Aspect-Oriented Metamodelling

Tony Clark[1], Andy Evans[2] and Stuart Kent[3]

[1]*Department of Computer Science, Kings College, London, UK*
[2]*Department of Computer Science, University of York, UK*
[3]*Computing Laboratory, University of Kent, UK*
Email: anclark@dcs.kcl.ac.uk, andye@ukc.ac.uk, sjhk@ukc.ac.uk

The Object Management Group's (OMG) Model Driven Architecture (MDA) strategy envisages a world where models play a more direct role in software production. To ensure that the burden of maintaining more than one formal model does not outweigh the potential benefits, powerful tool support is required. However, each domain, organisation, even project, is likely to need its own particular process supported by its own particular configuration of modelling languages. The challenge is to provide definitions of languages that not only support the configuration and extension of those languages for use with particular processes, but also can be used directly in the customisation/generation of tools. This paper describes an aspect-oriented, metamodelling approach to language definition which aims to meet this challenge. This exploits two mechanisms (package extension and package templates), which are similar to mechanisms proposed to support aspect-oriented design with UML. Thus the paper can also be viewed as describing a case study in aspect oriented design. The approach is illustrated by extracts from the 2U submission to the UML 2.0 RFPs issued by the OMG in 2001. The paper concludes with a discussion on the customisation and generation of tools from such definitions.

1. INTRODUCTION

As the complexity of systems continues to increase, industry is looking for better ways to abstract from detail and separate concerns. There are two complimentary approaches that seem to be gaining some credence: aspect-oriented and model driven approaches to software development. The aspect-oriented approach treats software as a multi-dimensional artefact [21], whose development can be made more tractable by allow different dimensions, or aspects, to be defined separately and then merged or woven together to produce the required result. The model-driven approach envisages a world where (high level) models play a more direct role in software production, being amenable to manipulation and transformation by machine. One realisation of this approach is the Object Management Group's (OMG) Model Driven Architecture (MDA) strategy [19], which focuses, in particular, on the separation between platform independent and platform specific models.

In [18], we argue for bringing together these two approaches, and envisage a multi-dimensional modelling space which includes the abstract/concrete separation as one of its dimensions. The idea is that for any particular development project one defines a development process that, amongst other things, identifies the artefacts (models) to be manipulated, and the mappings between them. It may be that the process is aimed at constructing new artefacts or refactoring existing artefacts, or some combination of the two. It may be that there is only one artefact — the program, as recommended by more recent agile approaches to software development.

Identifying models might proceed as follows. First, identify the modelling dimensions that need to be considered, which includes defining the points of interest along those dimensions. Some of those dimensions are likely to be standard across projects, for example authorship and version, only needing to decide whether the dimension needs to be included or not. A subject area dimension is likely always to be present, and that will require the subject areas to be identified. Similarly one will need to decide the relevant points on (an) aspect dimension(s), for example whether points are required for concurrency or distribution, for information and data, and so on. One will also need to define the stakeholders involved; and the levels of abstraction that are of interest (for MDA, one may choose that the only two points of interest are PIM and PSM along the latter dimension).

Second, determine what intersections of the various dimensions are of interest for the development project at hand. This determines the models that need to be considered. In theory, there could be a different language for describing models at each intersection. In practice, the language used at a particular point of intersection will be determined by one (or a small subset) of the dimensions that determine that point. For example, a subject area may be defined from the perspective of many different aspects, and the language used might

be determined by the aspect being described. The language is unlikely to be different for different subject areas. It may also be that the level of abstraction and/or stakeholder will influence the language to be used. With regard to the latter, a business expert and programmer are likely to need to see the specification of a system in different languages, even though the two renderings would be isomorphic — they are at the same level of abstraction, looking at the same aspect and subject area.

If one is to believe this scenario, then it is very likely that processes, and accompanying modelling spaces, will be populated differently for different domains, organisations and even projects. Evidence of this is already emerging in the recent surge of proposals for UML profiles (specialisations/variations on UML for modelling in specific contexts) at the OMG. It is also likely that there will be considerable overlap between languages used for modelling at different points in the space, as well as differences. If one were able to support the definition of product lines or *families* of languages, then this overlap could be exploited to make definition of specialised/bespoke languages, and the mappings between them, easier, and in provisioning tool support.

An alternative approach is to provide a single, general-purpose language which includes mechanisms for defining and combining models from different perspectives. For example, one could take a general purpose modelling language, like UML, and add mechanisms, such as (package) templates and (package) composition. This would allow, for example, aspects to be encoded as templates capturing parameterised, commonally-used modelling patterns, and for different modelling perspectives (some stamped out from templates and some constructed bespoke) to be combined using composition. Such an approach to modelling, which builds on work in subject-oriented and aspect-oriented programming, has been proposed by [9, 10].

This approach is attractive because it means that only one language needs to be supported, for example in the provision of educational materials and tooling. It is quite restricted, however, as it presumes that the general purpose modelling language will be accessible to all stakeholders, and be rich enough to support all modelling requirements. Of course, in some restricted domains it may be all that is needed — indeed, this paper proposes the use of such mechanisms for aspect-oriented metamodelling; but in general, there will remain a need to define languages to support the modelling of different aspects of a system, and, what is more, those languages are likely to vary across domains, organisations and projects.

To ensure that the burden of maintaining more than one formal model does not outweigh the considerable benefits of models as tools for abstraction, summarising and providing alternative perspectives, powerful tool support is required. On the other hand, building tools by hand is resource intensive, and certainly could not be done for every project-specific configuration of languages and mappings. And here is the challenge: to provide an approach to defining modelling languages and mappings, which not only supports the flexibility to specialise and configure those definitions, but which also can be used to customise and/or generate tools to support that particular configuration.

This paper describes an approach to language design which begins to meet this challenge. The approach extends metamodelling techniques with mechanisms suitable for aspect-oriented modelling. Metamodelling has come to mean the use of an object-oriented modelling language for the definition of languages, and is the approach used by the OMG to define its modelling languages. Currently these are defined using a subset of UML, or the Meta Object Facility (MOF). UML and MOF are currently being revised [4, 5], and one goal of this revision is to make the MOF language *the* subset of UML used for metamodelling. In our approach to metamodelling, we have employed mechanisms (package extension and package templates) for aspect-oriented modelling. These originated with the Catalysis method [11] and are similar in spirit, though slightly different in execution, to those suggested by [9] for aspect-oriented software design. (The paper does not provide a metamodel for these mechanisms, which are being defined elsewhere [8].) The approach proposes that language definitions should be architected to support the development of families of languages, and that this is made possible by the use of package extension and template mechanisms in metamodelling. In particular, the mechanisms can be used to combine existing language units to form languages, as well as customise (construct) existing (new) units, using predefined templates that capture patterns or aspects of language design. The templates can also be customised.

The paper is structured as follows. Section 2 describes package extension and package template mechanisms which we use in conjunction with usual object modelling techniques. Section 3 describes the overall architecture of the definition of a family of languages, and how this supports extension and customisation of languages. Section 4 outlines the application of this approach to the definition of UML in the 2U submission to UML 2.0, using fragments of that submission as illustration. Section 5 discusses the challenges in using the metamodel definitions to customise/generate tools. Section 6 provides a summary, considers limitations of the approach and looks forward to future work.

2. PACKAGE EXTENSION AND TEMPLATES

The package extension and template mechanisms were originally suggested as part of the Catalysis approach to software development [11]. The mechanisms used in this paper are a refinement and concrete realisation of those original ideas, and are in the process of being rigorously defined [8]. These mecha-

nisms are similar in principle to those described in [9] for aspect-oriented software design. Clarke defines different notions of package composition, one of which is to merge packages, matching model elements across packages by name. Let the Clarke mechanism be represented by an operation `mergeByName`, so that `mergeByName(name,{P1,..,Pn})` results in a package, with name `name`, that is the merge of `P1` to `Pn`, matching model elements by name. Now let the Catalysis extension mechanism be characterised as a tuple `<Q,body,{P1,..,Pn}>`, where `Q` is the name of the child package, `P1` to `Pn` are the parent packages, and `body` is the locally-defined part of `Q`. Then the expansion of this structure could be defined as `expansion(<Q,body,{P1,..,Pn}>) = mergeByName(Q,{body,P1,..,Pn})`. This equation also shows that `mergeByName` could be defined in terms of `expansion`, by making `body` empty.

As we see below, package extension also allows elements to be renamed. Similarly, Clarke allows elements to matched explicitly, rather than by name. These mechanisms could be factored in, but the basic comparison between the two approaches would remain the same.

The package extension mechanism is illustrated by Figure 1.

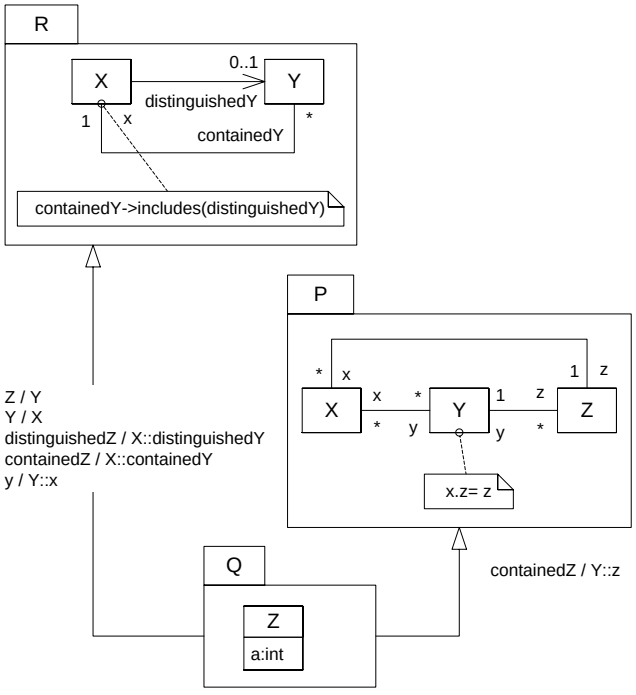

FIGURE 1. Package extension example

`Q` is a package that extends `R` and `P`. Extension between packages is shown by a UML generalisation arrow. The contents of `R` and `P` get included in `Q`, with anything common between the two being merged. Common model elements are elements of the same kind with the same name. Renaming clauses may be used to annotate a package extension either to prevent a merge or to force one. In this case, the classes `X` and `Y` in `R` are renamed to `Y` and `Z`, respectively, to force them to be merged with the classes `Y` and `Z` in `P`. `Q` also contains a fragment a class `Z`, with an attribute `a`, that is also merged with the `P::Z` and `R::Y` (which is renamed to `Z`). The unfolding of both package extensions results in the expansion of `Q` which is given in 2.

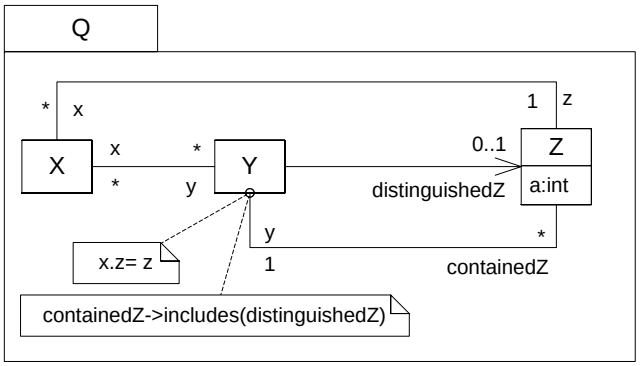

FIGURE 2. Expansion of child package

Package templates allow a package definition to be parameterised over arguments, thereby supporting the encoding of common patterns which can be bound to particular fragments of model through parameter substitution. The package template mechanism is illustrated by Figure 3.

This is similar to the package extension example of Figure 1, except that now package R has been turned into a package template. The template takes two string arguments (X and Y in the dashed box), and names of elements in the package are parameterised by these arguments. Not only are the names of classes parameterised, but also the labels on the association ends, which are referred to in the accompanying constraint. Instantiation of a template is shown using a generalisation arrow, which must be annotated by a substitution for the arguments, shown by a dashed box called out from the arrow. Template instantiation works by evaluating the expressions that provide the names for elements in the template with arguments substituted. The result is then merged with the target of the instantiation. A template instatiation may be annotated further with one or more renaming clauses, which override any names calculated from the argument substitutions. In this example there are no such renamings.

Templates effectively allow a (sometimes large) set of renamings to be calculated from a small number of arguments. In this example, the five renamings on the extension from `R` to `Q` in Figure 1 are replaced by a substitution for two arguments. Not only does this save work for the modeller, it also ensures more accurate use of the template by forcing a particular set of renamings (which may be overridden in extremis) whenever the

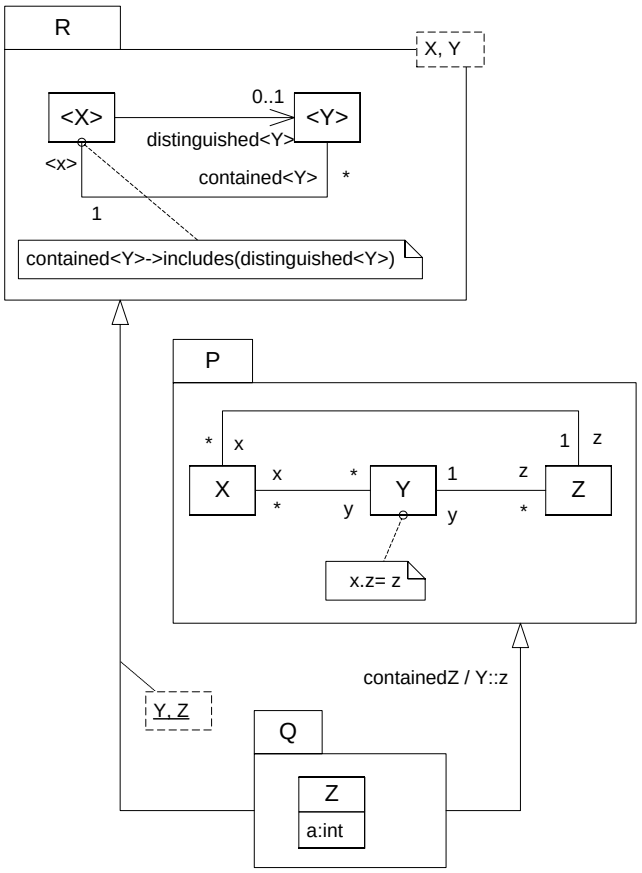

FIGURE 3. Package template example

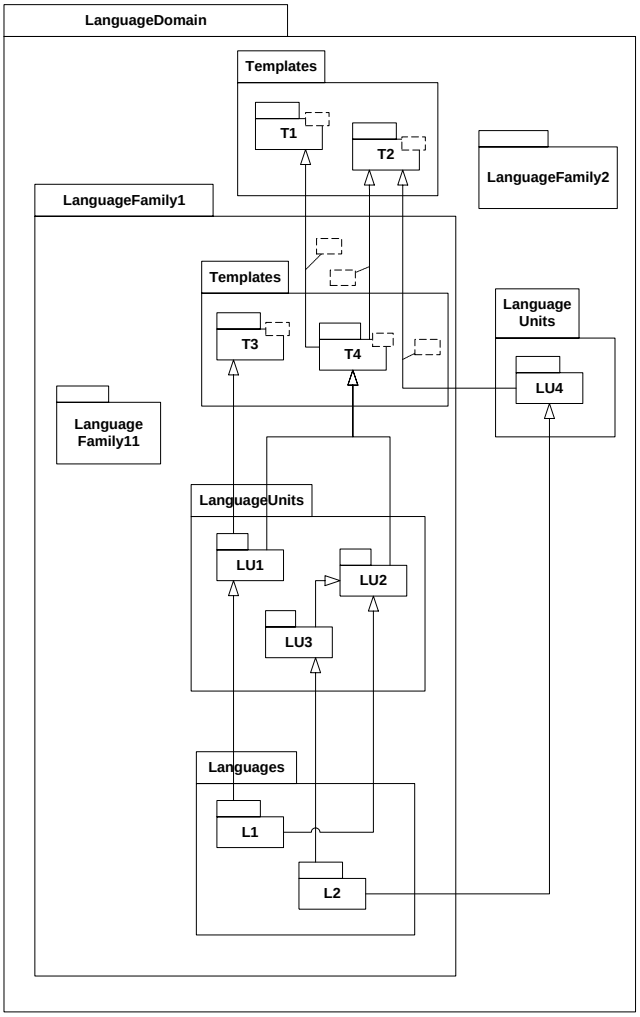

FIGURE 4. Language definition architecture

template is applied.

A rigorous and detailed definition of package extension is provided in [8]. A definition of package templates is in preparation.

3. METAMODEL ARCHITECTURE

Figure 4 illustrates the general architecture of our approach to language definition. The outermost package represents the domain of *Language Definition*. Within this a number of language families are defined (e.g. `LanguageFamily1` and `LanguageFamily2`). There may also be some templates (most likely) and language units (less likely) generic to language definition. A language family (e.g. LanguageFamily1) may itself have its own locally defined templates and language units (both likely). A language family may have sub-families (e.g. `LanguageFamily11`), which may extend/instantiate language units/templates from any family in which they are nested. A language family includes a set of languages which may extend/instantiate language units from that family or any family nested within it.

Language units allow related features of a language to be grouped into separate fragments; fragments may be common to many languages. Language units can be composed, using package extension, to form complete languages. Package extension may also be used to incrementally extend language units. Package templates capture cross-cutting architectural patterns, and can be used to impose a uniform and consistent architecture across definitions. The latter is essential for the composition of language units to work correctly. They also ensure more complete definitions by enabling reuse: important structures and constraints are captured once in a template and reused many times over in stamping out definitions of language units. In this way, one is able to reap the rewards from effort invested in a template.

The architecture of language units and languages is given by Figure 5. There is an *abstract syntax*, to which a *semantics* is given through a mapping to a *semantics domain*. The abstract syntax may take on many concrete forms; the mapping between each of these and the abstract syntax can also be defined as part of the metamodel.

The abstract syntax characterises, in abstract form, the expressions of the language in question. If the lan-

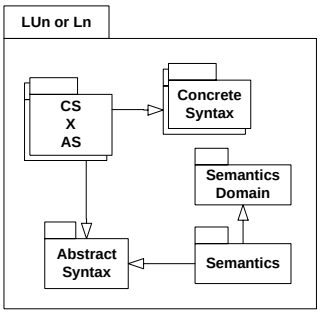

FIGURE 5. Language unit architecture

guage is used to make general statements about the behaviour of the system being described, the semantics domain represents, again in abstract form, the domain of examples and counter-examples of that behaviour. Semantics is a statement of what it means for an element of the semantics domain to be a valid example of a described behaviour. We say the example *satisfies* the behaviour; a counter-example is one that does not.

The distinction between language units and languages is somewhat fuzzy. Although many language units will be mini-languages in their own right, it is expected that they only become practically useful when combined with other units to form a language. Thus the languages are combinations of language units that the designer of the language family has deemed to be fit for a particular purpose.

The proposed language architecture supports the development of new languages through the following steps, which assume that (a) the appropriate language family has already been identified and (b) there is not already a language in that family suitable for the task at hand. The steps are listed in order of difficulty.

- Identify appropriate language units. It may be that all that is required is some composition of the existing language units. In which case, defining the language is a matter of having the new language extend each of the chosen language units.
- Specialise existing language units. The language may require some specialisation of language units before they are composed. For example, it may have stronger well-formedness constraints, or specialist forms of certain model elements. In this case, those units should be extended, and the extended versions merged with any other units required to form the language.
- Create new language units. If there are elements of the language which can not be supplied by existing language units, then it will be necessary to construct new language units. These could be created from scratch, or existing templates used to generate the new unit. The application of templates will depend on the richness and flexibility of the template library. (One would hope that there would already be a rich library of templates cultivated to support language definition.

4. APPLICATION TO UML2

The approach is being used in the 2U submission [20] to the UML 2 revision process, and fragments of this approach have been published in a series of papers [13, 1, 3, 7]. This section gives an overview of some of the templates, language units and languages being defined in that submission. (Please note that the 2u submission is still work in progress at the time of writing, so the fragments presented here may not exactly correspond with the final version of the submission.)

4.1. Templates

The 2U submission follows the architecture outlined in Section 3. According to this, templates are organised into two groups: those fundamental to language definition in general, and those particular to the language family being defined, in this case UML. Templates capture cross-cutting patterns of language definition.

Figures 6 and 7 provide an overview of some of the more fundamental templates.

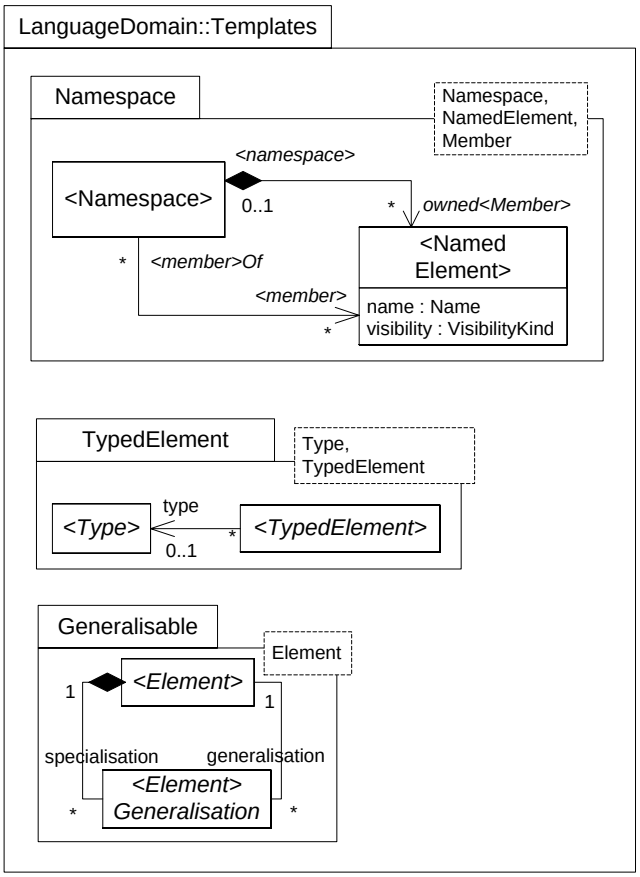

FIGURE 6. Generic templates (1)

The `Namespace` template characterises the situation common in many languages where named elements are

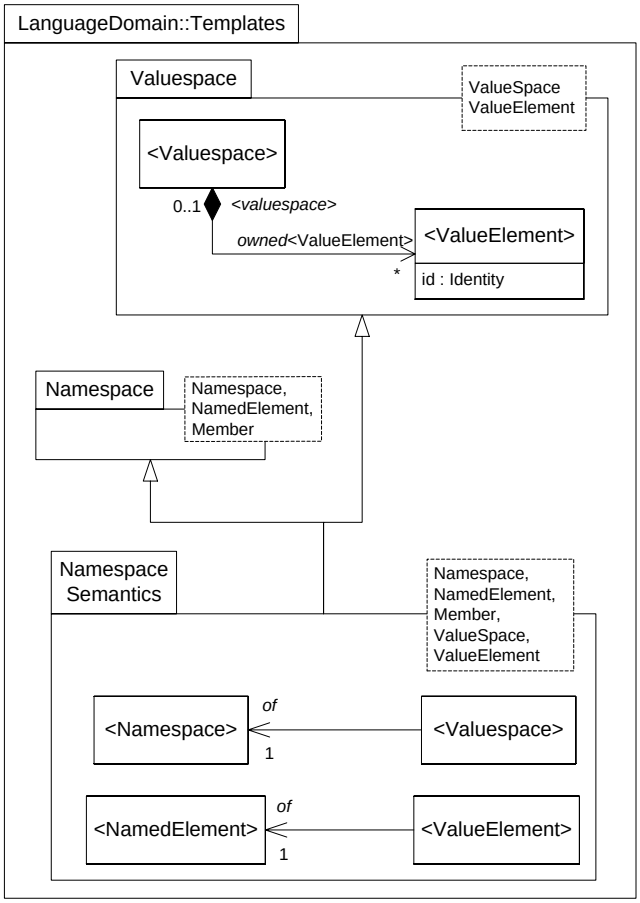

FIGURE 7. Generic templates (2)

owned by some namespace. It also identifies *members* of a namespace, which includes those owned by a namespace (for example, a class has members which it owns and elements which it inherits). `TypedElement` is self-explanatory. `Generalisable` is a template that captures the notion of generalisation and specialisation. `Valuespace` is like `Namespace` except that it is intended to be used in the semantics domain part of a language definition. This is then combined with `Namespace` to give a `NamespaceSemantics` template.

The templates are accompanied by various query operations and well-formedness constraints written in OCL. For example, the template `NamespaceSemantics` has the following well-formedness constraint, which ensures that the value elements in a valuespace are values of named elements in a namespace.

```
context NamespaceSemantics inv:
Values of <Namespace> contain
values of <NamedElement>
self.owned<NamedElementValue>->forAll(c |
   self.of.owned<NamedElement>->exists(d |
      c.of = d))
```

Note how the constraint, including the initial comment, is also parameterised by arguments of the template. When the template is instantiated, the result will include a constraint of this form, with template arguments substituted appropriately.

These templates illustrate the general principle of building a template library. That is, start simple, building up more complex templates in small incremental templates, often by combine other templates. For example, `NamespaceSemantics` is constructed by merging two more basic templates, then adding a couple of associations and constraints.

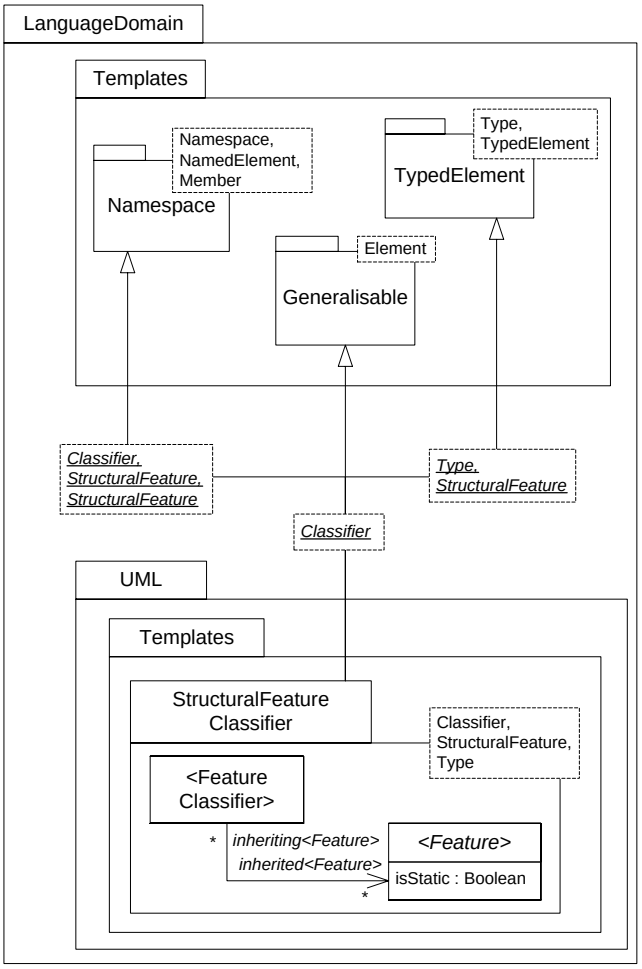

FIGURE 8. A UML family template

In the same vein, Figure 8 illustrates the derivation of a UML-specific template from the generic templates. This uses terminology familiar to the UML community, and provides, as a template, some of what is provided through abstract classes in a traditional metamodelling approach. It combines `Generalisable` with `Namespace` and `TypedElement`. The expansion of the template `StructuralFeatureClassifier` is given by Figure 9.

This template illustrates how package extension and template instantiation can be used to weave together cross-cutting aspects captured in separate templates. Here, the a generalisation aspect is being combined with

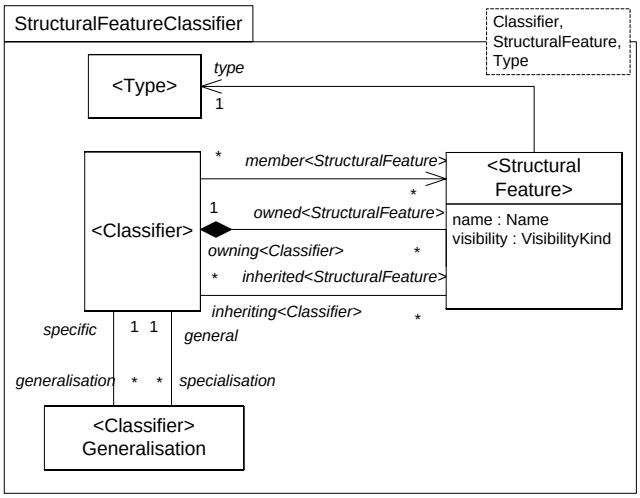

FIGURE 9. StructuralFeatureClassifier template

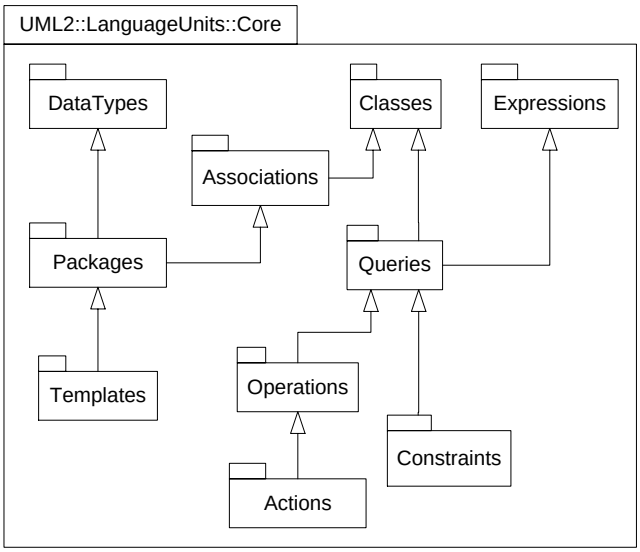

FIGURE 10. UML2 language units

a typed element aspect and a namespace aspect. In the future, we expect to develop families of templates corresponding to one aspect, and the language designer will make a choice as to which version of that aspect they wish to use. For example, this only one model of generalisation shown here. A simpler model would be to omit the `Generalisation` class. A more complex model would have to deal with the possibility of named elements being renamed when a namespace is generalised, and might look like something like the rather sophisticated model defined in [8].

Although not shown here, there is an equivalent semantics template, which, in particular, deals with semantics of generalisation. Other templates in development are aiming to capture the essence of expression languages, of action systems, and semantics for dynamic behaviour (traces, filmstrips, etc.).

There are also more basic templates that we are exploring. In particular, [2] describes a family of modelling patterns for describing mappings in metamodels, that can easily be captured as templates. Mappings are important not only for defining translations between languages, but also for defining the relationship between different aspects of a language: concrete syntax, abstract syntax and semantics domain.

4.2. Language Units

Figure 10 shows the current arrangement of language units in the 2u submission.

This arrangement grows language units in layers. Datatypes, Classes and Expressions are at the top of the tree. Associations are layered on Classes (their ends refer to classes) and Packages on Associations and Datatypes (packages can contain associations, classes and datatypes). Queries are layered on top of Classes (classes can contain queries) and Expressions (the result of a query is defined by expressions). Constraints are layered on Queries (constraints may refer to queries and are special kinds of expression—Queries extends Expressions). Operations are layered over Queries, and actions (which define the meaning of both operations and queries, plus a core action "calculus") on them.

Language units are built from templates. For example, consider the derivation of the `Classes` and `Associations` units, in Figure 11.

Both are built from the same template (`StructuralFeatureClassifier`) with `Associations` also being derived from a `Multiplicity` template. Only the derivation of abstract syntax has been shown. Semantics would be derived from one of the semantics templates (see Section 4.1) in combination with this. We have also omitted to show the model elements defined locally in each of the LUs. This can be deduced by comparing the expansion of the LUs in Figure 12 with the `StructuralFeatureClassifier` template in Figure 9. For example, in `Classes` a generlisation arrow between `Class` and `Classifier` has been added, and in `Associations` some extra detail about navigable ends is required.

The choice of language units is a matter of design. The current arrangement is quite coarse-grained, and has the advantage of remaining relatively uncomplicated, whilst still maintaining a separation of concerns. A disadvantage is that it limits the ways in which language units can be combined. For example, whilst it is possible to build a language from these units which, say, combines associations with classes queries and constraints, it is not possible to build one which combines packages with classes and constraints, but no queries and no associations (in such a language, constraints would apply to attributes). If one examines the language units closely, one sees that actually the dependencies between the language units are less than the

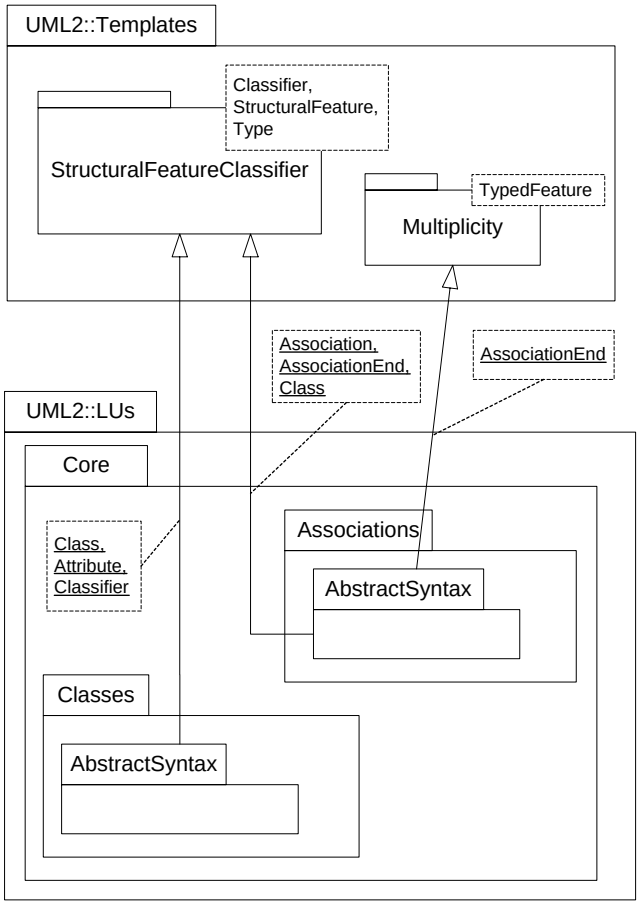

FIGURE 11. Derivation of Classes and Associations LUs

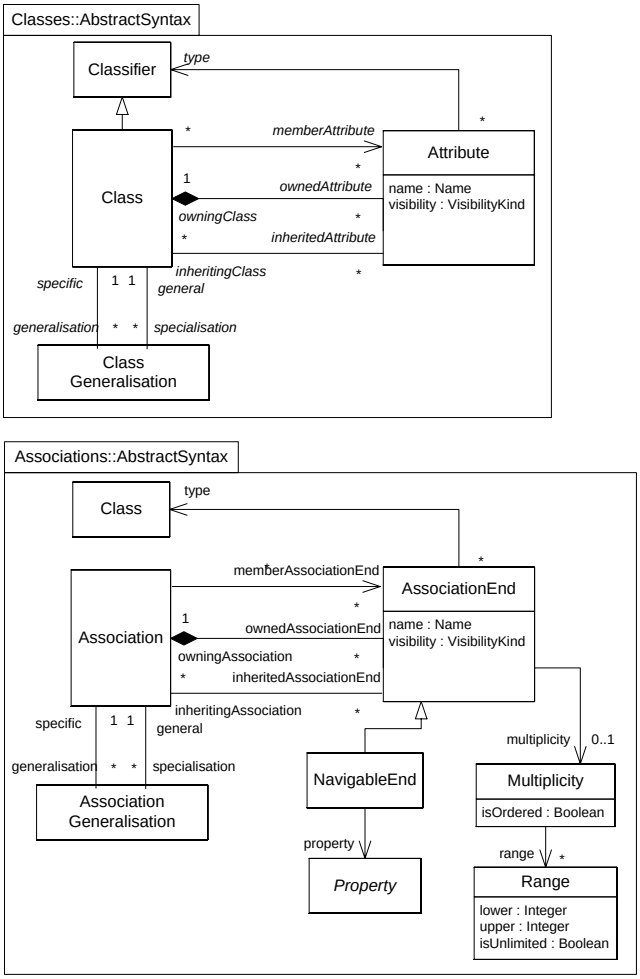

FIGURE 12. Classes and Associations LUs

current layering would suggest, so a refactoring that reduces layering and introduces greater orthogonality between the packages might be possible. Indeed, this has been achieved to some extent in the current model.

Consider again the `Classes` language unit in Figure 12. This shows the definition of an `Attribute` class with type `Classifier`. `Class` is a subclass of `Classifier`. Given that, in this language unit, the only possible types for attributes are classes, one might surmise that `Classifier` is redundant. It is included to provide a "plug point" for combination with other language units. So, if one wants a language that includes datatypes as well as classes, one can combine the Classes package with the Datatypes package. This has classes like Datatype and CollectionType which are also subclasses of Classifier. Separating `Classes` and `Datatypes` into separate LUs in this way, gives greater flexibility to the language designer, by allow him or her to mix a definition of classes with different collections of datatypes, or indeed anything else that the designer chooses could act as a type for an attribute.

It is interesting to note that subclassing is the device that has been used to support plugpoints when merging composing language units. The underlying reason is that a class in one language unit needs to a polymorphic reference to another class (an attributes type is a `Classifer`), which can be fulfilled by objects of classes provided by another language unit (`Datatype` and `CollectionType` in the `Datatypes` LU). Thus package extension does not mean that class inheritance can be dispensed with; only that the latter need not be used for reuse (templates can do that), but instead reserved for cases where a polymorphic reference is required.

4.3. Languages

The third element in the metamodelling architecture is the definition of languages. The difference between languages and language units is one of intended use. Language units are intended to provide relatively small, self-contained packages of language features that have been designed to be composed together in different combinations to form different languages in the family. Languages are intended to be used in a particular context.

There are many different applications to which UML (or, more accurately, specialised subsets of UML) are being used, and this has given rise to a number of dif-

ferent UML profiles [14], in addition to ad-hoc subsets that, for example, are typically used in books on UML. The 2U submission has so far focussed on the language units required to define two languages. The first is the modelling language that is part of the Meta Object Facility (MOF) [15]. This is the language used for metamodelling, which is also currently being revised [5]. The definition of MOF is given by Figure 13.

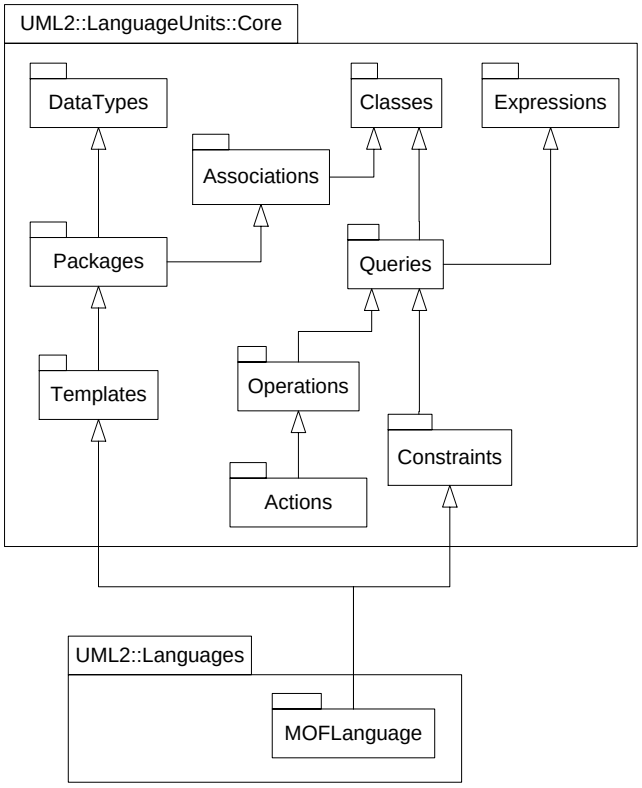

FIGURE 13. MOF language

The second language is a general purpose language for platform, independent modelling (PIM). This is like MOF, with the possible exclusion of templates (although the growing popularity of AOSD would suggest that they should be included), and the definite inclusion of actions. Such a language will also include additional language units layered on actions, dealing with interactions, state machines and operation specifications. Of course, the language architecture, means that there may be a small sub-family of PIM languages, which will provide different combinations of the language units.

Other languages deemed important, but which have not yet been brought into focus, include:

- Programming language specific variants of UML (e.g. depicting Java programs using class diagrams and interaction diagrams).
- Languages for describing component-oriented systems, using notions such as component, port and connector. The need for such languages is illustrated by two UML profiles, for Enterprise Distributed Object Computing (EDOC) [17] and Enterprise Application Integration (EAI) [16].

5. TOOLS

The vision is not only to have a means for defining families of languages and the mappings between them, but also for those definitions to be used to customise and/or generate tools to support the languages and mappings so defined.

We have been investigating how this might be done. A ptototype tool (dubbed MMT — Meta-Modeling Tool) has been constructed. This is still under development and has its roots in OO meta-programming theory and systems such as Smalltalk, CLOS and the ObjVLisp model. It directly interprets metamodel definitions expressed in our language, including a definition of the metamodeling language itself, on which it is bootstrapped. Only a tiny kernel of the tool is hardcoded; a semantics for this kernel is defined in [6].

It is possible to input and check metamodel definitions in the tool. This includes some support for instantiation of templates and composition of packages through package extension. For the tool to directly interpret a metamodel defintion, it is necessary to provide that definition in an 'executable' form. This can make the definition less readable, and we are working on defining templates which combine both a descriptive and executable definition of common metamodelling patterns. The tool is itself bootstrapped using a metamodel definition of the metamodelling language being employed. This includes a definition of concrete syntax, so the user interface can be configured by altering the metamodel definition.

This characterises one approach to tooling. Another approach would be to take the MOF route, and provide tools that generate other tools. MOF is now supported by tools that, for example, generate working repositories for models in a language from metamodel definitions of that language [12].

One possibility for further work in this area would be to generate editors and viewers from definitions which include concrete syntax as well as abstract syntax. Another, to generate tools that can exploit the semantics definition. One kind of tool we have in mind, is a tool that exploits the definition of semantics: the mapping between abstract syntax and semantics domain. We imagine a tool that allows examples and counter examples to be created and checked against a model, and that provides assistance in constructing a model from a body of examples and counter-examples. Such a tool would be useful, for example, in validating a model with a customer, or constructing a model from customer requirements, which are often expressed in terms of examples and counter examples, such as use case scenarios.

6. CONCLUSIONS

This paper has described an approach to language design and definition called *aspect-oriented metamodelling*. This uses mechanisms for aspect-oriented modelling, namely package templates and package extension, for metamodelling. We have shown how these mechanisms can be used to define a metamodel architecture that supports the definition of language families, and illustrated this with fragments of the 2U UML submission, which is applying the approach in response to the UML 2.0 RFP from the OMG. This demonstrates that it is possible to capture commonly-occurring aspects of language definition as templates, which can then be used to stamp out significant parts of the definition of UML, both syntax and semantics, as language units that can then be combined to form different languages.

Although the work with UML looks promising, there are still questions to be answered with respect to this approach to metamodelling. One concerns the granularity of language unit required to allow languages to be constructed just by composing different units. It is hard to determine the optimum without building a number of languages in the family. One future work item is to apply the approach to defining a number of the UML profiles currently being ratified by the OMG [14]. This should establish a relatively stable set of language units for UML. Another concerns the applicability of templates: how applicable is the existing set of language definition templates we have defined? This can be determined by trying to use the templates to define other languages which are not part of the UML family. Some of the languages for which XML is being used to define might be interesting candidates, including XML itself. A third question concerns the template mechanism itself. In Section 4.1 we mentioned the need to develop families of templates dealing each dealing with a particular aspect, such as generalisation, in the context of language design. The development of such families could be facilitated by the inclusion of conditional statements in a template, which could be used to control which parts of a template are instantiated. The conditions could be controlled (though not exclusively) by boolean switches passed as argument to the template. This would allow a template family to be captured as a single template.

The prospect of tools that can interpret metamodel definitions is exciting. Effort so far has focussed on the development of tools which ignore templates once the result of instantiating the template has been obtained. This fails to exploit the fact that a particular template has been used. For example, we have been developing templates for expressing mappings between models, based on patterns for modelling relationships [2]. We suspect that if mappings are modelled as instantiations of this template, then it will be much easier to generate tools that implement those mappings to perform useful tasks, such as tracking consistency between models, generating one model from another, and so on. Solving this problem, could help with the generation of tools to support the concrete/abstract syntax and semantics mappings, such as those suggested in Section 5.

ACKNOWLEDGEMENTS

Many of the ideas described herein were developed in conjunction with and support of a number of collaborators. Specific thanks go to Steve Cook, who facilitated the early development of the ideas in a collaboration with IBM [1], and members of the 2U UML2 submission team [20]. Stuart Kent is partially supported by a Royal Society Industry Fellowship grant. Tony Clark and Andy Evans are partially supported by Tata Consultancy Services and Andy Evans is partially supported by BAE Systems.

REFERENCES

[1] Clark A., Evans A., Kent S., Brodsky S., and Cook S., *A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach*, Available from www.puml.org, September 2000.

[2] D. Akehurst and S. Kent, *A relational approach to defining transformations in a metamodel*, Submitted to UML'02, March 2002.

[3] J. M. Alvarez, A. Clark, A. Evans, and P. Sammut, *An action semantics for MML*, Proceedings of The Fourth International Conference on the Unified Modeling Language (UML'2001) (C. Kobryn and M. Gogolla, eds.), LNCS, Springer, 2000.

[4] OMG Analysis and Design Task Force, *UML 2.0 requests for proposals*, OMG document number ptc/02-01-23, available from www.omg.org/techprocess/meetings/schedule/, 2001.

[5] ———, *MOF 2.0 requests for proposals*, OMG document number ptc/02-01-23, available from http://www.omg.org/techprocess/meetings/schedule/, 2002.

[6] A. Clark, A. Evans, and S. Kent, *The meta-modeling language calculus: Foundation semantics for uml*, Proceedings of ETAPS 01 FASE Conference, LNCS, Springer, April 2001.

[7] ———, *Engineering modelling languages: A precise meta-modelling approach*, Proceedings of ETAPS 02 FASE Conference, LNCS, Springer, April 2002.

[8] ———, *Package extension*, Submitted to UML'02, March 2002.

[9] S. Clarke, *Composition of object-oriented software design models*, Ph.D. thesis, Dublin City University, January 2001.

[10] ———, *Extending standard uml with model composition semantics*, Science of Computer Programming (2002), To appear.

[11] D. D'Souza and A. Wills, *Objects, components and frameworks with UML: The Catalysis approach*, Addison-Wesley, 1998.

[12] DSTC, *dMOF: An OMG Meta Object Facility implementation*, available from

http://www.dstc.edu.au/Products/CORBA/MOF/, 2002.

[13] Andy Evans and Stuart Kent, *Core meta-modelling semantics of UML: The pUML approach*, UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings (Robert France and Bernhard Rumpe, eds.), LNCS, vol. 1723, Springer, 1999, pp. 140–155.

[14] Object Management Group, *Catalog of OMG modeling specifications*, www.omg.org/technology/documents/modeling_spec_catalog.htm.

[15] ———, *The Meta Object Facility (MOF) version 1.3.1*, OMG document number formal/2001-11-02, available from http://www.omg.org/, 2002.

[16] ———, *Uml profile for enterprise application integration EAI (final adopted specification)*, OMG document number ptc/02-02-02, available from [14], 2002.

[17] ———, *Uml profile for enterprise distributed object computing EDOC (final adopted specification)*, OMG document number ptc/02-02-05, available from [14], 2002.

[18] S. Kent, *Model driven engineering*, Proceedings of Third International Conference on Integrated Formal Methods, May 2002, Invited paper.

[19] OMG Architecture Board ORMSC, *Model driven architecture (MDA)*, OMG document number ormsc/2001-07-01, available from www.omg.org, July 2001.

[20] 2U Submitters, *Submission to UML 2.0 RFP's*, available from http://www.2uworks.org.

[21] P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton, *N degrees of separation: Multidimensional separation of concerns*, Proceedings of the 21st International Conference on Software Engineering (ICSE'99), May 1999, pp. 107–119.