

Enterprise Modelling and the UML: (sometimes) a conflict without a case

G. Berio

Dipartimento di Informatica, Università di Torino, Torino, Italy

M. Petit

Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium

ABSTRACT: Nowadays, the UML™ language is one of the most widely accepted software modelling language. Enterprise modelling is concerned with the externalisation of knowledge about the enterprise, for a different set of purposes. One possible, future and challenging objective of enterprise modelling is to provide a base for developing, composing and selecting enterprise software applications. Therefore, there is a tendency to apply the UML for modelling the enterprise behaviours and structures mainly because the concept of “object” is ubiquitous and allows, safe or unsafe, reuse of already existing definitions. However, this reuse can sometimes become abusive. Furthermore, because there is no consensus on languages and concepts for enterprise modelling, it does not make much sense to evaluate UML by comparing it to any specific enterprise modelling technique.

As a consequence, in this paper, rather than trying to evaluate UML as a candidate for enterprise modelling, we take a more general view, trying to explain the possible roles of the UML with respect to enterprise modelling. We investigate mainly three kinds of use of UML and show that only one of these may sometimes be conflicting.

KEYWORDS: Enterprise Modelling, UML, Language, Syntax, Semantics, Ontology, Meta-modelling.

1 INTRODUCTION

Enterprise Modelling is concerned with the explicit representation of knowledge about the enterprise into models (Vernadat 1996). Enterprise modelling requires *Enterprise Modelling Languages* for describing specific and needed aspects of the enterprise.

1.1 Why modelling

Modelling (i.e. the process to produce models) always happens before spending lot of time and resources in building, delivering, choosing or providing solutions (and products). In this sense, modelling may allow to reduce possible trivial and important errors in products and solutions. Modelling and models may also be useful for reusing practices (*patterns*) or (software) products.

Modelling always focuses on specific aspects while taking into account all important details about these aspects. Therefore, issued models represent these specific aspects and details while hiding other

aspects and details. In other words, modelling and models always refer to some purpose: accordingly, some aspects and details are or not included into models. As a consequence, models can effectively be analysed for:

- i) understanding the reality, managing complexity and communicating knowledge about some reality;
- ii) getting confidence in the fact that models are coherent with some reality or purpose (validation, reuse),
- iii) checking that models satisfy some given properties (verification, reuse).

These three tasks above are related: we just provide three examples. Before (i) and (iii), the task (ii) should be performed for validating models. We check properties of models also for better understanding solutions or wishes (concerning a given reality). Finally, both tasks (ii, iii) can be performed for reusing of existing models.

Sometimes, however, we try to distinguish more clearly between these three tasks. For instance, whenever the wishes can be expressed formally as

properties, validation is called verification. The distinction, well-known in software engineering, between validation and verification can be naturally applied to all modelling domains (including enterprise modelling).

1.2 What is a modelling language

Modelling languages are languages used for building models. Modelling languages, as opposed to *programming languages*, are for modelling i.e. representing through models a given reality or thinking or possible solutions to given problems. Programming languages are for creating programs. Programs can be understood in a broad sense as prescriptive descriptions of a particular behaviour. The distinction between programming languages and modelling languages is nowadays changing. In fact, with for instance the MDA™ (*Model Driven Architecture*) (OMG 2001), workflow systems and so on, what has been historically called modelling language may be actually understood as high level programming language. The distinction may also apply to “enterprise languages”. Besides *enterprise modelling languages*, we may introduce *enterprise programming languages*, needed for *programming the enterprise* (defining enterprise programs that describe behaviours that can be “executed” through the use of a supporting *enterprise architecture* or *enterprise infrastructure*).

In the last ten years, the classical problem in software engineering of creating (and finding) reusable software components and reusable analysis and design practices (patterns), has dramatically made progress: new complex high level software components and patterns which actually implement or design large parts of enterprise software applications or business services appear on the scene, thereby reducing the distance between modelling and programming. However, the distinction between modelling and programming is still important because using or implementing a component requires modelling it before. The classical problem of generating programs from models remains a research objective especially for really complex components. If eventually modelling languages and programming languages overlap, the enterprise infrastructures should be able to interpret models as their programs (see e.g. the *execution services* defined in the *CIMOSA integrating infrastructure* (AMICE 1993)).

1.3 Which are the basic features of modelling languages

A modelling language is first of all a language. Therefore, all the features applicable to the concept of language should be explored. In this paper, we mainly refer to the notion of language as it appears in

Computer Science. There are two basic features of a language:

- *syntax*, defining the set of all possible sentences that may legitimately be written in the language (also called the physical part of the language),
- *semantics*, defining (one of) the set of possible meanings of sentences written in the language.

In the remainder, we denote by $\text{Syntax}(L)$, the syntax of a given language L and by $\text{Semantics}(L)$ the meanings of sentences written in $\text{Syntax}(L)$.

As explained for instance in (Harel & Rumpe 2000), these features are represented by using (other) *appropriate languages*. In fact, at least in Computer Science, it is typical to express syntax of a language with a *formal grammar* or *abstract syntax* or *meta-model* or with a *visual syntax*. The same happens for semantics which is represented through some (other) language.

As a consequence, $\text{Syntax}(L)$ holds a relationship to $\text{Syntax}(L^*)$ where L^* is an appropriate language for the *specific purpose of describing the syntax of L itself*. The same amounts for $\text{Semantics}(L)$ which holds a relationship to $\text{Syntax}(L')$ where L' is an appropriate language with the objectives of describing the semantics of L itself. This means that eventually syntax and semantics are always some kinds of syntax. Figure 1 provides a good overview of our previous discussions.

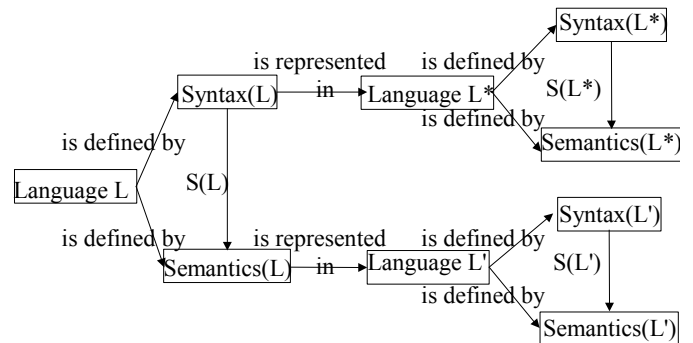


Figure 1. Syntax and semantics of a language and how to represent them.

As argued in (Harel & Rumpe 2000), sometimes syntax and semantics are not well distinguished because for instance, some *language designers* claim they have a *formal language* while having just defined a *formal syntax* and missing a *formal semantics* yet (see Section 1.5). Also, confusion may arise because the same language may be employed for describing both the syntax and semantics i.e. $L^*=L'$ (as in some *meta-modelling techniques*, such as e.g. Telos (Mylopoulos 1990)). Other possible source of confusion is whenever a language is defined by itself (i.e. $L=L^*=L'$), such as in MOF™ (OMG, 2002) which is defined by using the MOF syntax.

1.4 The relationship between syntax and semantics

Given a language L , the relationship between its syntax and its semantics can be defined as a *function* $S(L)$ (in the mathematical sense):

$$S(L): \text{Syntax}(L) \rightarrow \text{Semantics}(L). \quad (1)$$

According to this approach, the syntactical artifacts (i.e. the elements of $\text{Syntax}(L)$), have no meaning till at least one mapping $S(L)$ is explicitly defined. In this sense, $\text{Syntax}(L)$ is at some extent analogous to *data in a database*.

L^* being the language used to define the syntax of a language L , the precise relationship between $\text{Syntax}(L)$ and $\text{Syntax}(L^*)$ can be defined through an element $\text{SyntaxRules}(L) \in \text{Syntax}(L^*)$. These syntax rules expressed in L^* usually define $\text{Syntax}(L)$, which is an often an infinite set, in a finite way (by e.g. providing an *intentional definition* of this set). In fact, provided some $S(L^*): \text{Syntax}(L^*) \rightarrow \text{Semantics}(L^*)$ as described before, then the mapping $S(L^*)$ allows to define (and sometimes to “generate”) $\text{Syntax}(L)$ as follow:

$$S(L^*)(\text{SyntaxRules}(L)) = \text{Syntax}(L). \quad (2)$$

Sometimes, it is possible and interesting to define the semantics of L through a mapping $S^*(L)$ defined on $\text{SyntaxRules}(L)$ rather than directly on $\text{Syntax}(L)$, and which is equivalent to $S(L)$. In most cases, however, this requires the replacement of the semantics domain $\text{Semantics}(L)$ by a more generic domain $\text{GenericSemantics}(L)$:

$$S^*(L): \text{SyntaxRules}(L) \rightarrow \text{GenericSemantics}(L). \quad (3)$$

As before, $\text{GenericSemantics}(L)$ is expressed in some language L' .

It should be noted that $\text{SyntaxRules}(L)$ is not necessarily a “simple set”: probably it is a richer algebraic structure. In this case, $S^*(L)$ is more than a function defined on sets, but rather a *morphism* on algebraic structures defined onto $\text{GenericSemantics}(L)$.

Some uses of classical *meta-modelling techniques* can be characterised in this framework by the facts that:

- $L^*=L'$ as described in Section 1.3 and
- in the language $L^*=L'$, is also possible to express the mapping $S^*(L)$.

This means that all the three distinct concepts (syntax, semantics and mapping) are confused (without discipline) into a single representation in the same language and things are often misunderstood. Therefore, the meta-model of a language L , $\text{MetaModel}(L)$, embeds $\text{SyntaxRules}(L)$, $\text{GenericSemantics}(L)$ and $S^*(L)$.

Given language L , languages for representing $\text{Syntax}(L)$ and $\text{Semantics}(L)$ (L^* and L' respectively) are not necessarily unique. If *several syntax and semantics representations exist* for the same language L , some fundamental conditions have to be satisfied:

- distinct syntax representations must be *compatible or equivalent*,
- distinct semantics representations must be *compatible or equivalent*,
- distinct S mappings must be *compatible or equivalent*,
- the languages used for representing the distinct syntax and semantics should be appropriate for the *objectives to be achieved*, as described in Section 1.3.

1.5 Further topics about modelling languages

Underlying of language features described in Section 1.4, there are other important topics. The first one is the *degree of formalisation* of appropriate languages such as L^* and L' . This degree results in (Harel & Rumpe 2000):

- formal, semiformal or informal syntax,
- formal, semiformal or informal semantics.

Formal syntax and semantics are achieved when the languages L^* and L' are “mathematical” languages (and a formal language is whenever both syntax and semantics are formal). Formal syntax is a necessary condition or a pre-requisite for building software applications that support the creation of models in the language. But usually these applications also need to understand the meaning behind the syntax (for e.g. performing analysis or simulation of models). Formal syntax is however the first and simpler area of possible standardisation.

Syntax, semantics and mapping should be associated to their natural *stakeholders* being both *humans* and *computerised machines*. A language to be used by humans does not need, for instance, to formalise all syntactical aspects. As noted before, that is not true for languages to be used by computerised machines that need to recognise the possible set of language sentences.

According to task (iii) mentioned in Section 1.1, a (modelling) language L may also enable *formal checking of some properties about its sentences*. In this case, both $\text{Syntax}(L)$ and $\text{Semantics}(L)$ of the language L should be formal; otherwise, no formal proof of any property is possible. To accomplish this formal checking, it is required (i) that properties are expressed in some (formal) language and (ii) some definition of when a sentence in the language L *satisfies* a property. Furthermore, formal checking of property falls in two categories:

- *automatic and semiautomatic checking*, focusing on explicitly expressing a *limited set of rules and methods for making proofs of (any) property* on $\text{Syntax}(L)$ (*theorem provers*) or $\text{Semantics}(L)$ (*model checking*),
- *humans checking*, without any limited set of rules, for instance, as applied in general mathematics.

Checking of properties is therefore an important topic even in *natural language*, mainly used by humans (mathematics provides a good example). Humans do some reasoning about perceived language sentences and try to get conclusions, implications, synthesis and so on.

If the language L is equipped with a informal semantics, humans only may try to provide some evidence that properties are satisfied or not.

Formal checking of properties suffers of well-known limits. As an example, *given two computer programs (or even complex enterprise software components)*, we are able to model them. However, it may appear impossible to check their *equivalence* because of two distinct reasons:

- 1) it is difficult or even impossible to represent *why the programs have been built* (because models are always partial),
- 2) whenever, we are able to represent why the programs have been built, through a set of properties (models), it may appear impossible to effectively check the equivalence between the two programs according to those properties (models).

The first reason is because programs do not contain the rationale behind them. That is related to the specific use and meaning that programs have in the specific context. The second reason is because of the *Gödel's incompleteness theorem*. This theorem applies explicitly to "effective check": indeed, for effectively checking a property we need a *reasoning system* (made of *rules*, *axioms* and a way for combining them in *proofs*). Under the hypotheses that the reasoning system is not trivial (it allows to represent *recursive functions* i.e. a "theory of computation", as usual when we talk about computer programs) and consistent (in logical sense, i.e. it does not contain contradictions), the theorem states that there is a property for which it is not possible to proof, through the reasoning system, if the property is true or if the negation of the property is true. In other words, we are not able to say, by using the reasoning system only, if the property is true or false.

A fundamental topic associated to modelling languages, also advocated in (Jackson 1995, Opdahl & Henderson-Sellers 2002), concerns the existence of, at least, *two possible kinds of semantics for a given syntax*, respectively (also related to point (1) above):

- dependent on the real-world situations to be represented by the modelling language itself,
- an abstract way to describe language semantics without linking to any real-world.

To illustrate this distinction, let's introduce the following example. Figure 2 shows one model of a simple production process, represented with some visual language:

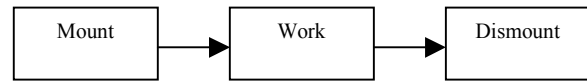


Figure 2. A simple production process with three activities.

From a purely *abstract point of view*, the above process is just a sequence of *three activities* and its *behaviour* (i.e. one of its semantics) can be formulated as the set of all possible sequences of activities i.e.

{<Mount>, <Mount, Work>, <Mount, Work, Dismount>}.

However, it is not true that this process is equivalent to, or in other words, can be substituted by, any other process performing three activities in sequence. This is because the semantics given above does not contain all the information concerning the production process as it exists in the real-world.

As a consequence, the same process representation above can be *complemented* with another meaning in term of the real-world that the process itself should represent. This is just another kind of meaning which should be coherent with the previous one (more abstract) and defines the model in term of generic concepts found in categorised real-world situations (or enterprises). For instance, the activity "Mount" may refer to a commonly accepted definition of a class of process that loads a product to be transformed on a machine. That is the typical domain of *ontology* (see e.g. (Hakimpour & Geppert 2002)) which provides generic concepts for organizing and common understanding of (part of) the real-world. However, we observe that any ontology needs to be expressed in some language which, in turn, needs to be understood in some way. *Ontology* also focuses on *human understanding*. However, often humans understanding is not really common even in restricted fields and eventually time-consuming (for instance, laws need courts of justice for addressing the common understanding!).

An ontology is sometimes interesting because it allows to evaluate if some language is able to represent real-world situations which are of interest (and referred to by the ontology). The general approach discussed in this Section also applies to ontology which may play the role of the GenericSemantics(L).

2 THE UML

The UML is a *software modelling language*. It is essentially centred on the *class diagram* allowing to represent *object-oriented software* as being made of classes. Other diagrams such as the *sequence and collaboration diagrams* are essentially used for *behavioural specifications* of the classes of the class diagram.

Related to the discussion about ontology in Section 2, the UML concerns the typical elements found and needed when building object-oriented software. To be considered as a good language for software modelling, the UML should provide support to all kinds of tasks found in *software engineering methodologies*. One typical example of such tasks is commonly called *requirement specifications* and aims at producing good and right descriptions of so-called *software requirements*. In some software development methodologies (Jacobson & Booch & Rumbaugh 1999), the UML is advocated for expressing requirement specifications. This approach implicitly assumes that requirements of a software application can be described by a software modelling language. However, from e.g. (Jackson 1995, Jackson 2002), it is clear that a requirements specification should be based on the representation on non-software aspects surrounding the software.

2.1 A source of ambiguity: requirements and requirements specification

The software market is characterised by pushing the software components that interoperate in software architectures to the business level. These business-oriented components can be reused many times because nearer to what are called requirements. The *UML profiles* such as EDOC (*Enterprise Distributed Object Computing*) and EAI (*Enterprise Application Integration*) are good examples of this trend. In this sense, profiles are really useful because they allow to customise the core elements of UML, used for generic software requirements specification, to address some more specific software applications domains.

As explained before, a UML objective is to support all typical tasks found in software engineering methodologies. The tasks of *specification of software requirements* required in every software project have therefore to be supported. Software requirements specification generally involves two basic views (Broy 2001):

- black box,
- glass box.

The first one is more related to the interaction of software application with its *environment* (which provides the boundary of the software applications) whilst the second one provides information on what the software should take into account or how the software should internally (and abstractly) behave (as happen for instance in workflow applications).

Software requirements should describe all the behaviours or constraints that a software application should satisfy. According to e.g. (Jackson 1995), requirement specification includes elements belonging to the *situation under analysis* (e.g. an enterprise). For instance, it is obvious that the requirement specification of a *software application*

controlling a production process might contain a UML class named “MyProduct” (i.e. a thing to be produced). This class MyProduct belongs to the enterprise under analysis. However, the way (i.e. the language) in which “MyProduct” is represented in a requirement specification for developing such a software application is specific to this objective. In this case, the definition of a stereotype <<Product>> can be valuable. It would collect in a single place all the information related to “thing to be produced” and represent it so that it can be reused in future software developments.

Therefore, the activity of software requirement specifications can benefit from the definition of such an “enterprise stereotype”. More generally, “enterprise stereotypes” can be used for defining some kinds of *analysis patterns*, to be later reused for developing (*object-oriented*) *enterprise software applications*.

In this sense, *the UML is not conflicting with the enterprise modelling field*. The UML is used during the software requirements specification, provided that is acceptable, where enterprise stereotypes and associated analysis patterns can be reused successfully for developing enterprise software applications.

2.2 The relationship between syntax and semantics in the UML

It is well known that UML 1.4 does not provide any formal semantics. It is our opinion that UML is mainly a notation (i.e. syntax). The UML has been defined by using a meta-model (let us call it MetaModel(UML)) in which, sometimes, there is some confusion between syntax and semantics.

The Syntax of the UML is mainly defined through a set of *visual diagrams* (i.e. annotated graphs, also expressed in MOF), *explained and equipped* with some natural language (NL) sentences and well-formedness rules in OCL (*Object Constraint Language*). Therefore,

$$\text{SyntaxRules(UML)} \in \text{AllVisualDiagrams} \cup \text{Syntax(NL)} \cup \text{Syntax(OCL)}$$

or even

$$\text{SyntaxRules(UML)} \in \text{Syntax(MOF)} \cup \text{Syntax(NL)} \cup \text{Syntax(OCL)}$$

where a AllVisualDiagrams is the set of all possible annotated visual graphs.

However, some points are ambiguous. In the *behavioural packages* some elements can be more associated to semantics than syntax. S*(UML), the semantic function of UML, is really simply defined by annotating the visual diagrams describing the UML syntax. The description of how to “generate” Syntax(UML) from SyntaxRules(UML) is informally defined through examples (or templates) in the UML *notation guide*.

Finally, we may say that the semantics of UML is expressed in natural language i.e.:

$$\text{GenericSemantics(UML)} \in \text{Syntax(NL)}. \quad (5)$$

In the UML, the *stereotype mechanism* has been introduced for making extensions to the basic UML. According to the UML 1.4, a stereotype is, essentially, a new subclass of some predefined class introduced in the MetaModel(UML).

For instance, it is possible to introduce a stereotype <<Business Process>> as a subclass of “Class” (an element found in AllVisualDiagrams). This stereotype can be used for extending the UML with the concept of “business process” dealing with (some kinds of) real process found in enterprises. However, as we see better in the remainder, in this way, any semantics established for the UML “Class” applies to <<Business Process>>.

3 UML AS AN ENTERPRISE MODELLING LANGUAGE

In this section, we study how and under which conditions UML can be used as a Enterprise Modelling Language (EML) (instead of a software modelling language). This analysis is, in our opinion, important in order to be able to correctly situate the use of UML as a language for Enterprise Modelling in current and future research proposals such as (Panetto 2002) and standardisation efforts such as (ENV12204 1995).

For using UML to model enterprises, two possibilities exist:

- UML is used as it is without introducing any stereotype or;
- UML is extended with a set of stereotypes, defined for the specific purpose before being used.

The first situation is not so interesting in this context and we limit our analysis to the second one.

Before going into details, it should be noted that, unfortunately, the mechanism of stereotype is rather loose because:

- most of the *associations* in the MetaModel(UML) are optional,
- the definition of semantics of stereotypes has to be given in natural language.

However, we feel that the objective of stereotypes is to reuse as much as possible the basic UML semantics. Therefore, added stereotypes should be as much *conservative* as possible.

Let’s suppose we introduce the stereotype for representing in some way the idea of “business process” in an hypothetical language EML for enterprise modelling. Firstly, we introduce a stereotype of “Class” (which is defined in MetaModel(UML)) named <<Business Process>> and specific to some modelling purpose. The stereotype mechanism, as defined in UML, allows to define the new syntax as:

$$\text{SyntaxRules(EML)} =$$

$$\text{SyntaxRules(UML+<<Business Process>>)} \quad (6)$$

where we have used “+” just to indicate the stereotype addition.

On the other hand, it is not really clear how the mapping S^* and the domain GenericSemantics should be updated when a stereotype is introduced.

3.1 Redefining the mapping $S^*(UML)$ and the *GenericSemantics(UML)*

UML can be considered as an EML only if these elements are defined explicitly and are coherent with the domain tackled by enterprise modelling languages (i.e. the enterprise). In order to clarify how a stereotype changes S^* and the GenericSemantics, UML (and its extension with the stereotype) should be defined in term of an *Enterprise Ontology EO* in which concepts relative to the enterprise are defined, and which plays the role of GenericSemantics (UML). The UML (or extensions thereof) can only be used in a clean way for enterprise modelling if and only if it can be “correctly” interpreted via a mapping S^* to that ontology.

As an example, a possible and simple, representation of EO can be given in term of a kind of *semantic network* (an old knowledge representation technique). Referring to (Brinkkemper & Saeki & Harmsen 1999), we may define an Enterprise Ontology EO as:

$$EO = \langle \text{Concepts, Associations},$$

$$\Phi: \text{Concepts} \times \text{Associations} \rightarrow \text{Concepts} \rangle \quad (7)$$

where Φ is a navigation function through associations between ontology concepts. Based on that, we define:

$$\text{GenericSemantics(EML)} =$$

$$\langle \emptyset(\text{Concepts} \cup \text{Associations}),$$

$$\Phi: \text{Concepts} \times \text{Associations} \rightarrow \text{Concepts} \rangle \quad (8)$$

because an element of the syntax of EML may be used for representing (i.e. may be mapped through $S^*(EML)$ to) a complex sentence made of several concepts or associations or a mix of both.

Let’s now suppose that

$$\text{“BusinessProcess”} \in EO.\text{Concepts}. \quad (9)$$

According to what has been discussed above, the definition of $S^*(UML+<<Business Process>>)$ is quite easy and mostly in natural language. Therefore, the mapping S^* may be redefined by adding a natural language description explaining how the stereotype <<Business Process>> maps onto the appropriate EO concept. This allows to interpret the new stereotype according to the real-world (enterprise) under modelling, via the ontology. In this case, we provide a new mapping $S^*(UML+<<Business Process>>)$ such that

$$S^*(\langle \langle \text{Business Process} \rangle \rangle) = \text{“BusinessProcess”}. \quad (10)$$

It is important to emphasise that S^* is defined under some constraints. For instance, according to (Brinkkemper & Saeki & Harmsen 1999),

if $\langle\langle C1 \rangle\rangle$ holds an association A to $\langle\langle C2 \rangle\rangle$ in
 SyntaxRules(UML+ $\langle\langle$ Business Process $\rangle\rangle$) then

$$\Phi(S^*(\langle\langle C1 \rangle\rangle), S^*(A)) = S^*(\langle\langle C2 \rangle\rangle). \quad (11)$$

This means that for all the *associations* in SyntaxRules(UML+ $\langle\langle$ Business Process $\rangle\rangle$) there should also be a corresponding navigation defined in EO.

In this sense, therefore, *the UML may be (even partially) conflicting to the enterprise modelling field, because constraints onto S^* are not verified*. Indeed, some researchers e.g. (Opdahl & Henderson-Sellers 2002) trying to compare UML with existing ontology, found some conflicts. On the other hand, however, this also may be seen as a way to improve the ontology.

However, conflicts do not occur in general because they depend on the chosen Enterprise Ontology. If no conflict occurs, the Enterprise Ontology may be perceived as a tool for improving and clarifying the UML as an enterprise modelling language e.g. (Wand & Storey & Weber 2000, Opdahl & Henderson-Sellers 2002).

4 UML FOR DEFINING THE SYNTAX OF AN ENTERPRISE MODELLING LANGUAGE

Another possible use of UML in the context of enterprise modelling is for defining the syntax of some EML. In this case, the UML (mainly the class diagram) is used as it is for defining EML *syntactical objects* that are to be managed by computers. These objects are part of an *abstract syntax* of the modelling language under definition (the EML). In this case UML is appropriate because we are just “prototyping” the language (defining the conceptual data model of a software tool manipulating models written in the EML). In this case, the UML has no hidden links with the EML since a *UML class model* is used only to define the syntactical artifacts which have no underlying meaning. In fact, according to the formula (2):

$$\text{Syntax(EML)} = S^*(\text{UML})(\text{SyntaxRules(EML)})$$

(12)

where $\text{SyntaxRules(EML)} \in \text{Syntax(UML)}$
 where SyntaxRules(EML) is just a class model in UML.

The meaning of the language EML can be independently defined through one or more semantic functions $S^*(\text{EML})$ to some Enterprise Ontology, as discussed before. In this case, the semantics of UML $S^*(\text{UML})$ does not need to be redefined: both $S^*(\text{EML})$ and GenericSemantics(EML) are defined without any dependency with $S^*(\text{UML})$ and GenericSemantics(UML). As a consequence, no reuse of the GenericSemantics(UML) is possible.

Referring to our previous example on the concept of business process, we may introduce a UML class “EMLBusinessProcess” which is part of SyntaxRules(EML). This UML class allows to define how a specific “business process” is syntactically defined according to $S^*(\text{UML})$.

5 CONCLUSION

In this paper we have analysed some roles of the UML with respect to Enterprise Modelling. We have noted that the stereotype mechanism, in the first case (Section 2.1), can be used without conflicts with enterprise modelling: there, stereotypes are used for specifying high level requirements models or analysis patterns relative to enterprise (object-oriented) software applications.

However, if UML is used as an enterprise modelling language (possibly being extended with stereotypes) (Section 3), some conflict may arise. In order to prevent these conflicts, we propose a safer use of UML: use it only to define the abstract syntax (meta-model) of the enterprise modelling language (Section 4). This syntax can be a base for developing software applications that support e.g. the creation of models from the EML. However, in this case, no already existing UML semantic definitions are reused. This is not so negative because this reuse may occur later:

- i) during the definition of the semantics of the EML, which is then a completely independent activity;
- ii) in some the requirements specification phase, when some software applications are in development.

ACKNOWLEDGEMENTS

The authors are indebted to Yves Bontemps, University of Namur, Namur, Belgium, and the anonymous reviewers, for their valuable comments on the first version of this paper.

REFERENCES

- AMICE, 1993. *CIMOSA: Open System Architecture for CIM*. 2nd Edition, Springer-Verlag, Berlin.
- Brinkkemper, S. & Saeki, M. & Harmsen, F. 1999. Meta-modelling based assembly techniques for situational method engineering. *Information Systems* 24(3): 209-228.
- Broy, M. 2001. Toward a mathematical foundation of software engineering methods. *IEEE Transactions on Software Engineering* 27(1): 42-57.
- ENV 12204 1995. *Advanced Manufacturing Technology – Systems Architecture – Constructs for Enterprise Modelling*. CEN TC 310 WG1.

- Hakimpour, F. & Geppert, A. 2002. Global Schema Generation using Formal Ontologies. In S. Spaccapietra, S.T. March, Y. Kambayashi (eds.); *Proceedings of Int. Conf. ER2002, Lecture Notes in Computer Science* 2503: 307-321.
- Harel D. & Rumpe B. 2001. Modeling languages: Syntax, semantics and all that stuff. Part I: the basic stuff. Personal notes (on www.wisdom.weizmann.ac.il/~dharel).
- Jackson, M. 1995. *Software Requirements & Specifications, a lexicon of practice, principles and prejudices*, Addison-Wesley, Workingham/England.
- Jackson, M. 2002. Some Basic Tenets of Description. *Sosym Journal* 1: 5-9 (www.sosym.org).
- Jacobson, I. & Booch, G. & Rumbaugh, J. 1999. *The Unified Software Development Process*. Addison-Wesley.
- Mylopoulos, J. Borgida, A. Jarke, M. Koubarakis, M. 1990. Telos: A Language for Representing Knowledge about Information Systems, *ACM Transactions on Information Systems*, 8(4).
- OMG Architecture Board ORMSC, 2001. Model Driven Architecture (MDA), Document number ormsc/2001-07-01, July 9, <http://www.omg.org/mda/>.
- OMG 2002, MetaObject Facility (MOF) Specification, Version 1.4.
- Opdahl, A.L. & Henderson-Sellers, B. 2002. Ontological evaluation of the UML using the Bunge-Wand-Weber Model. *Sosym Journal* 1: 43-67 (www.sosym.org).
- Panetto, H. 2002. UML Semantics Representation of Enterprise Modelling Constructs. In K.Kosanke, R. Jochem, J.G. Nell, A.O. Baz (eds.); *Proceedings of International Conference on Enterprise Integration and Modeling Technology (ICEIMT'02)*, 381-387.
- Vernadat F.B. 1996. *Enterprise Modeling and Integration: Principles and Applications*. Chapman & Hall, London.
- Wand, Y. & Storey V.C. & Weber R. 1999. An Ontological Analysis of the Relationship Construct in Conceptual Modeling. *ACM Transactions on Database Systems* 24(4): 494-528.