Initial Submission to OMG RFP's: ad/00-09-01 (UML 2.0 Infrastructure) ad/00-09-03 (UML 2.0 OCL)

20 August 2001

Submitted by

Data Access
Project Technology[1]
Kinetium
Softlab
Siemens

In association with

Dr A. Clark, pUML group & Kings College, London, UK
Dr A. Evans, pUML group & University of York, UK
Dr S. Kent, pUML group & University of Kent, UK

Supported by

University of Kent
Kings College London
University of York

1. Project Technology is an Infrastructure submittor only

Contents

Preface

This submission is a response to the Object Management Group’s Request For Proposals ad/00-09-01 (UML 2.0 Infrastructure) and ad/00-09-03 (UML 2.0 OCL). An updated version of this document will be submitted for the superstructure RFP. This Preface summarizes the relationship of this submission to the RFPs. The main body of the document describes the technical submission itself.

0.1. Submission Summary

Submission contact point

Stephen J. Mellor: steve@projtech.com

Desmond D'Souza: desmond.dsouza@kinetium.com

pUML:

Tony Clark: anclark@dcs.kcl.ac.uk

Andy Evans: andye@cs.york.ac.uk

Stuart Kent: s.j.h.kent@ukc.ac.uk

Guide to material in the submission

See Chapter 1: Introduction.

Overall design rationale

See Chapter 1: Introduction, Chapter 2: Context, and the Resolution of RFP Issues in this Preface.

Statement of proof of concept

The approach outlined in this submission has been implemented in a prototype tool (MMT) that creates instances of objects in a semantics domain and verifies that the correct instances of metamodel concepts then exist. Many of the models defined in this submission have been automatically checked using MMT and are known to be well defined and consistent.

0.2. Resolution of Infrastructure RFP Requirements

The infrastructure RFP requirements are addressed in detail below.

0.2.1 General Requirements

§ Proposals shall enforce a clear separation of concerns between the specification of the metamodel semantics and notation, including precise bi-directional mappings between them.

The initial submission clearly separates metamodel semantics and notation. The mapping between them is bidirectional and expressed using precisely formulated associations in the metamodel.

§ *Proposals shall minimize the impact on users of the current UML 1.x, XMI 1.x and MOF 1.x specifications, and will provide a precise mapping between the current UML 1.x and the UML 2.0 metamodels. Proposals shall ensure that there is a well-defined upgrade path from the XMI DTD for UML 1.x to the XMI DTD for UML 2.0. Wherever changes have adversely impacted backward compatibility with previous specifications, submissions shall provide rationales and change summaries along with their precise mappings.*

The initial submission minimizes the impact of the current UML 1.x, XMI 1.x and MOF 1.x specifications.

The final submission will provide a precise mapping between the current UML 1.x and the UML 2.0 metamodels. As a consequence of the precise mapping between UML 1.x and UML 2.0, the final submission will ensure that there is a well-defined upgrade path from the XMI DTD for UML 1.x to the XMI DTD for UML 2.0.

The final submission will list changes that have adversely impacted backward compatibility with previous specifications and rationales and change summaries for these changes along with the precise mappings.

Proposals shall identify language elements to be retired from the language for reasons such as being vague, gratuitous, too specific, or not used.

The final submission will identify language elements to be retired from the language for reasons such as being vague, gratuitous, too specific, or not used.

Proposals shall specify an XMI DTD for the UML metamodel.

The final submission will specify an XMI DTD for the UML metamodel. The initial submission relies on the fact that an XMI DTD can be generated from the UML 2.0 metamodel provided by the submission.

0.2.2 Architectural alignment and restructuring

§ *Proposals shall specify the UML metamodel in a manner that is strictly aligned with the MOF meta-metamodel by conformance to a 4-layer metamodel architectural pattern. Stated otherwise, every UML metamodel element must be an instance of exactly one MOF meta-metamodel element. If this architectural alignment requires that the MOF meta-metamodel also needs to be changed, then those changes (including changes to XML and IDL mappings) should be fully documented in the proposal.*

The submission takes an innovative approach to addressing this issue once and for all.

The submission provides the tools to generate both UML 2.0 and MOF from the same source. As a consequence, UML X.x and MOF will forever be mutually compatible. The approach is to define the meta-model for UML 1.x. and MOF 1.x., and then meta-model a 2-way translation mapping between these and UML 2.0. This mapping defines precisely how to move from one meta-model to another. XMI can be viewed as an XML concrete syntax for the abstract syntax of UML/MOF. Both concrete syntax and abstract syntax, and the mappings between them can be defined using a meta-model. Thus we can also define mappings from XMI into UML 1.x., MOF 1.x. and UML 2.

These mappings, combined with the mappings between UML/MOF 1.x. and UML 2 (abstract syntax) meta-models, deals with backwards compatibility of XMI. These mappings would also make clear the differences between the old and new meta-models.

Note that the above mappings would be defined as part of the final submission.

Because this submission focuses first on providing the tools to generate a UML 2.0, there is a marked dearth of abstract metaclasses defining the conceptual entities actually in UML 2.0.

The issue of whether to generate MOF-compliant models first, then abstract metaclasses for UML 2.0 or vice versa is left open in the initial submission.

The final submission will benefit from the combined wisdom of the ADTF and competing proposals.

§ *Proposals shall strive to share the same metamodel elements between the UML kernel and the MOF kernel, so that there is an isomorphic mapping between MOF meta-metamodel kernel elements and UML metamodel kernel elements.*

See above.

§ *Proposals shall restructure the UML metamodel to separate kernel language constructs from the standard elements that depend on them. The standard elements shall be restructured consistent with the requirements in 6.5.3.*

The submission separates the kernel language constructs from the standard elements that depend on them, as described in Chapter 6.

§ *Proposals shall decompose the metamodel into a package structure that supports compliance points and efficient implementation.*

See above.

§ *Proposals shall identify all semantic variation points in the metamodel.*

None.

0.2.3 Extensibility

§ *Proposals shall specify how profiles are defined.*

Profiles will be defined using two methods:

(1) Model extension. This involves using first class extension mechanisms to extend both syntax and semantics. This is required if the base language is not expressive enough to start out with (e.g. adding filmstrips or time to a semantics that does not support them). There are many examples of this approach in the submission. For example, extending classes with constraints.

(2) Translational. This involves no extensions. Instead, syntax and semantics are translated via a mapping from the new language to the old. It is useful since no new concepts are involved - everything is viewed as sugar. However, it requires the base language to be sufficiently expressive, which may not be the case.

Stereotypes will be viewed as syntactic sugar for both mechanisms. Tools may or may not implement this interpretation of stereotypes, however, if they do, they will gain the ability to check that extensions are both well-defined and consistant.

§ *Proposals shall specify a first-class extension mechanism that will allow modelers to add their own metaclasses, which will be instances of MOF meta-metaclasses. This mechanism must be compatible with profiles and consistent with the 4-layer metamodel architecture described in 6.5.2.*

A first class extension mechanism based on the use of packages and package generalisation has been precisely defined in Chapter 5.

§ *Proposals shall identify model elements whose detailed semantics preclude specialization in a profile. If proposals need to generalize these model elements, they should propose refactoring consistent with the architecture and restructuring requirements described in 6.5.2.*

Not applicable.

0.2.4 Architectural alignment and restructuring

§ *Proposals may refactor the UML metamodel to improve its structure if they can demonstrate that the refactoring will make it easier to implement, maintain or extend.*

The submission refactors the UML metamodel somewhat. The refactoring is a direct consequence of defining the semantics in terms of primitives on which the remainder of UML 2.0 is built. This layered approach is easier to implement, maintain and modify.

§ *Proposals may consider architectural alignment with other specification language standards.*

Not applicable.

0.2.5 Extensibility

§ *Proposals may support the definition of new kinds of diagrams using profiles.*

Not applicable.

0.2.6 Issues to be discussed

Proposals should provide guidelines to determine what constructs should be defined in the kernel language and what constructs should be defined in UML profiles and standard model libraries.

This issue is discussed at length in Chapter 1: Introduction and Chapter 2: Context.

Proposals should stipulate the mechanisms by which compliance to the specification will be determined, recognizing that determination of conformance is on a subset of the specification and that not all parts of a metamodel package are always needed. For example, proposals might submit XMI DTDs to test the compliance of a tool to the specification in a subset of a metamodel package.

The separation of concerns in the proposed architecture of the UML infrastructure, means that a statement of conformance can be factored by elements in that architecture. For example, one can claim to support only the static core. Or one can claim only to support the graphical syntax, or only to support a textual syntax, etc. The precise nature of the definition, makes automatic conformance checking feasible.

Proposals should discuss the impact of any changes to the UML metamodel on adopted profiles. In particular, the impact of any refactoring should be discussed.

The initial submission has no comment.

0.3. Resolution of OCL RFP Requirements

0.3.1 Mandatory requirements

Proposals shall provide a metamodel for OCL that integrates with the UML metamodel.

A meta-model for the abstract syntax of OCL, as used for writing static, invariant constraints, is defined, and this integrates with the proposed meta-model for the static core of UML. The final submission will extend this with a meta-model for those aspects of OCL used only to write dynamic constraints (pre/post conditions).

0.3.2 Optional requirements

Proposals may define a formal semantics for OCL.

A semantics is defined using a metamodelling approach.

Proposals may define changes or extensions to OCL that increase its expressive power, for example, that improve its applicability to behavioral specification or to component assembly. Such proposals shall preserve the declarative nature and side-effect-freeness of OCL expressions.

The proposed OCL metamodel is a specialisation of the metamodel that provides the foundation for the definition of a family of expression languages. This will make it easy to adapt or extend OCL for other purposes. It will also make clearer the similarities and differences between expression languages used in conjunction with UML.

Chapter 1

Introduction

This introduction highlights the key contributions of our submission, and where we think more work would be required to fully address the RFP requirements. It provides an outline for the rest of the document.

1.1. Aims and Objectives

The main aim of this submission is to address the problems defined in the UML 2.0 Infrastructure and OCL RFP's by delivering the methods and notations necessary to fully support the infrastructure of UML 2.0 and a family of OCL like constraint languages[1]. The specific objectives are as follows:

1. To propose a systematic method for the precise and complete definition of UML 2.0 in terms of modular components. A complete definition means that semantics and notation are defined to the same precision as the abstract syntax.
2. Using the method defined in (1), to define a kernel of the UML meta-model which provides a firm foundation for the UML superstructure, including the language defined in (2).
3. Using the method defined in (1), to define the meta-model of the meta-modelling language used in the method.
4. To address issues concerned with alignment and backwards compatibility with UML 1.4 and MOF 1.3.

1.2. Key contributions

- A meta-modelling approach that is able to deliver modularity and reuse of language component definitions with precision. Specifically, we extend the meta-modelling language with a notion of templates and package/template extension and composition.

  Templates allow patterns of language definition to be captured precisely and then used to stamp out language components. Templates deliver the following benefits to the meta-modeller:

  – They enforce a high degree of architectural integrity over the meta-models they are used to build – the meta-models must conform to templates! This makes meta-models much easier to comprehend. Architectural integrity is virtually impossible to maintain in a vanilla meta-modelling approach, especially for languages on the scale of UML.

1. As such, this work is a follow on from [Clark00], re-architected using templates (c.f. frameworks in Catalysis) that capture meta-modelling patterns suitable for modelling UML as a family of languages.

  - They lead to more complete meta-models. Ensuring that a meta-model is complete and consistent requires considerable skill and hard work. Templates allow this work to be reused many times over.
  - Once a set of templates has been established, constructioin of new (and comprehensive) meta-models proceeds very quickly. This allows more ground to be covered in the time available, as well as leading to a higher quality and more comprehensive result.

  Package extension and composition constructs allow language components and templates to be composed, merged and specialised. This is essential to support the notion of a language family and to allow that family to be extended.

- An approach to defining all aspects of a language, including notation and semantics, to the same level of precision. Specifically, we identify a set of templates that allow us to mimic formal language theory using a meta-modelling approach.
- A transformation-based solution to the backwards compatibility/alignment issue, that does not leave the meta-modeller shackled to UML 1.4 and MOF 1.3.

1.3. To be completed

As this is an initial submission, there are some omissions that we would expect to be included in a final submission. The key ones are:

- The meta-modelling language itself has not been completely defined in itself. Specifically, we have not yet provided a meta-model definition for templates. All other aspects of the language have been defined using the approach. A meta-model for templates will be provided in the final submission.
- A transformational approach to dealing with alignment/backwards compatibility issues has been outlined, but not fully implemented. The complete meta-model definitions of (two way) translations between UML 1.4/MOF 1.3 and UML 2.0 would be provided as part of the final submission. These translations would also provide a precise definition of the exact relationship between UML 1.4/MOF 1.3 and UML 2.0. Separate documentation of the changes would not be required, though informal explanation of the decisions made in defining the transformations would no doubt be helpful for users to make the transition from UML 1.4/MOF 1.3 to UML 2.0. The translations will also support the continued use of MOF 1.3 to provide repository services for UML 2.0 models. Of course this would not be necessary, if MOF 1.3 was revised to be an extension of the meta-modelling language used here; this would require extension with a new capability (templates and composition/merging mechanisms) and revision of its meta-model definition to confirm with the UML 2.0 architecture.
- Already in this submission, we have dealt with some aspects of superstructure as proof of concept of our approach. The final submission to provide complete meta-models for superstructure as required by the RFPs.
- Meta-model definitions of concrete notation, and their mapping to abstract syntax, have not been provided. These will be included in the final submission, with partiucular emphasis on XML and graphical syntax.

1.4. Outline

The objectives outlined in section 1.1, if achieved, deliver more than is required by the RFP. Chapter 2 describes the background and motivation for the approach used here. It discusses how the approach can both address the issue of backwards compatibility with existing meta-models and frontwards compatibility with the challenges of MDA. Chapter 3 describes our approach to meta-modelling, in particular describing the meta-modelling language employed. Chapter 4 describes a set of templates for language engineering, that are instantiated to deliver language components that can be combined in the definition of UML 2. Chapter 5 defines a core set of static modelling concepts. Chapter 6 extends these modelling concepts to deal with dynamic aspects.

1.5. Acknowledgements

The authors wish to thank Laurence Tratt (Kings College London) and Paul Sammut (University ofYork) for their invaluable help in the preparation of this document.

Chapter 2

Context

This chapter provides an overview of the submission, describing it's philosophy and overall architecture, and the problems addressed. It includes an informal explanation of how the approach relates to the 4-level architecture adopted by the OMG., describes the different aspects of a language definition and how these relate. It rehearses the motivation for using a meta-modelling approach, defines what that means, and explains how such an approach can be used to cleanly specify the different language aspects and relationships between them. It motivates the use of package composition and templates to support the definition of UML as a family of languages. Lastly, it explains how backwards compatibility can be ensured by a transformation-based approach, and discusses options for alignment with MOF in its present or future form.

2.1. Introduction

2.1.1 Pattern-based infrastructure

The class/object centric view of the 1990's, based on subclassing, is not the best way to structure models. Most interesting properties we wish to model are about the patterns of static and dynamic relationships between all granularity of objects, both when defining the UML family of modelling languages as well as when defining end-user models of some problem domain. Hence our submission focuses strongly on the package as a unit of extensible models, as well as a unit of modelling patterns.

The overall submission defines:

1. How packages can be extended and used as patterns.
2. A library of reusable patterns for defining families of related languages.
3. A static core (and examples of a dynamic core) built using the above two.

2.1.2 Main Principles

The main principles behind this submission are the following:

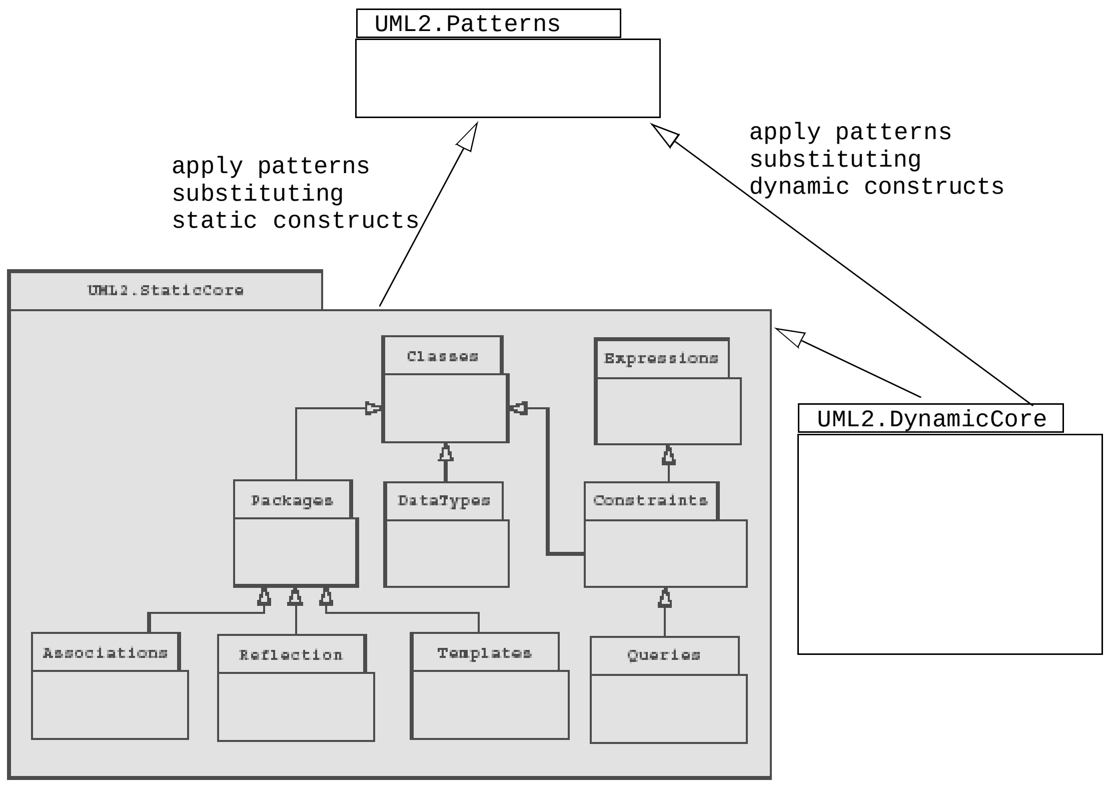
1. Extensible core: core language constructs are extended using package generalization and package templates. A new modelling construct is defined, optionally with its own concrete syntax, with clear semantics including a translation into more familiar constructs where needed. This is a powerful way to leverage patterns of arbitrary model fragments to define both modelling languages and end-user models.
2. Minimal core and layered definitions: we use as few constructs as possible. Additional constructs (associations, qualifiers, n-ary associations) are defined consist-

ently in terms of this core constructs using the extensibility features. Our goal is that the UML 2.0 superstructure can use this to properly unify objects and values, attributes with associations and qualifiers, states and attributes, template packages and template classes/ collaborations, activities and actions and state transitions, etc.

3. No duplication: we aim to define any class, attribute, constraint, or recurrent pattern across any of these, exactly once. Language definitions should re-use these definitions at whatever granularity is suitable.
4. Effectively factored and easily re-factored: the semantics of both package generalization and package templates make it easy to re-factor parts of a package into other packages (including template packages) with no impact on clients of that package. The submission itself has factored the definitions into small, easily understandable parts, themselves built using a library of patterns.
5. Incremental definition and composition: classes, packages, and even patterns can be incrementally defined. The proposal specifies clearly the meaning of such incremental definition in the semantic domain by defining multiple views of the “instances” of a class or a package. This will form the basis for the UML 2.0 superstructure to uniformly support incremental definition and composition of state charts, sequence and collaboration diagrams, etc.
6. Patterns for consistent architecture: We use the facilities of package templates to define major architectural patterns that structure the entire meta-model for the infrastructure (and the corresponding superstructure, OCL, profiles, etc.).

2.1.3 The Meta-Model - Semantics and Composition at Core

The Static Core model is described in detail in Chapter 5; the Dynamic Core is introduced in Chapter 6 (and will be expanded in the superstructure submission). However, the constructs used in these core language definitions themselves make use of architectural patterns for all major language definition tasks: structuring syntax, semantics, language structure, naming, etc. Hence, this core is itself built using a library of model templates that is part of this submission. These templates are equally applicable to the definition of any new profiles, and to end-user models.



The definitions are based on a consistent structure of language syntax and semantics, and an underlying approach of incremental definitions that can be readily composed. The StaticCore, in essence, describes:

- The semantic domain: states described as snapshots of interconnected objects that we wish to model.
- The abstract syntax domain: modelling constructs to define static invariants that distinguish valid snapshots from invalid ones, including a consistent expression-based language for static constraints.
- Notations: concrete syntax that is used to express the abstract syntax.
- Multiple views: ways to incrementally define static models syntactically in terms of other models (e.g. one class extending another, or one package extending another), with corresponding definitions of how the snapshots in the semantic domain can be viewed from these different partial and composed syntactic constructs.

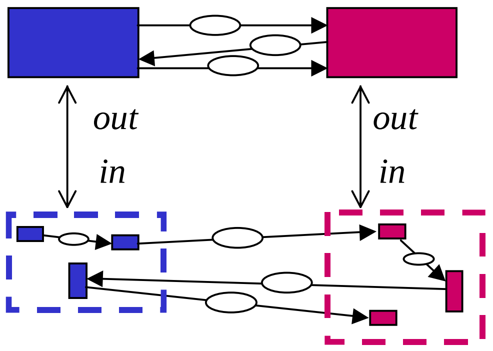
The Dynamic Core, briefly introduced in this submission, describes:

- The semantic domain: histories of executions representing partial orderings of action occurrences, each with at least its before and after states.
- The abstract syntax domain: modelling constructs to define dynamic invariants that distinguish valid histories from invalid ones; including an expression-based action language for dynamic constraints.
- Notations: concrete syntax used to express the abstract syntax.

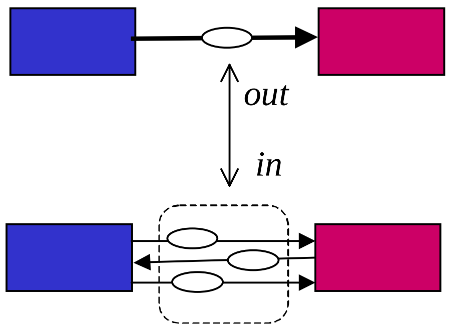
- Multiple views: ways to incrementally define dynamic models syntactically in terms of other models (e.g. one state machine specializing another, or two collaboration models being composed with some objects playing roles in both collaborations), with definitions of how the corresponding snapshots and histories of executions correspond to these different partial and composed syntactic constructs.

There is clearly a large-grained structural similarity across the static and dynamic cores. That architectural pattern is itself defined once, as a pattern in a template package, and used across many parts of the language definition.

In addition, relationships across models can be as important as an individual model. We believe that "Refinement" is an essential construct for describing large-scale models, and for the separations between platform-independent and platform-dependent models required by the MDA. In particular, we are interested in refinement of both objects and actions, as shown below:



(a) Zooming in/out – objects



(b) Zooming in/out – interactions

- *"Zooming" out* of a model showing a complex network of objects, and have a simplified model with fewer large-grained objects and attributes; and correspondingly zoom *in* to see those details.
- *"Zooming" out* of a model of a detailed interaction protocol to get a simplified model with a single more abstract action with the same overall effect; and correspondingly zoom *in* to see those detailed interactions.

There will be different kinds of refinement needed, some corresponding to automated translations or model/code generation; others having at best some helpful patterns that are selected manually by a modeller. Refinement should definitely be supported by the UML superstructure; it is not clear whether MOF or the UML infrastructure will incorporate support for refinement. In the final submission, we will include some underlying patterns needed for refinement in Chapter 4, based on Declarative Generalizable Containers (Section 4.9).

2.2. Support for a Family of Languages

UML 2.0 and MOF will form the foundation for a family of different dialects of modelling languages. It is essential that these languages share a consistent language architecture, with clear semantics, to enable interoperability. For example:

- The OMG already has multiple, independent, and overlapping constructs in the area of defining connection points on components separately from the connections between those points. The real-time UML profile defines constructs such as *capsules*. The EDOC profile defines *ports*, *connectors* and *protocols*. The Corba component model uses *facets*, *receptacles*.

- Even within the UML, there are multiple ways of defining dynamic behaviors, with poorly defined relation between their semantics: preconditions and postconditions on operations, state transitions on state charts, activities and object flows in activity diagrams, and sequence and collaboration diagrams.

Architectural consistency across a large family of complex languages is not easy. Our submission enables this consistency by starting with precise definition of any language, and sharing patterns of partial language definitions across languages.

Based on this submission, a "profile", or language dialect, is simply another package defined using sharing of the relevant partial language packages and patterns.

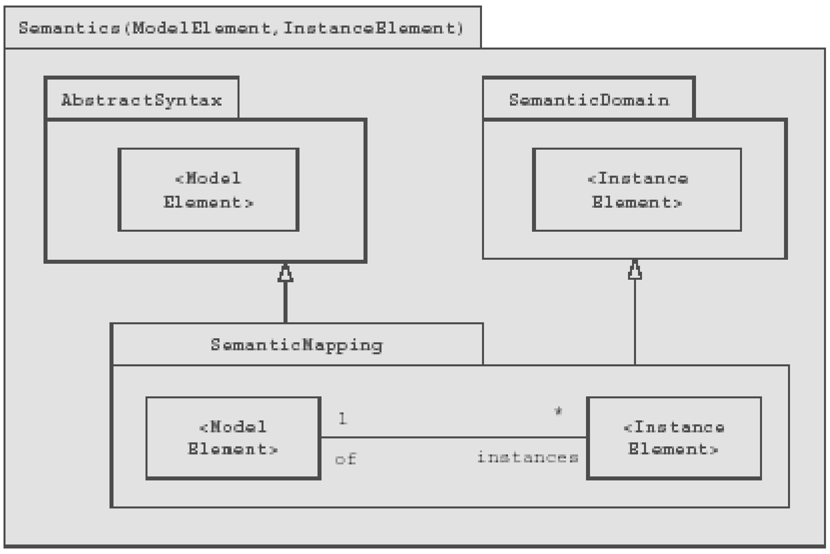
2.2.1 Language Definition

A modelling language has a concrete syntax or notation, an abstract syntax with its well-formedness rules, and semantics that give meaning to the syntax[1].

- Abstract syntax domain: defines the structure of the modelling constructs and the rules for that abstract syntax to be well-formed in a given model e.g. a class has attributes and static invariants; static invariants can refer only to attributes of **self**.
- Semantics domain: defines the elements that are being described by the abstract syntax e.g. objects have slots, and at any point in time a snapshot of object defines the values of those slots as references to other objects.
- Semantic mapping: gives the "meaning" of a model in the abstract syntax, either by translating it into a more basic abstract syntax that has a well-defined meaning, or by relating the syntax to constraints on things in the semantic domain e.g. a static invariant on a class must hold true of any snapshot of an instance of that class.
- Notation: a concrete (graphical or textual) syntax, with its rules of well-formedness. e.g. a static invariant is written **context** class **inv:** boolean_expression
- Syntax mappings: rules that map concrete syntax to abstract syntax, and back.

1. These have often been mis-characterized in prior UML and MOF work.

The submission defines patterns for these aspects of languages definition. An example is shown below, detailed in Section 4.7.



2.2.2 Architectural Consistency across Language Definitions

In addition to the macro pattern of concrete syntax, abstract syntax, and semantics, language dialects have to repeatedly address recurring language definition issues: naming and namespaces, the tree "container" structure of abstract syntax elements, variables and constants in expressions, run-time semantics of expressions, etc.

We use package generalization and package templates to define these language elements in a way which addresses numerous shortcomings of the subclass-based approach prevalent in UML 1.x and MOF 1.x. Chapter 4 includes several templates, some of which are listed below:

Basic Templates: templates used as a foundation for most language constructs.

- Container: one element is contained within another.
- NameSpace: a naming context binds names to a set of elements.
- Semantics: separation and mapping between abstract syntax and semantic domain.

Extensions: capture "extension" relationships between modelling elements.

- Generalisable Container: contents of container are derived from its parents.
- Declarative Generalisable Container: container obliged to extend parents' contents.

Semantics: templates that capture meaning relationships between modelling elements.

- Container Semantics: how container and contents constrain their semantic domain.
- Generalisable Semantics: how generalization constrains its semantic domain.

Constraints: templates for the definition of constraint languages, such as OCL.

- Expression Semantics: how syntactic expressions constrain semantic calculations.

- Binding: binding syntactic elements (e.g. variables) to semantic ones (instances).

Mappings: express transformations across modelling elements and across languages.

- Container Mapping: maps one kind of container to another.

Graphs: graphs, trees, etc. which are very useful to define languages.

- Graph: a nodes-and-edges description of a relation.
- Tree: a graph with specific constraints on its structure.

Reflection: relate model elements to their reified representation as metaobjects, addressing the core issues of the OMG's 4-level meta-modelling architecture and MOF.

- Object Reflection: a container model element has an object + slot representation.

Abstract Class Hierarchy: these patterns painlessly construct an abstract class hierarchy similar to that used in UML 1.x.

2.2.3 N-level Meta-modelling

The OMG standards have long described the benefits of "strict" 4-level meta-modeling. However, the UML and MOF have never satisfied the few clear definitions of what strict N-level meta-modelling means. Moreover, "strict" 4-layer rules make some simple things difficult, even impossible:

- Modelling concepts that cross layers e.g. the class-instance relationship.
- Having one definition of "pattern" used in MOF, UML, Profiles, and user models.
- Specific patterns, such as Namespace, Tree, Graph, DAG, … used at all levels.

This submission provides a precise and clear basis for representing a model element in more than one way e.g. reified as an object in the semantic domain, or as an element in the abstract syntax domain. These approaches to reflection are represented as patterns. Using these, we can define the meta-model for UML and MOF, and then model a 2-way translation mapping between these and UML 2.0. Thanks to our clean handling of abstract syntax, concrete syntax, and semantics, we can view XMI as a concrete syntax for the abstract syntax of UML/MOF.

Based on this, we can generate both UML 2.0 and MOF from the same source, so UML and MOF 2.0 and beyond could forever be mutually compatible. In fact, this submission enables MOF to **be** the UML 2.0 core, without need for any translation or isomorphic mapping between them.

2.2.4 Relationships and Translations across Languages

The template facility in this submission can be used to define relations and transformations across languages, at the level of abstract syntax, concrete syntax, or semantic domain. We illustrate this with some simple examples.

- Templates can parameterize any model by designating elements as placeholders. Thus, `<Container>` and `<Contained>` could be placeholders in a template that described the structure and OCL constraints for containment.
- Templates can parameterize identifiers (names) used in a model. For example, given an attribute name `<attr>`, a template can define `set_<attr>` and `get_<attr>` methods.
- UML 1.x constructs such as N-ary associations and qualified associations can be easily represented as templates, with a translation into, for example, simple attributes and queries with parameters.
- Since we make a complete separation of abstract syntax, concrete syntax, and semantic domains, languages such as XMI and IDL could be defined as alternate syntaxes to UML 2.0[1]; OCL could be used with a far more friendly syntax.
- The mapping from MOF to XMI or IDL could be easily represented as a template, even through to concrete IDL syntax such as:

  ```
  module <PackageName> {
  typedef
      sequence < <BuiltInType> > <BuiltInTypeName> <CollectionKind>;
  ... }
  ```

1. In the case of IDL, this might require the definition of some new `#pragma`’s in IDL.

Two examples from the MOF 1.3 specification are included here. Their representation as a template will be included in the final submission.

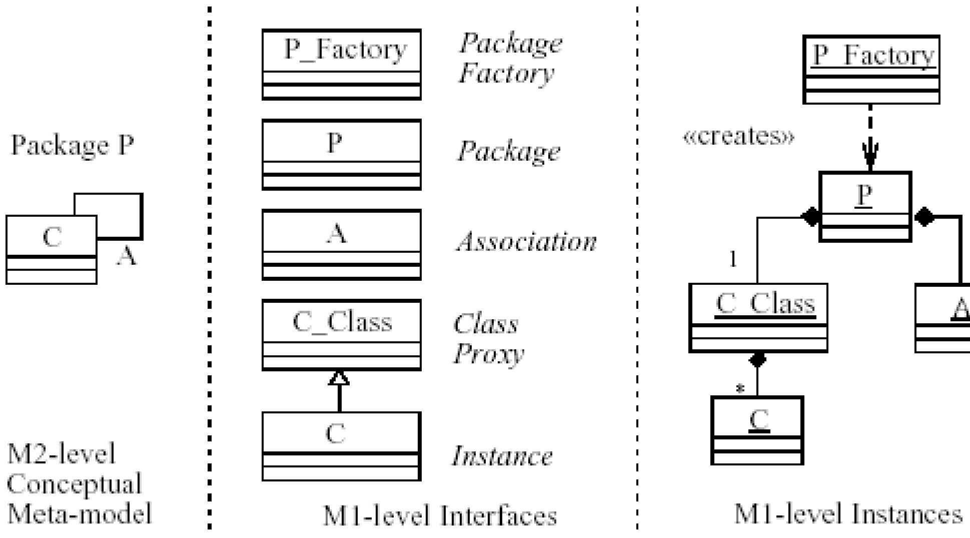


Figure 5-1 Relationships between M1 and M2 level

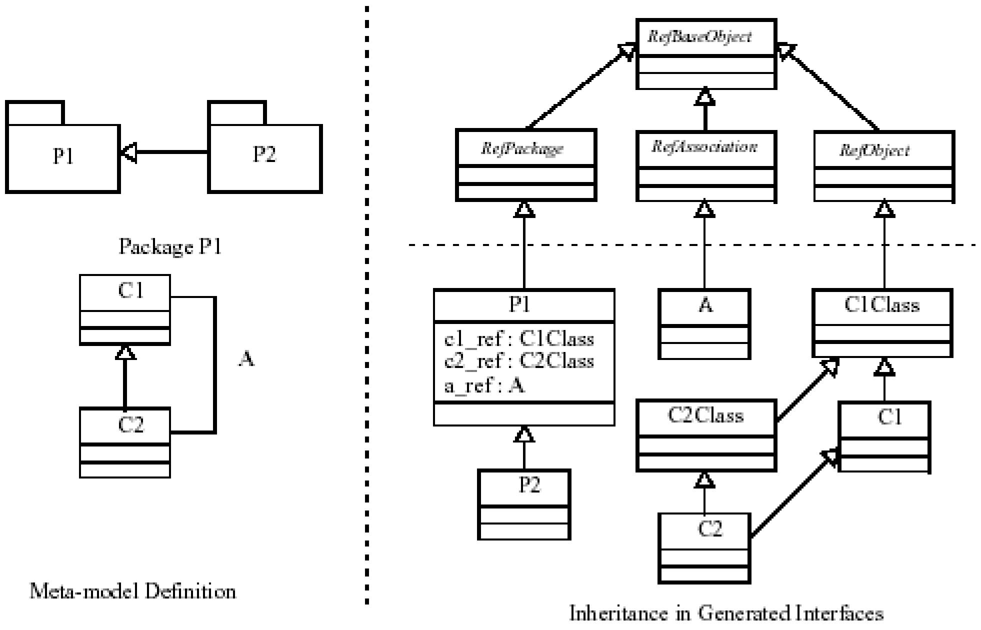


Figure 5-2 Generated IDL Inheritance Patterns

2.2.5 Improved Model Interchange

In the presence of a family of modeling languages, model interchange takes on some new nuances. For example, suppose you have a model in `Profile-A`, which includes states and state-charts. In pseudo-text form, your model might define class `Person` with a state `child` and disjoint substates `infant, toddler`:

```
class Person {  States: child substates (infant, toddler) }
```

Suppose someone else has a tool which does not understand `Profile-A` (states), but does understand attributes and static invariants (`Profile-B`). The approach in this submission enables model interchange to transfer a profile-based model as:

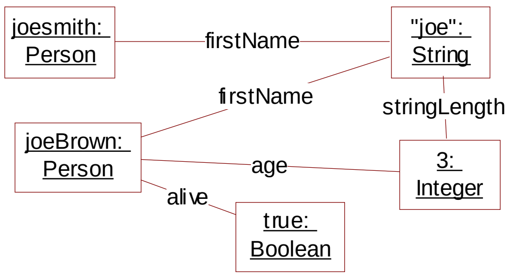
```
class Person {
   attributes  child, infant, toddler: Boolean
   static invariants:
         child = infant OR toddler
         EXCLUSIVE (infant, toddler)
}
```

In the presence of multiple profiles, with unresolved overlapping areas across profiles and across the domains the models address, and with tools picking and choosing what they implement, such smart interchange will be a blessing.

2.3. Unified Concepts

This submission (with its corresponding OCL and UML/superstructure submissions) unifies several disparate concepts that are not cleanly related in UML 1.x, including:

- Template packages: unifies the current concepts of packages and renaming, collaboration patterns, and parameterized classes. In addition, it replaces many inappropriate uses of abstract classes in UML and MOF 1.x, and supports patterns of packages, patterns of refinement, etc.
- Objects and values: these are unified in the current submission, so the same



  approach can be used to define OCL collections, integers, strings, and enumeration types, as is used to define other UML classes, user-defined classes, or user-defined “value” types. Concepts such as *tuples* and *n-ary associations* become easy to define as patterns. A particular platform’s call-by-value rules are deferred to the appropriate language dialect or “profile”, as would be required by MDA.
- Actions: this submission unifies and makes consistent the current UML notion of action, state transition, activity, and collaboration.
- Language semantics and syntax: many elements of UML have their abstract syntax and well-formedness rules of the abstract syntax defined; few have clearly defined semantics. This submission unifies and make consistent the approach to defining and relating concrete notation, abstract syntax, and semantics.
- Inter-language mappings: this submission provides a systematic way to define mappings that cross languages, at the level of concrete syntax, abstract syntax, or semantics domains.

- UML 1.x concepts of system, subsystem, model are unified using packages and refinement of the basic concepts of object and action.
- UML 1.x concepts of role, object, classifier role, etc. are unified using the clear separation of objects in the semantic domain, from variables and bindings.
- Containment and ownership have been separated from namespace. The definition of a class should be considered independently from the name by which you refer to that class definition, hence name bindings should be separated from the model elements being named.

Chapter 3

Meta-Modelling Approach

This chapter introduces our approach to meta-modelling, in particular the language being used. The language is the expected subset of UML, extended to include a rich notion of package templates and package inheritance, both crucial to defining families of languages.

3.1. Introduction

The meta-modelling approach is dictated by the meta-modelling language chosen and the way in which that language is deployed.

3.1.1 The Meta-modelling language

Unsurprisingly, our approach uses a meta-modelling language based on well-understood object modelling principles which are at the heart of UML and MOF. There are four main components of the language:

Static basics. Includes the usual constructs (classes, attributes, associations, query-operations) for capturing the static structure of a meta-model.

Packages. For organising meta-models into manageable chunks. Includes facilities to handle package composition and merging.

A constraint language (OCL). For expressing well-formedness constraints on the structures admitted by the meta-model.

Templates. Allows meta-modelling patterns to be encoded in a precise and effective way. Includes facilities to handle template composition and merging.

These four components are described in the main body of the chapter.

The static core, hence this meta-modelling language, is formally defined in Chapter 5.[1]

Meta-models will be presented using diagrams and text. Text is used only to write out the OCL constraints and write out the definition of supplementary operations.

1. It should be noted that templates have not been formally defined for this initial submission, although there is a placeholder in Chapter 5. They will be formally defined in the final submission. Templates have been implemented in a prototype tool as proof of concept.

3.1.2 Deployment

Our deployment of the meta-modelling language is considerably affected by the inclusion of templates. Templates deliver three key benefits for the meta-modeller:

- They allow commonality between language components to be captured in one place, which leads to a high degree of architectural integrity in the definition of languages constructed through the application of templates;
- They take much of the slog out of producing meta-models, in particular well-formedness constraints, which allows more complete definitions to be constructed more quickly;
- They avoid the proliferation of abstract meta-classes, which are an ineffective mechanism (we argue) for supporting reuse and extensibility.

Thus, when deploying the meta-modelling language, one should strive to isolate templates that capture useful patterns of language definition. The templates used in this submission are described in Chapter 5.

One should also strive to deliver complete meta-models, especially for a definition of a standard language. The component of a meta-model that is most often omitted is the well-formedness constraints. There are three good reasons for putting the effort into defining well-formedness constraints:

- Writing such constraints helps to uncover ambiguities and subtleties in the language that would otherwise be missed, and often this can precipitate changes to other aspects of the definition, such as class diagrams.
- Constraints capture important aspects of the language definition that can not be captured through structure diagrams alone. This becomes particularly noticeable in complex aspects such as semantics and even the syntax of some sophisticated language constructs.
- If the constraints are written precisely enough they can be directly processed by tools that have a constraint capability. This can be used, for example, to check whether a model satisfies the well-formedness conditions defined in the abstract syntax component of a meta-model; or whether an instance of a model satisfies a model, as defined by the semantics captured by the constraints in the meta-model. Indeed, if the meta-model language itself is cast as a meta-model, then meta-models (which will be models of this cast) can be checked to ensure that they are well-formed meta-models using the proposed tool.

This submission has been careful to define constraints as precisely and completely as possible. Templates have been of considerable benefit here, both by delivering many constraints automatically through their instantiation, and by helping us to identify where constraints need to be added by language components which instantiate a template.

All the meta-models in this submission have been checked by a prototype tool that understands and checks constraints. The same tool also supports the expansion of templates, which has helped to ensure that the template mechanisms employed in this submission give the expected results.

3.1.3 Organisation of chapter

The remainder of this chapter focuses on providing a description of the meta-modelling language itself, dealing with each main component in sequence. More attention is paid to the novel aspects of the language (templates & package/template specialisation/ composition/merging). Subsequent chapters deploy that language to deliver the UML 2 infrastructure.

3.2. Classes

Classes and associations (next section) are used to describe structural aspects of a meta-model. In the meta-modelling language used, classes are restricted to contain

- Attributes
  - take the form `x:T,` where `T` may take the form `X`, `Set(X)`, `Seq(X)`and `X` may be a class or a basic data type – `Integer`, `Boolean` or `String`.
- Query operations
  - take the form `q(x1:T1,...,xn:Tn):T,` where `Ti` and `T` take the same form as `T` above.

Visibility distinctions on attributes or query operations are not needed.

Classes may only be defined in the context of a package.

The usual UML graphical notation for classes will be used.

3.3. Binary associations

Binary associations have two ends. An association end is identified with a source class `C` and target class `D`, a cardinality constraint `m..n` and a label `s`.

Associations which are only navigable in one direction are also allowed, in which case their semantics is given by the semantics of the association end which can be navigated.

A fuller treatment of UML associations, including n-ary associations, is given as part of Chapter 5. The translational semantics given here is consistent with that definition.

Associations may only be defined in the context of a package.

The usual UML graphical notation for binary associations will be used.

3.4. Packages

Packages may contain classes, associations and other packages. The ends of an association in a package must be sourced on classes contained in that package. This avoids associations delivering new attributes on classes in distant packages. In practice, this means that only one-way associations may refer to classes of a different package (with the source being in the package owning the association). Of course, attributes and queries can refer to classes of different packages.

A package generalisation mechanism is also supported. This is illustrated by the example in Figure 1, which shows a package with multiple (two) parents.

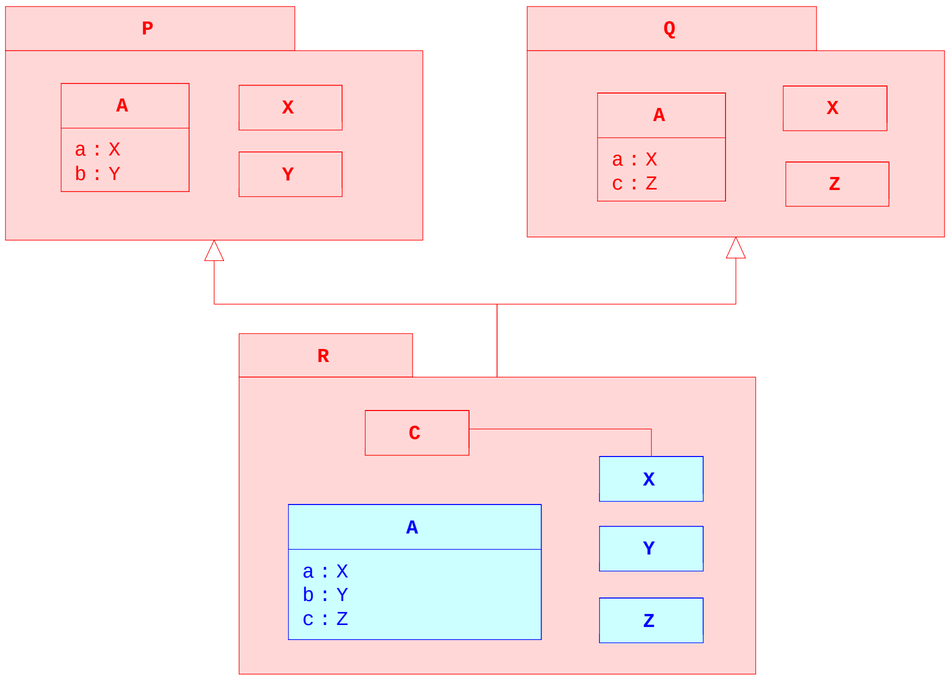


Figure 1. Package generalisation example

The UML graphical notation for packages is used. The UML class generalisation arrow is used to show a generalisation relationship between packages. Anything shown in blue in the child is generated from the parent. Although it is not strictly necessary to show these elements, it can be extremely helpful to the modeller to see them displayed. In some cases it is essential to show generated elements as they interact with new elements introduced in the package (so, for example, class *R.C* interacts with *R.X*).

The basic rule is that the child gets everything from each parent, where model elements of the same kind (class, attribute, constraint, etc.) and which are "indistinguishable" get merged. What it means for a model element to be distinguishable from another element, and how a model element is merged with another, depends on the kind of model element. For example, two classes will be deemed indistinguishable if they have the same name, and the merge of two classes will require the list of attributes of each to be merged, merging attributes as appropriate. Two attributes are also deemed indistinguishable if they have the same name, and their merge replaces the type with one that is a subtype of the types of the attributes being merged.[1]

1. This is the not the only possibility for merging attributes, but it is the one adopted here.

The child may also specialise any model element from any parent in a conformant way, where the definition of conformance depends on the kind of model element. So, for example, the cardinalities on association ends may be strengthened.

Clearly this merging idea can be extended to class generalisation, and this is assumed for the meta-modelling approach used here.

In principle, one may wish to prevent a merger taking place or, indeed, force a merge to happen. This could be achieved through a renaming mechanism. However, we have found no need to do this (or at least nothing that can not be easily circumvented), when templates are deployed. The need for renaming will be reconsidered in the final submission.

3.5. Constraint Language

OCL as defined in UML is used to write invariant constraints using our meta-modelling approach. There is a single caveat: constraints are given names, where the name is a single text string on a separate line below the context clause (see constraints in Section 3.3 for an example).

OCL expressions are also used to provide the definition of queries, using the syntax

```
class C
q(x1:T1,...,xn:Tn):T
  OCLexp
```

where *OCLexp* is an expression of type *T*, which may depend on the arguments of *q* and attributes of *C*.

3.6. Package Templates

Package templates are used to provide chunks of (meta-)model with substitutable parts. They are very effective in encoding modelling patterns. Figure 2 gives a simple example of a template.

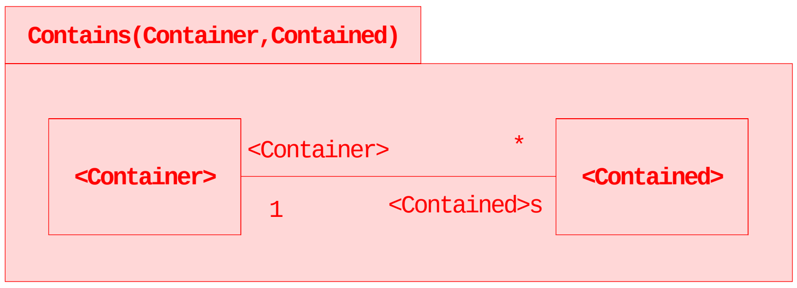


Figure 2. Simple example of template

The parameters to the template are just strings, representing names of things. The names of these parameters are declared in brackets after the package name. The arguments may be quoted anywhere a string is required. Quoting is shown by angled brackets.[1]

Templates may be instantiated to generate standard packages or other templates. The former is illustrated by Figure 3.

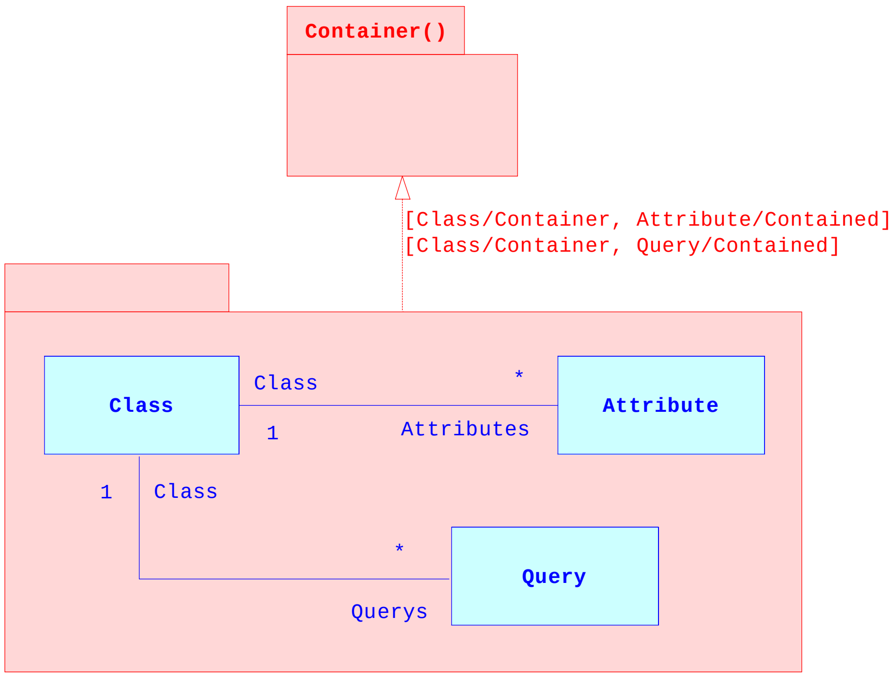

Container()
[Class/Container, Attribute/Contained]
[Class/Container, Query/Contained]
Class
Class
*
Attribute
1
Attributes
1
Class
*
Query
Querys

Figure 3. Template instantiation

Template instantiation is shown by a dashed generalisation arrow, which is annotated by substitutions for the parameters of the template. More than one substitution may be given, in which case the template is instantiated for each substitution and the results merged.

So, in the example, the template is instantiated twice. In both cases, *`Container`* is substituted for *`Class`*, but in the first case *`Contained`* is substituted for *`Attribute`* and, in the second, *`Container`* is substituted for *`Query`*. By merging rules, only one copy of *`Class`* appears in the child package.

The syntax of a substitution takes the form *`[A1/X1, ..., An/Xn]`* which means, for a template with parameters *`X1, ..., Xn`* that *`A1`* is substituted for *`X1`* and so on. A parameter may only be substituted once for any single template instantiation, but there may be more than one instantiation of the same template into the same package.

1. Rather than just allowing quoting, one can instead introduce the concept of string expression, and use a small string expression language involving the parameters to generate names. For this submission, we have found concatenation to be sufficient.

A package resulting from an instantiation of a template may specialise, in a conformant way, the elements generated from the template through the instantiation. This is analogous to the package generalisation case.

Of course templates may contain more that just classes and associations. For example, they may also contain constraints, query definitions and so on. Consider the template defined by Figure 4, and query definitions that follow.

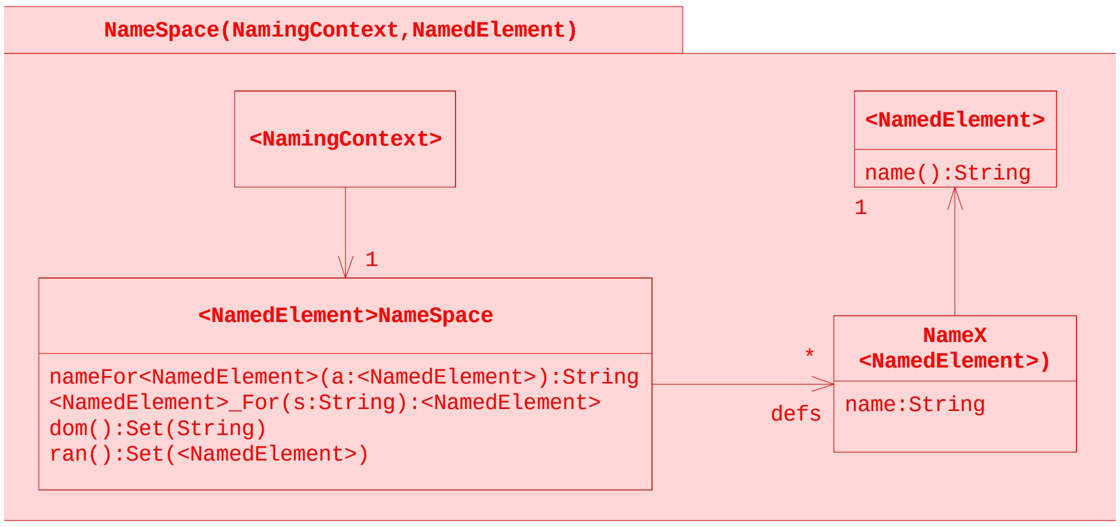

NameSpace(NamingContext,NamedElement)
<NamingContext>
1
<NamedElement>NameSpace
nameFor<NamedElement>(a:<NamedElement>):String
<NamedElement>_For(s:String):<NamedElement>
dom():Set(String)
ran():Set(<NamedElement>)
<NamedElement>
name():String
1
*
defs
NameX
<NamedElement>)
name:String

Figure 4. Template with method definitions

nameFor<NamedElement>(): Returns a name for a NamedElement in a NameSpace.

```
context <NamedElement>NameSpace
  nameFor<NamedElement>(a:<NamedElement>):String
    self.defs->select(pair | pair.name = a).selectElement().name
```

<NamedElement>For(): Returns a NamedElement for a name in a NameSpace.

```
context <NamedElement>NameSpace
  <NamedElement>For(s:String):<NamedElement>
    self.defs->select(pair | pair.name = s).selectElement().target
```

dom(): All names in the NameSpace.

```
context <NamedElement>NameSpace
  dom():Set(String)
    self.defs->collect(pair | pair.name)
```

ran(): All named elements in the NameSpace.

```
context <NamedElement>NameSpace
  ran():Set(<NamedElement>)
    self.defs->collect(pair | pair.target)
```

name(): Returns the name of an element in the context of a NameSpace.

```
context <NamedElement>
  name(nameSpace:<NamedElement>NameSpace) : String
    nameSpace.nameFor(self)
```

When this template is instantiated the effect will be to generate methods with the right names and appropriate definitions. So given the instantiation in Figure 5, the following method will be generated, for example:

```
context Attribute_NameSpace
 Attribute_For(s:String):Attribute_
   self.defs->select(pair | pair.name = s).selectElement().target
```

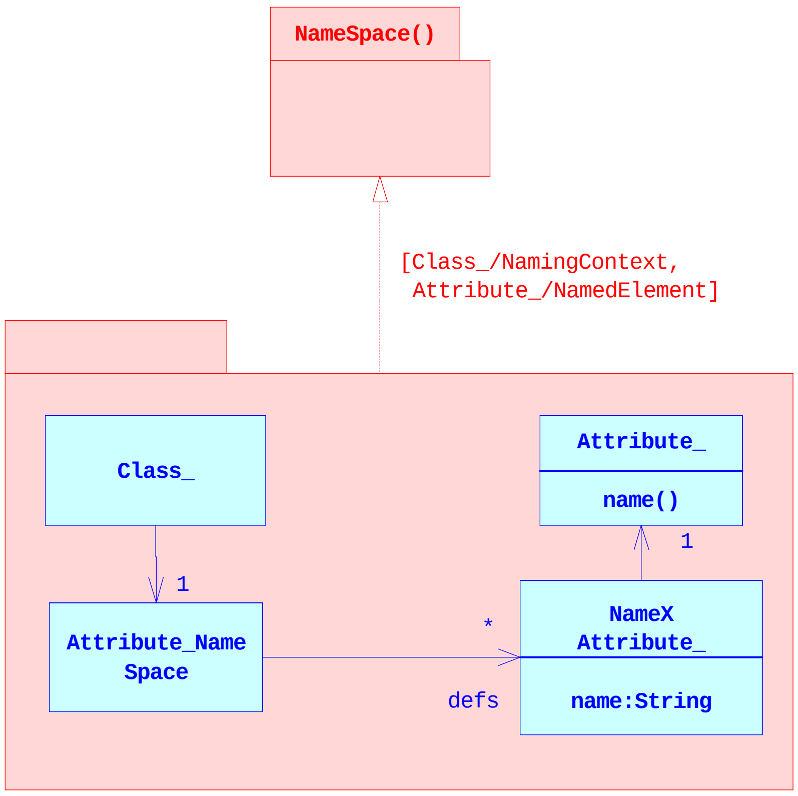


Figure 5. Instantiation of template with query definitions

Templates may also be generated from other templates. This is illustrated by Figure 6.

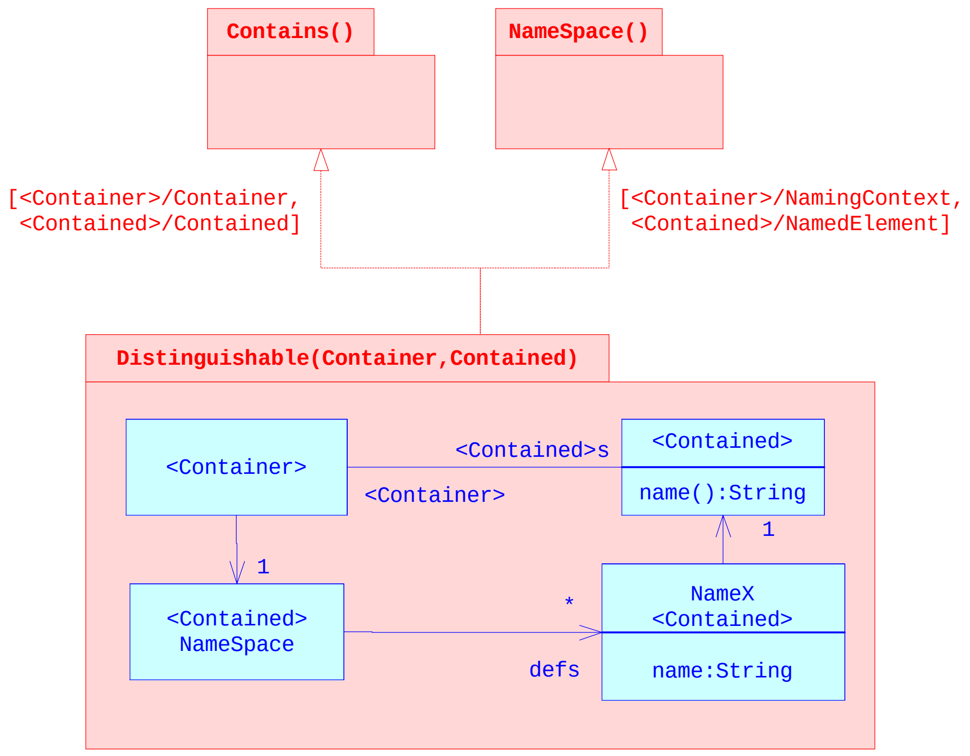


Figure 6. Template generation from templates

As shown by the diagram, this is achieved by instantiating one or more templates, with parameter substitutions that substitute parameters of parent templates with parameters (or not) of the child template. This technique is used extensively in the meta-modelling approach adopted here.

Chapter 4

Templates

This chapter describes a number of templates that are key to defining modelling languages. The application of these templates is demonstrated using fragments of UML2.

4.1. Introduction

The purpose of this chapter is to describe the templates used to define UML 2. Each of the templates described in this chapter represent a self-contained unit of concepts and properties that capture a specific aspect of language design. Taken as a whole they constitute a standard library of template definitions, and for this reason this chapter is presented in the form of a reference manual.

The templates can be broadly categorised as follows:

Basic Templates: fundamental templates that are used to define more complex language templates.

- Contains (Section 4.2 on page 46)
- NameSpace (Section 4.3 on page 48)
- Distinguishable Container (Section 4.4 on page 51)
- Related1 (Section 4.5 on page54)
- Generalisable (Section 4.6 on page 55)
- Semantics (Section 4.7 on page 57)

Extensions: templates that capture relationships between modelling elements, where one modelling element can be viewed as an extension of another modelling element.

- Generalisable Container (Section 4.8 on page 59)
- Declarative Generalisable Container (Section 4.9 on page 62)
- Generalisable Related1 (Figure 4.10 on pa ge65)
- Importable Container (Section 4.11 on page 66)
- Mergeable Container (Section 4.12 on page 66)

Semantics: templates that capture meaning relationships between modelling elements, where the meaning of a modelling element is defined in terms of a mapping to a semantic domain (which is itself a model).

- Container Semantics (Section 4.13 on page 67)
- Generalisable Semantics (Section 4.14 on page 69)
- Generalisable Container Semantics (Section 4.15 on page 71)

- Generalisable Related1 Semantics (Figure 4.16 on page74)
- Generalisable Feature Container Semantics (Section 4.17 on page 76)

Constraints: templates necessary for the definition of constraint languages, such as the Object Constraint Language.

- Expression Semantics (Section 4.18 on pa ge 76)
- Binding (Section 4.19 on page 76)

Mappings: templates that define a vocabulary for expressing transformations between modelling elements and between languages.

- Mapping (Section 4.20 on page 77)
- Container Mapping (Section 4.21 on pa ge 77)

Refinement: templates that define a refinement relationship between modelling elements.

- Refinement (Figure 4.22, “Refinement,” on page 77)

Graphs: templates that define variants of graphs. Graphs and trees have an important role to play in the definition of modelling languages. Many of the templates used in this document are related to these concepts (although this relationship has not been made explicit as yet).

- Graph (Section 4.23 on pag e 78)
- Tree (Section 4.24 on page 79)

Reflection: templates that define the relationship between model elements, their meta-objects (ObjectReflection) and their meta-classes (Class Reflection).

- Object Reflection (Section 4.25 on page 80)
- Class Reflection (Section 4.26 on page 83)

Abstract Class Hierarchy: templates that can be used to construct an abstract class hierarchy similar to that used in UML 1.X. Abstract class hierarchies have a useful role to play for the tool vendor, who may use them to define plug-points for new modelling elements (using a traditional framework style). The following abstract class templates are defined:

- Abstract Container (Section 4.28 on page 87)
- Abstract GeneralisableElement (Section 4.28 on page 87)
- Abstract NameSpace (Section 4.29 on page 88)

The remainder of this chapter consists almost entirely of independent sections, each introducing a definition of a template, a summary of its purpose, a description of the definition and small examples of its usage (including some snapshots). Full examples of the use of the templates can be found in the remaining chapters of the submission. An introduction to the template notation and their informal semantics can be found in chapter 3. An overview of the most important templates and their relationship to each other is shown in Figure 7.

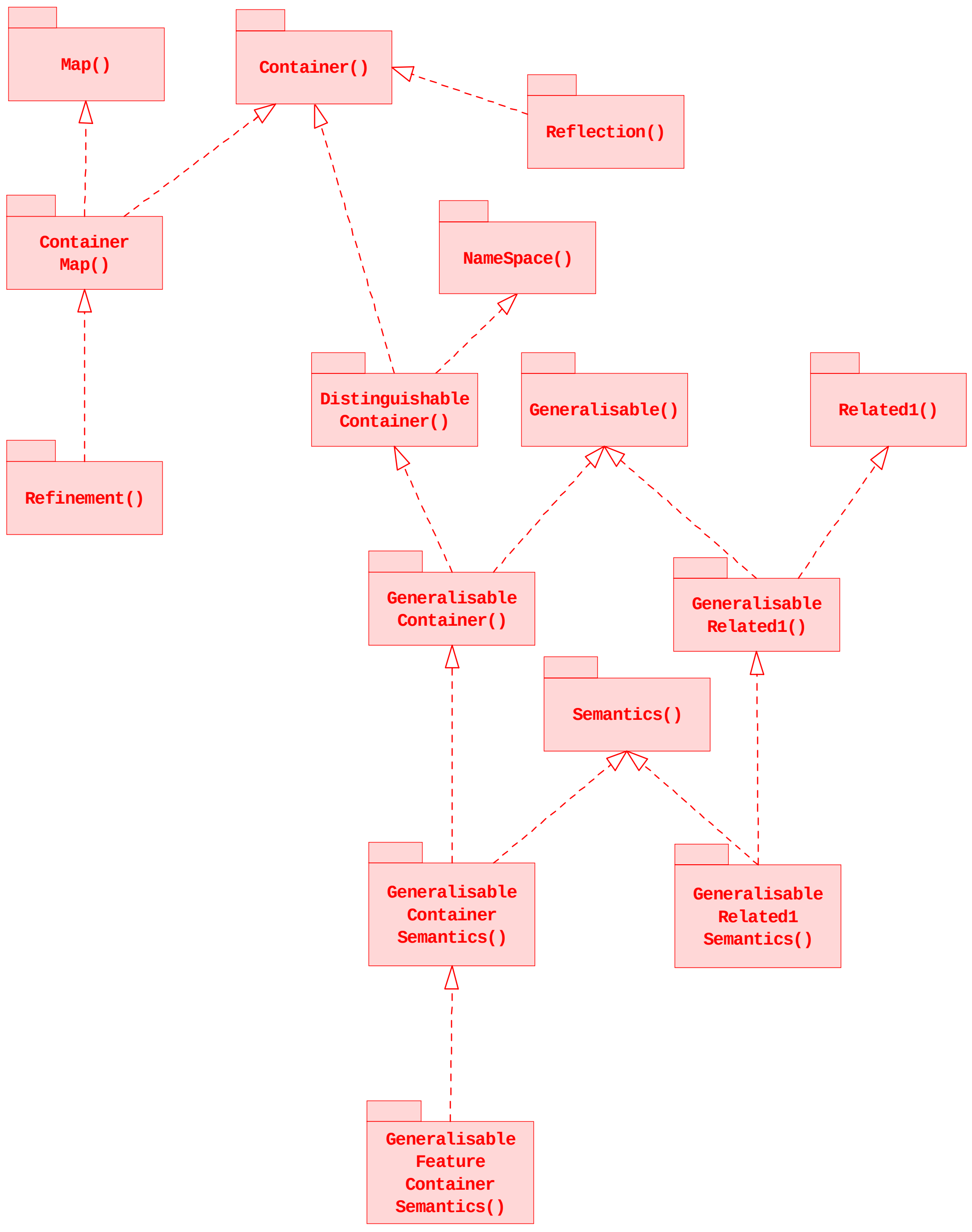

Map()
Container()
Reflection()
Container Map()
NameSpace()
Distinguishable Container()
Generalisable()
Related1()
Refinement()
Generalisable Container()
Generalisable Related1()
Semantics()
Generalisable Container Semantics()
Generalisable Related1 Semantics()
Generalisable Feature Container Semantics()

Figure 7. Overview of Templates

4.2. Contains

Summary

A containment relationship, in which one element, the Container, conceptually contains another element (the Contained element).

Definition

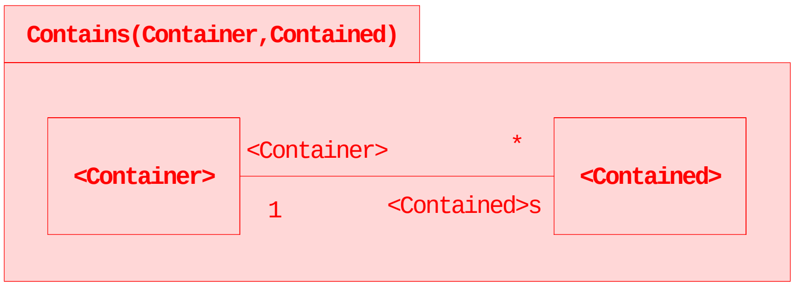


Figure 8. Contains

Queries

None.

Well-formedness Rules

None.

Description

Containers are one of the most fundamental patterns found in a modelling language. Many language elements “contain” other language elements.

Example

A class contains its attributes and queries.

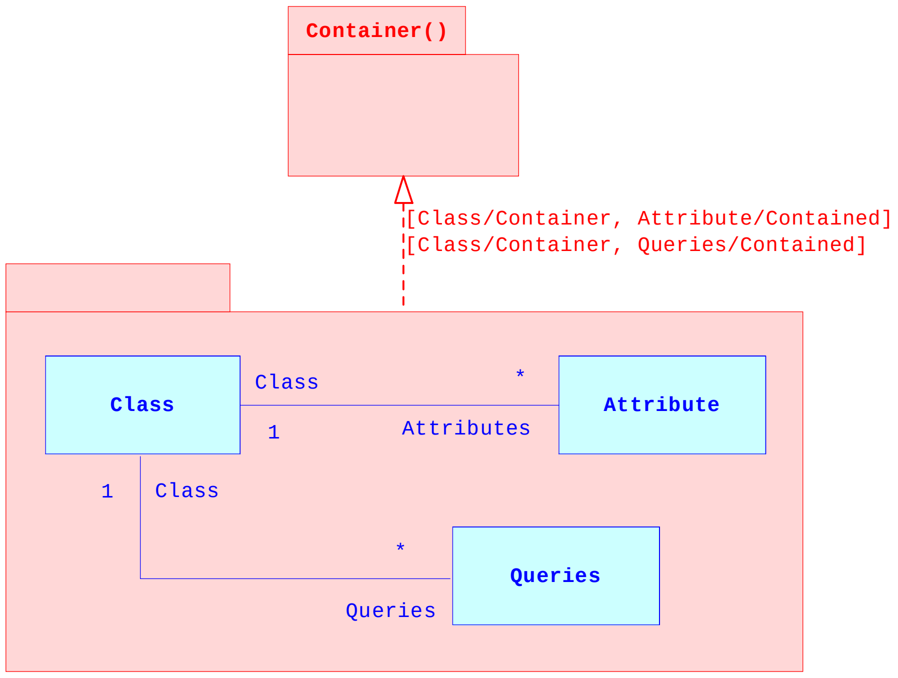

Container()
[Class/Container, Attribute/Contained]
[Class/Container, Queries/Contained]
Class
Class
*
Attribute
1
Attributes
1
Class
*
Queries
Queries

Figure 9. Contains Example

4.3. NameSpace

Summary

Defines a name space for a named element in a naming context. A namespace is a collection of mappings from names to named elements.

Definition

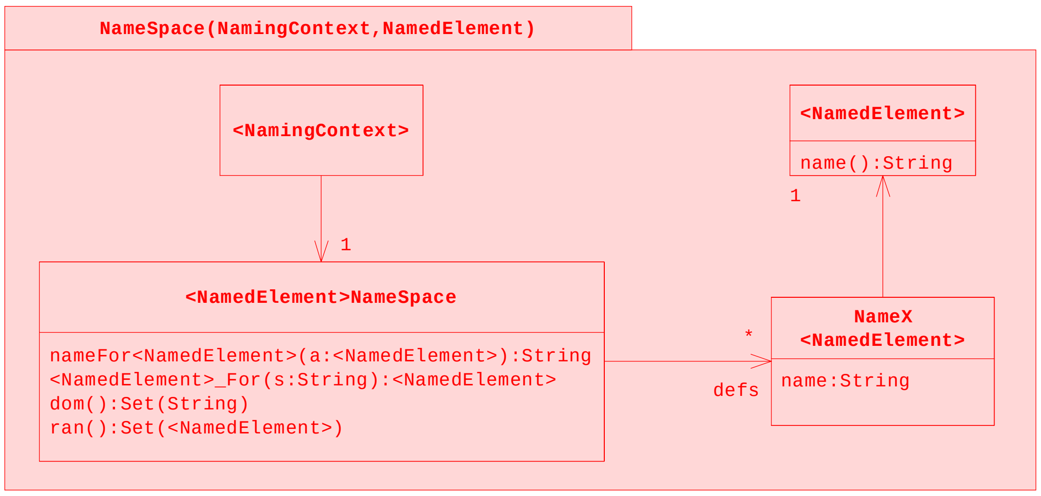

NameSpace(NamingContext,NamedElement)
<NamingContext>
<NamedElement>
name():String
1
1
<NamedElement>NameSpace
nameFor<NamedElement>(a:<NamedElement>):String
<NamedElement>_For(s:String):<NamedElement>
dom():Set(String)
ran():Set(<NamedElement>)
*
defs
NameX
<NamedElement>
name:String

Figure 10. NameSpace

Queries

nameFor<NamedElement>(): Returns a name for a NamedElement in a NameSpace.

```
  context <NamedElement>NameSpace
    "nameFor"<NamedElement>(a:<NamedElement>):String
      self.defs->select(pair | pair.name = a).selectElement().name
```

<NamedElement>_For(): Returns a NamedElement for a name in a NameSpace.

```
   context <NamedElement>NameSpace
    <NamedElement>_For(s:String):<NamedElement>
      self.defs->select(pair | pair.name = s).selectElement().target
```

dom(): All names in the NameSpace.

```
   context <NamedElement>NameSpace
    dom():Set(String)
      self.defs->collect(pair | pair.name)
```

ran(): All named elements in the NameSpace.

```
    context <NamedElement>NameSpace
      ran():Set(<NamedElement>)
        self.defs->collect(pair | pair.target)
```

name(): Returns the name of an element in the context of a NameSpace.

```
context <NamedElement>
  name(nameSpace:<NamedElement>NameSpace) : String
    nameSpace.nameFor(self)
```

Well-formedness Rules

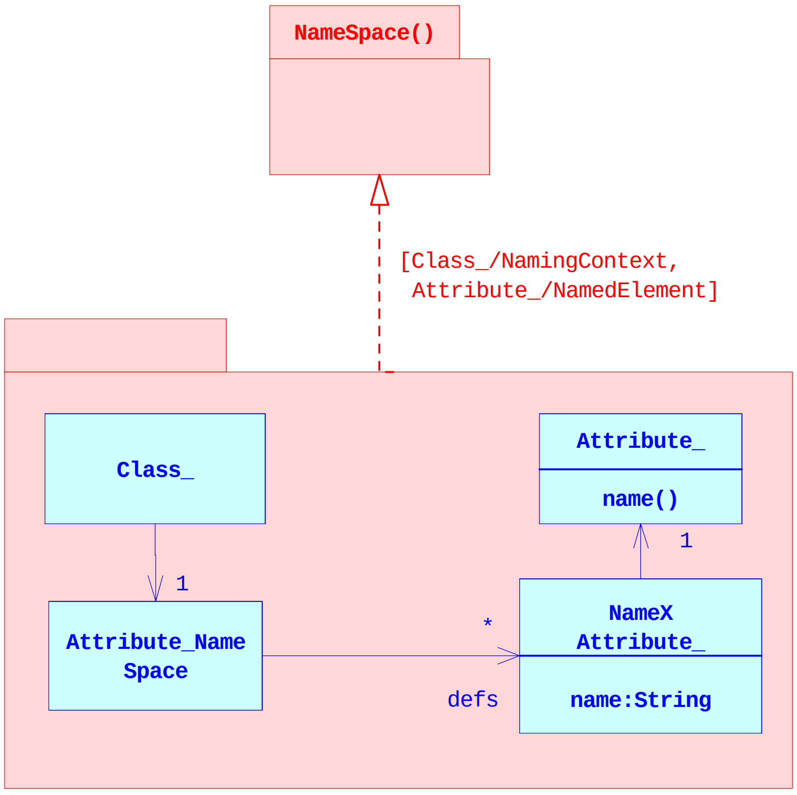
None.

Description

Namespaces are a mechanism for decoupling names from modelling elements. A naming context defines a namespace for all the elements that it can name. Lookup queries are provided for retrieving names and named elements from a namespace.

Example

A class has a namespace for its attributes.



Example Snapshot

Figure 11 shows the definition of a single class named C with two attributes named a and b respectively. Note, in practice, attributes and other namespace elements will not have name attributes. However, we have found that local names considerably reduce the noise associated with name spaces, and should be viewed as interchangeable, i.e. an attribute with name "a" is equivalent to an attribute that is named "a" in a attribute namespace.

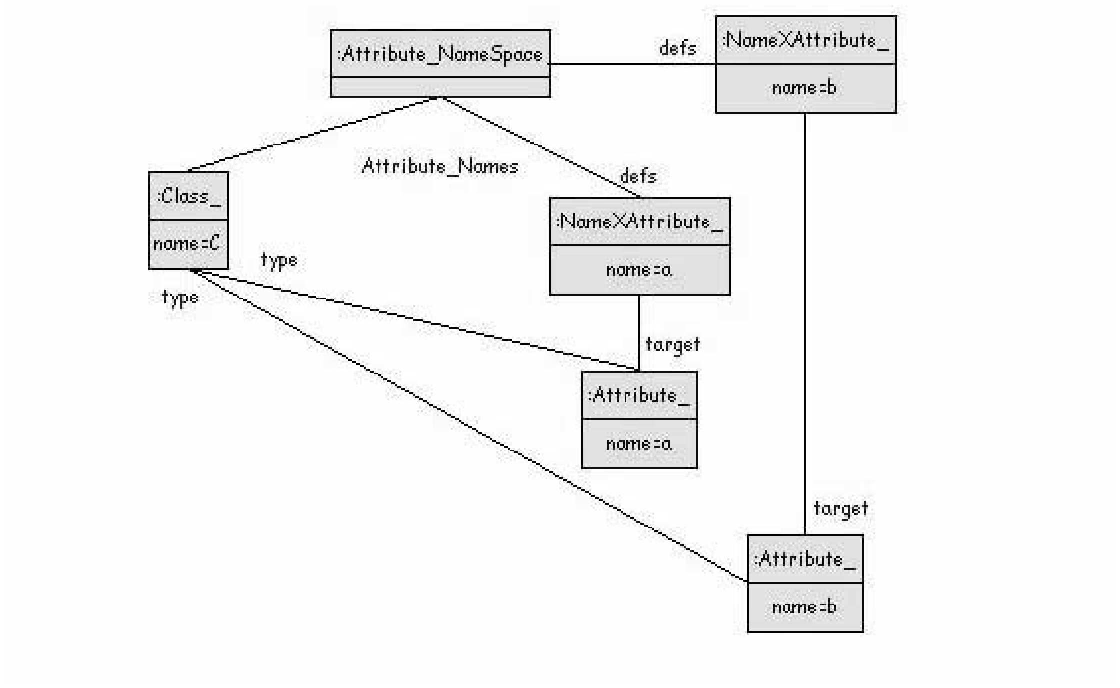


Figure 11. NameSpace Example

4.4. Distinguishable Container

Summary

A distinguishable container is a container that cannot contain two or more elements with the same name.

Definition

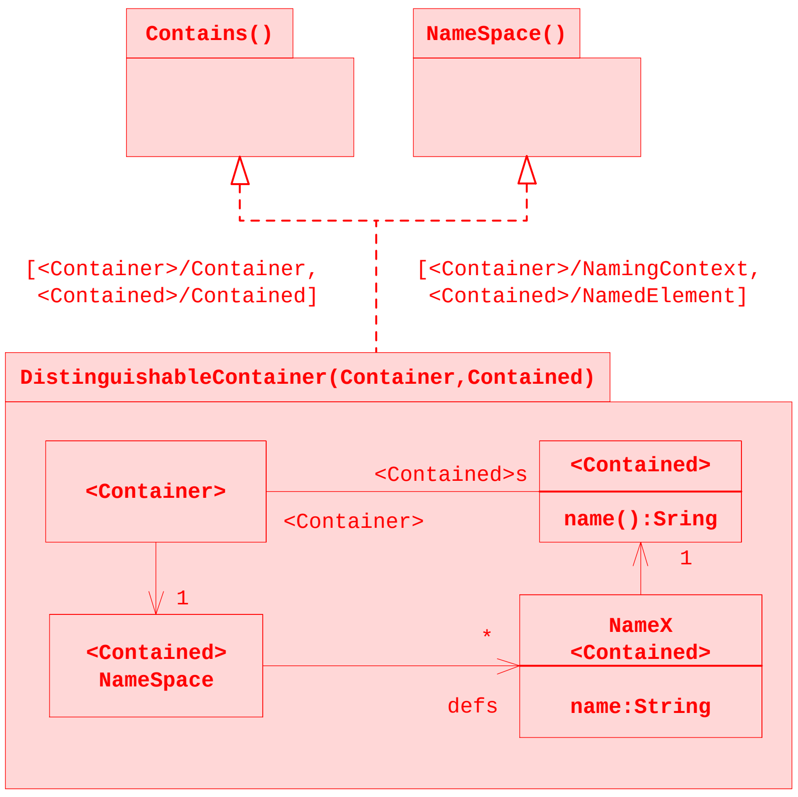


Figure 12. DistinguishableContainer

Queries

None.

Well-formedness Rules

All<Contained>sHaveDistinctNames: All contained elements have distinct names.

```
context <Container> inv:
  All<Contained>sHaveDistinctNames
    self.<Contained>s->forAll(element, element’ |
      element.name(self.<Contained>NameSpace) =
      element'.name(self.<Contained>NameSpace) implies
         element = element')
```

Description

A distinguishable container contains elements that are distinguishable by name through their name() method. This method is defined in the context of a namespace of contained elements (see NameSpace template, Section 4.3 on page 48).

Example

A class's queries and attributes are distinguishable by their name, i.e. no two attributes can share the same name. However, because two separate constraints are generated for each contained element, it is quite legal for a class to contain an attribute *and* a method with the same name.

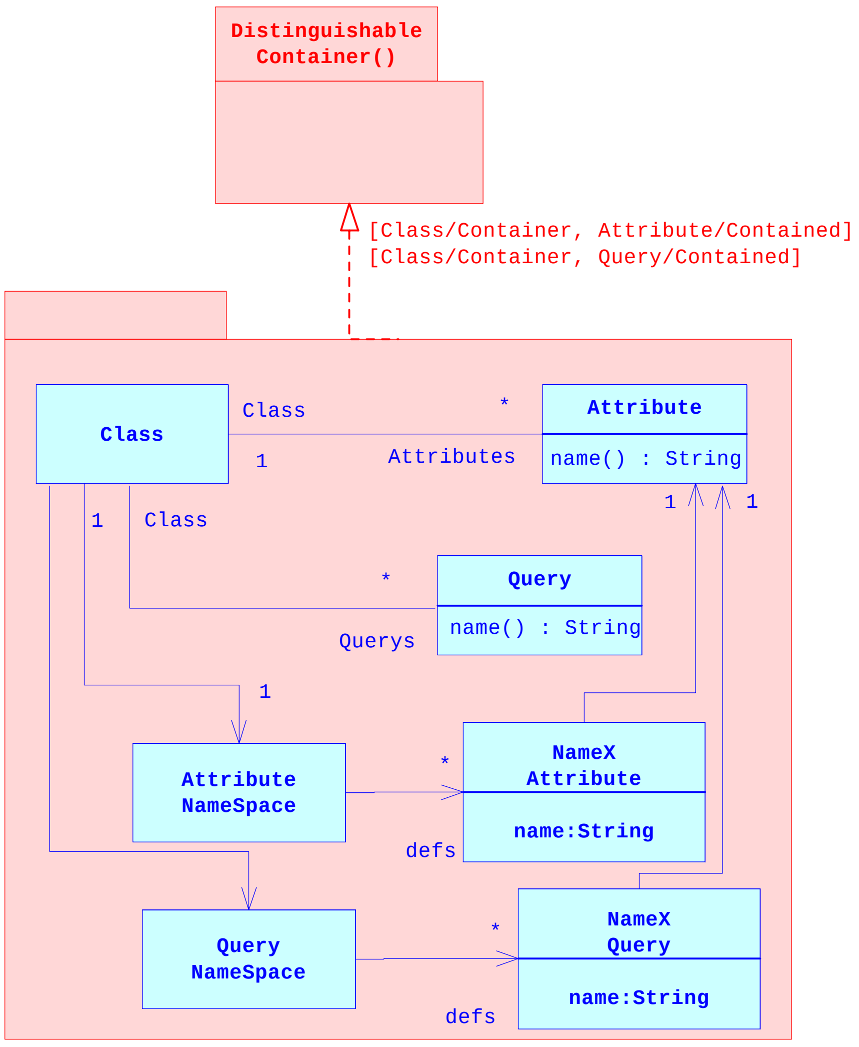


Figure 13. DistinguishableContainer Example

Generated Constraints

AllAttributesHaveDistinctNames: All attributes have distinct names as defined by their namespace.

```
context Class inv:
  AllAttributesHaveDistinctNames
    self.Attributes->forAll(element, element’ |
      element.name(self.AttributeNameSpace) =
      element'.name(self.AttributeNameSpace) implies
        element = element')
```

AllQuerysHaveDistinctNames: All queries have distinct names as defined by their namespace.

```
context Class inv:
  AllQuerysHaveDistinctNames
    self.Querys->forAll(element, element’ |
      element.name(self.QueryNameSpace) =
      element'.name(self.QueryNameSpace) implies
        element = element')
```

4.5. Related1

Summary

Defines a many to one relationship between two elements.

Definition

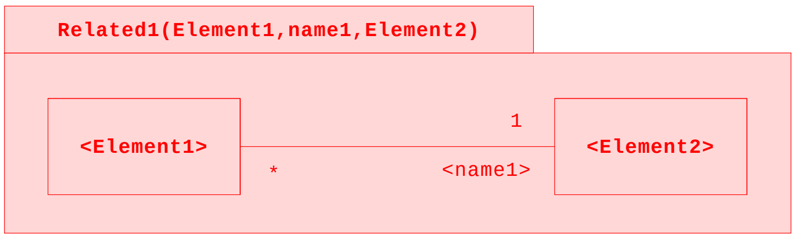


Figure 14. Related1

Description

Related1 is just one of many “relation” templates that can be defined between two elements. It describes a function, where many instances of <Element1> are mapped to a single instance of <Element2>. In future, this template will be generalised to permit the substitution of multiplicities (thus avoiding the need to define N templates for N different relations). The Contains template (section 4.2) is probably just another variant.

Example

Attribute, owner, “type” and Class can be readily substituted for <Element1>, <name1>, and <Element2> respectively to model the relationship between an Attribute and its type.

4.6. Generalisable

Summary

A generalisable type has parents and children.

Definition

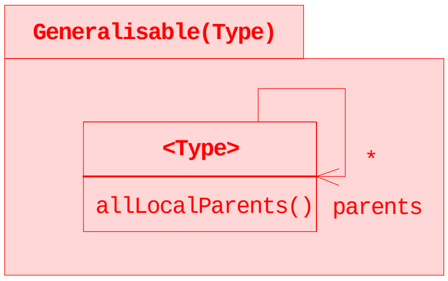


Figure 15. Generalisable

Queries

allLocalParents(): Returns all local parents of the type.

```
context <Type>
   allLocalParents() : Set(<Type>)
      self.parents->iterate(parent S = self.parents |
        S->union(parent.allParents()))
```

Well-formedness Rules

No Circular Inheritance: A generalisable element cannot specialise itself.

```
context <Type> inv:
   NoCircularInheritance
   self.allLocalParents() -> excludes(self)
```

Description

Model elements are *generalisable* when they have parents.

Example

Class and Package can both be substituted for <Type>. The result of substituting Class into the template is shown in Figure 16.

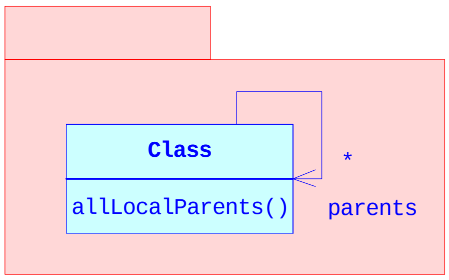


Figure 16. Generalisable Example

Example Snapshot

A snapshot satisfying the properties of Figure 16 is shown on the left hand side of Figure 17. Note that the right hand side shows an example of a snapshot that invalidates the circular inheritance rule.

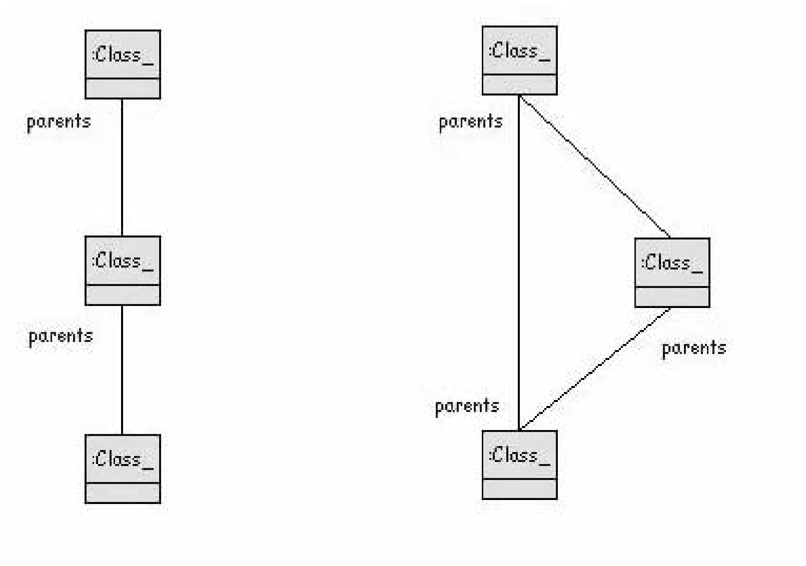


Figure 17. Generalisable Snapshots

4.7. Semantics

Summary

The semantics of a model element are defined in terms of a mapping to multiple instance elements in a semantic domain.

Definition

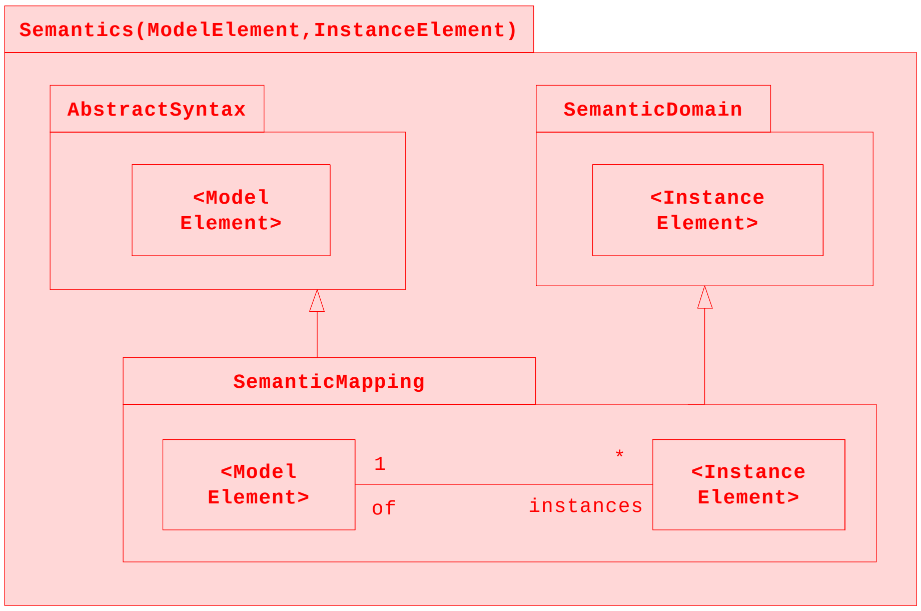


Figure 18. Semantics

Queries

None.

Well-formedness Rules

None.

Description

The Semantics template captures the model element/instance relationship that occurs between abstract syntax elements and their instances in a semantic domain. The different roles played by elements in a semantic definition are clearly delineated by partitioning the Semantics package into an AbstractSyntax package (which contains all model elements), a SemanticDomain package (which contains all instance elements) and a SemanticMapping package. The SemanticMapping Package specialises both the abstract syntax and semantic domain packages and defines an instances/of association between model elements and their instances. Thus, an instance will always be able to determine what it is an instance of.

Example

Instances of associations are links.

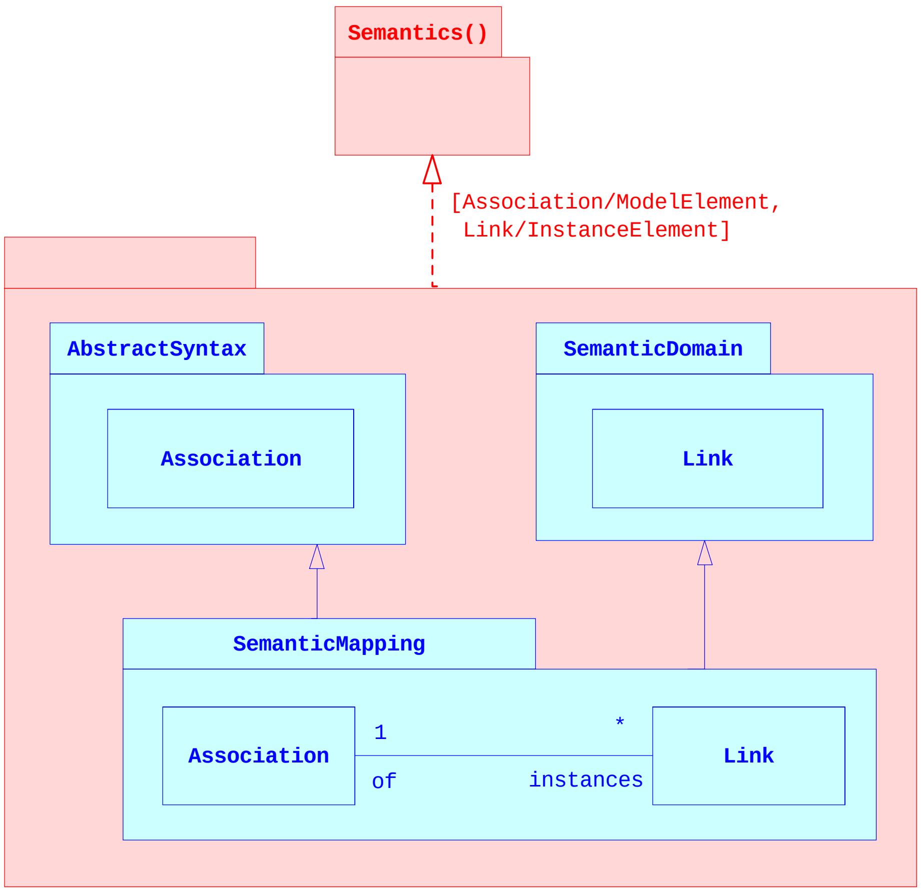


Figure 19. Semantics Example

4.8. Generalisable Container

Summary

Defines a query-based generalisable container, in which the contents of a container are calculated from its parents contents. Note, namespaces are not shown for brevity.

Definition

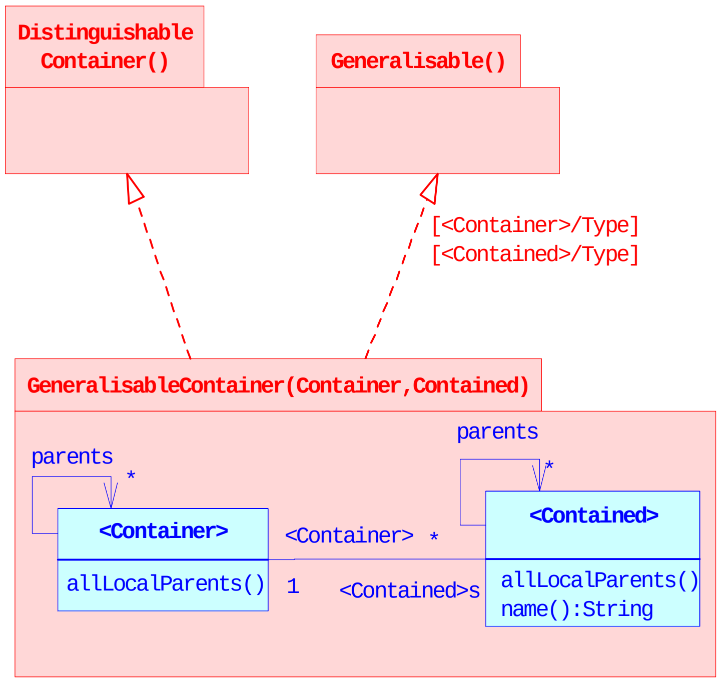


Figure 20. GeneralisableContainer

Queries

allParents(): Returns all parents of a contained element, including all local parents and all derived parents.

```
context class <Contained>
  allParents():Set(<Contained>)
    self.allLocalParents()->union(self.allDerivedParents());
```

all<Contained>s(): Returns all contained element.

```
context class <Container>
  all<Contained>s() : Set(<Contained>)
    self.allParents() -> iterate(p s = self.<Contained>s |
        s->union(p.all<Contained>s()->
          reject(c | s->exists(c' |
            c.name(self.<Contained>NameSpace) =
            c'.name(self.<Contained>NameSpace) or
            c'.parents ->includes(c)))))
```

allDerivedParents(): Returns all parents of an element that share the same name as the contents of its container's parents.

```
context class <Contained>
  allDerivedParents() : Set(<Contained>)
    if self.<Container> <> self then
      self.<Container>.allParents() -> iterate(p s = Set{} |
        s->union(p.<Contained>s ->
          select(c |
            c.name(self.<Contained>NameSpace) =
            self.name(self.<Contained>NameSpace))))
      else Set{}
    endif
```

Well-formedness Rules

None.

Description

A generalisable container can calculate its contents by finding the contents of all its parents, and determining whether it indirectly inherits from them.

Example

Figure 21 shows the template applied twice to Package and Class (a Package contains Classes) and to Class and Attribute (a Class contains its Attributes). Note. namespaces aren't shown for brevity.

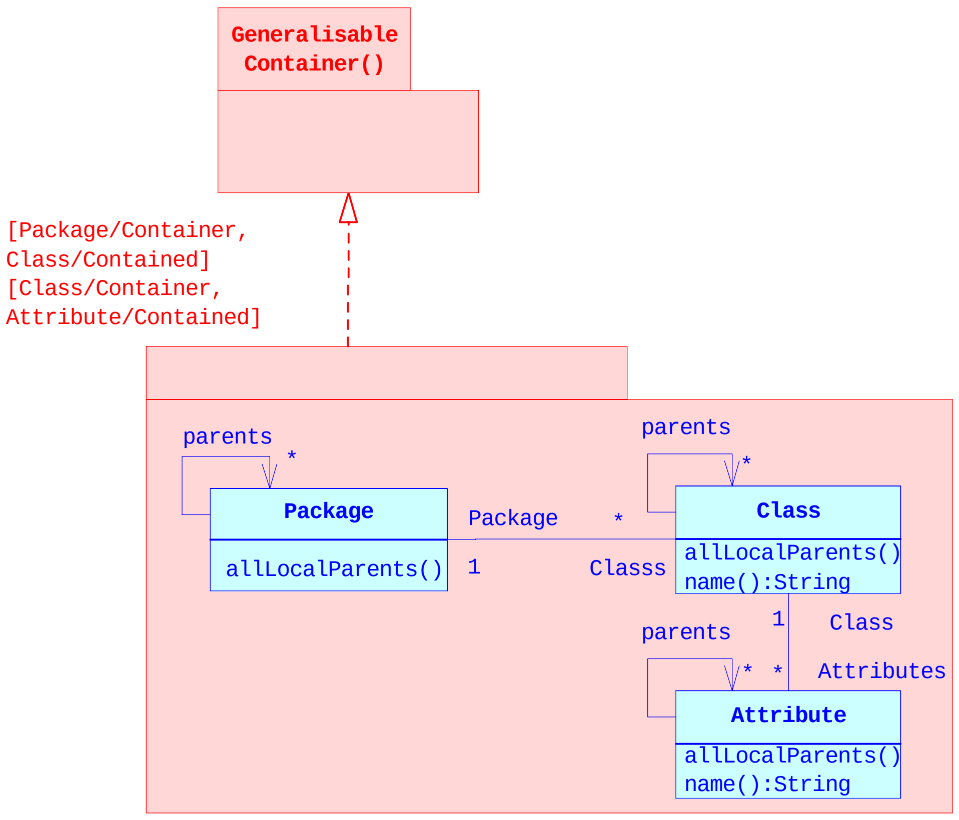


Figure 21. GeneralisableContainer Example

A class can now calculate all its attributes (including those inherited from its parents) via the query:

```
context Class
  allAttributes() : Set(Class)
    self.allParents() -> iterate(p s = self.Attributes |
        s->union(p.allAttributes()->
          reject(c | s->exists(c' |
            c.name(self.AttributeNameSpace) =
            c'.name(self.AttributeNameSpace) or
            c'.parents ->includes(c)))))
```

This will iterate through all the parents of the class including all its derived parents (see below) rejecting any duplicate attributes that might otherwise be inherited from the parents or any redefined attributes.

All derived parents of Class will return all parents of Class that share the same name as the classes of its package's parents.

```
context Class
  allDerivedParents() : Set(Class)
    if self.Package <> self then
      self.Package.allParents() -> iterate(p s = Set{} |
        s->union(p.Attributes -> select(c |
          c.name(self.ClassNameSpace) =
          self.name(self.ClassNameSpace))))
      else Set{}
    endif
```

Example Snapshot

In Figure 22, allAttributes() of the class class c in the package a will return the attributes {e, f, g}. Attribute f is inherited from the localParents of the class c in the package a, whilst attribute e is inherited from the derived parent of c (i.e. the class c in package b). Note that for brevity, names shown on objects are the names that would be returned by their container's namespace.

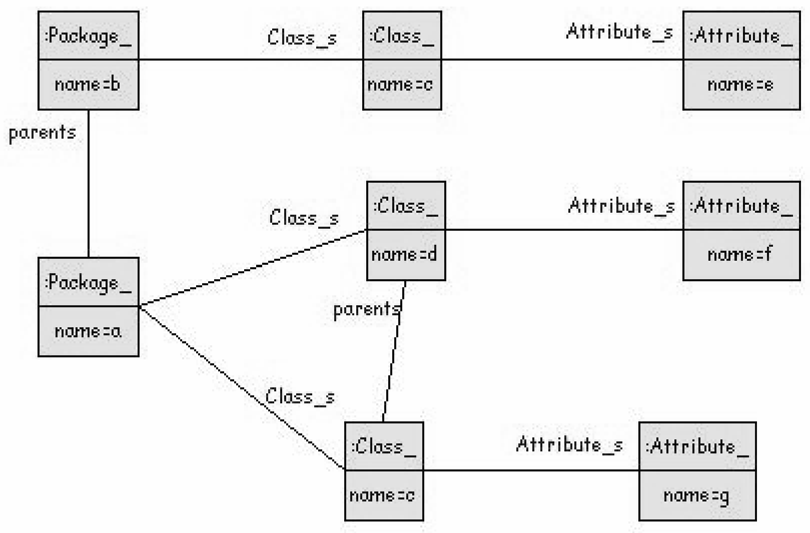


Figure 22. GeneralisableContainer Snapshot

4.9. Declarative Generalisable Container

Summary

A declarative generalisable container has the property that it must specialise the contents of its parents. It is given as an example of a more specification-oriented approach to defining generalisable containers. Note, namespaces aren't shown for brevity.

Definition

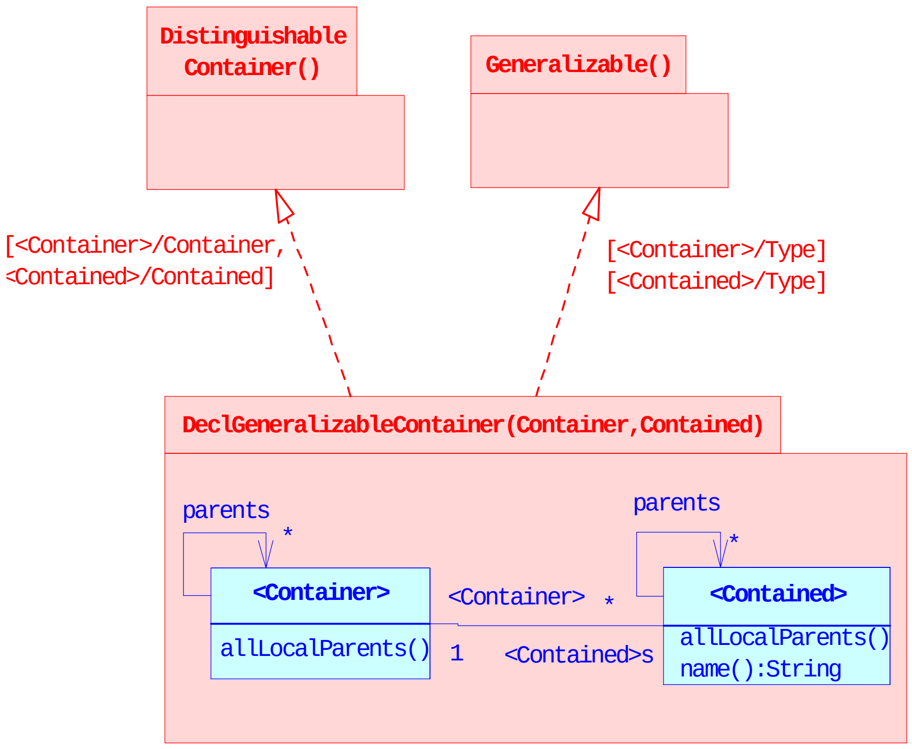

Distinguishable Container()
Generalizable()
[<Container>/Container, <Contained>/Contained]
[<Container>/Type] [<Contained>/Type]
DeclGeneralizableContainer(Container,Contained)
parents
*
<Container>
allLocalParents()
parents
*
<Contained>
allLocalParents()
name():String
<Container>
*
1
<Contained>s

Figure 23. DeclarativeGeneralisableContainer

Queries

parents<Contained>s(): Returns the contents of all parents of a container.

```
context class <Container>
  parents<Contained>s() : Set(<Contained>)
     self.parents->iterate(parent S = Set{} | S->
     union(parent.<Contained>s));
```

<Contained>sParents(): Returns the parents of all contents of a container.

```
context class <Container>
  <Contained>parents() : Set(<Contained>)
      self.<Contained>s->iterate(
      x S = Set{} | S->union(x.parents))
```

Well-formedness Rules

<Contained>sParents(): A container's parents' contents must be a subset of its contents' parents.

```
context class <Container>
  ContentsParentsAreCorrect inv:
     self.parents<Contained>s().subset(self.<Contained>sParents())
```

Description

A declarative generalisable container does not calculate its contents like a generalisable container. Instead it must guarantee that it will specialise the contents of its parents, whilst also permitting the addition of new contents.

Example

Note, namespaces aren't shown for brevity.

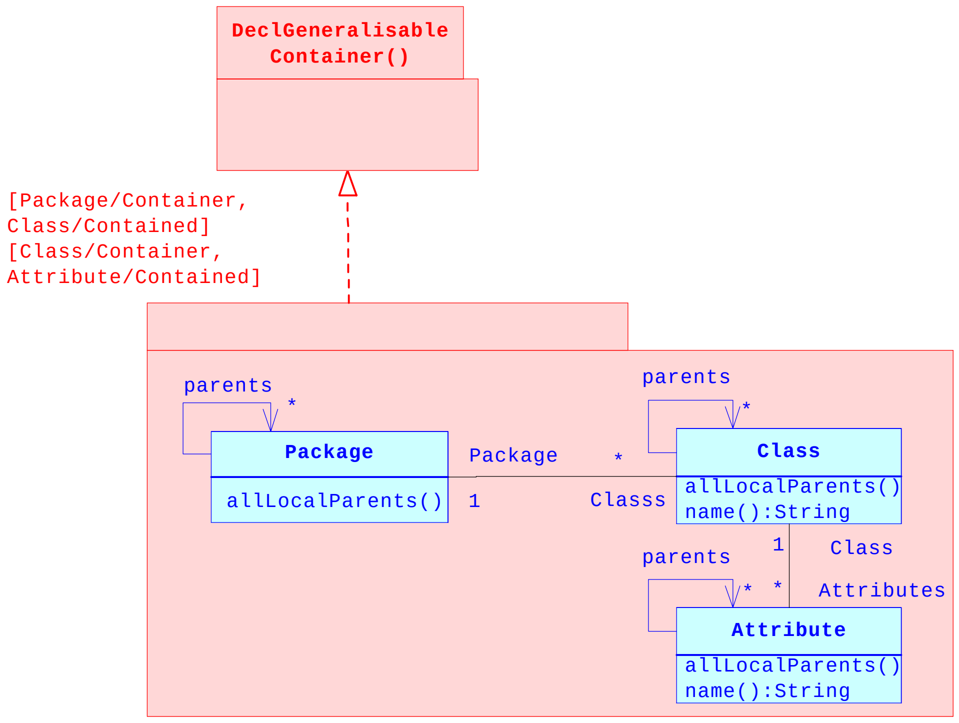


Figure 24. DeclarativeGeneralisableContainer Example

Snapshot Example

Figure 25 shows an example of a snapshot that satisfies the well-formedness rules of the model shown in Figure 24. Here, class c in package a must contain additional attributes f and e as *specialisations* of the attributes of classes b.c and a.d. Note that for brevity, the names shown on the objects are the names that would be returned by a namespace.

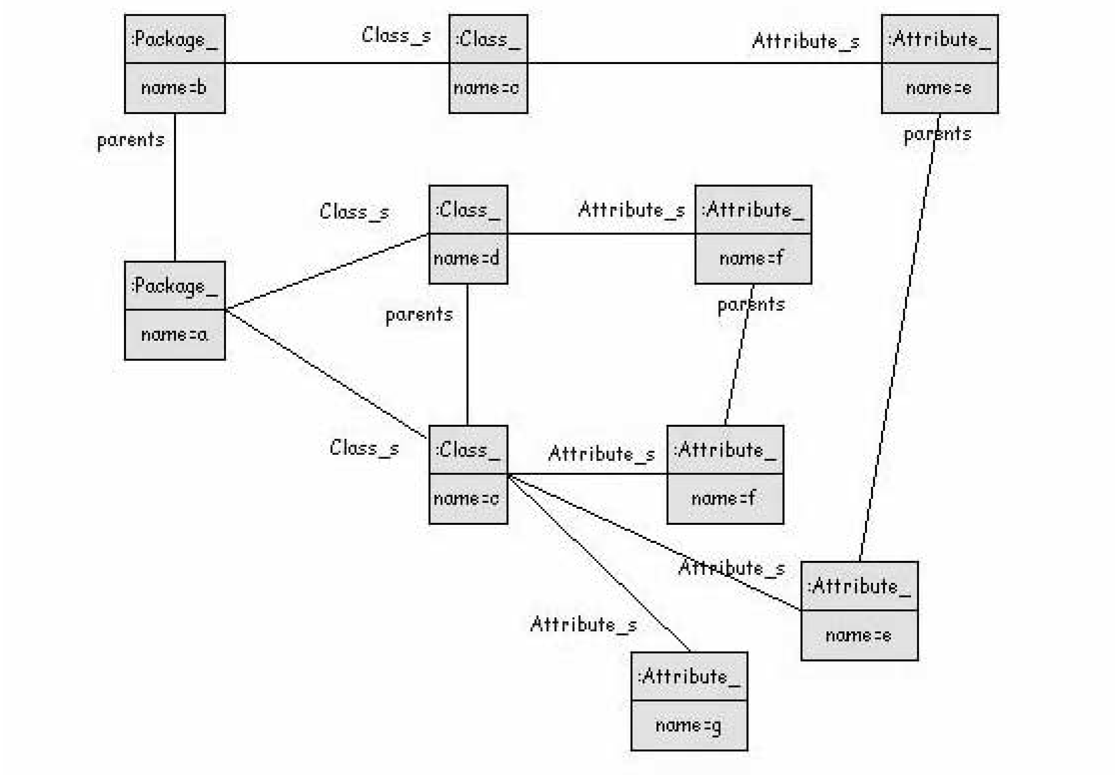


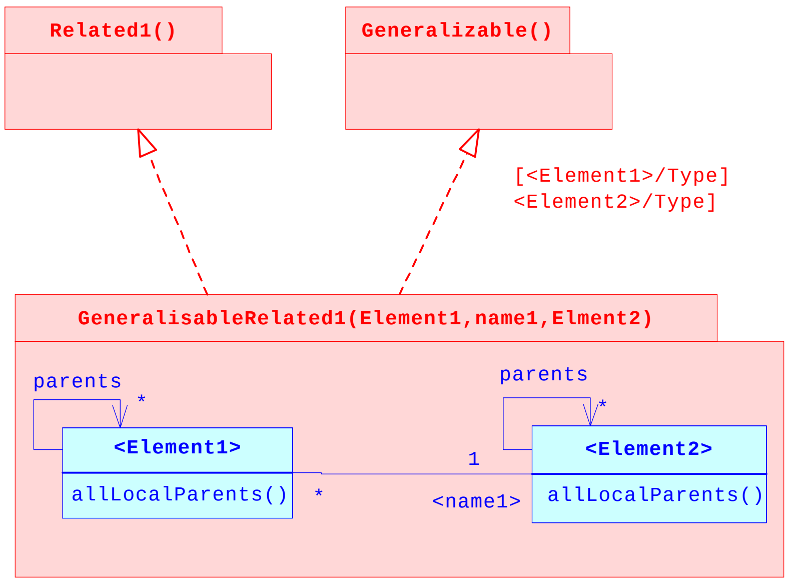
Figure 25. DeclarativeGeneralisableContainer Snapshot

4.10. Generalisable Related1

Summary

Defines the relationship between two generalisable element, where one element is related to the other.

Definition



Well-formedness Rules

RelatedElementCommute: All an elements parents must be related to the parents of the elements its is related to.

```
context <Element1> inv:
  RelatedParentsCommute
  self.<<name1>>.parents = self.parents ->
    iterate(p s = Set{} | s->including(p.<<name1>>))
```

Description

A generalisable related element has the property that all its parents must be related to the parents of the elements it is related to.

Example

None.

4.11. Importable Container

Summary

An importable container can import another containers namespace.

Definition

To be defined in the final submission.

Description

E*xample*

4.12. Mergeable Container

Summary

A mergeable container can be merged with another container.

Definition

To be defined in the final submission.

Description

E*xample*

4.13. Container Semantics

Summary

Containers have instances, whose elements are instances of the container's contents.

Definition

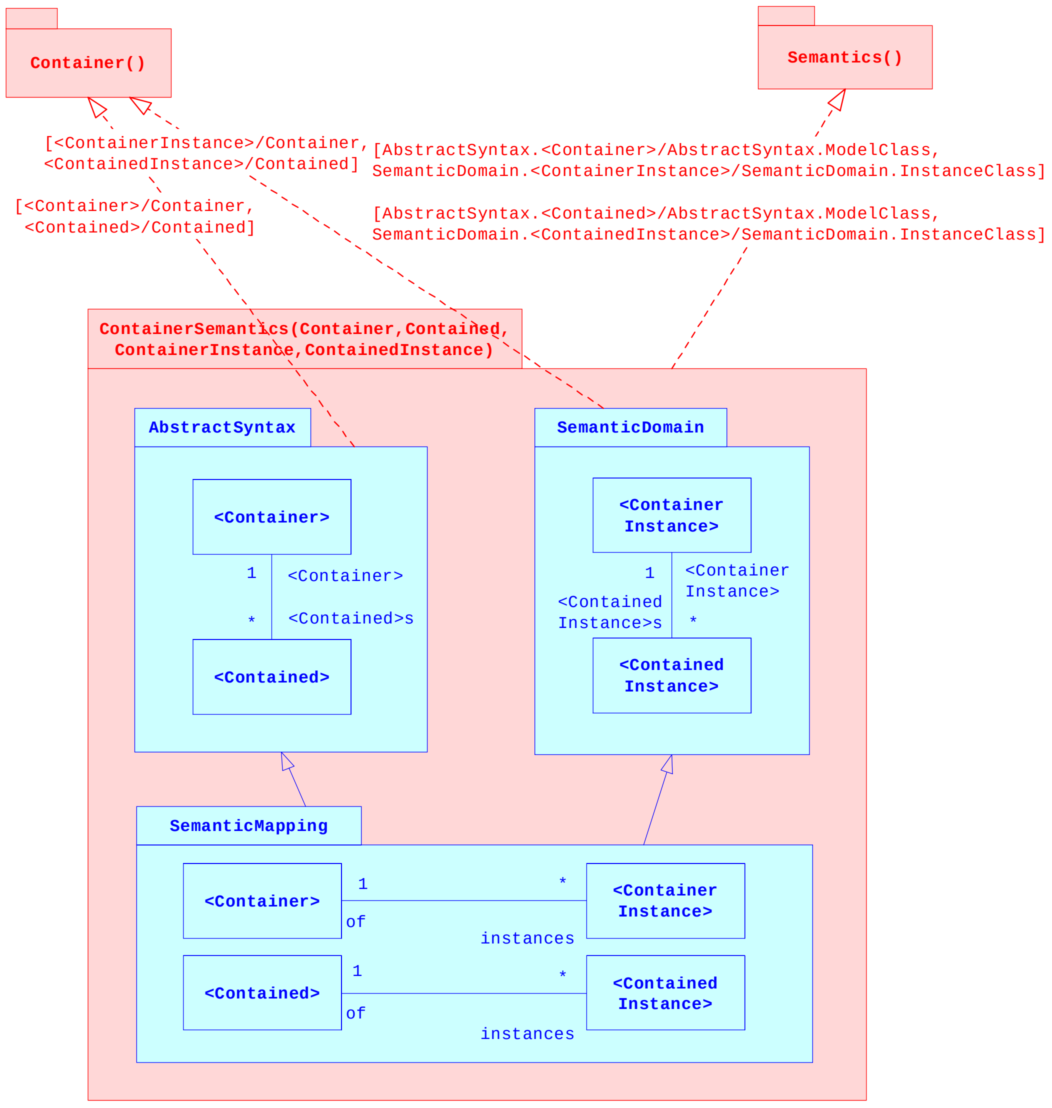

Container()
Semantics()
[<ContainerInstance>/Container,
<ContainedInstance>/Contained]
[<Container>/Container,
<Contained>/Contained]
[AbstractSyntax.<Container>/AbstractSyntax.ModelClass,
SemanticDomain.<ContainerInstance>/SemanticDomain.InstanceClass]
[AbstractSyntax.<Contained>/AbstractSyntax.ModelClass,
SemanticDomain.<ContainedInstance>/SemanticDomain.InstanceClass]
ContainerSemantics(Container,Contained,
ContainerInstance,ContainedInstance)
AbstractSyntax
<Container>
1
<Container>
*
<Contained>s
<Contained>
SemanticDomain
<Container Instance>
1
<Container Instance>
<Contained Instance>s
*
<Contained Instance>
SemanticMapping
<Container>
1
*
of
instances
<Container Instance>
<Contained>
1
*
of
instances
<Contained Instance>

Figure 26. ContainerSemantics

Queries

None.

Well-formedness Rules

InstanceContentsCommute: Instance contents commute.

```
context class <ContainerInstance> inv:
  InstanceContentsCommute
     self.of.<Contained>s = self.<ContainedInstance>s ->
     iterate(element S = Set{} | S->union(Set{element.of}))
```

Description

This template describes the relationship between containers and their instances. A container’s structure is reflected on the instance side. Each instance of a container will contain instances of the container’s contents.

E*xample*

See Generalisable Container Semantics, Section 4.15.

4.14. Generalisable Semantics

Summary

Defines the semantics of generalisation as an inheritance preserving relationship between elements and their instances.

Definition

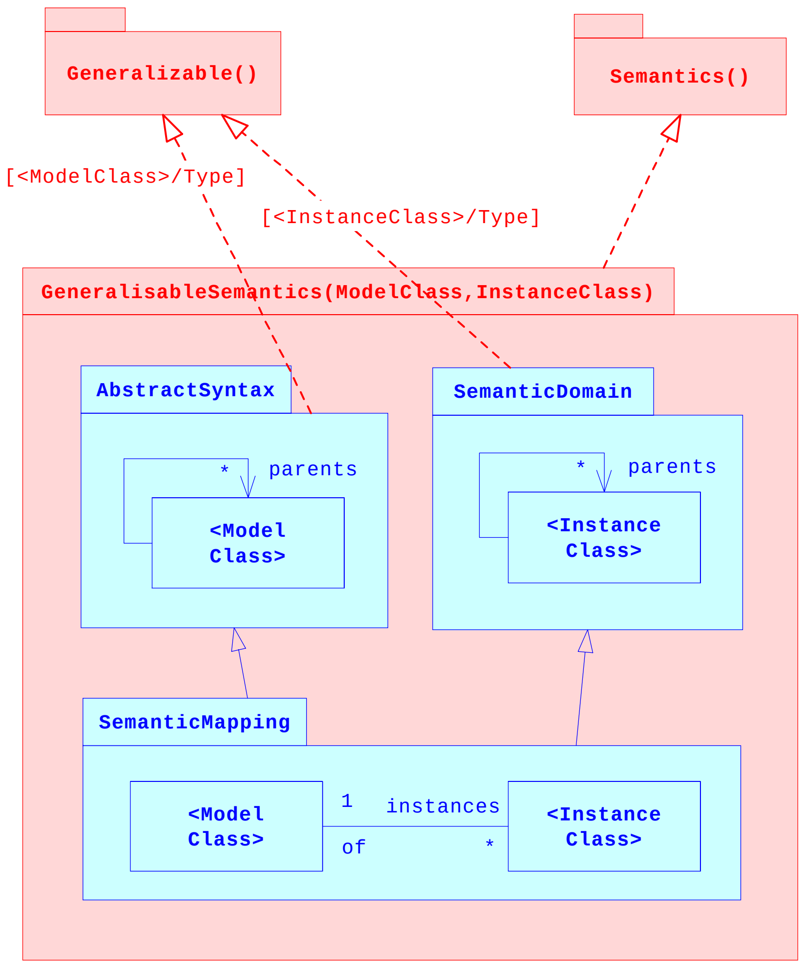


Figure 27. GeneralisableSemantics

Queries

None.

Well-formedness Rules

InstanceParentsCommute: Instance parents commute.

```
context class <InstanceClass> inv:
  InstanceParentsCommute
     self.of.parents = self.parents ->
     iterate(parent S = Set{} | S->union(Set{parent.of}))
```

Description

Instances of generalisable elements have the same generalisation hierarchy as their elements. The intention is that a semantic domain element that is an instance of an element may easily be viewed as an instance of a parent by selecting the appropriate super-instances.

Example

See Generalisable Container Semantics (Section 4.15 on page 71).

4.15. Generalisable Container Semantics

Summary

Query-based generalisable container semantics (note namespaces aren't shown for brevity).

Definition

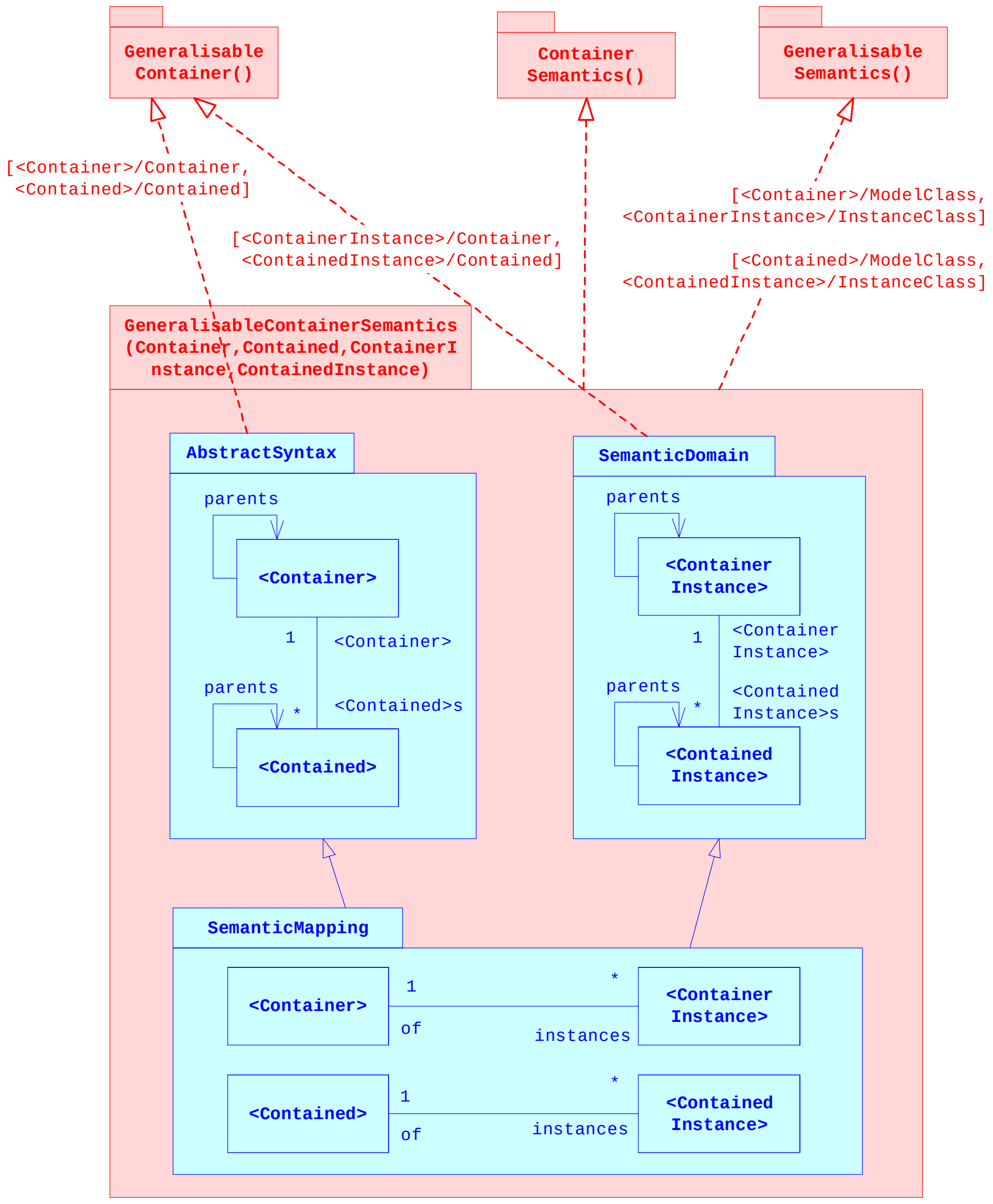

Generalisable Container()
Container Semantics()
Generalisable Semantics()
[<Container>/Container, <Contained>/Contained]
[<ContainerInstance>/Container, <ContainedInstance>/Contained]
[<Container>/ModelClass, <ContainerInstance>/InstanceClass]
[<Contained>/ModelClass, <ContainedInstance>/InstanceClass]
GeneralisableContainerSemantics (Container,Contained,ContainerInstance,ContainedInstance)
AbstractSyntax
parents
<Container>
1
<Container>
parents
*
<Contained>s
<Contained>
SemanticDomain
parents
<Container Instance>
1
<Container Instance>
parents
*
<Contained Instance>s
<Contained Instance>
SemanticMapping
<Container>
1
*
of
instances
<Container Instance>
<Contained>
1
*
of
instances
<Contained Instance>

Figure 28. GeneralisableContainerSemantics

Queries

None.

Well-formedness Rules

None.

Description

The generalisable container semantics (GCS) template merges properties of containable, generalisable and instantiable modelling elements. Any element E that contains other elements must guarantee to have instances that contain instances corresponding to the elements of E. Furthermore, the parents of an instance should reflect the same generalisation hierarchy as the element. In this way, a GCS element can itself be viewed as a template for stamping out its instances, i.e. each of its instances will guarantee to preserve the structure of the GCS element.

Example

Figure 29 shows an example of defining a partial semantics for classes and attributes. Instances of classes are objects using the GCS template. Instances of attributes are slots. The slots contained by an object must match the attributes contained by its class and the object must preserve the same inheritance hierarchy as its class. Note namespaces aren't shown for brevity.

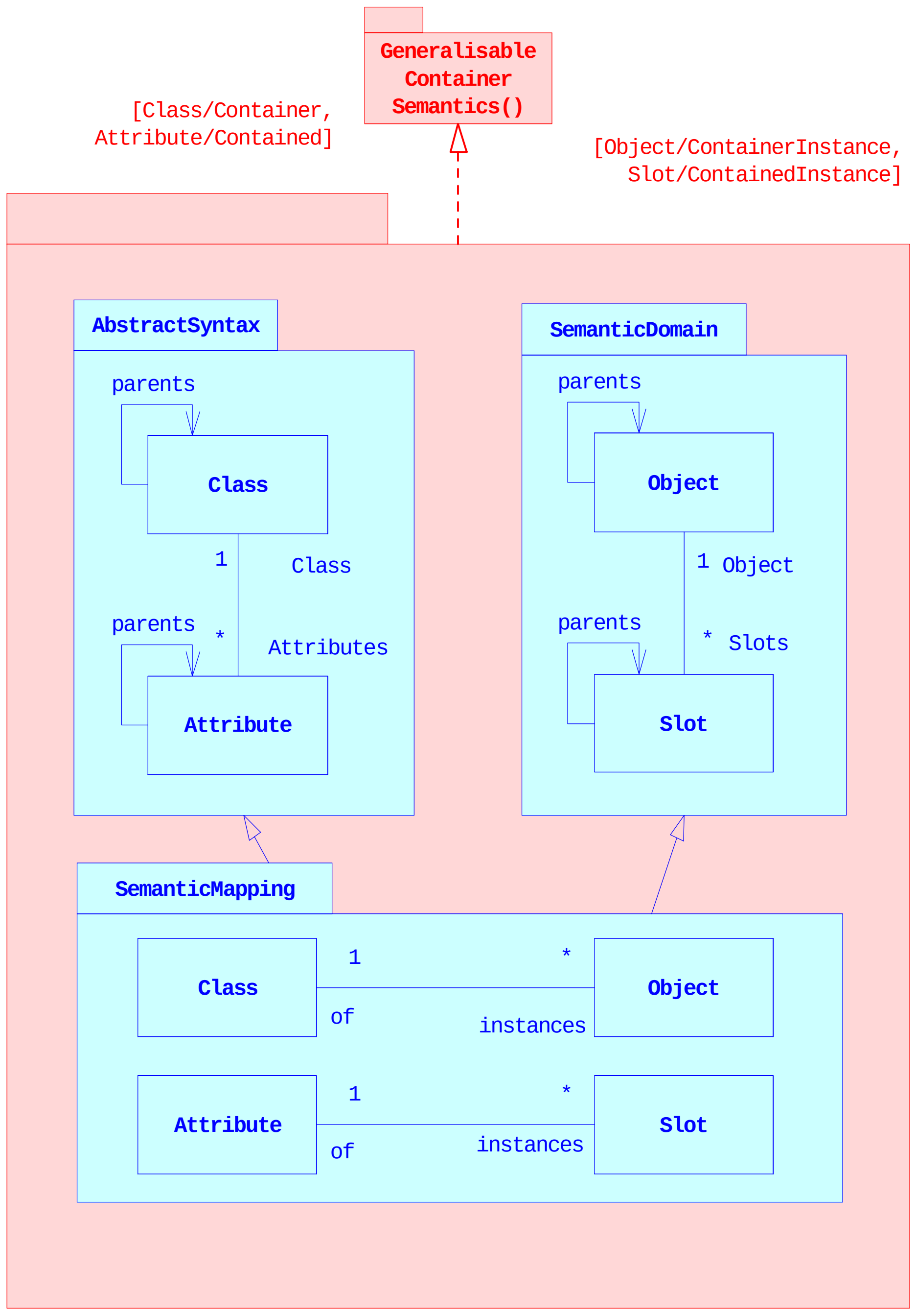


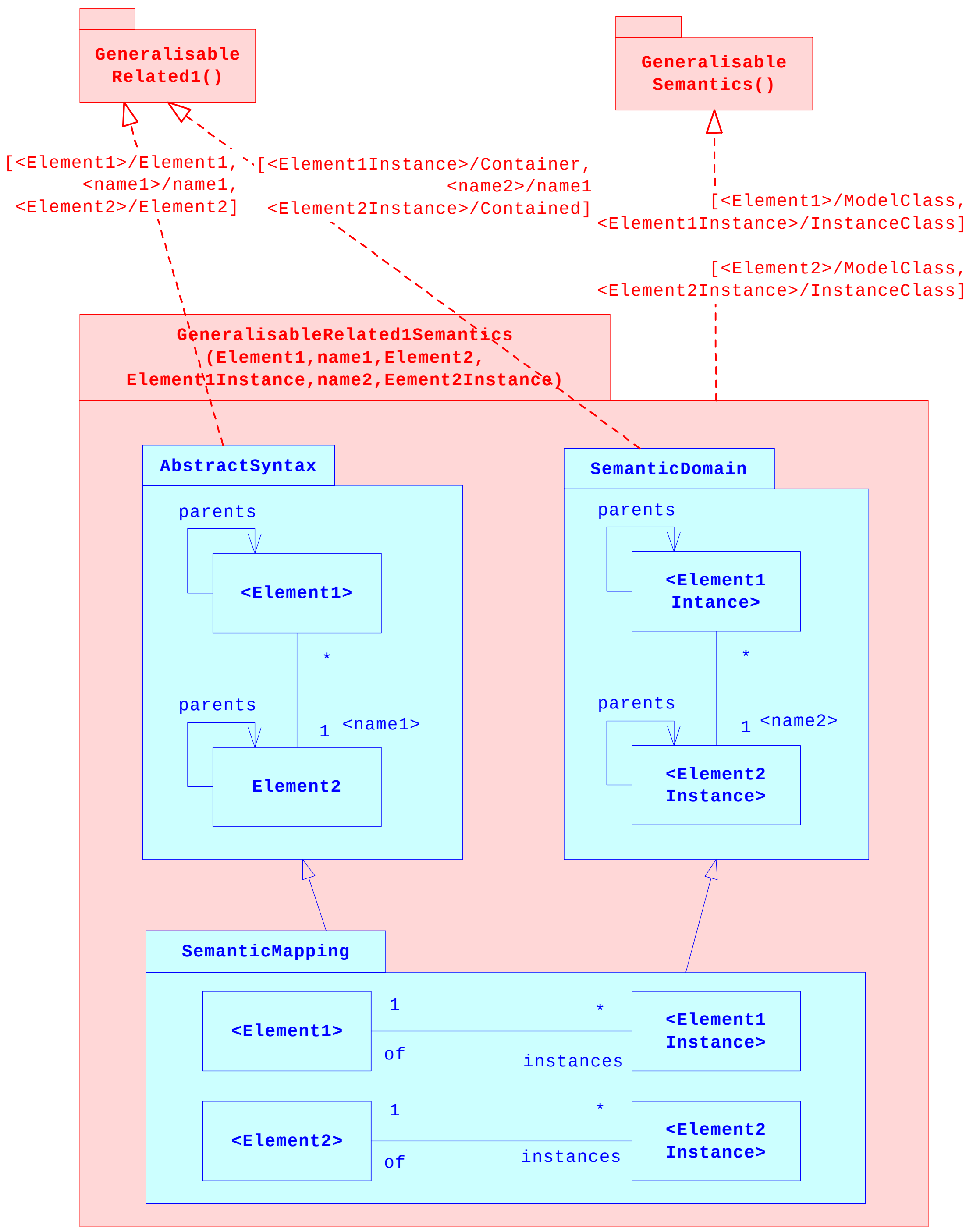
Figure 29. GeneralisableContainerSemantics Example

4.16. Generalisable Related1 Semantics

Summary

Defines the semantic relationship between generalisable related elements and their instances.

Definition



Generalisable Related1()
Generalisable Semantics()
[<Element1>/Element1, <name1>/name1, <Element2>/Element2]
[<Element1Instance>/Container, <name2>/name1 <Element2Instance>/Contained]
[<Element1>/ModelClass, <Element1Instance>/InstanceClass]
[<Element2>/ModelClass, <Element2Instance>/InstanceClass]
GeneralisableRelated1Semantics (Element1,name1,Element2, Element1Instance,name2,Eement2Instance)
AbstractSyntax
parents
<Element1>
*
parents
1 <name1>
Element2
SemanticDomain
parents
<Element1 Intance>
*
parents
1 <name2>
<Element2 Instance>
SemanticMapping
<Element1>
1
of
*
instances
<Element1 Instance>
<Element2>
1
of
*
instances
<Element2 Instance>

Well-formedness Rules

InstanceElementsCommute:

```
context <Element1Instance> inv:
  inv InstanceElementsCommute
    self.of_.<<name1>> =
    self.<<name2>>.of_
```

Description

The generalisable relatable semantics (GRS) template merges properties of containable, generalisable and instantiable modelling elements. Any element E that is related to other elements must guarantee to have instances that are related to instances corresponding to the elements of E. Furthermore, the parents of an instance should reflect the same generalisation hierarchy as the element.

E*xample*

A classic example is the relationship between an attribute and its type (a class), and their instances (slots and objects). Specialising an attribute must result in the type being specialised, and their instances must reflect the same generalisation hierarchy.

4.17. GeneralisableFeatureContainerSemantics

Summary

A generalisable feature container guarantees to have the same number of contained elements as it contained instances.

Definition

To be defined in the final submission.

Description

Example

4.18. Expression Semantics

Summary

Expressions can be evaluated against an instance to return a result.

Definition

To be defined in the final submission.

Description

E*xample*

4.19. Binding

Summary

Elements can be bound to instances.

Definition

To be defined in the final submission.

Description

E*xample*

4.20. Mapping

Summary

A mapping relates elements to elements.

Definition

To be defined in the final submission.

Description

E*xample*

4.21. Container Map

Summary

A container map relates containers to containers.

Definition

To be defined in the final submission.

Description

E*xample*

4.22. Refinement

Summary

A refinement is a relation between two containers, an abstraction and a realization, in which some guarantees of the abstraction are maintained by the realization with a suitable re-interpretation. The kind of refinement relation dictates the required mapping.

Definition

To be defined in the final submission.

Description

Refinement will include automated translations from platform-independent specificaiton models to platform-specific implementation models, as well as less automated refinement of specifications e.g. by decomposing objects or actions.

4.23. Graph

Summary

A graph is an arbitrary structure of connected nodes.

Definition

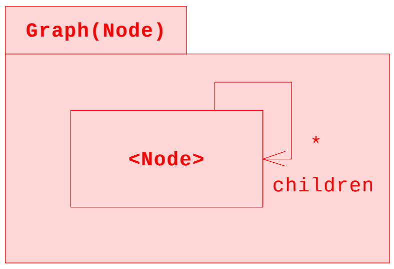


Figure 30. Graph

Description

A graph is an arbitrary structure of connected nodes. The children of a Node are its connectors to other nodes.

Example

In the context of a meta-model, one can readily visualise modelling elements such as classes, packages, attributes as nodes in a graph, where the children of each node represent the connections that exist between the modelling elements (associations, links, and so on).

4.24. Tree

Summary

A rooted directed acyclic graph with a single path from root to any other node.

Definition

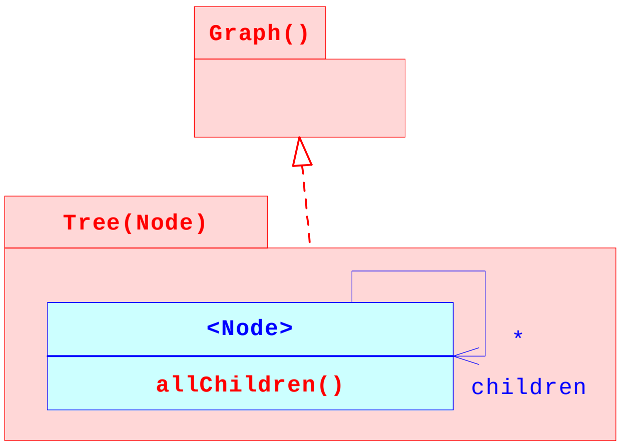


Figure 31. Tree

Queries

To be defined.

Description

A tree is a special kind of graph.

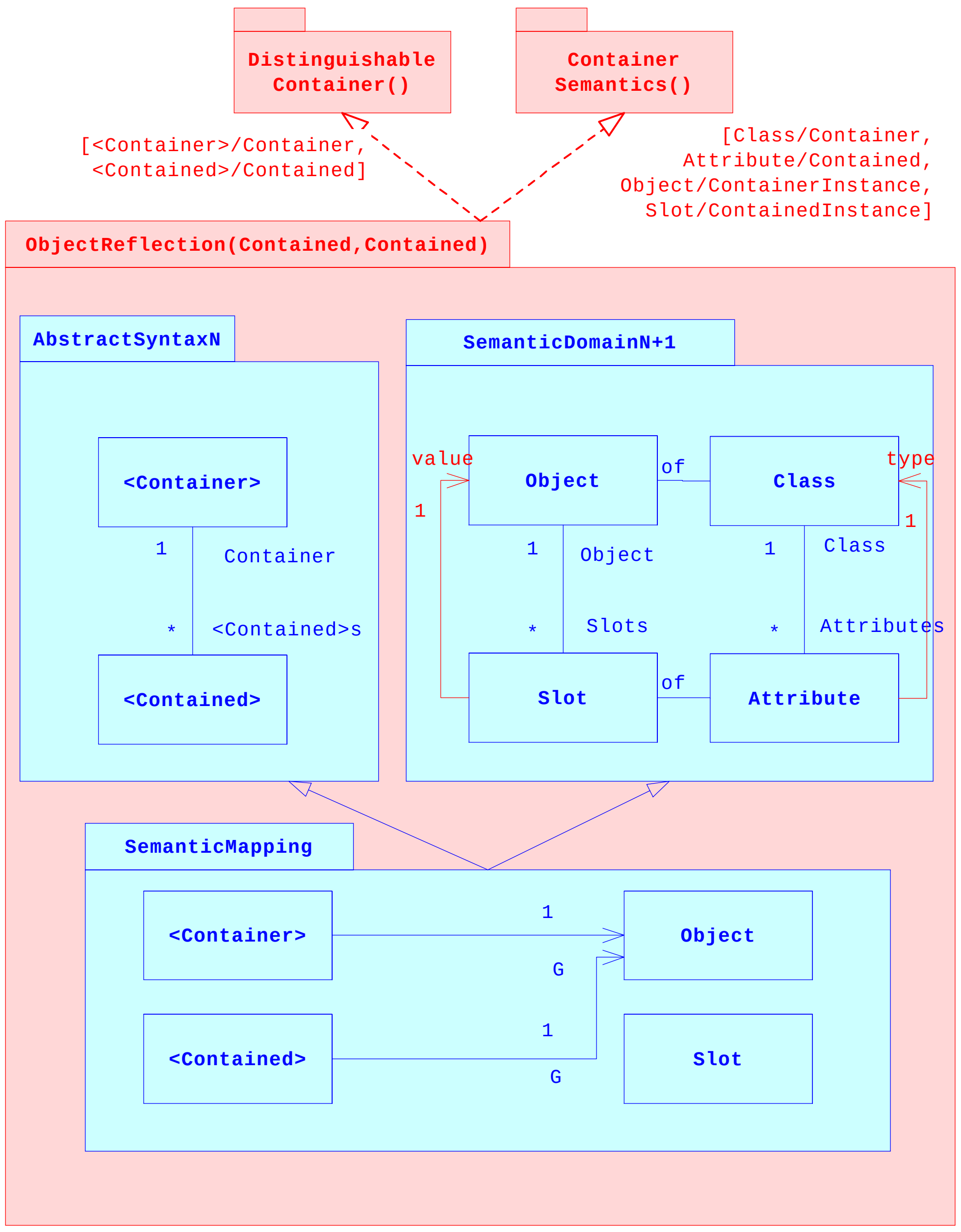
E*xample*

Tree structures are common in meta-models, typically where there is a containment hierarchy between elements, for example a package contains other packages, but cannot contain itself or be contained by any of its sub-packages.

4.25. Object Reflection

Summary

Defines a mapping, G, between containers and their object and slot representation at the next meta-level up.



Description

Every element at a level N in a layered meta-model architecture can be mapped to an object of class “element.name” at the Nth+1 level. This formalises the rules defined in [Alvarez01b]. Note, additional reflection templates will be required for other structures, e.g. generalisable elements, named elements.

Well-formedness Rules

<Container>sAreInstancesOfTheClass<Container>: A container is an instance of the class whose name is the same as the container.

```
context <Container> inv:
  <Container>sAreInstancesOfTheClass<Container>
  G.of.name = <Container>
```

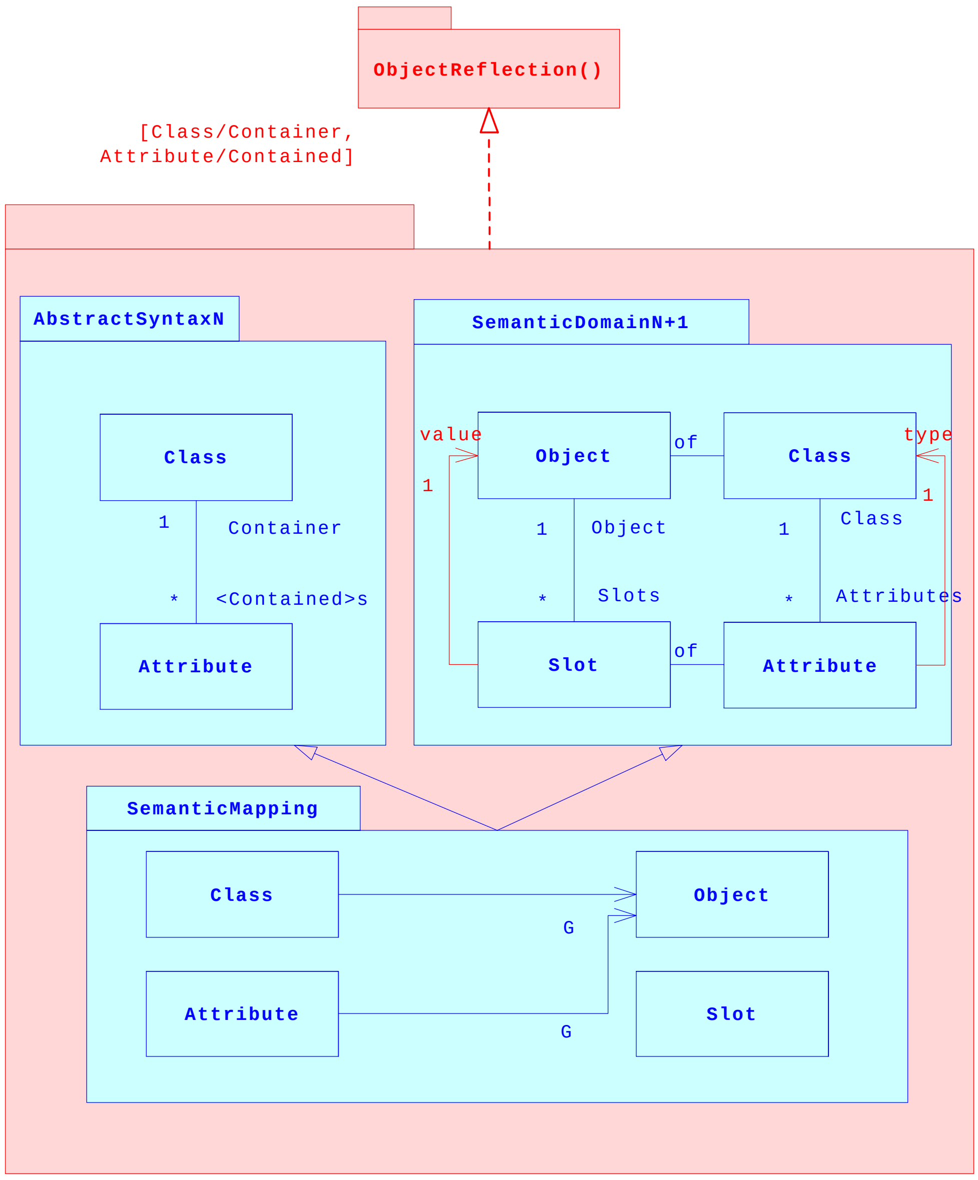
<Contained>sAreInstancesOfTheClass<Contained>: A contained element is an instance of the class whose name is the same as the contained element.

```
context <Contained> inv:
  <Contained>sAreInstancesOfTheClass<Contained>
  G.of.name = <Contained>
```

<Container>sAreObjectsWithSlots:

```
context <Container> inv:
  <Container>sAreObjectsWithSlots
  self.<Contained>s -> forAll(c | self.G.slots ->
    exists(s | s.of.name = <Contained>s and
                s.value = c.G))
```

Example



ClasssAreInstancesOfTheClassClass: A class is an instance of the class named “Class”.

```
context Class inv:
  ClasssAreInstancesOfTheClassClass
  G.of.name = Class
```

AttributesAreInstancesOfTheClassAttribute: An attribute is an instance of the class named “Attribute”.

```
context Attribute inv:
  AttributesAreInstancesOfTheClassAttribute
  G.of.name = Attribute
```

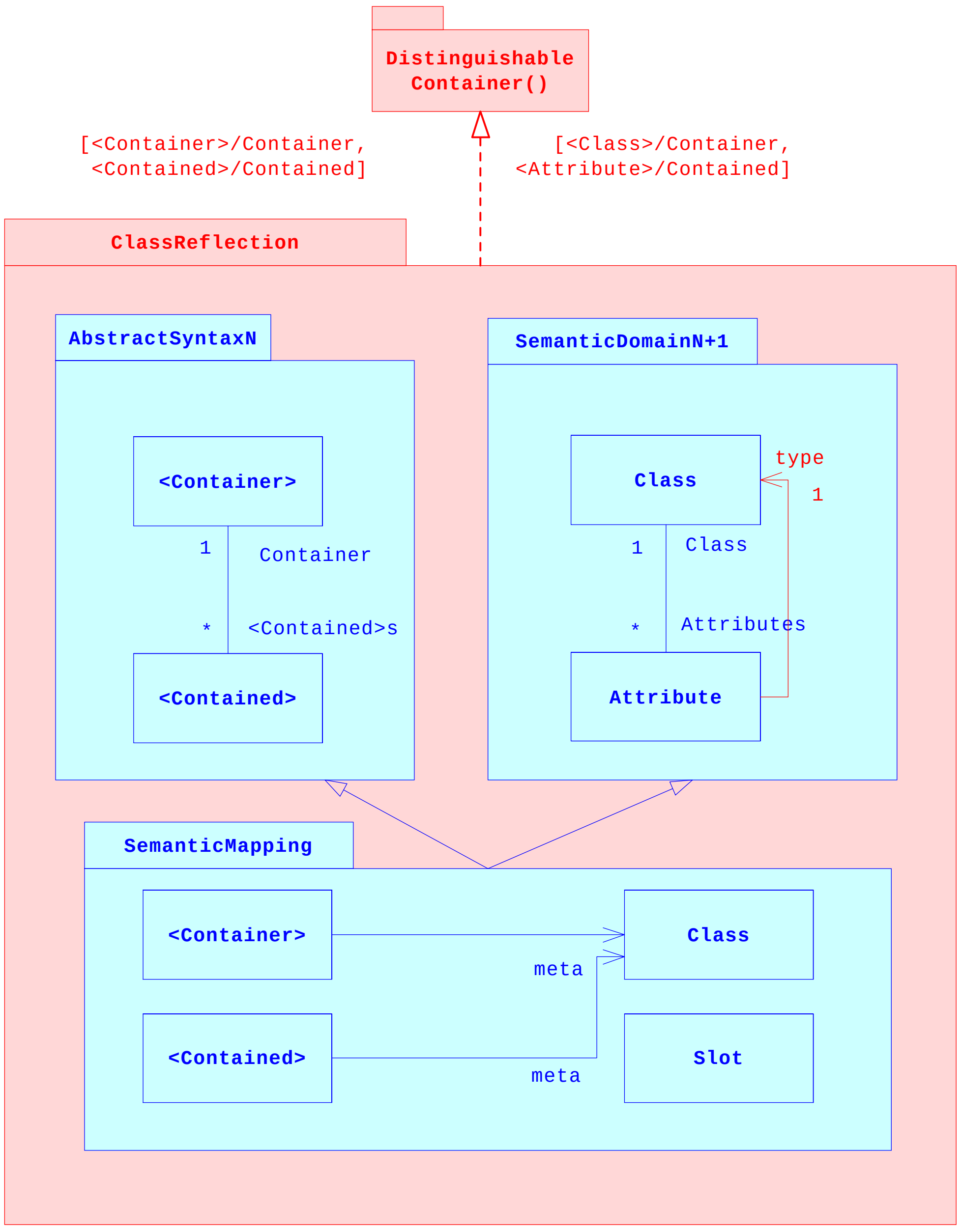
ClasssAreObjectsWithSlots:

```
context Class inv:
  ClasssAreObjectsWithSlots
  self.Attributes -> forAll(c | self.G.slots ->
    exists(s | s.of.name = “Attributes” and s.value = c.G))
```

4.26. Class Reflection

Summary

Defines a mapping between a container and their meta-class.



Description

Every element at a level N in a layered meta-model architecture can be viewed as a meta-instance of the class “element.name” at the Nth+1 level. This formalises the dashed arrow commonly used to describe the meta-instance relationship between model elements. It also formalises the approach used in [Atkinson01]. Note, additional reflection templates will be required for other structures, e.g. generalisable elements, named elements.

Well-formedness Rules

<Container>sAreInstancesOfTheMetaClass<Container>: A container is an instance of the class whose name is the same as the container.

```
context <Container> inv:
  <Container>sAreInstanceOfTheMetaClass<Container>
  meta.name = <Container>
```

<Contained>sAreInstancesOfTheMetaClass<Contained>: A contained element is an instance of the class whose name is the same as the contained element.

```
context <Contained> inv:
  <Contained>sAreInstancesOfTheMetaClass<Contained>
  meta.name = <Contained>
```

<Container>sHaveMetaClassesWithAttributes:

```
context <Container> inv
  <Container>sHaveMetaClassesWithAttributes
  self.<Contained>s -> forAll(c | self.meta.<Contained>s ->
    exists(s | s.name = <Contained>s and
                s.type = c.meta))
```

4.27. Abstract Container

Summary

A containment relationship between elements that specialises an abstract Container class. Abstract Container is a reified version of the Tree template (see Section 4.24.).

Definition

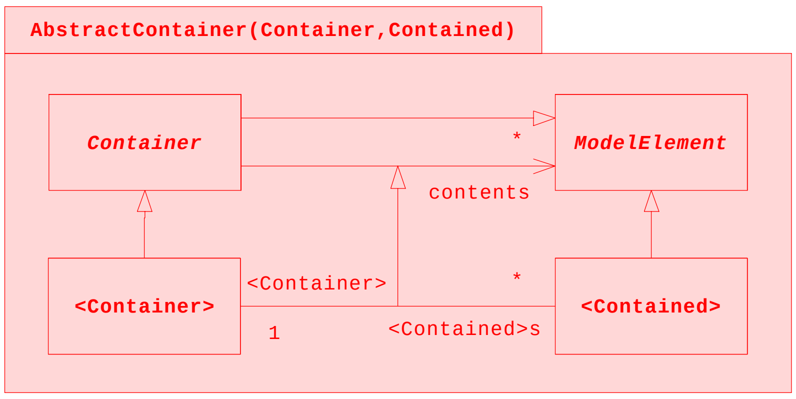


Figure 32. AbstractContainer

Queries

nestedContents() : returns the set of nested contents of a container

```
context Container
  nestedContents() : Set(Element)
    self.contents -> iterate(element S = Set |
      S -> union(element.nestedContents()))
```

Well-formedness Rules

NoCircularContainment: a container cannot contain itself directly or indirectly

```
context Container
  NoCircularContainment inv:
    not(self.nestedContents() -> includes(self))
```

Description

This template makes explicit the relationship between the Container template (see Section 4.2.) and the traditional framework based approach in UML 1.X, whereby concrete container classes specialise a single abstract class. A benefit of this approach is that global constraints on relationships across containers can be easily described. For example, a container forms part of a directed acyclic graph - a container must therefore not contain itself or be contained by any of its containers. This property can be described by a no circular containment constraint on the abstract container class.

Example

A class contains queries and attributes. The no circular containment rules ensures that an attribute or method cannot contain their class.

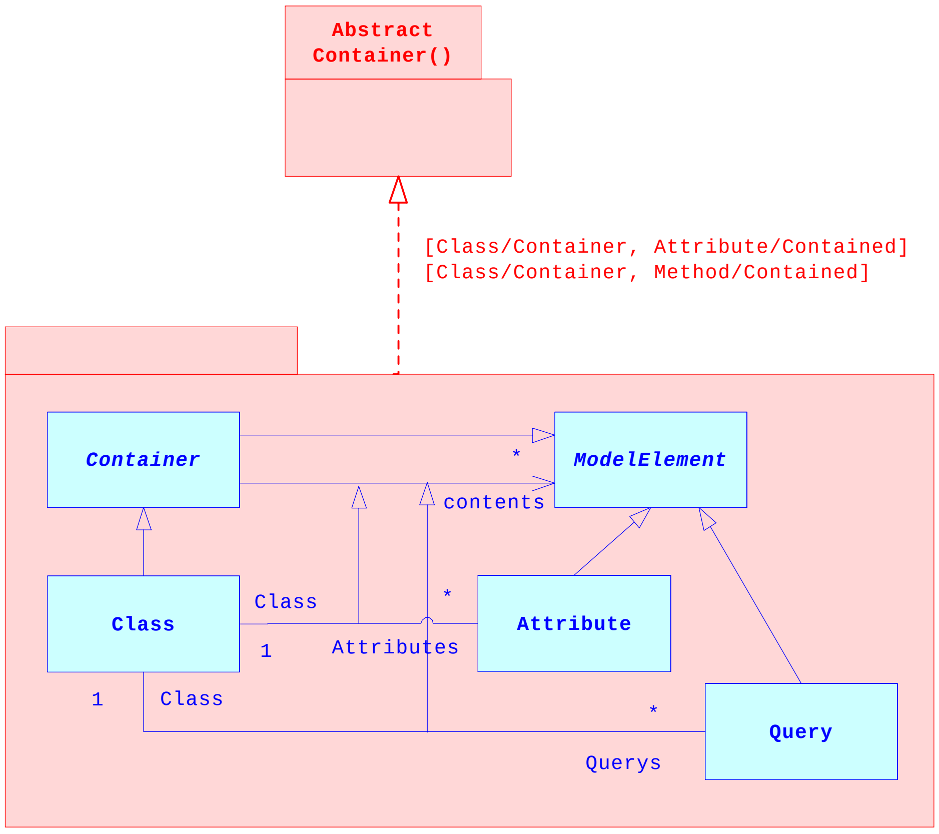


Figure 33. AbstractContainer Example

4.28. Abstract GeneralisableElement

Summary

A generalisable element that specialises an abstract GeneralisableElement class ala UML 1.X.

Definition

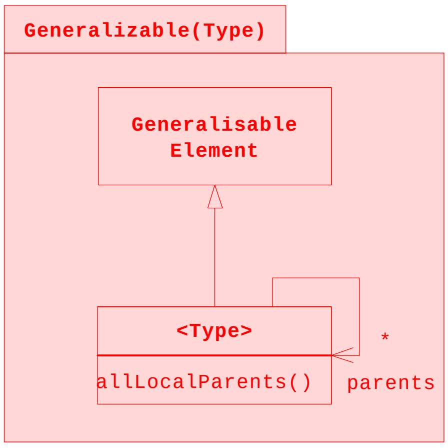


Figure 34. AbstractGeneralisableElement

Queries

None.

Well-formedness Rules

None.

Description

This template makes explicit the relationship between the Generalisable template (see Section 4.6.) and the traditional framework based approach in UML 1.X, whereby concrete generalisable classes specialise a single abstract class.

Example

None.

4.29. Abstract NameSpace

Summary

A NameSpace template that specialises an abstract NameSpace class ala UML 1.X.

Definition

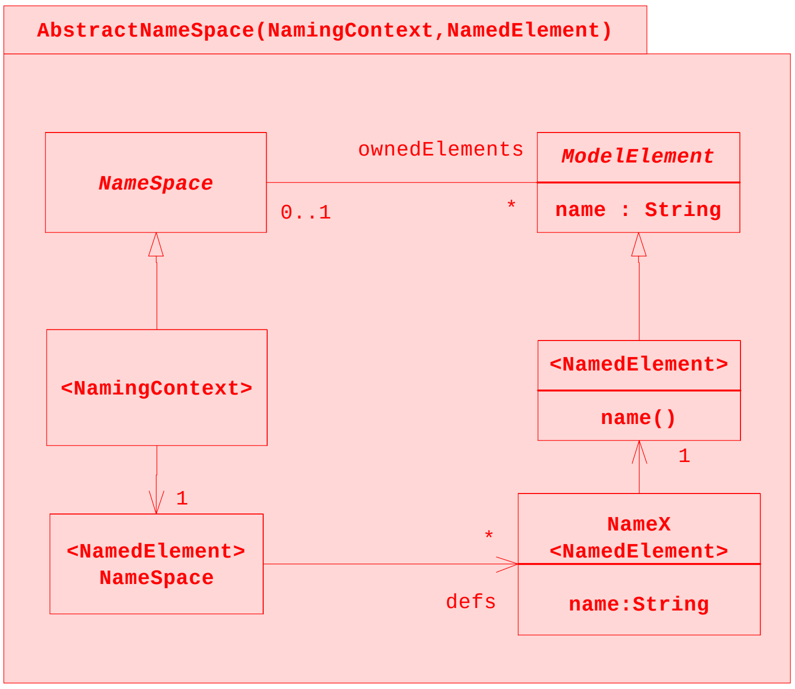


Figure 35. AbstractNameSpace

Queries

As for Section 4.3 on page48.

Well-formedness Rules

NamedElementsAreOwnedElements : A <NamingContext>'s NameSpace contains names for all its owned elements.

```
context <NamingContext> inv:
  NamedElementsAreOwnedElements
  self.ownedElements ->
    includesAll(self.<NamedElement>NameSpace.ran())
```

Description

This template makes explicit the relationship between the NameSpace template (see Section 4.3.) and the traditional framework based approach used in UML 1.X, whereby concrete namespace classes specialise a single abstract class. Unfortunately, UML 1.X confuses containment with namespace, even though they are orthogonal concepts (hence the association “ownedElement” on the abstract class), which means that this model is likely to need updating.

Example

None.

Chapter 5

Static Core

This chapter describes the core aspects of UML that are necessary to describe static models of systems: classes, packages, associations, datatypes, expressions, constraints, queries, templates and reflection.

5.1. Introduction

This chapter defines the core aspects of UML that are necessary to describe static models of systems. It is structured as follows: an overview of the package structure is given, followed by a description of the sub-packages. Each sub-package addresses a different aspect of static modelling. Each sub-package will consist of a description of concrete syntax, abstract syntax and semantics. The presentation of semantics will depend on the sub-package. For example a semantic domain and semantic mapping may be used; alternatively the syntax may be defined as sugar using a syntax to syntax mapping.

The components of the static core package are shown in Figure 36. In order to make the definition manageable and adaptable, a layered, extensible architecture has been adopted. Packages are used to separate out different concepts in the language. Package generalisation is then used to combine and extend these concepts in a logically consistent manner.

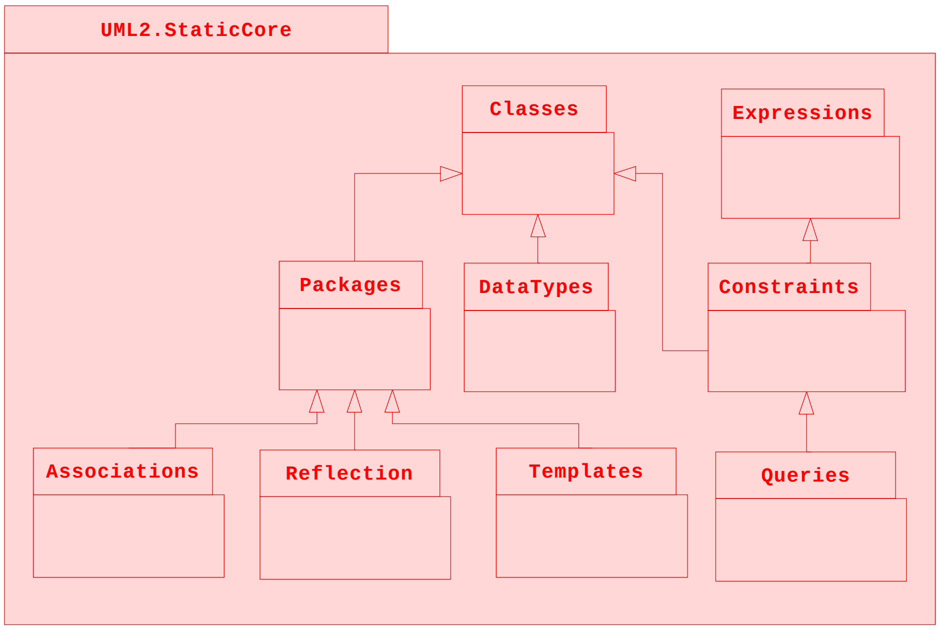


Figure 36. Package Structure of UML.Static Package

A brief description of each package follows:

Classes: this package provides a definition of classes and attributes, but not queries, which are dealt with in the Queries package.

Packages: general mechanisms for managing collections of UML elements. The primary extension mechanism provided is package generalisation.

Associations: contains a definition of UML associations.

Datatypes: a package of UML data types, including basic data types such as integers and strings. It also includes collection types, such as sets, sequences and bags.

Expressions: a collection of expression primitives that support a family of expression languages.

Constraints: constructs relating to the expression of constraints. It defines a minimal constraint language for UML that is similar to OCL, but which has a precise metamodel semantics.

Queries: a definition of static methods or queries. A static method has parameters and a return type, but no side effects.

Templates: a definition of package templates, which are the foundation of our metamodelling approach.

Reflection: a definition of meta-levels and meta-instances. It provides a precise definition of what it means for a model to be a meta-instance of another model, a pre-requisite for defining the meaning of reflection and layered meta-models in the UML.

As described in the introductory chapters, each component of the static core is further divided into a strict pattern of abstract syntax, semantic domain and semantics packages. The abstract syntax package describes the modelling concepts defined in a component, the semantic domain package describes the semantic elements which are denoted by the modelling concepts. A mapping from each concept to its semantic domain is given in the semantics package, which extends both packages.

The remainder of this chapter gives a detailed description of the contents of these packages, thus providing a definition of the semantics of the core static language. In each case, a description is given of how each package of concepts has been generated using the package templates described in chapter 4.

Details of the syntax packages will be left until the final submission.

5.2. Classes

5.2.1 Overview

This package defines the essential abstract syntax and semantics of classes. Classes are key modelling concepts in the UML. They describe the structure of objects in terms of attributes and queries. Classes also support the notion of generalisation: the ability to reuse structural definitions from one class (the parent, or super-class) in another (the child, or sub-class).

5.2.2 Templates

Figure 37 shows the templates used to “stamp out” the Classes package (see chapter 4 for a full description)

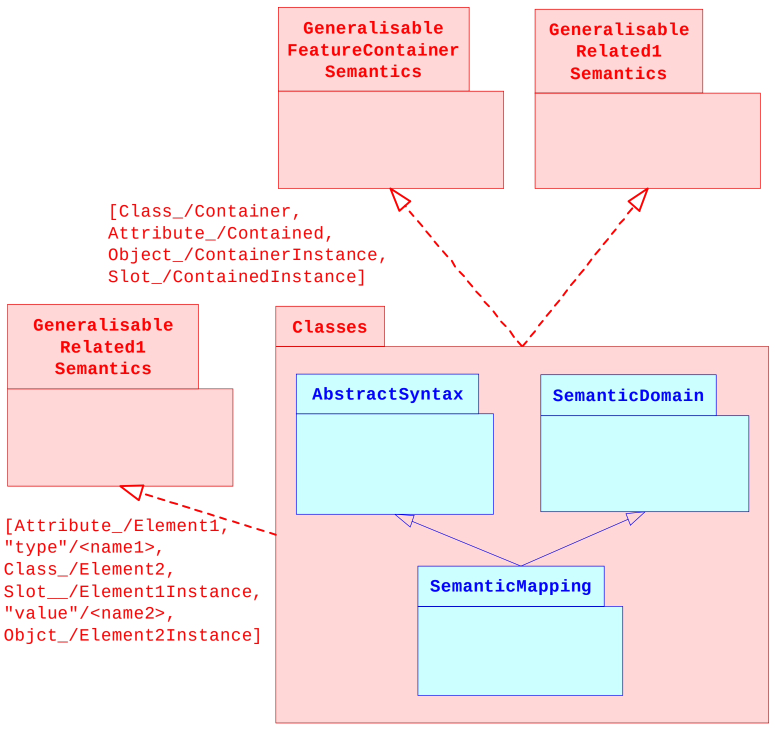

Generalisable FeatureContainer Semantics
Generalisable Related1 Semantics
[Class_/Container, Attribute_/Contained, Object_/ContainerInstance, Slot_/ContainedInstance]
Generalisable Related1 Semantics
Classes
AbstractSyntax
SemanticDomain
[Attribute_/Element1, "type"/<name1>, Class_/Element2, Slot__/Element1Instance, "value"/<name2>, Objct_/Element2Instance]
SemanticMapping

Figure 37. Templates used in the Classes Package

Classes have *namespaces* for the things they contain. Classes are *containers* of attributes. Classes are *generalisable*. Attributes are *related* to their types and are also *generalisable*. The *semantics* of classes are described by their instances. Instances of classes are objects; instances of attributes are slots. Objects are *namespaces* for their slots. Objects are *generalisable*. Slots are *related* to their values and are also *generalisable*.

5.2.3 Abstract Syntax

Figure 38 defines a model for classes. Each class defines a number of locally defined attributes.

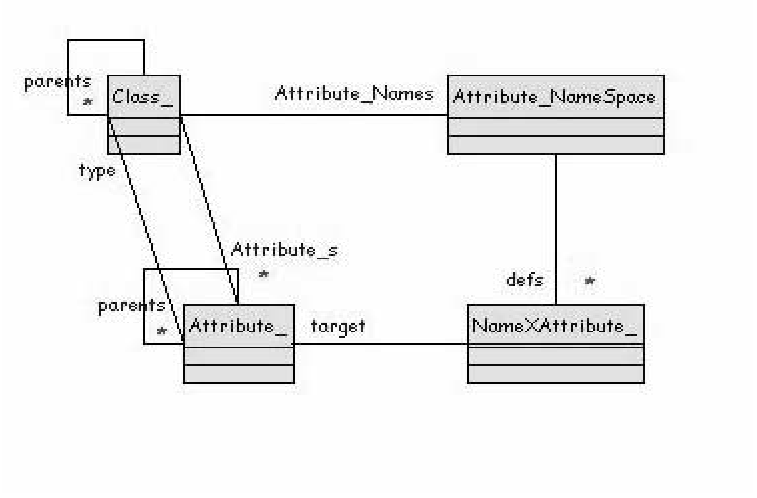


Figure 38. UML2.Static.Classes.AbstractSyntax

Further attributes are inherited from the parents of the class. Each class contains a name space for the attributes. Each name space contains a set of definition pairs associating each local attribute with a name.

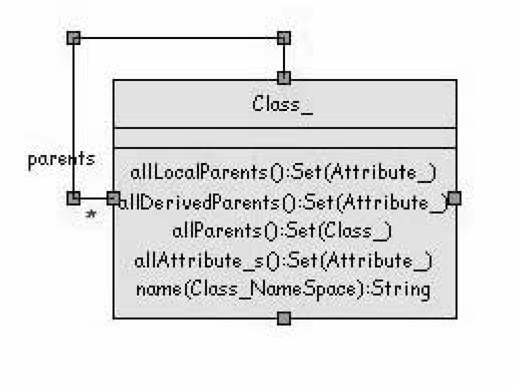

Class_
parents
*
allLocalParents():Set(Attribute_)
allDerivedParents():Set(Attribute_)
allParents():Set(Class_)
allAttribute_s():Set(Attribute_)
name(Class_NameSpace):String

Figure 39. Class Queries

Figure 39 shows the definition of class queries. The name of the class can be requested with respect to a given name space. The attributes which are both defined and inherited by the class are given by Class_::allAttribute_s(). The locally defined parents of a class are given by Class_::parents. The transitive closure of this relationship is Class_::allLocalParents().

Well-formedness Rules

[1] All attributes in a class’s namespace have different names.

```
context Class_ inv
  AllAttribute_sHaveDistinctNames
    self.Attribute_s->forAll(element, element’ |
      element.name(self.Attribute_Names) =
      element'.name(self.Attribute_Names) implies
        element = element'))
```

[2] The parents of an attribute’s type are related to the attribute’s parents

```
context Class_ inv:
  RelatedParentsCommute
    self.type.parents = self.parents ->
    iterate(p s = Set{} | s->including(p.type))
```

Queries

[1] Returns all attributes of a class including its local attributes and those derived from its parents (except those with the same name as the class’s local attributes or those that are redefined).

```
context Class_
  allAttribute_s() : Set(Attribute_)
    self.allParents() ->
      iterate(p s = self.Attribute_s |
        s->union(p.allAttribute_s()->
          reject(c | s->exists(c' |
            c.name(self.Attribute_Names) =
            c'.name(self.Attribute_Names) or
            or c'.parents ->includes(c)))))
```

[2] Returns all the derived parents of an attribute. A parent is derived if the container of the attribute (i.e. its class) or any of the container’s parents contain an attribute with the same name.

```
context Attribute_
  allDerivedParents() : Set(Attribute_)
    if self.Class_ <> self then
      self.Class_.allParents() -> iterate(p s = Set{} |
        s->union(p.Attribute_s -> select(c |
          c.name(self.Attribute_Names) =
          self.name(self.Attribute_Names))))
      else Set{} endif
```

[3] Returns all parents of an attribute, including its local and derived parents.

```
context Attribute_
  allParents():Set(Attribute_)
    self.allLocalParents()->union(self.allDerivedParents());
```

[4] Returns all local parents of an attribute.

```
context Attribute_
  allLocalParents() : Set(Attribute_)
    self.parents->iterate(parent S = self.parents |
      S->union(parent.allLocalParents()))
```

Example

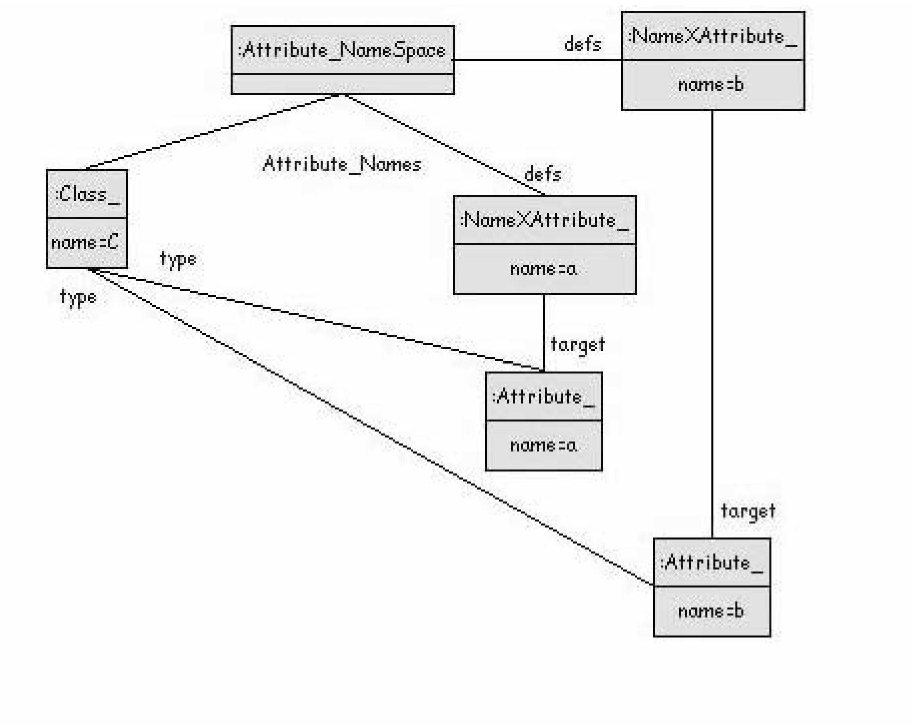


Figure 40. A class with two attributes

Figure 40 shows the definition of a single class named C with two attributes named a and b respectively.

5.2.4 Semantic Domain

The classes semantic domain package is shown in Figure 41. Classes denote objects. Each object is an instance of a class. Each object contains a collection of slots which is referred to as its *state.* A slot has a name and a value. The value of a slot is an object.

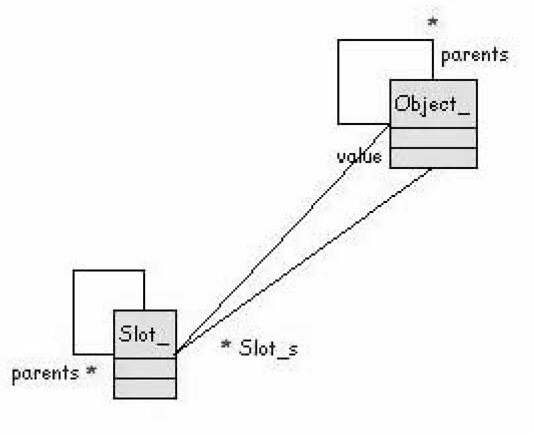


Figure 41. UML2.Static.Classes.SemanticDomain

Both objects and slots have parents. This structure reflects the parent relationships between classes and attributes respectively. An object has local slots and inherited slots which correspond exactly to the local and inherited attributes of its class. In this way a semantic domain element that is an instance of a class C may easily be viewed as an instance of a super-class of C by selecting the appropriate super-object. This structuring facilitates varieties of object-oriented polymorphism and the notion of *run-super* as pioneered by Smalltalk.

Well-formedness Rules

[1] All slots in a object's namespace have different names.

```
context Object_ inv
  AllSlot_sHaveDistinctNames
    self.Slot_s->forAll(element, element’ |
      element.name(self.Slot_Names) =
      element'.name(self.Slot_Names) implies element = element'))
```

[2] The parents of a slot's value are related to the slot's parents

```
context Slot_ inv:
  RelatedParentsCommute
    self.value.parents = self.parents ->
    iterate(p s = Set{} | s->including(p.value))
```

Queries

[1] Returns all slots of an object including its local slots and those derived from its parents (except those with the same name as the object's local slots or those that are redefined).

```
context Object_
  allSlots() : Set(Slot_)
    self.allParents() ->
      iterate(p s = self.Slot_s |
        s->union(p.allSlot_s()->
          reject(c | s->exists(c' |
            c.name(self.Slot_Names) =
            c'.name(self.Slot_Names) or
            c'.parents ->includes(c)))))
```

[2] Returns all the derived parents of an slot. A parent is derived if the container of the slot (i.e. its object) or any of the container's parents contain a slot with the same name.

```
 context Slot_
  allDerivedParents() : Set(Slot_)
    if self.Object_ <> self then
      self.Object_.allParents() -> iterate(p s = Set{} |
        s->union(p.Slot_s -> select(c |
          c.name(self.Slot_Names) =
          self.name(self.Slot_Names))))
      else Set{}
    endif
```

[3] Returns all parents of a slot, including its local and derived parents.

```
context Slot_
  allParents():Set(Slot_)
    self.allLocalParents()->union(self.allDerivedParents());
```

[4] Returns all local parents of a slot.

```
context Slot_
  allLocalParents() : Set(Slot_)
    self.parents->iterate(parent S = self.parents |
      S->union(parent.allLocalParents()))
```

5.2.5 Semantic Mapping

The semantic mapping package for classes (see Figure 42) defines all possible legal configurations of objects and slots for a class and its attributes. In other words, given a candidate object, the semantic mapping package tells us, for any given class whether or not the object is a legal instance of the class.

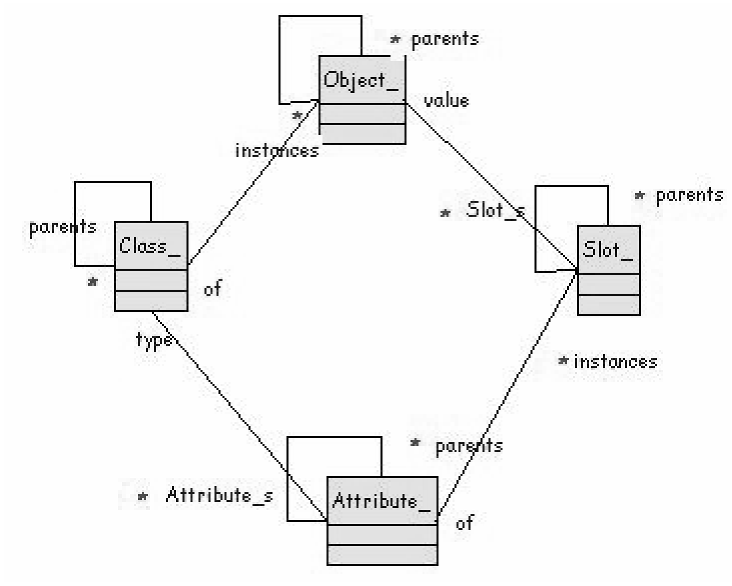


Figure 42. UML2.Static.Classes.SemanticMapping

In order for an object to be an instance of a class the object must define slots for all the local attributes of the class, the slot values must be instances of attribute types (or a super-class thereof) and the super-objects must be instances of appropriate super-classes.

Well-formedness Rules

[1] Every slot belonging to an object is an instance of the object's class's attributes.

```
context Object_
  inv InstanceContentsCommute
    self.of_.Attribute_s =
      self.Slots -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[2] An object contains the same number of slots as the number of attributes contained by its class.

```
context Object_
  inv SameNumberOfElementsAsContainer
    self.Slots -> size =
      self.of_.Attribute_s -> size
```

[3] A slot’s parents are instance’s of its attribute’s parents.

```
    context Slot_
      self.of_.parents =
      self.parents -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[4] The value of a slot is an instance of it’s attribute’s type.

```
  context Slot_ inv:
    inv InstanceElementsCommute
    self.of_.type =
    self.value.of_
```

Example

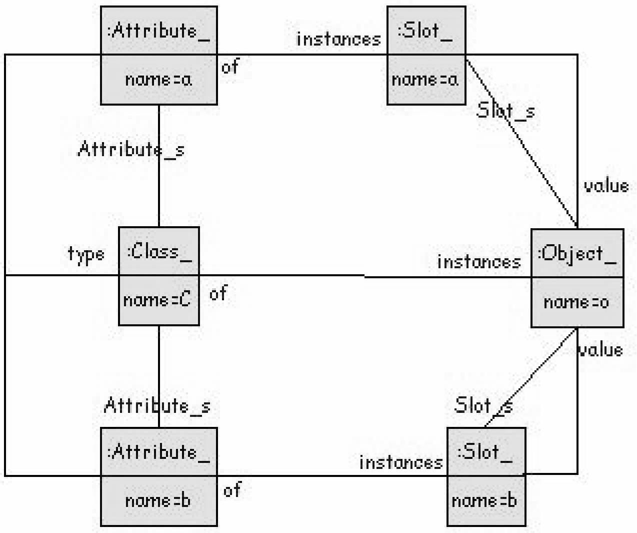


Figure 43. A class and an instance

Figure 43 shows an example of class and a legal instance. Because the class defines two attributes named a and b its instance must have two slots with the same names. The values of the slots must be instances of the types of the attributes (for simplicity we reuse the type C and the value o)

Figure 44 shows a class D with a single super-class C. Class C defines one attribute named a and D defines an attribute named b. D has a single instance with a slot for b and a parent link to an instance of C with slot for a. The instance of D therefore has two slots named a and b; the structure of the instance of D reflects its ability to be thought of as an instance of C.

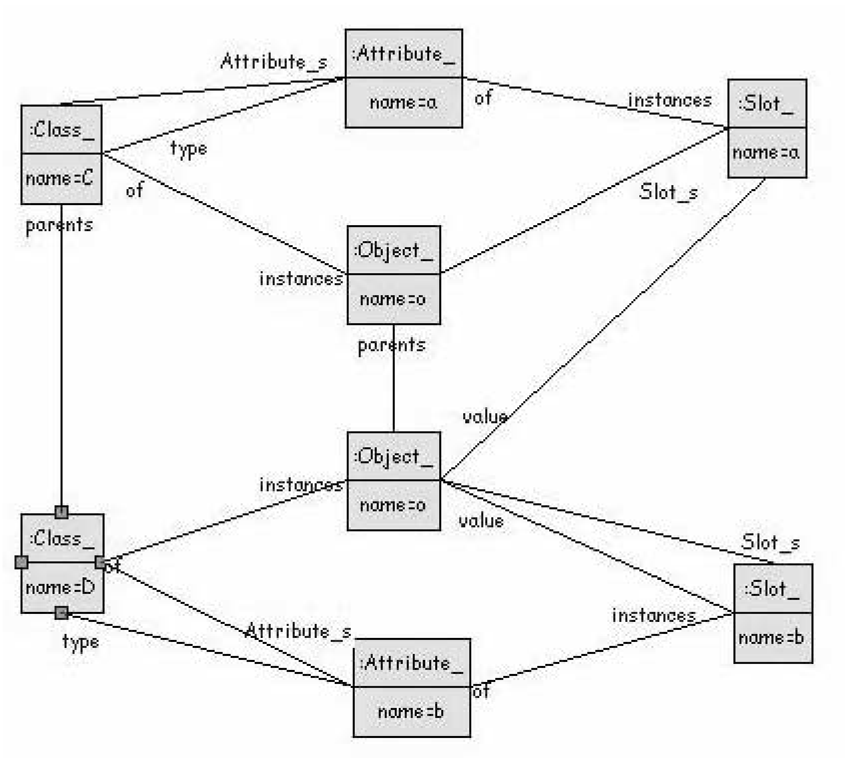


Figure 44. A class with parents and instances.

5.3. Packages

5.3.1 Overview

Packages provide a way of grouping together and managing related parts of a model. In the static core, packages can contain sub-packages, associations and classes. Package extension is the basic mechanism for reusing models. A package may reuse another package by specialising it. In this case, the contents of the generalised package are also available in the specialised package, and may also be extended.

5.3.2 Templates

Figure 45 shows the templates used to “stamp out” the Packages package (see chapter 4 for full descriptions)

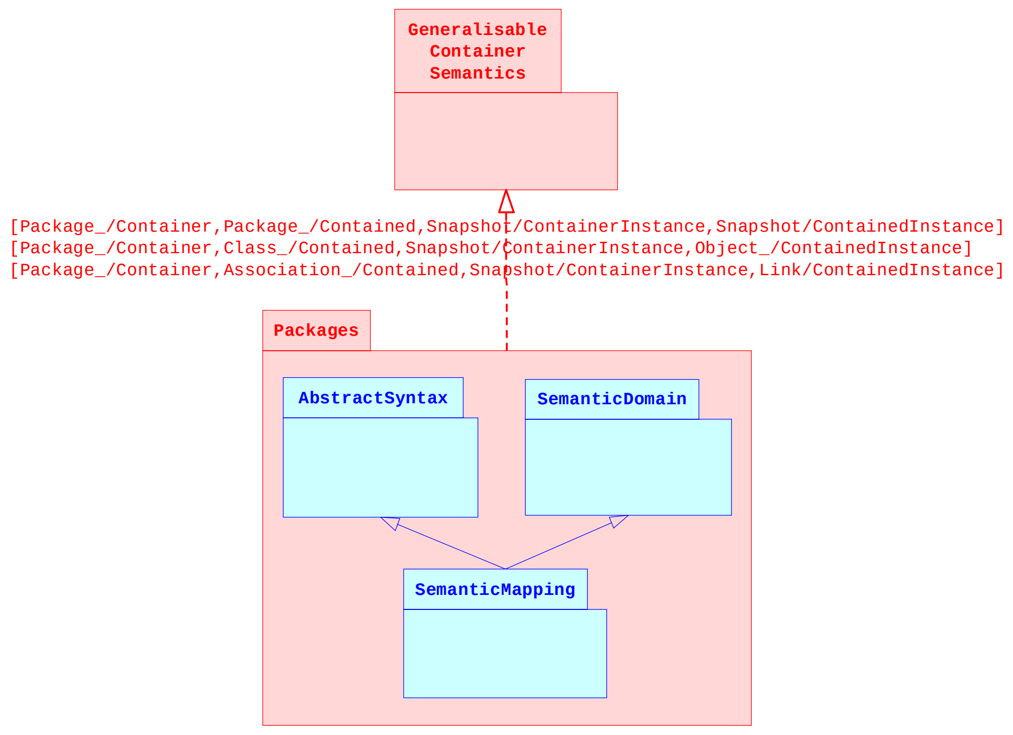

Generalisable Container Semantics
[Package_/Container,Package_/Contained,Snapshot/ContainerInstance,Snapshot/ContainedInstance]
[Package_/Container,Class_/Contained,Snapshot/ContainerInstance,Object_/ContainedInstance]
[Package_/Container,Association_/Contained,Snapshot/ContainerInstance,Link/ContainedInstance]
Packages
AbstractSyntax
SemanticDomain
SemanticMapping

Figure 45. Templates used in the Packages package

Packages are *containers* of classes, associations and sub-packages, and have *name-spaces* for the things they contain. The *semantics* of packages are described by their instances. Instances of packages are snapshots. A snapshot is a *container* of objects, links and sub-snapshots and has *namespaces* for its contents. Links are instances of associations; classes are instances of objects, sub-snapshots are instances of sub-packages. Packages and snapshots are *generalisable*.

5.3.3 Abstract Syntax

Figure 46 defines the abstract syntax package for packages. Each package contains a number of locally defined classes, associations and sub-packages.

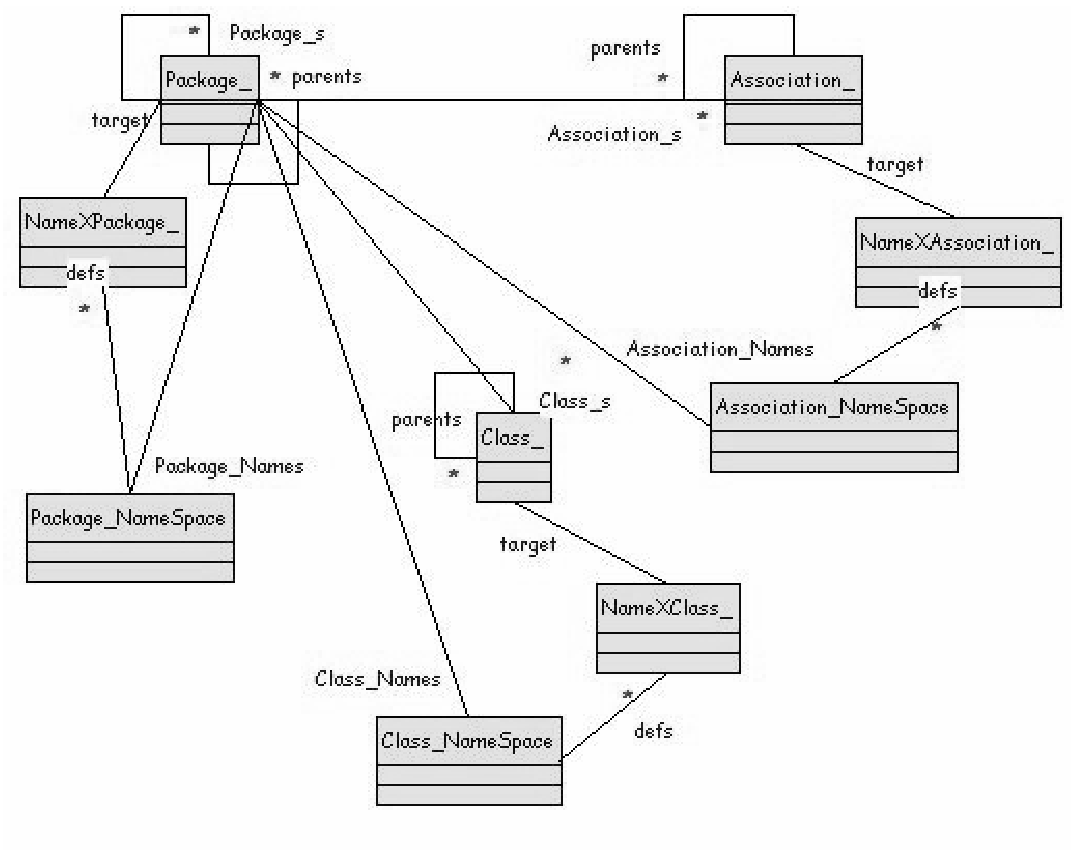


Figure 46. UML2.Static.Packages.AbstractSyntax

Further classes, associations and sub-packages can be inherited from the parents of the package. Each package contains a name space for its classes, associations and sub-packages. Each name space contains a set of name/element pairs associating each contained element with a name.

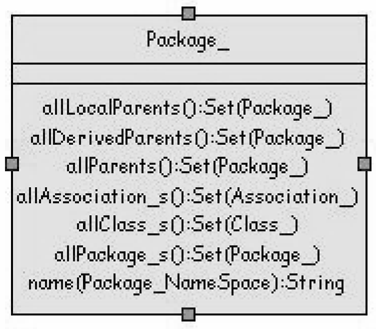

Package_
allLocalParents():Set(Package_)
allDerivedParents():Set(Package_)
allParents():Set(Package_)
allAssociation_s():Set(Association_)
allClass_s():Set(Class_)
allPackage_s():Set(Package_)
name(Package_NameSpace):String

Figure 47. Package Queries

Figure 47 shows the definition of package queries. The name of the package can be requested with respect to a package name space. The classes which are both defined and inherited by the package are given by Package_::allClass_s(). identical operations exist for associations and sub-packages. The locally defined parents of a package are given by Package_::parents. The transitive closure of this relationship is Package_::allLocalParents().

Well-formedness Rules

[1] All classes in a package’s namespace have different names.

```
context Package_ inv
  AllClass_sHaveDistinctNames
    self.Class_s->forAll(element, element’ |
      element.name(self.Class_Names) =
      element'.name(self.Class_Names) implies
        element = element'))
```

[2] All associations in a package’s namespace have different names.

```
context Package_ inv
  AllAssociation_sHaveDistinctNames
    self.Class_s->forAll(element, element’ |
      element.name(self.Association_Names) =
      element'.name(self.Association_Names) implies
        element = element'))
```

[2] All sub-packages in a package’s namespace have different names.

```
context Package_ inv
  AllPackage_sHaveDistinctNames
    self.Package_s->forAll(element, element’ |
      element.name(self.Package_Names) =
      element'.name(self.Package_Names) implies
        element = element'))
```

Queries

[1] Returns all classes contained in a package including its local classes and those derived from its parents (except those with the same name as the package’s local classes or those that are redefined).

```
context Package_
  allClass_s() : Set(Class_)
    self.allParents() ->
      iterate(p s = self.Class_s |
        s->union(p.allClass_s()->
          reject(c | s->exists(c' |
            c.name(self.Class_Names) =
            c'.name(self.Class_Names) or
            c'.parents ->includes(c)))))
```

[2] Returns all associations contained in a package including its local associations and those derived from its parents (except those with the same name as the package's local associations or those that are redefined).

```
context Package_
  allAssociation_s() : Set(Association_)
    self.allParents() ->
      iterate(p s = self.Association_s |
        s->union(p.allAssociation_s()->
          reject(c | s->exists(c' |
            c.name(self.Association_Names) =
            c'.name(self.Association_Names or
            c'.parents ->includes(c))))))
```

[3] Returns all sub-packages contained in a package including its local sub-packages and those derived from its parents (except those with the same name as the package's local sub-packages or those that are redefined).

```
context Package_
  allPackage_s() : Set(Package_)
    self.allParents() ->
      iterate(p s = self.Package_s |
        s->union(p.allPackage_s()->
          reject(c | s->exists(c' |
            c.name(self.Package_Names) =
            c'.name(self.Package_Names or
            c'.parents ->includes(c))))))
```

[4] Returns all the derived parents of a class. A parent is derived if the container of the class (i.e. its package) or any of the container's parents contain a class with the same name.

```
 context Class_
  allDerivedParents() : Set(Class_)
    if self.Package_ <> self then
      self.Package_.allParents() -> iterate(p s = Set{} |
        s->union(p.Class_s -> select(c |
          c.name(self.Class_Names) =
          self.name(self.Class_Names))))
      else Set{}
    endif
```

Identical queries are generated for Association_ and Package_.

[5] Returns all parents of a class, including its local and derived parents.

```
context Class_
  allParents():Set(Class_)
    self.allLocalParents()->union(self.allDerivedParents())
```

Identical queries are generated for Association_ and Package_.

[6] Returns all local parents of a class.

```
context Class_
  allLocalParents() : Set(Class_)
    self.parents->iterate(parent S = self.parents |
      S->union(parent.allLocalParents()))
```

Identical queries are generated for Association_ and Package_.

Example

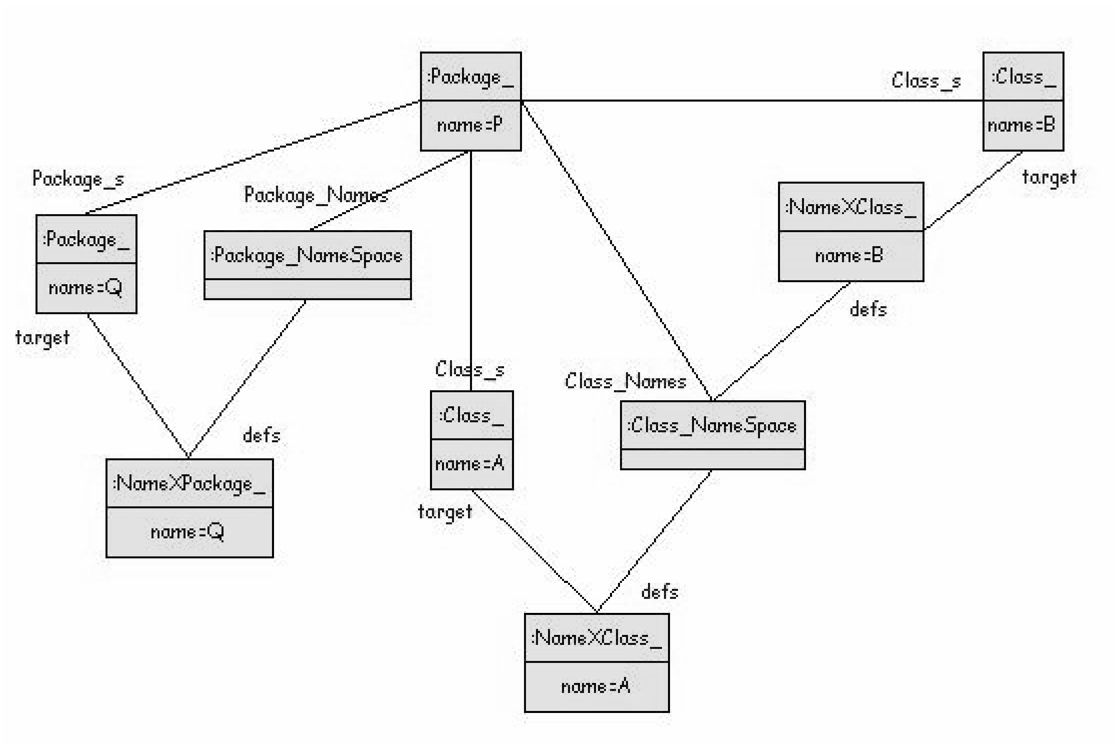


Figure 48. A package with a sub-package and classes

Figure 48 shows an example of a single package named P with a sub-package Q and classes named A and B respectively.

5.3.4 Semantic Domain

Packages denote snapshots. Each snapshot is an instance of a package. Each snapshot has a collection of objects, links and sub-snapshots. An object is an instance of a class and contains slots (see section 5.2). A link is an instance of an association and contains linkends (see section 5.4). The semantic domain package is shown in Figure 49.

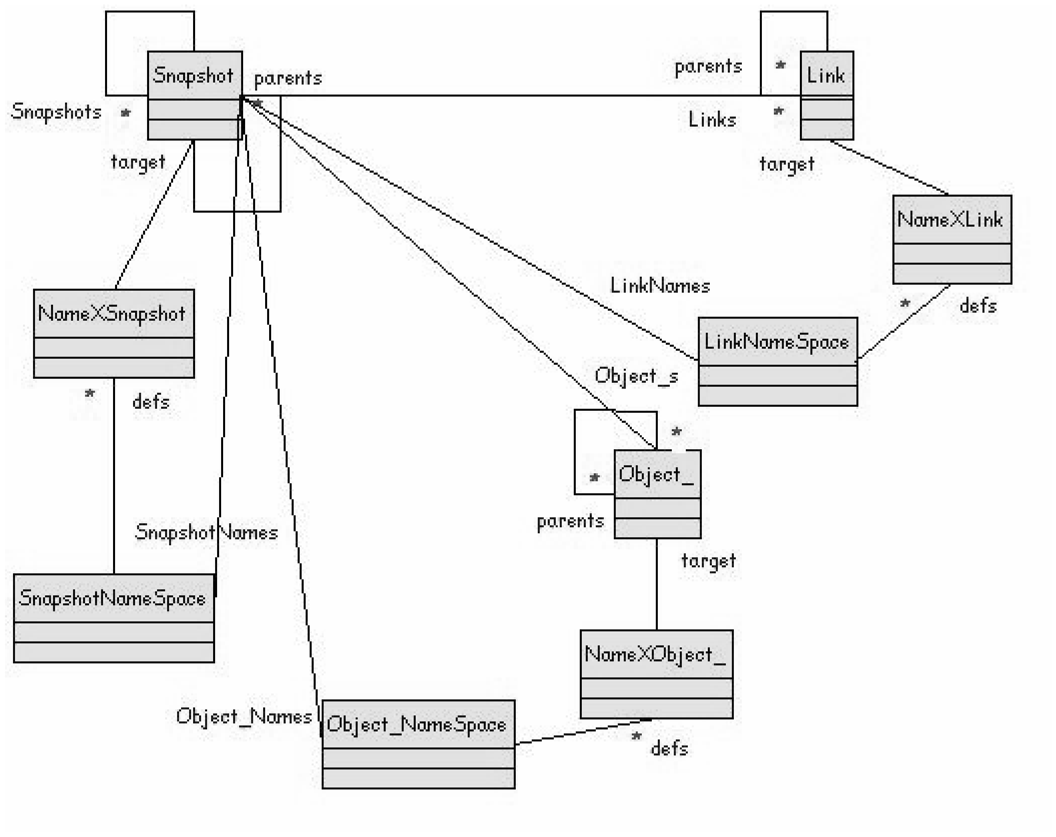


Figure 49. UML2.Static.Packages.SemanticDomain

Snapshots have parents and contain objects, links and sub-snapshots, which themselves may have parents. This structure reflects the parent relationships between packages and their classes, associations and sub-packages. A snapshot has local objects, links and sub-snapshots and inherited objects, links and snapshots which correspond exactly to the local and inherited contents of its package. In this way a semantic domain element that is an instance of a package P may easily be viewed as an instance of a super-class of P by selecting the appropriate super-snapshot.

Well-formedness Rules

[1] All objects in a snapshot’s namespace have different names.

```
context Snapshot inv
  AllObject_sHaveDistinctNames
    self.Object_s->forAll(element, element’ |
      element.name(self.Object_Names) =
      element'.name(self.Object_Names) implies
        element = element'))
```

[2] All links in a snapshot’s namespace have different names.

```
context Snapshot inv
  AllLinksHaveDistinctNames
    self.Object_s->forAll(element, element’ |
      element.name(self.LinkNames) =
      element'.name(self.LinkNames) implies
        element = element'))
```

[2] All sub-snapshots in a snapshot’s namespace have different names.

```
context Snapshot inv
  AllSnapshotsHaveDistinctNames
    self.Snapshots->forAll(element, element’ |
      element.name(self.SnapshotNames) =
      element'.name(self.SnapshotNames) implies
        element = element'))
```

Queries

[1] Returns all objects contained in a snapshot including its local objects and those derived from its parents (except those with the same name as the snapshot’s local objects or those that are redefined).

```
context Snapshot
  allObject_s() : Set(Object_)
    self.allParents() ->
      iterate(p s = self.Object_s |
        s->union(p.allObject_s()->
          reject(c | s->exists(c' |
            c.name(self.Object_Names) =
            c'.name(self.Object_Names) or
            c'.parents ->includes(c)))))
```

[2] Returns all links contained in a snapshot including its local links and those derived from its parents (except those with the same name as the snapshot’s local links or those that are redefined).

```
context Snapshot
  allLinks() : Set(Link)
    self.allParents() ->
      iterate(p s = self.Links |
        s->union(p.allLinks()->
          reject(c | s->exists(c' |
            c.name(self.LinkNames) =
            c'.name(self.LinkNames or
            c'.parents ->includes(c))))))
```

[3] Returns all sub-snapshots contained in a snapshot including its local sub-snapshots and those derived from its parents (except those with the same name as the snapshot’s local sub-snapshots or those that are redefined).

```
context Snapshot
  allSnapshots() : Set(Snapshot)
    self.allParents() ->
      iterate(p s = self.Snapshots |
        s->union(p.allSnapshots()->
          reject(c | s->exists(c' |
            c.name(self.SnapshotNames) =
            c'.name(self.SnapshotNames)or
            c'.parents ->includes(c)))))
```

[4] Returns all the derived parents of a object. A parent is derived if the container of the object (i.e. its snapshot) or any of the container's parents contain a object with the same name.

```
context Object_
 allDerivedParents() : Set(Object_)
   if self.Snapshot <> self then
     self.Snapshot.allParents() -> iterate(p s = Set{} |
       s->union(p.Object_s -> select(c |
         c.name(self.Object_Names) =
         self.name(self.Object_Names))))
     else Set{}
   endif
```

Identical queries are generated for Link and Snapshot.

[5] Returns all parents of an object, including its local and derived parents.

```
context Object_
  allParents():Set(Object_)
    self.allLocalParents()->union(self.allDerivedParents())
```

Identical queries are generated for Link and Snapshot.

[6] Returns all local parents of an object.

```
context Object
  allLocalParents() : Set(Object_)
    self.parents->iterate(parent S = self.parents |
      S->union(parent.allLocalParents()))
```

Identical queries are generated for Link and Snapshot.

5.3.5 Semantic Mapping

The semantic mapping for packages defines all possible legal configurations of snapshots for a package and its contents. In other words, given a candidate snapshot, the semantic mapping package tells us, for any given package whether or not the snapshot is a legal instance of the package.

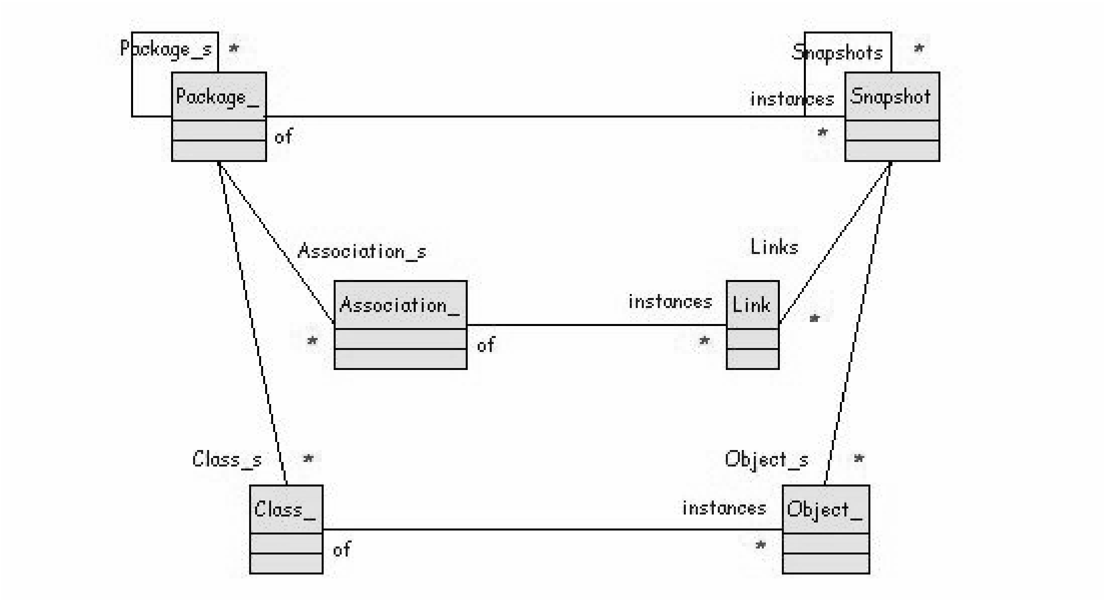


Figure 50. UML2.Static.Packages.Mapping

In order for an snapshot to be an instance of a package the snapshot must define objects, links and sub-snapshots for all the local classes, associations and sub-packages of the package, and each of their parents must be instances of appropriate superclasses, associations or packages.

Well-formedness Rules

[1] Every object belonging to an snapshot is an instance of the snapshot's package's classes.

```
context Snapshot_
  inv InstanceContentsCommute
    self.of_.Class_s =
      self.Object_s -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[2] Every link belonging to a snapshot is an instance of the snapshot's package's associations.

```
context Snapshot_
  inv InstanceContentsCommute
    self.of_.Association_s =
      self.Links -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[3] Every sub-snapshot belonging to an snapshot is an instance of the snapshot's package's sub-packages.

```
  context Snapshot_
    inv InstanceContentsCommute
      self.of_.Package_s =
        self.Snapshots -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[4] A snapshot's parents are instance's of its package's parents.

```
   context Snapshot_
    self.of_.parents =
    self.parents -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[5] An object's parents are instance's of its class's parents.

```
   context Object_
    self.of_.parents =
    self.parents -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[6] A link's parents are instance's of its association's parents.

```
   context Link
    self.of_.parents =
    self.parents -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

Example

Figure 51 shows a package and a legal instance. The package defines a sub-package

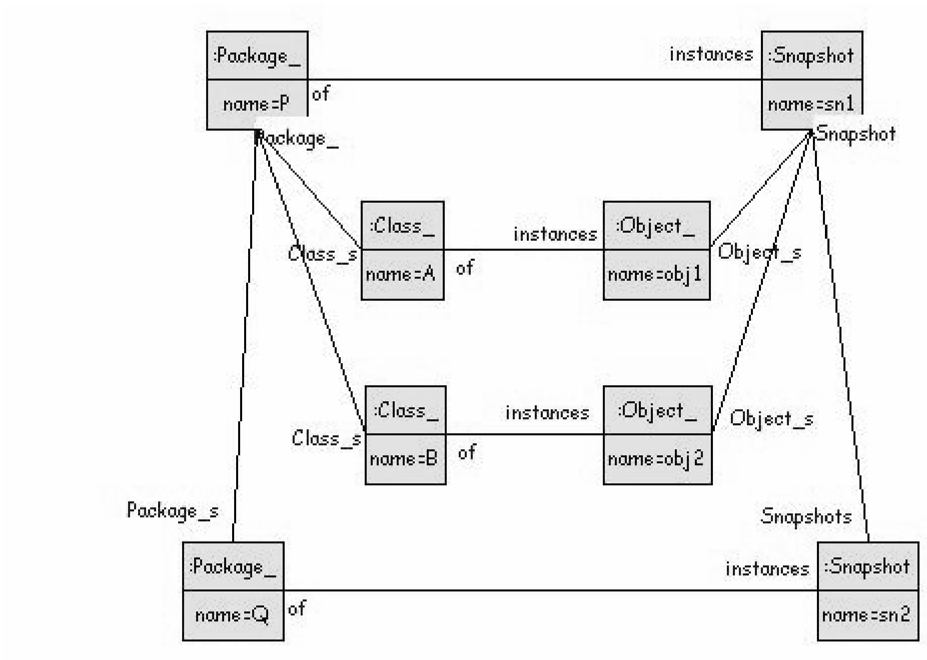


Figure 51. A package with instances.

and two classes named A and B. The snapshot is a valid instance of the package because its sub-snapshots and objects are valid instances of the contents of the package.

Figure 52 shows a package P with a single super-package Q. Package P defines one class named B, Q defines one class named A whose parent is B. P has a single instance with an object for A and a parent link to an instance of Q. The instance of B also has a parent link to the instance of A reflecting its ability to be thought of as an instance of Q.

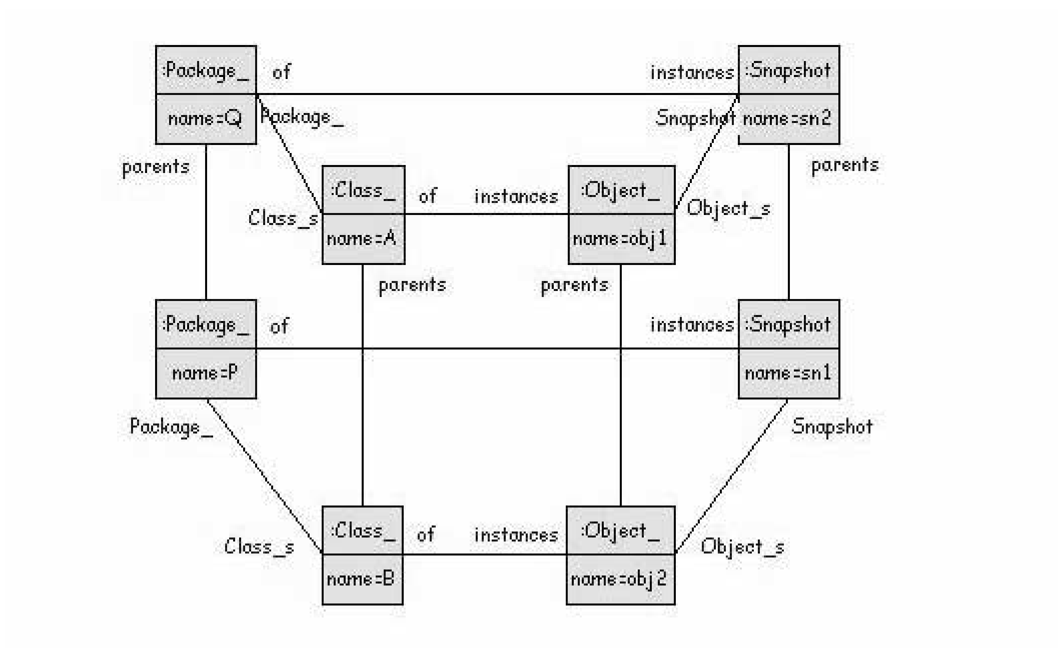


Figure 52. A package with parents and instances.

5.4. Associations

5.4.1 Overview

Classes are related by associations. Association have association ends that connect the association to two or more classes. Each association end defines the number of objects that can be related by instances of the association through its multiplicity.

5.4.2 Templates

Figure 53 shows the templates used to “stamp out” the Classes package (see chapter 4 for a full description).

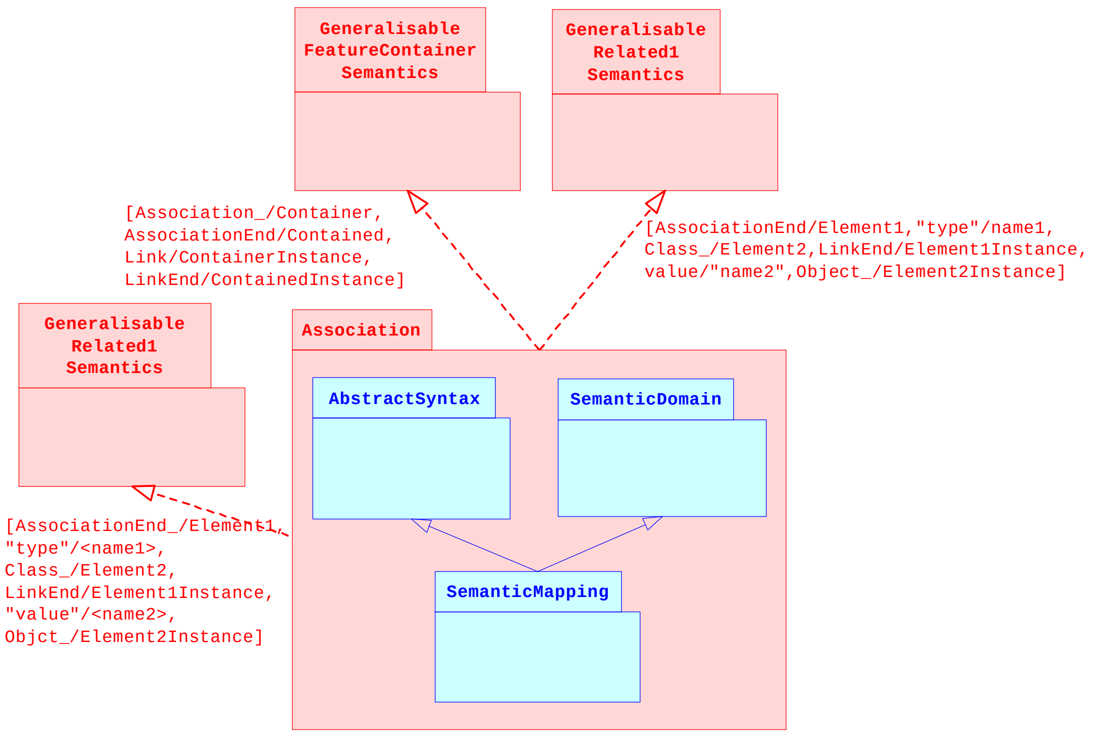

Generalisable FeatureContainer Semantics
Generalisable Related1 Semantics
[Association_/Container, AssociationEnd/Contained, Link/ContainerInstance, LinkEnd/ContainedInstance]
[AssociationEnd/Element1,"type"/name1, Class_/Element2,LinkEnd/Element1Instance, value/"name2",Object_/Element2Instance]
Generalisable Related1 Semantics
Association
AbstractSyntax
SemanticDomain
SemanticMapping
[AssociationEnd_/Element1, "type"/<name1>, Class_/Element2, LinkEnd/Element1Instance, "value"/<name2>, Objct_/Element2Instance]

Figure 53. Templates used in the Associations package

Associations are containers of association ends. Associations have a namespace for their association ends. Links are instances of associations. Linkends are instances of association ends. Each linkend relates instances of associated classes according to the multiplicity of their association ends. Links are containers of Linkends. Associations are generalisable.

5.4.3 Abstract Syntax

Figure 54 defines a model for associations. An association is generalisable and contains a number of association ends.

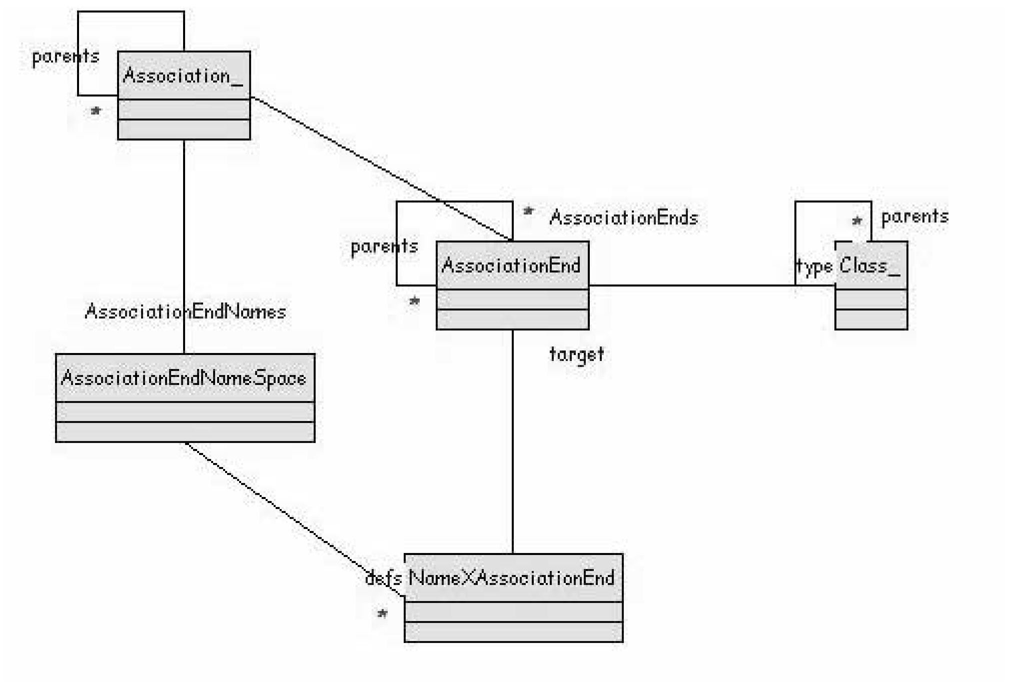


Figure 54. UML2.Static.Associations.AbstractSyntax

Each association contains a name space for its association ends. Each name space contains a set of definition pairs associating each association end with a name. An association end has a multiplicity, and is related to a class (a participant in the association).

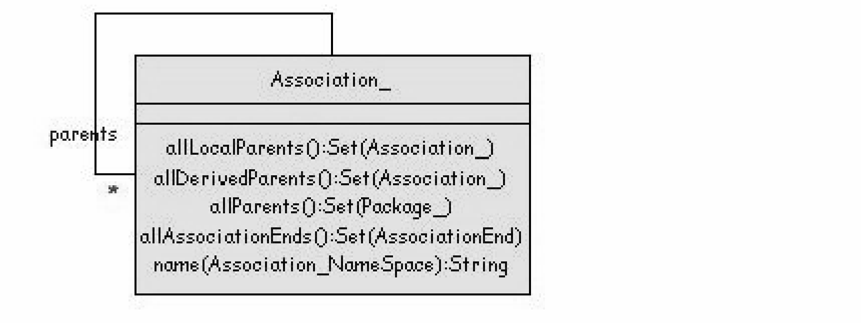

parents
*
Association_
allLocalParents():Set(Association_)
allDerivedParents():Set(Association_)
allParents():Set(Package_)
allAssociationEnds():Set(AssociationEnd)
name(Association_NameSpace):String

Figure 55. Association Queries

Figure 55 shows the definition of association queries. The name of the association can be requested with respect to a given name space. The association ends which are both defined and inherited by the association are given by Association_::allAssociationEnd_s(). The locally defined parents of an association are given by Association_::parents. The transitive closure of this relationship is Association_::allLocalParents().

Well-formedness Rules

[1] All association ends in a association's namespace have different names.

```
context Association_ inv
  AllAssociationEndsHaveDistinctNames
    self.AssociationEnds->forAll(element, element’ |
      element.name(self.AssociationEndNames) =
      element'.name(self.AssociationEndNames) implies
        element = element'))
```

[2] The parents of an associationend's type are related to the associationend's parents

```
context AssociationEnd inv:
  RelatedParentsCommute
    self.type.parents = self.parents ->
    iterate(p s = Set{} | s->including(p.type)
```

Queries

[1] Returns all association ends of an association including its local association ends and those derived from its parents (except those with the same name as the association's local association ends or those that are redefined).

```
context Association_
  allAssociationEnds() : Set(AssociationEnd)
    self.allParents() ->
      iterate(p s = self.AssociationEnd |
        s->union(p.allAssociationEnds()->
          reject(c | s->exists(c' |
            c.name(self.AssociationEndNames) =
            c'.name(self.AssociationEndNames)or
            c'.parents ->includes(c)))))
```

[2] Returns all the derived parents of an association end. A parent is derived if the container of the association end (i.e. its association) or any of the container's parents contain an association end with the same name.

```
 context AssociationEnd
  allDerivedParents() : Set(Association_)
    if self.Association_ <> self then
      self.Association_.allParents() -> iterate(p s = Set{} |
        s->union(p.AssociationEnds -> select(c |
          c.name(self.AssociationEndNames) =
          self.name(self.AssociationEndNames))))
      else Set{}
    endif
```

[3] Returns all parents of an association end, including its local and derived parents.

```
context AssociationEnd
  allParents():Set(AssociationEnd)
    self.allLocalParents()->union(self.allDerivedParents())
```

[4] Returns all local parents of an association end.

```
context AssociationEnd
  allLocalParents() : Set(AssociationEnd)
    self.parents->iterate(parent S = self.parents |
      S->union(parent.allLocalParents()))
```

[2] The parents of an linkend's value are related to the linkend's parents

```
context LinkEnd inv:
  RelatedParentsCommute
    self.value.parents = self.parents ->
    iterate(p s = Set{} | s->including(p.value)
```

Example

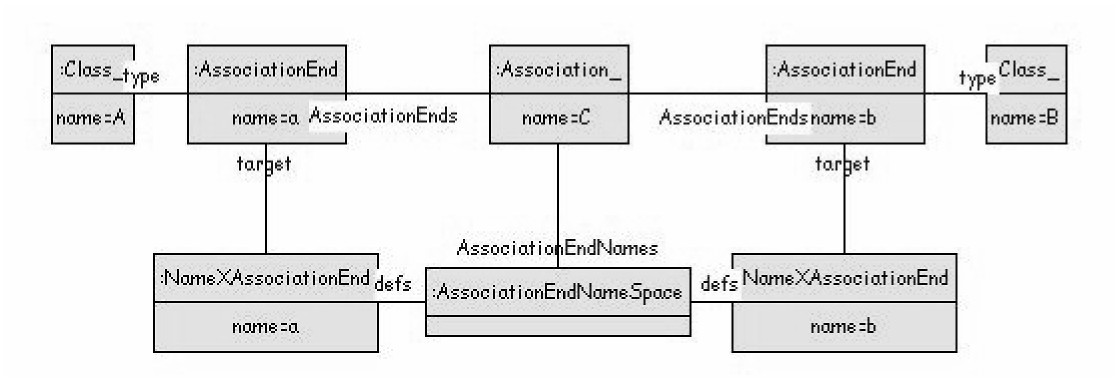


Figure 56. An association with two association ends

Figure 56 gives an example of a single association named C with two association ends named a and b respectively that relate two classes A and B.

5.4.4 Semantic Domain

Associations denote links, whilst association ends denote linkends. A link contains two or more linkends depending on the arity of the association. A linkend has a name and a value. The value of the linkend is an object. The semantic domain is shown in Figure 57.

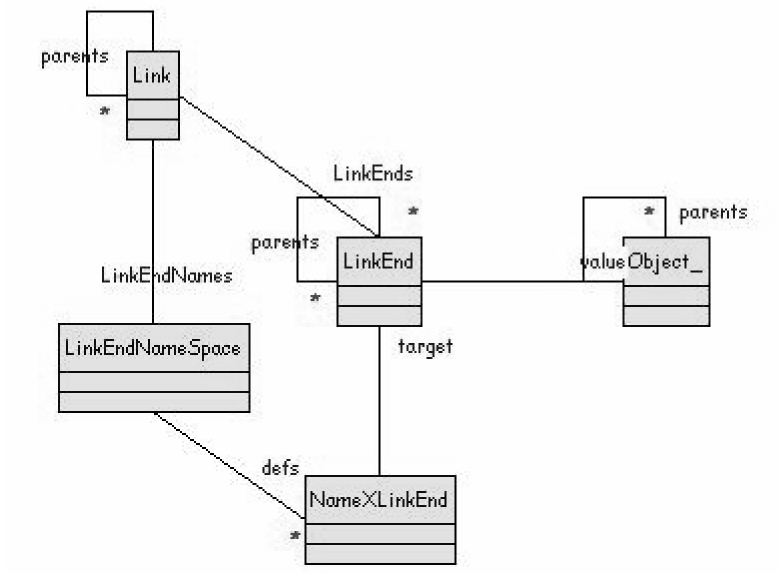


Figure 57. UML2.Static.Associations.SemanticDomain

Both links and linkends have parents. This structure reflects the parent relationships between associations and association ends respectively. A link has local linkends and inherited linkends which correspond exactly to the local and inherited association ends of its association. This permits a semantic domain element that is an instance of a association C to be easily be viewed as an instance of a super-class of C by selecting the appropriate super-links.

Well-formedness Rules

[1] All linkends in a link’s namespace have different names.

```
context Link inv
  AllLinkEndsHaveDistinctNames
    self.LinkEnds->forAll(element, element’ |
      element.name(self.LinkEndNames) =
      element'.name(self.LinkEndNames) implies
        element = element'))
```

Queries

[1] Returns all linkends of a link including its local linkends and those derived from its parents (except those with the same name as the link’s local linkends or those that are redefined).

```
context Link
  allLinkEnds() : Set(LinkEnd)
    self.allParents() ->
      iterate(p s = self.LinkEnd |
        s->union(p.allLinkEnds()->
          reject(c | s->exists(c' |
            c.name(self.LinkEndNames) =
            c'.name(self.LinkEndNames)or
            c'.parents ->includes(c)))))
```

[2] Returns all the derived parents of a linkend. A parent is derived if the container of the linkend (i.e. its link) or any of the container's parents contain a linkend with the same name.

```
context LinkEnd
  allDerivedParents() : Set(LinkEnd)
    if self.Link <> self then
      self.Link.allParents() -> iterate(p s = Set{} |
        s->union(p.LinkEnds -> select(c |
          c.name(self.LinkEndNames) =
          self.name(self.LinkEndNames))))
      else Set{}
    endif
```

[3] Returns all parents of a linkend, including its local and derived parents.

```
context Link
  allParents():Set(LinkEnd)
    self.allLocalParents()->union(self.allDerivedParents());
```

[4] Returns all local parents of a link end.

```
context Link
  allLocalParents() : Set(LinkEnd)
    self.parents->iterate(parent S = self.parents |
      S->union(parent.allLocalParents()))
```

5.4.5 Semantic Mapping

The semantic mapping for associations (see Figure 58) defines all possible legal configurations of links and linkends for every legal configuration of association and association ends. In other words, given a candidate link and linkends, the semantic mapping package tells us, for any given association whether or not the link is a legal instance of the association. In order for a link to be an instance of an association the link must define linkends for all the local association ends of the class, the link end values must be instances of association types (or a super-class thereof) and the super-links must be instances of appropriate super-associations

.

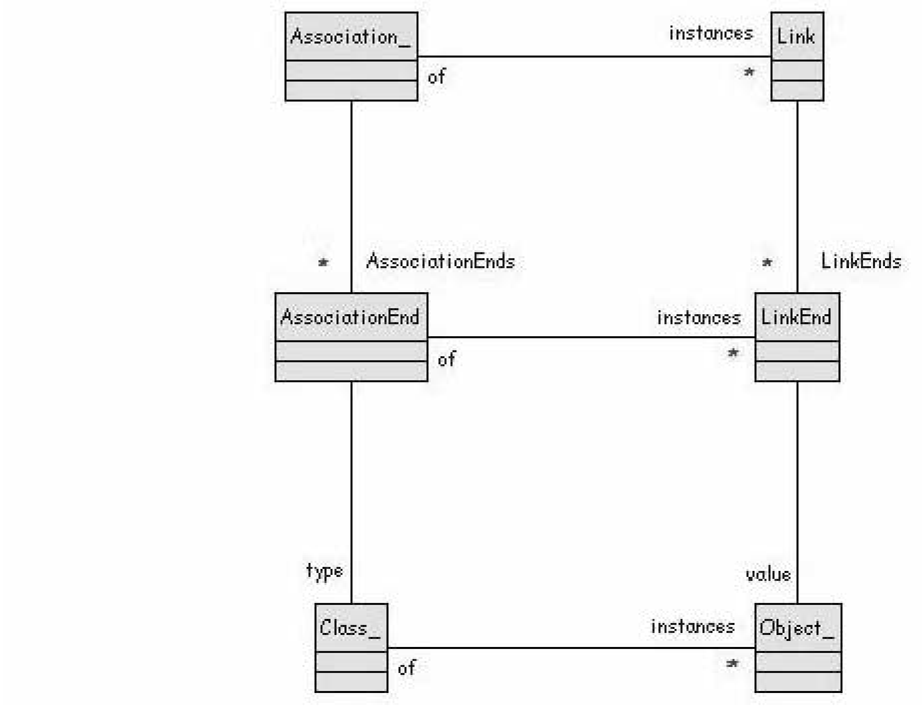


Figure 58. UML2.Static.Associations.SemanticMapping

Well-formedness Rules

[1] Every linkend belonging to an link is an instance of the link’s association’s association ends.

```
context Link
  inv InstanceContentsCommute
    self.of_.AssociationEnds =
      self.LinkEnds -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[2]A link contains the same number of linkends as the number of association ends contained by its association.

```
context Link
  inv SameNumberOfElementsAsContainer
    self.LinkEnds -> size = self.of_.AssociationEnds -> size
```

[3] A linkend’s parents are instance’s of its association end’s parents.

```
 context LinkEnd
  self.of_.parents =
  self.parents -> iterate(p s = Set{} | s->union(Set{p.of_}))
```

[4] The value of a linkend is an instance of it’s association end’s type.

```
context LinkEnd inv:
  inv InstanceElementsCommute
  self.of_.type =
  self.value.of_
```

[5] The number of links between two objects must conform to the multiplicity of their association ends.

```
To be defined in the final submission.
```

Figure 59 shows an association and a legal instance.

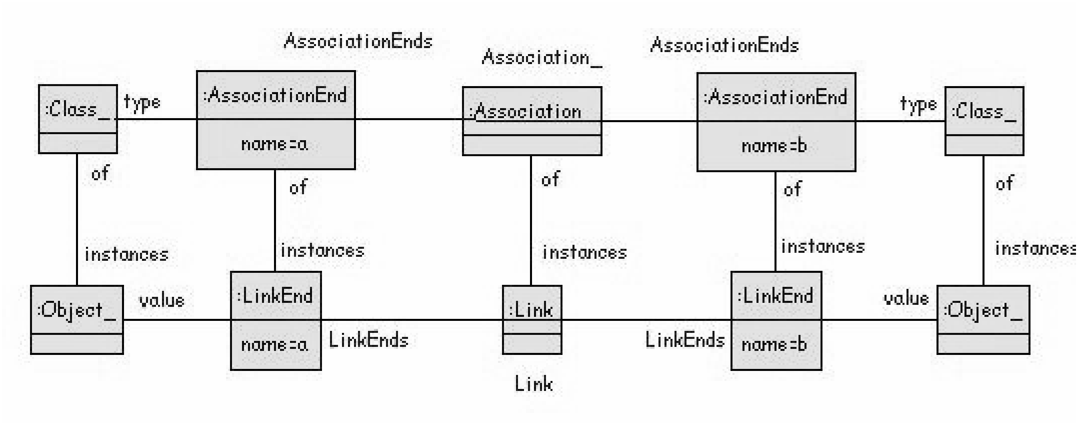


Figure 59. An association and an instance

The association defines two association ends named a and b. Therefore its instance must have two link ends with the same names. The values of the link ends must be instances of the types of the association end (for simplicity we reuse the type C and the value o)

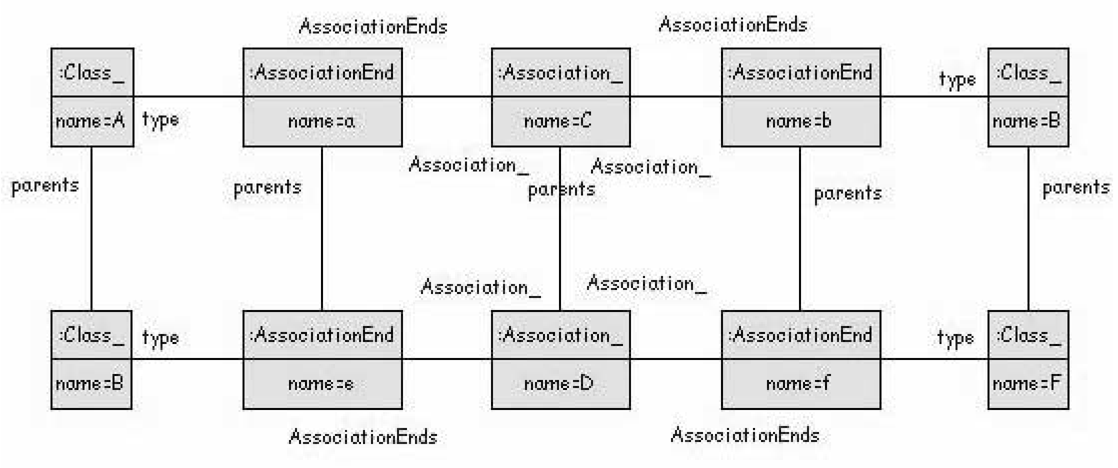


Figure 60. An association with parents

Figure 60 shows an association D with a single super-association C. Association C defines two association ends named a and b associated with classes A and B. D defines two association ends named e and f that specialise a and b and which are associated with classes E and F. Because the association ends e and f redefine the association ends a and b, then D::allAssociationEnds() will return e and f as its association ends. Furthermore, because e and f are subclasses, their types must also be specialised (see section 5.4.3, constraint [2]). Interestingly, if C and D were binary associations (i.e. C,D::allAssociationEnds() -> size must be equal to 2) then association ends *must* be redefined to avoid inheriting additional association ends. Because the association ends are specialised, their type's must be specialised too, leading to the traditional kind of type specialisation and roleend redefinition seen in the literature.

5.5. DataTypes

UML provides a number of built-in data types. There are two categories of data type: *ground* data types such as Integer and Char, and *parametric* data types such as Collection and its associated sub-classes (Set, Bag, Seq), Functions and Tuples. Ground data types are classes (named Integer, Boolean etc.) that define a standard interface (+, >, not, and etc.). Therefore instances of ground data types are objects that behave in the expected way.

Parametric data types are classes whose *instances* are ground data types. For example Set is a class whose instances are data types Set(Integer), Set(Class), Set(Set(Boolean)) etc. Each parametric data type defines a standard interface which applies uniformly to all the instance data types. For example Set defines an operation *size* whose definition applies to all ground set types: the size of an instance of Set(Integer) is calculated in the same way as the size of an instance of Set(Set(Class)).

5.5.1 Templates

To be defined in the final submission.

5.5.2 Abstract Syntax

The DataTypes abstract syntax package defines *ground* data types (Integer, etc.). Ground data types are sub-classes of the class Class_. Ground data types are meta-instances of Class_ (not shown here) thus ensuring that only a single instance of each ground data type can exist within a model. A tuple is defined over a sequence of types, for example (Integer x String x String). A function is defined over a binary Tuple (its domain and range types).

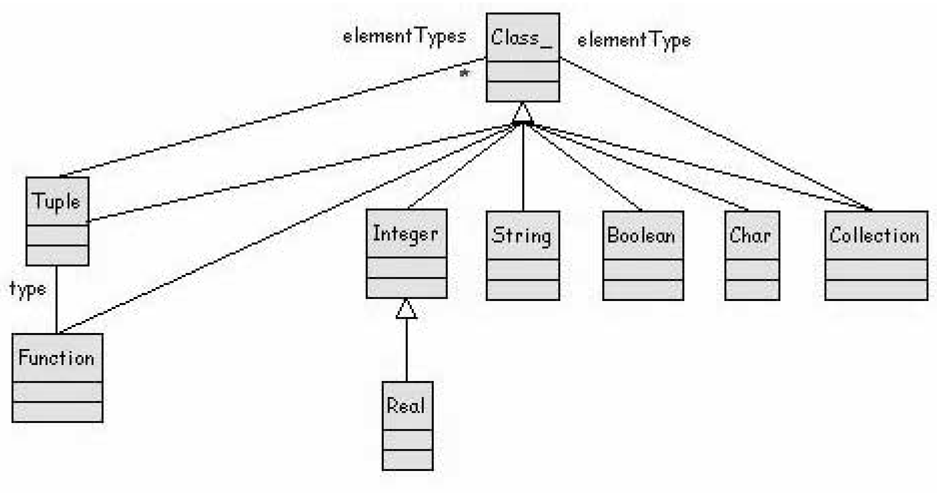


Figure 61. UML2.Static.DataTypes.AbstractSyntax

Well-formedness Rules

[1] A Function has a domain and range (a binary tuple type).

```
context Function
  self.type.elementTypes -> size = 2
```

5.5.3 Semantic Domain

The DataTypes semantic domain package introduces instances of collections, tuples and functions. A collection value is a set of objects. A tuple value is a sequence of objects. A function is a set of binary tuple values.

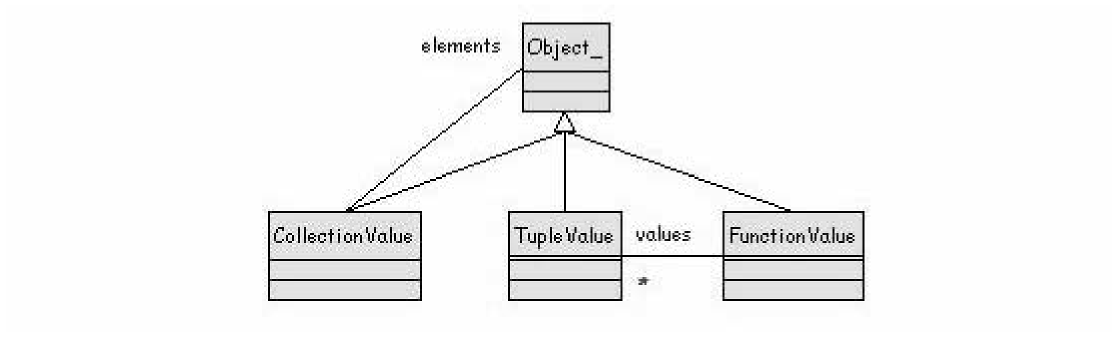


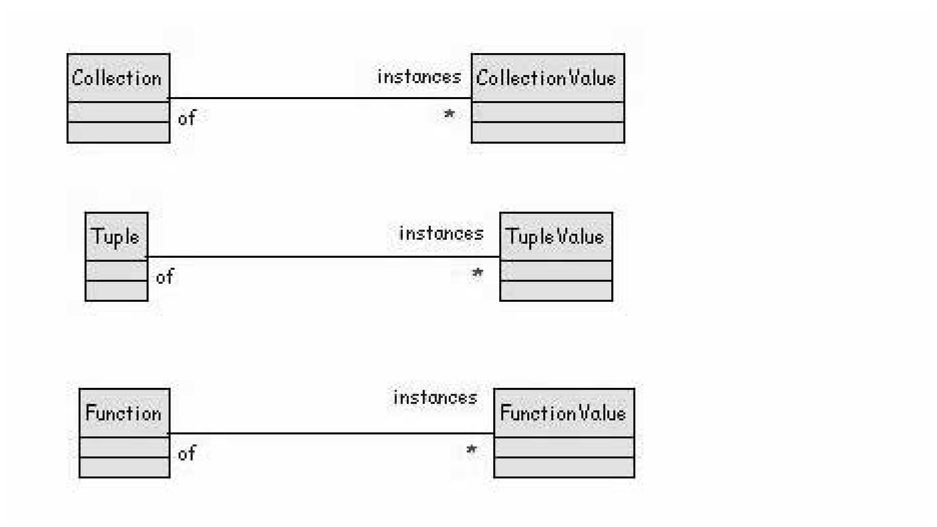
Figure 62. UML2.Static.DataTypes.SemanticDomain Package

Well-formedness Rules

None.

5.5.4 Semantic Mapping

A collection value is an instance of a collection. A tuple value is an instance of a tuple. A function value is and instance of a function.



Well-formedness Rules

[1] A tuple value’s object’s are instances of its tuple’s types.

[2] A collection value’s objects are instances its collection’s type.

[3] A function’s tuple value’s are instances of the function’s tuple type.

To be defined.

5.6. Expressions

Languages which navigate static models and which express properties of semantic models will rely on a basic collection of expression primitives. Examples of languages which require these facilities are OCL and action languages.

UML 2.0 must support families of expression languages and therefore must provide a suitably expressive collection of expression primitives. In addition to providing support for basic expression types such as constants and variables, a package of expression primitives must address more complex features such as method invocation, local variable binding and recursion. This is achieved using the standard technique which provides first class functions as basic semantic elements, thereby addressing all *binding* and *invocation* issues in a single language feature.

5.6.1 Templates

To be defined in the final submission.

5.6.2 Abstract Syntax

The key features of all expression languages are as follows:

- Constants, for example integers, booleans and the empty sequence. We view everything as an object. Therefore, constants are a collection of predetermined objects. For example the integer constant 1 is a predefined object which behaves in the usual way when asked for its successor, predecessor etc.
- Variables which are bound to values in expressions and may be introduced by binding constructs such as method parameters, *let* expressions and *iteration* expressions.
- Relational expressions. Relations include boolean relations (e.g. >), integer functions (e.g. +) and object field references (e.g. name).We do not prescribe a fixed collection of relations beyond the relations which are necessary to support the definition of the kernel language. This allows the expression language to be extended with new relations to support language families.
- Collection expressions. The most important forms of collection are *sets* and *sequences*. Other forms of collection can be constructed as objects implemented in terms of basic collection types. For example OCL bags can be implemented as objects that use sequences for data storage.
- Conditional expressions.
- Query invocation. A class may define queries. A method is a parameterised expression. This is to be contrasted with general method invocation which may cause object side effects. Side effects are not supported by the expression primitives.
- Functions. A function is a parametric expression. Queries are functions, invariants are functions, and functions can be used to represent all possible expression constructs that *compute* values over a model (for example *iterate* expressions in OCL). Functions are applied to argument values.
- Recursion. Recursion is wide-spread in expression languages. For example OCL allows query operations to refer to themselves. In order to support families of

expression languages we must provide a general purpose recursion mechanism that is not tied to any particular language features.

- Tuples. A tuple is an ordered collection of values. A tuple is different from a collection in that the elements of the tuple may have different types (all elements of a sequence must be instances of a given type). Elements of a tuple are referenced by position in the tuple. Tuples are useful when operations wish to produce more than one value with unrelated types.

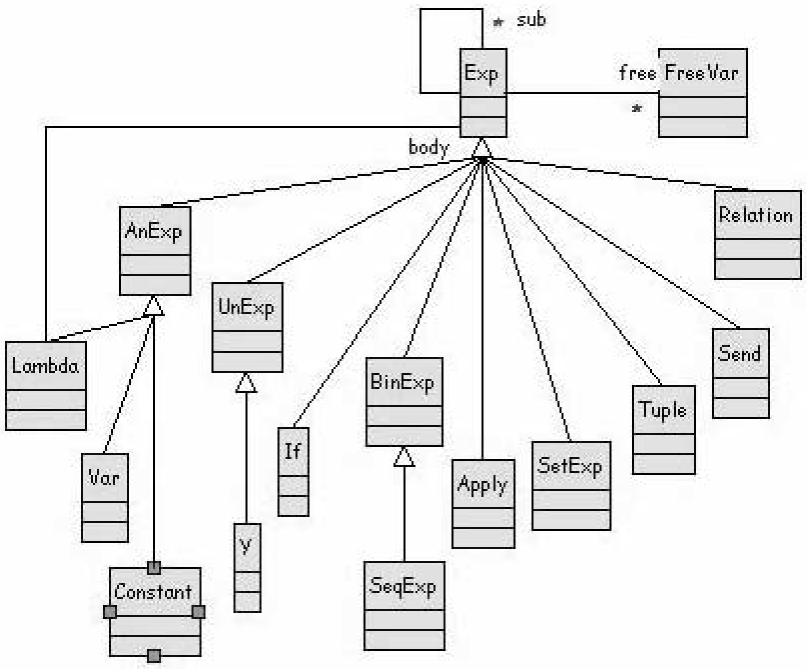


Figure 63. UML2.Static.Expressions.AbstractSyntax

Figure 63 on page124 shows a model of the abstract syntax for expression primitives. An expression contains an ordered collection of sub-expressions and has a set of free variables and a type. Each free variable has a name and a type:

context Env **inv**: free = sub->iterate(e F = Set{} | F->union(e.free))

The class Exp has three abstract sub-classes. An AnExp has no sub-expressions. An UnExp has exactly one sub-expression. A BinExp has exactly two sub-expressions. The rest of this section describes the concrete expression classes.

Figure 64 shows the definition of the primitive expression classes. In order to provide motivation for these classes we will incrementally construct an example expression. Suppose that we have a set of people and wish to add up their ages. If each person is called ‘person’ and the current running total of ages is ‘ages’ then the following expression:

person.age + ages

adds the current person to the current total. Therefore we have a relational expression (+) including a relational expression (.age) and a variable expression (ages). The relational expression contains a single variable expression (person).

The particular person will be varied, as will the current running total so we abstract over these names:

```
\(person,ages) person.age + ages
```

which is a function expression. Call this function ‘body’ we will use it later. Suppose that the collection of people is called ‘people’. We can check whether there any further people using a relation expression:

```
people.isEmpty
```

NB this is not OCL, but we support a builtin relation which OCL could use. Another relation is selectElement:

```
people.selectElement
```

If the set of people is empty then we produce the current running total otherwise we select a person and add it to the current running total using a conditional expression:

```
if people.isEmpty
then ages
else people.selectElement.age + ages
```

We already have a function that performs the increment so:

```
if people.isEmpty
then ages
else body(people.selectElement,ages)
```

The expression above adds one person’s age to the running total. We must do this for all elements of ‘people’. We can reduce the current set of people by the selected element if using the send expression ‘excluding’:

```
people.excluding(people.selectElement)
```

We must do this for the selected element supplied to body. Suppose that ‘iterate’ (to be defined below) is applied to a set of people, a body and a running total. The operator ‘iterate’ applies the body to all the elements and returns the running total:

```
iterate(people,body,0)
```

applies ‘body’ to all people, adds up their ages and adds it to 0. Then we can use ‘iterate’ to add up the rest of the people:

```
if people.isEmpty
then ages
else let person = people.selectElement
     in iterate(people.excluding(person),body,body(person,ages))
```

The above is the definition of ‘iterate’. Therefore, ‘iterate’ is a recursive operator. We can construct such operators using Y:

```
let iterate =
   Y \(iterate) \(people,body,ages)
        if people.isEmpty
        then ages
        else let person = people.selectElement
               in iterate(people.excluding(person),body,body(person,ages))
in iterate(people,\(person,ages) person.age + ages,0)
```

which is equivalent to the following OCL expression:

```
people->iterate(person ages = 0 | person.age + ages)
```

Of course the OCL expression above is much more succinct. But the expression primitive language is much smaller is more expressive and can therefore by used as the basis of a family of expression languages including OCL.

The rest of this section gives the OCL well-formedness constraints for the abstract syntax of the expression primitives.

A constant has a value:

context Constant **inv**: value.of = type and free->size = 0

A variable has a name and a type. The free variables in this expression is itself:

context Var **inv**: free->forAll(f | f.name = name and f.type = type)

A relation is implemented as a function that, when applied to elements from its domain produces an element in the range.

The free variables of a function are those of its body with the arguments removed[1]:

context Lambda **inv**: free = body.free->SetDifference(args->asSet)

1. Note that a function with 0 arguments is well-formed. It is equivalent to its body expression; although the value of the expression must be 'released' by function application. This provides a systematic way of modelling queries in Section 5.7 on pa ge135.

NB This section requires type constraints.

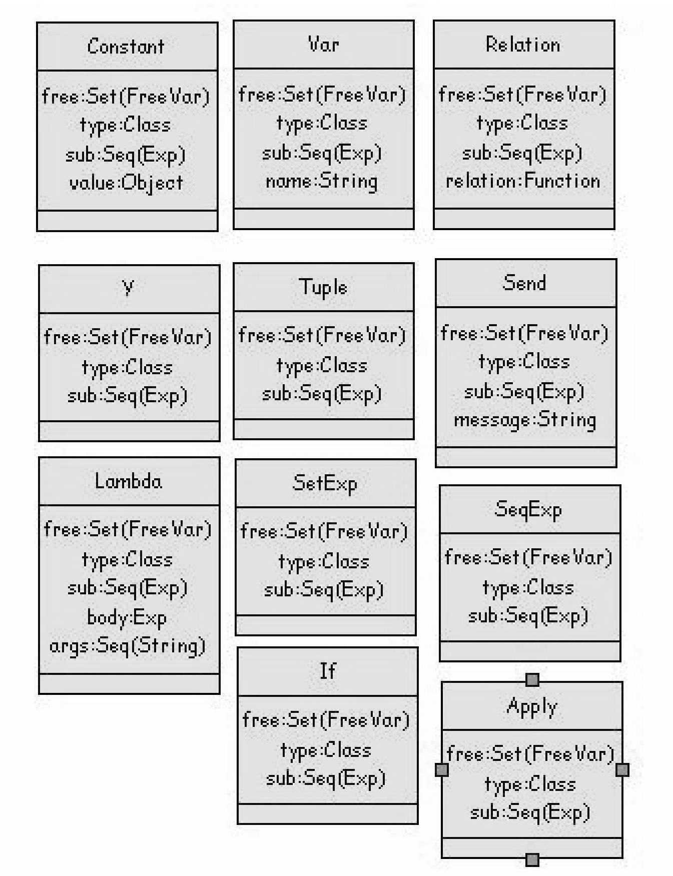


Figure 64. Definition of Expression Classes

5.6.3 Semantic Domain

Expressions *calculate* in a given context to produce values. Therefore, we will talk of expressions having instances called *calculations*. An expression (cf. Class) may contain variables (cf. attributes). A calculation (cf. Object) defines particular values for all of the variables (cf. slots).

A calculation is an historical record of one particular execution of an expression. It records the values of the variables that were required by the expression and also records the result. Since expressions can be nested, calculations are also nested. Nested calculations record data dependencies; typically, the value produced by a calculation will depend on the values of the calculations it contains.

In order to motivate the use of calculations we present a simple example. Consider a calculation that involves adding two constant numbers together. Three individual calculations are involved: two independent calculations produce the numbers and a containing calculation adds them together. If the number are 3 and 4 then the structure of the calculation is shown in This can be represented on a simple diagram as shown in Figure 65.

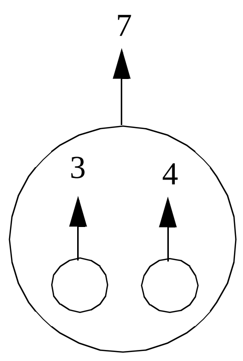


Figure 65. A simple calculation adding two constants.

Now suppose that the calculations producing numbers 3 and 4 were performed in a context involving variables. The sub-calculation producing 3 could have arisen from the evaluation of a variable whose value was 3. In this case the context for the sub-calculation would associate the variable with the value 3. The sub-calculation producing 4 could have arisen from the doubling of a variable whose value was 2. In this case the context for the sub-calculation would associate the variable with the value 2.

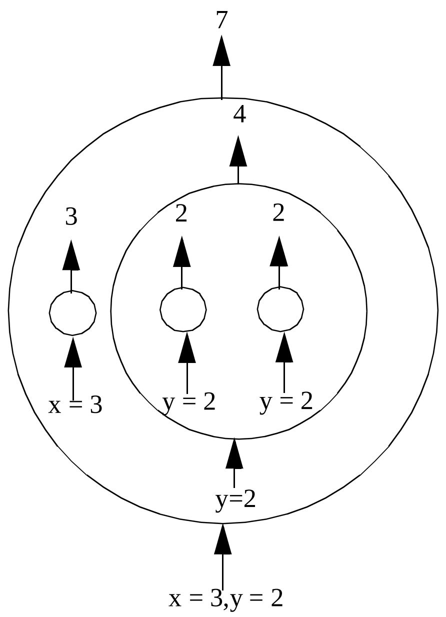


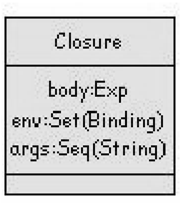
Figure 66. Calculations involving context.

Figure 66 shows a calculation involving contexts. The diagram shows contexts being *fed into* a calculation and values being *produced* by the a calculation. Notice that sub-calculations must have sub-contexts with respect to their containers.

NB calculations don't record what they did with the values. There is no labelling of the calculations, for example to show that it arose from an addition expression. This seems unnecessary but could be added in the form of specializations of an abstract calculation class or in terms of tags.

Figure 67 shows a model of calculations. A calculation is a container of sub-calculations. Each calculation has a context, referred to as an *environment*, containing variable-value pairs. The class Binding has a name of type string and a value of type Object. Each calculation produces a single value.

A closure is the result produced by evaluating a function expression. A closure has the following structure:



where args is a sequence of argument names, body is an expression that will be evaluated when the function is invoked and env is a collection of bindings for all of the free variables in the body of env.

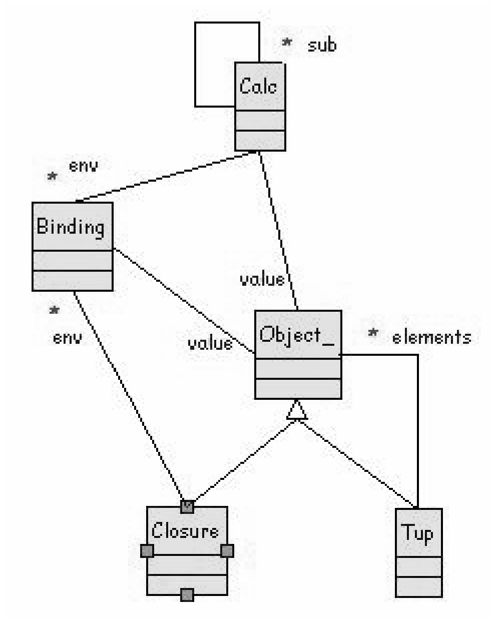


Figure 67. UML2.Static.Expressions.SemanticDomain

5.6.4 Semantic Mapping

Expressions have calculations as instances as shown in Figure 68. Each expression

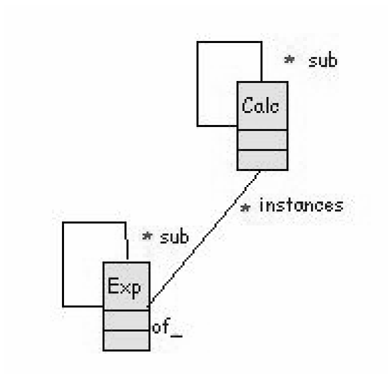


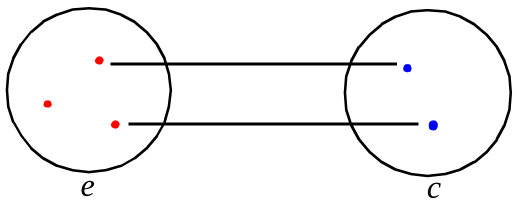
Figure 68. UML2.Static.Expressions.SemanticMapping

contains variables whose values are defined in the corresponding environment of the calculation. The types of the variables in the expression must be matched by the types of the values in the environment. Each expression contains sub-expressions; this containment structure is reflected in the calculations although it is not an exact match (see below). Expressions have types and calculations produce values; the type of the value produced by a calculation must conform to the type of the classifying expression.

There are three types of instantiating containment pattern (referred to as *Instantiable Container1*, *Instantiable Container2* and *Instantiable Container3* respectively[1]) involved in defining the expressions semantic mapping. In each case we have an expression *e* and an instance *c*. We are interested in the relationship between the sub-expressions of *e* and the sub-calculations of *c*:

1. the sub-calculations of *c* is a sub-set of the instances of the sub-expressions of *e*

   For example, this situation occurs when *e* is a conditional expression and when *c* contains calculations describing either the test and consequent or the test and alternative.

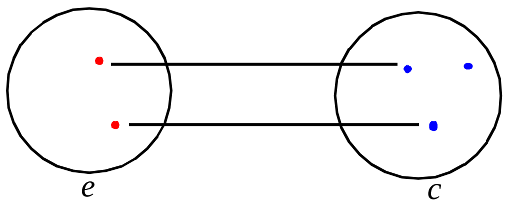


2. the sub-expressions of *e* is a sub-set of the classifiers of the sub-calculations of *c*:

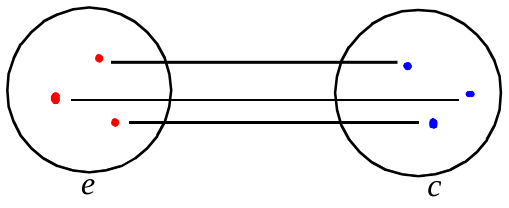
   For example this situation occurs when *e* is a method call (of one argument in the case shown above) and when *c* contains the calculations for the target expression, the single argument and also the calculation arising from the evaluation of the method body. The calculation *c* therefore contains sub-calculations arising form the

1. Note, InstantiableContainer1 is equivalent to ContainerSemantics in section 4.13. InstantiableContainer2 and InstantiableContainer3 are variations on the same template and will be defined later.

sub-expressions of *e* but also contains an extra calculation which arises from the method body.



3. the classifiers of the sub-calculations of *c* are exactly the sub-expressions of *e*. This situation occurs when *e* is a relational expression: every sub-calculation must correspond to exactly one sub-expression and vice versa.



Expressions have free variables and calculations have environments. Free variables describe the names and types of the variables which are used in the expression. Environments describe the names and the values of the variables which are used in the calculation. A calculation is an instance of an expression only if the environment corresponds to the free variables in terms of names and types:

context Exp **inv**: instances->forAll(i | i.env.name = free.name)

context Exp **inv**: i.env->forAll(b |
free->exists(f | f.name = b.name and f.type.instances->includes(b.value)))

Expressions have types and calculations produce values. A calculation is an instance of an expression providing that the value produced by the calculation is of the corresponding type:

context Exp **inv**: instances->forAll(i | type.instances->includes(i.value))

The rest of this section describes the meaning of each type of expression in turn.

Constants are *Instantiable Container3*. The value produced by an instance of a constant expression must be the value of the constant:

context Constant **inv**: instances->forAll(i | i.value = value)

Variables are *InstantiableContainer3*. The free variable constraint on expressions and calculations requires the calculation to produce a value which is defined in the environment and has the correct type.

Relations are *Instantiable Container3*. The value produced by an instance of a relation expression must be the value in the range of the relation corresponding to the values of the sub-calculations in the domain:

context Relation **inv**: instances->forAll(i |
    i.value = relation(i.sub->collect(c | c.value)))

A Y expression is used to create recursive values. It is an *Instantiable Container2*. The calculation arising from a Y expression is shown in Figure 69.

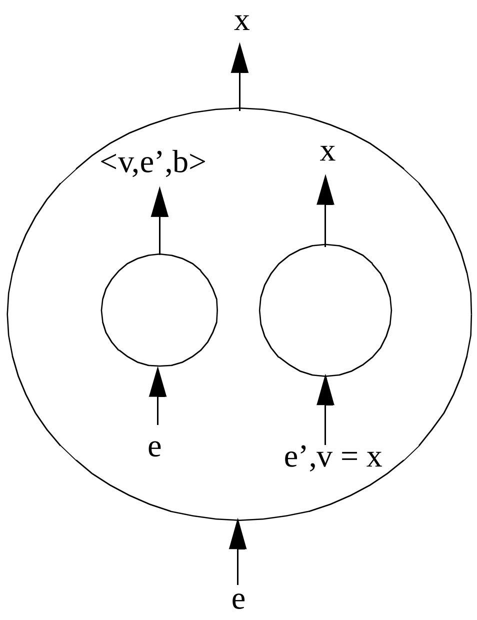


Figure 69. Calculation creating a recursive value.

The calculation contains two sub-calculations. The first (shown on the left) is an instance of the sub-expression of Y. It describes an evaluation which produces a *closure* <v,e',b> containing a single argument v, an environment e' and an expression (the body of a function) b. The second sub-calculation (shown on the right) is an instance of the function body b. The context is e',v=x which is the environment e' contained in the closure extended with a value for the argument v. The particular value x must be the same as that produced by the calculation (creating a recursive value).

context Y **inv**: instances->forAll(i | i.sub->size = 2)

context Y **inv**: instances->forAll(i | i.sub->at(0).env = env)

context Y **inv**: instances->forAll(i | tie(i.value,i.sub->at(0).value,i.sub->at(1).env))

context Y **inv**: instances->forAll(i |
    i.sub->at(1).value.body.instances->includes(i.sub->at(1))

Y::tie(v:Object,c:Closure,s:Set(Binding)):Boolean
    c.env->subSet(s) and
    let b = s - c.env
    in b.name = c.args->at(0) and b.value = v
    end)

Tuple is an *Instantiable Container 3.* The elements of the tuple produced by a tuple calculation must be the values of the sub-calculations:

context Tuple **inv**: instances->forAll(i | i.value.elements = i.sub.value)

Send is an *Instantiable Container 2.* An extra calculation arises from the evaluation of the query body. Given an object o and a message name m, o.getQuery(m) produces the query named m defined by the classifier of o. The query is a closure:

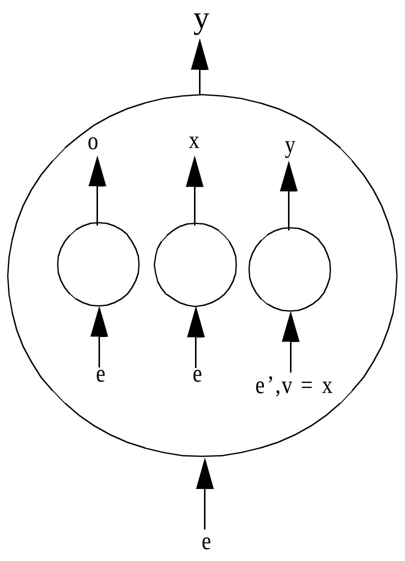


Figure 70. Calculation describing message passing

Figure 70shows a send calculation involving a single message argument. It consists of three sub-calculations. The first calculation (on the left) describes the evaluation of the target of the message. The target of the message is an object o. The second calculation (in the middle) describes the evaluation of the single message argument. In generate a message may have any number of arguments. Assuming that:

o.getQuery(m) = <v,e’,b>

Where v is the query formal argument, e’ is the method context and b is the query body then the third calculation (on the right) arises from the evaluation of the query body in the appropriate context.

context Send **inv**: instances->forAll(i | i.sub->size = 3)

```
context Send inv: instance->forAll(i |
        let c = i.sub->at(2); o = i.sub->at(2).value; a = i.sub->at(1).value
        in isSend(c,o.getQuery(name),a,value)
        end
```

```
Send::isSend(c:Calc,m:Closure,arg:Object,result:Object):Boolean
        m.env->subSet(c.env) and
        let b = c.env - m.env
        in b.name = m.args->at(0) and b.value = arg and c.value = result
        end
```

Lambda is an *Instantiable Container 3*. The result produced by the calculation is a closure that captures the current context which must contain values for all the free variables of the function:

context Lambda **inv**: instances->forAll(i |
    i.value.args = args and i.value.body = body and i.value.env = i.env and i.env.name = body.free.name)

SetExp is an *Instantiable Container 3*. The result produced by the calculation is a set containing the elements of the sub-calculations:

context SetExp **inv**: instances->forAll(i | i.value = i.sub.value)

SeqExp is an *Instantiable Container 3*. The result is a pair:

context SeqExp **inv**: instances->forAll(i |
    i.value->first = i.sub->at(0).value and i.value->rest = i.sub->at(1).value)

If is an *Instantiable Container 1*. A calculation is an instance of an if expression when it contains exactly 2 sub-calculations. The first sub-calculation must produce a boolean value. If the outcome is true then the second sub-calculation is an instance of the consequent of the if. If the outcome is false then the second sub-calculation is an instance of the alternative of the if.

context If **inv**: instances->forAll(i |
    i.sub->size = 2 and sub.instances->includes(i.sub->at(0))

context If **inv**: instances->forAll(i |
    i.sub->at(0).value implies sub->at(1).instances->includes(i.sub->at(1)))

context If **inv**: instances->forAll(i |
    not i.sub->at(0).value implies sub->at(2).instances->includes(i.sub->at(1)))

Apply is an *Instantiable Container 2*. The semantics of applications is exactly the same as that for send expressions except that instead of the extracting a closure from an object given a message name, the first sub-expression produces the closure directly.

5.7. Queries

A model snapshot is a configuration of objects. Given a snapshot we may wish to apply a predicate to test whether the objects satisfy a particular property (an example predicate tests whether all the people in the model are over 18 years old). We may also wish to apply an operation to the objects in the snapshot that computes some value (an example operation finds the oldest person in a collection of people). Collectively these predicates and operations are referred to as *queries*.

A class is a container of queries. Each query has a name and 0 or more parameters. The body of the query is an expression. Queries are inherited by sub-classes and all queries in a given class must have different names[1].

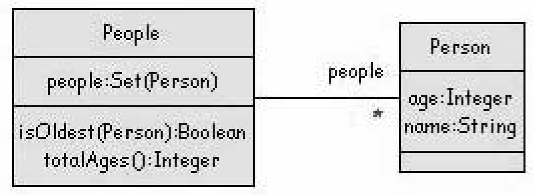


Figure 71. Example class with queries

Figure 71 shows a simple model involving a collection of people. The class People defines a query operation:

People::totalAges()
    people->iterate(person ages = 0 | person.age + ages)

The expression representation of the body of this query given in section 5.6.2. The second query involves an argument:

People::isOldest(p:Person):Boolean
    not people->exists(person | person.age > p.age)

5.7.1 Templates

To be defined in the final submission.

1. This requirement could be relaxed to allow interesting variations. For example, a class may define multiple queries with the same name providing the argument signature of each query is different in some way for example having different arities or different argument types. An interesting variation is to order the queries of a class and to allow overlapping definitions via a Smalltalk-like *run-super* mechanism.

5.7.2 Abstract Syntax.

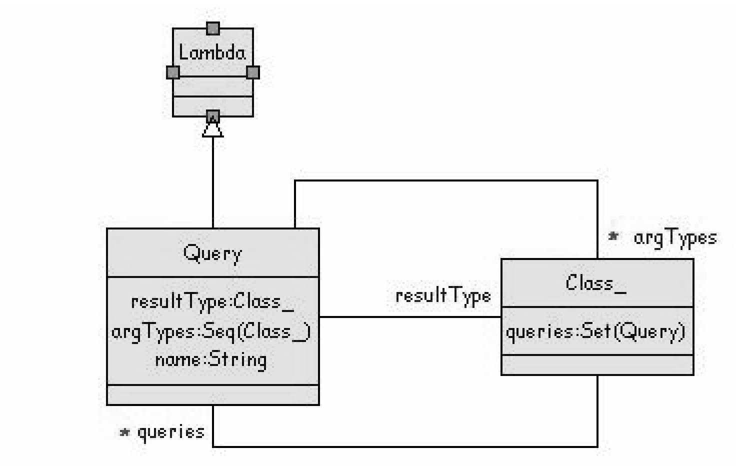


Figure 72. UML2.Static.Queries.AbstractSyntax

The abstract syntax for queries is given in Figure 72. A query is a specialisation of a function. The query is named, has argument types and a result type. A class is a container of queries. A class is also an inheritor of queries. A class defines a query called 'allQueries' which returns all the queries defined locally by the class and all the queries inherited by the class.

Every query has a special argument named 'self' which is used to refer to the target of the message (in a send expression) that caused the query to be performed. By convention we supply the value of this argument first:

context Query **inv**: args->at(0) = "self" and argTypes->at(0) = self.Class

5.7.3 Semantic Domain

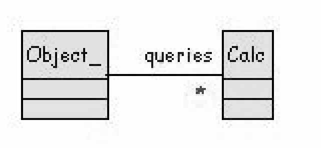


Figure 73. UML2.Static.Queries.SemanticDomain

Figure 73 shows the semantic domain for queries. An object is associated with a collection of query calculations. Each query calculation describes a particular query evaluation with respect to the current state of the object. The query calculation must have an environment that includes all the slot values of the object and associates the free variable "self" with the object:

context Object **inv**: queries->forAll(q |
    allSlots()->forAll(a | q.env->exists(b | b.name = s.name and b.value = s.value)) and
    q.env->exists(b | b.name = "self" and b.value = self)

5.7.4 Semantic Mapping

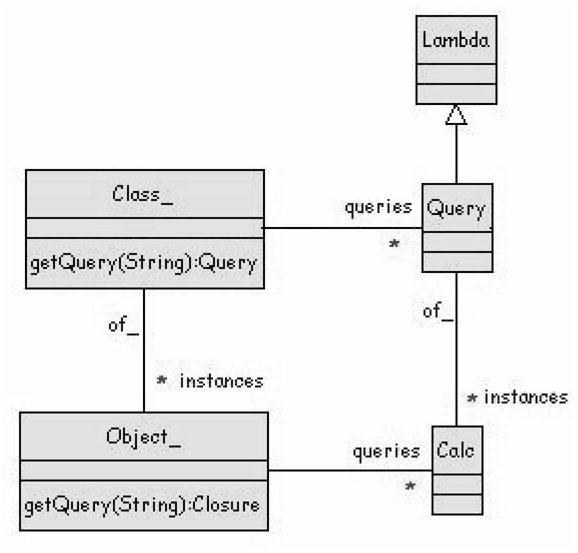


Figure 74. UML2.Static.Queries.SemanticMapping

Figure 74 shows the semantic mapping for queries. Objects are instances of classes and calculations are instances of queries. In order to be well-formed, the query calculations of an object must be instances of the corresponding queries of its class:

context Class **inv**: instances->forAll(o |
    o.allQueries()->forAll(c | allQueries()->exists(q | q.instances->includes(c)))

An object defines a query operation getQuery that is used to find a query closure:

context Object **inv**:
    getQuery(m).args = of.getQuery(m).args->tail and
    getQuery(m).env->exists(b | b.name = “self” and b.value = self) and
    get(Query(m).body = of.getQuery(m).body

5.8. Constraints

Given a snapshot consisting of a configuration of instances there are a number of properties which must hold true. These properties are referred to as *static constraints*. Static constraints may refer to individual objects, in which case they are defined for the classifier of the objects. Static constraints may also refer to collections of objects in a snapshot in which case they are defined for the classifier of the snapshot (i.e. a package).

Each static constraint is a function of one argument which returns a boolean result. Constraints are contained by classifiers. Every constraint contained by a classifier must produce the value *true* when supplied with an instance of the classifier. In the case of class constraints, every constraint must return true for all objects that are instances of the class. In the case of packages, every constraint must return true for all the snapshots that are instances of the package.

Constraint differ from queries since they cannot be arbitrarily parameterised. A constraint is a function of a single argument named ‘self’.

5.8.1 Templates

To be defined in the final submission.

5.8.2 Abstract Syntax

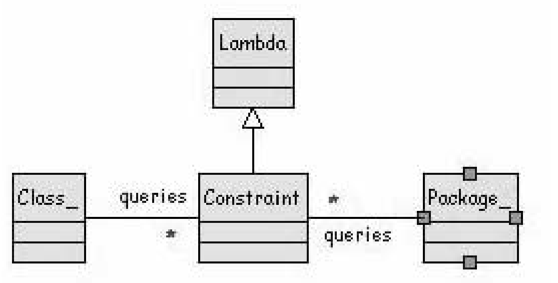


Figure 75. UML2.Static.Constraints.AbstractSyntax

Figure 75 shows the abstract syntax of constraints. Both classes and packages are containers of constraints. Classes and packages are also inheritors of constraints. They define queries ‘Class::allConstraints’ and ‘Package::allConstraints’ which return the union of the locally defined and inherited queries in both cases.

5.8.3 Semantic Domain

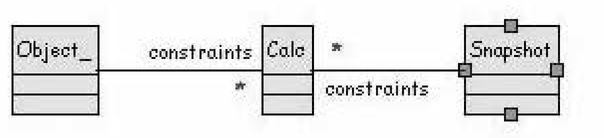


Figure 76. UML2.Static.Constraints.SemanticDomain

Figure 76 defines the semantic domain for constraints. Objects have calculations that arise from evaluating constraints in the context of the object. Snapshots have calculations that arise from evaluating constraints in the context of the snapshot. In each case the calculations must produce a boolean value. Objects and snapshots are containers and inheritors of constraint calculations; they both define queries ‘allConstraints’.

```
context Object inv: constraints->forAll(c | c.value.of = Boolean)

context Object inv: constraints->forAll(c |
  allSlots()->forAll(s | c.env->exists(b | b.name = s.name and
  b.value = s.value))

context Object inv: constraints->forAll(c |
  c.env->exists(b | b.name = “self” and b.value = self)

context Snapshot inv: constraints->forAll(c | c.value.of = Boolean)

context Snapshot inv: constraints->forAll(c |
  c.env->exists(b | b.name = “self” and b.value = self)
```

5.8.4 Semantic Mapping

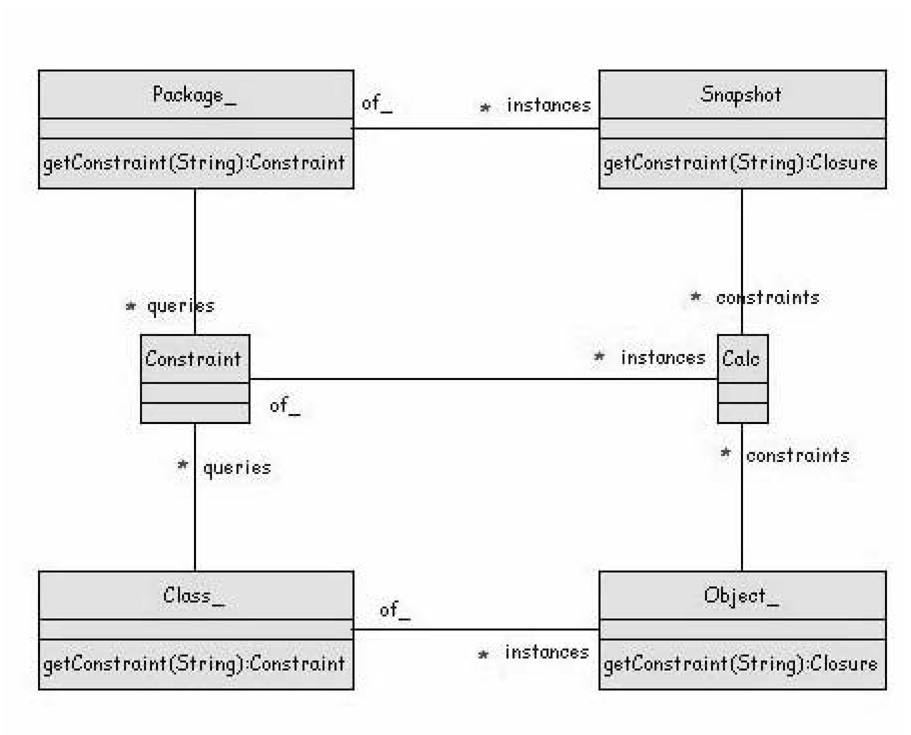


Figure 77. UML2.Static.Constraints.SemanticMapping

Figure 77 shows the semantics of constraints. For both classes and packages, all the constraints must hold for their instances. Every constraint calculation must be an instance of the associated constraint:

```
context Object inv: allConstraints()->forAll(calc |
  of.allConstraints()->exists(constraint |
  constraint.body.instances->includes(calc)))
```

```
context Snapshot inv: allConstraints()->forAll(calc |
  of.allConstraints()->exists(constraint |
  constraint.body.instances->includes(calc)))
```

Every constraint must be performed so for each classifier the set of constraint calculations for each instance must match the set of all constraints.

```
context Class inv: instances->forAll(instance |
  allConstraints()->forAll(constraint |
           instance.allConstraints()->exists(calc |
                    constraint.instances->includes(calc)
                        and calc.value)))
```

```
context Package inv: instances->forAll(instance |
  allConstraints()->forAll(constraint |
           instance.allConstraints()->exists(calc |
                    constraint.instances->includes(calc)
                        and calc.value)))
```

5.9. Templates

A template is a model element that contains replaceable names. Model elements that do not contain replaceable names are referred to as *ground* model elements. Templates are translated to ground model elements by supplying concrete names for replaceable names.

Templates will be defined in the final submission.

5.9.1 Abstract Syntax

Classes for Template and ReplaceableName.

5.9.2 Semantic Domain

Template instantiation can be conveniently expressed using graph transformations. Definition of Graph, Node, Edge, Mapping.

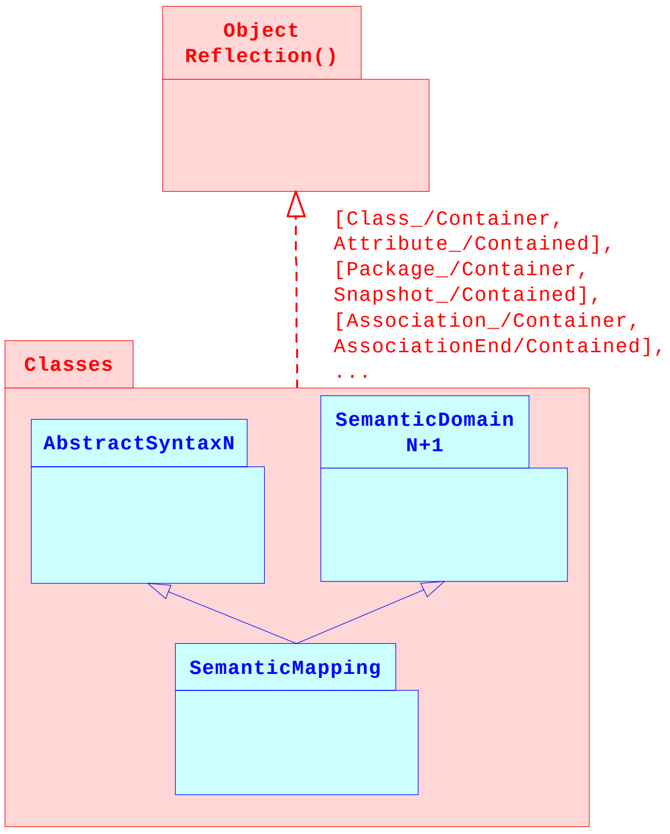
5.9.3 Semantic Mapping

Definition of PackageXGraph; ClassXGraph; AttributeXGraph.

5.10. Reflection

The Static UML Core should be expressive enough to be self describing. In other words, it should be possible to view the core as an instance of itself. This package describes the necessary mapping from core abstract syntax elements to core instances to show that this is feasible.

5.10.1 Templates



5.10.2 Semantic Mapping

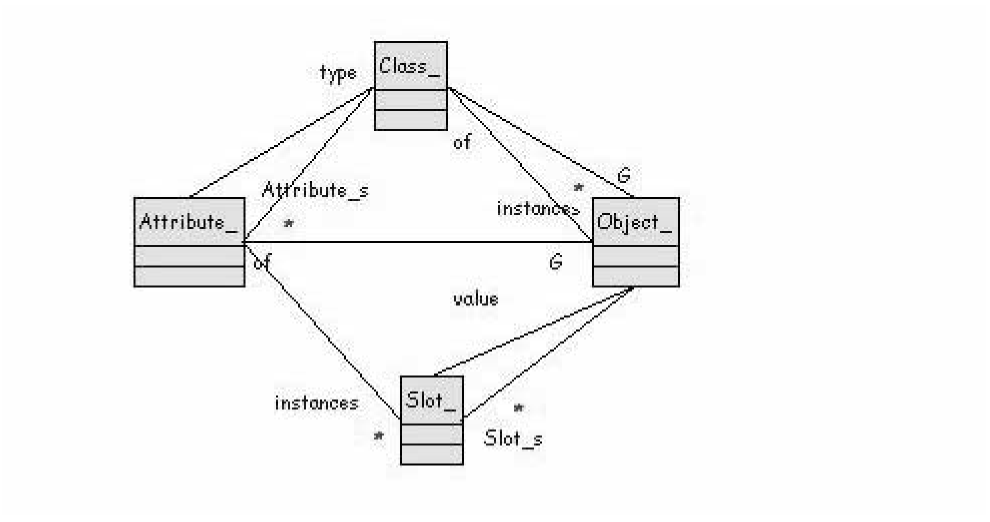


Figure 78. UML2.Static.Reflection.SemanticMapping

Every element in the Static UML Core will have a G mapping to its instance representation at the next level-up in a meta-model architecture. Figure 78 shows the G mappings that are generated for classes and attributes. Here, classes and attributes have a G mapping to a object that is an instance of the Static UML Core. Note, G mappings will be required for other structures, e.g. generalisable elements, named elements.

Well-formednessRules

[1] A class is an instance of the class named “Class”.

```
context Class inv:
  ClasssAreInstancesOfTheClassClass
  G.of.name = Class
```

[2] An attribute is an instance of the class named “Attribute”.

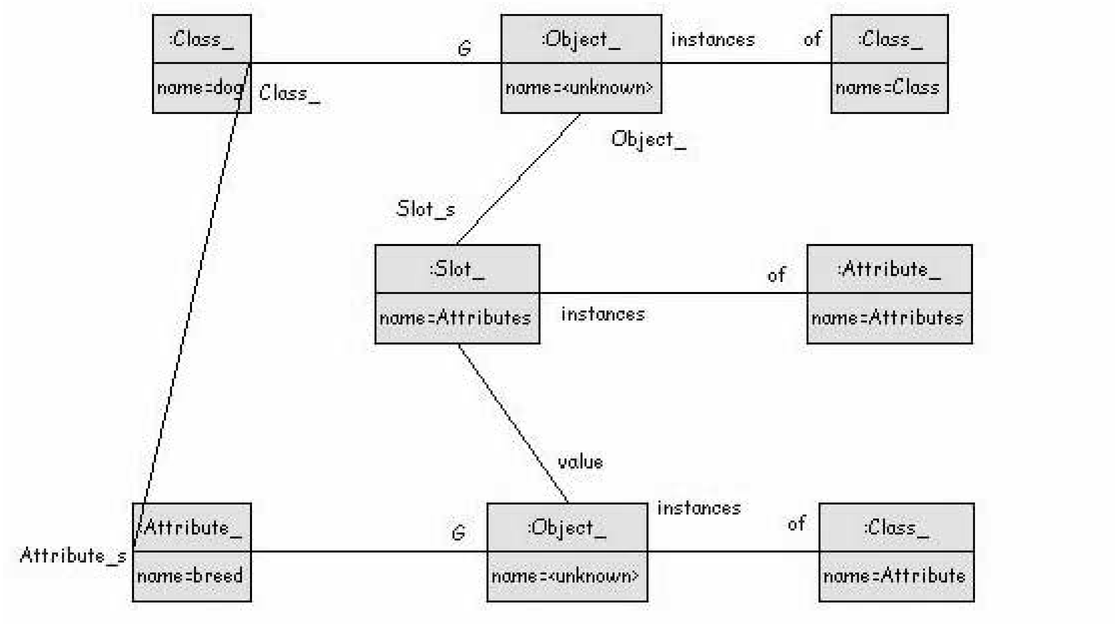
```
context Attribute inv:
  AttributesAreInstancesOfTheClassAttribute
  G.of.name = <Contained>
```

[3] Classes are objects with slots whose values are their attributes as objects:

```
context Class inv:
  self.Attributes -> forAll(c | self.G.slots ->
    exists(s | s.of.name = “Attributes” and s.value = c.G))
```

Example Snapshot

Figure 78 shows that applying G to a class Dog with attribute "breed" maps Dog to a meta-object of class "Class" and "breed" to a meta-object of class "Attribute", with an intermediate slot that maps the dog object to its attribute object.



Chapter 6

Dynamic Core

This chapter describes a primitive language for constructing a family of precisely defined action languages for UML. An example is given of the translation of state machines into the action language as proof of concept.

6.1. Introduction

UML has a variety of notations for expressing dynamic behaviour. The Actions package defines a primitive language for constructing many different action languages. The essential features of dynamic behaviour in UML are:

- Evaluation of simple side-effect free expressions as defined in chapter 5.
- Messages between objects. There are a number of alternative message passing schemes including synchronous and asynchronous. An action may cause a synchronous or asynchronous message to be set. Sub-types of message action must support a wide variety of user defined message passing protocols.
- Object update. Everything in UML is ultimately an object which has an internal state. An action may cause the internal state of an object to change.
- Concurrent actions. Dynamic behaviour in UML supports both *possible* concurrency and *necessary* concurrency. Possible concurrency occurs when the models place no restrictions on the order in which the actions take place. Necessary concurrency occurs when the models require actions to occur concurrently, for example in order to satisfy timing constraints or to satisfy reactivity constraints.
- Sequential actions. An ordering can be imposed on actions. This is in addition to the ordering which arises due to logical data dependencies.

Dynamic aspects of UML models are explained in terms of histories of execution or *calculations*. UML supports concurrent execution and a variety of operation invocation mechanisms. A calculation is a structure that records just the logical dependencies between objects in an historical record of execution. The intended interpretation is that a given UML model is implemented and executed on a class of machines. The machines all execute in terms of objects and actions. A machine will give rise to its own historical records of execution. A machine correctly executes a model implementation when the machine calculations are consistent with the calculations defined for the model by the UML standard.

Certain actions are atomic, for example slot update. Such actions cannot occur concurrently with any other actions at a given object. Given atomic actions and a suitably expressive collection of action primitives it is possible to construct a wide range of control constructs.

6.2. Actions

6.2.1 Templates

To be defined in the final submission.

6.2.2 Abstract Syntax

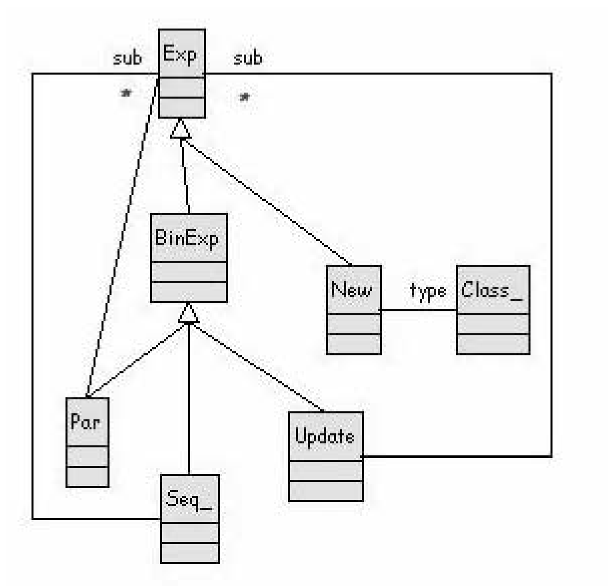


Figure 79. UML2.Dynamic.Actions.AbstractSyntax

The primitive action language is an extension of the primitive expression language defined in chapter 5. All expressions are specialised to become actions. Figure 79 shows the new types of action expressions. The class Par defines parallel actions. The class Seq defines sequential actions. The class Update defines slot update actions. The class New defines object creation.

The collection of action primitives can be extended to develop action languages. Suppose we want to define a class of objects that handle message pools. Clients of the objects send asynchronous messages. Messages are conveyed through the ether and are placed in the message pool of the receiver. Object which handle messages in this way must treat the message pool as a shared resource. Suppose that we extend the action primitives with a flipif primitive. This atomic primitive toggles a boolean value providing the current value of the boolean variable is true:

```
Object::monitor()
  if !pool->isEmpty
  then flipif poolFree;
         let m = pool.selectElement
         in pool := pool->excluding(m);
              handle(m) |
              poolFree := true;
              monitor()
         end
  else monitor()
  endif
```

New messages are handled by ‘receive’:

```
Object::receive(m:Message)
  **flipif** poolFree;
  pool := pool->including(m);
  poolFree := true
```

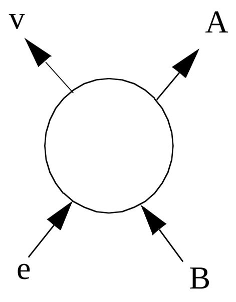
When an object is created it starts the monitor. Calls to ‘receive’ will occur in parallel:

```
Object::init()
  poolFree := true;
  pool := Set{};
  monitor()
```

By extending the basic action primitives with a simple atomic operator it is possible to conveniently ensure that ‘monitor’ and ‘receive can occur concurrently without interference.

6.2.3 Semantic Domain

Actions are described in terms of histories of execution. These are referred to as *calculations*. Chapter 5 defines calculations for expressions with no side-effects. Actions involve changes to object states. Therefore expression calculations are extended with two state descriptions. The *pre* state defines a set of objects required by the action. The *post* state defines a set of objects modified or produced by the action Given a collection of bindings e, a pre state B, a value v and a post state A, a calculation that describes the result of performing an action in the context e and B producing value v and new state A can be drawn as:.



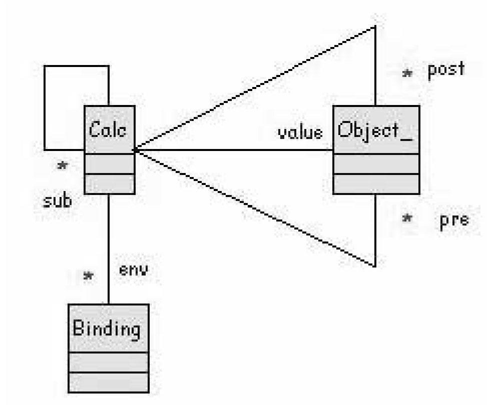


Figure 80. UML2.Dynamic.Actions.SemanticDomain

Figure 80 on page147 shows the definition of the action semantic domain as an extension of the semantic domain for expressions. The pre state of a calculation may contain any objects supplied to its containing calculation and any objects produced or modified by its siblings. The post state of a calculation may contain any objects in the post state of its sub-calculations:

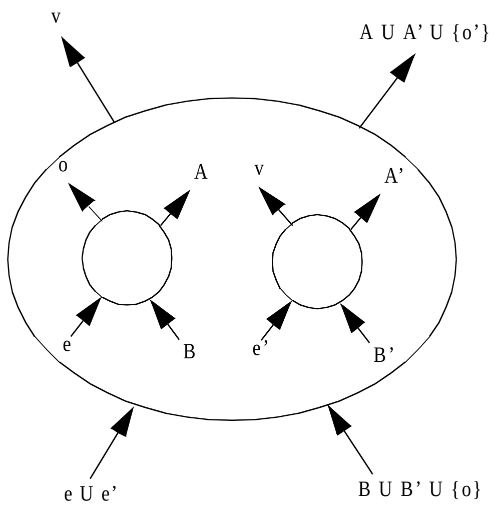
```
context Calc inv:
  pre = sub.pre and post = sub.post
```

Notice that the above constraint rules out calculations that describe inconsistent updates to the same object since objects cannot contain multiple slots with the same name and different values. Therefore, two sub-calculations may change a slot independently providing they change the slot to the same value.

6.2.4 Semantic Mapping

The semantic mapping for actions defines legal calculation instances for action calculations. The definition of expression instances generalises with no further modification to action instances.

Consider the case of an update action. There are two sub-actions which evaluate to produce an object o and a value v respectively. The update action causes the value of a slot named s to be updated to the value v in the object o. This produces a an object o’ which is exactly the same as o except that the slot named s has changed. The update calculation has the following form::



This is expressed as a constraint as follows:

```
context Update inv:
  instances->forAll(c |
    value = sub->at(1).value and
    sub->at(0).post->union(sub->at(1).post->subSet(post) and
    objectChanged(sub->at(0).value,post,name,sub->at(1).value)
```

where ‘objectChanged’ relates the objects o and o’ on the diagram above.

A new action simply produces an instance of an object. The well-formedness constraints for action calculations will require the instance to be unique:

```
context New inv: instances->forAll(c |
  type.instances->includes(c.value) and
  c.post = Set{c.value})
```

Sequential composition of actions requires an ordering on the processing of states:

```
context Seq inv: instances->forAll(i |
   i.sub->at(1).pre = i.sub->at(0).post)
```

There are no further constraints on Par action instances other than those required by the well-formedness constraints on calculations. Therefore Par actions are free to be performed concurrently.

6.3. Operations

UML operations are equivalent to queries (see chapter 5) except the body of the of the operation is an action. A class is a container and inheritor of operations. An operation has a number of parameters and a return type. An example of a simple operation is People::addPerson which adds a person to a collection of people. Using a simple concrete syntax based on OCL with attribute update:

```
Person::addPerson(p:Person) people := people->including(person)
```

6.3.1 Templates

To be defined in the final submission

6.3.2 Abstract Syntax

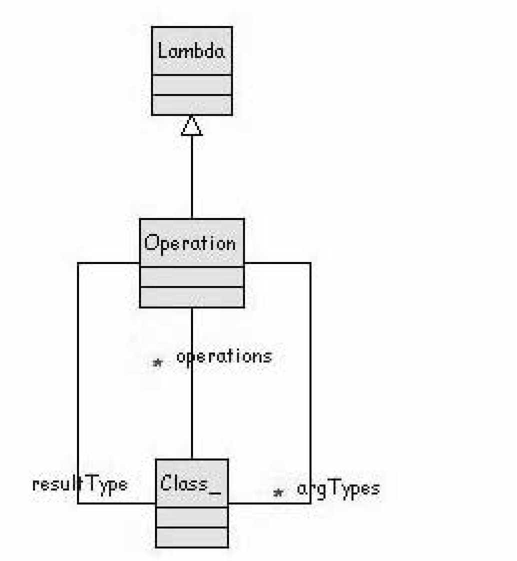


Figure 81. UML2.Dynamic.Operations.AbstractSyntax

Figure 81 shows the definition of the operations abstract syntax. Class defines a query ‘allOperations’ which returns the local operations defined by the class and those that it inherits from its parents.

6.3.3 Semantic Domain

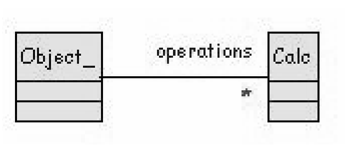


Figure 82. UML2.Dynamic.Operations.SemanticDomain

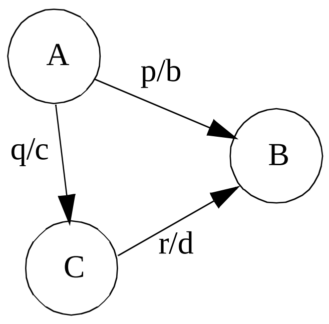
Figure 82 shows the semantic domain for operations. Each object has a collection of calculations which arise from performing operations. The context for the calculations includes the slots of the object and the variable ‘self’ which is bound to the object.

6.4. State Machines

Dynamic behaviour can be expressed in UML using state machines. A state machine is associated with a class and consists of a collection of states and transitions. This section describes how state machines can be given a precise semantics using a translation to an operation. This is an example of *translational semantics*. Translational semantics is to be contrasted with *model based semantics.* A model based semantics defines three components: a syntax domain; a semantic domain; and, a semantic mapping. Instances of the semantic domain provides a *model* for instances of the syntax domain. A translational semantics involves three components: a source syntax domain; a target syntax domain; and, a translation from the source to the target. The source language is defined entirely in terms of a translation to the target language. No new semantic domain elements are introduced for the source language.

A state machine is defined using a translation to an operation. The operation handles messages. The operation has a test for each transition of the machine. Each test involves a check for the current state of the object, the message and performs the action associated with the transition.

This section is intended to show how a simple model of state machines can be translated to operations. Consider the following state transition machine associated with a class X:



The machine can be translated to an operation as follows:

```
C::machine(m:Message)
  if inA()
  then if m = p then b endif | if m = q then c endif
  endif;
  if inC()
  then if m = r then d endif
  endif
```

6.4.1 Abstract Syntax

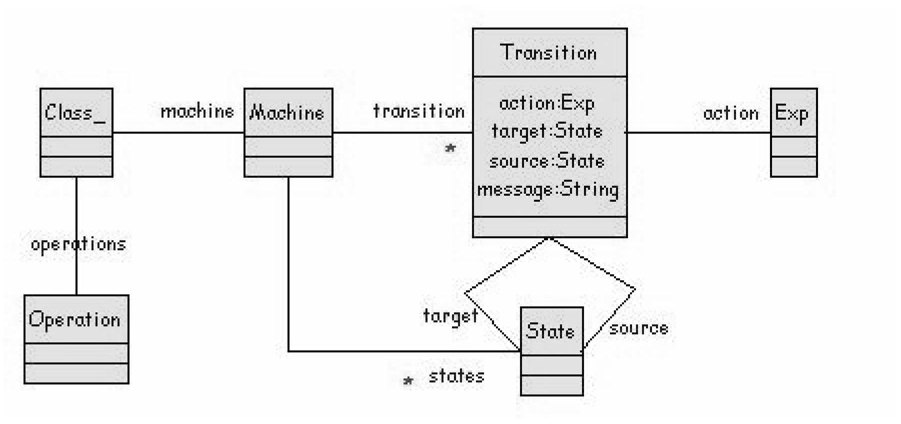


Figure 83. UML2.Dynamic.StateMachines.AbstractSyntax

6.4.2 Translation

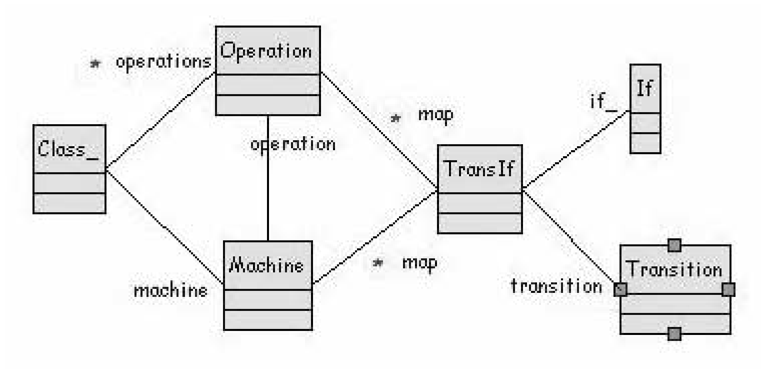


Figure 84. UML2.Dynamic.State<achines.SemanticMapping

Figure 84 on page 152shows the translational semantics for state machines. Each machine is associated with an operation. The machine is to be viewed as *sugar* for the operation. The class TransIf defines pairs consisting of a machine transition and a conditional expression. The test of the conditional expression is a conjunct of a test for the source state and a message. The consequent of the conditional expression is the action of the transition:

context TransIf **inv**:

```
    if.test.andLeft.name = transition.source.name and
    test.andRight.eqlLeft.name = “m” and
    test.andRight.eqlLeft.value = transition.message and
    consequent = transition.action
```

References

[Alvarez01a] Alvarez A, Clark A, Evans A. and Sammut, A. (2001) *An Action Semantics for MML,* in [Gogolla01].

[Alvarez01b] Alvarez A, Evans A. and Sammut, A. (2001) *Mapping Between Levels in the Metamodel Architecture*, in [Gogolla01]

[Atkinson01] Atkinson C. and Kühne T. (2001) *The Essence of Multilevel Metamodeling,* in [Gogolla01].

[Catalysis00] http://www.catalysis.org/publications/papers/UML-Infrastructure.pdf

[Clark99] Clark A, Evans A, Kent S, France R. and Rumpe B. (1999) *pUML Response to UML2.0 RFI,* available from www.puml.org.

[Clark00] Clark A, Evans A. and Kent S. (2000) *Re-architecting UML as a Family of Languages using a Precise Meta-Modeling Approach,* available from www.puml.org.

[D'Souza98] D'Souza D. and Wills A. (1998) *Objects, components and frameworks with UML*, Object Technology Series, Addison-Wesley.

[D’Souza99] D'Souza D, Sane A. and Birchenough A. (1999) *First-Class Extensibility for UML - Packaging of Profiles, Stereotypes and Patterns*, in [France99], (1999).

[France99] France R. and Rumpe B. (eds.) (1999) *Proceedings of UML'99 - The Unified Modeling Language, Beyond the Standard: Second International Conference*, Fort Collins, CO, USA. LNCS 1723, Springer Verlag.

[Gogolla01] Gogolla M. (ed.) (2001) *Proceedings of UML'2001 - The Unified Modeling Language: Fourth International Conference*, Toronto, Springer Verlag

[Kleppe01] Kleppe, A. Warmer, J. *Unification of Static and Dynamic Semantics of UML. A Study in redefining the Semantics of the UML using the pUML OO Meta Modelling Approach*, available at www.puml.org.

[OMG] Object Management Group (1999) *OMG Unified Modeling Language Specification*, *version 1.3.*, found at: http://www.rational.org/uml.

[OCLWK00] UoK (2000), Workshop on OCL, University of Kent, details available from http://www.cs.ukc.ac.uk/research/sse/oclws2k/index.html.

[Reggio01] Reggio, G. and Astesiano, E. *A Proposal for a Dynamic Core for UML Meta-Modelling with MML*, available at ftp://ftp.disi.unige.it/person/ReggioG/ReggioAstesanio01a.pdf