An Overview of UML Consistency Management

By

M. Elaasar, L. Briand

Technical Report SCE-04-18

Department of Systems and Computer Engineering
1125 Colonel-By Drive, Ottawa, Ontatio, K1S 5B6 Canada

August 24, 2004

Table of Contents

List of figures

Abstract

The ambiguity inherent in UML coupled with its support of multiple viewpoint modeling pose a great risk of inconsistency. Many classifications of UML inconsistencies exist in the literature today. Several proposals are also made for the mitigation of that risk. The essence of these proposals is to clear the existing ambiguity and to seek the formalization of UML. Unfortunately most of these proposals are not implemented in today's UML CASE tools. For these tools to fulfill their promises of supporting automation and hiding complexity, they need to employ a consistency management framework with certain characteristics. This paper surveys the state of the art in UML consistency management and proposes a research agenda for the implementation of a successful consistency management framework.

1 Introduction

The Unified Modeling Language (UML) has become the de-facto notation for software design and modeling. The main strength of UML lies in its ability to express many facets of design, ranging from structural (ex: class diagram) to behavioral (ex: interaction and state machine diagrams), using a single integrated formalism. Like any other language, UML comes with a set of syntactic and semantic rules that derives the understanding and interpretation of models written in this language. Some of the rules that are deemed necessary are formally expressed in the specification [1, 2] to assert the well-formedness of models. Others, mainly semantic rules, are informally defined in the prose of the specification to give more flexibility and expressive power to designs at different levels of abstraction, by different modeling methodologies or for different application domains

Unfortunately, the power of such a generic multi-view formalism comes along with an unavoidable risk, namely inconsistency. The notion of consistency has been investigated in different domains and at various levels in the literature. The Webster's dictionary definition of inconsistency is: "the relation between propositions that cannot be true at the same time, or the lack of harmonious uniformity among parts". To put this definition into perspective, there has to be an understanding of what has to be consistent in the context of UML. Another point is the level of consistency that has to be enforced during every development cycle before breaching the practical limits or restraining the creative process. Additionally, there has to be an evaluation of the different mechanisms for

detecting, preserving and restoring consistency. Finally, a general framework for analyzing consistency and how it integrates with today's UML modeling tools has to be studied.

Talking about consistency begs a very important question: why do we need to check consistency in UML models? The first motivation for model consistency is correctness. Usually, consistency problems reveal design problems or misuse of UML. When those problems are discovered early in the design process, it is easier and more cost effective to fix than if they were discovered at a later stage. The second motivation is implementability, which usually involves translating a UML model into a programming language, a usually precise and unambiguous notation. These two motivations come into play during a typical iterative development process, where a model is built incrementally starting with requirements and ending into code. The process usually involves several viewpoints and a number of contributors with different skills. Without consistency analysis, it would be hard to evolve the model and ensure that the collaborative effort is coherent.

The notion of consistency, or lack of, has its roots in formal methods [3, 4]. To assert that something is consistent, you have to declare what it is consistent with. Like any other language, UML has its own unique syntax and semantics. The syntactic correctness or well-formedness of a UML model is usually a prerequisite to any further consistency analysis. Syntax is what makes the model readable and hence verifiable. For example, a classifier having a unique fully qualified name within a model is a syntactic requirement.

Failing to maintain the well-formedness of a model often leads to ambiguity. Another source for ambiguity is the existence of incomplete semantics. While UML itself is not a formal language, often enough UML models need to be translated to a more formal notation, typically source code. Usually a UML model goes through a series of refinement transformations before finally getting translated into code, an inherently formal language. While syntax helps readability, semantics is what gives meaning to language constructs. For example, a classifier at the source of a generalization relationship with another classifier inherits all the target classifier's structure and behavior. However, some semantics of UML are unspecified, like how to inherit attributes with the same name in the case of multiple-inheritance, which opens the door for multiple interpretations [10]. While consistency at the semantic level is generally a desired property to ensure the integrity of a UML model, it is mostly needed when transforming a model into a formal notation [44].

The model syntactic and semantic correctness represents the basic level of UML consistency. They can be further augmented by the application of UML profiles, a standard extension mechanism for UML. Profiles include families of stereotypes that enrich the semantics of UML for a given domain. For example, the UML real-time profile [45] contains the stereotype "capsule". Applying that stereotype to a class means that the class is active, i.e. it has its own thread of control. It also means that the class can communicate with other capsules exclusively through its public ports, a set of properties realizing certain protocol roles. One way to assert consistency here is to check that the

messages exchanged between capsules via their ports in structure diagrams belong to the same protocols realized by the ports

In fact, UML consistency analysis goes beyond checking the language's own syntax and semantics. It is usually customized to encompass other related domains like the used modeling methodology, the targeted programming language, the modeled system…etc. [16]. Most languages, including UML, have sets of common expressions and best practices that guide the usage of the language through various development cycles. These concepts are usually artifacts of modeling methodologies that strive to put order into the complex domain of modeling. These methodologies usually impose more restrictions to constrain the use of the language and to reduce the inherent ambiguity. Other domains targeted by a UML model could be its modeling domain, application domain and implementation domain. Since UML is a universal modeling language, not all legal expressions of UML make sense for all these domains. For example, the notion of multiple- inheritance of classes is not feasible in the Java programming language, and therefore the use of this notion should be restricted in UML models that are to be translated into Java code. Another example is the restrictions imposed by some design patterns like the model-view-controller (MVC) pattern [35]. In that pattern, model classes should not depend explicitly or implicitly on controller classes. It is therefore imperative to further constrain the use of UML for these domains to help modelers stay on track.

More UML consistency classifications are offered in the literature. According to one classification [3, 4, 17], consistency is either intra-model (also called horizontal), which

is a property of a model asserting its syntactic and semantic conformance, or inter-model which is between different models related together by one or more transformation relationships. In general, most of the work in this area tends to focus on ensuring that these transformations are consistency-preserving [20]. Another consistency classification distinguished between static and dynamic constraints [10]. A static constraint is one that can be verified statically without running the model, while a dynamic constraint cannot be verified until runtime. More classifications are discussed in section 2.

Defining inconsistency is one task; detecting it is another. The keyword to look for in this area is formalization [3, 4]. The research in UML consistency assertion or inconsistency detection falls within three main categories. The first category tries to complete the UML meta-model to allow for easier accessibility from a model element to all its associated elements. As mentioned earlier, a lot of the semantics of UML is either missing or only informally expressed in the prose of the specification. The second category tries to enhance the language used for expressing constraints on the meta-model. The Object Constraint Language (OCL) is the native language for expressing constraints in UML. Research in this category either suggests enhancing the expressive power of OCL to allow for better expression of constraints, or suggest new languages or notations for constraints all together. The third and widest category of proposed solutions [4] accepts that UML is inherently ambiguous, probably for good reasons, and proposes that consistency analysis be only performed on UML models after translating them into a more formal notation that naturally supports this kind of analysis. Obviously, the process of translating a UML model into a formal notation involves first clearing the ambiguity

by agreeing on the interpretation of the UML expressions according to some semantic domain. What is most important about this approach though is the availability of tools that perform consistency analysis on models expressed in these more formal notations.

Unfortunately, most of today's UML modeling tools do not incorporate good consistency management frameworks (CMF). Building a good CMF is not a trivial task [44]. The framework has to be flexible, to allow for different configurations, and extensible, to allow for new constraints to be added. It should also support a batch checking mode in addition to an incremental on-demand mode. The framework has to be user-friendly in its presentation and allow for easy expression of constraints. Most importantly, the framework has to be efficient, which means among other things scalable to the size of the model, fast in completing the analysis, and accurate in reporting results. It is also desirable if the framework can offer consistency correction actions and design assist tips.

This report surveys the state-of-the-art in the area of UML consistency management. Section 2 tries to answer the questions: what is meant by consistency? And what has to be consistent in UML? Section 3 examines the different propositions in the literature to detect and resolve inconsistencies in UML. Section 4 outlines the desired characteristics of a perfect UML consistency management framework. The last section concludes and gives future directions.

2 UML Consistency Classification

UML is an amalgamation of several historic modeling languages and methods (Booch, OMT, OOSE…etc) [1]. The main objective of UML is to unify the semantics and notation for object-oriented modeling. However, the language has been kept intentionally informal to allow for different interpretations based on the target domains. UML also allows for the modeling of different intersecting viewpoints ranging from structural to behavioral. This inherently ambiguous and multi-faceted nature of UML contributes to its popularity and strength. Unfortunately, it also introduces a significant risk: that of inconsistency. Many proposals in the literature give different classifications of inconsistencies in UML. In this section, all the main classifications will be outlined. The following table summarizes these different classifications:

Classification	Features
Syntactic vs. Semantic	Consistency rules that can be expressed by a formal language are syntactic, otherwise they are semantic
Static vs. Dynamic	Consistency rules that can be verified without executing a model are static, otherwise they are dynamic
Intra-Model vs. Inter-Model	Consistency rules within the same model are intra-model. Those that span models are inter-model
Multi-Level	Consistency rules are grouped according to the semantic domain they target (specifications, profiles, modeling processes, modeled domain …etc)
Nature of Error	Consistency rules are grouped based on the nature of the error (contradiction, incompleteness, ambiguity…etc)

Table 1: UML Consistency Classifications

2.1 Syntactic vs. Semantic Consistency

The published work in UML consistency tends to go in different directions depending on the used definition of consistency. However, all of them agree that consistency generally

entails lack of contradiction and conformance to expectations. The language specifications [1] introduce two initial levels of consistency, the meta-model and the well-formedness rules. The meta-model is a schema that precisely defines the constructs and rules needed for creating models. For example, an association can have two or more association ends. Figure 1 depicts a subset (the class diagram) of the UML meta-model.

A model is inconsistent if it does not conform to the meta-model. However, it may not be consistent even if it conforms to the meta-model. This is mainly due to the limited expressiveness of the UML meta-language, a formalism used to define the UML meta-model. In an effort to complement the meta-language, the OMG proposed OCL [2], a higher order logic language for denoting well-formedness constraints in UML. OCL, a pure expression language, has constructs to inspect and navigate objects and their structure and return a true or false value but it does not change the model. Well-formedness, as expressed by OCL constraints, is usually a prerequisite to any further consistency analysis in UML. The following are some examples of well-formedness rules as expressed in OCL [1]:

- An element may not directly or indirectly own itself:

  not self.allOwnedElements()->includes(self)

- Elements that must be owned must have an owner

  self.mustBeOwned() **implies** owner->notEmpty()

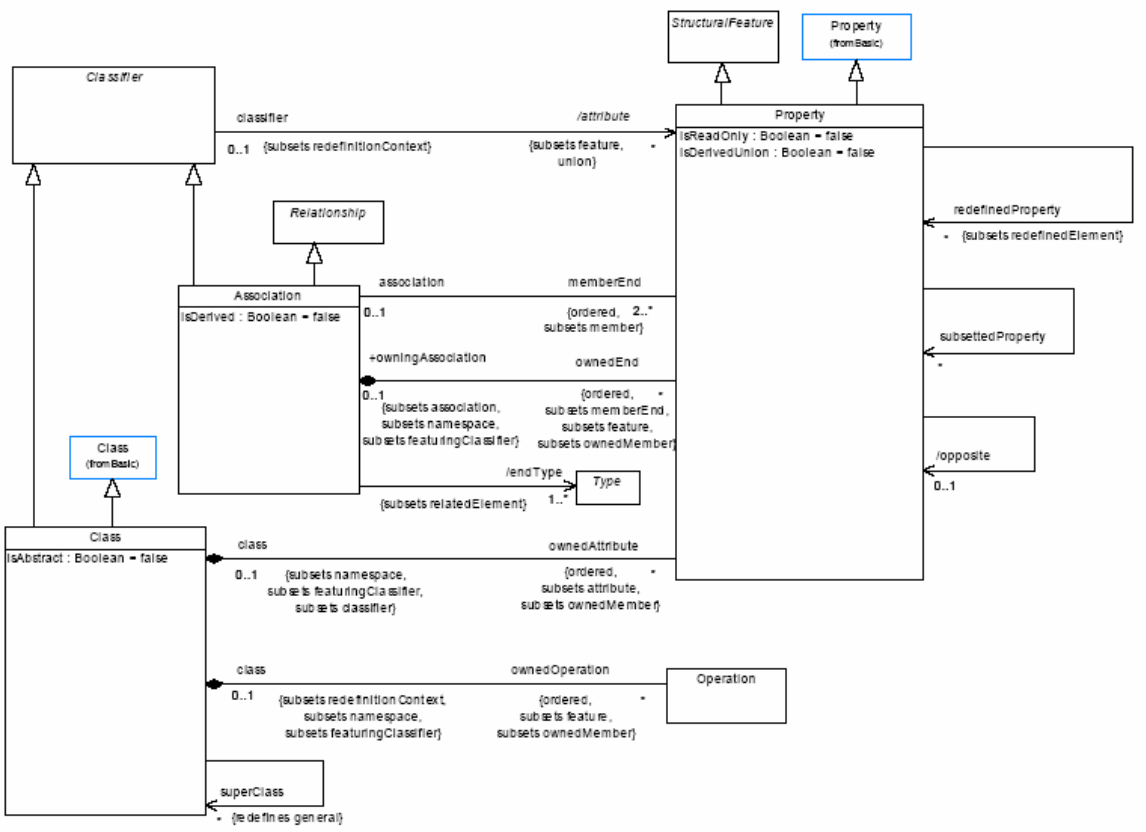

Figure 1 The class diagram subset of the UML meta-model [1]

Unlike formal languages, some of the UML semantics is ambiguously defined which opens the door for different interpretations. For example, there is no agreement on the proper way to inherit attributes with the same name in multiple-inheritance. Another example is the lack of specification of the strategy for dequeuing events processed by a state machine [10]. According to [18], ambiguity is a double edged sword. On one hand, it gives the modeler the flexibility to express the design at a higher level of abstraction without prematurely committing to details. On the other hand, it complicates consistency analysis, which usually requires the semantics to be completely and precisely specified. This begs the question: can UML be formalized? Several attempts have been made to formalize UML, most notably by the precise UML group [29]. The general approach is to

draw limits on the legitimate interpretations of the UML semantics. Once the semantics are formalized, further consistency analysis can proceed by tailoring rules to the selected interpretations.

2.2 Static vs. Dynamic Consistency

Most languages distinguish between their static and dynamic semantics. In [10], the authors explain that UML is no exception to that. The static semantics, or the syntax, of UML is formally described in terms of its meta-model and OCL constraints, in addition to some descriptions in natural language like "a Constraint attached to a stereotype must not conflict with Constraints on any inherited Stereotype, or associated with the baseClass" [16]. While syntax can usually be checked by a static inspection of a model, dynamic semantics cannot be completely verified until runtime. For example, it may not be possible to statically check that a precondition to an operation is satisfied before the operation is called in an interaction diagram. The problem here is that UML is not an executable language, and therefore the dynamic constraints have to be embedded into an executable formalism (like code) that is translatable from UML.

2.3 Intra-Model vs. Inter-Model Consistency

One recurring classification of UML consistency in the literature [3, 4, 17] distinguishes between intra-model and inter-model consistency or between horizontal and vertical consistency. Intra-model or horizontal consistency is a property of a model. It indicates that all the elements of a model are syntactically and semantically correct. As mentioned earlier, UML offers multiple viewpoints for modeling the same system. They range from

structural (ex: class and instance diagrams) to behavioral (ex: interaction and state machine diagrams). These viewpoints or perspectives are usually inter-dependent. For example, a synchronous message in a sequence diagram has to match an operation in a class diagram. This intersection of viewpoints leads to a very interesting class of inconsistencies that is not formalized as part of the specifications. Numerous research attempts [5, 6, 7, 27, 28] have tried to formally define constraints to check for these inconsistencies. While some of these constraints are obvious, others are mainly heuristic in nature. The following is a sample list of constraints between the class and the object diagrams [27]:

- The number of values for an attribute of a given object violates the multiplicity lower bound for that attribute as defined by the object's classifier.
  self.value->size() >= self.definingFeature.lowerBound()

- The attribute's value in an object does not conform to the corresponding attribute type as defined by the object's classifier.
  if self.definingFeature.type->isEmpty()
  then true
  else
  self.value->forAll(v:ValueSpecification |
  v.type.conformsTo(self.definingFeature.type))
  endif
  endif

- The object is not classified (heuristic since it is allowed by the specifications)
  self.classifier.size() > 0

On the other hand, inter-model consistency is a relationship between models [10]. These models are usually related to each other by some sort of a transformation relationship. A transformation describes the application of some procedures to one model to create a new model [20]. The new model can be another representation of the same information in the old model, or a modified version of the original model. In fact, the relationship between transformation and consistency has been elaborated further by the researchers at the

University of Paderborn [20]. They characterized a transformation as consistent if a model before the transformation is consistent with the new model after the transformation.

Refinement is a transformation that takes a model from an abstract level to a more concrete or detailed level. A typical development process, like the Rational Unified Process (RUP) [30], is a sequence of refinement transformations on models. One essential characteristic of this kind of transformation is consistency preservation. Such consistency is also called a vertical consistency since it is between models at different levels of abstraction. The authors of [9, 11] presented several heuristics in the context of class and collaboration diagrams refinement that can be used to check and preserve consistency. Figure 2 shows an example of such heuristics, where a generalization between class A and B in the old model is refined into two generalizations in the new model by introducing class C in between A and B. Such a refinement transformation left the old and the new models consistent between each other since A is still specializing B in both models.

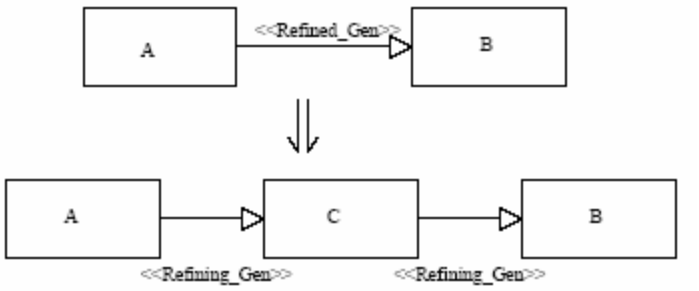

Figure 2 Refinement heuristics for generalizations (left) and associations (right) [9]

Other vertical transformations can occur from requirements to analysis models and from analysis to design models. Berenbach [31] outlined sets of heuristics for creating verifiable analysis and design models. Keeping models consistent with these heuristics is claimed to be a prerequisite to the smooth application of the intended transformations. The first set of heuristics deal with model organization, which is a key requirement to the automated verification and readability of a model. On the analysis side, some heuristics are proposed for the organization and definition of use cases, their relationships and the modeling of the domain's business objects. On the design side, other heuristics are defined to facilitate the tracing from analysis to design and vice-versa.

Unfortunately, not all of these heuristic can be automated since some of them require human cognition to implement or check. The following is a list of example heuristics:

- Package dependencies should be based on content (model organization): a dependency between two packages should exist if and only if there is a dependency between artifacts belonging to these packages.
- A concrete use case cannot include an abstract use case (use case relationships): this situation leads to ambiguity since the concrete use case cannot be defined.
- A concrete class must be instantiated in some interaction diagram (business object modeling): otherwise, the class is redundant and increases maintenance cost.
- Every class in the design model should trace back to a use case in the analysis model (design modeling): it is important for impact analysis to understand what feature or detailed requirement resulted in the need for the design class.

2.4 Multi-Level Consistency

Another classification of consistency is given by Sourrouille and Caplat in [16]. They suggest that consistency can be managed in five different levels. The first level, called the paradigmatic level, is based on the syntax and semantics of the UML language itself. The second level comes from the extensions to the UML meta-model through *profiles* along with their *stereotypes* and OCL constraints. For example, a stereotype <<enumeration>> is used to indicate that a given class represents an enumerated type. Therefore, an accompanying OCL constraint could assert that a class with that stereotype does not have any methods; its attributes have 'public' visibilities and no initial values. Extending the meta-model by the use of stereotypes is a commonly used practice in many domains, like in [34].

A third level of consistency is based on a modeling process. The UML specifications describe the language syntax and semantic but do not prescribe a process to construct different UML diagrams. Modeling processes fill this gap by limiting the scope of allowed expressions in UML and providing style guides to help choose appropriate representations. Constraints could be defined for every modeling process to help keep a model consistent with that process. A modeling process could be generic like RUP [30], or specific to a certain group, project, corporation or community. For example, in one project there could be a constraint preventing the use of nested classes. Other constraints on the same level come from best practices or industry standards. For example, there could be a constraint checking that all classes declare their constructors private and provide static creation operations (factory pattern) [35].

A fourth level of consistency is related to the specific targeted platform or implementation language. No all expressions in UML have equivalents in the targeted platform or programming language. Therefore, constraints are usually specified to limit the use of expressions in UML to only those that can be translated to these domains. These constraints are usually needed during the process of code generation. A prominent example here is the Eclipse Modeling Framework (EMF) [33]. The framework allows for java code generation of models expressed in UML class diagrams. One feature of EMF is to allow for the definition of new data types whose semantics are externally expressed by java code by assigning to classes the pre-defined stereotype <<datatype>>. However, since there is no way for EMF to capture the structure and behavior of such classes, it introduces a constraint that makes sure such those classes cannot be further sub-classed.

The last level of consistency in the same classification deals with the target domain. A target domain is the real world domain that is being modeled. As the authors indicate, constraints in this level are usually dynamic in nature and expressed in terms of the specific modeled domain rather than the generic UML modeling domain. For example, in figure 3, the size attribute of the stack class is constrained to be more than or equal to zero.

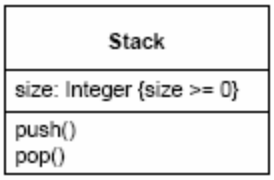

Figure 3 Constraint attached to an attribute [1]

2.5 Nature of Consistency Error

Another way to classify inconsistencies in UML is according to the nature of the error. One kind of inconsistency is a contradiction, where two or more modeling expressions contradict each other. For example, if two instances of different classifiers are communicating, while the two classifiers are not related to each other, then it is a contradiction. This particular kind of inconsistency is common due to the inherent redundancy in UML [23]. The usage of multiple views to define the same system always brings with it the risk of contradiction between these views.

Another kind of inconsistency is incompleteness, which arises when some information is missing from a model [8, 16]. A model is complete when all overlapping diagrams have corresponding elements. However, the specification of these corresponding elements is heuristic and dependent on the followed modeling methodology. For example, a classifier, defined in a class diagram, which does not have an instance specified in a sequence diagram, could be considered a case of incompleteness. Also, a sufficiently complex class, based for example on the frequency of participation of its instances in different interactions, cannot be complete without having a corresponding state machine. Incompleteness could also be inter-model. For example, use cases, defined in an analysis

model, which do not have corresponding interaction diagrams describing them in a design model should be considered a case of incompleteness. There are other cases that exist when the relationship between the models is a refinement transformation. Elements in the original model that do not appear in a refined model are called leaves, while in the opposite situation they are called orphans.

3 UML Consistency Detection and Resolution

Most of the UML inconsistencies reviewed in the previous section can be detected if corresponding constraints are available in the model. It is well known that errors that occur early in the development cycle are the most expensive to correct if they remain undetected. Unfortunately, the majority of today's UML modeling tools do not have satisfactory consistency management features. However, the topic has attracted more attention from the research community. Several approaches are being investigated and prototyped to determine their applicability for integration in UML modeling tools.

If there is one thing that the different proposals dealing with UML consistency in the literature agree on, it is the need to reduce the ambiguity inherent in UML before attempting to analyze a model for consistency. The approaches that deal with UML consistency analysis fall in three main categories. The first category strives to remove the ambiguity by formalizing the meta-model. The idea here is to propose changes to the UML meta-model, or extensions to it, that allow for easier detection of inconsistencies. The second category attempts to remove the ambiguity by having better constraints. Some propose extensions to OCL, claiming that the available OCL primitives are not

sufficient enough to express all kinds of constraints. Others suggest using other constraint description languages like graphical consistency conditions (GCC) [15]. The last category, which is also the most common, proposes the translation of UML models to some formal notation that usually has an inherent consistency management process and a strong tool support.

It is important to note that most of the work in this area is presented in the form of proposed frameworks for consistency management. There is virtually no attempt to quantify these frameworks or obtain metrics for their performance or effectiveness. The literature also lacks serious experimental comparisons of the different proposals. Except in few cases, there are little details about the strategies used for integrating these frameworks in existing commercial tools or about their scalability for real industrial models. The following table outlines the different categories of proposals in this area and summarizes their advantages and disadvantages:

Approach	Advantages	Disadvantages
Meta-Modeling	Natural extension to the language	Strict commitment to the chosen semantics
Constraint Language	Enhanced meta-language allowing for better constraints	Non-trivial implementation and usually needs access to some unavailable meta-model data
Formal Notations	Ease of check consistency and availability of consistency management frameworks	Could be inefficient (not scalable) to implement and difficult to integrate with tools

Table 2: Approached for Dealing with UML Inconsistency

3.1 Meta-Modeling Approaches

The meta-modeling approaches strive to define a fully formal semantics for the UML meta-model. The idea here is that a more formal UML would be easier to check for consistency. According to one inconsistency classification given in section 2.2, any constraint that can be formalized is considered to be syntactic. Only those that cannot be formalized are considered to be semantic like for example "a Constraint attached to a stereotype must not conflict with Constraints on any inherited Stereotype, or associated with the baseClass" [16]. Therefore, it makes sense to increase the expressive power of the meta-language of UML, in order to reduce the number of semantic constraints that are not formally defined [10]. OMG [1] has defined the syntax of the UML language using the meta-model and the OCL well-formedness rules. However, a lot of the semantics is still described in English prose. One outstanding contribution to this category of approaches is by the precise UML (pUML) group [29]. This group aims at identifying areas of ambiguity in UML and defining precise semantics for it. The objective is to ensure that every element in UML is complete, where completeness is defined by having a precise syntax, being well-formed and having a precise denotation in terms of some fundamental aspect of the UML core semantic model. Once an element is complete, the group seeks to capture that by making conservative changes to the meta-model and/or adding more OCL constraints.

Having a complete and fully formal meta-model makes every model element have all its associated elements navigable from it [10]. This facilitates the writing of conditions in OCL [2] or any other constraint language that is dependent on the existence of clear

relationships between associated elements. Another prominent work in the same category is by the OMEGA group [36]. This group aims to select a sufficiently expressive subset of UML, the OMEGA-subset, that is suitable for real-time embedded applications and specify formal semantics for it. This leads to the so-called Omega Kernel model. Based on that kernel, a development methodology is defined that suggests how to develop embedded systems with formal techniques. The goal of the formalization here is to allow for automated model-checking, synthesis and use of theorem proving techniques.

Another group [9] suggests the extension of the UML meta-model to better support model checking for software refinement. They focus on consistency checks between models at different levels of abstraction, which are not covered in the UML standard specification. They identified some consistency preserving refinement rules and captured those using UML stereotypes along with associated OCL constraints. Stereotypes and their grouping in profiles are a light weight mechanism for extending the UML meta-model to better handle consistency.

3.2 Constraint Language Approaches

While the purpose of the first category of approaches to consistency in UML is to enhance the meta-model to make it as precise as possible, the second category's main goal is to increase the expressiveness of constraint definition languages in order to formalize more semantic constraints. Some proposals within this category go about doing this by suggesting enhancements to the OCL. Constraints, like "the data value of a tagged value must conform to the data type specified by the 'tag type' attribute of the tag

definition", cannot be expressed in OCL since it does have a function to convert a data string to a meta-type [10]. Another limitation of OCL is its inability to express consistency restoration rules or actions, which could be useful to implement automatic resolution of inconsistencies. Finally, a claim against the current syntax of OCL is that it hard to use especially for novice users without a modeling experience [10], although there is no substantiated consensus on that. It is also important to realize that OCL is also used at the model level, to describe semantic model constraints, as well as its classical use to constrain the UML meta-model.

One extension to OCL is proposed by the researchers in [37] and used in [5]. They suggest enhancing OCL to include action clauses in the format: if <condition> to <targetSet> send <eventSet>. These actions could be specified for operations; and in that case will be executed at post condition time. They can also be specified for classifiers as part of their invariants. The problem with that extension is that it forces the modeler to commit prematurely to implementation details. An example of the use of action clauses is the following, which describes a credit card expiry situation:

context: CreditCard
inv: validFrom.isBefore(goodThru)
action: **if** goodThru.isAfter(Date.now) **to** self **send** invalidate()

context: CreditCard::invalidate()
pre: none
post: valid = false
action: **if** customer.special **to** customer **send** politeInvalidLetter()
action: **if** not customer.special **to** customer **send** invalidLetter()

Two more extensions to OCL are proposed in [21]. Since navigation over the models and meta-models is often recursive, the authors suggest adding a transitive closure operator to OCL to facilitate this computation. The operator would have the following syntax:

"Given a collection *e* of type *T* and an OCL expression *f(x)* returning a set of elements of type T, the expression ***e->closure(x : T | f(x))*** returns the set of elements of *e* by the transitive closure of the function *f*"

The second extension is to enhance OCL with temporal logic operators to increase the expressive power of the language. The authors acknowledge that temporal constraints could alternatively be expressed in sequence and state chart diagrams at the meta-model level, but according to them this requires knowledge of the UML meta-model, which is not always available to application designers. The proposed extension is based on Computational Tree Logic (CTL), a formalism which assumes that time has a tree-like structure and that a system can evolve towards one of possible directions. A CTL expression is a Boolean expression and therefore could be implemented in OCL by extending the *propertyCall* non-terminal like this:

propertyCall ::= …
"Possibly" ? "Finally" block-expression
"Possibly" ? "Globally" block-expression
"Possibly" ? block-expression "until" block-expression

However, the implementation of these operators is context specific and requires in most cases access to more meta-level information about elements. For example, the implementation of the operator "Globally(expression)" on a sequence diagram message should return true if the expression is satisfied by every future message, which requires access to the collection of messages of a given interaction.

One example of proposals suggesting alternatives to OCL is the one given in [15, 20], where consistency constraints are expressed as graphical consistency conditions (GCC). A GCC is a typed and attributed graphical specification that specifies a negative application condition (NAC): a pattern that must not occur for the condition to be satisfied. Every GCC has a name and a context element. In most cases, a GCC specifies relations between model elements. Each relation has range and domain objects and could be characterized as one or more of the following (see Figure 4):

- **Total**: all objects of the domain must participate in such a relationship
- **Onto**: all objects of the range must participate in such a relationship
- **Functional**: An object in the domain must participate only once in such a relationship
- **Inverse functional**: An object in the range must participate only once in such a relationship

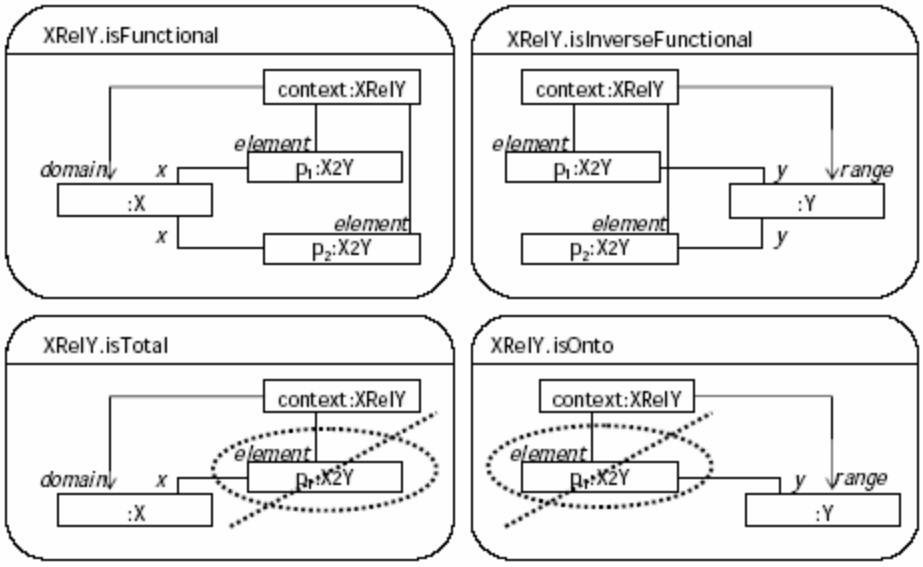

Figure 4 Basic characteristics of relationships as expressed by a GCC

According to the authors, a GCC is more intuitive in describing structural constraints than an OCL expression due to its graph-like notation, which resembles the meta-model. However, non-structural constraints (like evaluation of properties) are still better expressed in OCL, so the authors recommend a combination of both. What is interesting about a GCC is that it can be used as a left hand-side of a graphical transformation rule to establish consistency. The right hand-side in this case would be another graph describing the improved consistent post-state. Three possible actions can be described in the right hand-side:

- Delete an element that violates the consistency
- Insert a "dummy" element, to be detailed later by a user through a tool, to reestablish consistency.
- Ask for user manual intervention in the decision

3.3 Formal Notation Approaches

The third category of approaches to UML consistency checking is generally based on the assumption that ambiguity is built-in to UML and instead of trying to formalize the meta-model or the meta-language, it is better to represent a UML model in or translate it to an already formal language. The main advantage of this approach is that it leverages the capabilities of a well-defined and mature formal language and its tools for the purpose of consistency analysis. There are two general approaches within this category. The first approach tries to keep the model's original representation, in terms of the UML meta-model, intact but also have another parallel representation in the formal language. The obvious question mark around these approaches is one about efficiency. What are the costs of keeping two synchronized copies of the model in terms of memory, speed and

complexity? The other approach ignores the meta-model representation of the model and only keeps one copy in the formal language. The problem here is the cost and complexity of integration with legacy and existing tools that already represent the model in terms of the meta-model like Rational Rose. Obviously, the greater the difference in the syntax of the two formalisms the more costly the integration becomes.

The process of translating from an ambiguous to a formal notation involves making some decisions to clear the ambiguity. These choices freeze the semantics of the ambiguous language. Since there is some ambiguity that needs to be clarified in the UML meta-model, the translation process has the risk of being unsuitable for some semantic domains. Therefore, it is recommended to have a mechanism to customize the translation process to suit a given semantic domain. Another challenge facing approaches in this category is that of traceability. A reported inconsistency would be detected in the formal notation and it would be hard to translate it back to the original language [18].

3.3.1 Viewpoint Unification

One proposal in this category is to perform viewpoint unification. This means translating one view of UML to another. The authors in [40] describe a technique to translate a sequence diagram to a collaboration diagram. They also encode a sequence diagram as a state chart. However, according to [18] this approach cannot apply to all aspects of UML since there is no single notation within UML that can represent all points of view (this also goes against UML having several notations to start with). This approach is only limited to certain diagram type pairs.

A more generic approach would be to find common refinements of different UML viewpoints in a given semantic domain like the Communicating Sequential Processes (CSP) [18]. For example, the authors of [19] defined a mapping from a UML class and its state machine to two separate CSP processes. Then they defined consistency to be *deadlock freedom* of the parallel composition of these two processes. Deadlock freedom here means that the restrictions imposed on a class by its static properties (as defined by the class) and dynamic properties (as defined by the state machine) are not contradictory. However, it was shown in [19] that this approach will only detect contradictions but not other kinds of inconsistencies like incompleteness or extraneous information. Another attempt was done in [41] to map two different UML view using various refinement relationships in CSP (like trace and failures-divergences). Each refinement relationship is good at finding some kinds of inconsistencies. Two views are declared consistent if one of them is a refinement of the other. That could be verified if both of them are CSP refinements when mapped to the common semantic domain.

A labeled transition system (LTS) is another formal framework that is proposed for transforming UML viewpoints and their consistency constraints. In [26], the authors view each model as an LTS, which is a set of states and labeled transitions between them. Every viewpoint of the model (like interaction or state machine views) is mapped to a finite state LTS (which is a subset of the model LTS) and a mode, which could have one of two values: *must* or *never*. Each model viewpoint defines consistency constraints as assertions over states or traces of the system. For example, traces not specified by a state

machine should *never* happen, where traces specified by a sequence diagram *must* happen. Two viewpoints are said to be consistent when their modes are equal or when their modes are different but, if restricted to their common objects, they do not intersect. This approach is more suited to checking the consistency of pure control aspects of behavior but not the data state of the system.

3.3.2 Logical Algebra

Moreover, there are other proposals for transforming UML views to different semantic domains. One of these proposals suggests using a logical algebraic formalism as a semantic domain [13]. Here, the authors define a mapping from UML to different notions in logical algebra like: Item, Signature, FormalModel, Formula and BasicModel. Then, a model is declared syntactically consistent if its signature is well-formed and all of its formulae are well-formed over this signature. Horizontal semantic consistency is established when the semantics of a model, a set of UML-formal systems, is not empty. In this case, a possible inconsistency is detected when one formula is unsatisfiable or when two formulae are mutually contradictory. A similar proposal along the same lines is given in [39] where the authors translate UML diagrams into algebraic specifications and VHDL. However, they did not specify how to check model consistency after the translation to the formal notation.

3.3.3 N-order Logic

Another formalism that is commonly proposed as a semantic domain is *logic* [14, 43]. Usually in these proposals, the meta-model is formulated in terms of n-order logic statements. The consistency constraints are also encoded as statements in the same

formalism. These statements usually contain predicates involving the UML elements that need to be consistent. Finally, a model, including its various viewpoints, is formulated as a knowledge base describing the various elements and their inter-relationships, expressed also in the same logic system. The interesting part is that most logic systems have inference mechanisms which allow for reasoning about the consistency of knowledge bases specified in these formalisms. The reasoning is usually done by giving the reasoning engine a predicate to verify its truth-ness. The predicate here is usually: "the model is consistent". In most cases, the engine applies a backward chaining algorithm on the knowledge base using the rules and finally come up with an answer. However, consistency analysis is only one possible reasoning here and some proposals go beyond that by leveraging the inference mechanisms to evaluate other types of predicates. For example, a predicate like "class1 is dependent on class2" can be checked.

An example of proposals using logic is the one in [14]. In that proposal, the authors use description logic (DL), a two-variable fragment of first-order predicate logic. DL allows representing knowledge by defining atomic concepts (unary predicates), atomic roles (binary predicates) and individuals (constants). One feature of DL is its ability to reason about subconcept-superconcept relationships. Another feature is the use of an open-world semantics, which allows the specification of incomplete knowledge. The authors translated a subset of UML (class, sequence and state diagrams) into a DL system. UML meta-classes are translated into concepts. Meta-associations are translated into roles between these concepts. The OCL well-formedness rules are translated into logic rules. Finally, the model elements that exist in a user model are translated into instances of

these concepts and roles. For example, “class1” is represented as an instance of the concept “Class”. Then, the authors run the inference engine to check for consistency.

3.3.4 Expert Systems

One proposal [16] suggests the use of an expert system as a basis for consistency management and design assist in UML. Most expert systems use an n-order logic language to describe the set of facts and rules in their knowledge base. A procedure similar to what is described earlier takes place to translate the meta-model, constraints and use models into the representation of the expert system. One interesting feature of expert systems though is the ability to write rules in the format: if <expression> then <action>, where expression is a predicate and action is an operation to execute. Formulating rules in this syntax allows for the use of planning algorithms. For example, writing rules, where the expressions are consistency conditions and the actions are appropriate resolutions to them can help in finding plans to restore consistency to an inconsistent model. A similar pattern could be followed to find a recipe for adopting a particular design pattern by finding a sequence of actions that change the model in a particular way. The authors used Sherlock [42], an over the shelf expert system.

Another proposal to use expert systems is given in [23]. In this proposal, inconsistencies are reported and stored in the knowledge base. This allows the user to delay the resolution of these inconsistencies to a later time. The proposal, prototyped in a tool called RIDE, also allows for incorporating user input in the resolution process. Therefore, several categories of expert rules are enumerated here:

- Rules that identify inconsistencies
- Rules that respond to user choice of fixing
- Rules that cleanup expired inconsistency facts
- Rules that cleanup orphaned facts, whose expired parents have been cleaned up

Another proposal that sounds close to an expert system solution is [43]. The authors here propose using a meta-language based on *linear-time temporal logic with actions*. They basically formulate a logical database (D, E), where D contains formulae representing the instance model and E contains formulae representing the meta-model. Consistency is established when the conjunction of the assertions of D and E is satisfiable. Using temporal logic, they specify how models should evolve over time by using the usual temporal operators like "next, last, sometime-in-the-future, always….etc". They also formulate consistency rules in terms of temporal logic constraints. When an inconsistency is identified, one or more action rules fire. They record a history of how they handled that inconsistency in the past, in addition to a history of how the inconsistent data got into the specification. These histories will be part of the context necessary for the inconsistency resolution.

4 UML Consistency Management Framework

4.1 Managing Consistency

As discussed in the previous sections, inconsistency can naturally occur during UML modeling. It can be caused by the collaboration of multiple modelers with different interpretations of the modeled system. It can also happen as a result of describing a system from several viewpoints. Another reason for inconsistency is the uncertainty

inherent in the modeling process, which favors leaving inconsistencies until a model is later refined and detailed. Finally, inconsistency can occur due to unintentional errors.

The traditional way of dealing with model inconsistencies is to seek their elimination. However, in a lot of cases, inconsistent models do not pose a compelling problem until a much later time. For example, inconsistency is desired when a modeler is still evaluating alternative solutions to his problem without prematurely committing to any of them. Another example is the temporary inconsistency resulting from doing an incremental change to a model. For these examples and others, the objective should be to *manage* consistency rather than to remove it. This means preserving consistency when it is desired and remedying inconsistencies only when needed (usually before doing actions that are based on consistent information). This is not a simple task and definitely requires a change in the way of thinking. The interested reader is referred to [43] for a discussion of why inconsistency is not such a "bad" thing.

Traditionally, most UML modeling tools either try to prevent inconsistencies from happening by providing a structured modeling environment, or support the so called *free-form* modeling without providing proper support to deal with inconsistencies. While the first approach is more suited to a closed and relatively homogeneous modeling culture, the second approach works only for modeling on a small scale. With today's increasingly complex software systems, different modeling approaches have to be employed. The new trends of distributed collaborative work, very large and complex systems, the variety of skill for different team members, the increasing ambiguity of requirements and the

expansion in scope of UML as a modeling language, require a modern UML consistency management framework, a set of strategies, policies and tools to cope with this complexity and assure consistency. In [44], the authors presented a strategy for managing consistency in general. A contribution of this paper, detailed in the remainder of this section, is a customization of this strategy from a UML modeling perspective, along with recommendations for the construction of a pragmatic consistency management framework (CMF).

4.2 Consistency Management Challenges

The first challenge is the ambiguity of UML as a modeling language. Without having exact semantics defined, it is hard to determine whether the assertions made in a model are consistent with each other and with the modeled system. Another problem can come from the nature of the inconsistency itself. Some inconsistencies are not localized but are distributed across diagrams, viewpoints or even models. Drilling down the related information in the mass of all assertions made in the model is not simple. Other inconsistencies are hidden or implicit and need a more careful and deeper analysis to figure out. There may also be many existing inconsistencies, and hence finding another inconsistency given the current polluted state is not trivial. Furthermore, it could be quite difficult sometimes to automatically decide if an inconsistency is safe or to decide on the course of action needed to remedy it without human cognition.

4.3 Consistency Management Characteristics

Unfortunately, most of today's UML modeling tools fall short of providing a comprehensive consistency management framework. Such a framework needs to have certain characteristics to properly deal with the complexity of consistency analysis. The following is an attempt to enumerate some of these characteristics.

4.3.1 Consistency Analysis Infrastructure

A basic requirement for a CMF is the ability to read a model, process it, reason about it and modify it. Models are usually represented in terms of some formalism. In the case of UML, the formalism is a combination of the MOF-based UML meta-model, as defined by the OMG [1], and OCL. However, as explained in section 3, the meta-model and OCL may not represent a good foundation for consistency analysis for various reasons. Therefore, a first step for implementing a CMF may be to consider some of the proposals made to enhance the representation of the UML model. Another consideration should be given to the method of expressing the consistency rules in order to better support the required features of the CMF.

4.3.2 Consistency Analysis Coverage

One of the basic expectations of a CMF is to deal with the syntactic and semantic correctness of UML models. This activity usually involves checking that a model conforms to its meta-model and satisfies the well-formedness rules. However, a lot of other semantic constraints are either informally expressed in the prose of the UML specification, implicitly specified or totally missing. Therefore, a more rigorous

inspection needs to be performed to extract or find these semantic constraints and express them either in OCL or in the chosen rule-based language of the CMF.

In addition to the well-formedness rules, a comprehensive CMF would optionally package other consistency rules for commonly used heuristics in analysis, design and architecture of OO software. For a good coverage of these heuristics, please refer to [31]. Consistency rules for other common semantic domains could also be optionally provided in the framework. For example, consistency rules implied by familiar modeling processes (like RUP [30]) or targeted at popular platforms (like Eclipse) or implementation languages (like C++/Java) could also make sense. Another level of coverage that is usually sought after is inter-model consistency analysis. For example, a CMF could provide rules to analyze the consistency of multiple models related by one or more transformation relationships. One common example of such a relationship is refinement.

4.3.3 Consistency Rules Specification

One of the main features of any CMF is its consistency rules management. A good CMF would allow for an elaborate definition of a consistency rule including one or more of the following fields [31]:

- A descriptive name for the inconsistency
- A brief and detailed general description of the inconsistency
- A contextual diagnosis message explaining the cause of the inconsistency
- A recommendation for the repair actions of the inconsistency
- A desired severity of the inconsistency (high/low or error/warning…etc.)
- An expression in the CMF chosen rule-based language for the detection and the optional automatic repair of the inconsistency.

In addition to the ability to express rules using the previous scheme, a good CMF needs to allow users to do perform some or all of the following operations on rules:

- The ability to dynamically enable and disable rules
- The ability to configure and customize existing rules
- The ability to express new rules easily using wizards
- The ability to contribute rules dynamically through an extension point
- The ability to check consistency or detect inconsistency between rules [23]
- The ability to order rules by priority [23], scheme or heuristic [24]
- The ability to classify rules into catalogs (well-formedness, methodology …etc)
- The ability to group rules into profiles (analysis, design, modeler, tester…etc)
- The ability to associate rules with checking levels (easy, strict …etc)
- The ability to define a granularity for rules (user, project, model, …etc)
- The ability to import/export rules

4.3.4 Inconsistency Detection

As discussed earlier, a good CMF has to have a strategy for dealing with inconsistencies. One strategy is to prevent inconsistencies from occurring by providing a controlled editing framework. In such a framework, only consistency preserving model transformations are allowed to happen. Alternatively, inconsistencies are allowed to occur but are detected and tolerated. In [15], the authors presented a strategy for tolerating inconsistency prototyped in the Fujaba tool suite.

One of the desired features of a CMF is to allow for the manual or automatic analysis of inconsistency. A manual analysis, also called batch analysis, checks the model as a whole based on a user request. No assumptions are made apriori to the consistent state of the model and the analysis process is mainly iterative over the whole scope of the model. On the other hand, an automatic, or just-in-time, consistency analysis is a more sophisticated strategy that starts by assuming that the model is initially consistent. As the model gets

changed, the analysis process starts by identifying the rules that might have been affected by the change. This is only possible if the rule specifies the necessary context (in terms of affected model elements) for its applicability. For every such rule, only the context elements are analyzed for consistency. Different strategies for selecting rule could apply.

Automatic detection is much more logical to have while the model is incrementally changing. However, the catch is that it has to be not intrusive to the user. Since consistency analysis could be computationally expensive, applying that process whilst a user is editing the model could lead to sluggishness in the editing experience. Therefore, a CMF needs to find a more efficient and highly scalable implementation for the automatic consistency analysis. On the other hand, the history of the user changes leading to the inconsistency could be invaluable in customizing the analysis to the correct context and therefore help present a more logical diagnostic feedback to the user.

Another nice to have feature of a CMF is the definition of a scheme to specify which rules to check, when they should be checked, and whether they should be manual or automatic in each case. The scheme could be static, dynamically changeable by the user or completely self-adjusting by adopting some machine learning algorithms. The last point is a really interesting one since it can help change the analysis process based on feedback from the user and the history of his actions and decisions in the editing session.

4.3.5 Inconsistency Visualization

Once an inconsistency is detected, it needs to be presented to the user. There are different strategies for accomplishing that. One way is to present the user with a generic textual description of the inconsistency. A slightly better way is to customize the description message to the actual context of the inconsistency. Although both these approaches are informative, it can really be hard for a user (especially a novice one) to grasp the essence of the inconsistency or to relate it to what he/she just did (in case of automatic analysis). A much nicer way is to provide a visual inconsistency feedback to the user. This feedback can both be contextual and localized to make it easy for the user to get it. An example of such feedback is to adorn the problematic element with an error or a waning decoration that display, when hovered over, a contextualized message of the problem. A more elaborate visualization strategy could be adopted in case the inconsistency spans multiple elements, diagrams or models. A lot of relevant research in usability could come into play in the point.

4.3.6 Inconsistency Resolution

One of the more interesting features about a CMF is its inconsistency resolution strategy. Most UML modeling tools, like Rational Rose, allow only a manual resolution of inconsistency through user intervention. However, a tool with a better CMF would allow for configuring the resolution strategy for every inconsistency rule to be either manual or automatic. In fact, a similar scheme to the detection process could be defined here to control when and how every inconsistency is resolved.

In case of automatic resolution, the possible actions are specified in the inconsistency rule itself [15, 16, 23]. If one action is specified, it is automatically applied. If multiple actions are given, either a schema for choosing which action to automatically apply is given or a user is polled to make a decision. Similar to the detection phase, a machine learning algorithm could be employed to learn why a user selects one action versus another in each context and then offer this choice by default the next time.

Options for the resolution of inconsistencies are many. Possible ones are to ignore the inconsistency, to reduce its severity, to delay handling it, to improve it or to completely resolve it [44]. The resolution could also be applied one inconsistency at a time or in a batch mode. In all cases, an efficient and scalable implementation of the resolution makes the CMF not intrusive to the user. Also, in a similar fashion to inconsistency detection, context sensitive resolution options could be presented to the user visually and close to the source of the problem. When the user decides to delay handling of an inconsistency, the CMF needs to preserve the inconsistency and maintain it by periodically checking if it still applies (house keeping). An example of this procedure exists in [23]

4.3.7 Inconsistency Diagnosis and Resolution Planning

A successful CMF would have the ability to diagnose a model and present a user with a summarized and a detailed consistency analysis report. The summarized version would report statistics about the model conformance to every consistency rule. It can also allow the user to establish quality metrics and report on the quality level of the model. A detailed analysis report on the other hand would have a listing of all the found

inconsistencies along with contextual information to the rationale of each problem. An example of such a feature exists in DesignAdvisor [31], a tool built by Siemens Corporation.

Another role a CMF can play is that of a modeling advisor. In addition to reporting consistency problems, it can also help the user by giving advice on how to restore consistency to a model or on how to improve the quality of a model. To be able to do that, the CMF needs the ability to reason and to plan given a configured set of rules. Various reasoning and planning algorithms could be applied here if the rules were expressed in the proper format (for example, if <conditions> then <actions>)

4.3.8 Consistency Framework Implementation

As discussed in this section, designing a successful CMF is not a trivial task. A lot of design decisions have to be made at various points. Another level of complexity presents itself when such a framework is integrated into UML modeling tools. An ideal CMF would have a clearly defined interface for integration with such tools. Every tool would implement all or part of this interface to take advantage of the framework. Another desired feature of a CMF implementation is to have an open architecture. Such architecture would allow for plugging in different algorithms for various modules in the framework. For example, the reasoning engine could be replaced with a new one seamlessly. This feature would allow the CMF to easily evolve over time.

5 Conclusion and Future Directions

UML consistency analysis is an important process for ensuring the quality of UML models. Consistency analysis goes beyond just conforming to the meta-model and the well-formedness rules available in the UML specifications [1, 2]. A model is consistent when it conforms to the semantics of all its targeted domains [16]. There are different ways to classify consistency presented in the literature. In every classification, various consistency detecting rules are defined. These classifications differ in their scope and locality (inter/intra model, static/dynamic…etc).

The UML specification is inherently ambiguous. Formalization has been identified as the key to the successful implementation of UML consistency analysis. Various proposals are made to help formalize UML at different levels. The first group of proposals seeks to remove the ambiguity from the UML meta-model by attempting to complete it with all the missing relationships and constraints. Another group suggests enhancing the UML constraint language to better capture the different aspects of consistency. A third group proposes converting a UML model into another more formal notation and then leveraging that notation's existing support for consistency analysis. A combination of these proposals is usually needed.

Building a UML consistency management framework is not a trivial task. It involves a lot of design decisions at various levels. First, the UML meta-model is enhanced to facilitate the detection and correction of inconsistencies. The enhancement could be in the meta-

model definition (by defining missing semantics or clearing ambiguity), implementation (by having reasoning capabilities for example) or both. Second, a language for expressing consistency rules is chosen. The language could have features to express both structural and non-structural constraints, operatives to apply corrective actions to restore consistency, and an easy notation that is understandable by the average modeler. Third, a strategy to manage consistency rules is defined. The strategy could allow for the addition, deletion, and customization of consistency rules. Fourth, different options to detect inconsistencies are presented. These options could range from totally automatic to manual, from batch to incremental and from predefined to continuously changing. Fourth, creative ways for the visualization of inconsistencies are designed. These ways should be non-intrusive and intuitive to the user. Fifth, a flexible consistency resolution strategy is introduced. Again, similar to the detection strategy, it can range from totally automatic to manual, from batch to incremental and from predefined to continuously changing. Sixth, an advisory role, where the framework assists the user in enhancing the quality of the model by presenting him with various tips and suggestions, is implemented. Last, an open architecture is used in designing the framework to allow it to integrate with various tools and to evolve over time.

The current state of the art is still far from being satisfactory. An evidence of that is the lack of adequate support for consistency management in today's open source and commercial UML CASE tools. More work is still needed in implementing various consistency profiles. These profiles customize the meaning of consistency to different targeted domains, since there is no universal agreement on the meaning of consistency.

Also, different strategies for detecting and resolving inconsistencies need to be quantified and compared in terms of their suitability for tool support. Data about speed and memory usage patterns of these strategies are essential for taking appropriate decisions on tool integration. More research is also needed in the areas of detecting, visualizing, presenting and correcting inconsistencies. Interesting ideas from the domains of usability and machine learning could also be useful here to enhance the user experience of consistency management.

References

1- OMG. UML 2.0 Infrastructure Specification, September, 2003. http://www.omg.org/docs/ptc/03-09-15.pdf

2- OMG. UML 2.0 OCL Specification, October, 2003 http://www.omg.org/docs/ptc/03-10-14.pdf

3- L. Kuzniarz, G. Reggio, J. Sourrouille and Z. Huzar. Workshop on Consistency Problems in UML-based Software Development. UML 2002. Blekinge Institute of Technology. Research Report 2002:06.

4- L. Kuzniarz, G. Reggio, J. Sourrouille and Z. Huzar and M. Staron. Workshop on Consistency Problems in UML-based Software Development II. UML 2003. Blekinge Institute of Technology. Research Report 2003:06.

5- H. Gomaa and D. Wijesekera. Consistency in Multiple-View UML Models: A Case Study. Available at [4], page 1-8.

6- L. Kuzniarz and M. Staron. Inconsistencies in Student Designs. Available at [4], page 9-17.

7- T. Feng and H. Vangheluwe. Case Study: Consistency Problems in a UML Model of a Chat Room. Available at [4], page 18-25.

8- C. Lange, M. Chaudron, J. Muskens, L. Somers and H. Dortmans. An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs. Available at [4], page 26-34.

9- W. Shen, Y. Lu and W. Low. Extending the UML Metamodel to Support Software Refinement. Available at [4], page 35-42.

10- J.L. Sourrouille and G. Caplat. A Pragmatic View on Consistency Checking of UML Models. Available at [4], page 43-50.

11- B. Hnatkowska, Z. Huzar, L. Kuzniarz and L. Tuzinkiewicz. Refinement relationship between collaborations. Available at [4], page 51-57.

12- G. Genova, J. Llorens and J. Fuentes. The Baseless Links Problem. Available at [4], page 58-61.

13- E. Astesiano and G. Reggio. An Algebraic Proposal for Handling UML Consistency. Available at [4], page 62-70.

14- R. Straeten, T. Mens and J. Simmonds. Maintaining Consistency between UML Models Using Description Logic. Available at [4], page 71-77.

15- R. Wagner, H. Giese and U. Nickel. A Plug-In for Flexible and Incremental Consistency Management. Available at [4], page 78-85.

16- J.L. Sourrouille, G. Caplat. Checking UML Model Consistency. Available at [3], page 1-15.

17- B. Hnatkowska, Z. Huzar, L. Kuzniarz and L. Tuzinkiewicz. A systematic approach to consistency within UML based software development process. Available at [3], page 16-29.

18- J. Derrick, D. Akehurst, and E. Boiten. A framework for UML consistency. Available at [3], page 30-45.

19- H. Rasch and H. Wehrheim. Consistency between UML Classes and Associated State Machines. Available at [3], page 46-60.

20- J. Hausmann, R. Heckel, and S. Sauer. Extended Model Relations with Graphical Consistency Conditions. Available at [3], page 61-74.

21- J.P. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazex and L. Feraud. Extending OCL for verifying UML models consistency. Available at [3], page 75-90.

22- R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. Available at [3], page 91-105.

23- W. Liu, S. Easterbrook, and J. Mylopoulos. Rules Based detection of Inconsistency in UML Models. Available at [3], page 106-123.

24- C. Gryce, A. Finkelstein and C. Nentwich. Lightweight Checking for UML Based Software Development. Available at [3], page 124-132.

25- K. Lano, D. Clark, and K. Androutsopoulos. Formalizing Inter-model Consistency of the UML. Available at [3], page 133-148.

26- P. Bhadrui and R. Venkatesh. Formal Consistency of Models in Multi-View Modeling. Available at [3], page 149-.

27- Y. Shaham-Gafni and S. Kremer-Davidson. Consistency in UML 2.0 – Classes vs. Objects. IBM Haifa Research Lab.

28- S. Kremer-Davidson and Y. Shaham-Gafni. UML 2.0 Model Consistency – The Rule of Explicit and Implicit Usage Dependencies, IBM Haifa Research Lab.

29- A.S. Evans and S. Kent. Meta-modeling semantic of UML: the pUML approach. In proceedings of UML 1999. LNCS 1723, 1999, 140-155.

30- P. Kruchten. The Rational Unified Process: An Introduction (2nd Edition). Addison Wesley Longman Inc., 1999.

31- B. Berenbach. The Evaluation of Large, Complex UML Analysis and Design Models. Siemens Corporate Research, Inc. In IEEE proceedings of ICSE 2004.

32- B. Berenbach. Comparison of UML and Text based Requirements Engineering. Siemens Corporate Research, Inc.

33- F. Budinsky, D. Steinberg, E. Merks, R. Ellersick and T. Grose. Eclipse Modeling Framework. Addison-Wesley Professional, August 2003.

34- H. Gomaa. Designing Concurrent, Distributed and Real-Time Applications with UML. Addison-Wesley Object Technology Series, ISBN: 0-201-65793-7, August 2000.

35- E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Addison-Wesley, ISBN: 0-201-63361-2, 1994.

36- J. Hooman. Towards Formal support for UML-based development of embedded systems. University of Nijmegen. Avaiable at: http://www-omega.imag.fr/doc/d1000230_1/HoomanPROGRESS.pdf

37- J. Warmer and A. Kleppe. Extending OCL to Include Actions. In Proceedings of UML 2000. p. 440-450, Springer-Verlag.

38- A. Egyed. Automating Architectural View Integration in UML. In proceedings of ESEC/FSE99, 1999.

39- B. Cheng, E. Wang and H. Richeter. Formalizing and Integrating the Dynamic Model within OMT. IEEE Proc. of International Conference on Software Engineering, Boston, MA, May 1997.

40- K. Kosikimies, T. Systa, J. Tuomi, and T. Mannisto. Automated Support for Modeling OO Software. IEEE Software, Jan 1998.

41- J. Davies and C. Crichton. Concurrency and Refinement in the Unified Modeling Language, Electronic Notes in Theoritical Computer Science 70 No.3 (2002).

42- G. Caplat. Sherlock Environment.
Available at //servif5.insa-lyon.fr/chercheurs/gcaplat/

43- A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In proceedings of European Software Engineering Conference, pages 84-99, 1993.

44- A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In M.T. Ibrahim, J. Kung, and N. Revell, editors, proc of the 11th International Conference on Database and Expert Systems Applications (DEXA'00), London, UK, LNCS 1873, pages 1-5. Springer-Verlag, September 2000.

45- B. Selic, G. Gullekson and P. Ward. Real-Time Object Oriented Modeling. Published by Wiley Professional Computing, April 1994.