Constraint Checking in UML Modeling

Jean Louis SOURROUILLE
INSA, Bat. Blaise Pascal
F69621 Villeurbanne Cedex , France
sou@if.insa-lyon.fr

Guy CAPLAT
INSA, Bat. Blaise Pascal
F69621 Villeurbanne Cedex , France
caplat@if.insa-lyon.fr

Abstract

Modeling aims to represent a system in a given formalism. As representations based on high level modeling languages can be interpreted in many ways, constraints are introduced to restrict the field of the possible. On the other hand, language semantics is defined using constraints as well. Within the context of the UML, a classification is proposed to clarify the nature of the constraints that must be fulfilled to ensure model correctness. Constraint violations are only warnings while others are serious. Depending on the kind and the context of the violation, help and advice can be supplied, and improvements may be conditionally done. In the UML, constraints are assertions described in a side-effect free language (OCL), while actions are not supplied. To extend capabilities, the constraints are translated into well-using modeling rules that form the knowledge base of an expert system in modeling. This modeling companion is briefly outlined.

Categories and Subject Descriptors

D.2.2 [**Software engineering**]: Design Tools and Techniques---*Object-oriented design methods*; D.2.6 [**Software engineering**]: Programming Environments.

General Terms

Design, Languages, Verification.

1. INTRODUCTION

Modeling is a central activity in software development. From the problem domain to the solution domain, models are enhanced, detailed and transformed until they are precise enough for the implementation step. During development, the initial field of possible interpretations of the models is reduced to lead to one implementation that should be a legal interpretation of these models.

A model is a representation of a system (from a given point of view) expressed in a formalism or language. A formalism is based on modeling principles, usually named a paradigm, that define the way any system should be considered (e.g., object-oriented, functional). Formalisms include syntactic and semantic constraints that define the meaning of the language primitives as well as the right way to use them. First, any legal model should fulfill these constraints. Then, to eliminate unwanted interpretations, users enforce additional constraints. These constraints stem from the modeled domain (e.g., to give a response before 15 ms), the implementation domain (e.g., no multiple inheritance), and even from the modeling process domain (e.g., a sequence diagram is required for each external event). How to define constraints to get only meaningful interpretations? How to describe and enforce these constraints? Further, what are the kinds of constraints? The paper examines deeply these central questions for the UML (Unified Modeling Language [21]). The UML includes the notion of constraint and supplies the OCL (Object Constraint Language) that apply to both metamodel and model elements. The UML metamodel, i.e., the model of the formalism, cannot be changed. New notions are added using an extension mechanism, and again constraints are used to define the semantics.

The OCL is very useful to describe well-formedness rules, but a more sophisticated solution is required because it is a pure expression, side-effect free, language: operation invocation, action and even message display are not possible. The advocated solution is the construction of an expert system in modeling.

Many works are related to software development assistance (e.g., [1]). Knowledge bases and reasoning capabilities are used to deal with the development process (e.g., [14]), requirement acquisition (e.g., [16]), and application domain description (e.g., [10]). Some works address the whole development process, including the refinement to formalize models (e.g., [19]). Unlike these works, the paper focuses on modeling in a given language, and the final aim is to build a modeling companion for the UML, in particular for beginners. As another main difference, most of the constraints apply to the model in which they are themselves described.

The rest of the paper is organized as follows. Section 2 clarifies the notion of constraint. Section 3 gives a characterization of the various constraints in UML models. Section 4 shows interesting and typical examples of these constraints. The production line of a modeling companion based on an expert system is proposed section 5. Finally, related works are examined.

2. THE ROLE OF CONSTRAINTS IN FORMALISMS

2.1. Modeling Notions

The Figure 1 gives a simplified model of the relationships between the main modeling notions. A *system* is anything to model. A *model* is an abstraction that represents a view of this system. The intent of a model is to represent a system into a *formalism* using a set of *abstract primitives*. These primitives are relevant notions of modeling, (e.g., a *class*) that are mapped to forms, i.e., *concrete primitives* (e.g., a *box*). The *semantics* of the notions (e.g., what a *class* is) expresses their meaning using any language, including natural language. This semantics includes *constraints* that induce restrictions on the use of notions in order to ensure that a model has at least one licit interpretation in the modeling domain.

The UML metamodel cannot be changed. This is justified at least for practical reasons: otherwise, models might be licit with some metamodels but illicit with another ones. However, a greater expressive power is often required for specific purposes: the UML abstract primitives (Figure 1) include the ingredients needed to define new notions. Pseudo-metaclasses based on UML metaclasses and defined by means of *stereotypes* (detailed below) may be stored in *profiles* shared by several users. Obviously, the semantics of the new primitives is not defined at metamodel level but at user model level in the profile. It is defined using OCL constraints and adornments in free text, the same as at the upper level, while an attribute maps the *stereotype* to a concrete form. Finally, any property can be added to any model element using a *tagged value*, that is a pair (*name*, *value*).

3. CONSTRAINT CLASSIFICATION

3.1. Syntactic vs. Semantic Constraints

Syntactic and semantic constraints are often distinguished in language definitions. For instance, "*if the class A inherits from the class B, then B cannot inherit from A*" is a semantic rule that defines the meaning of the *inheritance* abstract primitive. When this rule is translated into a formal expression, it becomes a simple syntactic rule. In fact, *all the rules are induced by the semantics of the language primitives*. By definition (as in [5]), *syntactic constraints are expressed in a formal language* (e.g., OCL) and theoretically can be checked automatically. *Semantic constraints are expressed in natural language* and should be checked manually. For instance, let the syntactic rules of a small language:

clause ::	*adjective*, *commonNoun*
commonNoun ::	'man' | 'woman' | 'apple'
adjective ::	'happy' | 'old' | 'mellow'

In this language, the clause "*happy apple"* is syntactically licit while it is semantically wrong in usual contexts. This clause becomes syntactically wrong after grammar modification:

clause ::	*clauseHum* | *clauseObj*
clauseHum ::	*adjectiveHum* , *commonNounHum*
clauseObj ::	*adjectiveObj* , *commonNounObj*
adjectiveHum ::	'happy'
commonNounHum ::	'man' | 'woman'
adjectiveObj ::	'mellow'
commonNounObj ::	'apple'

This example shows that the distinction between syntactic and semantic constraints is somewhat arbitrary.

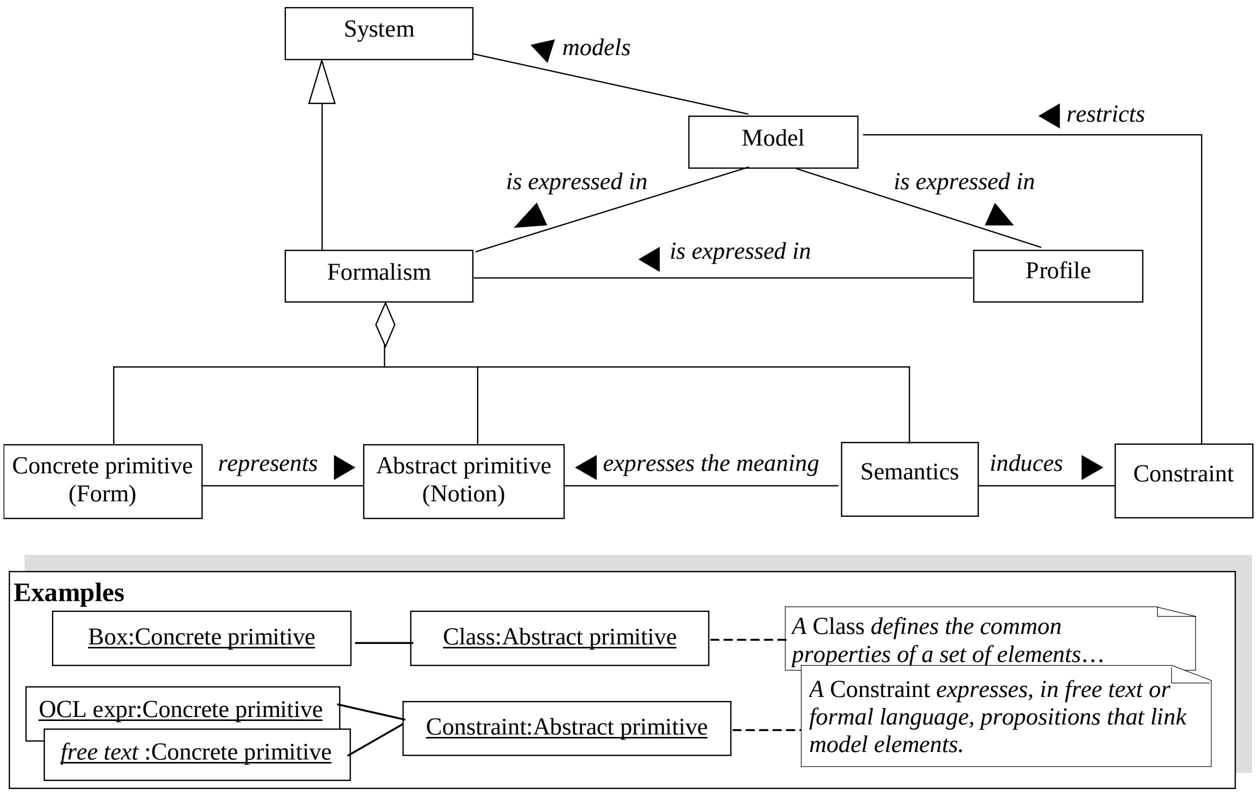

System
models
Model
restricts
is expressed in
is expressed in
is expressed in
Formalism
Profile
Concrete primitive (Form)
represents
Abstract primitive (Notion)
expresses the meaning
Semantics
induces
Constraint
Examples
Box:Concrete primitive
Class:Abstract primitive
A Class defines the common properties of a set of elements…
OCL expr:Concrete primitive
free text :Concrete primitive
Constraint:Abstract primitive
A Constraint expresses, in free text or formal language, propositions that link model elements.

Figure 1. Relationships between system, model and formalism (expressed in the UML)

3.2. Models, Interpretations and Constraints

Models[1] expressed in the concrete syntax (language of forms) enforce syntactic constraints. The *interpretation* is the mechanism by which sense is given to a model, but commonly the result of this mechanism is also called interpretation. Generally, several interpretations can be associated with a model, the many interpretations reflecting the under-specification of the model being built, e.g., when the direction of navigation between objects is unknown. During the building process models become increasingly precise, and the number of possible interpretations decreases (Figure 2).

Illicit models are detected by syntactic constraint checking, generally done by checks on demand in tools for modeling in the UML (*UML tool* to be brief). These automated checks should be completed with manual checks since all the constraints cannot be formally described (e.g., "*A constraint attached to a stereotype must not conflict with constraints on any inherited stereotype, or associated with the baseClass*" [21]). Constraints coming from the formalism stand at ***paradigmatic level*** and are unchangeable. Constraints coming from stereotypes are quite similar but changeable and they stand at ***paradigmatic extension level***. In addition, other kinds of constraints are of interest.

On the one hand, as in any language, there are well modeling rules described in a style guide, and more generally, specific rules common to a community or a type of application. These

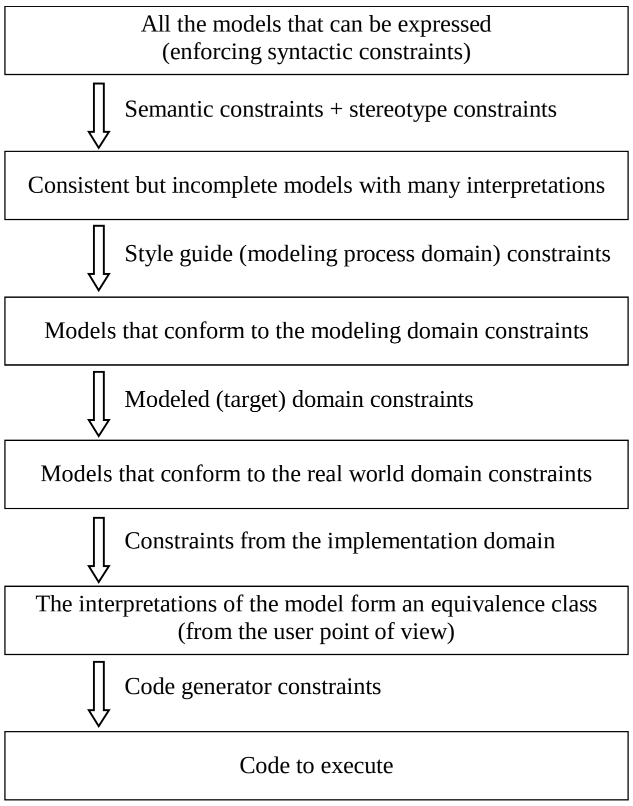

All the models that can be expressed
(enforcing syntactic constraints)
Semantic constraints + stereotype constraints
Consistent but incomplete models with many interpretations
Style guide (modeling process domain) constraints
Models that conform to the modeling domain constraints
Modeled (target) domain constraints
Models that conform to the real world domain constraints
Constraints from the implementation domain
The interpretations of the model form an equivalence class
(from the user point of view)
Code generator constraints
Code to execute

Figure 2. Decreasing of the number of interpretations

[1] The term *model* should be understood in the broad sense of a set of UML expressions describing views of a system.

constraints at ***modeling process level*** forbid expressions normally licit in the formalism, while models still conform to the UML metamodel.

On the other hand, a licit model in the modeling world has not necessarily a sense in the modeled domain. Again, the meaning of the elements of the model has to be defined adding constraints thus decreasing the number of allowed interpretations. Unlike previous ones, these constraints related to the modeled world are expressed in terms of this modeled domain: they are at ***target or modeled domain level***. As they express domain-specific sense, they cannot be expressed in terms of metamodel primitives.

Finally, some but not all of these constraints can be *statically* checked on the model under construction. Constraints that involve instances can be checked *dynamically* only, e.g., *balance* > 0 or *thread number* < 100. A complete dynamic checking would require a monitoring of the running program: this is out of the scope of this work that only deals with dynamic check of instances described in the user models.

3.3. Interpretation and Equivalence Class

The formal definition of the UML (works about precise UML [15]) decreases the number of the licit model interpretations. Is it possible and/or desirable to have a unique interpretation of a model? The fact to have several interpretations is not a problem as far as the reader knows the possible interpretations. Problems arise when a model is not interpreted in the same way by everybody. Therefore, a precise definition of the UML is enough: all the possible interpretations of a model should be specified.

Even going down to the level of detail of a programming language, any UML model will remain under-specified in the sense that it could be implemented in several ways. In a programming language, the expressions are interpreted (i.e., translated into machine code) in a different way by each compiler. Therefore, to have a unique interpretation is not the actual problem: *when the interpretations form an equivalence class, there is no ambiguity*. It is the case particularly for programming languages since users do not want to know implementation details. When implementations are not viewed as equivalent, due to performance for instance, often there are ways to change the implementation in the target language program.

During model translation, code generators interpret specific adornments (*tagged values* in the UML) and add specifications and/or constraints (***implementation level***). These details are required to implement the class diagrams, and even more the dynamic model because it is not fully specified [11].

4. SEMANTIC CONSTRAINT CHARACTERIZATION

The UML constraints are first compared with expert system rules, and then examples illustrate the constraint classification.

4.1. UML Constraints vs. Expert System Rules

The above brief review shows the diversity and the importance of constraints during development. In the UML, a constraint is a boolean expression on an associated model element: this expression must be true for the model to be well formed. In other words, a constraint is an *invariant* (*pre* and *post*conditions are not taken into account), i.e., an assertion that should be always true. Constraints can be written in any language with a well-defined semantics, including natural language and of course the OCL. The straightest solution would be to write constraints in the OCL, but this does not solve the entire problem:

– Many UML tools do not include OCL, and separate OCL tools still do not provide complete solutions [18],
– OCL is not easy to understand (and even harder to write), and most users will not accept it,
– OCL is not convenient to express useful but simple constraints (see name conflict below),
– Some constraints at paradigmatic level cannot be expressed,
– OCL is a pure declarative language: only expressions whose value should be true for the model to be valid can be expressed. More generally, UML constraints should be written using a side-effect free language.

Although the latter point is a major drawback, it is justified since constraint checks resulting in direct model modification would raise numerous problems. The description of rules leading to recommendations or modeling tips requires writing general expressions such as:

if Expression *then* Action

where *Expression* is a predicate and *Action* any operation to execute, for instance to display a message.

To go beyond the capabilities of the UML, the advocate solution is to associate UML modeling tools with an expert system that we call the *Constraint Checker* (CC). It aims to analyze models, to provide diagnosis and suggestions, and to execute operations on models. Moreover, its rule-based formalism provides logical deduction chains and inferences. The above *Expression* can be described using either OCL when its expressive power is enough [9] or a textual form of the CC language. This solution is fully UML compatible with the additional advantages (unordered):

– When a constraint is violated, not only it is detected but also the CC analyzes the context and proposes a solution based on its knowledge base. For instance it can ask for a missing information or suggest a new structure that will apply to if the user agrees.
– As the CC supplies variables, a large range of new capabilities opens among which: constraints can be processed in different ways according to their level of seriousness and according to the current state of the CC ; users may choose the messages they want to display, the warnings that must not be redisplayed, etc.
– Expressions can be written in the guided CC language using small, linked together and reusable pieces of information.
– Notions (metaclasses extended with stereotypes) can be defined in models while their constraints are defined in the CC. Thus, the constraints associated with a notion can depend on parameters, for instance, parameters related to the implementation domain (target language, libraries, etc.).
– The UML metamodel is described using the CC language, that is the CC adds a metalevel that makes it possible to check constraints that cannot be expressed in the UML metamodel.
– The CC holds a library of predefined notions that can be used in any model, independently of the profiles.

4.2. From UML Constraints to Production Rules

Constraints from each of the above semantic levels contribute to give sense to the model elements. The translation of these constraints into *well-using modeling rules* provides the declarative part of an expert system in modeling. The constraints are stored into several knowledge bases to use simultaneously or distinctly according to the modeling step and the type of the constraint to check.

How to transform constraints into production rules? Every UML constraint *C* owns in its attribute *body* a boolean expression that must be true. The mapping from *constraint* to *checking rule* is done by inferring model defaults from a negative form *expr* of the constraint body:

if C.expr *then* *let* (Model.correct = *false*).

For instance, the UML well-formedness rule "*no attributes may have the same name within a Classifier*" becomes in OCL the invariant "*if two attributes have the same name, they should be the same attribute*" (attributes are features):

```
context c : Classifier    inv :
    c.feature->select (a | a.oclIsKindOf (Attribute)) ->
                    forAll ( p, q | p.name = q.name implies p = q )
```

The general form of the CC language textual representation is:

(Name, Level) Declaration | Filter : *if* Expression *then* Action

Using this form, the above OCL invariant leads to the production rule named "Name conflict":

```
∀ p, q ∈ Attribute, ∀ c ∈ Classifier |
    (p≠q) and isAttributeOf (p,c) and isAttributeOf (q,c) :
      if (p.name=q.name) then let(Model.default=inconsistency)
```

To refine the diagnosis, constraints are distinguished according to the type of the default, and therefore the conclusion of the corresponding rules. When the property *default* = {*inconsistency, illegality, incompleteness,* etc} is added to the CC notions of *Constraint* and *Model*, the CC generic rule becomes:

if C.expr *then let* (Model.default = C.default)

The form of the production rule "*if* Expression *then* Action" makes it possible to deduce new facts (Action: *let*), to give modeling advice (Action: *message*), and more generally to input/output data and compose operations. The rules are stored into knowledge bases and processed by the inference engine *Sherlock* [3].

In UML models, we define rules using pseudo-metaclasses that specialize (via stereotypes) the core metaclass *Constraint*. For instance, the pseudo-metaclass *CC_Constraint* specifies common properties such as *constraintLevel*, *constraintFilter*, *constraintDeclaration*, etc. From this root, specific pseudo-metaclasses are defined, each one with its own properties. For instance, to deal with the action *Let*, the pseudo-metaclass *CC_ConstraintLet* requires the property *constraintDefault* to be defined.

In the sequel, the rule form is kept for the sake of readability, but the UML representation is shown in the CC implementation section below.

4.3. Paradigmatic Level (1)

This level defines the semantics of the modeling primitives of the UML. The metamodel includes in particular:

– UML class diagrams that use only basic notions to specify the metaclasses and their relationships. Some adornments such as multiplicity are in fact constraints.
– Well-formedness rules that express invariant property of metaclass instances. These constraints link attributes and associations of the metamodel.

UML modeling tools have to enforce paradigmatic level constraints. Nevertheless, it is always possible to translate constraints into rules as shown above for *Name conflict* ("*no attributes may have the same name within a Classifier*").

4.3.1. Informal Rules

Some constraints of the UML semantics are expressed in natural language with no corresponding formal rule. The burden of the developer is reduced by increasing the number of formal constraints (automatically checked) thus decreasing the informal ones (manually checked). For instance, no rule specifies that the *receiver* of a *Message* should own the *operation* defined in the message *action* when it is a *CallAction*:

```
∀ m ∈ Message,∀ o ∈ Operation,∀ c ∈ Classifier |
  (c = m.receiver) and (o = m.action) and (o.type = 'CallAction') :
      if not ( isOperationOf (o, c) )  then
                              let (Model.default = inconsistency)
```

4.3.2. Unexpressed Rules

Some rules are not expressed in OCL, for instance: "*Tags associated with a ModelElement must not clash with any meta-attributes associated with the Model Element*" [21]. This constraint is useless in current UML modeling tools since meta-attributes are not accessible. Conversely, the CC makes use of them, and no name conflict is allowed:

```
  ∀ c ∈ ModelElement, ∀ t ∈ Tag , ∀ a ∈ Attribute |
      isAttributeOf (a,c) and isTagOf (t,c) :
          if (t.name = a.name) then
                              let (Model.default = inconsistency)
```

4.3.3. Constraints Consistency

UML semantics well-formedness rules are assumed to be consistent. Restrictions can be added as far as compatibility is ensured: added constraints should not conflict with already defined ones, whatever their origin (example in 3.2.). As long as there is a conflict, there is no licit interpretation. This rule cannot be expressed in OCL, while by definition the CC automatically detects such conflicts at meta-metalevel (rule base consistency).

4.4. Paradigmatic Extension Level (2)

To deal with domain-specific applications such as real-time applications (Figure 3), users should agree on the introduced notions and their semantics. For a domain, these "addenda" are gathered together into a *profile*.

4.4.1. Stereotype

A *stereotype* extends an existing metaclass (*baseClass*) or stereotype. Extended elements behave like elements they derived from, with possibly a different representation, additional constraints, and tags. The stereotype *BinaryAssociation* with *baseClass Association* specifies the number of *connection*s:

```
  ∀ e ∈ Element  |  extended (e, BinaryAssociation) :
      if e.connection.size ≠ 2 then
                              let (Model.default = inconsistency)
```

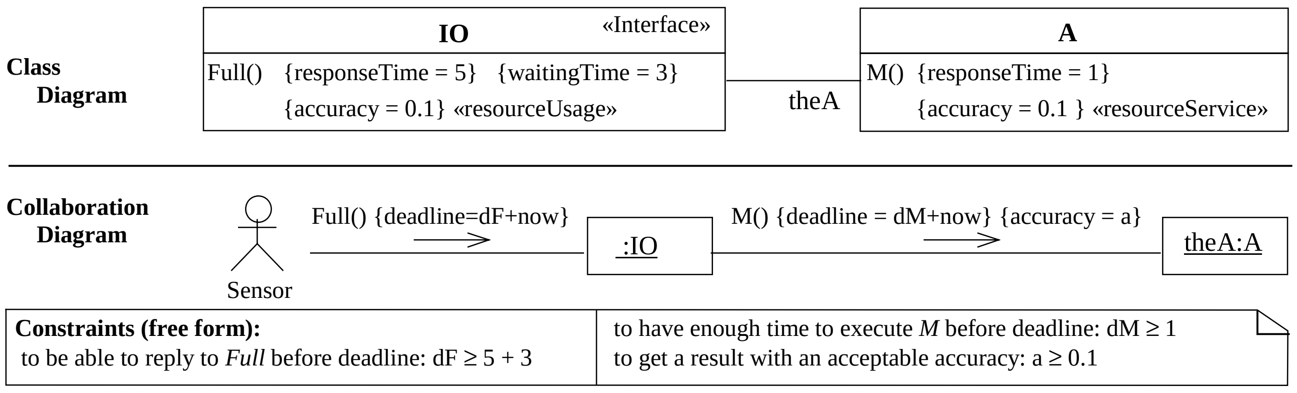
Usually, stereotypes are described at user model level and the metamodel elements cannot be handled since they are in another namespace. CC expressions can hold any metamodel element since, from the CC point of view, the metamodel is a user model. Again, constraints that cannot be expressed in the UML can be checked by the CC.

4.4.2. TaggedValues

A tagged value is a pair (*tag*, *value*) attached to any model element. The interpretation of a *tag* is in charge of the user (or tool). Let assume that an element extended with *RTtimeString* should have a tag whose name is *time* and type *String* (example from the profile *Schedulability, Performance, and Time* [13]). The corresponding rule is:

```
  ∀ e ∈ Element, t ∈ Tag |  extended(e, RTtimeString) :
      if not (∃ t | isTag(t, e) and (t.name = 'time') ) then
                              let (Model.default = incompleteness)
```

Moreover, the *time* value should conform to a syntax such as "hh:mm:ss":



Class Diagram
IO
«Interface»
Full() {responseTime = 5} {waitingTime = 3}
{accuracy = 0.1} «resourceUsage»
theA
A
M() {responseTime = 1}
{accuracy = 0.1 } «resourceService»
Collaboration Diagram
Sensor
Full() {deadline=dF+now}
:IO
M() {deadline = dM+now} {accuracy = a}
theA:A
Constraints (free form):
to be able to reply to Full before deadline: dF ≥ 5 + 3
to have enough time to execute M before deadline: dM ≥ 1
to get a result with an acceptable accuracy: a ≥ 0.1

Figure 3. Examples of target domain constraints expressed using tags and notes in free form.

∀ e ∈ Element, t ∈ Tag |

(t.name = 'time') *and extended* (e, RTtimeString) *and isTag*(t, e):

if not (*hasALegalTimeValueString*(t)) *then*

let (Model.default = Illegality)

4.4.3. Example

Extension features are described as adornments in user diagrams and translated into rules. In the fictitious example Figure 3, the stereotype *Interface* extends the *Class* notion (i.e., the metaclass *Class*). *IO* has a method *Full* with tags such as *responseTime* and *accuracy*, the latter being marked with a stereotype. Here are constraints expressed in natural language:

– Some model element has to define a tag. For instance, an event (*Full*) coming from an actor (external entity of the studied system) should have a *deadline*.
– The meaning of tags such as *deadline* or *responseTime* is unambiguous since implicitly a *deadline* is a request for a quality of service while a *responseTime* is related to a service. Tags such as *accuracy* should be marked with stereotypes («*ressourceService*»/«*resourceUsage*») to precise their meaning.
– A stereotype may extend any model element inheriting from its *baseClass*, even when it does not make sense. To avoid misuse, stereotype attachment is restricted to some metaclasses, e.g., «*resourceUsage*» can only extend the metaclasses *Action*, *Classifier*, etc.

4.5. Modeling Domain Level (3)

This level customizes the formalism for a class of applications concerning modeling principles or any specific need, and adds modeling norms and standards.

4.5.1. Specific Constraints

When modeling for *Java*, templates (genericity) and multiple inheritance are forbidden. This leads to:

∀ a, b, c ∈ Class:

if isKindOf (b , a) *and* isKindOf (b , c) *then*

let (Model.default = Illegality)

4.5.2. Style Guide

In any language, style guides prove to be useful. User guides (such as [2]) give tips that may be expressed using constraints. A few examples are given Figure 4:

a In this class diagram, *Room*s and *Client*s without *Reservation* are not accessible from *Hotel*. Since rooms and clients exist in the absence of hotel reservation (multiplicity 0), associations from *Hotel* to *Client* and *Room* should be added. The first time, the CC warns the user, asks a question and keeps the answer to prevent from asking the same question later.

b The state diagram at the left can be rewritten in the right form to take advantage of the hierarchy capabilities. If the user agrees, the diagram would be replaced automatically.

c This sequence diagram (time from top to bottom) has two external stimuli: *On* and *Alarm*. It is not possible to know if *Reply* will occur before or after *Alarm*. Therefore the time chronology of the diagram is not ensured, and the stimuli should be processed in separate diagrams.

4.6. Target Domain Level (4)

This level involves objects from the real world domain and gives sense to notions defined within the user application. At this level, most of the constraints are dynamic and can only be checked at run time (e.g., salary>1000$), except when instances and values are known. In the example Figure 3, the collaboration diagram describes the interactions between instances and the exchanged messages from the initial stimulus. The *Sensor* sends the event *Full* to an instance of *IO* that in turn sends a message *M* to *theA*, that is the instance of *A* associated with the instance of *IO*.

At modeling domain level, constraints ensure tags completeness, e.g., *Full* has a *deadline*. This lower level enforces constraints related to the user model that cannot be described at previous level, for instance: dF ≥ $\text{responseTime}_{\text{Full}}$ + $\text{waitingTime}_{\text{Full}}$ (constraints in the note Figure 3).

4.7. Implementation Level (5)

Implementation constraints depend on target language and tools. Most UML tools enforce constraints at syntactic level, e.g., the state diagram should be deterministic, concurrent states are forbidden, etc. As models are under-specified, decisions are taken by default, e.g., an association is translated into a pointer in C++. To change this behavior, UML tools provide a concrete syntax to add tags that will guide the translation into a programming language.

At this level, the main goal of the CC is to verify consistency and completeness rules. These rules are described into specific

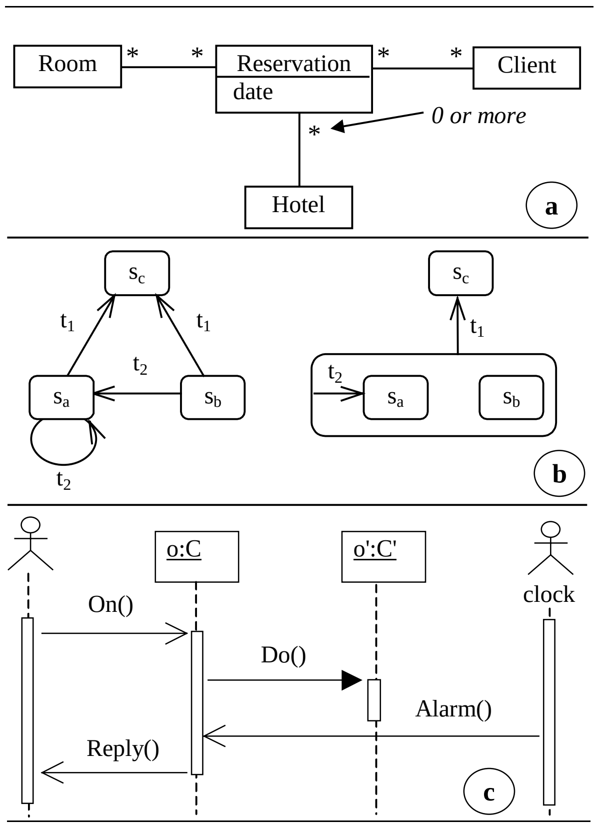


Figure 4. Style guide examples

knowledge bases according to the language, the tool etc. Further, diagrams that are not translated into code, such as interaction diagrams, are scanned. For instance, object communication is a recurrent problem for beginners, and it is very useful to check that each message sender knows the receiver, that associations are navigable in the sense of the declared interactions, etc.

5. CONSTRAINT CHECKER IMPLEMENTATION

The CC is mainly composed of the inference engine *Sherlock* [3] linked with a UML CASE tool (Figure 5). Models are built using the UML modeling tool and adorned with tags and constraints. Due to the negative form required by rules (if the invariant is false then the model is erroneous, see 4.2.), the constraints are not expressed in OCL but in a form easier to parse. Using the stereotype «*CC_ConstraintLet*» that extends the standard UML *Constraint* for the *Let* action, the constraint "Name conflict" is described as follows (syntax: {tag.name = tag.value }):

«*CC_ConstraintLet*»
Name conflict

{constraintLevel = paradigmatic}
{constraintDeclaration = ∀ p, q ∈ Attribute, ∀ c ∈ Classifier}
{constraintFilter = (p≠q) *and isAttributeOf* (p,c) *and isAttributeOf* (q,c) }
{constraintExpression = p.name = q.name }
{constraintDefault = inconsistency }

As there is no specific graphical representation for constraints and tags, the chosen way is to define them into notes associated with model elements.

For the UML tool and the CC to cooperate there was two main architectures: to link tools directly using a standard access protocol (e.g., DCOM), or else to communicate using a standard format (e.g., XML [22]). The former is more integrated but tool-dependant, while the latter is more general but requires a parser and makes it difficult to act on models inside the UML tool (arrow *Action* Figure 5). We have chosen the latter solution and developed an additional tool to translate XML files into CC ones.

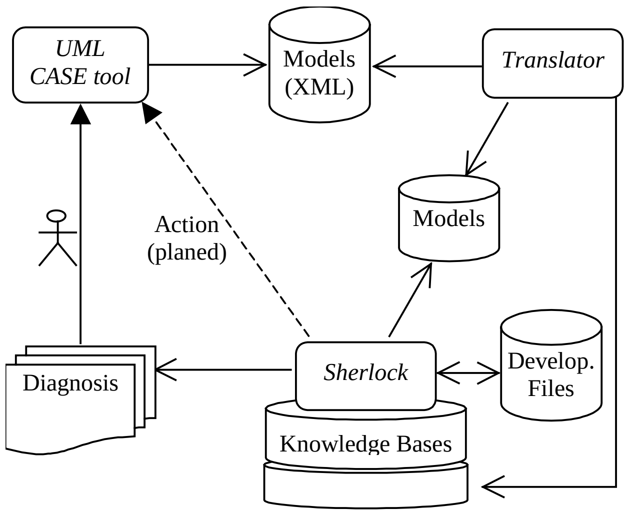


Figure 5. The CC process steps and components

The next question was the description of the UML metamodel within the CC. The MOF (Meta-Object Facility [12]) is the language adopted by the OMG to specify metamodels. A general solution would have been to describe the MOF in the CC, and then to add the UML metamodel on top of the MOF. As a lighter alternative, we have chosen to describe the UML metamodel directly using the inference engine language. It is based on notions such as concept, attribute, relationship, instance, task, and rule, which make it possible to implement quickly a basic metamodel. Next, models are expressed in terms of the metamodel. Unlike the four-layer metamodeling architecture [21], languages defined at lower layers are enabled.

In its current state, the CC includes knowledge bases corresponding to the constraint levels, each one lying on the lower ones. Since dependency cycles are not allowed, the definition order is very important. The starting layer describes UML metamodel notions and generic rules in inference engine terms (e.g., "any model element marked with a stereotype should define the tags required by this stereotype"). Then, various knowledge bases describing the profile notions and rules are added (e.g., "the value of the tag *constraintDefault* should belong to the set {*inconsistency*, *illegality*, *incompleteness*, etc}"). These rules might be described in the UML, but it is simpler to write them directly in the CC. Finally, the UML model is loaded in the CC. Again, the order is important to avoid forward references. The constraints are first added in a knowledge base, and then the model is checked.

The CC Figure 5 is currently under development. It works well for a few examples and grows progressively. Models are automatically translated into XML files by the UML tool. Then, using a XML parser library, the *Translator* translates data into either knowledge base or specific files, depending on the data nature. Next, the needed knowledge bases are loaded into the CC, and finally the models are checked and a diagnosis is given. During the process, the CC asks the user further precision about models and stores replies in *Development-specific files*, e.g., "the detected warning should it be displayed again?". The user is in charge of model improvement, and the process cycle may start again. In the future, we aim to get a CC complete enough to be usable in a limited context.

6. RELATED WORKS

This work is close to the *precise UML group* works [15] that aim to define formally UML [4]. In the mapping *Model* to *Interpretation*, a precise semantics draws clearly the limits of the licit interpretations. Then, additional constraints remove the undesirable interpretations in the modeling domain, and all those that have no meaning in the modeled or implementation domain. Thus the CC takes place later in the process and assumes a formal definition of the UML.

This work is also close to approaches aiming to verify properties on UML models by translating them into formal languages (e.g., B, ASM [15][8][11][20]). Specific properties that cannot be expressed as simple rules are verified (e.g., deadlock detection, temporal behavior validation). It is clear that the CC role should be limited to constraints based on rules.

Next, this work is close to the works about OCL constraints ([6][7][17][18]). Although OCL is included into UML, UML tools generally do not check OCL constraints. To make up for these shortcomings, numerous attempts have been made to check directly

OCL constraints [18]. The aim of this work is to go further: to verify well-formedness rules that cannot be expressed in OCL, to manage and share constraints, to give advice, to correct errors, and to help during the modeling process.

Finally, this work is related to software development assistance (e.g., [1][10]), with a focus on modeling. Unlike these works (or [6] that defines a new constraint diagram), a major limit was to remain UML compatible: the CC is only a companion tool that may be revoked at any time. It acts on models only through standard extension mechanisms, thus any UML tool can be used (via the standard translation from UML to XML [12]).

7. CONCLUSION

The semantics of modeling languages first defines the licit model interpretations. Then constraints are added to restrict these interpretations to the desirable ones only. Therefore, it is fundamental to make clear how constraints define semantics. The entire UML constraints, including the metamodel well-formedness rules, are classified into five levels:

– The *paradigmatic* level in which the semantics of the primitives of the formalism is defined,

– The *paradigmatic extension* level in which the semantics of the extensions is defined,

– The *modeling* level that customizes a formalism for a specific application domain or process and enforces the style guide,

– The *target domain* level that defines the meaning of the notions defined within the user application,

– The *implementation* level that ensures that the translation into a given target language is possible.

Whenever a constraint is not satisfied, there is a modeling default. These constraints are particularly useful for beginners to enforce the style guide.

OCL is a pure expressive, side-effect free language. It aims at verifying invariant on model instances, while the needs are rather to supply recommendations and modeling tips, to complete and to improve models. The advocate solution is to associate a knowledge-based environment (constraint checker) with a UML modeling tool. This solution introduces both flexibility and capabilities. The constraints can be managed in a more reusable and rational way, while the checking process can be parameterized according to the application domain, the modeling process, the target language or anything else like the level of the user. Unlike using OCL, the diagnosis can be adapted, missing information added, improvements proposed. Moreover, constraints that cannot be expressed in the UML can be checked. Finally, modeling is a human process subject to improvements and evolutions, and the knowledge-based approach, thanks to its great modification abilities, provides the easiest way to deal with.

REFERENCES

[1] C. Balzer, T. Cheatham, C. Rich, "Report on a knowledge-based software assistant", *A.I. and Software Engineering*, Morgan Kaufman publishers, 1986

[2] G. Booch, I. Jacobson, J.s Rumbaugh, *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, 1998

[3] G. Caplat, "Sherlock Environment", //servif5.insa-lyon.fr/chercheurs/gcaplat/

[4] A.S.Evans, S.Kent, "Meta-modelling semantics of UML: the pUML approach", UML'99, LNCS 1723, 1999, 140-155

[5] D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics and All That Stuff", TR MCS00-16, The Weizmann Institute of Science, 2000. Available on www.cs.york.ac.uk/puml

[6] S. Kent, "Constraint diagrams: visualising invariants in OO models", OOPSLA' 97, ACM Press, 327-341

[7] S. Kent and Y. Gil, "Visualising action contracts in OO modelling", *IEEE Software Engineering Journal*, 1999.

[8] K.C. Lano and A.S. Evans, "Rigorous Development in UML", FASE'99, LNCS, 1999

[9] L. Mandel, M.V. Cengarle, "On the Expressive Power of the Object Constraint Language OCL", FM'99, LNCS 1708, 1999, 854-874

[10] J. Mylopoulos, L. Chung, B. Nixon, "Representing and using nonfunctional requirements: a process-oriented approach", *IEEE Trans. on Software Engineering*, Vol. 18(6), 1992, 483-497

[11] I. Ober, "More meaningful UML Models", *TOOLS - 37*, IEEE Press, 2000, 146-157

[12] www.omg.org/

[13] OMG, "Schedulability, Performance and Time" Reply to the RFP of the OMG, ARTiSAN Software Tools, I-Logix, Rational Software Corp., Telelogic AB, TimeSys Corporation, Tri-Pacific Software, 2000

[14] B. Peuschel, W. Schaefer, S. Wolf. "A knowledge-based software development environment supporting cooperative work", *I.J.S.E.K.E.*, Vol.2(1), 1992, 79-106

[15] http://www.cs.york.ac.uk/puml/

[16] H.B. Reubenstein, R.C. Waters, "The requirements apprentice: automated assistance for requirements acquisition", *IEEE Trans. on soft. eng.*, Vol.17(3), March 91, 212-225

[17] M. Richters, M. Gogolla, "Validating UML Models and OCL Constraints", UML 2000, LNCS 1939, 2000, 265-277

[18] M. Richters, M. Gogolla. "OCL - Syntax, Semantics and Tools". In *Advances in Object Modelling with the OCL*, LNCS 2263, 2001, 43-69

[19] J.L. Sourrouille. "A Knowledge-based Framework for O-O software development environments", *I.J.S.E.K.E.*, Vol.4(4), 1994, 451-479

[20] J.L. Sourrouille, "UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages", *UML'99*, LNCS 1723, 1999, 457-472

[21] UML, "OMG Unified Modeling Language Specification (draft)", Version 1.3, March 1999

[22] W3C (World Wide Web Consortium). Extensible Markup Language (XML), 1998