Proposal for Improving the UML Abstract Syntax

Dan Chiorean, Vladiela Petraşcu, Ioana Chiorean
Babeş-Bolyai University, Cluj-Napoca, Romania
chiorean@cs.ubbcluj.ro, vladi@cs.ubbcluj.ro, ioana@math.ubbcluj.ro

***Abstract*—Different types of consistency of UML models have been described in the literature. The consistency of UML models with the UML static semantics, usually referred as well-formedness, and the consistency between different versions of the same model are among the most cited. The UML models' well-formedness is a precondition for any other kind of consistency, being described by means of the UML abstract syntax. Unfortunately, this abstract syntax specification is bogus. As a consequence, checking UML models' consistency is not a natural practice, as it should be. Beginning with 2000, there have been several papers reporting this state of facts, but without any visible consequences on the state of practice. In this paper, the authors propose a new approach (including specification techniques and a process) meant to overcome this specification drawback. Our proposal is based on a long practice of improving the specification of UML Well-Formedness Rules in the OCLE tool.**

***Index Terms*—UML model consistency, UML abstract syntax, Well-Formedness Rules**

I. Introduction

The purpose and relevance of the abstract syntax specification of MOF-based metamodels are explicitly mentioned in all OMG documents and acknowledged by the entire community of modelers. As stated in [11], "The abstract syntax defines the set of UML modeling concepts, their attributes and their relationships, as well as the rules for combining these concepts to construct partial or complete UML models." The same document claims that "Relative to UML 1, this revision of UML has been enhanced with significantly more precise definitions of its abstract syntax rules and semantics, a more modular language structure, and a greatly improved capability for modeling large-scale systems." Unfortunately, the state of facts does not live up to these claims. The paper [14] proposed by Wilke and Demuth at the OCL 2011 Workshop is a relevant proof of this, through both its title and contents, even though it has been written before [11]. The errors identified within the abstract syntax specification and the solutions proposed for some of them concern all UML and MOF releases, with no exception (see [1] - [8], [13], [14]). Although there have been enhancements in the abstract syntax specification from one version of the standards to the other, things could and should be seriously improved given the fact that: the reported errors, except compilation errors, have not been eliminated yet, new errors have emerged, the reading and analysis of the UML 2.x specification is much more tedious compared to its 1.x version. Thus, the understanding of a single 2.x well-formedness rule (WFR) generally requires a detailed analysis of several class diagrams and additional operations (AOs). The fact that the standard abstract syntax specifications fail to reach a stable and correct version is symptomatic and triggers the necessity of a change of attitude in writing such specifications. The approach proposed here is driven by the paramount importance of a complete, non-ambiguous informal specification, accompanied by relevant examples and by the need of a thorough validation of all specifications, based on adequate test cases. The technical aspects concerning the OCL specifications reported by previous papers, as well as the need of a testing-oriented OCL specification (meant to support an efficient error detection and diagnosis [5]) are also considered.

Within the abstract syntax specification, the role of the informal (natural language) descriptions is twofold. On the one side, they are used for detailing the structure described by class diagrams, by providing additional information concerning the concepts, attributes and associations involved. On the other, they describe the constraints that should be fulfilled by the modeling concepts, as well as the additional operations used for querying the model or needed in the specification of WFRs. This second role influences both the design of OCL specifications and their validation.

Unlike previous work on the topic, this paper introduces a natural approach concerning the specification of a static semantics. Although the term "approach" may seem a bit pretentious, we claim it is the most adequate, since it refers to a process in which the involvement of the OMG standards' authors is a must. The gaps and ambiguities residing in the standard specifications cannot be fixed in absence of their authors or in the absence of an explicit, unequivocal description, due to the risk of altering the original intentions. Our proposal takes into account the major differences among the 2.x and 1.x versions of the UML standard.

The reading of the UML 2.x specifications is more demanding and tedious as compared to the 1.x versions. This is due to the fact that most of the concepts are progressively described in several packages, their understanding requiring the investigation of various diagrams and associated textual descriptions [2]. The amount of newly-introduced concepts and their dispersed presentation are a strong argument towards the adoption of a complete, rigorous and clear description style, as proposed by this paper.

II. UML 2.x Abstract Syntax - From Goals to State of Facts

The primary purpose of an abstract syntax definition is that of providing a complete, non-ambiguous and rigorous description of a modeling language. These requirements are mandatory for both the correct understanding and consistent

use of the modeling language, and for enabling conformance verifications of user-models against it. Failure to fulfill these requirements triggers inability to fully validate models and the risk to encounter different interpretations of the same specifications. Moreover, it compromises the chance to ensure a safe and predictable model transfer among tools, as required by [11]: "One of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability. However, to enable meaningful exchange of model information between tools, agreement on semantics and notation is required."

Modeling languages share far more commonalities than differences with programming languages. Therefore, similar to programs, compilability is a mandatory requirement on models as well. However, the similarity among the two types of languages does not only involve compilability. Similar to programs, models should be executable, and the results should conform to the requirements [6]. The formal model specifications should be preceded by informal equivalents. The informal description should be complete and non-ambiguous, since, according to Kristen Nygaard "Programming is understanding". Therefore modeling, similar to programming, cannot be imagined in the absence of a thorough problem understanding. Mathematical proofs of specifications' correctness are only seldom realized; testing remains therefore the best alternative, at least in the current context. Similar to program development, ensuring model compilability is mandatory only at some key points of model development (usually, prior to transforming the models or prior to performing simulations). Such a requirement is best captured in [10]: " during model editing, the model will frequently be syntactically incorrect, and the tool needs to be able to allow for syntactical incorrectness in this mode.". Thus, it is easy to understand the reason why the static semantics is described exclusively by means of invariants (WFRs), without pre/post-conditions, as promoted by Design by Contract. Despite this, some of the additional operations encountered in UML have preconditions. We judge this practice as right and useful, since the additional operations are not employed exclusively in the specification of invariants. An AO being targeted to model navigation, the fulfillment of its precondition guarantees that its evaluation is meaningful, while the fulfillment of its postcondition ensures the correctness of the evaluation results. As regarding the result of a model compilability check, this should provide more than a simple yes/no message. In case of non-compilability, it is essential to be provided with meaningful information enabling efficient error diagnosing and allowing a real-time model adjustment.

The informal specifications included in the UML 2.x documents (for both WFRs and AOs) fail to comply with the quality requirements mentioned in the beginning of this section. This is a high-priority issue, in our view. In the last UML specification, 2.5, WFRs are compilable - a step forward compared to the previous specification, 2.4.1. However, the runtime testing and debugging issues are much more tedious than the compilability ones, as unanimously acknowledged by software developers. An important cause of the existing errors is the inappropriate informal specification. We will deal with these issues in Section 5 of this paper.

III. Related Work

Beginning with 2000, several papers have focused on the specification and usage of WFRs. In the following, we summarize the ones considered to be the closest to our approach.

In [13], the authors have given a first quasi-exhaustive analysis of the WFRs specified in UML 1.3. The work has focused on the Foundation::Core package (31 classes and 27 associations) that has been specified in USE, in order to check the corresponding 43 WFRs. Also, 28 Additional Operations were tested. Errors have been found in 39 out of 71 tested expressions. Four categories of errors have been identified: syntax errors, minor inconsistencies, type checking errors, and general problems. The paper was the first to draw an alarm with respect to the quality of the UML WFRs specifications. The following statement worth mentioned: "For future work we plan to extend the analysis to the complete UML metamodel including all of its wellformedness rules and making it available in USE. This might not only be useful for improving the state of the standard but also implies another very nice application: in principle, any UML model can be checked for conformance to the UML standard.". In [2], authors from the same research team present a similar analysis performed with USE, this time for the UML 2.0 Superstructure.

In [8], the second published paper on this topic, the authors claim having tested the entire set of WFRs specified in the context of the UML 1.3 metamodel. They report 450 errors of three kinds: non-accessible elements, empty names, and miscellanea. The proposed solutions for fixing the reported problems seem a bit bizarre. Namely, they suggest to "Take the empty names into account in every rule of the metamodel (296 errors). Consider access and contents as two different concepts (138 errors). Avoid two opposite association ends with the same name (18 errors)".

In [3], the authors present two techniques for checking UML models. One, implemented in Rational Rose, that enables to navigate and check the contents of the UML metamodel by means of an appropriate VBA specification, and the other by means of OCL AOs and WFRs. Some AOs and WFRs are analyzed both with respect to the identified bugs to the actions undertaken for correcting them. In [4], the same authors analyze different kinds of errors identified in the OCL specification. The focus is on proposing "good practices" meant to support "a correct, clear and efficient specification". The consistency among the formal and informal specifications, the clearness of OCL expressions, the fact that evaluating OCL specifications instead of only compiling them is imperative, are among the proposed and exemplified practices. The paper [5] is focused on describing OCL specification patterns intended to support a specification style targeted at an easier model debugging. In [6], the focus is on the similarities between programming and modeling languages. The paper emphasizes

the fact that, in the context of the model-driven paradigms, producing compilable models is a must, not an option.

[14] is focused on the study of UML 2.3 Superstructure WFRs. As acknowledged by its authors, there are many similarities between the topic and results reported in this paper and those of [2]. The differences concerns the metamodels (UML 2.3 in this last analysed paper and UML 2.0 in the previous case) and the tools employed (Dresden OCL toolkit in the last paper and USE in the other).

The common feature of papers published by the teams from Bremen and Dresden is their focus on the compilation phase. As regarding the papers published by our team, the analysis overpasses the mere compilation. The runtime results and their conformance to the informal specifications are also considered.

In [9], the authors present coherence rules grouped on metamodel elements and diagrams. Although the idea looks nice, there are some drawbacks. Firstly, the rules are presented exclusively in an informal manner (in spoken language); moreover, for some rules the semantics is not clear enough. Secondly, there are no comments about incorrect rules and about authors' proposal for improving the existent semantics.

Finally, in [7] and in some other papers on the same topic, A. Egyed presents "an approach for quickly, correctly, and automatically deciding when to evaluate consistency rules." As the title of the paper suggests, the author's work is focused on doing an automated quick evaluation. In the experience presented, only 24 rule were evaluated - some being WFRs and others defined by model designers. There are no mentions about the correctness of the evaluated rules. From this point of view, the approach is significantly different from ours. However, the author is convinced about the importance of consistency checking in case of UML models.

IV. The Proposed Approach for Specifying the Abstract Syntax

The state of facts in specifying the abstract syntax of UML, together with a thorough analysis of the published literature on the topic allow us to argue that a significant amount of all the existing specification errors are due to failure in obeying to a number of elementary requirements, validated by the software engineering practice. Given our experience in the field, we propose conforming to the following rules when specifying the abstract syntax of UML/MOF.

1) A complete and non-ambiguous informal equivalent of all OCL specifications (both WFRs and AOs) is the first and the most important of these rules. Moreover, there should exist a full conformance among the informal specification and its formal correspondent. It would be helpful if the informal specification would be accompanied (possibly in an attached document) by examples illustrating cases of validation and invalidation of each rule, as well as exceptional cases that may arise throughout the evaluation of AOs.
2) Runtime validation of formal specifications on significant data sets (models) is mandatory. Mere compilability is not enough.
3) The formal specifications of WFRs must be testing-oriented [5]. Accomplishment of this requirement supports an easy error diagnosing of models that do not comply with the WFRs in question.
4) Choosing the appropriate context for the specification of WFRs that refer to features of several metaclasses is also an important issue [6].
5) For both efficiency and clarity, the use of OCL specification patterns is recommended, whenever the case.
6) Indented and syntax highlighted OCL expressions enable an easier lecture of specifications. Prefixing the formal specifications with a short informal description of requirements (similar to comments in programming) is useful, as well.
7) The WFRs which are specified in an informal manner exclusively should be complemented by relevant examples of their fulfillment or failure to be fulfilled, possibly accompanied by an overview of how the authors imagine the validation process. Obeying to this requirement allows a better understanding of the rules and provides support in finding appropriate specification and validation solutions.

Except for the forth recommendation, which seems to be obeyed by almost all specifications of the UML, all the others are not met. As illustrated in the following, taking them into account will help in increasing the quality of the standard specifications.

V. Analyzing Import Relationships on UML Namespaces

The import in a namespace of elements from different namespaces is one of the most important relationships in both programming and modeling languages offering modular development support. This allows the imported elements to be directly referred by their name or by an alias, whenever there is no name conflict among the imported elements and the elements belonging to the namespace which performs the import. In case of conflict, the use of a qualified name is mandatory. The programming languages come with clear specifications concerning the import relationship. Thus, we argue that any differences in the import rules specified for UML and MOF compared to the programming languages should be clearly justified. Even more, examples are needed to illustrate the manner in which various cases tolerated by modeling languages can be coded in a programming language (direct engineering) or the reverse (reverse engineering). Otherwise, the support offered by modeling languages to the MDA, MDE and MDD paradigms remains only in statements.

Similar to programming languages, the UML allows two types of import:

1) *explicit*, by defining an individual import relationship for each imported element (which should be a `PackageableElement`). Such an import relationship is modeled by the `ElementImport` concept (direct descendent of `DirectedRelationship`), that has two attributes: `visibility` and `alias`. Graphically,

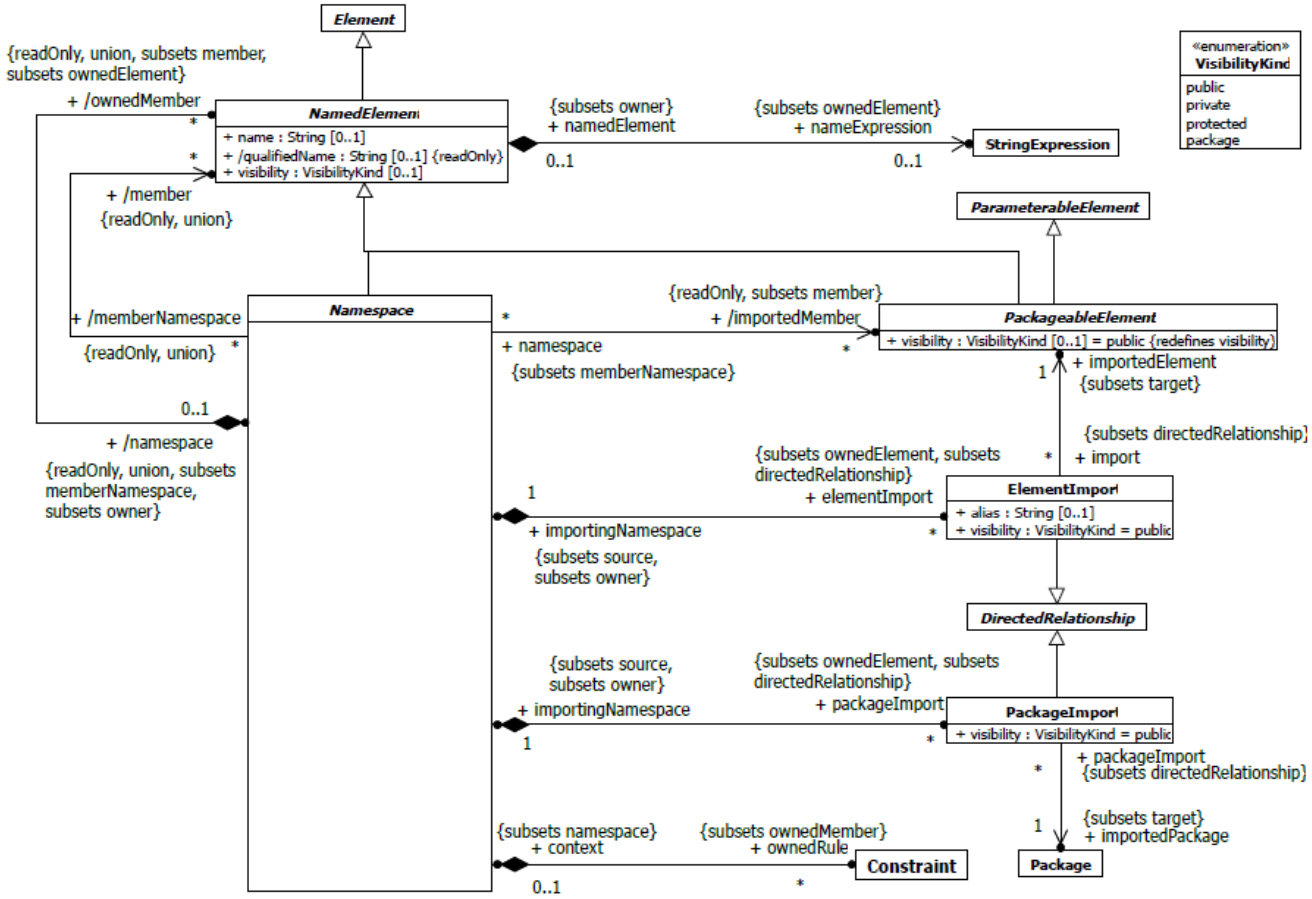

Fig. 1. The `Namespaces` diagram of the `Constructs` package (Figure 7.5 of [11])

`ElementImport` is represented by a "a dashed arrow with an open arrowhead from the importing namespace to the imported element. The keyword `import` is shown near the dashed arrow if the visibility is public, otherwise the keyword `access` is shown to indicate private visibility. If an element import has an alias, this is used in lieu of the name of the imported element. The aliased name may be shown after or below the keyword `import`."

2) *implicit*, by means of an import relationship among the importing namespace and the imported package. "Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import." Similar to the case of an element import, this relationship is modeled by the `PackageImport` metaclass, having an analogous graphical representation. It "is shown using a dashed arrow with an open arrowhead from the importing package to the imported package. A keyword is shown near the dashed arrow to identify which kind of package import that is intended. The predefined keywords are `import` for a public package import, and `access` for a private package import."

The concepts involved in the import relationships and their interconnections are illustrated in Figure 1. Even if not explicitly stated, the set `importedMember` is needed when computing the set of potential servers of a model element or when checking if an element is legally imported.

In the following, we will analyze the formal OCL specifications (and their informal descriptions) regarding the import relationship defined between a `Namespace` and a `PackageableElement` or a `Package`.

In the `Namespace` context, the AO `getNamesOfMember()` previously specified in the `NamedElement` context "is overridden to take account of importing. It gives back the set of names that an element would have in an importing namespace, either because it is owned; or if not owned, then imported individually; or if not individually, then from a package."

In our opinion, the last part of the second phrase (marked by the underlined words) is a bit confusing. A clearer and more explicit statement (at least for a non-native speaker) could be: either because it is owned, or imported individually or by a package import.

Following, there is the corresponding OCL specification, as

provided in the standard.

```
Namespace::getNamesOfMember(element:NamedElement):String[0..*];
getNamesOfMember =
 if self.ownedMember->includes(element)
 then Set{element.name}
 else let elementImports:Set(ElementImport)=self.elementImport
        ->select(ei|ei.importedElement=element) in
      if elementImports->notEmpty()
      then elementImports->collect(el|el.getName())
      else self.packageImport->select(pi|
       pi.importedPackage.visibleMembers()->includes(element))
       ->collect(pi|pi.importedPackage.getNamesOfMember(
                                              element))->asSet
      endif
 endif
```

In [11], the type returned by the observer defined is written as `String[0..*]`, notation not accepted in OCL 2.4 (the current specification [12]) and previous specifications. The AO `visibleMembers()`, used by composition in the above specification is bogus, as we will prove in the following. As a consequence, the result returned by the AO `getNamesOfMember()` will be incorrect in some cases.

In the `Package` context, the query `visibleMembers()` identifies those members of a `Package` that can be accessed outside it. The specification provided for the AO visibleMembers() in [11] is:

```
body: member->select(m|m.oclIsKindOf(PackageableElement)
      and self.makesVisible(m))->
          collect(oclAsType(PackageableElement))->asSet()
```

Analysing figure 12.1 Packages, pp. 29 from [11] we wonder why the OMG has not proposed: `self.packagedElement->select(pe | self.makesVisible(pe))`. In this case, the condition `oclIsKindOf(PackageableElement)`, the cast at `PackageableElement`, and the conversion `asSet` are redundant.

The query `makesVisible()` specified itself in the `Package` context, "defines whether a `Package` makes an element visible outside itself. Elements with no visibility and elements with public visibility are made visible.".

```
Package::makesVisible(el:Namespaces::NamedElement):Boolean;
pre: self.member->includes(el)
makesVisible =
  -- the element is in the package
  (ownedMember->includes(el)) or
  -- it is imported individually with public visibility
  (elementImport->select(ei|ei.visibility =
  VisibilityKind::public)->collect(importedElement.oclAsType(
        NamedElement))->includes(el)) or
  -- it is imported through a package with public visibility
  (packageImport->select(pi|pi.visibility =
  VisibilityKind::public)->collect(pi|
  pi.importedPackage.member->includes(el))->notEmpty())
```

As regarding this specification, there are some things we would like to analyze.

Firstly, the informal specification states that "Elements with **no visibility** and elements with **public visibility** are made visible." As no explanation is offered regarding why elements with no visibility are visible outside the package, this requirement seems strange to us. Especially since in programming languages (Java, for instance) only public members of a package can be explicitly exported or referred by their qualified name outside the owner package. That is why, at the beginning of this section, we have emphasized the necessity of an explicit description of the rationale behind certain decisions.

Secondly, it is easy to notice that the formal specification does not comply with the informal one, since only the visibilities of `elementImport` and `packageImport` are considered, without taking into account the visibility of the element itself (irrespective if it being owned by the package, imported individually or by means of a package import). Moreover, the authors of the OMG specification say nothing with respect to what happens in particular cases, such as the one in which the same element is imported both individually (with `visibility = VisibilityKind::private`) and by means of a `packageImport` with `visibility = VisibilityKind::public`. In this case, evaluating the specification above, `makesVisible` will be evaluated to true, even if it is stated that the individual import has priority compared to package import.

In order to exemplify our reasoning, let us consider the model shown in Figure 2. As illustrated there, between packages `P2` and `P1` there is an `<<import>>` relationship. The package `P1` owns the class `A` having `visibility = VisibilityKind::private`. In the context of the `P1` package, we are interested to see if `el = A` is visible outside its owning package. The class `A` is a member of `P1`, so we have to evaluate the `makesVisible()` AO. `P1.ownedMember->includes(A)`, therefore `makesVisible() = true`. Thus, due to the `packageImport` relationship, `A` is added to the `P2` namespace and can be accessed by name, in case there are no name collisions between `A` and other elements of the `P2` namespace. This result is incorrect, since the element in question has private visibility.

Thirdly, when the element is imported in the package by `packageImport`, the result returned will be wrong if the visibility of the element transmitted by parameter (`el`) is `VisibilityKind::private`. A sample situation is illustrated in Figure 3. In the package `P2`, the private class `A` is imported by the `importPackage` relationship stereotyped `<<import>>`. In `P3`, private class `A` is imported by means of the `importPackage` relationship, also stereotyped `<<import>>`, between `P3` and `P2`. In the OCL specification of `Package::makesVisible` (see above), this corresponds to the OCL expression following the second **or** (imported through a package with public visibility). Similar to the previous case, the result is wrong, due to the visibility of class `A`.

Concluding, we notice that even though the `makesVisible()` AO is compilable, its returned results do not fully comply with the informal specification. Even more, the usefulness of **elements with no visibility** has not been explained and taken into account in the formal specification. Apart of these, there could be particular cases, like those mentioned above, when the results are debatable.

A possible solution would be to include in the precondition the restriction regarding the visibility of `el`.

Fig. 2. Import of a private class through a single package import

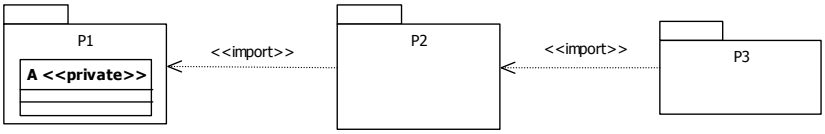

Fig. 3. Import of a private class through double package import

```
Package::makesVisible(el:Namespaces::NamedElement):Boolean;
pre: self.member->includes(el) and
     el.visibility = VisibilityKind::public
```

or

```
Package::makesVisible(el:Namespaces::NamedElement):Boolean;
pre: self.member->includes(el) and
     (el.visibility=VisibilityKind::public or
      el.visibility.oclIsUndefined
     )
```

Since we have no idea about the semantics of `el.visibility.oclIsUndefined` in this case, and due to other particular cases, our opinion is that the first thing to do is to clarify the informal specification.

In a namespace, a `NamedElement` is valid if it is distinguishable from any other element owned by the namespace [11], [6]. The WFR checking this requirement uses the AO `isDistinguishableFrom(p1,p2)`. This operation is firstly defined within the `NamedElement` context, and redefined in the `BehavioralFeature` context. As stated in the [11] (pp. 73), "...By default, two named elements are distinguishable if (a) they have unrelated types or (b) they have related types but different names."

```
context NamedElement::isDistinguishableFrom(n:NamedElement,
                                            ns:Namespace):Boolean
def: isDistinguishableFrom(n:NamedElement,
                           ns:Namespace):Boolean =
 if self.oclIsKindOf(n.oclType) or
    n.oclIsKindOf(self.oclType)
 then ns.getNamesOfMember(self)->intersection(
      ns.getNamesOfMember(n))->isEmpty()
 else true
 endif
```

The formal specification fully complies with the informal requirements. However, stating that two elements having unrelated types are distinguishable could cause unpleasant situations, such as the one in which a package contains both a class and an enumeration having the same name (e.g `Test`). In this case, `enumeration.oclIsKindOf(class.oclType)` and `class.oclIsKindOf(enumeration.oclType)` are always evaluated to `false`, irrespective of the `enumeration` instance of the metaclass `Enumeration` and `class` instance of the metaclass `Class` (see Figures 4 and 5). By consequence, the type of an attribute `testKind:Test` belonging to a different class of the same package is uncertain. This is due to the fact that we cannot distinguish between these two types having the same name. In order to fix this bug, the simplest solution would be to remove "part a)" from the above requirements. However, the best decision would be to provide a clearer specification, including suggestive examples of models with both distinguishable and not distinguishable named elements.

The `visibleMembers()` AO analyzed at the beginning of this section is used by composition in computing the set of elements imported in a namespace.

"The `importedMember` property is derived from the `ElementImport`s and the `PackageImport`s."

```
importedMember = self.elementImport.importedElement->asSet()
  ->union(self.packageImport.importedPackage->collect(p |
  p.visibleMembers())->asSet())
```

Here, the drawback is due to previously-discussed `visibleMembers()`. Another problem is that both in case of a direct `elementImport` and `packageImport`, we have to consider also the elements having `visibility = VisibilityKind::private` and marked with the stereotype `<<access>>`, not only those marked with `<<import>>`. That is why, the importedMember in the `P3` namespace of Figure 6, will return `Set{A, C, B, D, E}`, so the `visibleMembers()` has a negative influence by means of `<<access>>` `packageImport` also.

Since in case of name-clashes the imported elements can be referred only by means of their `qualifiedName`, let us take a short look at the derived attribute `qualifiedName`, specified in the `NamedElement` metaclass. In the standard it is stated that this attribute "is constructed from the names of the containing namespaces starting at the root of the hierarchy and ending with the name of the `NamedElement` itself." The specification of the `qualifiedName():String` AO is:

```
body:
 if self.name<>null and self.allNamespaces()->select(ns|
                                  ns.name=null)->isEmpty()
     then self.allNamespaces()->iterate(ns:Namespace;
          agg:String = self.name |
          ns.name.concat(self.separator()).concat(agg))
     else null
 endif
```

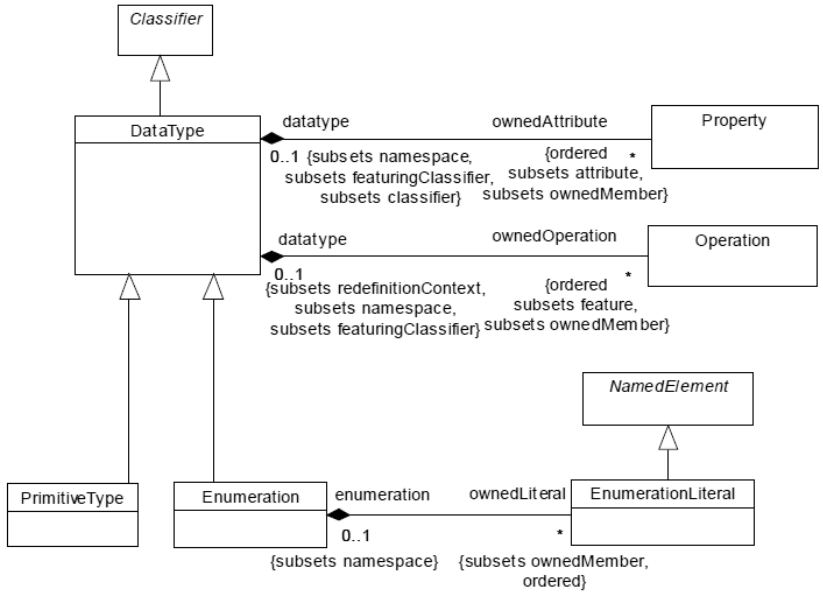

Fig. 4. The classes defined in the DataTypes diagram - from [11], Figure 11.18

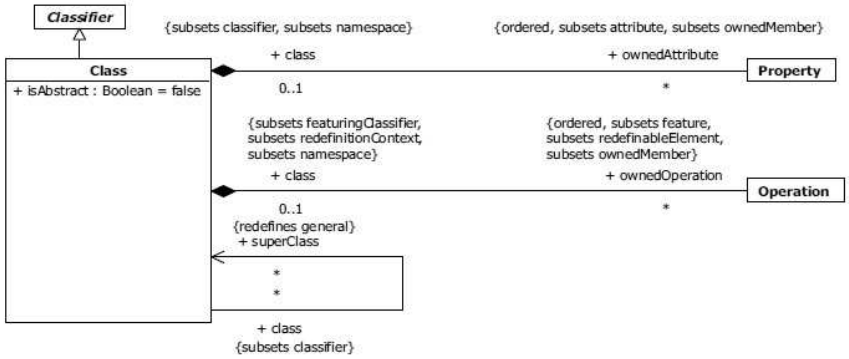

Fig. 5. The Classes diagram of the Constructs package - from [11], Figure 11.15

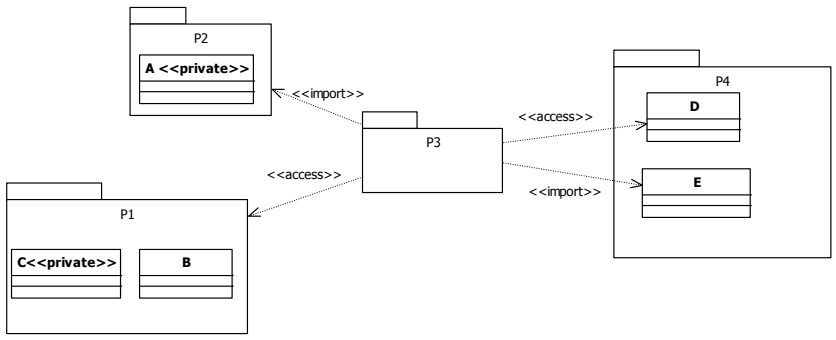

Fig. 6. Package import

While this specification is compilable and roughly correct, there can arise a little problem, because an empty `String` is not forbidden as a name value, and `self.''->notEmpty() = true`, the value computed for `qualifiedName` in such cases, is meaningless. That is why, such cases must be forbidden. More general, we consider that rules similar to those applied in programming languages must be used in modeling language as well.

The aspects analyzed in this section prove that, even in simple cases, the specifications must be realized carefully, not in a superficial manner. The reccomendations made in the previous section must be taken into acount.

VI. Conclusions

The purpose of this paper has been to propose a change of attitude with respect to the definition of the UML's abstract syntax, expected to positively affect the quality of the standard specifications. This improvement is a "sine qua non" condition for attaining the target of model-driven technologies and paradigms.

Our proposal is argued by means of meaningful examples taken from the latest UML specification [11]. The first requirement to be accomplished concerns the quality of the informal specifications: they have to be complete, accurate and clear. Once this precondition is accomplished, the associated postcondition is that the formal OCL specifications must fully conform to their informal equivalents. Our experience has proved that this conformance is achievable through an iterative process. The results obtained by evaluating the formal specifications must be compared to the informal ones and should trigger a synchronization among the two, if needed. This is an important contribution through which the OCL specifications may increase the quality of abstract syntax definitions, in general.

Another important message is that the mere compilability of formal specifications does not value much if these specifications are not validated on comprehensive models. Technical aspects related to the particularities of the specification language and the support that the formal specification style brings in achieving compilable models are important as well. In this respect, we recommend the adoption of a testing-oriented specification style, as introduced in [5].

Apart from the advice related to the specification style, all the others have been validated in Software Engineering. That is why, noticing that so much good specification practice has not been considered comes as an unpleasant surprise.

Achieving a good specification of MOF-based languages is a tedious process, requiring a quality feedback both from scientists and users. Our intent has been to make a first step, by proposing a set of "good practices" to be considered in the process, as well as a number of examples supporting our proposal. Hoping that our proposals will be analyzed, improved and extended by the OMG and thus a better abstract syntax specification will support a more efficient and widespread usage of modeling languages.

References

[1] Bauerdick, H., Gogolla, M., Gutsche, F. - Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. - In Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J., eds.: UML 2004 - The Unified Modelling Language. Volume 3273 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2004) pp. 188-196

[2] Fabian Bttner and Martin Gogolla - On Generalization and Overriding in UML 2.0 - in UML'2004 Modeling Languages and Applications. UML'2004 Satellite Activities, Springer 2004

[3] Chiorean, D., Carcu, A., Pasca, M., Botiza, C., Chiorean, H., Moldovan, S. - UML Model Checking in Studia Informatica vol XLVII (2002) pp. 71-88

[4] D. Chiorean, A Carcu, C Botiza, etc. Ensuring UML models consistency using the OCL environment - Electronic Notes in Theoretical Computer Science - ENTCS/102, 2004, pag. 99-110, http://dx.doi.org/10.1016/j.entcs.2003.09.005

[5] D. Chiorean, V. Petrascu, I. Ober. Testing-Oriented Improvements of OCL Specification Patterns. In Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics - AQTR. Volume II, pp. 143-148. IEEE Computer Society, 2010

[6] D. Chiorean, V. Petrascu. Towards a Conceptual Framework Supporting Model Compilability. In Proceedings of the Workshop on OCL and Textual Modelling (OCL 2010). Volume 36(2010), ECEASST

[7] Alexander Egyed, Automatically Detecting and Tracking Inconsistencies in Software Design Models, In IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 188-204, 2011.

[8] J. M. Fuentes, V. Quintana, J. Llorens, G. Genova, R. Prieto Diaz. Errors in the UML metamodel? ACM SIGSOFT Software Engineering Notes 28(6):3-3, 2003.

[9] Hugues Malgouyres, Jean-Pierre Seuma-Vidal, Gilles Motet, Regles de coherence UML 2.0 - Version 1.1 - INSA - Toulouse, online at: http://www.lesia.insa-toulouse.fr/UML/CoherenceUML_v1_1_100605.pdf

[10] Michael Moors - Consistency Checking; Rose Architect, Spring Issue, April 2000, http://www.therationaledge.com/rosearchitect/mag/index.html

[11] Object Management Group (OMG) - OMG Unified Modeling Language (OMG UML) Version 2.5, 2015, http://www.omg.org/spec/UML/2.5/PDF

[12] Object Management Group (OMG) - Object Constraint Language version 2.4 - formal/2014-02-03, http://www.omg.org/spec/OCL/2.4

[13] M. Richters, M. Gogolla. Validating UML models and OCL constraints. In Evans et al. (eds.), UML 2000 The Unified Modeling Language. Advancing the Standard: Third International Conference Proceedings. Lecture Notes in Computer Science 1939, pp. 265-277. Springer, 2000.

[14] Claas Wilke and Birgit Demuth - UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure - paper proposed at: OCL 2011 Workshop