

# Domain-specific language modelling with UML profiles by decoupling abstract and concrete syntaxes

Jesús Pardillo\*, Cristina Cachero

Department of Software and Computing Systems, University of Alicante, P.O. Box 99, E-03080, Spain

## ARTICLE INFO

### Article history:

Received 15 December 2009

Received in revised form 11 August 2010

Accepted 11 August 2010

Available online 19 August 2010

UML  
Diagramming  
Modelling  
Profiles  
Syntax  
Visual languages

## ABSTRACT

UML profiling presents some acknowledged deficiencies, among which the lack of expressiveness of the profiled notations, together with the high coupling between abstract and concrete syntaxes outstand. These deficiencies may cause distress among UML-profile modellers, who are often forced to extend from unsuitable metaclasses for mere notational reasons, or even to model domain-specific languages from scratch just to avoid the UML-profiling limitations.

In order to palliate this situation, this article presents an extension of the UML profile metamodel to support arbitrarily-complex notational extensions by decoupling the UML abstract and concrete syntax. Instead of defining yet another metamodel for UML-notational profiling, notational extensions are modelled with DI, i.e., the UML notation metamodel for diagram interchange, keeping in this way the extension within the standard. Profiled UML notations are rendered with DI by defining the graphical properties involved, the domain-specific constraints applied to DI, and the rendering routines associated. Decoupling abstract and concrete syntax in UML profiles increases the notation expressiveness while decreasing the abstract-syntax complexity.

© 2010 Elsevier Inc. All rights reserved.

## 1. Introduction

UML profiling (Object Management Group, 2009) is a straightforward technique to extend UML. UML profiles allow modellers to adapt UML to fit the representational needs of particular domains. For this purpose, they define *stereotypes* (Wirfs-Brock et al., 1994) that, applied over the UML *metaclasses*, redefine their notation, syntax and semantics (Berner et al., 1999). Therefore, the core extension concept is the (meta-) metaclass *Stereotype*.<sup>1</sup> This metaclass specialises the metaclass *Class*, to which it adds *Properties* (called *tagged definitions*) and *Constraints* (by means of the *Namespace::ownedRule* property), which are usually defined in OCL (Object Management Group, 2006a). Stereotypes are related with the corresponding metaclasses by means of *Extensions* (Object Management Group, 2009).

UML profiles also support notational extensions through arbitrary *Icons* associated to *Stereotypes*. These *Icons* decorate the notation of extended metaclasses, increasing diagram comprehensibility (Staron et al., 2006). The ulterior representation of these icons is managed by modelling tools according to a set of pre-

defined rules, defined by UML (Object Management Group, 2009) (p. 674): e.g., *Classes*, which are diagrammatically represented by a box containing various compartments, can be represented by a mere icon *iff* a single stereotype is applied to the class and class properties are hidden.

Moreover, stereotypes have a certain meaning associated. Unfortunately, UML does not properly formalise (in the mathematical sense) many of its metaclass semantics (Kong et al., 2009; Opdahl and Henderson-Sellers, 2002). Therefore, the stereotypes semantics is usually expressed (like the UML semantics itself) as an informal description in natural language.

Depending both on their base metaclass and their complexity, stereotypes can be classified into: *decorative* (concrete-syntax<sup>2</sup> extension), *descriptive* (abstract-syntax extension), *restrictive* (descriptive with syntactic constraints) and *redefining* (removal of the original UML syntax) (Berner et al., 1999). Some authors refer to the UML profiles that only contain decorative, descriptive or restrictive stereotypes as *conservative extensions* of UML, while UML profiles that contain redefining stereotypes are called *non-conservative extensions* (Turski and Maibaum, 1987). This distinction is important because only conservative extensions are 'safe', in the sense that they avoid undesired side-effects in UML

\* Corresponding author. Tel.: +34 965 90 3400x2075; fax: +34 965 90 9326.

E-mail addresses: [jesuspv@ua.es](mailto:jesuspv@ua.es), [jesuspv@dlsi.ua.es](mailto:jesuspv@dlsi.ua.es) (J. Pardillo), [ccachero@dlsi.ua.es](mailto:ccachero@dlsi.ua.es) (C. Cachero).

<sup>1</sup> The reason why *Stereotypes* can be considered meta-metaclasses is that they are defined over UML metaclasses.

<sup>2</sup> Concrete syntax and notation are herein used as synonyms. The notions of both concrete and abstract syntax are defined in Appendix A.

models. The reason is that conservative extensions abide by a variant of the Liskov's *substitution principle* (Liskov, 1987), which makes possible to substitute an instance of a stereotyped metaclass by an instance of the extended metaclass without incurring in syntactic inconsistencies (although, of course, with this substitution we may incur in an expressiveness loss). This means that modellers should try to come up with conservative extensions whenever possible. Unfortunately, in the last years we have witnessed how many of the myriad of UML profiles proposed in literature are still non-conservative. This situation is often due to the fact that mere notational reasons lie behind the selection of many extended metaclasses in existing UML profiles (Section 2). Choosing base metaclasses for notational reasons also often provokes the unnecessarily complex redefinition of UML elements in order to support the domain-specific syntax.

This article dives into this situation and argues that both this complexity and the problems caused by non-conservative profiles could be alleviated if UML decreased the coupling between profiled concrete and abstract syntaxes. This decoupling would allow designers to extend from metaclasses whose abstract syntax is closer to the concept represented by the stereotype, without being forced to also use its notation. In other words, modellers should not be forced to extend from a given metaclass in order to be able to use its notation.

Going one step further, whenever it is possible to find (a) closer matches between representation and representee notations or (b) aesthetically more pleasant representations than the ones provided by the general-purpose UML notation, modellers should not be forced to stick by the UML notation, since appropriate notations foster a more effective diagrammatic communication (Green and Petre, 1996).

Many modelling tools, being aware of this fact, have already incorporated their own alternatives to the UML 'icon-based extension rules' (Object Management Group, 2009) (p. 674). For example, tools like the award-winning MagicDraw<sup>3</sup> allow to enrich the notation of UML Associations with additional properties, such as end shapes, colour, and thickness, to cite a few. The decoupling between abstract and concrete syntax in UML profiles would make these 'ad-hoc' solutions unnecessary. Otherwise stated, UML would benefit from allowing for a standard way to enrich notations independently from abstract syntax or semantics.

This article is organised as follows. Next section (Section 2) dives into the problems that derive from the high coupling between abstract and concrete syntaxes in UML profiles. In particular, Section 3 reviews the case of UML profiling for the data-warehousing domain. These problems are further discussed by means of an ER modelling running example (Section 4). Section 5 outlines the foundations behind the UML notation, namely the DI metamodel. Section 6 presents how it is possible to solve some common profiling problems by using this DI metamodel, and illustrates the approach by applying the proposed solution to the aforementioned ER running example. Section 7 completes the approach with a prototype implementation that shows how a DI-based UML profiling tool may work in practice. Last, Section 8 discusses the presented solution and its implications for UML profiling.

## 2. Abstract and concrete syntax coupling in UML profiles

During the definition of UML profiles, modellers need to make decisions on the ideal candidates for extension. Such suitability is usually decided based either on the abstract syntax or on the notation of the profiled concept. Whatever the criterion, the modeller

usually chooses the base metaclass whose either abstract syntax or notation most closely matches that of the profiled concept.

During this process, the modeller can be faced with two extension situations that are prone to causing problems later on:

- The modeller chooses a base metaclass whose abstract syntax is similar to that of the profiled concept. This base metaclass, however, provides a notation that greatly differs from the desired one, and whose adaptation requires more than the mere addition of icons. Since this icon addition is the only normative notational extension in UML, the resulting notation is not suitable for the domain, and therefore the profile is dismissed in practice.
- The modeller chooses a base metaclass whose notation (concrete syntax) is similar to that of the profiled concept. This base metaclass, however, needs to be heavily constrained in order to represent the abstract syntax of the profiled concept, what unduly increases the complexity of the profile and turns it into a non-conservative extension.

Let us exemplify these situations.

The first situation occurs when a metaclass is extended for syntactic reasons, but its notation (which has to be adopted – decorated or not with icons – too) is unsuitable for the domain-specific language.

**Example 2.1** (*ER profile for conceptual data modelling*). Below in this paper, we present as a running example different versions of an ER profile (Section 4). Fig. 4 presents the version where the most suitable candidates from a syntactic point of view have been selected. The resulting notation can be seen in the right side of the figure. This iconised class-diagram notation is clearly different from the look one would expect to find for an ER diagram.

The second situation occurs when a metaclass is extended because of its notation suitability, at the price of having to deal with a syntax that does not correspond with that of the target concept. When this happens, it is common to define an extra set of constraints over the profiled UML metamodel in order to adapt such syntax. Constraints that redefine the syntax of base UML metaclasses, violating the syntax rules of the base metaclass, not only increase the profile complexity, but they also often turn it into a non-conservative extension, much more error-prone for designers.

**Example 2.2** (*UML profile for data warehouses by Abelló et al. (2006)*). Abelló et al. (2006) present a domain-specific language for data warehouses. In this UML profile, facts and dimensions of analysis are modelled as Classifier Stereotypes. Such mapping allows the modelling of domain-specific relations (e.g., aggregations, data flows, etc.) between them. UML Classifiers are a 'classification of instances' (Object Management Group, 2009) (Section 7.3.8). However, in this UML profile, neither facts nor dimensions should have instances; in fact, syntactically speaking, both notions are closer to containers than to classifiers. This profile is a good example of a non-conservative extension. This UML profile still poses another notational problem: the Dimension Stereotype has two notations. At a higher level, dimensions are denoted as rectangles with a text label (the Classifier name) inside. At a lower level, they are denoted as rectangles containing several classes which they are composed of. This contrasts with UML, where the Classifier is an abstract model Element and so, properly speaking, it should have no notation (Object Management Group, 2009) (Section 7.3.8).

This situation can become even worse. Sometimes the UML notation, even with the addition of icons, is simply not expressive enough to denote elements of the domain-specific language. As an example we can cite how, in case of scalability concerns, specific graph layouts (Eichelberger and Schmid, 2009; Dobing and Parsons,

<sup>3</sup> <http://www.magicdraw.com>

2006) or even non node-link diagrams, such as matrix representations (Ghoniem et al., 2004), are often preferred.

**Example 2.3** (UML profile for web-sites by Conallen (2002)). One very popular domain-specific language, supported in commercial tools like Rational Rose,<sup>4</sup> is the profile presented by Conallen (Conallen, 2002) for the Web domain. In this UML profile, Web pages at the logical level are modelled as Class Stereotypes. Examples of Class Stereotypes are ServerPage, ClientPage, Form, etc. Relationships between them are modelled as Association Stereotypes. Some examples are: Link, Build, Submit, Redirect, etc. This extension is non conservative: while Classes have instances, some types of Web pages do not (e.g., static pages). But this UML profile poses even more urgent problems: our experience is that the profile diagrams become unmanageable for medium to complex websites; current websites are characterised by hundreds of pages heavily interrelated. For this domain, class diagrams do not scale well, reason why this notation has not been accepted by practitioners.

These examples, far from being exceptions, show practices that are very common in the definition of profiles in the scientific literature. In order to illustrate this fact, next we present a literature review focused on the domain of data warehousing, where our expertise makes possible to judge the notational, syntactic and semantic mismatches. This review follows the systematic review protocol proposed in Kitchenham (2004).

### 3. Systematic review of UML profiles: the data-warehousing domain

In the last years we have had access to an increasing number of papers whose main contribution is a profile aimed at solving certain modelling problems in different domains, among which that of data-warehousing outstands. Most of these papers have been presented in conferences specialised in conceptual modelling in general and in data-warehousing conceptual modelling in particular. For this reason, the selected sources of primary studies for our review are:

- The International Conference on Conceptual Modelling (ER).
- The International Conference On Model Driven Engineering Languages And Systems (MODELS), formerly known as the International Conference on the Unified Modelling Language (UML, until 2004).
- The International Conference on Data Warehousing and Knowledge Discovery (DaWaK).

These three conferences are, to our knowledge extent, the three major sources for UML profiles in the data warehousing domain. For all conferences, the review also includes the workshop sessions.

These sources have been reviewed during a 9-year period (2001–2009), that is, since the first contribution on UML profiles for data warehousing appeared in UML'01.

The initial set of primary studies has been selected by searching in the title, abstract, and keywords of the papers the following case-insensitive key-phrases (square brackets mean optional and '|' means disjunction):

- extend[ing|s] [the] UML
- UML extension[s]
- UML profile[s]

**Table 1**

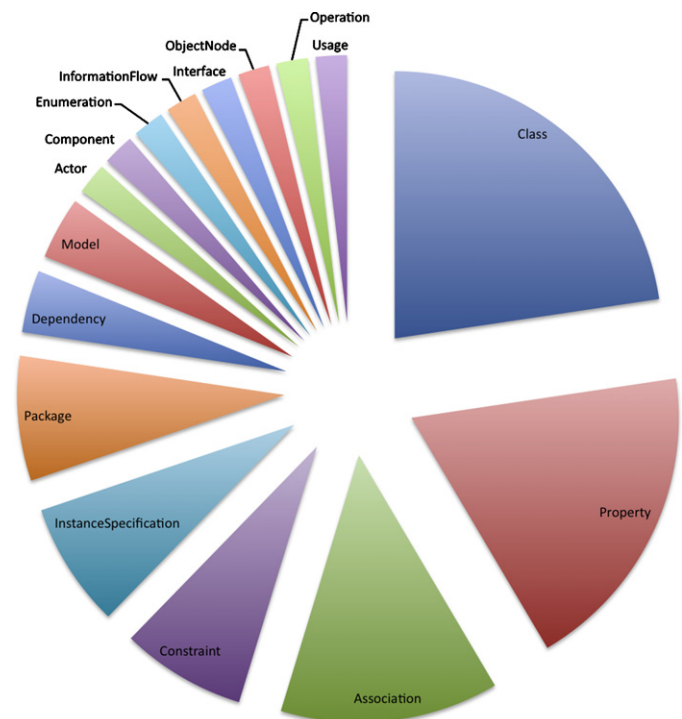
Number of contributions gathered between 2001 and 2009.

Venue	Selected	Discarded	Total
ER	3	3	6
ER Workshops	2	8	10
DaWaK	7	0	7
MODELS	0	11	11
MODELS Workshops	0	5	5
UML	2	29	31
Total	14	56	70

The results of this query have been manually filtered for UML profiles modelling specific aspects involved in data warehousing; each paper has been reviewed by both authors independently, and any contribution not presenting a brand new UML profile has been discarded. Table 1 summarises this process output. A total of 70 contributions were returned by the initial search. On 14 of them, a UML profile for data warehousing was identified.

Next, each one of the 14 contributions has been studied in further detail, in order to obtain the stereotype definitions. Table 2 shows the results of this task. A total of 158 stereotypes have been identified. The number of stereotypes presented largely varies between contributions: while some profiles present just three stereotypes (i.e., Luján-Mora et al., 2002a; Golfarelli and Rizzi, 2008), others include 20 or more stereotypes *per* profile (e.g., Fernández-Medina et al., 2004; Marcos et al., 2001).

In particular, Fig. 1 shows the number of UML profiles for data-warehousing proposals that extend each of the 16 distinct base metaclasses that appear in this study. The most extended metaclass is Class (22%), followed by Property (19%), and Association (13%). It is worth noting that these metaclasses are precisely the ones related with class diagrams. This is not surprising, since it is a well-known fact that the class diagram is the most popular UML diagram in practice (Dobing and Parsons, 2006).



**Fig. 1.** Comparison of base metaclasses by number of UML profiles extending from them. According to our study, the three most extended metaclasses are Class (22%), Property (19%), and Association (13%).

<sup>4</sup> <http://www.ibm.com/software/rational>

**Table 2**

UML profiles for data warehousing and the number of defined stereotypes (#S col.) and base metaclasses (#M col.).

Venue	Reference	Topic	#S	#M
ER	Fernández-Medina et al. (2004)	Security	20	8
	Luján-Mora et al. (2002a)	Multidimensional data	3	1
	Luján-Mora et al. (2004)	Data mappings	7	4
ER Workshops	Kurz et al. (2006)	Data mappings	10	6
	Stefanov and List (2007)	Usage	6	4
DaWaK	Golfarelli and Rizzi (2008)	What-if analysis	3	2
	Pardillo et al. (2009)	Data flows	6	3
	Stefanov et al. (2005)	Business intelligence	10	1
	Stefanov and List (2007)	Data states	15	3
	Zubcoff and Trujillo (2005)	Association rules	13	6
	Zubcoff and Trujillo (2006)	Classification	16	3
	Zubcoff et al. (2007)	Clustering	10	3
UML	Luján-Mora et al. (2002b)	Multidimensional data	8	3
	Marcos et al. (2001)	Object-oriented and relational data	31	7
Total of stereotypes and distinct metaclasses			158	16

Some interesting findings of the study are summarised in Table 3, which presents some descriptive profiling statistics. The average number (arithmetic mean) of stereotypes *per* contribution is 11. With regard to the number of base metaclasses that appear in these contributions, it varies between 1 and 8, with an average of 4 base metaclasses per contribution and 16 distinct base metaclasses in all. The average number of tags per stereotype is notably low (0.6); in fact, only 22% of the identified stereotypes define tags.

This percentage is closely similar to the number of stereotypes that are accompanied by some iconography (28%). Although this percentage may be interpreted as icons having a low presence in existing UML profiles for data warehousing, a deeper study on how the icons are distributed among profiles shows that 71% of the surveyed profiles manage iconography (a strong indicator of the use of UML profiling for notational purposes). While the definition of icons suggest the need for enriched notations, the presence of OCL expressions is usually associated with a need to constrain/modify the original abstract syntax. From the surveyed UML profiles, 64% include some OCL constraint. Also, 43% define some kind of formal semantics.

In order to further confirm or refute our hypothesis about the notational reasons that lie behind the definition of most UML profiles, we have carried out an in-deep analysis of the rationale behind all 14 contributions. The results can be seen in Table 4. In this table, it can be observed how the rationale that lies behind the selection of base metaclasses in all the studied profiles is the UML notation reuse. However, since notation and abstract syntax are coupled in UML, reusing this notation can only be done if abstract syntax is also adopted. Being in all the cases the abstract syntax of the profiled elements different from the abstract syntax of the original UML metaclasses, all the profiles include a high number of restrictions that redefine such abstract syntax. With this redefinition, all

the studied profiles (that is, a 100%) fall into the category of non-conservative extensions of UML, with all the potential problems associated with this kind of extensions.

Summarising, a systematic review of UML profiles in the domain of data warehousing has shown the impact of the coupling between abstract and concrete syntax in UML profiling. In order to illustrate our solution to this coupling, a profile for ER diagramming is being used as a case study in the remaining of this article. This case study is presented next.

#### 4. Case study: the UML profile for ER diagramming

ER diagrams (Chen, 1976) are widely-known design artefacts to model databases. ER diagrams classify data in two main abstractions: *entities* (with its corresponding *attributes*) and *relationships* among them. As this model has spread among practitioners, different variations have been developed, which vary in both expressiveness and notation. As an example, Fig. 2 shows some of the most popular ER variations for relation cardinalities.

Database modelling with ER diagrams is a central topic in database-design courses. This contrasts with the intensive use of UML as the chosen design notation in general software-engineering courses. This situation, together with the use of different tools, causes that some students perceive the two topics as disconnected. The definition of a UML profile for ER diagramming provides students with the opportunity to use the same tool for both courses. In this way, they can experience the pros and cons of using ER diagrams vs. UML class diagrams for the definition of databases. For this aim, a faithful visual representation of ER models over a UML tool is necessary. An example of such faithful notation can be seen in Fig. 3 (left side).

One of the distinguishing features of ER *Entities* with respect to UML *Classes* is the absence of operational semantics. Despite this fact, the UML metaclasses *Class*, *Property*, and *Association* are the syntactically-closer UML metaclasses to the *Entity*, *Attribute*, and *Association* ER metaclasses, respectively. Class diagrams can also express additional notions such as cardinalities or existence restrictions, which are common to both languages. The *AbstractER* profile that profits from this abstract-syntax closeness can be seen in Fig. 4 (left side). However, a UML profile that models this mapping is bound to offer a notation that strongly differs from that of ER diagrams. This notation is illustrated on the right side of this figure, where attributes (*name* and *title*) are denoted by containment within the entity notation (of *Conference* and *Paper*, respectively).

**Table 3**

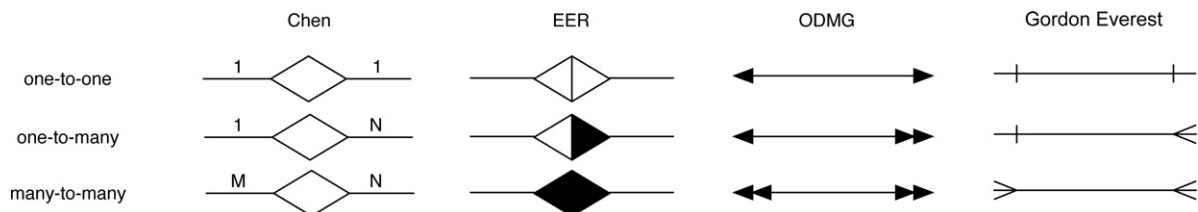
Statistics on the literature UML profiling.

Variable	Value
Stereotypes per contribution	11
Base metaclasses per contribution	4
Tags per stereotype average	0.6
Stereotypes with tags	22%
Stereotypes with icons	28%
Stereotypes with tags and icons	5%
Profiles with iconography	71%
Profiles with OCL constraints	64%
Profiles with formal semantics	43%
Profiles with OCL and formal semantics	29%

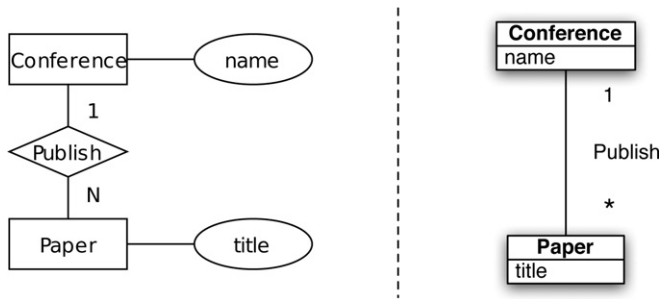


**Table 4**  
Samples on the notational UML profiling for data warehousing.

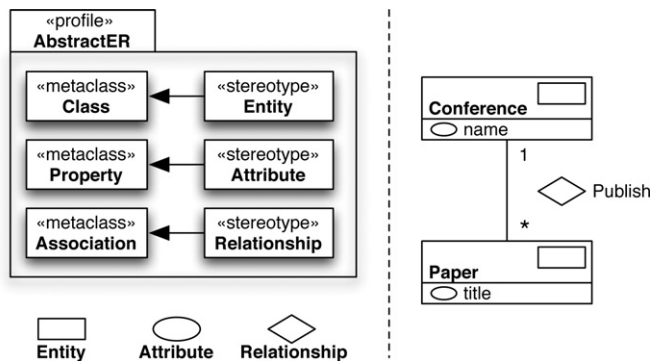
Reference	Domain	Extension	Notational rationale behind the extension
Fernández-Medina et al. (2004)	Security	UML <i>Classes</i> are stereotyped as <i>user profiles</i> . UML <i>Constraints</i> model the <i>security constraints</i> associated with these profiles.	<i>Security constraints</i> are related to the <i>user profile</i> for the mere appearance in the same diagram (without any explicit association).
Luján-Mora et al. (2002a)	Multidimensional data	UML <i>Packages</i> are stereotyped as <i>star</i> , <i>fact</i> , and <i>dimension packages</i> .	<i>Packages</i> are used to visually separate diagram concerns.
Luján-Mora et al. (2004)	Data mappings	UML <i>Classes</i> are stereotyped as <i>attributes</i> .	Extending <i>attributes</i> from UML <i>Classes</i> instead of UML <i>Properties</i> allows authors to visually associate <i>Properties</i> among them.
Kurz et al. (2006)	Data mappings	UML <i>Classes</i> are stereotyped as <i>mapping operations</i> .	Extending <i>operations</i> from <i>Classes</i> allows authors to visually relate these mapping operations.
Stefanov and List (2007)	Usage	UML <i>InformationFlows</i> are stereotyped as <i>usage information channels</i> . <i>Usages</i> relate stereotyped <i>Classes</i> .	Having <i>Classes</i> related by <i>InformationFlows</i> clearly violates the UML syntax, but this way of depicting usage is visually convenient.
Golfarelli and Rizzi (2008)	What-if analysis	UML <i>Classes</i> are stereotyped as <i>scenarios</i> . Their <i>Properties</i> are stereotyped as <i>scenario parameters</i> .	The <i>scenario parameters</i> stereotype is not needed, but has been defined in order to visually label the <i>scenarioProperties</i> .
Pardillo et al. (2009)	Data flows	<i>Data cubes</i> are represented as <i>String</i> tags associated with the <i>cube element</i> stereotype.	The <i>String</i> tag is used as a visual reference to another diagram that defines the <i>data cube</i> .
Stefanov et al. (2005)	Business intelligence	UML <i>Activity Pins</i> are stereotyped as <i>data &amp; presentation objects</i> , and <i>data repositories</i> .	These data objects and repositories stereotypes are defined over the <i>Pin</i> metaclass in order to be visualised on the edges of the containers, even if the abstract syntax of these concepts are more closely related with the UML <i>Class</i> semantics.
Stefanov and List (2007)	Data states	UML <i>Classes</i> stereotyped as <i>data objects</i> and <i>data repositories</i> are linked to UML <i>States</i> by <i>Associations</i> .	The <i>Association</i> between <i>Classes</i> and <i>States</i> clearly violates the UML syntax, but is visually convenient.
Zubcoff and Trujillo (2005)	Association rules	Several data mining <i>attributes</i> and <i>parameters</i> , such as <i>inputs</i> , <i>cases</i> , or <i>minimum support</i> , are modelled as tags of the corresponding analysis <i>fact</i> .	The semantics of <i>attributes</i> and <i>parameters</i> suggest that they should be modelled as instances of a UML metaclass and not at an upper abstraction level (as stereotype tags). However, the visualisation convenience has prevailed.
Zubcoff and Trujillo (2006)	Classification	UML <i>Properties</i> are stereotyped as <i>mining attributes</i> .	A <i>mining attribute</i> is a composed name that represents the path of a <i>Property</i> owned by some other <i>Class</i> , thus, being semantically closer to a kind of <i>Relationship</i> rather than a <i>Property</i> . However, this representation is visually more concise.
Zubcoff et al. (2007)	Clustering	UML <i>Classes</i> are stereotyped as <i>mining attributes</i> . The <i>mining attributes</i> own a reference (implemented with a tag value) to another <i>Property</i> .	An <i>Attribute</i> is modelled to visually simplify the referred <i>Property</i> name and path.
Luján-Mora et al. (2002b)	Multidimensional data	UML <i>Classes</i> are stereotyped as <i>dimensions</i> of analysis and aggregation <i>bases</i> .	<i>Dimensions</i> are logical units for <i>bases</i> ; however, authors have defined them as <i>Classes</i> in order to obtain a visually flat representation.
Marcos et al. (2001)	Object-oriented and relational data	UML <i>Classes</i> and <i>Properties</i> can both be stereotyped as <i>arrays</i> ( <i>VARRAY</i> stereotype).	The <i>VARRAY</i> stereotype is used to visually indicate both the definition (when applied over <i>Classes</i> ) and the use (when applied over <i>Properties</i> ) of <i>arrays</i> .



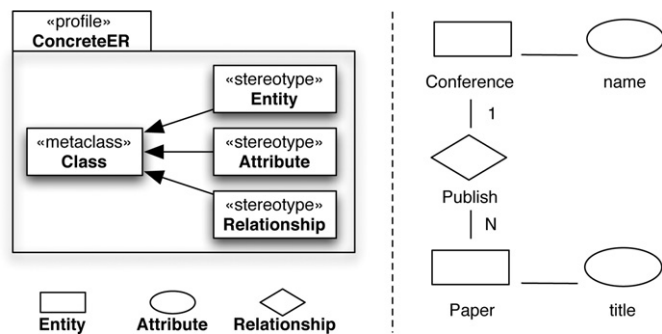
**Fig. 2.** ER notational variations for denoting the maximum cardinality of relations






**Fig. 3.** ER diagram corresponding to a conference-review system following Chen's notation (Chen, 1976) (left side) and UML (right). Despite the notational differences, both diagrams express the same sentences about conferences and papers: "every conference publishes many papers, but every paper is published in only one conference", "every conference has a name" and "every paper has a title".

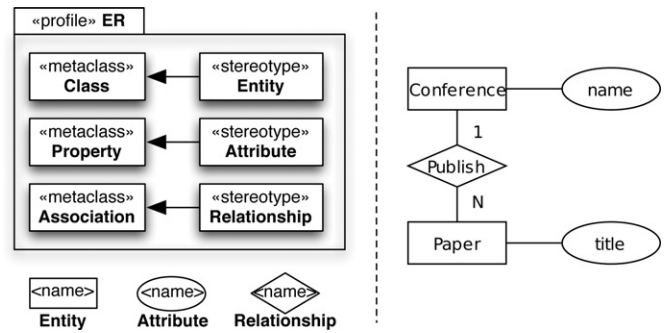


**Fig. 4.** AbstractER profile for ER diagramming (iconography below). This UML profile extends from metaclasses whose abstract syntax is similar to the abstract syntax of the ER stereotypes. However, the concrete syntax only vaguely resembles Chen's notation, due to the lack of expressiveness of the UML notation. This profiling results when UML metaclasses are extended for abstract-syntax reasons.



**Fig. 5.** ConcreteER profile for ER diagramming (iconography below). This UML profile extends from metaclasses whose concrete syntax can be easily adapted to the concrete syntax of the ER stereotypes. However, (i) such concrete syntax is still not able to perfectly match Chen's notation, due to the lack of expressiveness of the UML notation and (ii) the abstract syntax needs to be heavily redefined, since the original abstract syntax of some of the stereotyped UML metaclasses greatly differs from the abstract syntax of the ER stereotypes. This profiling results when UML metaclasses are extended for concrete-syntax reasons. Note that the lines linking the different stereotypes stand for (non-stereotyped) UML Associations, under the standard UML notation.

The ConcreteER profile, shown in Fig. 5, avoids this mismatch by extending metaclasses whose notation is closer to the one intended. This UML profile owns the same three stereotypes than did its AbstractER counterpart to denote ER metaclasses. However, unlike it, they are all defined over the *Class* metaclass. With respect to their notation, it can be observed (see Fig. 5, right side) how the corresponding icons (i.e., , , and ) have been arranged to mimic the ER diagrammatic notation (Fig. 3). In this



**Fig. 6.** Syntactically-decoupled ER profile (extended iconography below). Both the abstract and concrete syntax can be modelled properly due to (i) their decoupled definition and (ii) the use of a rich notational profiling, such as DI (Object Management Group, 2006b). This UML profile requires the enrichment of the profiling capabilities of UML.

way, ER Attributes can be depicted as connected to, and not contained by, the corresponding *Entity* notation. However, this decision hampers the abstract syntax of the extended metaclasses. ER attributes are not UML classes; UML classes own properties, are associated to others, can be abstract, have identity, behaviour, etc., which are features that ER attributes do not share. The same happens with ER relationships.<sup>5</sup>

Comparing both the diagrams modelled by the AbstractER and ConcreteER profiles, it can be observed how the ConcreteER profile allows to mimic much more accurately the original ER notation. However, a more careful analysis identifies several problems in this way of UML profiling. The most obvious one is the increase of model structural complexity, that is, the need for an extra collection of artificial metaclass instances that would not be necessary if the AbstractER profile had been used instead. The left side of Fig. 7 shows a UML object diagram corresponding to the ConcreteER diagram of Fig. 5 (link end names are hidden for conciseness). In this figure, the ER entities Conference and Paper, which are stereotyped UML classes, are linked by sequences (Property, Association, Property) with its corresponding attributes (name and title), which are also stereotyped UML Classes. The same happens with the relationship (Publish). This diagram contrasts with the one presented in the right side of Fig. 7, which shows the AbstractER-diagram counterpart (Fig. 4), and where each Entity, Attribute and Relationship is mapped into instances of the extended metaclasses Class, Property and Association, respectively.

However, the most important problem of the ConcreteER profile is not this structural complexity (which, after all, modelling tools usually hide), but the additional syntactic constraints that are needed to restrict the abstract syntax of the extended metaclasses. These constraints must assure that the profiled UML models are well-formed according to the ER abstract syntax.

**Example 4.1.** The constraint that states that ConcreteER Entities mapped into UML Classes must not own any UML attribute (since ConcreteER Attributes are also mapped into Classes) is modelled in OCL as:

```
context Entity inv noAttributes:
self.baseClass.ownedAttribute->isEmpty()
```

**Example 4.2.** The constraint that states that ConcreteER Entities only connect to Attributes and Relationships is modelled in OCL<sup>6</sup> as:

<sup>5</sup> The ER profile presented in Fig. 6, which overcomes the limitations of the previous ones, will be described in Section 6.

<sup>6</sup> *isStereotypedBy* and *getAssociations* are convenience operations that were defined in Zubcoff et al. (2009).

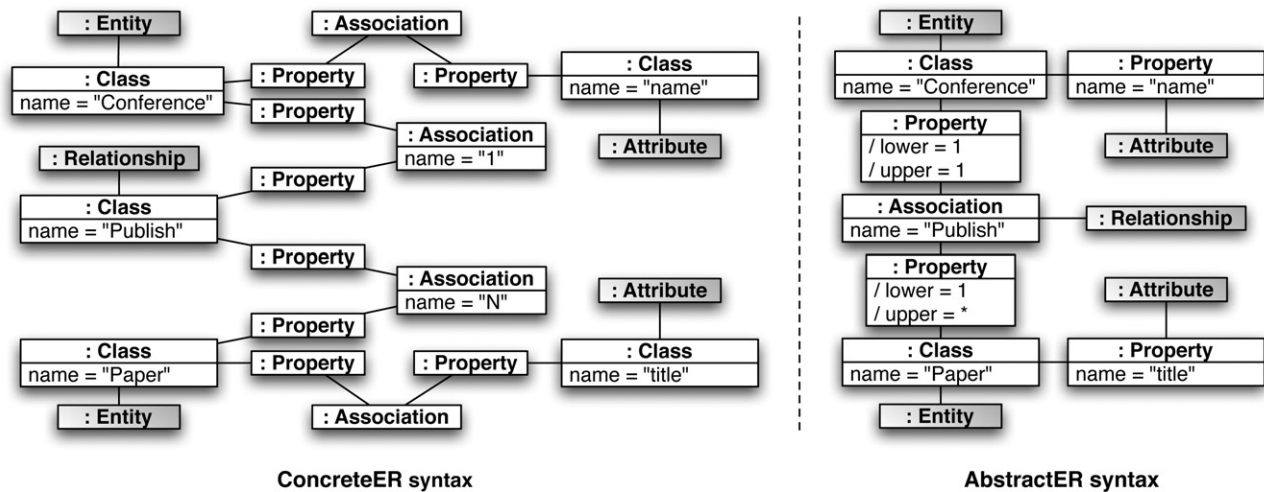


Fig. 7. Syntactic augmentation due to profiling by notational reasons (stereotypes shadowed).

```
context Entity inv associativity:
  self.getAssociations()->forall(a |
    a.memberEnd.type->exists(t |
      t.isStereotypedBy(Attribute) xor
      t.isStereotypedBy(Relationship)))
```

Summarising, artificial complexity and expressive constraints are the result of profiling UML for notational reasons. Part of this complexity can be overcome if the UML profile is modelled following the abstract-syntax closeness criterion. However, this cannot always be done without sacrificing the intended notation. A solution to this problem is of paramount importance in the context of MDE technologies (Favre, 2004), which promote diagrams (visual models) as the design artefacts that drive software development.

Next section presents how to model UML profiles that are at the same time syntactically and notationally sound.

## 5. UML notation with DI

UML diagrams are modelled by DI (Object Management Group, 2006b), whose metamodel is presented in Fig. 8. The DI metamodel is organised around an inheritance hierarchy, whose root is the *DiagramElement* metaclass. DI defines the visibility of the modelling elements (*DiagramElement::isVisible* property) and registers a collection of visual *Properties* (e.g., colours, fonts, line styles, and so on) that act as parameters of the rendering routines. The *DiagramElement* metaclass is specialised into *LeafElements*, which can be of different kinds (e.g., *TextElement*, *Image*), and *GraphElements*, also of different kinds: (i) *GraphNode*s, which contain graphical information (position, size, style, etc.) about certain UML metaclasses, such as *Class* or *Property*, (ii) *GraphEdge*s, which manage the same information about UML metaclasses that serve to connect others, such as *Association* or *Dependency*, and (iii) *GraphConnectors* between them.

These *GraphElements* are the backbone of DI models, and can be complex structures, made up of other nested *DiagramElements*, following a composite pattern (Gamma et al., 1995). For instance, the *GraphNodes* of a *Class* contain nested *GraphNodes* for modelling *Property* and *Operation* compartments, among others. Moreover, a *Property* compartment contains the *GraphNodes* of such *Class* *Properties*, which in addition contain *GraphNodes* modelling their visibility, name or type. Such nested graphs are rendered by means of a pre-order traversal process through the ordered contained relationship of a *GraphElement*. This means that the deeper nodes are rendered on top of the shallower ones,

whereas the siblings are rendered following the order in which they are contained.

For each *GraphNode*, certain constraints apply. These constraints, which determine the allowed nesting combinations, are specified in DI by means of textual templates.

**Example 5.1.** The diagramming of UML *Classes* is constrained as follows (Object Management Group, 2006b):

```
Name: DiClass
DI-Element: GraphNode{<Class>}
Nested elements: NameCompartment,
  [CompartmentSeparator, AttributeCompartment],
  [CompartmentSeparator, OperationCompartment],
  [CompartmentSeparator, ToolCompartment]*
```

where *DiClass* is the label given to the *GraphNode* whose *CoreSemanticModelElement* is linked to the *Class* metaclass, and *NameCompartment*, *CompartmentSeparator*, etc., are the labels given to the corresponding nested *DiagramElements*. Notice that, in this template, square brackets mean 'optional' and the asterisk means 'many'.

In this way, well-formed DI models can be validated by checking the *GraphNode* nesting structure against these DI constraints.

In order to model DI-based notations, values must be provided for *DiagramElement* *Properties* (such as colour, font, etc.). However, the rendering of concrete graphics (such as rectangles, circles, or complex shapes) of UML diagrams requires rendering routines. They are associated to *GraphElements* by a semantic model relationship (see Fig. 8). All these routines are reused among UML diagrams. In fact, DI models are the parameters for such routines, whether they be built-in or custom (Section 6.3).

Rendering routines are not (formally) modelled by neither DI nor UML. They are described informally by means of recommendations and examples included in the UML specification (Object Management Group, 2009) (see Section 'Notation' of any UML metaclass). In this way, UML modelling tools are free to implement them at their will as long as their diagrams have the expected appearance.

Regarding UML profiles, they reuse the DI rendering routines associated with the base metaclasses (see Fig. 9). These rendering routines can receive *Stereotype* icons as parameters. The DI rendering routines are associated with UML metaclasses (be them stereotyped or not) through *CoreSemanticModelBridges* that connect them with UML *Elements*. This means





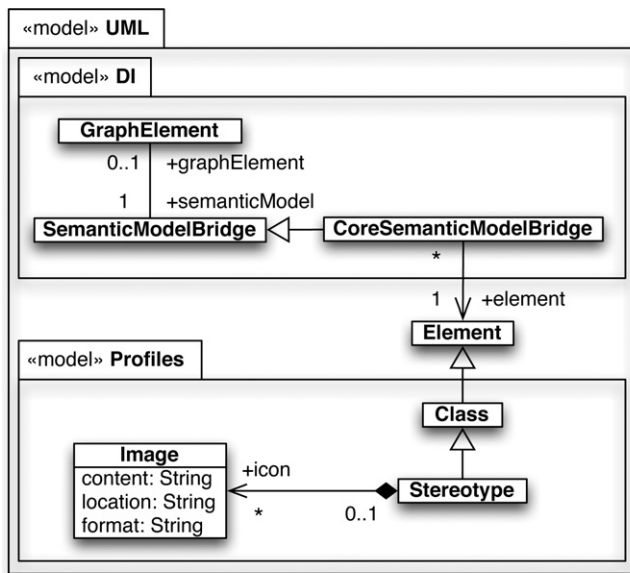


Fig. 9. Relationship between DI and UML-profile metamodels.

- A set of constraints over *DiagramElements*. These constraints involve both *DiagramElements* and their corresponding UML modelling *Elements* (e.g., *Stereotypes*), establishing the correct binding between concrete and abstract syntax.

**Example 6.2.** The ER profile may own a constraint that states that ER Entities, denoted by DI *GraphNode*s, are associated by ER Relationships, denoted by DI *GraphEdge*s. This constraint may be stated in natural language or in some formal language that enables automatic model checking, such as OCL (see Appendix C).

- A set of rendering routines that, parameterised with *DiagramElement* Properties, depict the concrete notation of a profiled UML model.

**Example 6.3.** A rendering routine in Java that diagrams the notational variations of Fig. 2 could be as follows:

```
void renderDiRelationship(di.GraphEdge ge) {
    drawLink(ge);
    di.Property p = ge.getProperty('Notation');
    switch (p.getValue()) {
        case 'Chen': 'EER':
            drawDiamond(ge);
            drawCardinality(ge);
            break;
        case 'ODMG': 'G.Everest':
            drawArrowHeads(ge);
            break;
    }
}
```

Next sections further develop each one of these components. The ER profile of Fig. 6, where abstract and concrete syntaxes are decoupled, will serve us to illustrate them.

### 6.1. Definition of DI properties

DI models may be seen as graphs whose nodes store graphical information to render a particular notation. This information is modelled by means of *DiagramElement* Properties. They are propagated along the nested nodes in the graph: every Property shared among all the nodes under the same *DiagramElement*, which acts as a container, can be simply modelled as a Property of this container.

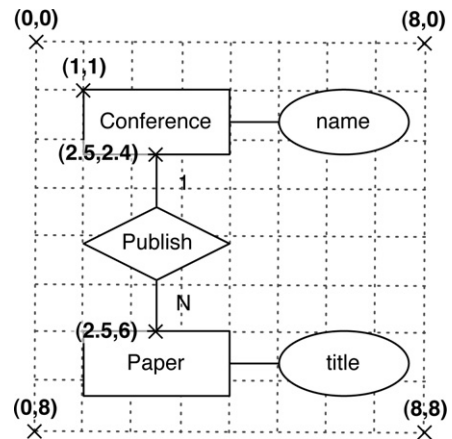


Fig. 10. Graphical information about positions to be stored by DI.

DI provides several standard Properties such as *FontFamily*, *LineStyle*, or *BackgroundColor* (Object Management Group, 2006b) (p. 13). Each of them is associated with the corresponding *DiagramElements*.

**Example 6.4.** *LineStyle* is a Property of the UML Association notation; however, *BackgroundColor* is not. The UML Class notation, on the other hand, has a *BackgroundColor* associated, but not a *LineStyle*.

For each *Stereotype*, several additional Properties may be modelled in order to parameterise the domain-specific notation. Each Property models a key and its permissible values (see Example 6.1). With this information, which is linked to the corresponding *DiagramElements* (those whose *semanticModel* refers to the *Stereotype*), the associated rendering routine can properly provide the concrete notation. Therefore, both rendering routines and Properties are interdependent, being the latter the parameters for the former ones.

**Example 6.5.** ER diagrams model, among others, the following Properties:

**Standard** *FontSize*, *LineThickness*, *BackgroundColor*, and *ShapeThickness* (Object Management Group, 2006b) (p. 13).

**ER-specific** Notation as defined in Example 6.1.

Fig. 10 shows a sample grid (managed by a hypothetical modelling tool) where the *DiagramElements* of Fig. 6 have been located in the two-dimensional space of the drawing canvas.

**Example 6.6.** *Conference* is located at  $(x, y) = (1, 1)$  (which corresponds to its upper-right corner position), whereas the related *GraphEdge* with the *Publish* diamond is located at  $(2, 3)$  and  $(2, 4)$  as waypoints (Object Management Group, 2006b).

This graphical information is modelled by DI as is shown in Figs. 11 and 12. These figures show an excerpt<sup>9</sup> of the DI model of the *DiagramElements* related to *Conference* & *Publish* nodes, respectively. The *Diagram* instance models the Properties that are inherited by the contained *GraphNode*s that, in this case, amount to *FontSize* (Fig. 11) and *LineThickness* (Fig. 12). The involved *GraphNode*s are related by means of container/contained relations. *GraphNode* positions are modelled as *Points*, defined as relative coordinates to the position of the container *GraphNode*. The same happens with size.

<sup>9</sup> For the sake of conciseness, some mandatory elements, such as *GraphNode::positions*, are not shown in these diagrams.

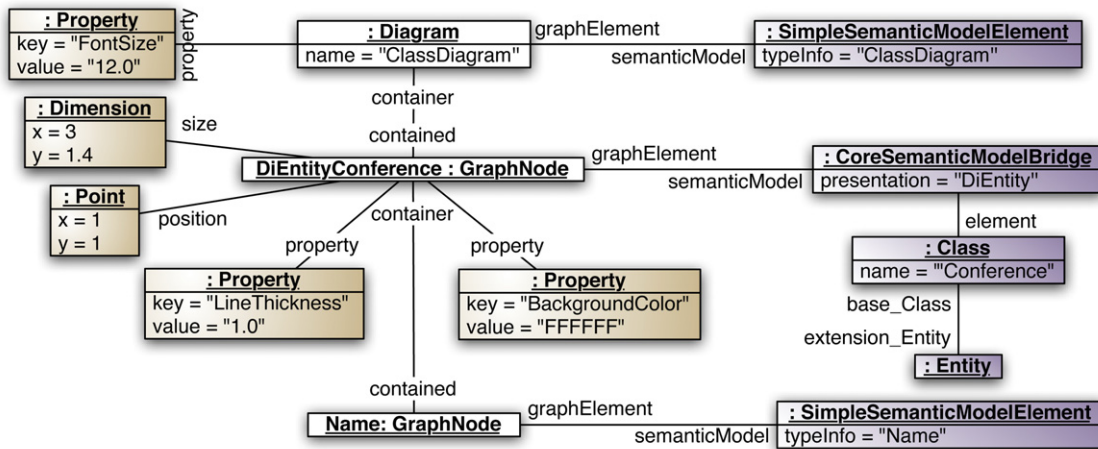


Fig. 11. An excerpt of the DI model for the diagram of Fig. 10 (details on Conference node).

Moreover, `DiagramElements` may be connected with the UML modelling `Elements` which they denote. Those `semanticModels` are shadowed in Figs. 11 and 12.

## 6.2. Definition of DI constraints

Additional constraints need to be introduced over the DI meta-model in order to assure well-formed DI models. DI characterises the syntactic rules that make up the standard UML notation as textual templates, as was shown in Example 5.1. Another possibility,

more familiar to UML practitioners, is to express them by means of OCL constraints (Object Management Group, 2006a). In this article, the last option is advocated due to the strong synergies between UML and OCL.

Constraints are associated with `DiagramElements` to indicate (a) which UML modelling `Element` is related to each `GraphElement`, (b) how `DiagramElements` are nested (e.g., name labels inside entity boxes), or (c) how `GraphElements` are interconnected (e.g., `GraphEdges` representing relations must connect two `GraphNodes` representing entities).

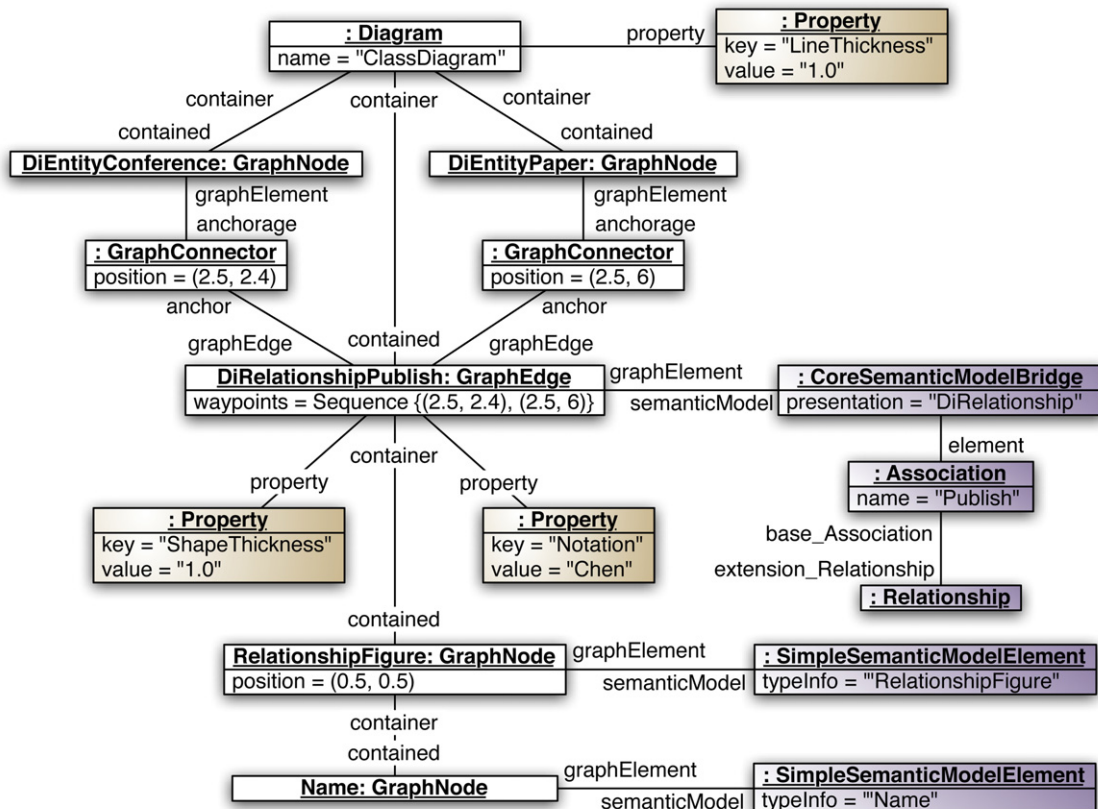


Fig. 12. An excerpt of the DI model for the diagram of Fig. 10 (details on Publish node).

**Example 6.7.** ER diagrams model the following `GraphElements`:

**DiEntity** for drawing rectangles of entities;  
**DiAttribute** for drawing attributes of entities,  
**EntityAttributeLink** for the lines that connect them,  
**AttributeFigure** for drawing attribute bubbles;  
**DiRelationship** for drawing relationships between entities,  
**RelationshipLink** for drawing lines that connect them,  
**RelationshipFigure** for drawing their diamonds,  
**RelationshipCardinality** for drawing cardinality labels.

The OCL constraint associated to the `DiEntity` `GraphNode` can be expressed as follows<sup>10</sup>:

```
context GraphNode inv DiEntity:
self.means('DiEntity', ■ntity')
implies self.contains('Name')
```

This constraint specifies that every `GraphNode` denoting an ER Entity has to contain a `DiagramElement` that is in turn constrained by the `Name` constraint, which is a standard constraint defined by DI (Object Management Group, 2006b) (Annex C.5, p. 66).

OCL constraints can be cumbersome to codify. However, they can be simplified by using syntactic short-hands by means of convenience operations.

**Example 6.8.** The following convenience operations were defined for the `DiEntity` constraint as syntactic short-hands:

```
DI::GraphElement::means (String p, s): Boolean;
DI::GraphElement::contains(String t): Boolean;
```

The first one evaluates whether a `GraphElement` is related to an occurrence of a given `Stereotype` `s`, which indeed has a presentation `p`. The second operation evaluates whether a `GraphElement` contains a `DiagramElement` that fulfils a constraint with name `c`.

### 6.3. Definition of DI rendering routines

The rendering pipeline of the UML notation is based on DI models. Enriching such models for profiling notations can be done by modelling new DI `GraphElement` instances for each `Stereotype`.

Such `Stereotypes` are connected to their `GraphElement` counterparts by DI `CoreSemanticModelBridge` instances (see Fig. 9). The rendering pipeline is driven by the `Stereotype` identification, which serves to select which rendering routine must be called to depict the `GraphElement`.

Fig. 13 shows the rendering pipeline that takes place whenever a domain-specific notation is profiled with DI. Its input parameters are the DI model and its associated UML model, both of them stored in XMI (Object Management Group, 2007a). It is important to note how, before proceeding with the rendering itself, both the UML and the DI model must be validated against the existing OCL abstract and concrete syntactic constraints.

In general, the rendering pipeline may be supported by any programming language whose graphical primitives are expressive enough to render the DI model. If, as happens in the ER case study, UML+DI models are stored as XMI models, the rendering can be seamlessly described in a XML-based transformation language such as XSLT.<sup>11</sup> The rendering-pipeline output is a view of the UML input model that, at the same time, conforms with the DI model. Again, this view can be described in any graphical format, such as SVG.<sup>12</sup>

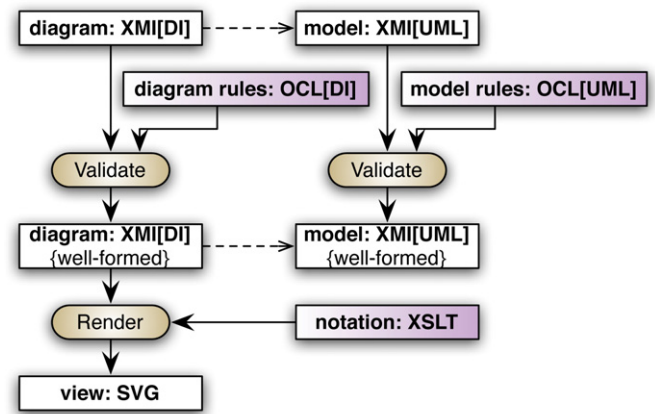


Fig. 13. UML+DI rendering pipeline with XSLT and SVG.

Each `DiagramElement` has a rendering routine associated that is responsible for drawing the desired notation. Such routines differ depending on the `DiagramElement` subtype (see Fig. 8):

**Leaf Elements** are directly rendered, since they do not require any additional (external to DI) syntactic information.

**Example 6.9.** A `LeafElement` rendering routine may write a `TextElement`, draw an `Image`, and so on.

**Graph Elements with Simple Semantics** are rendered according to a textual parameter that is stored in the `SimpleSemanticModelElement::typeInfo` attribute.

**Example 6.10.** A `GraphElement` associated with a `SimpleSemanticModelElement` whose `typeInfo = 'attributeCompartment'` is rendered as a box properly connected to the notation of the corresponding `Class` name and `Operation` compartments.

**Graph Elements with Core Semantics** are rendered according to some parameters that are accessible as `CoreSemanticModelBridge::elements` and the `SemanticModelBridge::presentation` selected.<sup>13</sup>

**Example 6.11.** A `GraphElement` associated with a `CoreSemanticModelBridge` whose `element type` is `Class` and has the default `presentation` (coded as an empty string by DI) is rendered according to the well-known UML `Class` notation.

These attributes are accessed by each particular `DiagramElement` rendering routine through the `GraphElement::semanticModel` property. It is worth noting how, thanks to the inheritance hierarchy of `SemanticModelBridge` (see Fig. 8), `Simple` & `CoreSemanticModelBridge` may also specify presentation variations of the standard UML notation.

**Example 6.12.** UML *actors* can be denoted as stick-mans or rectangles.

DI allows to model rendering routines in a way that fosters model reuse. This can be accomplished by defining: (i) a general rendering pipeline that deals with `GraphNode` traversal order and specific rendering routines selection and (ii) a set of specialised rendering routines, each one corresponding to a given `SimpleSemanticModelElement::typeInfo` or

<sup>10</sup> See Appendix C for a comprehensive list.

<sup>11</sup> <http://www.w3.org/TR/xslt>

<sup>12</sup> <http://www.w3.org/Graphics/SVG>

<sup>13</sup> A third metaclass, `Uml1SemanticModelBridge` is also kept in DI just for compatibility reasons with older UML versions; in UML 2.0, it has been replaced by the more expressive `CoreSemanticModelBridge::element`.

CoreSemanticModelBridge::element attribute value. In this way, rendering profiled notations with DI simply consists in referencing such Stereotypes through CoreSemanticModelBridge::elements (as any other UML model Element might be referenced), and adding a rendering routine to deal with the Stereotype.

There are several programming techniques, such as reflection or declarative coding, that permit to add these new code snippets and make them be called by the rendering pipeline without needing to recompile the whole environment. In order to illustrate this process, we have taken the case of XSLT+SVG, since they are both declarative, XML-based, standard, widely known and supported, all of them characteristics that make them ideal candidates to be integrated with DI and UML.

**Example 6.13.** The XSLT code for rendering in SVG the ER Entities can be defined through the following XSLT template:

```
<xsl:template name='DiEntity''>
  <g id='DiEntity''>
    <rect
      x='{position/@x}' y='{position/@y}'
      width='{width}' height='{height}'
      style='fill:white; stroke:black'/'>
    </g>
  </xsl:template>
```

This template translates DI GraphElements of Entities into SVG predefined rectangle elements (see below). Rectangles have attributes for size<sup>14</sup> (i.e., width, height), style and position (i.e., x, y), that must be set in order to render the Entity notation.

Notice that this template works in conjunction with a Name template that renders the Entity name as a text label inside such Entity rectangle.

Two additional features are shared between DI and SVG that make the latter one a suitable candidate for rendering output. First, SVG can manage relative positions, in the same way as DI does for DiagramElements. For this purpose, SVG defines a group transform element as follows:

```
<g transform='translate({position/@x},{position/@y})''>
```

Second, SVG can also support attribute inheritance, in the same way as the Property inheritance is modelled along DI DiagramElements.

**Example 6.14.** Back to Example 6.13, the DiEntity template explicitly codifies the fill and stroke colour elements. However, other applicable attributes, such as shape opacity or stroke-width, are implicitly inherited from a container SVG element.

Given a SVG template, a SVG rendering engine can obtain the diagram with the concrete profiled UML notation.

**Example 6.15.** Given the template of Example 6.13 and the DiagramElement locations shown in Fig. 10, the DiEntity template can be instantiated for rendering the ER Entity Conference as

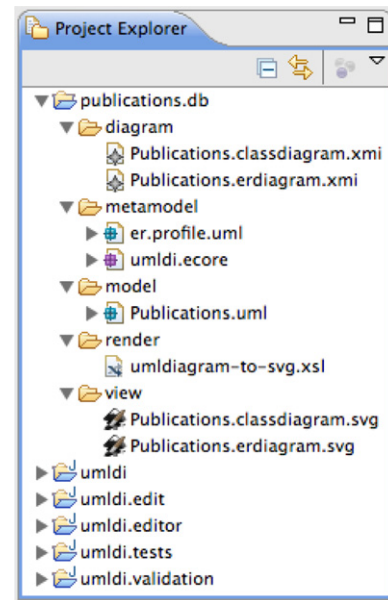


Fig. 14. Eclipse projects involved in the prototype implementation.

follows:

```
<g id='DiEntity''>
  <rect
    x='1' y='1'
    width='3' height='1.4'
    style='fill:white; stroke:black'/'>
  </g>
```

## 7. Validation

In order to show the feasibility of using DI-based UML profiles for the decoupling of abstract and concrete UML syntaxes, and how existing environments should be extended to support the approach, a prototype has been implemented under the Eclipse development platform.<sup>15</sup> Readers interested in examining the source code can download it from '<http://jesuspardillo.com/lib/2010/jss>'. Eclipse provides a framework for software development, but also for building development tools, such as those based on the model-driven paradigm, thanks to an extensible architecture. In particular, the implemented prototype is supported by two projects: the Eclipse Modelling Project and the Web Tools Platform (WTP) project. On the one hand, the former project includes an EMF component for a MOF-like metamodeling, an EMF Validation Framework, and other components, homonyms of their aim, such as OCL or UML2. On the other hand, the Web Tools Platform (WTP) project supports the prototype implementation by providing an XSLT engine.

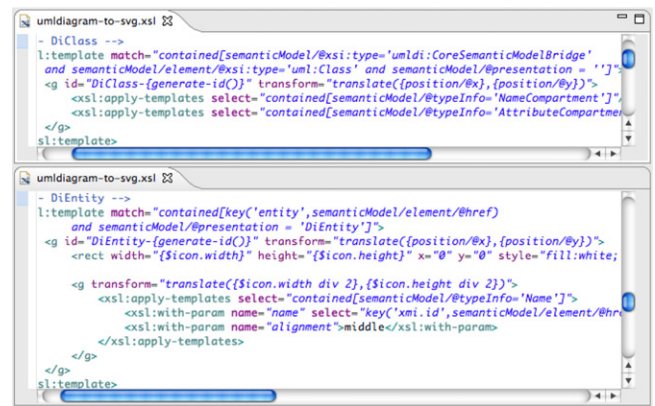
Fig. 14 shows the Eclipse projects that implement the prototype. The publications.db project contains the code that has been manually implemented for this prototype, whereas umldi, and umldi.\* contain code that has been automatically generated by Eclipse to provide a convenient tree-editor for DI models over UML. In addition, umldi.validation contains the OCL constraints that validate the DI models manipulated with such editor.

The publications.db contains the generic code for the syntactically-decoupled ER profile (see Fig. 6) and the DI metamodel (metamodel folder), and the rendering routines implemented in XSLT for both non-profiled and ER-profiled UML models

<sup>14</sup> For the sake of simplicity, this example defines size with two global variables. However, shape sizes are generally calculated from the contained DiagramElements (Object Management Group, 2006b) (p. 14).

<sup>15</sup> <http://www.eclipse.org>





**Fig. 17.** XSLT renderer to SVG for class diagrams and ER-profiled notations.

nalities according to what was specified in the DI metamodel. The code presented in the figure is the constraint provider for the `DiEntity` constraint (see [Appendix C](#), note that convenient operations and derived properties listed in the appendix are coded unfolded in the prototype) that validates the `GraphNode`s that represent `EREntities`.

Each validated DI model can be then rendered in order to obtain the SVG views. Fig. 17 shows the implemented XSLT code. The XSLT template that renders non-stereotyped UML `Classes` is presented at the top of the figure, whereas the `Entity`-stereotyped counterpart is presented at the bottom. The latter matches `GraphElements` that represent UML elements stereotyped as `ER Entities` (for which the `ER` presentation, identified as `DiEntity`, has been selected). It is worth mentioning that the `ER` rendering routines coexist with the traditional UML routines without interfering with them, what means that UML routines can be conservatively extended by DI-based profiles.

Based on both the built-in and the coded Eclipse tree-editors, validation mechanisms, and rendering routines, the running example on conference publications has been implemented as it is shown in [Fig. 18](#). At the bottom of the figure, the  $\text{ER}$ -profiled UML (abstract) model is presented. Each involved model element has been defined and stereotyped with the corresponding  $\text{ER}$  stereotype. In the middle of the figure, the DI models for both the non-profiled UML notation (left-hand side) and the  $\text{ER}$ -profiled notation (right-hand side) are presented for comparison purposes. During this comparison it becomes evident the differences regarding the containment hierarchy of the  $\text{DI GraphNode}$  that stores the `Conference Class`: on the left-hand side, the hierarchy is the one established by DI ([Object Management Group, 2006b](#)) (Annex C.2.1, p. 58), while on the right-hand side the hierarchy is defined according to the  $\text{ER Entities}$  ([Appendix C.1](#)). At the top of the figure, the XML code generated by our prototype for the SVG view of each DI model is shown. The result of rendering this code (by any SVG viewer) corresponds to the ER diagrams that were presented in [Figs. 3 and 6](#).



The idea of splitting up abstract and concrete syntaxes, far from being a new concept, is a basic principle in software engineering. In fact, this principle is a target of every successful environment for the definition of domain-specific languages. Two of these leading commercial solutions are MetaCase MetaEdit+ and Microsoft DSL

**Fig. 16.** Constraint provider for the OCL constraints of the **ER** profile.

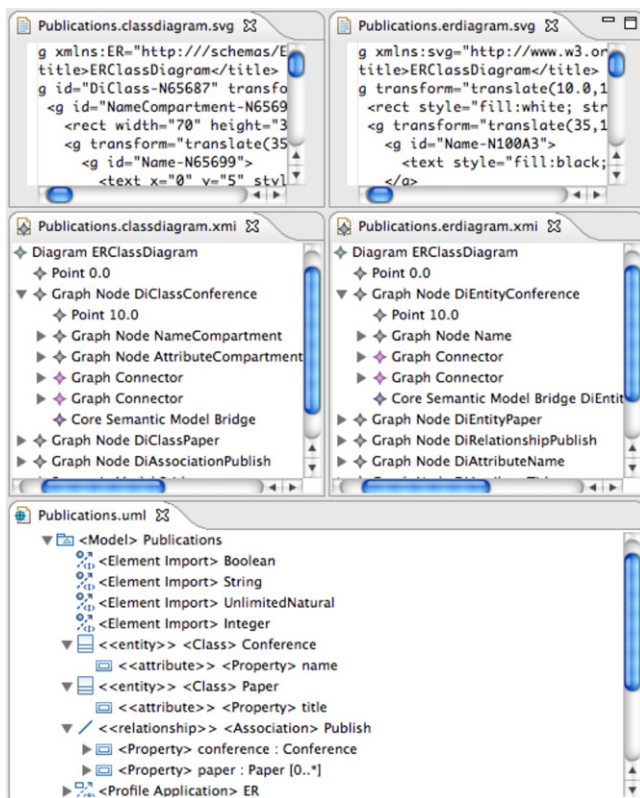


Fig. 18. Implementation of the running example with SVG views as output.

Tools. On the one hand, MetaCase MetaEdit+<sup>16</sup> provides designers with a built-in and extensible library of graphical primitives, manageable with a visual editor. Such primitives can be adapted to the model properties being represented. It also supports multiple diagrams per model, and matrix and tabular views. However, this separation between abstract and concrete syntax is not completely achieved, since graphics need to be managed with respect to some modelling element. On the other hand, the Microsoft DSL Tools<sup>17</sup> provide a built-in visual editor for metamodeling both concrete and abstract syntaxes. In these tools, the lack of complete separation between abstract and concrete syntax is reflected in the fact that only one visualisation can be associated with each concept. Concerning open-source solutions, the Eclipse project clearly leads the market. The EMF/GEF solution provides a simple tree-view editor for concrete syntax that needs to be complemented with additional programming tasks. It is worth noting how the abstract syntax provided by EMF can be aligned with MOF (Object Management Group, 2006c), the metamodel of both UML and DI. The GMF solution improves EMF/GEF with several metamodels that allow to define concrete and abstract syntaxes, their mappings, and simple user-interface customisation. These metamodels are manageable via tree-view editors, and the design is guided by wizards. GMF allows several graphic primitives to represent the same modelling element. Finally, the openArchitectureWare solution is based on text editors (Xtext and Xpand). Concrete syntax is modelled as a context-free grammar. Models are managed by means of a textual syntax. Abstract and concrete syntaxes are related via a one-to-one mapping, and additional support is needed in order to provide graphics.

The consequences of our proposal for abstract and concrete syntax decoupling for UML go far beyond the mere enrichment of the profiled UML notations. Syntax decoupling in UML profiles opens the path to notation reuse, by simply associating the same notation to different metaclasses (of the same or different metamodels). Also, such decoupling allows modelling different notations for the same metaclass, adding flexibility to the model visualisation. In this way, a modeller that is used to, e.g., class diagrams following Booch's notation (Booch et al., 2007), could define a notational extension that allowed him to apply this notation while still remaining inside the boundaries of UML.

The extension from two metamodels (UML and DI) that cover different purposes emphasises the different concerns involved in UML profiles, and supports a new classification that clarifies the profile purpose:

**Syntactic Profiles** whose stereotypes have no extended notation over UML.

**Notational Profiles** oriented towards the provision of notational changes, without altering the syntax of the extended metaclasses.

**Hybrid Profiles** that imply both syntactic and notational changes over UML.

This classification is orthogonal to the definition of conservative vs. non-conservative extensions (Turski and Maibaum, 1987).

There are two main drawbacks to the extension presented in this article. First, DI is focused on node-link visualisations. For this reason, were other the visualisation needs (e.g., matrix visualisations, which have proven useful for scalability reasons by Ghoniem et al. (2004)), DI would not suffice, and other notation metamodels would need be used to extend the UML notation instead. Second, the introduction of a notation metamodel increases the learning curve of the UML-profile modeller.

Finally, UML profiles usually depend on modelling tools. These modelling tools provide adaptable mechanisms to facilitate the application of Stereotypes to certain modelling Elements. These 'aids' can favour the extension of one metaclass over another for mere tool convenience reasons. A clear example of this fact occurs when a certain Stereotype needs to extend from the UML metaclass AssociationClass for syntactic reasons, (see e.g., data-warehouse facts in Luján-Mora et al. (2006)); in this case, it is common that designers, conditioned by the tool constraints, end up extending from simpler metaclasses (e.g., Association), because the artificial Stereotype can be applied with simpler interactions. The effect that modelling tools have on the effectiveness of the model visualisation (which is influenced by the way UML profiles are modelled) is usually dismissed in the MDE community, although it is recognised as a paramount factor in HCI. For example, the so-called 'cognitive dimensions of notations' (Green and Petre, 1996) state that one of the success factors for notations is precisely the environment in which they are defined and manipulated, that is, the UML modelling tool in which UML profiles are modelled. Although the prototype presented in this paper paves the way for the implementation of an operational UML modelling tool that supports DI-based UML profiles, its whole development remains an open line of work.

## Acknowledgements

This research has been partially funded by the DEMETER (GVPRE/2008/063) project from the Valencia Government (Spain) and the MESOLAP (TIN2010-14860) project from the Spanish Ministry of Science and Innovation. The work of Jesús Pardillo has been financed by the grant FPU AP2006-00332 from the Spanish Ministry of Science and Innovation. Special thanks to the journal's anony-

<sup>16</sup> <http://www.metacase.com/mep>

<sup>17</sup> <http://msdn.microsoft.com/en-gb/library/bb126235.aspx>

mous referees for their careful reviews and insightful comments, which have greatly contributed to improving the quality of this article.

## Appendix A. Glossary

Most of the notions and terms used along this text are already formalised in the mathematical formal language theory and model theory (e.g., language, model, semantics). Graph theory provides the supporting notions for UML modelling.

In particular, we highlight the supporting definitions given by UML about its two kinds of syntaxes:

**Abstract syntax (or syntax)** The set of UML modelling concepts, their attributes and their relationships, as well as the rules for combining these concepts to construct partial or complete UML models (Object Management Group, 2007b).

**Example Appendix A.1 (Numbers).** Any mathematical object can be an element of an abstract syntax.

**Concrete syntax (or notation)** Human-readable elements for representing the individual UML modelling concepts as well as rules for combining them into a variety of different diagram types corresponding to different aspects of modelled systems (Object Management Group, 2007b).

**Example Appendix A.2 (Arabic & Roman numbers).** Numbers can be codified under many notations. For instance, the symbols ‘5’ and ‘V’, in Arabic and Roman numeral system respectively, stand for the same conceptual entity, i.e., the number five.

For the sake of completeness, interested readers can consult Seidewitz (2003) for an interpretation of the ‘model-driven’ terms into the mathematical framework of the model theory. On the differences between abstract and concrete syntax, Baar (2008) provides a good discussion with many examples on both.

## Appendix B. List of acronyms

DI	Diagram Interchange
ER	Entity-Relationship
EMF	Eclipse Modelling Framework
GEF	(Eclipse) Graphical Editing Framework
GMF	Graphical Modelling Framework
HCI	Human-Computer Interaction
HUTN	Human-Usable Textual Notation
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
SVG	Scalable Vector Graphics
UML	Unified Modelling Language
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	Extensible Mark-up Language
XSLT	XSL (Extensible Style-sheet Language) Transformations

## Appendix C. OCL constraints for ER diagramming over diagram interchange

The collection of notational constraints needed for ER diagramming have been defined herein in OCL (Object Management Group, 2006a). This collection includes constraints for every `DiagramElement` that is stereotyped by any stereotype belonging to the ER

profile. Also, for the sake of conciseness, the following convenience operations and (derived) properties have been defined (see Fig. 8 as reference) for the `GraphElement` metaclass:

- ‘contains(String t): Boolean’ indicates whether a `GraphElement` contains a `GraphElement` whose semanticModel is a `SimpleSemanticModelBridge` with typeInfo t (Section 6.2). It is overloaded for also handling `OrderedSets` of `Strings`.
- ‘containsPresentation(String p): Boolean’ indicates whether a `GraphElement` contains a `GraphElement` whose semanticModel has a given presentation p. It is overloaded for also handling `OrderedSets` of `Strings`.
- ‘simpleModel: SimpleSemanticModelBridge’ is derived from semanticModel.oclassType(SimpleSemanticModelBridge).
- ‘coreElement: Element’ is derived from semanticModel.oclassType(CoreSemanticModelBridge).element.
- ‘means(String p, String s): Boolean’ indicates whether a `GraphElement` is attached to a model element stereotyped by the stereotype with name s by means of its `CoreSemanticModelBridge`, whose presentation equals to p.

Next, we present the OCL constraints over DI, organised by the related ER metaclass.

### C.1. Entity notational constraints

```
context GraphNode inv DiEntity:
self.means('DiEntity', entity) implies self.contains('Name')
```

`DiEntity` constraints the `GraphNode` whose semanticModel element is an ER entity (a UML Class stereotyped as `Entity`, see Fig. 6) that is to be presented as an ER entity (`DiEntity`). The constraint is so that such `GraphNode` should be the container (see Fig. 8) of a `DiagramElement` that represents the entity name (fulfilling the `Name` constraint).

### C.2. Attribute notational constraints

```
context GraphNode inv DiAttribute:
self.means('DiAttribute', attribute) implies
self.contains(OrderedSet {
entityAttributeLink', attributeFigure'})
```

`DiAttribute` constraints the `GraphNode` whose semanticModel element is an ER attribute (a UML Property stereotyped as `Attribute`) that is to be presented as an ER attribute (`DiAttribute`). The constraint is so that such `GraphNode` should be the container of two `DiagramElements` that represent the attribute link with the corresponding entity and the attribute figure (the characteristic bubble). They fulfil the `EntityAttributeLink` and `AttributeFigure` constraints, respectively.

```
context GraphEdge inv EntityAttributeLink:
self.simpleModel.typeInfo = entityAttributeLink' implies
self.anchor.at(1).graphElement.means('DiEntity', entity) and
self.anchor.at(2).graphElement.means(
'DiAttribute', attribute) and
-- semantics
let entity = self.anchor.at(1).graphElement.coreElement
.oclassType(Class) in
let attribute = self.anchor.at(2).graphElement.coreElement
.oclassType(Property) in
entity.ownedAttribute->includes(attribute)
```

`EntityAttributeLink` constraints the `GraphEdge` that represents the attribute link with the corresponding entity. The constraint is so that such the `GraphEdge` connects a `GraphNode` that represents an ER entity and another representing an ER attribute. In addition, this attribute should be one of such entity



(included as an ownedAttribute).

---

```
context GraphNode inv AttributeFigure:
self.simpleModel.typeInfo = <attributeFigure' implies
self.contains('Name')
```

---

AttributeFigure constraints the GraphNode that represents the attribute figure (the characteristic bubble). The constraint is so that such GraphNode should be the container of a DiagramElement that represents the attribute name (fulfilling the Name constraint).

### C.3. Relationship notational constraints

---

```
context GraphNode inv DiRelationship:
self.means('DiRelationship', 'Relationship') implies
self.contains(OrderedSet {'RelationshipLink',
'RelationshipFigure'}) and
self.containsPresentation(OrderedSet {
'RelationshipCardinality', 'RelationshipCardinality'})
```

---

DiRelationship constraints the GraphNode whose semanticModel element is an ER relationship (a UML Association stereotyped as Relationship) that is to be presented as an ER relationship (DiRelationship). The constraint is so that such GraphNode should be the container of four DiagramElements that represent the relationship link, relationship figure (the characteristic diamond), and two relationship cardinalities.

---

```
context GraphEdge inv RelationshipLink:
self.simpleModel.typeInfo = 'RelationshipLink' implies
self.anchor->forall(graphElement.means(
'DiEntity', 'Entity')) and
- - semantics
self.container.coreElement.oclaAsType(Association).memberEnd
.type = self.anchor.graphElement.coreElement.
oclaAsType(Entity)->asBag()
```

---

RelationshipLink constraints the GraphEdge that represents the relationship link with the corresponding entities. The constraint is so that such the GraphEdge connects two GraphNode that represent ER entities. In addition, the ends of the stereotyped Association should be the ones that are represented by the connected GraphNodes.

---

```
context GraphNode inv RelationshipFigure:
self.simpleModel.typeInfo = 'RelationshipFigure'
implies
self.contains('Name')
```

---

RelationshipFigure constraints the GraphNode that represents the relationship figure (the characteristic diamond). The constraint is so that such GraphNode should be the container of a DiagramElement that represents the relationship name (fulfilling the Name constraint).

---

```
context GraphNode inv RelationshipCardinality:
self.simpleModel.typeInfo = 'RelationshipCardinality' and
self.coreElement.oclaIsTypeOf(Property) implies
- - semantics
self.container.coreElement.oclaAsType(Association).memberEnd
->includes(self.coreElement.oclaAsType(Property))
```

---

RelationshipCardinality constraints the GraphNode that represents a relationship cardinality (being associated to Properties). The constraint is so that the represented Property should be one of the related Association.

## References

- Abelló, A., Samos, J., Salto, F., 2006. YAM<sup>2</sup>: a multidimensional conceptual model extending UML. Inform. Syst. 31 (6), 541–567.
- Baer, T., 2008. Correctly defined concrete syntax. Software Syst. Model. 7 (4), 383–398.

- Berner, S., Glinz, M., Joos, S., 1999. A classification of stereotypes for object-oriented modeling languages. In: UML, pp. 249–264.
- Booch, G., Maksimchuk, R., Engle, M., Young, B., Conallen, J., Houston, K., 2007. Object-oriented Analysis and Design with Applications. Addison-Wesley.
- Chen, P., 1976. The entity-relationship model—toward a unified view of data. ACM Trans. Database Syst. 1 (1), 36.
- Conallen, J., 2002. Building Web Applications with UML. Addison-Wesley.
- Dobing, B., Parsons, J., 2006. How UML is used. Commun. ACM 49 (5), 109–113.
- Eichelberger, H., Schmid, K., 2009. Guidelines on the aesthetic quality of UML class diagrams. Inform. Software Technol. 51 (12), 1686–1698.
- Favre, J., 2004. Towards a basic theory to model model driven engineering. In: 3rd Workshop in Software Model Engineering, WiSME.
- Fernández-Medina, E., Trujillo, J., Villarreal, R., Piattini, M., 2004. Extending UML for designing secure data warehouses. In: ER, pp. 217–230.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J.M., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley.
- Ghoniem, M., Fekete, J., Castagliola, P., 2004. A comparison of the readability of graphs using node-link and matrix-based representations. In: InfoVis, pp. 17–24.
- Golfarelli, M., Rizzi, S., 2008. UML-based modeling for what-if analysis. In: DaWaK, pp. 1–12.
- Green, T.R.G., Petre, M., 1996. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. Vis. Lang. Comput. 7 (2), 131–174.
- Kitchenham, B., 2004. Procedures for performing systematic reviews. Keele University and National ICT Australia Ltd, pp. 1–28.
- Kong, J., Zhang, K., Dong, J., Xu, D., 2009. Specifying behavioral semantics of UML diagrams through graph transformations. J. Syst. Software 82 (2), 292–306.
- Kurz, S., Guppenberger, M., Freitag, B., 2006. A UML profile for modeling schema mappings. In: ER Workshops, pp. 53–62.
- Larman, C., 2004. Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development. Prentice Hall.
- Liskov, B., 1987. Keynote address—data abstraction and hierarchy. In: OOPSLA, pp. 17–34.
- Luján-Mora, S., Trujillo, J., Song, I.-Y., 2002a. Multidimensional modeling with UML package diagrams. In: ER, pp. 199–213.
- Luján-Mora, S., Trujillo, J., Song, I.-Y., 2002b. Extending the UML for multidimensional modeling. In: UML, pp. 290–304.
- Luján-Mora, S., Vassiliadis, P., Trujillo, J., 2004. Data mapping diagrams for data warehouse design with UML. In: ER, pp. 191–204.
- Luján-Mora, S., Trujillo, J., Song, I.-Y., 2006. A UML profile for multidimensional modeling in data warehouses. Data Knowl. Eng. 59 (3), 725–769.
- Marcos, E., Vela, B., Cavero, J.M., 2001. Extending UML for object-relational database design. In: UML, pp. 225–239.
- Object Management Group, May 2006a. Object Constraint Language (OCL), version 2.0. <http://www.omg.org/technology/documents/formal/ocl.htm>.
- Object Management Group, April 2006b. Unified Modeling Language: Diagram Interchange (DI), version 1.0. <http://www.omg.org/spec/UMLDI>.
- Object Management Group, January 2006c. Meta Object Facility (MOF), version 2.0. <http://www.omg.org/spec/MOF/2.0>.
- Object Management Group, 2007a. XML Metadata Interchange (XMI) Specification v2.1.1. <http://www.omg.org/xmi>.
- Object Management Group, February 2007b. Unified Modeling Language (UML), version 2.1.1. <http://www.omg.org/technology/documents/formal/uml.htm>.
- Object Management Group, February 2009. Unified Modeling Language (UML), version 2.2. <http://www.omg.org/technology/documents/formal/uml.htm>.
- Opdahl, A.L., Henderson-Sellers, B., 2002. Ontological evaluation of the UML using the Bunge-Wand-Weber model. Software Syst. Model. 1 (1), 43–67.
- Pardillo, J., Golfarelli, M., Rizzi, S., Trujillo, J., 2009. Visual modelling of data warehousing flows with UML profiles. In: DaWaK, pp. 36–47.
- Seidewitz, E., 2003. What models mean. IEEE Software 20 (5), 26–32.
- Staron, M., Kuzniar, L., Wohlin, C., 2006. Empirical assessment of using stereotypes to improve comprehension of UML models: a set of experiments. J. Syst. Software 79 (5), 727–742.
- Stefanov, V., List, B., 2007. A UML profile for modeling datawarehouse usage. In: ER Workshops, pp. 137–147.
- Stefanov, V., List, B., 2007. A UML profile for representing business object states in a data warehouse. In: DaWaK, pp. 209–220.
- Stefanov, V., List, B., Korherr, B., 2005. Extending UML 2 activity diagrams with business intelligence objects. In: DaWaK, pp. 53–63.
- Turski, W., Maibaum, T., 1987. The Specification of Computer Programs. Addison-Wesley.
- Wirfs-Brock, R., Wilkerson, B., Wiener, L., 1994. Responsibility-driven design: adding to your conceptual toolkit. ROAD 1 (2), 27–34.
- Zubcoff, J., Trujillo, J., 2005. Extending the UML for designing association rule mining models for data warehouses. In: DaWaK, pp. 11–21.
- Zubcoff, J., Trujillo, J., 2006. Conceptual modeling for classification mining in data warehouses. In: DaWaK, pp. 566–575.
- Zubcoff, J., Pardillo, J., Trujillo, J., 2007. Integrating clustering data mining into the multidimensional modeling of data warehouses with UML profiles. In: DaWaK, pp. 199–208.
- Zubcoff, J., Pardillo, J., Trujillo, J., 2009. A UML profile for the conceptual modelling of data mining with time series in data warehouses. Inform. Software Technol. 51 (6), 977–992.