A Systematic Approach to Domain-Specific Language Design Using UML

Bran Selic
IBM Canada
bselic@ca.ibm.com

Abstract

UML includes special extensibility mechanisms, which are used to define domain-specific modeling languages that are based on UML. These mechanisms have been significantly improved in the latest versions of UML. Unfortunately, there is currently a dearth of published material on how to best exploit these capabilities and, consequently, many UML profiles are either invalid or of poor quality. In this paper, we first provide an overview of the new extensibility mechanisms of UML 2.1 and then describe a method for defining profiles that greatly increases the likelihood of producing technically correct quality UML profiles.

1. Introduction

With increasing knowledge and experience comes the need for greater domain specialization. In the field of software engineering this trend has many manifestations including the emergence of specialized computer languages. For example, Fortran focused on scientific numerical computing, whereas Cobol supported business data processing. These early examples were followed by hundreds and possibly thousands of specialized computer languages for a wide spectrum of application domains such as artificial intelligence, accounting, financial report writing, real-time, etc.

The essential feature of all such languages is that specialized domain concepts are rendered as first-order language constructs, as opposed to being realized through a combination of one or more general constructs. This can greatly ease the programmer's task because it enables direct expression of problem-domain patterns and ideas.

Yet, although new domain-specific languages are still being proposed, the vast majority of current software development is written using standard general-purpose programming languages such as Java, C, or C#. While this may seem as an anomaly that bucks the trend towards increasing specialization, it is due to some very practical considerations.

The first and probably most important reason for this is that it is the typical lack of an adequate support infrastructure for highly specialized languages. This includes compilers, editors, debuggers, linkers, and other tools needed for large-scale industrial software development. Without such tools, even the most elegant and well-designed programming language is of little practical value. Furthermore, the mere availability of such tools is not enough, they must also be well-designed, user friendly, and effective in what they do. The time and effort required to develop an effective, scalable, and robust infrastructure for a computer language typically far exceeds the time and effort required to define the language itself. Since specialized languages are used by a small community there is little economic incentive for major tool vendors to develop tools for them.

A second factor that favors general-purpose languages is the ready availability of trained programmers. The more specialized a language, the more difficult it is to find people who are proficient in it. Consequently, projects that opt for specialized languages may be incur additional costs for training as well as for developing courses, course materials (books, exercises, etc.) and teaching staff.

Yet another typical problem with domain-specific languages is the lack of pre-packaged program libraries like those that are typically available for most general-purpose languages. The open source movement is effective because it is limited to a small number of general-purpose languages.

For all these reasons, instead of relying on specialized languages many organizations rely on domain-specific program libraries (written using general-purpose programming languages), and specialized programming frameworks such as .Net [7], J2EE [11], and Eclipse [2]. *Program libraries* typically encapsulate domain-specific concepts in the form of packaged subroutines or as reusable classes (for object-oriented languages), thereby providing facilities similar to first-order language constructs. *Programming frameworks* directly implement the common domain-specific concepts (patterns) and functionality and also provide facilities for specialization to suit individual applications.

While these two approaches do not completely eliminate the drawbacks of domain-specific languages, the cost involved can be significantly reduced. For example, some domain-specific training and its related costs are still required, but not as much as for a domain-specific language since the widely-available knowledge of the base language is reused.

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

Nevertheless, because they are based on general-purpose programming languages, from the programmer's viewpoint, program libraries and frameworks are usually not quite as powerful or as expressive as domain-specific languages. For instance, the programmer is still bound to the programming model and syntax of the general-purpose programming language, even though these may not be the most suitable for the chosen application domain.

In this paper, we explain how the UML profile mechanism can be used to define expressive domain-specific *modeling* languages (DSMLs) while retaining most of the benefits of program libraries and frameworks. While this approach is only valid for domain-specific languages that are conformant with (i.e., not in conflict with) the semantics and abstract syntax of standard UML [7], it still covers a very broad range of application domains thanks to the generality and the diversity of UML.

In the following section we first discuss different approaches to defining domain-specific modeling languages. In section 3, we provide an overview of the profile mechanism of UML 2, with special emphasis on the important new features specifically designed to enhance support for domain-specific modeling languages. Section 4 describes how these mechanisms should be used to ensure formally correct and effective DSML designs. To date, there has not been sufficient methodological guidance published on how to design UML profiles.

2. Domain-specific modeling languages

Effective programming language design requires significant skill and expertise. Fortunately, there is a substantial body of accumulated theory and experience to assist language designers. However, this is not the case for modeling language design, which is still an emerging discipline with very few proven and established guidelines and patterns.

As might be expected, some programming language theory is also applicable to modeling language design, but there are several key challenges in designing modeling languages that are unique. These include:

- *The need to simultaneously support different levels of precision*. A modeling language such as UML may be used in a variety of very different ways each requiring different degrees of formality and precision. Initially, a modeling language is likely to be used for rough and informal "sketching" of raw design ideas (see [3]. This requires a simple syntax and relatively lightweight and loose semantics. As the design firms up and turns towards implementation, to avoid error-inducing discontinuities, the same modeling language might be used for "high-level programming", with all the heavyweight formality and semantic and syntactic precision that this entails.
- *The need to represent multiple different but mutually consistent views of certain elements of the model*. This stems from the requirement of modeling languages to provide a more problem-oriented approach to system specification, as opposed to the implementation-oriented nature of most programming languages. A complex system may need to be specified from a variety of different perspectives, all of which should be complementary and mutually consistent.
- *The graph-like nature of most modeling languages*. Models of complex systems represent complex hypergraphs that are not easily reduced to a linear text-based representation that is behind most programming languages.

The lack of an established theory of modeling language design is a major impediment and should be taken into account when considering whether or not to define a brand-new DSML, even if there is adequate domain expertise available. A more prudent approach would be to model a DSML on an already established and proven modeling language.

There are three primary methods for defining a DSML, two of which are based on reusing an existing language:

1. *Refinement of an existing more general modeling language* by specializing some of its general constructs to represent domain-specific concepts.
2. *Extension of an existing modeling language* by supplementing it with fresh domain-specific constructs.
3. *Definition of a new modeling language* from scratch.

Clearly, the last of these has the potential for the most direct and succinct expression of domain-specific concepts. However, it suffers from the serious drawbacks described previously. Furthermore, it is generally more difficult and expensive to develop tools for modeling languages than for programming languages, due to the usually more sophisticated semantics behind many modeling language constructs. For example, a tool that compares two state machine models must "understand" the semantics of state machines.

In essence, the same set of problems is encountered in the second method but in a somewhat milder form because some degree of infrastructure and expertise reuse can be expected.

For these reasons, the first method based on refinement is often the most practical and most cost-effective solution to DSML design. If properly designed, an extension-based DSML allows reuse of the tooling infrastructure of the base language, access to a broader base of trained experts, and usually requires less specialized training. On

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

the other hand, its principal disadvantage is that the expressive power of the DSML may be diminished due to the semantic and syntactic limitations of the base language. Therefore, it would be wrong to conclude that this approach is optimal in all situations.

3. The UML profile mechanism

The *profile mechanism* of the Unified Modeling Language (UML)[1] was designed to support the refinement approach to DSML design. It is restricted to DSMLs that fall within the syntactic and semantic envelope defined by standard UML. This is because, by definition, a UML profile cannot violate any of the abstract syntax rules and semantics of standard UML.

From its inception, UML was designed to be customizable, as a potentially family of languages. Its definition includes many *semantic variation points* and it also provides special language constructs for refinement. Semantic variation points are areas in which the UML specification either explicitly or implicitly allows for multiple possible interpretations. These can be reduced by supplying additional constraints andother refinement mechanisms of UML such as stereotypes. For example, standard UML allows supports both single and multiple inheritance. However, this can be limited to just single inheritance simply by defining one additional constraint that limits the number of ancestors of a class to no more than one.

The basic refinement mechanisms of the initial versions of UML, *stereotypes* and *tagged values*, were relatively lightweight and, unfortunately, rather vaguely defined. They permitted attaching domain-specific semantics to selected elements of UML models. For instance, a particular class in a UML model could be selected to represent a mutual-exclusion semaphore device by tagging that class with a custom-built "semaphore" stereotype. Through the application of the stereotype, the model element automatically acquired the semantics associated with the "semaphore" stereotype in addition to its standard UML class semantics. Conversely, removing the stereotype from the model element resulted in the loss of its semaphore semantics.

A *stereotype definition* consisted of a user-defined stereotype name, a specification of the base UML concept (e.g., Class) for the stereotype, optional constraints that specified how the base concept was specialized (e.g., a Class that can have at most one parent), and a specification of the semantics that the stereotype adds to the base concept semantics. The latter was typically specified using informal natural language.

A UML tool supporting the standard treated an element tagged with a stereotype like any other UML element without any special treatment (although it might check to determine if the constraints associated with the stereotype were valid—but only when such constraints were expressed in OCL). However, an external tool or a custom version of a standard UML tool that was sensitive to the stereotype semantics could detect such elements and interpret them in the appropriate way. For example, a concurrency analysis tool might be able to detect potential concurrency conflicts in a model by analyzing the accesses to model elements tagged as semaphores.

Since stereotypes capture domain-specific concepts, they are typically used in conjunction with other stereotypes from the same domain. This led to the concept of a *profile*, a specially designated UML package that contained a collection of related stereotypes. (To the best of the author's knowledge, the UML profile concept was first proposed by Philippe Desfray.)

Unfortunately, these early UML refinement mechanisms often proved inadequate and lacked the expressive power and precision required to design truly useful DSMLs required for modern model-driven development.

3.1. Innovations to profiles introduced in UML 2

Many of the shortcomings of the initial profiling mechanism were eliminated in UML 2. Unfortunately, these improvements have received very little coverage in popular UML textbooks and are still relatively unknown and underutilized. The following are the most important of these innovations:

- An expanded and more precise definition of profiles and stereotypes was provided. Thus, in UML 2, a stereotype is semantically very close to the concept of a metaclass, which it is intended to emulate (in a restricted way). Many ambiguities in the original definition were clarified.
- Formal rules for writing OCL constraints attached to stereotypes were introduced.
- A new and more scalable notation for stereotypes and their attributes was added.
- The semantics of applying (and un-applying) profiles to UML models were both expanded and clarified.
- The rules for an XMI representation of profiles and their contents were defined.
- It became possible for a profile definition to be based on an just a subset of the UML metamodel (as

[1] In principle, the profile mechanism is part of the MetaObject facility (MOF) [4], which means that it can be applied to any MOF-based metamodel, not just UML. However, in this article we will focus exclusively on the case of UML, since it is the most widely used MOF-based modeling language, with the understanding that most of the techniques and recommendations discussed here apply equally well to other modeling languages.

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

opposed to the full metamodel), resulting in potentially very compact and simple DSMLs.
- The ability to create new associations between stereotypes and other metamodel elements was provided. This allowed stereotypes to have associations that do not exist for their metaclasses.
- A default mechanism was introduced that freed modelers from having to explicitly stereotype each model element.
- The profile mechanism was generalized beyond the UML context so that it could be used with any MOF-based modeling language.

In the remainder of this section, we provide a brief description of the principal features of the UML 2 profile mechanisms.

3.2. Profiles

There are two significantly different ways in which UML profiles can be used:

A profile can be created *to define a DSML*. This language can then be used to construct domain-specific models using concepts from that DSML. An example of such a profile might be a language for modeling real-time applications. This language might provide domain-specific concepts such as Priority, Task, Deadline, etc. in place of the corresponding general-purpose UML concepts such as Class, Behavior, etc.

Alternatively, a profile can also be created to define a *domain-specific viewpoint*. Such a viewpoint, when applied to a standard UML model, casts that model in a domain-specific way and may also add supplementary information to the model relevant to the viewpoint. This allows the model to be interpreted from using the concepts of the viewpoint as opposed to standard UML. For instance, it may be useful to understand the performance characteristics of a particular design expressed as a UML model. One common way of analyzing performance characteristics of a system is based on queueing theory [1], which views a system as a dynamically balanced network of clients and servers. Using a performance viewpoint profile, it is possible to identify individual model elements as playing the roles of clients or servers along with the corresponding performance metrics, and then to compute the predicted performance characteristics of the design [12].

The ability to dynamically apply and un-apply a UML profile without affecting the underlying model is crucial to the second type of profile usage, because it allows the same model to be viewed from different viewpoints (e.g., performance, security, availability, timing).

A model to which a viewpoint profile has been applied can be passed to a specialized tool that can then automatically generate a new analysis-specific model based on the information embedded in the UML model. This new model can then be analyzed for interesting properties and the results fed back into the original UML model. By this technique, modelers can take advantage of many useful analyses techniques without having to be experts in those techniques. This cam be highly useful since many of these analyses are quite complex and require levels of expertise and skill that are scarce and expensive.

In general, it is possible to arbitrarily combine stereotyped elements with non-stereotyped elements in the same model. However, in some cases, modelers may need to limit their models to only those UML concepts allowed by a profile. This is known as a *strict* application of a profile. When the profile is applied in this way, all elements that are not allowed by the applied profile (or profiles), will be hidden from view by the modeling tool, although they remain unchanged in the model itself.

Note that "strictness" is not a characteristic of the profile itself, but of the application of a profile. This choice of which profile is applied, when it is applied, and the way in which it is applied (strictly or non-strictly) is made when a model is linked to a profile via «apply» dependencies between the model and the desired profiles[2].

The determination of which elements of the UML metamodel are visible and which ones are not is controlled by element or package import relationships between the profile and the model containing the UML metamodel. Package imports will make an entire metamodel package with all its elements available (visible), whereas element imports will import individual metaclasses from the metamodel. Because metamodel elements are dependent on other metamodel elements, care must be taken in selecting which elements to import to ensure that the resulting models are semantically complete. For example, if a profile does not make the Association concept visible, then applying that profile will effectively hide all associations in the model, which may result in incomplete and meaningless models. For a more detailed explanation of the "visibility" feature of profiles, the reader should refer to the standard itself [7].

3.3. Stereotypes

As noted previously, stereotyping of model elements is a convenient way of identifying elements in a UML model that have additional non-standard semantics; i.e., semantics that go beyond the confines of the UML standard itself. A stereotyped element can thus be distinguished from other elements and its associated

[2] To avoid having to dynamically insert and remove such dependencies, a modeling tool should provide the modeler with the capability to choose whether or not to enforce the «apply» dependencies.

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

semantics identified independently of any modeler-defined name.

A UML stereotype is, in effect, a subclass that specializes (refines) its base metaclass. But, why not simply define a new element of the UML metamodel in the standard way instead of using stereotypes?

There are two main reasons for this. The first is to allow tool implementers flexibility in choosing their preferred implementation approach. Note that adding a new metamodel element implies that, to accept new user-defined stereotypes, the modeling tool has to be a meta-case tool to a certain extent, that is, a tool that can easily and dynamical modify the metamodel (language) that it supports. However, while some modeling tools are indeed constructed in this way, many others are not. The latter category usually encode the rules and constraints defined by the UML metamodel in program code. This implies that, to change the metamodel it is necessary to reprogram the tool – an impractical option that does not allow users the flexibility to easily refine UML.

A second major reason is the need to support viewpoint type profiles, which require the ability to dynamically apply and un-apply stereotypes.

In implementation terms, when a stereotype is applied to a model element, a special attachment is created and linked to that model element. This attachment contains the information about the applied stereotype and any values associated with its attributes. This makes it possible to "un-apply" a stereotype without affecting the model element to which the stereotype was attached.

Stereotype definitions are often supplemented by constraints written in OCL. These are used to capture domain-specific constraints that apply to the stereotype but not to its base classes.

Note that a stereotype can specialize more than one base metaclass. The semantics of this are *not* the semantics of multiple generalization (i.e., multiple "inheritance"). Instead, it means that that stereotype can be applied to model elements that are instances of any of the base metaclasses (or any of their subclasses). This feature is used to allow a given domain-specific concept to be realized by more than one UML concept. A typical case where this is useful is when we need to apply a stereotype to either a type (e.g., Class) or to instances of a type (model elements typed by InstanceSpecification).

Modelers can apply stereotypes selectively to model elements of their corresponding base class. For instance, if «clock» is a stereotype of the Class metaclass, then it is not mandatory for all model elements that are instances of Class to be stereotyped by this stereotype. However, in some cases, it may actually be required that a stereotype must be applied to all instances of the base metaclass. For instance, if a model represents a C++ program, then all classes in the model should have the same C++ stereotype applied, since C++ only recognizes one type of class. For those situations, the profile designer has the choice to declare the stereotype to be "required", which means that all instances of the base class and its subclasses *must* have the stereotype applied to them whenever the corresponding profile is applied.

3.4. Model Libraries

A model library is a stereotyped package[3] that contains useful model fragments intended to be reused by other models, most notably profiles – although they can be reused by any model. If the library is defined in the context of a profile (package), then it is part of the profile definition. This allows complex domain-specific concepts to be captured using the full power of standard UML modeling constructs, unhampered by the limitations of stereotype modeling. A common application of such libraries is to use them to type stereotype attributes (tagged values).

However, it is important to note that, even when a library is part of a profile, the elements it describes are not metamodel elements but model elements. This means that, unlike stereotypes, they do not have any special semantics outside those provided by UML (unless, of course, they are themselves stereotyped). Still, they can be used to capture domain-specific semantics by association, so to speak, due to the fact that they are defined in the context of a particular profile.

For example, a model library for robotics, might introduce classes such as Robot, Manipulator, VisionSystem, etc. Outside the context of a profile, these are just classes with nothing to differentiate them from any other UML model elements. However, when they are used within or part of a profile, they implicitly acquire domain-specific semantics by association. In such situations, a tool that is sensitized to the encompassing profile can interpret these elements in a domain-specific way.

As a special degenerate case, a profile may be defined without any stereotypes, containing (or importing) nothing but model libraries. The drawback of this, however, is that such a profile cannot take advantage of some of the features of profiles, notably the ability to be dynamically applied or unapplied.

4. A systematic method for defining UML profiles

There has been little material published to guide designers of UML profiles. The consequence is that there are very many UML profiles that are either technically invalid because they contravene standard UML in some

[3] Model libraries are packages identified by the «modelLibrary» system-defined stereotype.

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

way (and, thus, cannot be properly supported by standard UML tools) or they are of poor quality. In this section, we describe a method for defining profiles that will avoid some of the most common pitfalls.

In this approach, the definition of a UML profile involves the construction of two distinct but closely related artifacts: a *domain model* (or metamodel) and the *profile* itself. The process commences with the initial definition of the domain model, which is then translated into the profile. However, depending on the complexity of the DSML and how well it conforms to UML, it may be necessary to iterate between these two artifacts. In addition to domain expertise, a close familiarity with the UML metamodel is also vital when defining a profile.

4.1. The domain metamodel

The primary purpose of a domain model is to specify what needs to be represented in the DSML and how. Experience with defining profiles has indicated that it is best if the initial domain model is defined strictly on basis of the needs of the domain, without *any* consideration of the UML metamodel (to which it will be mapped subsequently). This achieves a useful separation of concerns and ensures that the initial DSML design is not corrupted by contingencies of the profile mapping.

Unfortunately, far too many profiles start with the UML metamodel trying to fit the various domain concepts one-by-one within the framework that it provides. This conflates domain modeling issues and profile mapping issues and typically leads to loss of focus on domain concerns. The usual result is a DSML that is well aligned with the UML metamodel but which are suboptimal for domain modeling.

An "ideal" DSML metamodel, on the other hand, is an unpolluted specification of what the corresponding profile should provide. In practice, it may not always be possible to map this model precisely to the UML metamodel since the latter may have some constraints, metaclass attributes, or relationships, which are in conflict with the domain model. When that happens, it may be necessary to adjust the domain model with some loss of expressive power as a result. Naturally, if the profile strays too far from the ideal, then it may be the case that the profile-based approach is inappropriate for that particular DSML. (Note that, in such cases, the work invested in developing the domain model can be fully reused for a non-profile DSML.)

A domain model is metamodel of the DSML that should include the following key elements:

1. The set of *fundamental language constructs* that represent the essential concepts of the domain. For example, a DSML for modeling multi-tasking operating systems might include concepts such as tasks, processors, schedulers, task queues, priorities, etc. as language primitives.
2. The set of valid *relationships* that exists between the above domain concepts. In our operating system example, for instance, a scheduler might own multiple task queues (one for each scheduling priority level).
3. A set of *constraints* that govern how the language constructs can be combined to produce valid models. For example, a constraint could specify that at most one operating system task can be in the executing state per processor. The combination of relationships and constraints constitutes the *well-formedness* rules of the DSML. Together with the set of language constructs they represent the *abstract syntax* of the DSML.
4. The *concrete syntax* or notation of the DSML. This, of course, is the textual and/or graphical expression of the abstract syntax. Note that it does not necessarily have to be derived from or reminiscent of the UML notation.
5. The *semantics* or meaning of the language. This is a specification of the mapping between the elements of the abstract syntax and the corresponding domain entities that the syntactical elements represent.

The abstract syntax of a DSML is usually expressed using the OMG MOF (Meta-Object Facility) language [4]. In principle, other metamodeling languages can be used for this purpose, but MOF has the advantage that its models are easily translated into UML profiles. This is because UML metamodel is defined using MOF.

The basic language constructs are typically captured using MOF classes and attributes, while the relationships between them are represented either by associations or through class attributes. As much as possible, profile constraints should be written using OCL, since that language was specifically designed to be used with MOF and because it is supported by many UML tools.

For example, the relationship between processors, tasks, schedulers in the operating system DSML example would be captured using the following MOF model fragment:

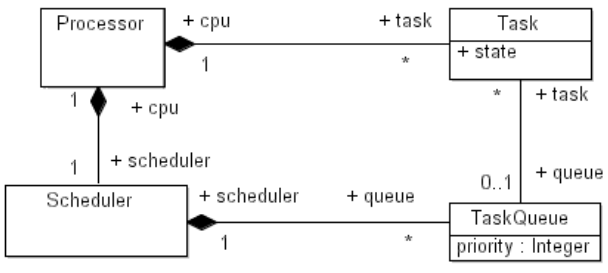

Figure 1. A partial domain model of a DSML

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

4.2. Mapping the domain model to a profile

Once the domain model is completed, the process of mapping it to the UML metamodel can commence. This is done by going step-by-step through the full set of domain concepts (specified as classes in the domain model) and identifying the most suitable UML base concepts for each.

For example, in our operating system DSML shown in Figure 1, we might conclude that the Processor domain concept is conceptually and semantically similar to the Node concept of UML, while the Task concept is related to the UML Class concept whose "isActive" attribute is set to "true". Thus, we would define a stereotype called Processor whose metaclass is Node and a stereotype called Task whose metaclass is Class. We could then define an attribute of the Task stereotype called "state" and also add a constraint that the "isActive" attribute of any Class tagged by this stereotype must have its "isActive" attribute set to "true".

Note that not all domain concepts need to be directly derived from corresponding UML metaclasses; some may be specializations (subclasses) of other "abstract" stereotypes in the profile. For example, a generic "semaphore" domain concept might be defined as a stereotype of the UML Class concept. This stereotype can then be refined further by multiple subclass stereotypes corresponding to different flavors of the generic concept (e.g., queueing semaphores, binary semaphores, etc.).

The richness and diversity of concepts in the UML metamodel provides a very good foundation on which to base concepts for a very large number of DSMLs. However, although it may be easy in most cases to find a corresponding base metaclass for a domain concept, this is not sufficient. Care must be taken to ensure that the selected base metaclass does not have any attributes, associations, or constraints that are contradictory to the semantics of the domain concept. Often, such issues can be resolved by simple constraints. For example, a conflicting attribute or association end from a base metaclass can be eliminated by forcing its multiplicity to be zero using a constraint (provided that it has a lower multiplicity bound of zero).

The following guidelines should be used for mapping domain concepts to UML metamodel elements:

1. *Select a base UML metaclass whose semantics are closest to the semantics of the domain concept*. This is very important since the semantics of UML concepts are often built into UML tools and also because people who know UML will naturally expect a stereotype to inherit the semantics of its base metaclass. After all, the reuse of UML tools and UML expertise are among the primary justifications for the profile approach.

   Unfortunately, all too often, base metaclasses for stereotypes are chosen on the basis of a purely syntactic match. For instance, contrary to what might be expected, the OMG SysML profile [5] does not use the UML Dependency concept to capture certain kinds of functional relationships that exist in a model. Instead, it uses the Class concept for this purpose because that allows reuse of some notational forms used with UML classes. In general, purely syntactical matches of this type should be avoided since they will lead to confusion and misinterpretation by tools.
2. *Check all the constraints that apply to the selected base metaclass to verify that it has no conflicting constraints.* Note that it may not be sufficient to check just the base metaclass for such constraints but also all of its superclasses, if they exist.
3. *Check to determine if any of the attributes of the selected base metaclass need to be refined.* This is a way of specializing the base concept for domain-specific semantics. For example, in modeling a task in the example operating system DSML, we added a constraint that specified that the "isActive" attribute must be set to "true". Constraints of this type may be used to define domain-specific default values of attributes and also to eliminate attributes that may not be relevant to the domain (by setting their lower multiplicity bounds to zero).
4. *Check to determine if the selected base metaclass has no conflicting associations to other metaclasses.* These would be conceptual links inherited from UML that contradict domain-specific semantics in some way. Fortunately, many of these can be eliminated by the above technique of setting their lower multiplicity bounds to zero. However, if this is not possible, then this may not be the appropriate metaclass despite its semantic proximity to the domain concept.

4.3. On specifying DSML semantics

As noted, semantics are a fundamental part of any computer language, including DSMLs. However, presently, semantics are mostly specified informally or semi-formally using natural language. This makes it difficult to ensure a precise shared interpretation of a DSML. Ideally, the semantics of a language should be specified in the same formal manner as the syntax. At present, there is no standard associated with UML that would allow a formal specification of the semantics of a UML profile. However, there is work currently in progress to address that shortcoming: the Executable UML Foundation specification [6].

Once completed, the intent is that this specification will provide a formal specification of the semantics for a carefully-selected subset of UML. This subset includes some of the very basic structural concepts of UML such as objects and links as well as some of the core primitive

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE

UML actions that operate on (read and write) these structural concepts, but it excludes most higher-level concepts such as state machines and more complex activities. The result is a kind of primitive (but formally specified) programming language, which can then be used to define the semantics of higher-level UML or DSML constructs by writing "programs" in this language. This is known as the *operational* approach to defining semantics. A profile definition (or any MOF-based DSML definition) could then include a formal (and interchangeable) specification of semantics of its constructs in addition to its formal syntax specifications. This specification is planned to be completed in the summer of 2007.

5. Summary

The profile mechanisms of UML 2 provide a powerful capability to define new DSMLs. They have some distinct and important advantages over other approaches to defining DSMLs. Specifically this approach allows reuse of standard UML tools, expertise, and related resources such as training materials, textbooks, etc. While its applicability is limited to DSMLs whose abstract syntax and semantics are confined to the envelope defined by standard UML, it still covers a very broad range of possible domains, due to the diversity and richness of UML.

The method for defining UML profiles that is introduced in this paper was designed to ensure technically correct and effective profiles, which take full advantage of the new and improved features introduced in UML 2. It clearly separates the DSML definition from the profile definition, achieving thereby an important separation of concerns. In addition, it identifies the elements required to construct a domain model and provides detailed guidelines on the process for mapping the domain model to a UML profile.

The method has been applied successfully to a number of profiles including, a few standard UML profiles such as the OMG UML profile for Schedulability, Performance, and Time [9] and the upcoming profile for Modeling of Real-Time and Embedded Systems (MARTE) [8] .

References

[1] R. Cooper, *Introduction to Queuing Theory*, 2nd edition, North-Holland (Elsevier), 1972.

[2] Eclipse Foundation, "What is Eclipse?", http://www.eclipse.org/org/ , 2007.

[3] M. Fowler, *UML Distilled*, Addison Wesley, 2005.

[4] Object Management Group, *Meta-Object Facility (MOF): Core Specification*, Version 2.0, OMG document formal/2006-01-01, 2006.

[5] Object Management Group, *OMG SysML Specification*, OMG Adopted Specification, OMG document ptc/06-05-04, 2006.

[6] Object Management Group, *Semantics of a Foundational Subset for Executable UML Models – Request for Proposal*, OMG document ad/05-04-02, 2005.

[7] Object Management Group, *Unified Modeling Language: Superstructure (convenience document)*, Version 2.1, OMG document ptc/06-04-02, 2006.

[8] Object Management Group, *UML Profile for Modeling and Analysis of Real-Time and Embedded Systems – Request for Proposal*, OMG document realtime/05-02-06, 2005.

[9] Object Management Group, *UML Profile for Schedulability, Performance, and Time*, Version 1.1, OMG document formal/2005-01-02, 2005.

[10] Microsoft Corporation, "What is .NET?", http://www.microsoft.com/net/basics.mspx , 2007.

[11] Sun Microsystems, *J2EE v1.4 documentation,* http://java.sun.com/j2ee/1.4/docs/ , 2007.

[12] D. Petriu and A. Sabetta, "From UML to Performance Analysis Models by Abstraction-raising Transformation", in *From MDD Concepts to Experiments and Illustrations*, (eds. J.P. Babau, J. Champeau, S. Gerard), ISTE Ltd., pp.53-70, 2006.

Proceedings of the 10th IEEE International Symposium on
Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)
0-7695-2765-5/07 $20.00 © 2007 IEEE