Description and Implementation of a Style Guide for UML

Mohammed Hindawi, Lionel Morel, Régis Aubry, Jean-Louis Sourrouille

Description and Implementation of a Style Guide for UML

Mohammed Hindawi, Lionel Morel, Régis Aubry, Jean-Louis Sourrouille

Université de Lyon
INSA Lyon, LIESP, Bât. B. Pascal, 69621 Villeurbanne, France
{Mohammed.Hindawi, Lionel.Morel, Régis.Aubry, Jean-Louis.Sourrouille}@insa-lyon.fr

Abstract. Model quality is still an open issue, and a first step towards quality could be a style guide. A style guide is a set of rules aiming to help the developer improving models in many directions such as good practices, methodology, consistency, modeling or architectural style, conventions conformance etc. First, this paper attempts to clarify the meaning of notions being used such as rule or modeling domain semantics. Then, several examples illustrate a possible classification of rules, and the verification process is detailed. A style guide is not universal: each project manager should be able to customize his/her set of rules according to specific needs. In addition to rules expressed in OCL, we describe a user interface to facilitate the specification of rules based on quantifiers, along with the translation of these rules into OCL.

1. Introduction

In the emerging context of Model Driven Engineering, software development more and more focuses on models. On the other hand, the software engineering community has known for a long time the advantages of early fault detection. Thus to check models at the beginning of the development cycle appears a promising direction. For a reader, models have a meaning related to the application domain, but generally, a model checker only knows the semantics of the modeling domain. As a result, application domain semantics is a matter for users, while tools may help for modeling issues. Beyond faults, which are all the more difficult to find that models are imprecise and abstract, many model properties are of interest for developers.

A style guide is a set of rules aiming to help the developer improving models in many directions such as good practices, methodology, consistency, modeling or architectural style, conventions conformance etc. Some rules are hints while others are warnings, which means that they are potential errors. These rules check “good properties”, which are kinds of quality criteria. However, the quality of a model is relative to application requirements, and errors are ignored as long as they do not go beyond the quality objectives that have been set according to requirements. Conversely, a style guide checker notifies all the rule violations: the developer defines his/her own objectives and priorities, often based on error gravity.

Developers could be in charge of rule checking. However, in practice, only automated checks are suitable not to increase developer burden, but also because manual checks are unsure. Consequently, as many rules as possible should be given a formal description, and only rules expressed in natural language will require manual checks. There is no universal style guide. Each development team may have its own needs depending on applications, hence we need an easy way to specify rules. UML provides OCL as a description language, but non-experts find it difficult to use. This implies the need for specific tooling to describe and manage a set of rules, and to control the verification process that should be as flexible and automated as possible.

The rest of the paper is organized as follows: section 2 gives definitions and attempts to clarify the meaning of used notions; section 3 classifies some rule descriptions; section 4 shows the verification process, defines the main tool components, and details the user interface to specify rules including the translation into OCL. Then we discuss related works and conclude.

2. Context and definitions

This section describes the context of the work and defines some notions used in the rest of the paper. Moreover, we aim to clarify what it means to apply rules to a model.

2.1. Syntactic vs. Semantic Correctness

In software engineering, a model is a representation, from a given point of view, of a system expressed in some formalism [4]. The formalism definition includes notions and their semantics. This semantics induces constraints on the model, for instance the semantics of inheritance induces that cycles are not allowed along the inheritance relationship. A model expressed in a formalism is *correct* when it conforms to all the constraints of this formalism. The UML specifies constraints in both OCL and natural language. We call the former *syntactic* constraints and the latter *semantic* constraints[1] [16][6]. A model that meets syntactic constraints is *syntactically correct*, and a model that meets semantic constraints is *semantically correct*. Syntactic constraints can be checked automatically while semantic constraints are left to human users, hence it is not possible to check automatically whether a model is correct or not. In everyday cases, UML models are at best syntactically correct but their semantic correctness is unknown. In addition, semantic variation points in the UML specification require human choices. For instance, the choice of the communication policy of a UML *Port* leads to different valid communication sequences: “If several connectors are attached on one side of a port, then any request arriving at the other side of this port will be forwarded on all links *or* only one link...” [17].

In the following, we consider only syntactically correct models. Additional constraints aim to increase the semantic correctness.

[1] Although surprising, this definition has the advantage of being precise: even in programming language, the difference between syntax and semantics is unclear. Going deeper into this issue does not help due to the lack of unambiguous difference between semantic and syntactic constraints expressed both in OCL. Anyway, the reader may think of semantic constraints as constraints specified in natural language.

2.2. Interpretations

A system can be modeled in different ways. A model *interpretation* is defined as the meaning of this model in a semantic domain. A model has generally several interpretations in a semantic domain (Fig. 1), but the set of interpretations of an incorrect model is empty in any domain. The natural semantic domain of a modeling language such as UML is the *modeling domain*. There is no consensus about the semantics of a universal modeling domain; hence, we assume that there are several modeling domains, each one with its own semantics, e.g., active objects do not behave the same according to modeling contexts. As all modeling languages, the UML does not meet all modeling needs, while on the other hand it allows expressions that the semantics of modeling domains may forbid, e.g., to send a signal to a set of objects that cannot catch it. Further, the relationship between the semantics of the modeling domains and the semantics of UML is an important issue, but it is out of the scope of this work. The modeling domain is not to mix up with the application domain. In the modeling domain, a class *Dog* may inherit from a class *Bird*, but in the application domain, this inheritance relationship is surely wrong.

An interpretation is *licit* in a semantic domain when it has a meaning in that domain, i.e., it conforms to the semantics of this domain. A UML model can be both correct and illicit. For instance, a *TypedElement* without *Type* is correct in UML, but when the element is the receiver of a message, it is illicit in most modeling domains.

Refinement. The number of interpretations of a model evolves during the development. An abstract model has a large number of interpretations due to the lack of details. Along the development process, models are refined and become more and more complete and precise; hence, the number of interpretations decreases (Fig. 2a). For example, an undirected association between two classes *A* and *B* has three potential interpretations: *AB* is navigable, *BA* is navigable or *AB* and *BA* are both navigable. Navigability restriction at a further modeling step may reduce these

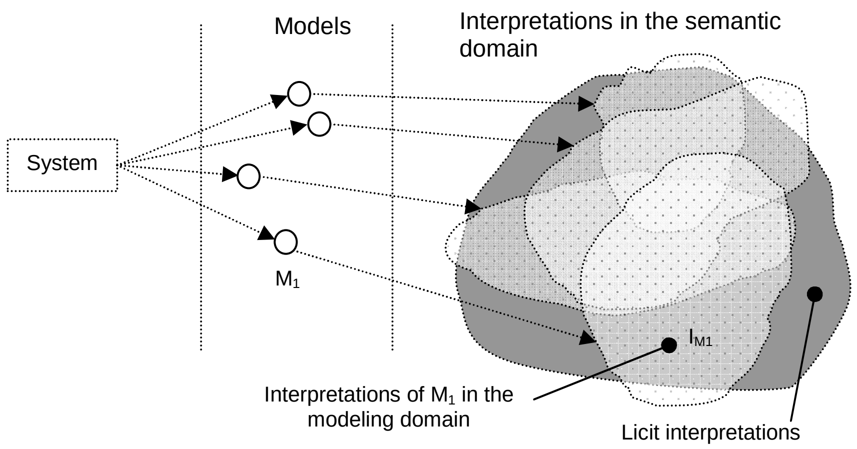

Models
Interpretations in the semantic domain
System
M1
IM1
Interpretations of M1 in the modeling domain
Licit interpretations

Fig. 1. “System – Models” and “Model – Interpretations” relationships interpretations to only one. At the end of the refinement process, one interpretation is selected to generate code. This code is a model whose interpretations form an equivalence class from the developer point of view, i.e., all the interpretations are equivalent: the expression *a*+*b*+*c* can be interpreted as either (*a*+*b*)+*c* or *a*+(*b*+*c*).

Checking models. Within a domain, a model is *consistent* when it conforms to the semantics of this domain. Hence, a correct UML model that is consistent in a modeling domain has at least one licit interpretation. There is no formal definition of the semantics of modeling domains, but many works propose consistency rules to check that models meet some semantic domain constraints (650 rules in [14]).

Modeling domain constraints that can be expressed in a formal language are easy to check. The outstanding issue is how to check models to meet semantic constraints? There is no fundamental difference between semantic constraints from UML and from the modeling domain. Both are expressed in natural language, both apply to models, and modeling domain constraints aim to reject models with no licit interpretation. A first idea is to define *consistency rules* that are stronger than the actual semantic constraints, but that we can express in a formal way. These rules reduce the expressive power of UML (Fig. 2b), i.e., reject potentially correct models and forbid some interpretations, but it is the price for automating checks. A second idea is to define rules that will help the developer to make well-formed models. These rules forbid model expressions leading *generally* to models with illicit or questionable interpretations. Finally, human reviews help finding problems that formal rules cannot detect. At code generation, model analysis will reveal errors but it is too late. The checking process fails when an error that was visible in a model is discovered at run time.

2.3. Style guide

A style guide defines a set of rules that any model must conform to. Style guides reduce the number of acceptable models and force developers to make models owning the wished properties, which results in smaller sets of licit interpretations (Fig. 2b).

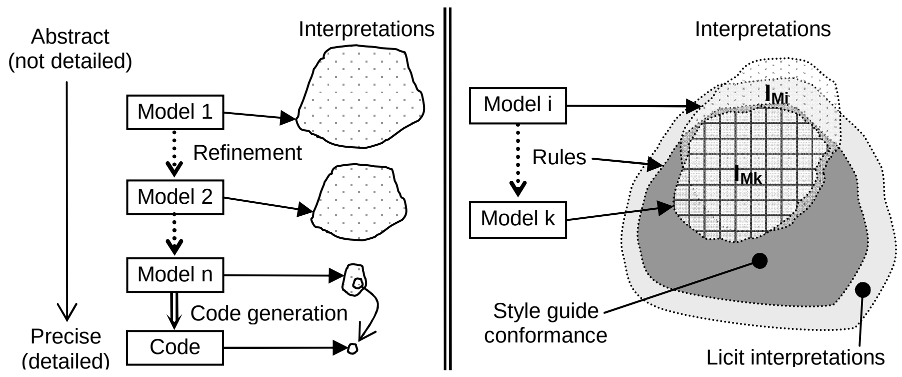


Fig. 2. (a) Model evolution and interpretations (b) Reduction of interpretations

Unlike language constraints that should not be violated, bypassing rules that are simple hints is allowed. In addition to consistency rules, we consider the following kinds of rules:

- Language conventions are rules agreed within a group,
- Guidelines aim to help developers making well-formed models,
- Methodologies induce constraints aiming to make better models from given points of view. They force activities order, model form, deliverables, etc. These rules depend on the chosen methodology,
- Good practice rules come from experience in specific domains, including modeling domain, for instance developers know for a long time that low coupling between elements is better.

Checking model conformance to a style guide is usually a human issue. Due to the high cost of human reviews, the return on investment seems unsure. To reduce the human burden, this paper details an approach to describe and implement an automatically checkable style guide. We define an architecture on top of existing tools, and a checking process. The description of a style guide goes beyond a paper because there are too many rules. Moreover, some are common to most users while others depend on needs that are context specific.

2.4. Style guide and Quality

The style guide defines the boundaries of the set of models owning the wished properties. Beyond model correctness and consistency, the style guide aims to help developers designing models with a better quality, for instance using good practices. From this viewpoint, a style guide is a key element in the quality management process. Of course, the quality of a model that does not meet all the style guide rules is not definitely low, hence the link between rule violation and quality is to be clarified.

Within a multilevel framework for quality assessment such as ISO9126 [8] or [11], rule violation is at the level of metrics. Metrics are aggregated to form attributes, which in turn are aggregated to form characteristics. The quality of a model is not subject to conformance to some individual rules, but rather to some statistical knowledge embodied as threshold values for attributes and characteristics. These thresholds come from quality objectives that are set according to specific needs of applications. From the quality point of view, only deviations from these values will lead to corrections, otherwise the model has the expected quality. While the style guide notifies all rule violations, non-quality is detected only when the combination of a set of metrics reach critical thresholds.

Both style guide and quality assessment detect failures, i.e., non-quality. The software engineering community usually applies the hidden rule “a model with no failure is good”, but nobody knows to what extent a model is good. To ease model comparison, each rule has a gravity level: some violations are just warnings or hints while others are serious errors. Developers know that no serious rule violation should remain while some warnings are acceptable.

Research about software quality in usual programming has reached a high degree of knowledge and skills for a long time. Software managers definitely know the impact of software quality. In spite of this high theoretical maturity level, code quality remains an under-exploited way to improve software. The main lessons learned from surveys show that quality should be provided at no cost, with a suitable support, and should not induce delays in the project. Thus we should pay a great attention to the implementation of the style guide: integration of the verifier within the modeling tool, very simple checking process, flexible user interface, easy rule description, etc.

3. Rules

3.1. Identifying and classifying rules.

Models are checked along several *dimensions* corresponding to different software engineering areas such as methodology, good practices, or modeling. The semantics of each area induces rules. Since generally this semantics is not described, experts from these areas are in charge of rule identification. The dimension is our main structuring property of rules. In addition, each rule owns a set of properties aiming to explain it, to give further comments, to specify gravity, to link it with model parts or development process stages, to specify and implement it, to describe correction actions, etc. These properties are needed at any moment, for instance to classify violations according to their importance, to organize and manage rule description, to help the developer dealing with errors. We give below examples of rules classified by dimension, although rules might often be attached to several dimensions. As mentioned above, some syntactic rules can be stronger than the actual semantic constraints to allow writing them in OCL. Rule descriptions are deliberately short and sometimes imprecise due to available space:

– **Methodology** rules come from method description, e.g., “The application domain model is mandatory”, or “Any communication between actors and subsystem goes through an interface class” (USDP [9]). This latter rule aims to limit changes to a set of well-identified classes when communication protocols between actors and subsystem are modified. The UML itself induces methodology rules, e.g., “Each use case describes at least one scenario to be specified as a sequence diagram”.
  Within the development process, methodologies distinguish steps or *phases* [15] such as requirement elicitation, elaboration, or detailed design. Whatever the methodology, these phases are required to identify moments in the life cycle of artifacts, and as a result to identify levels of abstraction. Each part of a model can be in a different phase. The phase is used to select the set of rules to be applied to each part of a model at a given moment. Beyond phases, to make a distinction between Platform Independent Models and Platform Specific Models allows detecting model expressions that are forbidden at a given stage of the development, and helps keeping the wished independency level.
– **Common methodology** gathers rules that applies whatever the methodology. They come from skills of experienced developers, e.g.: “A black box sequence diagram only holds actors and a subsystem (definition)”, “A white box sequence diagram

holds objects, ports and components and the only actor that triggers the initial stimulus (definition)" or even "A black box sequence diagram only holds communications between actors and a subsystem, not between actors".

- **Consistency** rules detect meaningless expressions in the modeling domain, e.g., "The initial stimulus in a sequence diagram is triggered by an *Actor* or a *Port*, i.e., neither a class instance nor a *Component*", or a usual one "Navigability: any message in a sequence diagram is sent to an accessible receiver, either method parameter or attribute/role in an association". Based on redundancies in the model, some rules detect inconsistencies, e.g., "Within a sequence diagram, actor-to-subsystem interactions should correspond to associations between actors and use cases of this subsystem".
- **Modeling style** rules detect expressions that are *generally* meaningless in the modeling domain. Unlike consistency rules, breaking these rules is tolerated, e.g., "Within any complete class model, a path through navigable associations should link the root class to any class (not a database schema)", or "A sequence diagram is triggered by only one stimulus which is the first in the chronological order". The former rule requires marking the *root* class in the model. The latter rule reduces (apparently) the expressive power but ensures some good properties for the model (no simultaneous waits). Similarly, the rule "Each *ConnectableElement* (from metamodel) in the sequence diagram should be either a port or a class instance" reduces the expressive power forcing components to be connected through ports.
- **Completeness** rules check missing elements from mandatory or even usual links between model elements, e.g., "When the subsystem *B* is an output actor of the subsystem *A*, then *A* should be an input actor in the description of the subsystem *B*", or "Each association actor-to-use case should be implemented in at least one sequence diagram describing a scenario of this use case".
- **Good practices** rules are often hints, e.g., "Cycles along class associations are to be avoided" which aims to reduce coupling, or "To specify systematically bi-directional navigability for associations in a final class model (just before code generation) is likely unnecessary".
- **Conventions** rules are group agreements about syntactic forms, e.g., "Any public name should be capitalized". Within contexts such as education, to meet convention rules is often mandatory, e.g., "When the class of an attribute is represented on the same diagram, drawing the association is mandatory" to avoid hidden associations. Unlike in [12], conventions are limited to a narrow field since we have many other dimensions.
- **Architecture style** rules aim to aid developers to meet software architecture styles such as low-coupling/high-cohesion or Model-View-Controller, e.g.,"A view knows its model but the model does not know its views". Unusual architectures can be detected, e.g., "A subsystem should not appear on its list of actors".
- **Refinement** and **trace** related rules aim to check consistency along the development cycle and to enforce links between model elements at different stages, e.g., "A sequence diagram should be associated with a use case or a less detailed sequence diagram (traceability)".
- **Specification gap** rules deal with non-standard UML. As mentioned before, we consider only models that conform to the UML syntactic specifications (Fig. 3). Modeling tools often allow expressions that do not conform to the UML

specifications. To deal with this issue, a set of specific rules fulfills the gap between each tool and standard UML specifications. This dimension avoids mixing up style guide additional constraints with UML syntactic constraints that tools do not check. To specify these rules is clearly the work of a UML expert.

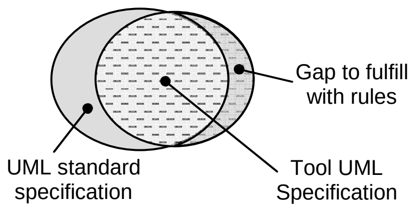


Fig. 3. Standard vs. tool UML specification

Since rules check a large variety of issues, errors resulting from their violation do not have the same gravity. We classify the violations into three categories: *error*, *warning* and *hint*. When a consistency rule is violated, the model has no meaning in the domain of modeling and the violation is an error to be corrected. A violation that might result in a further problem is a warning, and the correction is likely to be preferred. When rules such as methodology are hints for a better modeling process, their violation reduces the quality but to correct them after model completion is not always desirable. Although there is a strong link between dimensions and gravity categories, the gravity is not attached to the dimension: experts and/or project managers set the suitable value.

Finally, to avoid experts specifying rules again and again, a set of standard/common built-in rules should be provided by tools implementing the style guide. Thus, only specification gap rules and customized rules are to be specified.

3.2. Expressing rules

First, rules are expressed in natural language. Next, they have to be formulated in a formal language, preferably OCL. OCL has a power of expression equivalent to first-order logic, but as a main drawback, non-experts generally find it difficult to use. To allow non-experts to formalize rules, we propose a graphical approach on top of OCL

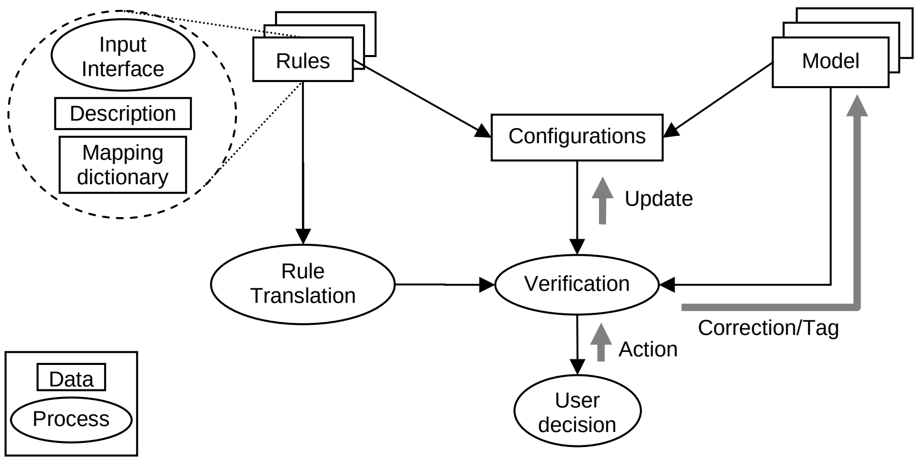


Fig. 4. Verification process.

that rely on generally well-known notions of first-order logic quantification. The rule "Each connected element in a white box sequence diagram should be either a port or an instance of a class" is written:

$$\forall x \in \text{WBSeqDiag}, \text{IsConnectableElem}(x) \Rightarrow \text{IsPort}(x) \vee \exists y, \text{IsClass}(y) \wedge x.\text{class}=y$$

where *WBSeqDiag* is a collection of model elements, *IsT*(*x*) is a predicate which is true when *x* is an instance of *T*. The general form of this rule is:

$$\forall x \in X, R_1(x) \Rightarrow R_2(x) \vee \exists y, R_3(y)$$

Based on quantifiers, the interface provides a limited set of standard forms that ease description but whose expressive power is lower that the OCL one.

The main remaining issue is the link between model notions such as object, class or interaction, and UML metamodel notions. To read and understand the meta-model is hard and reserved to UML experts. In the implementation section, we propose an approach based on rewriting rules that makes it easier to use metamodel notions.

4. Verification Process

The verification process lies on the architecture illustrated in Fig. 4. The set of rules to check depends on the role of the user, the phase in the development process, the kind of model, temporary choices of the developer, etc. Links between rules, model and users are expressed through *configurations* that control the verification process.

To check the style guide and to ensure traceability, we need to annotate the model with data such as the phase or the root class. On the other hand, the implementation of the verification process requires marking models. During the development process, developers regularly check models and occasionally, they are not interested in some types of errors because they focus on other aspects. Thus, marks on model elements specify which rules should be checked. To summarize, we need two types of model tags: adornments to complete the model description, and error-processing tags to control the verification process. UML *taggedValues* are suitable for both purposes.

4.1. Architecture and Process

The Fig. 4 gives the main processes and data of the verification process:

- **Rules** are managed through a user interface described below. Rule properties are stored in a description file separating common rules and properties from specific ones. The mapping dictionary maps rule names expressed in natural language to metamodel notions or OCL expressions (detailed further table 1).

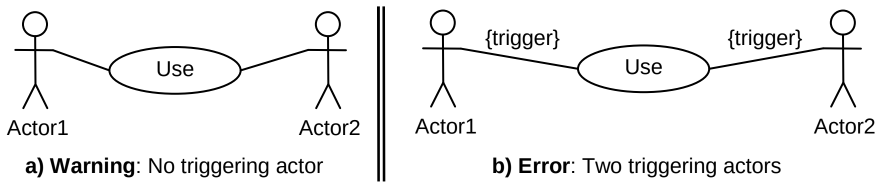

Use
Actor1
Actor2
a) Warning: No triggering actor
{trigger}
{trigger}
Use
Actor1
Actor2
b) Error: Two triggering actors

Fig. 5. Two violations of the same rule

- A **configuration** links together a model and a set of rules that we call a rule package. Packages are defined either using rule properties, e.g., phase *Elaboration*, dimension *Good practices*, gravity *error*, or manually for specific needs. For instance, a team member may decide to keep watch on a particular set of rules. Within rule packages, flags signal rules not to check temporarily.
- The **verification** is directed by configurations and results in messages and actions. When the checking result is *failed*, data about the related rule and model elements are displayed. According to the rule, several choices are offered: to annotate the model with a tag, to invalidate the rule either in the configuration or in the model, to automatically correct the model. Let us take an example to show the verification process for the rule "All initial stimuli (triggers) of a use case come from the same actor". When no trigger is specified (Fig. 5a), checking results in a warning, assuming that the developer does not still complete the model. When two triggers are specified (Fig. 5b), checking results in an error because two actors trigger the use case.

In this particular case, the same rule has two *diagnosis* according to the number of triggers: 0 is a warning, and greater than 1 an error. More generally, a rule can check several properties of an element. Diagnoses avoid several descriptions of the same rule, but each diagnosis holds its own message, gravity, correction, etc.

4.2. Implementation

To check easily the rules and to aid correction, the style guide is embodied in an IDE tool that supplies all the required services for a quicker implementation (the implementation is an ongoing work, we choose to implement on top of Eclipse). The integration into one tool reduces the cost of training and use, and simplifies the checking process. We need plug-in extensions to check rules, to manage errors, to manage configurations, and to manage corrections using model transformations, but the tricky point is rule description. The style guide implementation should provide a set of common built-in rules that users may select through customized configurations. The proposed user interface allows specifying the rules. This section focuses on this interface aiming to hide the trickiest aspects of OCL.

To provide a simplified description of constraints in OCL while keeping the same expressive power is difficult. Our approach is a compromise: the simpler constraints are specified through the provided interface, while the remaining ones are to write directly in OCL. The main form of the user interface (Fig. 6) provides fields to define rules, which allows to expressing a subset of all the possible OCL expressions. We illustrate the description interface with the rule: "Each connected element in a white box sequence diagram should be either a port or an instance of a class". An equivalent expression using quasi-natural language could be: "For any element *e* in a white box sequence diagram, for any connected element *e*, either *e* is a *Port* or *e* is an instance of a class". The later expression is close to first-order logic and its structure fits well with our generic input form that reads as follow:

For any *Sequence diagram* in *Model diagrams* **such as** *White box* is true
    For any connected element *e*
        e is a *Port* **or** *e* is an instance of a *Class*

The next step is the translation into OCL. Quantifiers like interface operators are translated into OCL operations such as *forall*, *select*, *exists*, etc. Notions such as *Sequence diagram* or *Class* are to translate into notions of the UML metamodel. Splitting the OCL rule into small expressions will ease reading.

Translation of "For any *Sequence diagram* in *Model diagrams* **such as** *White box* is true"

First, UML does not supply the notion of diagram: sequence diagrams are *Interaction* owned by packages. From *Package*, the interactions are (Fig. 7):

```
    self.ownedMember->select( i | i.oclIsKindOf(Interaction) )
```

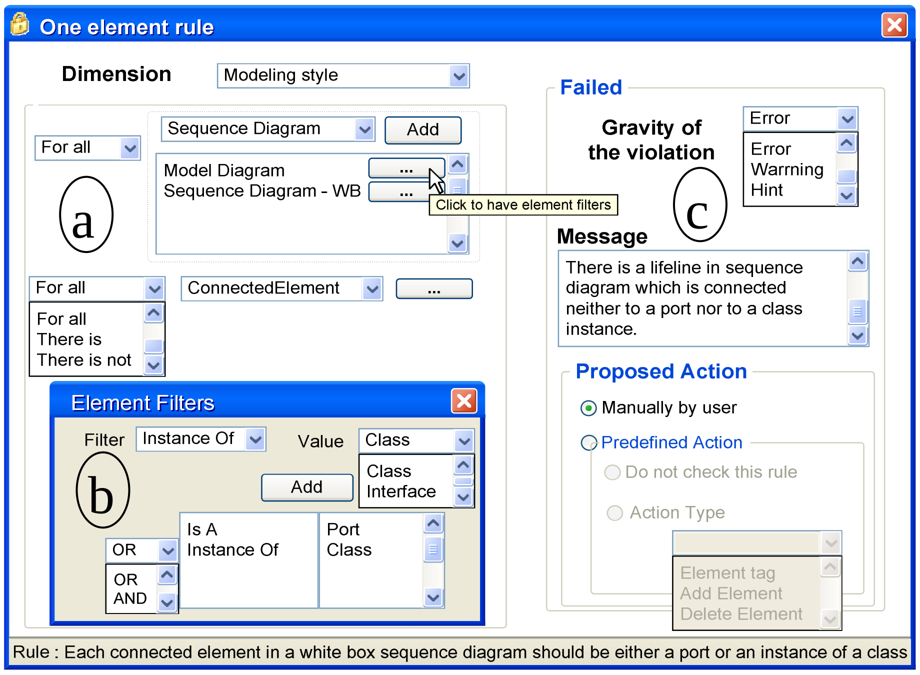
We extract model packages from the metaclass *NamedElement*:

```
NamedElement::allPackages(): Set(Package) ; -- standard operation
    allPackages = NamedElement.allInstances->select( p | p.oclIsKindOf(Package) )
```

Selecting *Interactions*:

```
NamedElement:: sdFilter() : Set(Interaction) ;
    sdFilter = allPackages()->iterate(p ; result :Set(Interaction)={} |
                     result->union(p.ownedMember->select( i | i.oclIsKindOf(Interaction) ) ) )
```

The UML does not provide the user defined notion of *White Box* sequence diagram, which means that the developer has to specify the kind of *Interaction* with a UML TaggedValue kindOf = {BlackBox, WhiteBox, Final}. The operation



There is a lifeline in sequence diagram which is connected neither to a port nor to a class instance.

Rule : Each connected element in a white box sequence diagram should be either a port or an instance of a class

Fig. 6. Rule description interface

GetKindOf('WhiteBox') returns *true* for a WhiteBox *Interaction*. The set of *White Box* sequence diagrams is:

```
NamedElement:: sdWBFilter() : Set(Interaction) ;
   sdWBFilter = sdFilter()->select( i | i.GetKindOf(‘WhiteBox’) )
```

Translation of “For any connected element *e*, *e* is a *Port* **or** *e* is an instance of a *Class”*

From *Interaction*, the form of the path to access to a *ConnectableElement* is:

```
   lifeline[f].represents     -- another rule checks whether connectable element exists
```

From *ConnectableElement*, either the element is a *Port* or the *Property* is typed with a *Class*:

```
   oclIsKindOf(Port)  or type.oclIsKindOf(Class)
```

From *Interaction*, the complete expression is:

```
   lifeline->forAll( f |
            f.represents.oclIsKindOf(Port)  or f.represents.type.oclIsKindOf(Class)   ) )
```

OCL final constraint:

```
NamedElement:: Rule() : Boolean ;
   Rule = sdWBFilter()->forAll( i | i.lifeline->forAll( f |
            f.represents.oclIsKindOf(Port)  or f.represents.type.oclIsKindOf(Class)   ) )
```

The interface is aided to avoid any syntactic error. The user selects values in lists, e.g., when *Model Diagrams* is selected (Fig. 6a), the next filter list supplies only allowed subsets. When *Sequence diagram* is selected, the filter only allows *BlackBox*, *WhiteBox* and *Final* (leaf according to the trace relationship). The translation of the interface expression into OCL is based on a rewriting principle (Table 1): each element in the list has a value in the mapping dictionary. We plan to build the dictionary from an aided interface that lists all the accessible item names in the

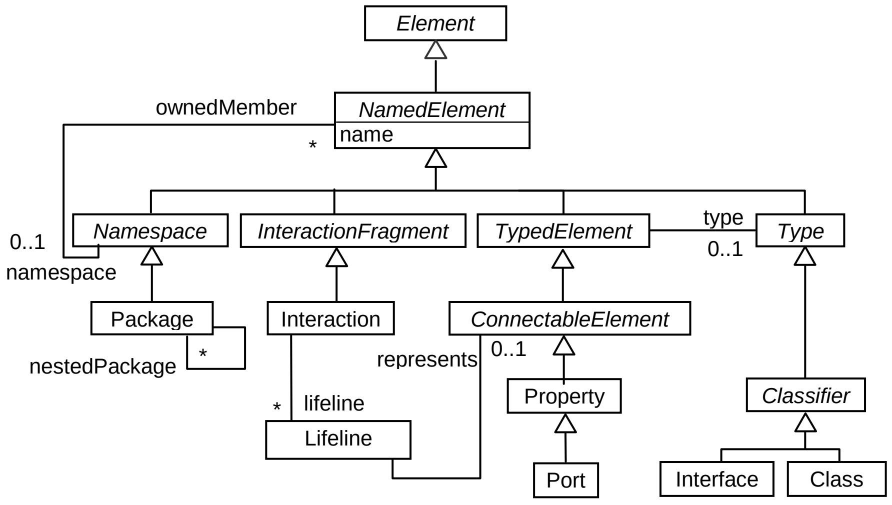


Fig. 7. Root of the required metamodel (from [17])

context. For instance in the metamodel Fig. 7, from *Interaction* the three only choices are *name, lifeline* and *namespace*. This work is close to the definition of a subset of UML [].

The right area of the interface (Fig. 6c) deals with additional properties and corrective actions when the checked element does not conform to the rule. During the verification process, the user is prompted to choose an action that may eliminate or temporarily hide errors. At the end of the process, the remaining violations are displayed: not to disturb the user, checks are only on demand. A command allows removing the model tags and configuration flags that prevent error messages.

4. Related works

Rules. Style guide rules come from various sources. The UML specification [17] is the first information source. Some UML books such as [1][2] include recommendations or style guides that help making "better" models. Methodology books such as RUP [15] or USDP [9] also provide tips and rules. Modeling conventions in [12] correspond to several dimensions within our classification. These modeling conventions proved to be useful to reduce model "defects", which confirms that a style guide is an important issue. In addition, papers related to rules or metrics for UML models are interesting sources [13]. Obviously, a complete style guide description requires a large space [14].

Implementation. A tool may enforce built-in rules that cannot be changed, which relieve from the burden of rule description but prevent customization. A template defines a framework, for instance related to a methodology (e.g., RUP [15]). The user cannot go out of the frame, but remaining dimensions are not checked. "The experience shows that templates are helpful, but they do not ensure that the model as a whole is complete" [7]. To summarize, templates enforce a subset of the required rules only, therefore a preferable way will be to include this subset into a more

Table 1. Mapping dictionary

Name	OCL expression
Model diagrams	NamedElement:: Rule() : Boolean ; R1 = allPackages() - - built-in operation, diagrams are owned by packages

Sequence diagram	R2 = R1->iterate(p ; result :Set(Interaction)={} |
	result->union(p.ownedMember->select(i | i.oclIsKindOf(Interaction))))

White Box	R3 = R2->select(i | i.GetKindOf('WhiteBox'))
Connected element	Rule = R3.lifeline->forAll(f ; x:ConnectableElement= f.represents | R5)
Is A	R5a = x.oclIsKindOf(Port)
Instance Of	R5b = x.type.oclIsKindOf(Class)
or	R5 = R5a **or** R5b

flexible solution.

When rules are written in natural language, the verification of the style guide must be done manually. The description of rules within books such as [1] comes into this category. Works aiming at automating the verification process should express rules in a formal language. The automated verification on demand is the best solution but proposals are still rare [5][7]. In [7], a checker prototype fully automatically verifies models from rules described using a specific language. Although rule description is different, this work is close to our project. We agree with [5] and many others that find it difficult to write rules in OCL. Instead of defining a new language as in [7], we provide a user interface to aid specifying rules that are next translated into OCL. This way we keep a standard language while aiding rule description. In this direction, some works aim to facilitate OCL writing: VisualOCL [3][10] visualizes OCL expressions in an alternative notation. It provides additional information, which increases the usability of OCL. However, to use such tool implies experience in OCL. We try to overcome this issue by proposing an interface easy to use, at a high abstraction level, but rather far from OCL, which implies an additional and tricky translation process.

5. Conclusion

This project is under development[2] and some issues are still pending. The advance of our solution lies in the integration of several technical artifacts to form a complete methodology and tooling. This integration associated with automated checking and style guide customization is a necessary condition for actual use in companies. Some particularly relevant elements in our approach include:

- Selective checking of model parts using tags, which avoid re-checking of rules and messages related to incomplete model parts, therefore lighten the user burden;
- Selective checking according to the current phase in the methodology;
- Customization of the set of active rules in a configuration file according to developer role and experience, application domain, expected “quality”, etc.
- Aid for correcting models: when a rule is violated, the developer may choose a predefined action including model change by applying patterns;
- Aid for defining rules: the graphical interface helps project managers in the definition of rules for their own style guide.

This work is part of a grant aiming to assess model quality. The companies involved in the project will help us to tune quality assessment from metrics. Model quality assessment is relative to application quality requirements and developers do not always know the important quality criteria. A style guide brings the educational aspect needed to help increasing models’ “good properties”: it detects all rules violations but also provides hints, warns to avoid potential errors, and may include company know-how. Finally, a style guide is a quite necessary complement to put into practice quality assessment.

[2] Partly financed by the grant PACTE QUALITE with the Rhône-Alpes regional government.

References

1 Ambler, Scott W., “The Elements of UML 2.0 Style”, Cambridge University Press, 2005

2 G. Booch, J. Rumbaugh, I. Jacobson: “The Unified Modeling Language User Guide” Addison-Wesley, 1998

3 P. Bottoni, M. Koch, F. Parisi-Presicce and G. Taentzer: “A Visualization of OCL using Collaborations”. UML 2001, LNCS 2185, Springer, pp. 257–271.

4. G. Caplat, J.L. Sourrouille, "MDA: Model Mapping using Formalism Extension", *IEEE Software*, Vol. 22(2), pp.44-51, 2005

5 Farkas, T.; Hein, C.; Ritter, T. : ”Automatic Evaluation of Modeling Rules and Design Guidelines”, proc. of the Workshop “From code centric to Model centric Soft. Eng.”, http://www.esi.es/modelware/c2m/papers.php

6. D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics and All That Stuff", TR MCS00-16, The Weizmann Institute of Science, 2000.

7 Hnatkowska, B., “Verification of Good Design Style of UML Models”, Proc. Int. Conf. Information System Implementation and Modeling, 2007, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-252/paper10.pdf

8 ISO, International Organization for Standardization, "ISO 9126-1:2001, Software engineering – Product quality, Part 1: Quality model", 2001

9 Jacobson I., Booch G., and Rumbaugh J., *The Unified Software Development Process*, Addison-Wesley, 1999.

10 C. Kiesner, G. Taentzer, and J. Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. Technical Report 2002/23, Tech. Univ. of Berlin, 2002

11 L. Kuzniarz, L. Pareto, JL. Sourrouille, M. Staron, "The 3rd Workshop on Quality in Modeling", Models in software engineering, LNCS 5002, Springer, 2008, pp.271-274

12 C.F.J. Lange, B. DuBois, M.R.V. Chaudron, S. Demeyer: “Experimentally investigating the effectiveness and effort of modeling conventions for the UML”, CS-Report 06-14, Tech. Univ. Eindhoven, 2006.

13 Malgouyres, H., Motet, G., “A UML model consistency verification approach based on meta-modelling formalization”. SAC 2006: 1804-1809

14 H. Malgouyres, J.P. Seuma-Vidal, G. Motet: “UML 2.0 Consistency Rules”, V 1.1 (in french) http://www.lesia.insa-toulouse.fr/~motet/UML/CoherenceUML_v1_1_100605.pdf

15 Rational Unified Process, IMB Corp. 1987 (2008).

16 Sourrouille, J.-L., Caplat, G.,"A Pragmatic View about Consistency Checking of UML Model", Work. Consistency Problems in UML-Based Software Dev., 2003, pp.43-50.

17 UML, "OMG Unified Modeling Language", Version 2.1.2, 2007

18 JL Sourrouille, M. Hindawi, L. Morel, R. Aubry, "Specifying consistent subsets of UML", Educators Symposium @ MODELS'08, 2008 (extended version http://www.if.insa-lyon.fr/liesp/~sou/Reports/sUML-RR2008_1.pdf)