# An Automated Process for Implementing Multilevel Domain Models

Frédéric Mallet, François Lagarde, Charles André, Sébastien Gérard, François Terrier

# An automated process for implementing multilevel domain models

Frédéric Mallet[1], François Lagarde[2], Charles André[1], Sébastien Gérard[3], and François Terrier[3]

[1] Université de Nice Sophia Antipolis,
INRIA Sophia Antipolis Méditerranée, 06902 Sophia Antipolis, France
fmallet@sophia.inria.fr, candre@sophia.inria.fr
[2] SAGEM R&D, 178 rue de Paris
91344 MASSY cedex, France
francois.lagarde@sagem.com
[3] CEA LIST,
Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
sebastien.gerard@cea.com, francois.terrier@cea.com

**Abstract.** Building a UML profile is tedious and error-prone. There is no precise methodology to guide the process. Best practices recommend gathering concepts in a technology-independent *domain view* before implementation. Still, the adequacy of the implementation should be verified. This paper proposes to transform automatically a domain model into a profile-based implementation. To reduce *accidental complexity* in the domain model and fully benefit from advanced profiling features in the generated profile, our process relies on the *multilevel paradigm*. The value of this paradigm for the definition of UML profiles is assessed and applied to a subset of the MARTE time model.

The original publication is available at www.springerlink.com

## 1 Introduction

When building a software system, the Model Driven Architecture (MDA) recommends starting by the definition of a platform-independent model (PIM), often referred to as a *domain model* or a *business model*. A model independent of the implementation technology enables not only the choice of another technology but also an easier interaction with business experts who are not necessarily familiar with the technology. However, when the domain model departs too much from the technology used, then the actual implementation becomes an issue. The correctness of the implementation with respect to the specification must be verified.

The Unified Modeling Language (UML) [1] together with its extensions like SysML [2, 3] are often chosen to build the domain model, and sometimes also the implementation itself. This is probably due to the large variety of aspects

it covers, from the specification to implementation and deployment. It offers modeling elements to specify both functional/behavioral and structural aspects.

Nevertheless, a good language for modeling domains must provide adequate primitives, expressive enough to cover all aspects of the targeted domain without altering the concepts or adding complexity. Unfortunately, like most object-oriented languages, the UML relies on the classical two-level class/object paradigm. This is sometimes [4, 5] deemed as a major impediment when the domain under consideration intrinsically contains more than two modeling levels. Examples of object-oriented languages that refuse to be confined within these two levels include prototypes [6, 7] and examplars [8]. Such languages usually also renounce strong typing and rely on mere constraints to ensure subtyping relations.

This work definitely follows a strong typing approach. Instead of evading the liability to the class/object paradigm by considering everything as an untyped object (as prototype-based approaches do), it rather considers the *clabject*-based approach proposed by the multilevel modeling community. Two implementations of the multilevel paradigm already exist. One for the programming language Java [9] and one in the metamodeling community Nivel [10] that comes with a formal semantics. Discussions about the possibility to extend the UML metamodel to tackle multilevel features are on-going but no lightweight extension has been considered until now, to the best of our knowledge. Therefore, we propose to build a UML profile for multilevel modeling and we investigate how such a profile could be used to bridge the gap between the domain model and its actual implementation. More specifically, we propose an automated process to generate an implementation from the domain model annotated with the multilevel information.

The proposed automated process is illustrated on a subset of the UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [11], recently adopted by the Object Management Group (OMG). The discussion focuses on MARTE time model that aims at extending the mainly untimed UML with precise and advanced time modeling features. This part of MARTE intrinsically covers several modeling levels. The lack of multilevel mechanisms in the UML has prevented the capture of such information within MARTE domain view, resulting in the use some unusual workarounds that makes it difficult to establish that the implementation correctly represents the domain. The same workarounds have been applied in several parts of MARTE and are worth to be considered more systematically in forthcoming UML profiles. Such a systematic usage can be straightforward provided that the multiple levels underlying the domain are explicitly identified. Making explicit all the levels reduces the *accidental complexity* and put into light the design choices.

More generally, we plead for the systematic use of automatic transformations from the domain model to the implementation. This is traditionally the case in the community of the domain-specific modeling (DSM) and unfortunately seldom the case in the profiling community. Apart from allowing the reuse of the domain model for other targeted implementation, the use of automatic transformations also facilitates the comparison of different approaches. Comparison

would require the definition of metrics (not addressed here) and metrics are difficult to apply consistently in manual processes. Furthermore, an automatic process makes explicit designer choices, which is a necessary base for argumentation and for establishing a strong connection between the domain model and the implementation. Finally, it opens the path to advanced transformation techniques like model weaving, which is critical when so many profiles offer such a high level of redundancy or contradiction. The profiling community would really benefit from advanced composition mechanisms. This contribution is a small step to address such an ambitious plan.

Section 2 recalls the basic mechanisms involved when using the multilevel modeling paradigm. Section 3 introduces a subset of the MARTE time model and emphasizes on the aspects related to multilevel modeling. A new partial domain model with deep characterization is then proposed. Section 4 proposes a new UML profile for multilevel modeling. This profile is then applied to generate a new implementation of the MARTE time model. The generated profile is compared to the one adopted by the OMG.

## 2 Multilevel modeling

Most[4] object-oriented languages abstract away common properties for a given set of objects inside a common structure: a class. For instance, when developing a system for a firm that manufactures personal digital assistants (PDA), a class PDA is created. This class gathers all the properties of PDAs relevant for the application to be designed. For instance, PDA properties are its brand and the screen size (see Figure 1a). The dependencies (dashed arrows) between classes and instances stand for an instantiation relationship. Since this is the only usage of dependency in this figure, the annotation «instanceOf» is omitted. The question marks are not part of the UML specification, they only indicate weaknesses of the proposed solution or shows what the intent of the designer is. The properties next to the question mark is either not conformant or not satisfactory.

In this kind of example and almost always when building a domain model for products, higher level properties related and specific to the brand are required. For instance, some PDA, *e.g.,* those from HTC, may have a kind (Touch, Shift, and Cruise). Some others, *e.g.,* those from ASUS, may have another classification or none at all. Anyway, this information of type HTCkind is not relevant for PDA built by Asus. With such a simple model, if an attribute kind is added to class PDA, this attribute cannot have a valid value for PDAs whose brand is Asus.

An alternative solution is to make different classifications for different brands using inheritance (see Fig. 1 b & c). Using inheritance in such cases is often not satisfactory. All PDAs have a brand; the first solution is then to add an attribute to abstract class PDA (see Fig. 1b). Such a solution requires additional constraints (see curly brackets) to ensure that brand "HTC" is used consistently in all instances of class PDA HTC. Moreover, the memory is not very well used

[4] Some object-oriented languages do not rely on classes but rely on examplars or prototypes

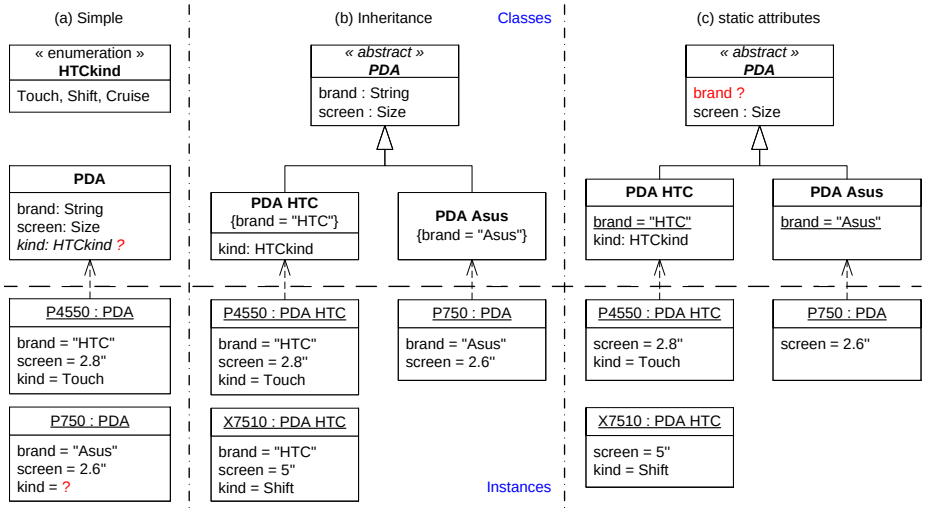


**Fig. 1.** Domain model for PDA

since each object has its own slot for the brand but has no freedom for the value. It is easy to avoid such a redundancy by making static attribute brand as in Figure 1c. However, there is no typing-based way to ensure that all PDAs have a static attribute brand. Another constraint (not shown on the figure) should be added on abstract class PDA. A third solution would be to rely on dynamic mechanisms to identify the brand (*e.g.,* based on the name of the object type). Dynamic programming offers higher flexibility but is also subject to higher risks of dynamic errors more difficult to identify and fix. Finally, programming languages usually offer some workarounds but our intent is to consider modeling languages rather than programming languages.

The use of *PowerTypes* provides a solution for such scenarios where product information comes with additional information about the type (or model) of the product. Figure 2d applies this pattern to the same example[5]. The inheritance relationship is replaced by an association that links a PDA to its brand. Abstract class PDA is further specialized in subclasses specific to a given manufacturer (*e.g.,* PDA HTC). Using powertypes the model becomes complete and flexible. However, it is quite complex and the relevant information about the PDA is scattered in several classes and instances. In particular, the information about PDAs manufactured by HTC is spread over class PDA HTC and instance HTC of type Manufacturer (as depicted by the circled area with a dark background). The same physical object is distributed into several elements, each of which belongs to a different modeling level.

[5] This is not the official UML notation for powertypes, we rather used the notation proposed by Henderson [12] because it better shows what actually happens

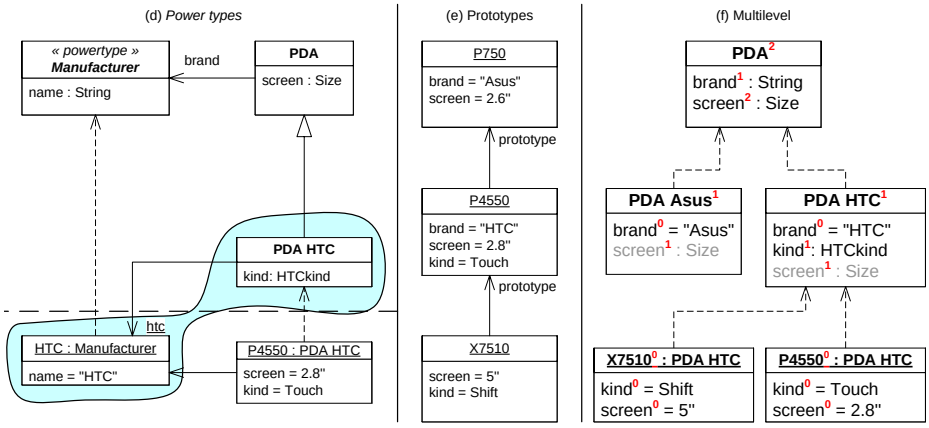


**Fig. 2.** Powertypes, Prototypes or Multilevel

When using prototypes [6] the model is flattened and only instances without any typing information are considered. Prototypes are actual objects that are well known and on which all other objects of the system are based. Only differences with already existing objects are put in newly created objects. This opposes to class-based approaches where properties of classes of objects are abstracted away. Prototype-based approaches deal with concrete objects instead of dealing with abstract concepts. Figure 2e shows the same system modeled with prototypes. The result is very simple and has little redundancy. However, the result may highly differ depending on the order in which objects are built and depending on the choice of prototypes. More importantly there is no typing information at all, and constraints are used to enforce subtyping relationships. Prototypes work by selecting a prototype (*e.g.,* `P750`) and building other objects by difference/addition. For instance, `P750` is a prototype for `P4550` because the latter also owns the two properties `brand` and `screen` defined in the former; the values for these properties are different though. It also has an additional property: `kind`. Similarly, `P4550` is a prototype for `X7510`. `X7510` keeps the property `brand` unchanged and modifies the values of properties `screen` and `kind`.

Multilevel modeling also flattens the levels but instead of flattening it down to mere instances, it relies on *clabjects* [4] that merge features of classes and objects. A clabject represents all the aspects of a concept, whatever the level at which it belongs. Whereas with classical approaches all properties of a given class belong to the same modeling level, clabjects have *fields*, which unify meta-attributes, attributes and instance slots. Fields are assigned *potency*, an integer representing the number of instantiations required to get an actual value. After each instantiation, the field potency is decremented. The field persists over instantiations as long as its potency does not reach the value zero; it then becomes a slot and gets a value. Having potency enables *deep instantiation* where fields can survive several instantiations whereas with classical instantiation, only one

instantiation is possible and every attribute gets a value during the instantiation. A field with a potency of 1 is equivalent to an attribute. A field with a potency of 2 is equivalent to a meta-attribute. All the fields of a given clabject do not necessarily have the same potency. Figure 2f applies the multilevel modeling approach to our example. *Clabject* PDA has a field brand, which ensures that every sub-class has such a feature and since its potency is one, the value must be given after the first instantiation and will not be replicated further, thus replacing the static attributes of Figure 1c. There is also a field screen with a potency of two. All PDAs must have some information about the size of the screen but it is not common to all models of a given manufacturer. At the second instantiation level, manufacturer specifics appear. Field brand gets a definitive value; the potency of field screen is decremented. For all PDAs from HTC, a new field (kind) is introduced and has a default potency of one. At the lowest level, all fields have a potency of zero and have a value, thus making the clabject equivalent to an object. Overall, the resulting model is flexible, compact and faithful. The mechanism could be even more powerful if potencies were variables instead of constant values, thus preventing premature choices for the number of instantiations required. For instance, had the potency of the field screen been a variable (denoted $\star$), the designer would have known that a description of the screen had to be given at some point without enforcing it to be given precisely after two instantiations.

In the following, we rely on this multilevel modeling to build a rich domain model of the MARTE time model.

## 3 MARTE time model

Having given a fairly neutral example to recall the benefits of using deep characterization and multilevel modeling we now look at the MARTE time model and focus on the multilevel aspects underlying this specific domain.

### 3.1 The OMG profile.

Figure 3 shows a simplified view of the MARTE time model as specified by the OMG. This profile has been introduced in detail earlier [13] and the point is not to go through the specifics but rather to use it as a support to illustrate our proposition.

The figure divides into three parts. The left bottom part is a simplified view of the MARTE time model. The right-hand side part shows some libraries defined in the profile. The upper part contains a partial description of the UML metamodel (UML::Classes::Kernel) on which the profile is based. Annotations (A) and (B) together with dependencies (A) and (C) are not part of the adopted specifications but are introduced to conduct the discussion.

The time profile introduces two main stereotypes: Clock to create new clocks and ClockType to gather in a same structure common features of a given set of clocks. The time model and more generally the MARTE profile applies unusual

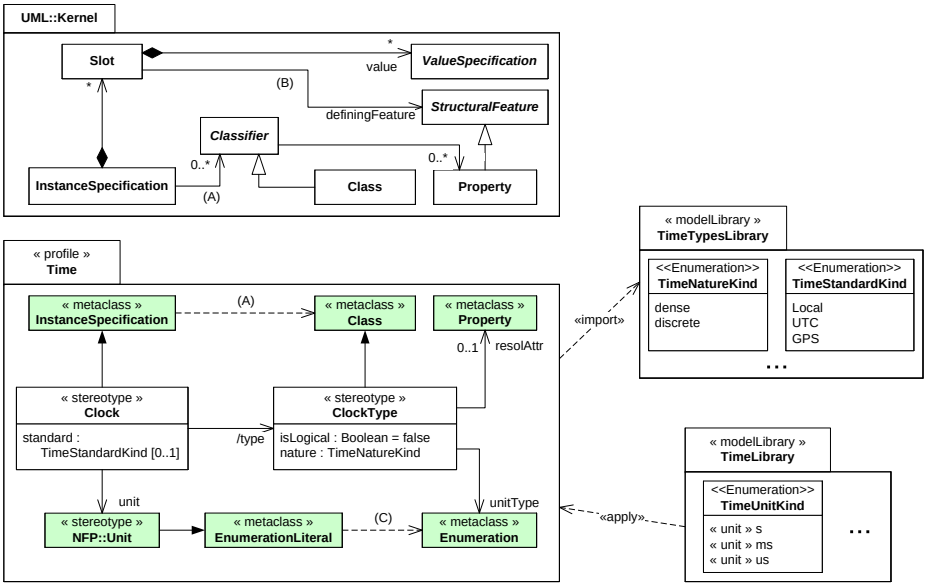


**Fig. 3.** Excerpt of the Marte time profile.

design patterns. Indeed, in most profiles, stereotype attribute types are restricted to primitive types and enumerations, whereas in MARTE, some stereotypes are associated with metaclasses or even other stereotypes[6]. This design choice has manifestations in stereotype ClockType and its attribute resolAttr of type Property, a UML metaclass. This meta-attribute references a property that acts as a resolution. Using a metaclass here rather than a primitive type like Integer avoids a premature choice of the type. Instead of an integer, the resolution could just as well be a rational number or an enumeration (*e.g.,* fine, medium, coarse). Delaying the choice of a property type can also be achieved with templates. Another requirement is to be able to annotate existing models from various communities and on which we have no control on the terminology. For instance, in an existing model the resolution may be called granularity or quantum. Not having a more elaborate ontological way of associating different words for identical concepts, the proposed design pattern achieves the same goal. The meta-attribute resolAttr points at the actual property that is deemed to act as a clock resolution, whatever its type and name. This design pattern has been applied several times in MARTE. Its usage and the possible automatic usages that can be expected from tools have been previously illustrated [14] on an example where model transformations for real-time operating systems are performed.

[6] OMG profiles have traditionally used only mechanisms simpler than what is actually allowed. This is partly due to the poor support offered by profiling tools. In the last two years, there have been good efforts to improve the tools thus allowing more advanced features.

Another case we use to justify our approach concerns attribute unit whose type is NFP::Unit, one stereotype defined in MARTE. A more classical choice would have been to choose a mere String. An alternative is to select metaclass EnumerationLiteral, which is the base metaclass of stereotype NFP::Unit. However, choosing the stereotype rather than the metaclass leads to a more precise characterization of what unit is. Not any enumeration literal can be used as a time unit; it has to be stereotyped by NFP::Unit thus giving additional information like conversion factor with respect to other units. The same effect could have been achieved by using the metaclass EnumerationLiteral as a type and specifying an OCL constraint. This alternative solution is often preferred because of the failure of most commercial tools to support stereotype-to-stereotype associations. We did prefer a strong-typing approach over a constraint-based solution. Our choice is even more justified since there are also very few tools that check the conformity of UML models against the OCL constraints. The essential choice criteria remain to reduce the accidental complexity to get simple and easy to understand domain models.

### 3.2 Multilevel aspects in the Time Profile

The major difficulty understanding Figure 3 comes from the use of the design pattern *Type/Object* [15], also known as *Item/Descriptor* [16] or even *PowerType* [17]. By emphasizing on these difficulties in this subsection, we want to show how difficult and hazardous the definition of a profile can be. The value of our proposition comes from the fact that such a profile can be generated automatically when the inherent multiple levels have been made explicit.

Indeed, it is common to think that metaclasses (*e.g.,* InstanceSpecification, Class, Enumeration, EnumerationLiteral) identify *meta* elements whereas classes and primitive types denote *model* elements. A careful look at the existing relationships in the UML metamodel is, however, required to accurately qualify the modeling levels involved. The two relationships (A) and (B) in Figure 3, in the UML metamodel, actually denotes a typing relation. Consequently, Class and Enumeration are higher-level concepts than InstanceSpecification and EnumerationLiteral. The relationship (A) tying an InstanceSpecification to one of its Classifier shows that clocks are instances whereas clock types are types. This is a classical usage of the pattern Type/Object. This relationship is made explicit in the profile by the derived attribute type of the stereotype Clock to highlight the designer intent, even though it is not strictly required since the relationship already exist in the metamodel. type is a subset derived attribute of attribute classifier from the metaclass InstanceSpecification.

The same pattern is applied in relationship (B) that indirectly links the metaclass Slot to the type Property (a subclass of StructuralFeature). Attribute resolAttr of stereotype ClockType highly depends on this relationship.

Dependency (C) between EnumerationLiteral and Enumeration also relies on the same pattern even though the relationship is not explicit in the UML metamodel. The dependency (dashed arrow) makes it explicit in Figure 3 to justify the intended relationship between unit and unitType. A clock unit must be chosen in

a set of enumeration literals owned by the enumeration identified as the unit type of the clock type.

Note also that a model library is a UML construct that purposely escapes modeling levels. The same model library can actually be used at different levels: the metamodel level (like MOF), within profiles or within models. Their usage is quite simple when they are limited to primitive types or enumerations. This is the case for primitive types defined in the UML standard library or in types defined in MARTE library TimeTypesLibrary. It becomes a bit tricky when it comes to types specifically devised for being used at a specific level. This the case with model library TimeLibrary of MARTE. Its types can only be used at the model level for MARTE end-users and can certainly not been used in the MARTE profile itself since it applies it. In Section 4, potencies make explicit the level at which each element of a model library should be used.

### 3.3 Applying the Time Profile

Figure 4 illustrates the use of the Time profile for two clock types. The left-hand side of Figure 4 shows a conventional usage of the MARTE profile. The clock type Chronometric is defined in the model library TimeLibrary. It models dense clocks, related to physical time, which are not necessarily perfect (skew, jitter). The property resolution, whose type is Real, is selected to play the role of resolAttr. The clock type Cycle represents a discrete logical clock that uses units like processorCycle or busCycle to date event occurrences. For Cycle, there is no need for a property playing the role of resolAttr.

The chronometric clock cc1 completes the specification by selecting one specific unit (s) out of the literals defined in the enumeration TimeUnitKind. It also chooses a standard (*e.g.,* UTC) and a value for the resolution. The cycle clock p1 also selects a unit, but from a different enumeration, CycleUnitKind. Clocks of the same type must use compatible units (from the same enumeration). In that regard, the clock types acts as a dimension.

The right-hand side of the figure is a conceptual representation that emphasizes the different modeling levels. The notation is inspired from clabjects as defined by Atkinson and Kühne [4] but the potency is not explicit. Instead, horizontal dashed lines serve to identify the logical modeling levels. Some model elements (*e.g.,* ChronometricClock and CycleClock have two compartments: one for attributes (fields) of potency 0 that carries a value, and one for fields of potency 1. The major difference with the use of static attributes is that here the typing relation guarantees that fields such as nature, isLogical and unitType are there for any clock type.

In the UML view, Clock and ClockType are both represented at the same level, as stereotypes. However, ClockType is a *descriptor* for a set of Clock (as defined by the pattern Item Descriptor). They therefore belong to a different modeling level. In the domain view, the three levels are clearly separated by the horizontal dashed lines.

This strategy imposed by the use of the pattern Type/Object leads to a model where the information about clocks is scattered. Part of the information

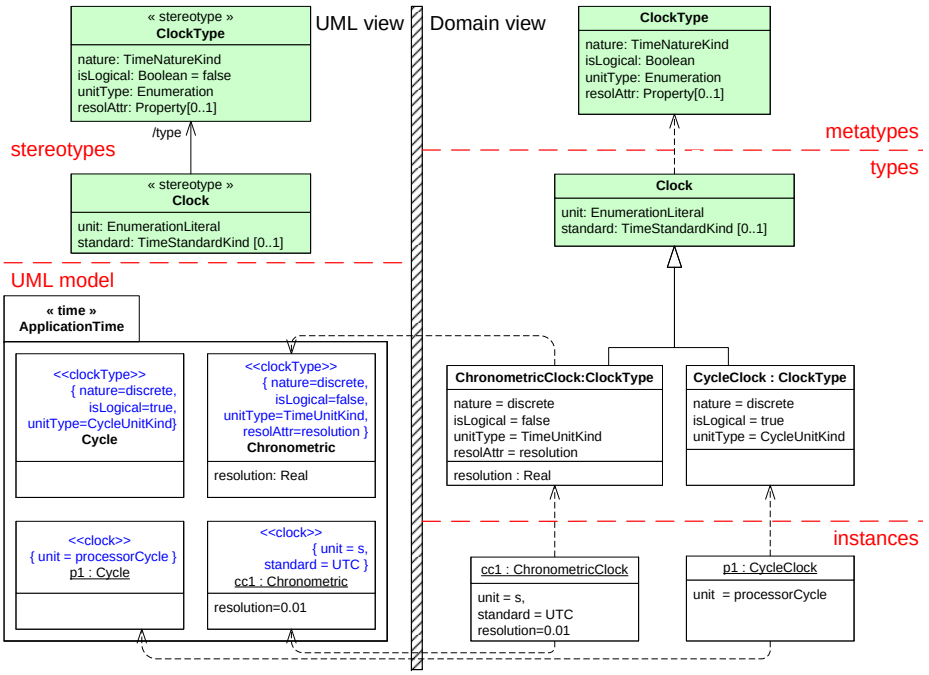


**Fig. 4.** Examples of clocks and clock types

is available in the class used as clock type, another part is in the meta-attribute of the stereotypes («ClockType», «Clock») and the remainder is given by the slots of clock instances. The information is progressively built and refined in the successive steps: class definition, instantiation and stereotype applications. Two examples of such scattering concern the clock *resolution* and *unit*. However, there are also examples where the information is located at only one level: the class level for the *nature* or the instance level for the *standard*.

The complexity of such a mechanism is mainly due to the restriction to only two modeling levels in most of the UML. This *accidental complexity* [18] could have been reduced by using the multilevel modeling paradigm. Such an alternative solution is discussed in the following section.

## 4 A profile for multilevel modeling

### 4.1 Principles

This section presents the mechanisms that have been devised for the creation of UML-based domain specific languages that support the multilevel modeling paradigm. Its usage is illustrated on the MARTE time profile, even though the proposal is general and could be used to generate a profile for other domain views.

Our proposal is based on a three-step process. The first step is to specify the domain model. Defining such a domain model with the UML requires the use of a simple specific profile for domain specification. This profile introduces a stereotype Potency. The potency is used, in a second step, to derive automatically a UML profile from the specification. Our premise is that using an automated transformation reduces the gap between the domain and the profile and ensures that every concept in the domain is actually implemented in the profile. Application of this profile, in a third step, enables modeling of elements that comply with the domain model specification. In subsequent sub-sections follows a step-by-step illustration based on the Time profile.

### 4.2 Domain model specification

The lack of efficient multilevel modeling mechanisms in the UML has made the implementation of MARTE domain view much more complex than it should have been. Focusing on the clock mechanism, the MARTE domain view describes clocks with a single class (pretty similar to the central class in Figure 5). However, the profile itself (Fig. 3) defines two stereotypes («ClockType», «Clock») for this concept. With explicit multilevel modeling constructs, going from the domain view to the implementation becomes straightforward. Figure 5 shows how to build a multilevel model for MARTE clocks.

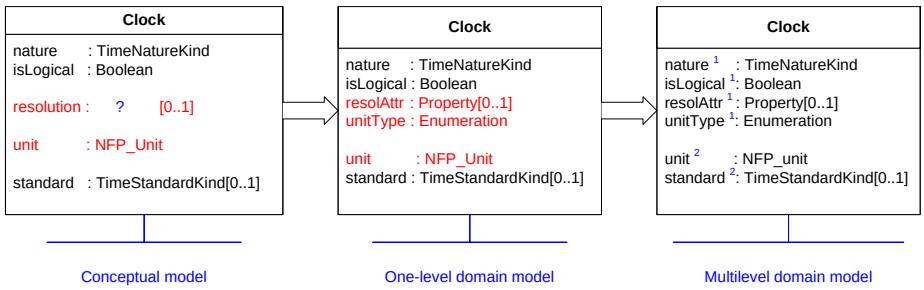


**Fig. 5.** A multilevel model for MARTE clocks.

MARTE domain view identifies the essential properties of clocks (see Fig. 5, left part). This representation is only partial in regard with the actual OMG model. To simplify the discussion only a subset of the properties is used, but this subset contains all the different cases that we want to discuss here.

As explained in subsection 3.1, it is rather complex to choose the right type for the property resolution. Additionally, modeling the unit with a mere literal is not completely satisfactory to ensure static compatibilities amongst clocks of the same domain. The model is refined as shown in the central part of Figure 5. Attribute resolution is typed by Property. The unit is split into two different concepts; the unit itself (unit) and its type unitType, which defines the set of authorized and compatible units. unitType is typed by Enumeration because of the

semantic relationship between EnumerationLiteral (and therefore NFP_unit) and Enumeration.

Finally, (Fig. 5, right part), levels are made explicit by adding a potency to the fields. The domain view in Figure 4 shows that there are actually three levels (*instance*=0, *type*=1, *metatype*=2). This justifies a potency of 2 for fields unit and standard, the value of which is given at the *instance* level. nature, isLogical and unitType get their values at the *type* level and are then assigned a potency of 1. The same holds for resolAttr, which also gets its value at the *type* level. Note that, its value is not the resolution itself (given at the *instance* level) but the property used to model the resolution (resolution:Real).

The final result (Fig. 5, right part) is very concise while still having the same expressiveness than the initial profile (Fig. 3). However, to implement this model domain we need an environment that supports multilevel modeling. For that purpose, we have defined a UML profile (called DomainSpecification) for domain specification (see Figure 6, left part). This profile allows the use of the UML environment to capture the domain view with the multiple levels.

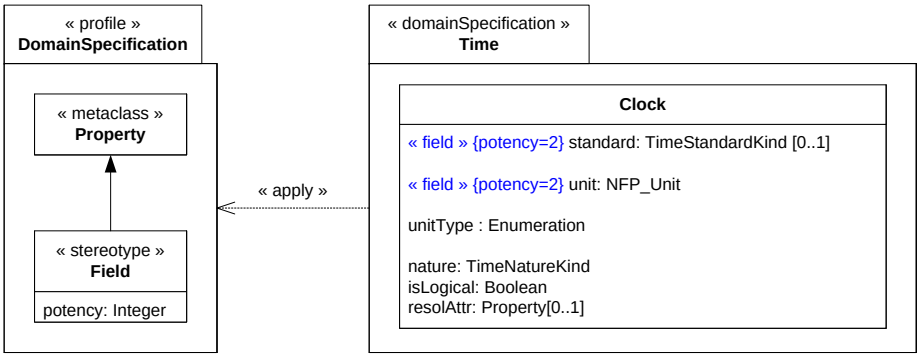


**Fig. 6.** Domain specification profile and its usage

It is used to annotate a UML model with information required for multilevel modeling. This enables declaring models in a way similar to the one presented in Figure 5. This profile consists of one stereotype (Field) that extends the metaclass Property and carries the potency information. Possible extensions to this profile are discussed in section 5. Potency is optional. Properties that are not stereotyped are considered as *regular* attributes, *i.e.,* having a potency of 1. The right part of Figure 6 is then the UML implementation of the domain view shown in the right-most part of Figure 5.

### 4.3 Automatic generation of a UML profile

In this second step, we use the domain specification as an artifact to build a profile-based implementation. The result is a profile with enough stereotypes to

describe each instantiation level for each concept. Consequently, we automatically derive the profile shown in Figure 7.

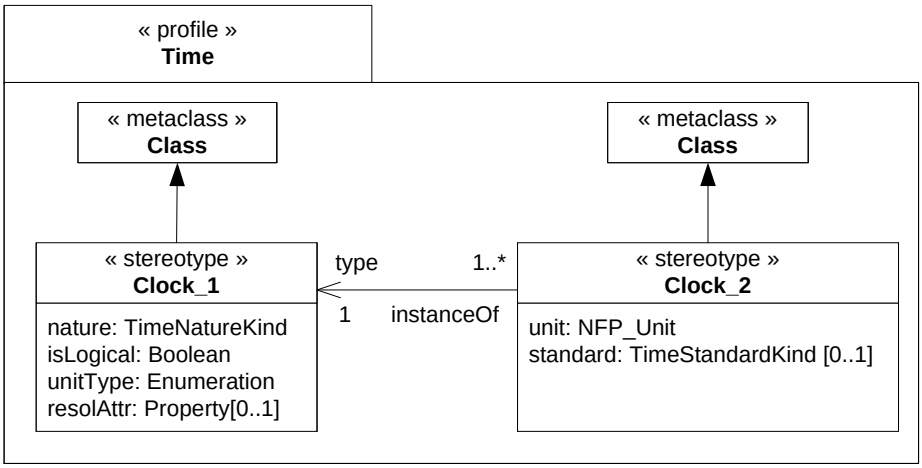


**Fig. 7.** Time profile generated from the domain specification

A straightforward algorithm is used for the transformation. Each class gives rise to as many stereotypes as instantiation levels. The name of each generated stereotype is derived from the name of the initial *clabject* suffixed by an integer that reveals the level of instantiation. In our example, there are two levels (potency=2 and potency=1), so we derive two stereotypes (Clock_2, Clock_1). Each stereotype systematically extends the metaclass Class. Clock_2 contains the fields with a potency of 2 and Clock_1 the fields with a potency of 1.

```
foreach clabject c do
  if c.potency = 1 then
    generate a stereotype s whose name is c.name
    each field of clabject c becomes a meta-attribute of s
  else
    for i from 1 to c.potency do
      generate a stereotype s whose name is (c.name+'_'+i)
      s extends metaclass Class
      foreach field f of c do
        if f.potency = 1 then
          add a meta-attribute to s same name and type as f
        else
          decrement f.potency
        end if
      end foreach
    end for
  end if
end foreach
```

Clock_1 represents the first instantiation level and is the equivalent of the former stereotype ClockType. Clock_2 represents the second instantiation level, *i.e.,* the former stereotype Clock. An association between the two stereotypes is added to maintain the relationship type/instance between the model elements that will eventually represent a given clock.

### 4.4 Applying the generated profile on the user model

In this section, we apply the generated profile to declare the chronometric and cycle clocks (Figure 8) and we compare to the solution with the actual MARTE profile (Figure 4). Modeling of Cycle clock entails two new classes; CycleClock stereotyped by «clock_2» and cycleClk stereotyped by «clock_1». The property type avoids mixing Cycle clocks with Chronometric clocks. For instance, cycleClk is associated with CycleClock, not ChronometricClock. This link is required to know that cycleClk is a discrete logical clock, whose unit must be selected within the literals of CycleUnitKind.

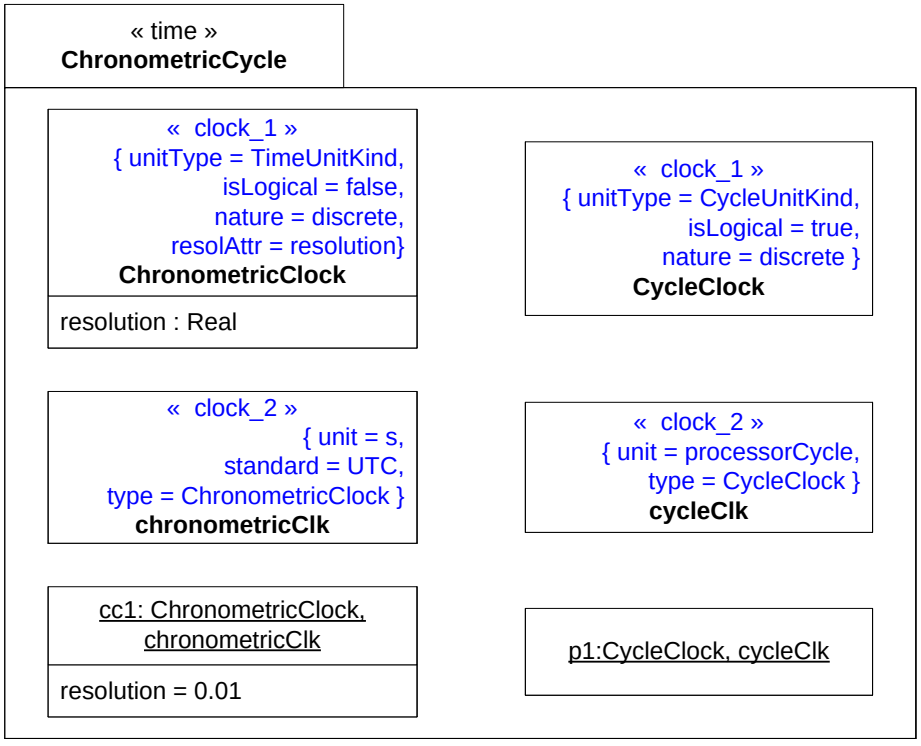


**Fig. 8.** Clock definition with the generated profile.

The structure of this model is, at first glance, similar to the model using the original MARTE constructs (Figure 4). One obvious difference, however, appears in the metaclasses used as bases for our stereotypes. In MARTE, clocks were instance specifications of classes ChronometricClock and CycleClock, themselves stereotyped by «clockType». The instance specifications carry information about the slots of this class and also provide information like values relating to properties defined by stereotype Clock. With the generated profile, since both Clock_2 and Clock_1 extend Class, an instance of a clock (*e.g.*, p1) must have two classifiers to gather within a single object all the values given at each level. Each classifier has properties related to one level. These differences are discussed thoroughly in the next section.

## 5 Discussions on our approach

This section compares our proposed approach with the one followed by the MARTE designers and by profilers in general. It describes the process workflow differences, and then discusses possible extensions to our approach.

### 5.1 Design flow comparison

Figure 9 shows comparison of the two process workflows, from conceptual domain definition to profile creation.

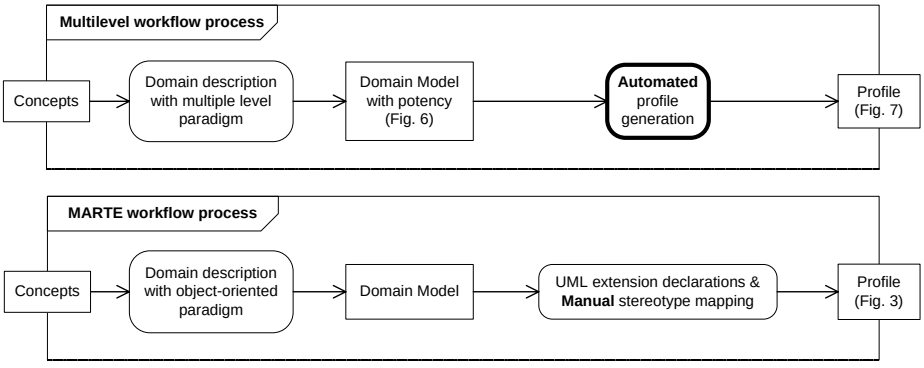


**Fig. 9.** Design activity flow comparison

The two approaches are obviously very similar. They rely on two-stage processes. The first stage is the description of the domain view. Our proposal requires making explicit the modeling levels. It mainly consists in identifying and applying design patterns. In our process, we recommend to apply the potencies when the pattern Type/Object is identified.

The second stage is essential and is a very sensitive activity. It consists in mapping the domain concepts onto similar UML concepts. Different designers may use different design solutions to map a given concept. This makes it difficult to compare two implementations. Our proposal is to make this choice systematic and explicit within a transformation tool instead of relying on a manual process. An automatic process allows traceability and reduces the gap between the domain description and the profile. It also makes the process more reliable and ensures that each concept is translated once and that nothing is added or skipped by mistake. Maintenance of models is made easier since the implementation is generated automatically and only the domain model must be maintained. Consistency is afforded between domain model and profile.

### 5.2 Possible extensions

Currently the profile DomainSpecification is minimal. A more complete profile has also been proposed [19]. These options are not described here in full for the sake of clarity. Our purpose is more to advocate for the use of automatic transformations rather than claiming that our specific transformation is the best solution. Nevertheless, new stereotypes can be introduced to guide the profile generation. Remember that, one of the original reasons for building a profile was to be able to customize a model according to a particular point of interest. Compliant tools should (even if it is mostly not the case now) provide an easy support to hide or restore annotations of a given profile. Following that idea, our profile could be customized differently to build a profile fitting the design team legacy. Each team can implement its own generation rules. Explicit profile generation process and tool support allow designers to assess different candidate generation processes.

As possible customizations, we can allow a systematic naming convention and the selection of base metaclasses more adequate than the by-default Class. This kind of customization would allow us to generate the exact same result as in the actual MARTE specification where several metaclasses other than Class have been used (InstanceSpecification, Event, Observation, Activity...). Figure 10 illustrates this extension and one of its possible usages.

Stereotype Clabject identifies the actual clabjects, whereas previously all classes where considered as clabjects. The clabject explicitly states the naming convention for the extended concept depending on the modeling level under which it is considered. The particular usage proposed in the figure makes explicit the concepts of *ClockType* and *Clock* as two particular aspects of a clock.

As illustrated in this paper, using generic metaclasses like Property and Class often leads to very flexible solutions. However, this may also lead to solutions with little semantics. It is often preferable to choose types and metaclasses that better represent domain concepts. For instance, we could also have defined the unit as being a Property instead of an enumeration literal. In that case, we wanted to be compatible with NFP_unit defined in the MARTE NFP subprofile. Being more general than an enumeration literal would have prevented us from using NFP_unit-specifics, like the conversion factor. Having a conversion factor and

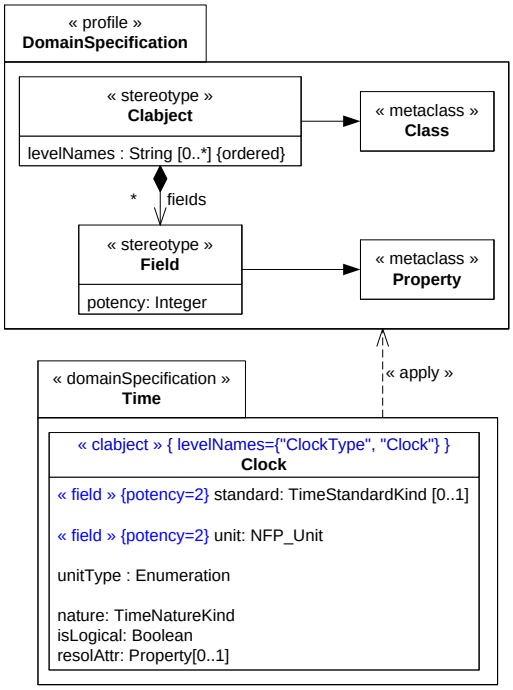


**Fig. 10.** Extended domain specification profile and its usage

relations amongst units is very useful and is absolutely required to perform dimensional analysis.

### 5.3 Related work

More than just providing a mere implementation of multilevel paradigm in the UML, the core motivation of this work is to automate the building of UML implementations of domain-specific languages (DSL). The expected outcome is to reduce design costs by reusing existing, almost mature, UML graphical editors. Reducing the number of tools is even more important now, since the number of available trained engineers is far from sufficient to deal with the demand.

In that matter, two communities confront each other, the meta-modeling and the profiling communities. Light-weight solutions often involve the creation of a profile, whereas more complete solutions require the use of meta-modeling. This work is only concerned with the building of profiles. However, use of the multilevel paradigm simplifies the domain model. This multilevel-aware domain model could also be used by meta-modeling tools to produce a more faithful and simple code.

In this paper, and in multilevel modeling in general, we focus on the relationship "instanceOf". Other approaches, related to meta-modeling, consider

relationships in general, "instanceOf" being just one of them. Other relationships include association, dependencies, conformance, composition.... However, the relationship "instanceOf" must be different somehow since it has specifically inspired lots of work. It seems obvious that this relationship plays a predominant role in the design activity, even more in object-oriented or component-oriented approaches. Section 2 thoroughly discussed various approaches that specifically focus on this relationship.

In profile-based approaches, despite the ever increasing number of profiles being built in many domains, there is little published literature available to support the process as a whole. There is no recognized metrics to measure the adequacy of the profile, even though this is probably essential for the credibility of profiling approaches. However, some progresses have been made already. Fuentes and Vallecillo [20] point to the need for first defining a domain model (using UML itself as the language) to delineate clearly the domain of the problem. In a more recent paper [21], Bran Selic describes a staged development of UML profiles and gives useful guidelines for mapping domain constructs to UML.

*Powertype*s dismiss the problem by using an association to implement the typing relationship. Other works have already described similarities between *powertypes* and profiling mechanisms [12]. Section 2 gives a thorough comparison between multi-level modeling and the use of powertypes.

Our proposal also leverages the use of a domain model but explores multilevel modeling capabilities at this stage. Almost all the material available on multilevel modeling can be found in research efforts conducted by Kühne and Atkinson. They have studied the foundations of such modeling and proposed an implementation [4] by a heavy integration as primary UML concepts. This work published in 2001 long before the release of UML 2 in 2005 should be adapted to UML 2 and at least, should take into account the PowerType Mechanism of UML 2. Our approach is different, since we do not recommend altering the UML metamodel but we rather propose a lightweight extension based on a profiling approach. More recently, Kühne and Schreiber[9] explored possibilities to support deep instantiation in Java. The multilevel paradigm has also been used with Nivel [10], which has been given a formal semantics. Our proposition concerns the design phase, before even the choice of an implementation language like Java.

The context of our proposal is somewhat different. We assess values of deep instantiation mechanisms in the context of UML profile definitions, and then demonstrate that the current UML specification already includes mechanisms for accessing the realm of multilevel modeling.

## 6 Conclusion

This paper presents an automatic process for generating a UML profile from a domain model by leveraging the use of the multilevel modeling paradigm. A profile for multilevel domain specification is introduced for this purpose. This profile extends UML with the concepts of *field* and *potency*.

The process begins with the specification of concepts required to cover a specific domain. The domain specification is then used to map elements onto equivalent profile constructs. The result is a profile-based implementation of the domain model that contains all stereotypes required to represent the concepts of the domain and their different instantiation levels. Application of this profile thus enables deep instantiation and modeling of elements complying with the domain model specification.

The proposition is illustrated with an excerpt from the Time sub-profile part of the MARTE profile, recently adopted by the OMG in June 2009. Use of the multilevel modeling paradigm provides new design opportunities and enables simplifications. It facilitates the domain specification by limiting implementation considerations. The domain description is more concise and clarifies the modeling levels. The resulting domain model gathers in a single class all the information related to a given concept, whereas it was previously scattered over several classes.

We also consider that this discussion highlights MARTE designers' intentions. With such a support, the discussion around MARTE could focus on the domain view without requiring more comments about the profile itself. It would also make the specification shorter, and therefore more accessible, since the full description of the UML view would not be required and would be generated by a transformation tool. Automatic transformations may also reconcile the profiling and the meta-modeling communities. The main research effort of both communities would be to improve and define new meta-modeling mechanisms. The implementation as a UML profile being just a solution amongst others.

Our proposition entails several model transformations. They are being automated in an Eclipse environment as a plug-in of the open source Papyrus[7] UML tool. This tooling support will enable generation of profiles that support domain elements and include the necessary OCL rule enforcements. Assessment of the user's model should be automated.

We advocate for a well-defined process that could consistently be used to define coherent profiles. This process must rely, as much as possible, on automatic transformations. The use of the multilevel paradigm is not specific to profiling and should also be used more frequently in meta-modeling approaches. Such a paradigm deserves to be further explored and its insertion with more general approaches still needs to be assessed.

The use of an automatic process would allow the use of metrics to compare different implementation processes. The implementation process is indeed very difficult to assess on an example if the result varies depending on the designer that applies the process.

## References

1. OMG: Unified Modeling Language, superstructure. OMG document formal/2007-02-03, Object Management Group (February 2007)

[7] http://www.papyrusuml.org/

2. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. The MK/OMG Press, Burlington, MA, USA. (2008)
3. INCOSE: Systems Modeling Language (SysML) Specification 1.1. Object Management Group. (May 2008) OMG document number: ptc/08-05-17.
4. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. UML - The Unified Modeling Language. Modeling Languages, Concepts, and Tools **2185** (October 2001) 19–33
5. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and System Modeling **7**(3) (2008) 345–359
6. Borning, A.: Classes versus prototypes in object-oriented languages. In: Proc. of the Fall Joint Computer Conference, IEEE Computer Society (November 1986) 36–40
7. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: OOPLSA'86: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, ACM (1986) 214–223
8. LaLonde, W.R., Thomas, D.A., Pugh, J.R.: An exemplar based smalltalk. In: OOPSLA. (1986) 322–330
9. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In: OOPSLA'07: Proc. of the $22^{nd}$ annual ACM SIGPLAN conf. on Object oriented programming systems and applications, ACM (2007) 229–244
10. Asikainen, T., Männistö, T.: Nivel: a metamodelling language with a formal semantics. Software and System Modeling **8**(4) (September 2009) 521–549
11. OMG: UML Profile for MARTE, v1.0. Object Management Group. (November 2009) Document number: formal/2009-11-02.
12. Henderson-Sellers, B., Gonzalez-Perez, C.: Connecting powertypes and stereotypes. Journal of Object Technology **4**(7) (2005) 83–96
13. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: Proc., 10th Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'07). Volume 4735 of LNCS., Springer (2007) 559–573
14. Thomas, F., Delatour, J., Terrier, F., Gérard, S.: Towards a framework for explicit platform-based transformations. In: 11th IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'08), IEEE Computer Society (May 2008) 211–218
15. Johnson, R., Woolf, B. In: Type Object. Volume 3. ADDISON-WESLEY (October 1997) 47–65
16. Coad, P.: Object-oriented patterns. Communications of the ACM **35**(9) (1992) 152–159
17. Odell, J.: 3. In: Power Types. Volume 12 of SIGS Reference Library. Cambridge University Press (1998)
18. Brooks, F.P.: No silver bullet essence and accidents of software engineering. Computer **20**(4) (1987) 10–19
19. Lagarde, F., Mallet, F., André, C., Gérard, S., Terrier, F.: Multilevel modeling paradigm in profile definition. Research Report RR-6525, INRIA (April 2008)
20. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An Introduction to UML Profiles. UML and Model Engineering **V**(2) (April 2004)
21. Selic, B.: A systematic approach to domain-specific language design using uml. In: $10^{\text{th}}$ IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC'07), IEEE Computer Society (2007) 2–9