept of conceptual schema quality issue that we presented in [8]. We give first an informal definition and a classification, and then we summarize its formalization. Table 1 shows several examples (see [8] for more).

2.1 Informal Definition

According to the dictionary, a quality issue is "an important [quality] topic or problem for debate or discussion"[2]. In essence, an issue is a condition.

We distinguish between two kinds of issues: problem and checking. A *problem* issue is a condition over the schema that should not be satisfied. The condition can be automatically checked. The fact that the condition is satisfied is the issue. Once raised, the issue should be solved, which can only be done by changing the schema itself.

Table 1. Examples of quality issues

Source/Kind	Problem Issue	Checking Issue
Syntactic	I_1 = There is a cycle in a generalization hierarchy	I_7 = A constraint expressed in natural language evaluates to a *Boolean* value [9, p. 57]
Syntactic+	I_2 = An attribute has no type I_3 = A property has no stereotype [7]	I_8 = A derivation rule gives consistent number of values
Basic property	I_4 = A property derived by union does not have specific properties	I_9 = An n-ary association defines all non-graphical cardinality constraints that are relevant
Naming guideline	I_5 = An attribute does not start with a lowercase letter	I_{10} = The name of the entity type is semantically meaningful
Best practice	I_6 = The type of an attribute is an entity type	I_{11} = The aggregation kind of a property is correct [7]

In Tab. 1, an example of problem issue is I_1 = "There is a cycle in a generalization hierarchy". An issue is (automatically) raised per each cycle detected by an IDE in a schema, and it should be solved by changing the schema itself. Most issues are of this kind.

A *checking* issue is a condition over the schema that can be automatically evaluated. The fact that the condition is satisfied is the issue, and it requires the

[2] Oxford dictionaries (`http://oxforddictionaries.com`)

conceptual modeler to manually check something. If the checking points out a problem that was unnoticed, the modeler has to perform an action to solve it.

In Tab. 1, an example of checking issue is I_9 = "An n-ary association defines all non-graphical cardinality constraints that are relevant". The rationale of this issue is that n-ary associations may be subjected to several cardinality constraints, but UML allows the graphical definition of only a few of them [10]. The others must be defined by means of invariants. This checking issue *automatically* determines whether an association is n-ary or not and, if it is, it requires the conceptual modeler to *manually check* that the required invariants are effectively defined in the schema. If these invariants are missing, the conceptual modeler has to *manually define* them and mark the issue as checked.

For analysis and presentation purposes, in this paper we also use a new orthogonal classification of issues according to their source. We distinguish the following five categories (see examples in Tab. 1):

Syntactic: An integrity constraint defined in the UML metamodel. An example is the above mentioned problem issue of I_1 = "There is a cycle in a generalization hierarchy".

Syntactic+: A syntactic integrity constraint applicable when UML is used as a conceptual modeling language or one defined in a UML profile. One example is the problem issue I_2 = "An attribute has no type". In UML it is not mandatory that attributes have a type, but it is so in conceptual modeling.

Basic property: A fundamental property that conceptual schemas should have to be semantically correct, relevant and complete [3]. An example is the above mentioned checking issue of I_9 = "An n-ary association defines all non-graphical cardinality constraints that are relevant", which is required for completeness.

Naming guideline: A guideline recommended by some authors to be used in naming conceptual schema elements. For example, the guideline that recommends attributes to start with a lowercase letter [9, p. 54] corresponds to the problem issue I_5 = "An attribute does not start with a lowercase letter".

Best practice: A practice (not including naming guidelines) recommended by some authors in some contexts to improve the quality of conceptual schemas. For example, some authors recommend that the type of an attribute should not be an entity type [11, p. 189]. This becomes the problem issue I_6 = "The type of an attribute is an entity type". Another example is the checking issue I_{11} = "The aggregation kind of a property is correct" which may be enforced by a method that analyzes conceptual schemas in terms of collaboration patterns and determines that an aggregation could be better expressed by a composition [7].

2.2 Formalization

In this section, we summarize the formalization of issue and issue type (see [8] for the complete details).

Let S be a schema that consists of n schema elements which are an instance of the corresponding schema metatypes. We define a conceptual schema quality issue instance (for short, issue) of type I_x as a fact $I_x(e_1, \ldots, e_m)$ where e_1, $\ldots$, e_m are schema elements, $m \geq 1$. In a schema there may be several distinct issues of the same issue type, and there may be several issues for the same tuple $\langle e_1, \ldots, e_m \rangle$.

For example, consider the problem issue I_4 = "A property derived by union does not have specific properties" (see Tab. 1). Assume that p is the schema element corresponding to the property *relative* (of entity type *Person*) which is derived by union, but the schema does not have at least one property defined as subset of *relative* (for example, *parent* or *sibling*). In this case, *relative* cannot have instances and therefore it is not correct. The issue can be formalized as an issue $I_4(p)$ of type I_4.

Formally, an issue type I_x is a tuple:

$$I_x = \langle \mathcal{S}_x, \phi_x, \rho_x, \mathcal{K}_x, \mathcal{A}_x, \mathcal{O}_x, \mathcal{P}_x \rangle \tag{1}$$

where

- $\mathcal{S}_x$ is the *scope*, which consists of a tuple $\mathcal{S}_x = \langle T_1, \ldots, T_m \rangle$, $m \geq 1$, of schema metatypes. At a given time, there could be an instance of I_x for each element of the Cartesian product of $T_1 \times \cdots \times T_m$. In the example, $\mathcal{S}_4 = \langle \mathit{Property} \rangle$.
- ϕ_x is the *applicability condition*. In general, only a subset of the elements of $\langle T_1 \times \cdots \times T_m \rangle$ may raise an issue of type I_x. The potential set $\mathit{Pot}(I_x)$ of elements of $T_1 \times \cdots \times T_m$ that may raise an issue of type I_x is defined as:

  $$\mathit{Pot}(I_x) = \{\langle e_1, \ldots, e_m \rangle \mid \langle e_1, \ldots, e_m \rangle \in T_1 \times \ldots \times T_m \wedge \phi_x(e_1, \ldots, e_m)\}$$

  where ϕ_x is the applicability condition. In the example, $\phi_4(p)$ = "p is a property derived by union".
- ρ_x is the *issue condition*. An instance of issue type I_x at a given time is an element of $\mathit{Pot}(I_x)$ that satisfies the issue condition $\rho_x(e_1, \ldots, e_m)$ at that time. The set $\mathit{Raised}(I_x)$ of issues of type I_x raised at a given time is:

  $$\mathit{Raised}(I_x) = \{\langle e_1, \ldots, e_m \rangle \mid \langle e_1, \ldots, e_m \rangle \in \mathit{Pot}(I_x) \wedge \rho_x(e_1, \ldots, e_m)\}$$

  In the example, the issue condition would be (written in the appropriate language) $\rho_4(p)$ = "there are no other properties defined as subsets of p".
- $\mathcal{K}_x$ is the *kind* of the issue type, which may be either *Problem* or *Checking*.
- $\mathcal{A}_x$ is the *acceptability* of the issue type, which may be *True*/*False*. An issue type may be defined as acceptable if the method engineer believes that some of its instances are acceptable in some circumstances. The exact meaning of the acceptability depends on the issue kind. If $\mathcal{K}_x$ = *Problem* issue, then:

  $\mathcal{A}_x$ = *True* means that a conceptual modeler may find it reasonable that there are some instances of I_x in a particular schema.

  $\mathcal{A}_x$ = *False* means that all issues of type I_x must be solved.

  If $\mathcal{K}_x$ = *Checking* issue, then:

$\mathcal{A}_x$ = *True* means that a conceptual modeler may find it reasonable not to check some instances of I_x in a particular schema.
$\mathcal{A}_x$ = *False* means that all issues of type I_x must be checked.

In the example, the issue should not be acceptable. If it was acceptable, the modeler would be able to accept issues of this type, resulting in an incomplete or incorrect schema.

- $\mathcal{O}_x$ is a set of *issue actions*. Each issue action of I_x with $\mathcal{S}_x = \langle T_1, \ldots, T_m \rangle$ is an operation $op(p_1{:}T_1, \ldots, p_m : T_m)$ whose intended effect depends on K_x. If it is a problem issue, then the execution of the operation solves the issue $I_x(e_1, \ldots, e_m)$. If I_x is a checking issue, then the execution sets the state of the issue to *Checked*. Issue actions can be *automated* or *manual*. In the example, one possible automated operation is to set the property to base.
- $\mathcal{P}_x$ is a set of *precedents*, which is a set of issue types such that the instances of I_x should not be considered if there are unsolved issues of the types in $\mathcal{P}_x$ (see [8] for more details). In the example, $\mathcal{P}_4$ would be the empty set. A different example is I_{11} = "The aggregation kind of a property is correct", which can only be evaluated if there are no unsolved issues of type I_3 = "A property has no stereotype" [7].

3 Conceptual Schema Quality Issues in Current IDEs

In this paper we aim to analyze several IDEs that can be used to perform UML conceptual modeling activities. The list of tools presented in Tab. 2 has been obtained from the Open Directory Project (ODP) [12] mainly, and complemented with some additional tools that, according to [13], are being used by UML practitioners nowadays. After a quick analysis of the tools included in ODP, we decided to exclude from our analysis the tools that are not intended for conceptual modeling tasks[3]. To our knowledge, this is the first work that provides a comprehensive review of the quality properties supported by current IDEs.

In order to determine how IDEs deal with quality issue types, we manually tested each tool, and we reviewed the feature list that is published in each tool's website. Table 2 summarizes the results for the following criteria:

Issue Types: It shows the number of issue types enforced by the IDE in each category. The results are presented according to the categories introduced in the previous section. For the syntactic category, the symbols used are:
- *full* (●), if the IDE claims it controls all metamodel constraints,
- *partial* (◐), if the IDE controls only a subset of these constraints, and
- *none* or *unknown* (○), otherwise.

Issue Tolerance: IDEs react differently when they detect an issue. The appearance of an issue may be:

[3] Some IDEs focus on other activities like code generation, reverse engineering, or transformations from natural language specifications to UML conceptual schemas, among others.

Table 2. Quality issue enforcement in current IDEs

Tool	Issue Type					Issue	Extensi-	Corr.
	S	S+	Ba	NG	Be	Tolerance	bility	Actions
ArgoUML	◐	3	3	3	5	ALLOWED	○	●
Astah	◐	-	-	-	-	FORBIDDEN	○	○
Blueprint Software Modeler	◐	-	-	-	-	MIXED	○	○
Cadifra UML Editor	○	-	-	-	-	NONE	○	○
Design Pattern Autom. TK	◐	-	-	-	-	NONE	○	○
Dia	○	-	-	-	-	NONE	○	○
Eclipse UML2Tools	◐	-	-	-	-	MIXED	○	○
Enterprise Architect	◐	-	-	-	-	MIXED	○	○
Fujaba	◐	-	-	-	-	MIXED	○	○
Gaphor	○	-	-	-	-	NONE	○	○
Generic Modeling Env. (GME)	◐	1	-	-	-	MIXED	●	○
IBM Rational Rose	◐	-	-	-	-	MIXED	●	○
MagicDraw UML	◐	-	-	-	-	MIXED	●	○
MetaEdit+	○	-	-	-	-	NONE	○	○
MosKITT	◐	-	-	-	-	MIXED	○	○
ObjectiF	◐	-	-	-	-	FORBIDDEN	○	○
Oclarity	●	-	-	-	-	ALLOWED	○	○
Poseidon for UML CE	○	-	-	-	-	NONE	○	○
SDMetrics	◐	6	3	2	2	ALLOWED	●	○
Umbrello UML Modeler	◐	-	-	-	-	FORBIDDEN	○	○
UML Sculptor	○	-	-	-	-	NONE	○	○
UML/INTERLIS-editor	◐	-	-	-	-	MIXED	○	○
UMLet	○	-	-	-	-	NONE	○	○
UModel	●	-	-	-	-	FORBIDDEN	○	○
USE	●	-	-	-	-	FORBIDDEN	○	○
Violet	○	-	-	-	-	NONE	○	○
Visio	◐	-	-	-	-	FORBIDDEN	○	○
Visual Case	○	-	-	-	-	NONE	○	○
Visual Paradigm (VP)	◐	-	-	-	1	MIXED	○	●

- *Forbidden*, which means that the IDE does not allow a schema change such that it raises the issue, and therefore it "rolls back" the modeler's change to avoid the issue,
- *Allowed*, which means that the change is accepted, and the IDE notifies the modeler somehow of the issue,
- *Mixed*, which means that the IDE may allow having some issues raised, whilst prohibiting others.

Extensibility: IDEs enforcing issue types may offer one or more mechanisms to extend the issue types they deal with. New issue types may be added using a constraint language such as OCL, or creating a plugin that implements an issue type interface. The symbols used are:

- ●, if the tool suggests some sort of extension mechanism, and
- ○, otherwise.

Corrective Actions: Whenever an issue is raised in the conceptual schema, the conceptual modeler has to take some action to fix it. The set of actions that may fix an issue can (and should) be included in the IDE. The symbols used are:

- ●, if the tool offers one or more issue actions, and
- ○, otherwise.

The analysis of the 29 IDEs showed very interesting—as well as unexpected—results. First of all, we can see in Tab. 2 that the vast majority of the analyzed IDEs assist modelers in dealing with syntactic issue types. However, only three IDEs fully support these category of issue types, whilst the others only deal with them partially. After testing each IDE individually, we discovered that many IDEs do not control syntactic issue types like having, for example, (1) a cycle in a generalization hierarchy, (2) a property whose lower multiplicity has a greater value than the upper, or (3) a namespace that contains two different elements that are indistinguishable.

Second, our analysis also shows that, in general, IDEs have little or no support on issue types that are not syntactic. Nonetheless, some IDEs already integrate a few issue types in their own catalogs. Table 3 shows the 21 conceptual schema quality issue types that are included in *ArgoUML*, *SDMetrics*, *Generic Modeling Environment*, and *Visual Paradigm*.

Some authors agree that inconsistencies should be tolerated—that is, model consistency has not to be preserved at all times—, and it is the IDE's responsibility to manage the "detection of inconsistencies" [14–16]. In this sense, current IDEs are moving towards this behaviour; that is, they allow some or all issues to be raised. In fact, only one third of the analyzed IDEs forbid the creation of issues at any time. It is also interesting to note that the four IDEs that include non-syntactic issue types tolerate issues: they allow issues of (some or all) types to be raised.

It is obvious that corrective actions are only useful if issues can be raised. However, solely two IDEs include corrective actions, which means that the assistance a modeler receives in order to solve issues is very little. *ArgoUML* is the most complete IDE in this sense: it is the tool that includes more corrective actions. Moreover, some of its actions can fix an issue automatically.

Finally, our analysis also shows that extension mechanisms are not widely present in current IDEs. Only four of them provide a powerful mechanism—that is, OCL or a similar language—to define new issue types. In the next section, we discuss in-depth the extensibility of these IDEs, comparing the issue type formalization they implement to ours.

4 A Catalog of Conceptual Schema Quality Issues

In principle, all quality properties proposed in the literature could be defined in terms of our formalization [8] and included in a catalog. The main advantage of such a catalog could be to have a centralized and uniform definition

Table 3. Conceptual Schema Quality Issues included in some IDEs

Syntax+	**ArgoUML**	**SDMetrics**	**GME**	**VP**
1. Overriding attribute does not redefine the overrides one	○	●	○	○
2. Named element has an illegal name (invalid characters)	●	○	●	○
3. Unnamed class	●	●	○	○
4. Unnamed attribute	●	○	○	○
5. Unnamed datatype	○	●	○	○
6. Property without a type	○	●	○	○
7. Class has specializations and it is marked as a *leaf*	○	●	○	○
8. *n*-ary association has a navigable member end	○	●	○	○
Basic Properties				
9. Binary association with both member ends as aggregate	●	●	○	○
10. Abstract class is not instantiable	●	●	○	○
11. Cycle of composition relationships	●	○	○	○
12. Abstract class has a parent class that is concrete	○	●	○	○
Naming Guidelines				
13. Class name is not properly capitalized	●	●	○	○
14. Property name is not properly capitalized	●	●	○	○
15. Namespace contains two elements with very similar names	●	○	○	○
Best Practices				
16. Data type as a member end of a binary association	●	○	○	○
17. Class without attributes	●	○	○	○
18. Class with too many associations	●	○	○	○
19. Class with too many attributes	●	○	○	○
20. Class with too many attributes and operations	○	●	○	○
21. Isolated class	●	●	○	●

of all available quality issue types. Once this information is openly and easily accessible, conceptual modelers, students, and practitioners could use it as a reference catalog to improve the quality of their conceptual schemas, specially if IDE developers integrate them into their tools.

The work we are presenting here includes an initial version of this catalog [17]. So far, most issues included in the catalog deal with UML class diagrams. The catalog contains all UML metamodel constraints as syntactic issue types [9], and 59 non-syntactic issue types: 37 problem issues and 22 checking issues. Specifically, it includes 11 syntactic+ issue types, 20 basic properties, 10 naming guidelines, and 18 best practices. These issue types are based on guidelines, recommendations, and best practices that can be found in the literature, as well as all conceptual schema quality issues that are already included in current IDEs. A few examples of the included issue types are making implicit constraints entailed by association redefinitions explicit [18], using a proper capitalization of schema element names

[9, p. 50], writing class names [19–21] or binary association names [22] using the correct syntactic form, ensuring schema satisfiability [23, p. 88], or detecting the situation in which a refactoring would be recommended [24].

Each issue type included in our catalog is defined in XML. The XML includes the different elements of the formalization introduced in Sect. 2.2, as well as some additional meta-data like the *name* or the *description* of an issue type. The applicability and issue conditions are defined using the Object Constraint Language (OCL). The usage of an XML representation provides two key benefits: on the one hand, the catalog and the specification of an issue type can be downloaded and parsed by an IDE automatically; on the other hand, they can be presented in a user-friendly manner by means of XSLT sheets, which allow these XML files to be browsed by conceptual modelers and practitioners using a web browser.

Name
A property derived by union does not have specific properties

Description
According to [9, p. 129], when a property is derived by union, the collection of values denoted by the property in some context is derived by being the strict union of all of the values denoted, in the same context, by properties defined to subset it. In order to compute this collection, it is necessary to specify the specific properties that subset it.

Scope
$\mathcal{S}_4 = \langle$ Property $\rangle$

Applicability Condition
$\phi_4 =$ self.isDerivedUnion

Issue Condition
$\rho_4 =$ not Property.allInstances().subsettedProperty
->includes(self)

Kind
$\mathcal{K}_4 = Problem$

Acceptability
$\mathcal{A}_4 = False$

Issue Actions $\mathcal{O}_4 =$
- [Manual] createAssociationAndAddSubsetTo(p:Property)
- [Manual] addSubsetTo(p:Property)
- [Automatic] setDerivedByUnionToFalse(p:Property)

Precedents
$\mathcal{P}_4 = \emptyset$

Fig. 1. Formalization of the issue type I_4 with our method [8]

Consider, for example, the issue type I_4 = "A property derived by union does not have specific properties", which is formalized[4] in Fig. 1. This issue type applies to all properties in the schema that are derived by union. For each of them, the issue condition checks that there are no properties that subset it. The kind of this issue type is, as we have already stated, *problem* and the issue type is considered to be not acceptable.

[4] We do not show the XML version for the sake of readability.

The formalization also includes several actions that assist the conceptual modeler in solving this type of issues: (1) creating a new association and making one of its member ends a subset of the derived property, (2) making one member end of an already existing association a subset of the derived property, or (3) making it a base property. The first two actions are *manual* because the conceptual modeler has to manually create the association (only in (1)) and select which property subsets the derived-by-union property (in (1) and (2)). The latter is *automatic* because an IDE could automatically change the value of the *isDerivedUnion* of the derived-by-union property to *False*.

4.1 Extending Current IDEs with the Catalog

In general, all IDEs that include some sort of issue types describe them by means of a *context*, which corresponds to the type of the instances for which issues may exist, and a *condition*, which determines whether the issue exists or not for each instance of the context. This formalization of issue type is very similar to a UML metamodel constraint. Besides this, IDEs may also include additional elements to their formalization, such as (1) permitting the modeler to *ignore* issues or (2) one or more *operations* that solve (or help the modeler to solve) an issue instance. According to this description, IDEs implement (ideally) the following formalization of issue type:

$$Z_y = \langle \mathcal{C}_y, \theta_y, \mathcal{I}_y, \mathcal{O}_y \rangle \tag{2}$$

where, as we have already seen,

- $\mathcal{C}_y$ is the *context* in which the issue type has to be evaluated,
- a *condition* θ_y that determines whether the issue exists or not for each instance of the context,
- a *Boolean* value $\mathcal{I}_y$ that specifies whether the issue can be or cannot be *ignored* by the modeler, and
- a set O_y of operations that solve (or help solving) the issue.

Assuming that the previous formalization represents the best case scenario we may currently find in an IDE (only *ArgoUML* includes all these elements into its definition of issue types), it is clear that this formalization (2) is less expressive than ours (1). At a first glance, the major problem IDE developers would face when trying to integrate our catalog in their tools is that they cannot define *checking* issue types. In general, a checking issue type requires the modeler to check something or perform some action manually, and then notify the tool that the check has been performed. Current IDEs do not provide any mechanisms to the modeler to perform this notification. To overcome this problem, an IDE developer may be tempted to map a *checking* issue type from our formalization to an *ignorable* issue type in (2). However, according to the dictionary, to ignore [an issue] means to "refuse to take notice of or acknowledge [the issue]; disregard [the issue] intentionally", so this mapping would be semantically incorrect and inaccurate.

Another problem that may arise when defining an issue type from (2) to ours is adapting the scope. Our formalization allows an issue type to have more than one metatype as a scope. Using more than one metatype provides better feedback to the modeler under certain circumstances.

Consider, for example, the *pull up property* refactoring, which consists in removing an attribute named n from one (or more) specific classes and defining it in the general class [24]. A situation in which this refactoring would be highly recommended is when the attribute is repeated among all specific classes of a complete generalization set. This situation would be formalized as a *problem* issue type and, obviously, one of the issue actions that may fix an issue of this type would be applying the refactoring. If the scope admits only one metatype, the best candidates to be set as the scope are *GeneralizationSet* or *Property*. The former would raise an issue for any generalization set whose specific classes have one (or more) attributes with the same name, whilst the latter would raise an issue for every single attribute whose name is repeated in the generalization set. However, using a scope with more than one element such as ⟨*GeneralizationSet*, *String*⟩, an issue would be raised only once for each attribute name n that is repeated among all specific classes of a generalization set and, thus, it would provide a more accurate and concise feedback to the modeler.

Finally, IDEs do not allow the definition of precedents. Precedents filter the amount of feedback the modeler receives, because only those issues that may be triggered are actually triggered. Currently, IDEs may be able to implement a similar behaviour by duplicating the definition of a precedent inside their issue condition. However, duplicating information is usually a source of errors, and makes the maintenance and comprehension of their catalogs more complicated.

5 Evaluation

This section describes an experiment conducted to evaluate the support given by the current IDEs to enforce the quality of conceptual schemas. We randomly selected 13 conceptual schemas developed by students, as part of their final projects, during the last year of their Computer Science degree. Table 4 summarizes the characteristics of the 13 conceptual schemas.

We defined all the conceptual schemas using the two current tools that provide better support to detect quality issues: *ArgoUML* and *SDMetrics*. As indicated in

Table 4. Summary of conceptual schema characteristics

UML Element	Average	Minimum	Maximum
Classes	47	10	366
Associations	33	5	264
Association Classes	7	0	55
Specializations	19	2	158
Attributes	144	11	1144
Invariants	39	0	386

Tab. 2, *ArgoUML* detects 3 syntactic+ issue types, 3 basic properties, 3 naming guidelines and 5 best practices, whereas *SDMetrics* detects 6 syntactic+ issue types, 3 basic properties, 2 naming guidelines and 2 best practices. Both tools partially check syntactic issue types but we do not consider them in this evaluation. We found that all the conceptual schemas present quality issues. Tables 5 and 6 show the problem issues detected for each category and for all conceptual schemas by *ArgoUML* and *SDMetrics*, respectively.

Table 5. Issues detected by *ArgoUML* [25]

	Syntax+	Basic Prop.	Best Pract.	Naming	Total	Avg.
Problem	0	15	500	23	538	41.38
Avg.	0	1.15	38.46	1.77	41.38	

Table 6. Issues detected by *SDMetrics* [26]

	Syntax+	Basic Prop.	Best Pract.	Naming	Total	Avg.
Problem	91	15	178	23	307	23.62
Avg.	7	1.15	13.69	1.77	23.62	

ArgoUML detects, on average, 42 problem issues for each conceptual schema whereas *SDMetrics* detects 24. In the light of the results it may seem that the support given by the tools to detect quality issues is adequate. However, to determine what *could* be done ideally, we analyzed the conceptual schemas using the catalog presented in the previous section. This catalog, defined following the formalization presented in Sect. 2, detects not only the problem issue types included in *ArgoUML* and *SDMetrics*, but also many others (including several checking issues types). When the schemas are analyzed using our catalog, the number of issues they present increases considerably. In fact, the number of detected problem issues is, on average, 8 times higher than the ones detected by *ArgoUML* and 15 times higher than the ones detected by *SDMetrics*. We also detected more than 1300 checking issues, which require the conceptual modeler to check something that *may* be a defect. Table 7 shows the detected issues for each category and for all conceptual schemas using our catalog.

Table 7. Issues detected by our catalog [17]

	Syntax+	Basic Prop.	Best Pract.	Naming	Total	Avg.
Problem	247	3317	320	585	4469	343.77
Checking	0	571	726	86	1383	106.38
Total	247	3888	1046	671	5852	450.15
Avg.	19	299.07	80.46	51.61	450.15	

The results are conclusive. Although current IDEs such as *ArgoUML* and *SDMetrics* give support to detect problem issues in conceptual schemas, this

support might be much greater. The use of a broader catalog with relevant quality issue types for conceptual modeling increases, as expected, the number of detected issues and, therefore, fosters the improvement of the quality of the developed conceptual schemas. Although *ArgoUML* and *SDMetrics* provide extension mechanisms to incorporate new issue types, they do not permit to add the 22 checking issue types defined in our catalog. Moreover, 6 of the issue types defined in our catalog have a scope with two or more metatypes and 10 have precedents. These issue types are more difficult to integrate into *ArgoUML* and *SDMetrics* than into our catalog.

6 Conclusions

The starting point of this paper has been the view that one of the most effective ways of increasing the quality of conceptual schemas in practice is by using an IDE that enforces all relevant quality issues. With that view, we have analyzed the support provided by twenty-nine IDEs in the enforcement of quality issues, and we have seen that only two of them provide some significant support. We have put forward the idea of a catalog that includes all quality issues proposed in the literature, uniformly defined. We have presented an initial, globally-accessible version of that catalog that includes 59 non-syntactic quality issues.

We have then compared the support provided by current IDEs with the one that could be provided by those IDEs if they enforced all quality issues defined in the catalog. The conclusion has been that there is a large room for improvement. We have experimentally evaluated the benefit of the additional support in the quality of thirteen conceptual schemas developed by students as part of their final projects. We have seen that the benefit is significant.

We plan to continue this work in two directions. The first is to extend the catalog so that it includes most of (if not all) quality issues published in books, journals, and conference papers in the field of conceptual modeling of information systems. We believe that the existence of such catalog would be of great value to the information systems engineering community in general, and particularly to IDE developers. So far, we have focused on the notion of quality issues of conceptual schemas, but this could be also applied to other model-based software artifacts, as it is currently done in some IDEs. A second direction then is to explore the benefits of our approach in that context.

Acknowledgments. This work has been partly supported by the *Ministerio de Ciencia y Tecnología* and *FEDER* under project TIN2008-00444/TIN, *Grupo Consolidado*, and by *Universitat Politècnica de Catalunya* under FPI-UPC program.

References

1. Bolloju, N., Leung, F.S.: Assisting novice analysts in developing quality conceptual models with UML. Commun. ACM 49(7), 108–112 (2006)
2. Krogstie, J.: Model-Based Development and Evolution of Information Systems – A Quality Approach. Springer (2012)

3. Lindland, O.I., Sindre, G., Sølvberg, A.: Understanding quality in conceptual modeling. IEEE Softw. 11(2), 42–49 (1994)
4. Moody, D.L.: Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data Knowl. Eng. 55(3), 243–276 (2005)
5. Shanks, G., Tansley, E., Weber, R.: Using ontology to validate conceptual models. Commun. ACM 46(10), 85–89 (2003)
6. Si-Said Cherfi, S., Akoka, J., Comyn-Wattiau, I.: Conceptual modeling quality - from EER to UML schemas evaluation. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 414–428. Springer, Heidelberg (2002)
7. Bolloju, N., Sugumaran, V.: A knowledge-based object modeling advisor for developing quality object models. Expert Syst. Appl. 39(3), 2893–2906 (2012)
8. Aguilera, D., Gómez, C., Olivé, A.: A method for the definition and treatment of conceptual schema quality issues. In: Atzeni, P., Cheung, D., Ram, S. (eds.) ER 2012 Main Conference 2012. LNCS, vol. 7532, pp. 501–514. Springer, Heidelberg (2012)
9. Object Management Group (OMG): Unified Modeling Language (UML), Superstructure – version 2.4.1 (2011)
10. McAllister, A.: Complete rules for n-ary relationship cardinality constraints. Data Knowl. Eng. 27(3), 255–288 (1998)
11. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley (2005)
12. Mozilla: Open Directory Project (ODP) – List of UML tools, `http://www.dmoz.org`
13. Davies, I., Green, P., Rosemann, M., Indulska, M., Gallo, S.: How do practitioners use conceptual modeling in practice? Data Knowl. Eng. 58(3), 358–380 (2006)
14. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental detection of model inconsistencies based on model operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 32–46. Springer, Heidelberg (2009)
15. Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. 20(8), 569–578 (1994)
16. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, pp. 329–380. World Scientific (2001)
17. Aguilera, D., Gómez, C., Olivé, A.: Issue catalog, `http://helios.lsi.upc.edu/phd/catalog/issues.php`
18. Costal, D., Gómez, C.: On the use of association redefinition in UML class diagrams. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 513–527. Springer, Heidelberg (2006)
19. Ambler, S.W.: The Elements of UML 2.0 Style. Cambridge University (2005)
20. Chen, P.: English sentence structure and entity-relationship diagrams. Inf. Sci. (2-3), 127–149 (1983)
21. Hay, D.C.: Data Model Patterns: Conventions of Thought, 1st edn. Dorset House Publishing (1996)
22. Meziane, F., Athanasakis, N., Ananiadou, S.: Generating natural language specifications from UML class diagrams. Requir. Eng. 13(1), 1–18 (2008)
23. Olivé, A.: Conceptual Modeling of Information Systems. Springer (2007)
24. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
25. ArgoUML: ArgoUML, `http://argouml.tigris.org`
26. SDMetrics: The software design metrics tool for the UML, `http://sdmetrics.com`