Building an evaluation instrument for OO CASE tool assessment for Unified Modelling Language support.

Radmila Juric [1,3]
Jasna Kuljis [2]

[1] South Bank University Business School
[2] Department of Mathematical and Computing Sciences, Goldsmiths College, University of London
[3] Radmila Juric now works within the School of Informatics, University of Westminster

Building an Evaluation Instrument for OO CASE Tool Assessment for Unified Modelling Language Support

Radmila Juric
Business School
South Bank University
103 Borough Road, London SE1 OAA, UK
Tel: +44(0)171 815 7815
Fax: +44(0)171 815 7793
E-mail: juricr@sbu.ac.uk

Jasna Kuljis
Department of Mathematical and Computing Sciences
Goldsmiths College, University of London
New Cross, London SE14 6NW, UK
Tel +44(0)171 919 7868
Fax +44(0)171 919 7853
E-mail: j.kuljis@gold.ac.uk

Abstract

The Unified Modelling Language (UML) as delivered in September 1997 offers the structure and dynamics of its modelling constructs developed in order to standardise different object oriented (OO) development practices. Represented as a language, UML covers some aspects addressed by any methodology and is expected to be accompanied by OO CASE tools through notation and implementation of the UML philosophy. This paper discusses the problem of OO CASE tools as methodology companions that encourage or enforce methodology support. The basis for an evaluation instrument has been developed in order to analyse how commercially available OO CASE tools support the UML. The evaluation instrument is based on extraction of a set of rules that are supposed to be followed in order to claim that the UML itself is being followed. The rules are extracted from the current UML Semantics document and its well-formedness rules. The evaluation instrument is tested against a few OO CASE tools in order to analyse how it can be used on a larger scale for assessing the level of automation and UML support embedded in the tools.

1. Introduction

In the last decade we have witnessed the significant influence of the OO paradigm (Booch, [1]) in solving problems encountered when developing complex software systems. The result is a shift towards the OO software development process described in different OO methodologies (Rumbaugh et al., [22]; Booch, [1]; Yourdon, [28]) and a plethora of new OO programming languages and environments that support the new technology. In order to standardise different OO practices, the process of UML definition was initiated four years ago. Despite being a language that can accompany any OO development process, UML covers many aspects addressed by any OO methodology. As it has become a standard language by the decision of the Object Management Group (OMG), the UML is expected to take a leading role in information systems development within the OO community.

The problem of choosing the right OO CASE tool in the OO development process has been a serious obstacle for OO practitioners. CASE tools have been playing a role of 'methodology companions', i.e. when buying a tool we were very often buying a methodology! This might not be an issue any more if we have agreed that the UML is a standard modelling language. However, it is still important to know how and to what extent OO CASE tools support the UML. This leads towards creating an evaluation instrument that can measure the level of the UML support in each OO CASE tool.

This paper attempts to analyse this problem and build the basis for such an evaluation instrument. The instrument is tested against a few OO CASE tools in order to analyse how it can be used on a larger scale for assessing the level of automation for the UML support embedded in the tools.

Section 2 discusses what methodologies for information systems development are supposed to address. A set of *rules* that represent a philosophy of the methodology is expected to be embedded in CASE tools if they offer automated support of the methodology. From this aspect section 2.1 discusses CASE tools as methodology companions.

Section 3 covers the problem of building an evaluation instrument for the UML support, and starts with an overview of some CASE tool evaluation frameworks and standards. Section 3.1 gives guidelines on how UML *rules*, that contribute to the creation of such an instrument, are to be found. Section 3.2 critically assesses the UML well-formedness rules and discusses inconsistencies found in the UML Semantics [24], giving some examples. Section 3.3 addresses the process of extraction of the UML *rules* and the way of representing them. All rules found in the Core Elements and the Extension Mechanisms of the UML Foundation Package [24] are listed in sections 3.4 and 3.5. In section 3.6 the evaluation instrument is defined and the application of its simplified version is suggested.

Section 4 describes the method and results of testing the evaluation instrument when applied to a few commercially available OO CASE tools. Implications of the evaluations are discussed in section 4.3. Section 5 gives conclusions and guidelines for future research.

2. Methodologies for Information Systems Development

The development of information systems is concerned with the definition of a set of *general* steps and activities in order to produce an effective information processing system (Jayaratna, [5]). A methodology, as defined in Jayaratna [5] (page 35) for information systems development must further elaborate this definition and represent a *specific* rather than general way of performing the systems development (Peters, [21]; Wainwright et al., [27]). It should give a detailed and specific structure on

(i) what *steps* are regarded as essential,
(ii) in what order they should be performed,
(iii) how they could be carried out, and
(iv) which products/models will result from them.

If we want to claim that a particular methodology is being used, we need to ensure that all steps of the methodology are followed in the prescribed order. Furthermore, methodologies are trying to help us in structuring our ‘thinking’ from the perspective of user and developer. This implies different philosophical assumptions of ‘reality’ to be modelled (Jayaratna, [5]). The result is a different philosophy and approach to the development process. Whatever the differences among various methodologies are, they are expected to define two primary aspects in the systems development: *process* and *product*. The former deals with the approach to (and employment of) many different techniques to manage an extremely complex task of systems development. The example is a top-down approach and functional decomposition of the analysis process in structured methodologies (De Marco, [3]; Sarson and Gane, [23]). The latter deals with different deliverables as results of applying a particular technique or undertaking a particular phase/stage of the development process. Both aspects: *process* and *product* could be further elaborated towards issues that every methodology is expected to address (Juric, [9]):

(a) The coverage of an SDLC and its role within an analysis and design,
(b) The set of concepts and guidelines that define all modelling elements,
(c) The set of *rules* that represent the philosophy of the methodology and are supposed to be followed,
(d) Fully described deliverables and their employment within different models, and
(e) Notation acceptable for both users and developers.

Not many methodologies cover all these requirements, and many users/developers face difficulties when applying structured and OO methodologies in order to clearly recognise (a)-(e) above. If we want to claim that a particular methodology is being used, then (c) above, i.e. *rules* of the methodology, are expected to be followed. It is not unreasonable to assume that CASE tools, which offer automated methodology support, should also have at least (c) above, embedded in them.

2.1 CASE Tools as Methodology Companions

CASE tools are expected to offer automated assistance in the development process that significantly contributes towards improving software productivity. The first and second generation of CASE tools addressed all problems when supporting *processes* of the system development life cycle. This includes the creation of *products* in the form of various diagrams and performing consistency, completeness and correctness checking. However, CASE tools are still deficient in a number of important areas such as providing support for defining new methodologies, or providing an exchange of *products* resulting from different *processes* expressed in different methodologies (Mehandjiska et al., [16]). The research into the development of an open architecture for software environments has addressed this problem and should result in the integration of independently developed CASE tools (Lang, [11]; Nilsson, [18]; Papahristos and Gray, [20]; Mehandjiska et al., [15]). However, the question of CASE tools methodology dependence remains a very important issue that has been addressed not only by researchers, but also by CASE tools vendors and OO practitioners.

If we consider CASE tools as methodology’s companions, as defined in (McClure, [14]), we expect them to implement a methodology support. This means that they should assist us in following all *rules*, applied to *process* and *products*, as prescribed by a methodology. The support from CASE tools is provided through an automated control of *rules* and an informative feedback on the violations of *rules* (Hatley, [4]). Any violation of *rules* may result in the CASE tool preventing a user/developer from proceeding in a non-methodological way or simply issuing a warning/error message. This determines the basic underlying philosophy of CASE tools as methodology companions, as discussed in Hatley [4], Vessey [25] and Juric, [6].

The *process* support from CASE tools should include a guideline, encouragement or enforcement of using prescribed steps defined by a methodology. Support for *products* should assist in the creation of different diagrams that are supposed to be as accurate/complete as possible given the rules of a methodology. This includes both internal and inter-product consistency checking. Hence, 'checks' on a methodology's *process* and *products* are the most important component embedded in the CASE tool. They form the basis for any evaluation instrument that can determine the extent to which a CASE tool supports a particular methodology.

3. Building an Evaluation Instrument for UML Support

Many different works have addressed the problem of CASE tools evaluation through formal and informal CASE tools comparison and evaluation frameworks such as Marttiin et al. [13], Mosley [17], Mehandjiska et al., 16], Vessey et al. [25], Vessey and Sravanapudi [26] and Juric [6]. Not many of them explicitly address OO CASE tools and they are not appropriate for the evaluation of UML support. The project described in Mehandjiska et al. [16] discusses the evaluation framework that acknowledges the variety of different OO CASE tools available today. The framework is based on an inheritance hierarchy of CASE tools categories and contains more than 80 carefully selected questions. Some questions are adopted from Mosley [17] and all of them are classified according to the nodes of the CASE tools hierarchy. This allows the classification of questions along with the various types of tools, hence the evaluation framework can be easily extended.

The standards for classifying and evaluating CASE tools described in Mosley [17] include six categories of questions designed to determine how well a tool does what it was intended to do. The assessment instrument is the generic set of 140 questions plus a tailored set of functional questions (the number varies from 30 to 100) specific to the type of a tool being evaluated. Many questions addressed tools' functionality and the quality of *process* and *product* support.

The evaluation instrument from Vessey et al. [25] contained questions in the form of *rules* that covered many aspects of structured analysis and design with entity relationship modelling, including pair-wise interrelationships between the techniques. A different evaluation instrument defined in Juric [6] was created before the development of the UML had been initiated. It contained a set of *rules* that originated in four different OO methodologies: OMT (Rumbaugh et al., [22]), Booch (Booch, [1]), Shlaer-Mellor (Lang, [12]) and Coad-Yourdon (Yourdon, [28]; Coad and Yourdon, [2]). This evaluation instrument proved to be very difficult to apply and interpret due to the lack of standardised OO modelling constructs at that time, and some conflicting issues among mentioned methodologies (Juric and Snaith, [7]).

3.1 UML Rules

This paper attempts to demonstrate how such an evaluation instrument could be created to determine the extent to which commercially available OO CASE tools support the UML. The first step in developing an evaluation instrument is to extract a set of *rules* for all UML modelling constructs. The *rules* should constitute a list of all UML 'checks' that support the development *process* and *products*. However, the UML is defined as being process free and not including any process/strategy definitions. It is represented as a language that can support any methodology for OO systems development. Despite that, the UML covers many aspects addressed by a methodology, stated in (b)-(e) from section 2. This helps in the extraction of all *rules* that play a role of 'checks' when applying the UML.

All *rules* (particularly (c) from section 2) in previous versions of the UML had to be extracted manually from its semantics and notation documents (Juric, [8] and [9]). In the current UML [24] we can see them better defined in the form of *well-formedness rules* as one of three different views of the UML metamodel. However, there are some obstacles and inconsistencies in their specification, which are also discussed in Juric and Song [10]. The examples are given in the next section. This is the reason why it has not been possible to collect all well-formedness rules defined in the Semantics document [24] and use them straightforwardly as part of an evaluation instrument.

3.2 UML Well-formedness Rules

The UML authors explicitly define in the current UML [24] well-formedness rules given in OCL expressions (Object Constraint Language Specification supplement, [24]). The well-formedness rules are part of static semantics defined in the UML, which are to be fulfilled by well-formed modelling constructs from dynamic semantics (described as *abstract syntax* under the headings Semantics). Some of the well-formedness rules like 'multiplicity' and 'ordered' constraints on relationships are defined in diagrams which are part of abstract syntax, showing all modelling constructs and their relationships. All these rules are defined as a set of *invariants* of an instance of a metaclass that are to be satisfied for the construct to be meaningful (page 11 from Semantics, [24]). The well-formedness rules specify all constraints over attributes and associations of each modelling construct in informal explanation and OCL expression. However, a detailed analysis of the Semantics document [24] revealed many discrepancies regarding what is represented in the well-formedness rules and how they support the abstract syntax.

The CORE Foundation Package [24] suffers from many inconsistencies in abstract syntax, particularly when addressing static and dynamic rules. Here are some examples.

(1) An Association is the first modelling construct represented in abstract syntax. Its definition in informal language explicitly states possible rules regarding AssociationEnds: *"An Association has at least two AssociationEnds"*, or *" The same Classifier may be connected to more than one AssociationEnds in the same Association"*. Both of these *rules* can be directly mapped from the diagram (page 17, figure 6 from Semantics, [24]), which is justified by the multiplicity constraints being represented within an abstract syntax. However, these *rules* are not present (or repeated) within the well-formedness rules section (pages 27 and 28 from Semantics, [24]) as we expect. Furthermore, the first rule is repeated within 'connection' (which is the association section of the Association construct's abstract syntax). This problem shows the inconsistency in abstract syntax between the short informal description and the short explanation of the modelling construct's attributes, and opposite role names of associations (connected to the modelling construct itself). It also raises the question of "what" is included in well-formedness rules if such an important rule as the one above is omitted from them.

(2) There are more *rules* (this time not related to multiplicity and ordered constraint) which are found within a short informal description of a construct's abstract syntax and not represented within well-formedness rules. An example is the definition of an Association construct that includes the sentence *"Each tuple value may appear at most once"*, or an AssociationEnd construct which ChangeableKind attribute 'frozen' specifies that *"No links may be added after the creation of the source object"*. Both of these could be very important *rules* to follow, but they can not be found within any well-formedness rules specified later within the same chapter. This problem (including the example from (1) above) is the result of a strict division of the UML syntax into static (well-formedness rules) and dynamic (abstract syntax) which immediately allows some 'static rules' (like multiplicity and ordered constraints) to be declared within the 'dynamic section'. It also allows some dynamic issues to be represented in the form of *rules* that are definitely as important as well-formedness rules, but do not find their place within them.

(3) Furthermore, the only attribute of the Association construct named 'name' is represented in abstract syntax through the rule: *"The name of Association which, in combination with its associated Classifier, must be unique within an enclosing namespace (usually a Package)"*. This gives a clear guideline for name uniqueness that is also expected within well-formedness rules. It is not found within an Association construct's well-formedness rules (which is expected) but within the well-formedness rules of a Namespace construct (page 33 from Semantics, [24]). This somehow contradicts the authors' decision to allow the expression of all current semantics of a modelling construct in its superclass. It should be a ModelElement (page 17, figure 6 from Semantics, [24]) for an Association construct where we can impose a constraint on its 'name' attribute. A very experienced practitioner can trace the name uniqueness rule easily after reading the Namespace abstract syntax where *"A Namespace is a ModelElement and can own other ModelElements"* and after reading well-formedness rule [4] for Association. However, this can be a confusing issue for any attempt to quickly map abstract syntax represented in different diagrams and informal language, with corresponding well-formedness rules. This problem is a consequence of the weak structure and connection between static and dynamic semantics of the UML modelling constructs and again points out the role and importance of well-formedness rules.

All points above: (1), (2) and (3) are related to the Core Package of the UML Semantics [24]. The Extension Mechanism modelling constructs, as a part of the Foundation Package, show similar weaknesses in representing well-formedness rules as the Core package. However, all components of the Behaviour Element Package, (Collaborations, Use Cases and State Machine) exhibit more stable presentation of well-formedness rules. Their abstract syntax does not contain any sentence in the form of *rules* i.e. it contains definitions only. One has to assume that all necessary *rules* regarding the UML behavioural modelling constructs are contained in their well-formedness rules.

3.3 Extracting UML Rules

In order to address the problems discussed in section 3.2, a detailed analysis of Core and Extension Mechanism modelling constructs of the Foundation package [24] was required. The task of extracting all UML *rules* consisted of collecting all important *rules* found outside well-formedness rules (from 'abstract syntax' and 'semantics') and incorporating them into the existing set of well-formedness rules. There are 67 *rules* from the Core elements listed in section 3.4 and labelled with CE. Section 3.5 shows that only 15 *rules*, labelled with EM, are extracted from the Extension Mechanism package. When listing all *rules*, bold print is used to note the introduction of a particular modelling construct; and italic is used to specify the attribute name of a particular modelling construct. Attribute values for a given modelling construct are in upper case except 'frozen' and 'addOnly', which are in inverted commas. All modelling construct names start with an upper-case letter and are spelled the same way as in the UML document [24]. All *rules* are grouped according to the semantics of the modelling construct they represent. Sometimes the additional letters a-d are needed in order to emphasise this grouping.

If we want a list of all 'checks' that support all the UML modelling constructs then we have to add the *rules* from sections 3.4 and 3.5 to the set of 106 well-formedness rules for behavioural modelling constructs defined in the Behavioural Element package of the Semantics document [24].

3.4 Foundation Package: CORE Elements Rules

CE1 **Association** may be defined between two classifiers which are not Interface or DataType (unless a DataType is part of a composite aggregation).

CE1a The name of Association (in combination with associated Classifier) must be unique (within enclosed Namespace).

CE1b Association must contain at least TWO AssociationEnds.

CE2 The name of **AssociationEnd** must be unique within an Association.

CE2a Each AssociationEnd is part of one Association only.

CE2b The same Classifier may be connected to more than one AssociationEnd within the same Association.

CE3 At most one AssociationEnd may be aggregation and composition.

CE3a If Association has 3 or more AsscoiationEnds than no AssociationEnd may be an aggregation /composition.

CE3b The 'part' of an aggregate may be contained in other aggregates.

CE3c The 'part' owned by the composite may not be 'part' of any other composite.

CE4a If *changeable* attribute of an AssociationEnd is specified as 'frozen' no links may be added after the creation of the source object.

CE4b If *changeable* attribute of an AssociationEnd is specified as 'addOnly' links may be added at any time from the source object, but once created a link may not be removed before at least one participating object is destroyed.

CE5 If *ordering* attribute of an AssociationEnd placed on the target end of Association, is specified (AssociationEnds are ordered within Association) the ordering must be determined and maintained by Operations that add links.

CE6 **AssociationClass** can not be defined between itself and any other Classifier.

CE6a The names of the AssociationEnds and StructuralFeature of AssociationClass do not overlap.

CE7a If *changeable* attribute of an **Attribute** is specified as 'frozen' the value may not be altered after the object is instantiated and its value initialise. No additional values may be added to a set.

CE7b If *changeable* attribute of an Attribute is specified as 'addOnly'(assuming that multiplicity is not fixed to a single value) additional values may be added to a set of values. However, once created a value may not be removed or altered.

CE8 No objects can be instantiated from an abstract **Class**.

CE8a If Class is concrete all the Operations of Class should have a realising method in the full descriptor.

CE9 Objects instantiated from a Class do not contain values corresponding to BehaviouralFeatures or class-scope Attributes, but all Objects of a Class share the definitions of the BehaviouralFeatures from the Class.

CE10 Class may realise zero or more Interfaces (its full descriptor must contain every Operation from every realised Interface).

CE10a One Interface may be offered by more than one Class.

CE11 For each an Operation in an Interface provided by the Class, the Class must have a matching Operation.

CE12 A Class can only contain Classes, Associations, Generalisations, UseCases, Constraints, Dependencies, Collaborations and Interfaces as a Namespace.

CE13 No BehaviouralFeature, of the same kind may have the same signature in a **Classifier**.

CE13a No Attributes and no opposite AssociationEnds may have the same signature within a Classifier.

CE13b The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier.

CE13c The name of an opposite AssociationEnd may not be the same as the name of an Attribute or a ModelElement contained in the Classifier.

CE14 An **Interface** can only contain Operations

CE14a An Interface can not contain any Classifier.

CE15 All Features defined in an Interface are public.

CE16 An Interface can not be used as the type of a parameter.

CE17 **GeneralisableElement** may only be a subclass of GeneralisableElement of the same kind.

CE18a If *isAbstract* attribute of GeneralisableElement is set to TRUE then the Generalisable element is incomplete (abstract) and is not instantiable.
CE18b If *isAbstract* attribute of GeneralisableElement is set to FALSE then the Generalisable element is complete (concrete).

CE19a If *isLeaf* attribute of GeneralisableElement is set to TRUE then the Generalisable Element is descendent and may not add descendents.
CE19b If *isLeaf* attribute of GeneralisableElement is set to FALSE then the Generalisable Element may add descendents whether or not has any descendent at the moment.

CE20a If *isRoot* attribute of GeneralisableElement is set to TRUE then the Generalisable element is ancestor and may not add ancestors.
CE20b If *isRoot* attribute of GeneralisableElement is set to FALSE then the Generalisable element may add ancestors whether or not has any descendent at the moment.

CE21 Circular inheritance is not allowed.

CE22 The supertype must be included in the Namespace of the GenralisableElement.

CE23 **Generalisation** (as a directed inheritance relationship) can contain a discriminator which name need not be unique (the empty string is considered just as another name).

CE24 Every ModelElement except a root element must belong to exactly one **Namespace** (the pathname of Namespace names starting from the system provides a unique designation for every ModelElement).

CE25 If a contained element which is not Association or Generalisation has a name then the name must be unique in the Nmaespace.

CE26a If *isQuery* attribute of **BehaviouralFeature** is set to TRUE an execution of the feature leaves the state of the system unchanged.
CE26b If is*Query* attribute of BehaviouralFeature is set to FALSE an execution of the feature indicates that side effect may occur.

CE27 The entire signature of BehaviouralFeature (name and parameter list) must be unique within its containing Classifier (All Parameters should have unique names).

CE28 Each Operation must contain a list of values that are compatible with the types of the Parameter.

CE29 The type of the **Parameter** should be included in the Namespace of the Classifier.

CE30a If *ownerScope* attribute of **Feature** is set to INSTANCE then each Instance of the Classifier holds its own value for the Feature.
CE30b If *ownerScope* attribute of Feature is set to CLASSIFIER then there is just one value of the Feature for the entire Classifier.

CE31a If a *visibility* attribute of Feature is set to PUBLIC than any outside Classifier with visibility to the Classifier can use the Feature.
CE31b If a *visibility* attribute of Feature is set to PROTECTED than any descendent of the Classifier can use the Feature.
CE31c If a *visibility* attribute of Feature is set to PRIVATE than any the Classifier itself can use the feature.

CE32 An **Operation** as a BehaviouralFeature (applied to Instances of the Classifier) must have a signature (possible parameters and return values).

CE33a If a *concurrency* attribute (semantics of concurrent calls to a passive instance, *isActive*=FALSE) of an Operation is set to SEQUENTIAL, only one call to an Instance (on any sequential Operation) may be outstanding at once.
CE33b If a *concurrency* attribute of an Operation is set to GUARDED, only one call of all calls occurring simultaneously is allowed to commence. The others are blocked until the performance of the first Operation is complete.
CE33c If a *concurrency* attribute of an Operation is set to CONCURRENT all calls may proceed concurrently with correct semantics. Concurrent Operations must perform correct in the case of simultaneous sequential or guarded Operations (or concurrent semantics can not be claimed).

CE34a If *isPolymorphic* attribute of Operation is set to TRUE then Methods can be defined on subclasses.
CE34b If isPolymorphic attribute of Operation is set to FALSE then the Method realising the Operation in the current Classifier is inherited unchanged by all descendents.

CE35 The Operation must be owned by the Classifier that owns the Method or be inherited by it.

CE36 The signature of the Method should be the same as the signature of the realised Operation.
CE37 The visibility of the Method should be the same as for the realised Operations.

CE38 In **Dependency** relationship the presence of *client* ModelElement requires the presence and the knowledge of the *supplier* ModelElement.

CE38a Both client and supplier ModelElements must exist at the same level of abstractions.

CE38b A Dependency indicates a semantic relationship among ModelElements themselves rather than their Instances.

3.5 Foundation Package: EXTENSION Mechanisms Rules

EM1 **Constraints, Stereotypes** and **TaggedValues** can be applied to ModelelEments referred as baseClass in the UML metamodel and can not be applied to Instances.

EM2 Any ModelElement can be marked by at most on Stereotype, but any Stereotype can be constructed as a specialisation of numerous Stereotypes.

EM2a The presence of a Stereotype may impose implicit Constraints on the ModelElement and may require the presence of specific TaggedValues.

EM3 A ModelElement may have a set of Constraints.

EM3a The Constraint is to be evaluated when the system is stable.

EM4 A Constraint attached to a Stereotype must not conflict with Constraints on any inherited Stereotype, or associated with the baseClass.

EM4a A Constraint attached to a stereotyped ModelElement must not conflict with any Constraints on the attached classifying Stereotype, nor with the Class (the baseClass of the ModelElement).

EM4b Constraint attached to a Stereotype will apply to all ModelElements classified by that Stereotype and must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the baseClass) of the ModelElement.

EM5 Stereotype names must not clash with any baseClass name.

EM5a Stereotype names must not clash with the names of any inherited Stereotype.

EM5b Stereotype names must not clash in the meta-class Namspace nor with the names of any inherited Stereotype, nor with any baseClass names.

EM5c The baseClass name must be provided; icon is optional and specified in an implementation specific way.

EM5d Tagg names attached to Stereotype must not clash with meta-attribute namespace of the appropriate baseClass ModelElement nor with Tagg names of any inherited Stereotypes.

EM6 Taggs associated with ModelElement (directed via a property list or indirectly via a Stereotype) must not clash with any meta-attribute associated with the ModelElement.

EM7 A ModelElement must have at most one TaggedValue with a given tag name

3.6 The Evaluation Instrument

The *rules* represented in sections 3.4 and 3.5 cover all modelling constructs defined in the Core and Extension Mechanisms of the UML Foundation package [24]. If we add 106 well-formedness rules from the Behavioural Elements package [24], then the total number of rules will be 188. This comprises only one aspect of the evaluation instrument. The way these rules are implemented as 'checks' in CASE tools should also be considered. This may include various questions like: "when/how does a 'check' occur" and "what type of feedback the CASE tool provides", as discussed and used in Vessey [25] and Juric [6].

Hence the complete evaluation instrument requires that for each 'check' the following be considered:

(1) WHEN a 'check' occurs:
  (a) When creating a modelling construct/diagram
  (b) When saving a modelling construct/diagram
  (c) When exiting a diagram where a modelling construct is used
(2) HOW a 'check' occurs:
  (a) Automatically
  (b) On request
(3) WHAT is the feedback when implementing a 'check':
  (a) Warning is issued
  (b) Error is issued

In order to test such an evaluation instrument against OO CASE tools it has been decided to limit the number of 'checks' to 82 UML rules that are listed in sections 3.4 and 3.5 They cover the creation of class/object diagrams only. Due to lack of time and for reasons of simplicity, only 3(a)(b) from the above is considered in the testing. Hence, for each *rule* it was asked:

(i) Is this *rule* embedded as a 'check' in a tool?
(ii) If YES, consider WHAT feedback a tool provides: warning or error.

4. Applying the Evaluation Instrument

The authors have aimed to test the evaluation instrument by applying it to the leading OO CASE tools, which contributed to previous research (Juric, [6]). However, on this occasion only two OO CASE tools vendors responded positively: *SELECT Software Tools* with their SELECT Enterprise and *Popkins Software & Systems* with their System Architect. An important role in the evaluation has been given to their consultants. They were analysing the evaluation instrument with the authors and answering questions (as suggested in section 3.6) in order to cut the time required for an independent evaluation of each tool by someone unfamiliar or not equally familiar with both tools. Unfortunately, RationalRose (*Rational)* and ObjectTeam (*Cayenne Software*) consultants declined the help needed when evaluating and approving the results of the evaluation, due to lack of time and non-availability of their consultants.

4.1 System Architect

System Architect from Popkins Software & Systems supports a range of methodologies, from structured to object-oriented. Specific rule 'checks' are provided for each notation, including UML. The *rules* from the evaluation instrument may be compared with those provided in System Architect. The following 11, out of 82 *rules* from the evaluation instrument, are examples of 'checks' applied by System Architect: CE1a, CE1b, CE2a, CE3, CE3b, CE6, CE9, CE10, CE10a, CE21, and CE27.

The main philosophy of this tool is to give as much flexibility as possible to the user/developer when applying the UML. This means that the number of 'checks' embedded in the tool, that are enforced by the tool and can not be over-ridden, is relatively small. However, the user/developer has the opportunity to implement and enforce a quite significant number of other optional 'checks':

(i) Prescribed in the UML but deliberately not enforced by the tool (e.g. a 'warning' message is issued!), or
(ii) User implemented (important for the modelling from the user's perspective).

The default UML *rules* checking in System Architect does not support all of the *rules* in the evaluation instrument. However, the *rules* checking mechanisms allow additional 'checks' to be written so that almost all 82 *rules* from the evaluation instrument could be covered. Furthermore, it is possible to identify additional modelling *rules* implemented by System Architect but not identified in the evaluation instrument (e.g. "each class must have at least one operation"). System Architect also enforces more 'checks' that are covered in the context of general input and storage mechanisms (like 'drop' and 'drag' from an allowed list), which might be included in (i) and (ii) above and are applicable in some circumstances.

4.2 SELECT Enterprise

SELECT Enterprise from SELECT Software Tools supports the UML (including Jacobson OOSE and OMT) and some structured analysis and design methodologies. It exhibits a specific philosophy in the OO development process, which resulted in many 'checks' being embedded in the tool. This means that many 'checks' described in the evaluation instrument, plus some other 'checks' that are not contained in it, are enforced by the tool (issuing an error when a particular *rule* is violated or when a particular checking is initiated). These enforced 'checks' can not be changed or over-ridden but they can be extended if required by the user/developer.

The following 30, out of 82 *rules* represented in the evaluation instrument are examples of 'checks' embedded in SELECT Enterprise: CE1, CE1b, CE2, CE2a, CE3, CE3a, CE3b, CE6, CE13, CE13a, CE17, CE18a, CE18b, CE21, CE24, CE25, CE26a, CE26b, CE27, CE30a, CE30b, CE31a, CE31b, CE31c, CE34a, CE34b, CE35, CE38, EM2, and EM5c.

The following 6 *rules* are also embedded in SELECT Enterprise: CE2b, CE10, CE10a, CE15, CE23, and EM3, but their roles are different from the set of 30 *rules* specified before. They do not enforce but they do allow some modelling elements to be constructed in a particular fashion. For example *rule* CE10a states that "*One Interface may be offered by more than one Class*". The tool will simply allow this and will follow the UML modelling principles. *Rule* CE23 which states that "*Generalisation can contain a discriminator which name need not be unique*" will also be supported by SELECT Enterprise and exclude the name uniqueness in this particular case.

Some of the *rules* from the evaluation instrument are not present in SELECT Enterprise for various reasons:

(i) Not considered to be 'vital' for representing the philosophy of the UML support given by the tool (e.g. *rules* CE20a, CE20b), or
(ii) Not clear enough to be implemented (e.g. *rules* CE38a, CE29), or
(iii) Not very desirable in the modelling process practised with SELECT Enterprise (e.g. *rules* CE16, CE29).

4.3 Interpretation and Implications of Results

The results show that System Architect applies 13% and SELECT Enterprise enforces 46% 'checks', specified as *rules* in the evaluation instrument. However, System Architect allows all other *rules* - that are not 'checked by default' - to be implemented to customise the tool. This might also include ALL *rules* from the evaluation instrument. In contrast SELECT Enterprise 'checks' by default many *rules* from the evaluation instrument, but does not allow some other *rules* from the evaluation instrument to be embedded in the tool. This tool could be customised by extending and not over-riding some of the embedded 'checks'. The set of *rules* from the evaluation instrument, which are checked by default in both tools, overlaps: all 'checks' (except *rule* CE1a) found in System Architect are also embedded in SELECT Enterprise.

These results could have implications for OO practitioners and CASE tools vendors. If an OO CASE tool has the majority of *rules* from the evaluation instrument embedded (i.e. *rules* that are checked by default and can not be over-ridden), it will exhibit a ‘strict philosophy’ and disciplined approach in applying the UML. On the other hand a flexible tool will not enforce many ‘checks’ and will allow different *rules* to be embedded in different circumstances. The evaluation instrument, as described in section 3.6, also gives some other criteria that polarise OO CASE tools into restrictive, guided and flexible tools, as in Hatley [4], Vessey [25] and Juric [6]. An experienced OO practitioner might prefer a flexible tool that will allow freedom in the development process, in contrast to a beginner who might benefit from the restrictive or guided tool when learning and applying the UML. In order to develop and clearly specify the tool’s philosophy as UML companion, OO CASE tools vendors could use the same guideline. This also helps users to buy the right tool.

Without guidelines from the UML authors, it could be difficult to define the minimum set of *rules* that an OO CASE tool should have embedded when supporting the UML. In addition, although Rational was instrumental in the creation of the UML, it has been adopted as part of the OMG’s standards drive. It is in the public domain now, and Rational does not have any criteria for OO CAE tools vendors to ensure they meet UML certification. One has to assume that the number of *rules* and the way they are implemented in a tool, as specified in the evaluation instrument, will determine the philosophy of the tool and serve as a guideline when deciding when and how to use which tool.

5. Conclusions and Future Research

This paper attempts to create a basis for an evaluation instrument that can measure the extent to which commercially available OO CASE tools support the UML. The instrument itself should consist of (a) a set of the UML *rules* that are supposed to be present within tools in the form of ‘checks’ and (b) the way the *rules* are implemented. In order to create a set of *rules*, the UML[24] has been consulted and the process of extracting them appears not to be a straightforward task. Despite having explicitly defined UML well-formedness *rules* there are inconsistencies within the document when defining them, which is particularly evident in the UML Core Foundation Package [24]. This is a serious obstacle when extracting all *rules*: it was not possible to rely on well-formedness *rules* only (defining static semantics). There are *rules* found within dynamics semantics of the modelling constructs that should not be missed. The solution is in the UML document that includes a section on all *rules* (well-formedness rules and *rules* that cover dynamics) that are to be fully stated, separated from abstract syntax and referred from abstract syntax whenever necessary. *Rules* should also be listed in hierarchical order as the modelling constructs are represented in the diagrams of the UML documentation set. This would improve the readability of the whole UML Semantics document and would be of significant value to OO CASE tools vendors.

The *rules* extracted in this paper are applicable in the evaluation process. Many of them can be and are embedded in OO CASE tools and many of them are even enforced by tools. However, the list of *rules* represented in sections 3.4 and 3.5 is not exhaustive and does not cover all *rules* that are expected to be available when creating class/object diagrams. This problem should be addressed by the UML authors. It also requires a careful comparison of well-formedness rules from the Core and Behavioural Elements Packages [24] in order to capture the *rules* that might have been missed.

The evaluation instrument defined in this paper proved to be applicable in the process of measuring how OO CASE tools support the UML. There is a possibility of applying it on a larger scale. It will polarise OO CASE tools as having a different underlying philosophy: from very flexible tools that do not enforce many ‘checks’ (e.g. System Architect may exclude the enforcement of *rules* significantly) to restrictive tools that enforce many ‘checks’ and give a strict guideline on the UML (e.g. SELECT Enterprise enforces almost 50% of the ‘checks’ listed in the evaluation instrument).

The first task in future work is to include the collection of all UML *rules*: 82 from sections 3.4 and 3.5, and 106 well-formedness rules explicitly defined for the behavioural modelling constructs in the UML Semantics document. Furthermore, all these *rules* should be ‘translated’ from strict UML syntax into vocabulary suitable for the wider OO community (not exclusively for OO practitioners comfortable with UML terminology). This requires careful analysis of all *rules* in order to discuss potential redundancy amongst them. *Rules* can also be categorised according to a particular modelling construct/diagram they cover. They should include an additional completeness /consistency checking between different diagrams/ models.

Such a collection of *rules* could be used in many different ways. It could be incorporated into existing evaluation frameworks, such as one described in Mehandjiska et al. [16] and find its place within more than one node of their CASE tools type classification hierarchy. It could also be used as the component of a stand-alone evaluation instrument, where for each *rule* all three aspects of *rule* implementation in a CASE tool are considered (as specified in (1)(a)(b)(c), (2)(a)(b) and (3)(a)(b) from section 3.6). This could be done on a large scale and include as many OO CASE tools as possible. It would be interesting to see how this evaluation instrument might contribute to the criteria that OVUM employs in evaluating CASE products [19]. This is particularly applicable to the analysis/ design/ construction sections of their assessment method.

Acknowledgements

We wish to acknowledge the assistance of the following consultants: Eugene McSheffrey and James Midgley from Popkins Software & Systems and Mark O'Hare from SELECT Software Tools. Without their help this paper could not have been accomplished.

References:

[1] Booch, G., *Object Oriented Analysis and Design with Applications*, The Benjamin Cummings Publishing Company, Wokingham UK, 1994

[2] Coad, P. and E. Yourdon *Object Oriented Analysis*, Yourdon Press, 1990

[3] De Marco, T., *Structured Analysis and System Specification*, Yourdon Press New York US,(1979)

[4] D.J. Hatley, "CASE Tool Evaluation: A real-time example", *Proceedings of CASE '88*, Boston 1988, pp. 28-32

[5] Jayaratna, N., *Understanding and Evaluating Methodologies NIMSAD: A Systematic Framework*, McGraw Hill Book Company, 1994

[6] Juric, R., *OO Methodologies and OO CASE Tools, To Which Extent Do Commercially Available OO CASE Tools Support OO Methodologies*, Technical report, Southbank University, 1995

[7] R. Juric, and J. Snaith, "A comparative analysis of Booch '93 and OMT object oriented methodologies", In D. Kalpic, V Hljuz-Dobric (eds.) *Proceedings of the 17th International Conference on Information Technology Interfaces '95*, (Pula, Croatia), Zagreb: University Computing Centre, June 1995, pp. 559-609

[8] R. Juric, "the unified method rules", In M.M. Tanik, F.B. Bastiani, D. Gibson, P.J. Fielding (eds.) *Proceedings of the Second World Conference on Integrated Design and Process Technology*, (Austin, Texas, US) IDPT Volume 2, Austin: Society for Design and Process Science Press, December 1996, pp. 272-279

[9] R. Juric, "The UML rules", accepted for publication *in ACM Software Engineering Notes (SIGSOFT*) in 1998

[10] R. Juric, and I.Y. Song, "The assessment of OO modelling elements of the UML 1.1", in T. Ozsu, A. Dogac, O. Ulusoy (eds.) *Proceedings of the third biennial world conference on Integrated Design and Process Technology*, (Berlin, Germany) IDPT Volume 2, Austin, Texas, US: Society for Design and Process Science Press, July 1998, pp. 464-473

[11] B. Lang, "CASE support for the software process: advances and problems", in A. Lamsweerde and A. Fugetta (eds.) *Proceedings of ESEC '91*, Springer-Verlag Berlin, 1991

[12] N. Lang, "Shlaer-Mellor object-oriented analysis rules", *ACM SIGSOFT, Software Engineering Notes*, Vol.18, No.1, 1993, pp.54-58

[13] P. Marttiin, M. Rossi, V. Tahvanainen, and K. Lyytinrn, "A comparative review of CASE shells: a preliminary framework and research outcomes", *Information and Management*, Elesevier Science Publishers B. V., 1993

[14] C. McClure, "The CASE for structured development", *PC Technical Journal*, August 1988, pp. 51-67

[15] D. Mehandjiska, D. Page, and M.D. Choi, "The development of an intelligent object oriented CASE tool", in Patel D, Sun Y and Patel S (eds.) *Proceedings of the first International Conference on Object Oriented Information Systems* (London, December), Springer-Verlag, December 1994, pp. 215-226

[16] D. Mehandjiska, D. Page, and M.D. Choi, "Meta-modelling and methodology support in object oriented CASE tools", in Patel D, Sun Y and Patel S (eds.) *Proceedings of the third International Conference on Object Oriented Information Systems* (London), Springer, December 1996, pp. 370-386,1996

[17] V. Mosley, "How to assess tools efficiently and quantitatively", *IEEE Software*, Vol 8, No 3, 1992, pp. 160-163

[18] E. Nilsson, "CASE tools and software factories", in Goos, G. and Hartmanis, J. (eds.) *Advanced Information Systems Engineering*, CaiSE '90, Springer-Verlag, Berlin, 1990

[19] *OVUM Evaluates CASE Products*, editor Budd M. OVUM Ltd, March, 1998

[20] S. Papahristos, and W. Gray, "Federated CASE environment", in G. Goos, and J. Hartmanis, (eds.) *Advanced Information Systems Engineering*, CaiSE '90, Springer-Verlag, Berlin, 1991

[21] Peters, L. (1988) *Advanced Structured Systems Analysis and Design*. Prentice Hall, New Jersey, US, 1988

[22] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object Oriented Modelling and Design*, Prentice Hall International,1991

[23] Sarson, T. and C. Gane, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, New York, US, 1979

[24] *The UML 1.1 Documentation set*, Rational Corporation, September, 1997,

[25] I. Vessey, S.L.,Javenpaa, and N. Tranctinsky, "Evaluation of vendors products: CASE tools as methodology companions", *Communications ACM*, Vol 35,No 4, 1992, pp.91-105

[26] I. Vessey, and A.P. Sravanapudi, "CASE tools as collaborative support technologies", *Communications ACM*, Vol.38, No.1, 1995, pp.83-95

[27] Wainwright, M., D. De Hayes, J. Hoffer, and W. Perkins, *Managing Information Technology*, Macmillan Publishing Company, UK, 1991

[28] Yourdon, E., *Object Oriented System Design, An Integrated Approach*, Prentice Hall International, Inc., 1991