A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach

Version 1.0. September 2000

Tony Clark, Andy Evans, Stuart Kent
(for pUML group)

Steve Brodsky, Steve Cook
(for IBM)

Contents

Executive Summary

This report describes a feasibility study in rearchitecting UML. It develops a theory of precise OO meta-modeling in order to fulfil this task, and checks the feasibility of that theory by developing the meta-model of various aspects of UML.

The report is organized as follows.

Chapter 1 rehearses the arguments for rearchitecting UML. It concludes with some requirements for a rearchitected definition.

Chapter 2 introduces an approach to precise OO meta-modeling. Some effort is made to relate this to the OMG's Meta-Object Facility - MOF, though this stops short of providing an exhaustive comparison. Based on the requirements for rearchitecting UML, some requirements of a precise OO meta-modeling facility are identified, and an approach to meeting these requirements, with regard to notations and tools, is outlined.

Chapters 3-6 support the theory with the details. Chapter 3 describes the structure of the UML as a family of languages, notated with the package constructs that are part of the OO meta-modeling language, MML. MML is itself a member of the UML family. We note that circularity in the definition of MML is avoided by providing an external characterization of the MML package, for example by implementing it as a tool. Certain essential patterns of packages and their relationships for language design are identified. Specifically, patterns are identified for separating models from instances, where semantics is the mapping between the two, and syntax from concepts (similar to abstract syntax for textual languages).

Chapter 4 defines those key packages in the UML family that contribute to the definition of MML, maintaining the focus on model concepts, instance concepts and semantics.

Chapter 5 gives examples of various syntax to concepts mappings, and shows how the same syntactic components may be mapped to different concepts, how syntax mappings may be extended as the set of concepts is extended, and how XMI may be thought of as just another syntax mapping.

Chapter 6 outlines how dynamics aspects of the UML, including semantics, may be defined using this approach.

Chapter 7 reflects on this feasibility study and concludes by identifying what next steps need to be taken to complete the work begun here. In particular, there is some discussion on how precise OO meta-modeling could be incorporated into the standardisation process, considering issues such as checking conformance to a standard, and the acceptance checks for a meta-model must include before it can be considered for standardisation.

Version 1.0, 11th September 2000:

Chapter 2 (previously Chapter 1) has been rewritten and considerably reduced in size. In particular, the detailed descriptions of the various tools were deemed too implementation-specific and have been replaced by much higher-level descriptions.

The remaining chapters have been reorganized. A new chapter 3, describes a proposed new architecture for UML 2.0, based around the notion that UML is a *family* of languages.

Chapter 4 (previously chapter 2) has been reworked to take into account that MML, the meta-modeling language previously referred to as mofk2, is part of the UML family.

Chapters 5 & 6 replace chapter 3 in a previous version, demonstrating the feasibility of using MML to define syntax and dynamic aspects, respectively. These chapters report work that is less developed than the work on MML.

The previous chapter 4, on tools, has been deleted. A prototype tool is being developed and this will be described and delivered separately. The tool (and this report) should be available in the near future from www.puml.org.

Chapter 7 has been added to reflect on the feasibility study, to outline how the work should proceed, including known issues and problems, and to indicate how precise OO meta-modeling may be usefully employed in the standardisation process.

A tool, MMT, has been constructed which implements the features of MML. MMT will be/is available from www.puml.org, and has some associated documentation. A future evolution of this report will include a write-up of the tool.

Version 0.1, 22nd May 2000:

Drafts of Chapters 1 & 2. No work done yet on Chapters 3 & 4. The alignment of Chapters 1 & 2 needs to be completed. We fully expect that the model described in Chapter 2, and outlined as part of the theory in Chapter 1, will need to be refactored as tools are developed.

Chapter 1

Rearchitecting UML

This brief chapter overviews the current definition of UML and rehearses the arguments, which have been set out elsewhere [Clark et al. 1999], for rearchitecting that definition, using a more precise, systematic and flexible approach.

1.1. Definition of UML 1.x

The UML 1.x family of definitions (e.g. [OMG 1999])each comprise the following main components:

- A description of the concrete syntax (i.e. diagrams), in terms of prototypical examples and natural language explanation.
- A description of the semantics, comprising a meta-model description of the abstract syntax and an English description of the semantics. The meta-model description of the abstract syntax is given in terms of UML class diagrams with associated OCL expressions, where class diagrams and OCL have themselves been defined in a similar manner, the meta-model being replaced by a BNF syntax for OCL.
- The beginnings of descriptions of key profiles of UML, comprising a list of the language constructs used in the profile and the stereotypes that may be applied to those constructs. In the 1.3. standard the profiles included are for general-purpose software development and for business modeling.

The introduction of profiles recognizes that, as the popularity of the UML increases, so does the pressure to include new features that are particular to specific application domains or styles of modeling activity. Uncontrolled evolution of the UML can lead to a bloated notation that is difficult to learn and apply in practice. Profiles are intended to allow one to develop variants of the UML suited for particular purposes. There are essentially two kinds of profile. *Horizontal* profiles focus on particular styles of modeling such as requirements, architecture design, software specification, business modeling, implementation design and so on. Vertical profiles focus on specific domains, ranging from quite general purpose (e.g. real time) to far more specific (e.g. the automotive industry). Vertical profiles will select and specialize fragments from horizontal profiles. Some profiles will be the cross-product of the two, for example requirements modeling for real time.

Our thesis in this report is that the current approach to defining UML, especially in the context of profiles, is inadequate. The next section provides evidence to support this thesis.

1.2. Why is the definition of UML 1.x inadequate?

We consider three key questions, for which, we believe, the current definition has no response.

How can conformance to the definition be checked?

Currently, the only way of checking that a tool or method conforms to UML definition, is by inspection: compare the source code/test results of the tool with the syntax and semantics definition on paper. This is certainly infeasible without having a method for systematically transforming the source code/test results to something that can be compared directly with those definitions, and probably infeasible without automating this mapping and the checks in some way. In order to systemize or automate conformance checking the definition needs to be far more precise than it currently is. In particular:

- The mapping of concrete syntax to abstract syntax (the meta-model) needs a more systematic and unambiguous definition, which in turn probably requires the definition of the concrete syntax itself to be more precise and generic (i.e. not just prototypical examples).
- The semantics must be more than an English definition, conformance to which is totally reliant on inspection and human interpretation.

How can we be sure that the definition is self-consistent and is comprehensive yet lean?

Obviously it is important that the definition of UML is self-consistent, that one part of the definition does not contradict another. It should be comprehensive, fully covering the definition of concrete and abstract syntax as well as semantics. On the other hand, it should be lean in the sense that redundancy of concepts is kept to the minimum. [Clark et. al 1999] cites evidence where the current definition is weak with respect to these aspects. Specifically:

- The informal description of concrete syntax, especially the use of prototypical examples, is not comprehensive, and the lack of a fully-documented systematic mapping from concrete syntax to abstract syntax (meta-model) makes this prone to error and ambiguity. It has also allowed aspects of concrete syntax to creep into the abstract syntax, as there is no clear separation between the two.
- The informally described semantics is also not comprehensive, and leaves room for error and inconsistency, as well as failure to properly relate and sometimes integrate concepts, leading to redundancy.

The specialization and extension of UML continues all the time. How does the existing definition allow this to be managed and controlled?

The proliferation of profiles suggests that UML is not a single language, but a continually evolving family of languages. The challenge is to develop an approach to definition that supports an architecture which facilitates the graceful evolution of the UML family.

The danger with the current definition of UML is that it will develop into an unmaintainable monolith. The danger is increased with its approach to profiles of subsetting and specialization. Subsetting implies that the definition must include all concepts that are ever likely to be required in any profile, suggesting that as profiles continue to be developed the monolith will grow and it will be even harder to manage, including maintaining backwards compatibility with previous versions. The current approach to specialization using stereotypes is quite shallow, unless supported by specializations in the abstract syntax and semantics. But how this is done is not sufficiently defined.

Instead, an additive and evolvable architecture is required. To achieve this, it must be possible to break the monolith into focused, clearly separated fragments which can be specialized/extended independently, and assembled in different configurations to form profiles. The relationships between profiles and language fragments should be explicit and unambiguous so that the impact of proposed refactorings and extensions can be identified simply and minimized where feasible.

1.3. Requirements for a rearchitected definition of UML

Three key requirements for a rearchitected definition of UML can be deduced from the preceding discussion:

1. It should be precise to the degree that conformance can be checked systematically, without argument and, preferably automatically, and that self-consistency of the definition can be established.
2. It should be comprehensive, covering syntax, both concrete and abstract, and semantics. On the other hand, redundant and overlapping concepts should be kept to a minimum.
3. It should accept that UML is a family of languages, providing mechanisms that allow profiles and language extensions to be defined in a controlled and managed way, and which makes the relationships of profiles and extensions to existing language fragments explicit and unambiguous.

A fourth requirement, which is met by the 1.x definitions to a large degree is that:

4. The definition should be accessible to tool builders and those involved in the standardization of the language.

In other words, the definition will be of little use if it is phrased in a language (for example, a complex piece of mathematics) that those who need access to it can not understand. That does not mean that it might not be supported by equivalent definitions in such a language, if there is a good reason for doing this, such as provision of sophisticated tools such as model checkers and theorem provers.

The next chapter argues that these requirements can be met by a precise object-oriented meta-modeling approach to language definition. Subsequent chapters then test the feasibility of that approach.

Chapter 2

Precise OO Meta-Modeling

This chapter outlines a framework for precise OO meta-modeling. This framework is motivated by a discussion of the benefits to be gained from a precise, OO approach, specifically with regard to the rearchitecting of UML.

2.1. Motivation

2.1.1 Why OO meta-modeling?

The OMG has adopted an OO meta-modeling approach to the definition of UML. Essentially this means using a subset of UML to define UML. This has proved itself to be an intuitive way of defining the conceptual basis (or abstract syntax) of UML.

We have started investigating the use of an OO meta-modeling approach to define the (concrete) syntax and semantics of languages, and have found that, for a certain class of languages, including UML, it is both feasible and intuitive, provided some important features, such as a constraint language, are included.

In this study, our focus is on the precise definition of languages, with particular reference to the definition of UML. In this chapter we isolate the requirements of a OO meta-modeling language (MML), as dictated by this focus. The remaining chapters give a definition of MML in terms of itself, and show how MML can be used to characterize significant fragments of UML, using an architecture that explicitly recognizes that UML is not a single language, but a family of languages.

2.1.2 Why precise? What about tools?

By a precise MML, we mean the following:

- There is a single, precise reference definition of MML to which all other definitions must conform.
- MML syntax, concepts and semantics are included in this definition.
- There should be tools which faithfully implement or embed the precise definition.

Precise does not mean that there need be an external mathematical definition of the language. The part most frequently missing from definitions of a language, and which is missing from UML 1.3, is a precise semantics. We see it as fundamental that this is defined. In one approach the definition of MML is embodied in a tool which checks conformance of models to meta-models defined in the language. For MML, this essentially encodes a model-theoretic semantics, which states what models are valid instances of any particular meta-model expressed in MML. Then MML (syntax, concepts and semantics) is defined in itself, and put through the tool. There is then a cycle where the model and tool are improved in tandem as one's confidence in the definition of MML increases. The definition of MML in itself can become the reference definition, and tools implementing the translation of syntax to concepts, for example, would be based on this definition. Indeed, it may even be possible to (partially) automate the construction of the latter.

If we have a precise MML, that is expressive enough to define syntax, concepts *and* semantics, then meta-models defined in MML will also be precisely defined, in the same fashion as MML itself. In particular, we expect that the checking tool embodying the semantics of MML can be used to realise the semantics part of any meta-model defined in MML, in the sense that it can be used to check that instances of a model conforming to that meta-model (e.g. instances of some UML model) satisfy the definition of that model. This will work provided the meta-model (e.g. of UML) includes a semantics component that can be defined fully in MML.

An idealised meta-modeling facility (MMF = MML + tools) is outlined at the end of this chapter.

2.2. Design Principles for MML

MML is for defining languages. Experience of language definition in general, and the UML in particular, suggests the following design principles, which, if followed, discharge at least the requirements for an approach to the definition of UML set out at the end of the previous chapter.

1. MML should be able to describe syntax (both textual and visual, as in the UML), concepts and semantics, including well-formedness rules and mappings.

   A language definition comprises two key orthogonal distinctions.

   The first is the distinction between *model* and *instance*. The model part comprises a definition of the valid expressions in the language. The instance part comprises a definition of the possible situations that expressions in the language may denote at any point in time. For example, the model part of the definition of the language of class diagrams defines what are valid class diagrams and what are not. The instance part defines the valid object configurations (visualized as object diagrams) which can be denoted by a class diagram at a point in time.

   The second key distinction is between *syntax* and *concepts*, and this distinction applies to both models and instances. Concepts are the essential concepts of the model or instance part of a language. So the concepts underpinning class diagrams are class, attribute, association, cardinality etc., and those underlying object configurations are objects, slots and links. A syntax for a set of concepts is a model of a particular rendering of those concepts so that they may be written down, displayed, read etc. A syntax for class diagrams could include boxes, lines, strings and so on. Instances also have syntax: an object diagram is a syntax for the object configurations.

   Syntaxes may be more or less concrete than other syntaxes. So a syntax involving only pixels on a page or screen is more concrete that one involving boxes, lines and strings.

   There may be many syntaxes for each set of concepts, and a syntax may be shared between two different sets of concepts. For example, sequence diagrams in UML may be regarded as a syntax for specifying the allowed action sequences in a model, or as a syntax for visualizing a single execution trace (i.e. an instance).

A language definition is not complete unless its various aspects are properly related. The relationship between the model part and the instance part represents the semantics of the language. In the tradition of denotational semantics in the programming language field, and model-theoretic semantics for logics, it gives the rules for when an instance is a valid instance of, *satisfies* a model and when it does not. It is sufficient to define semantics as a mapping between the model concepts and instance concepts.

It is also necessary to stipulate how the syntax(es) map into the concepts. An implementation of this mapping in the direction of syntax to concepts is essentially a parser. An implementation of the mapping in the other direction is usually harder, as one has to supply layout information – notoriously difficult for graphical languages.

To summarise, MML must include constructs to define the model and instance parts of a language, to define syntax(es) and concepts for each of these parts, and to define the mappings between all these components.

2. MML should be accessible to stakeholders in the activity of language design. For UML this means those actively engaged in defining UML, and those involved in providing tool support. Users of the UML notation and tools may be presented the definition in a form that facilitates learning of the notation, for example a catalogue of positive and negative examples.
3. MML should provide explicit support for the definition and evolution of language families. There is strong evidence that UML is not a single language, but a family of languages. Every organization seems to use its own UML ‘profile’ – a subset of the language, extended with organization-specific stereotypes. Some ‘standard’ profiles are already on the drawing board. There is no evidence to suggest that UML will not continue to evolve in this way.
4. MML should be precise and unambiguous, so that definitions in MML are precise and unambiguous. The reasons for this have already been explained in Section 2.1.2.
5. MML is itself a language. Thus the definition of MML should distinguish between syntax, concepts and semantics, and these should be defined in a precise and unambiguous way. Or, to put it another way, it should be possible to define MML in itself. At the very least this will be a good test of the capabilities of MML.
6. MML should be a declarative language. Issues related solely to the implementation of a repository to store language definitions (e.g. UML) and expressions of those languages (e.g. a UML model) need not be part of MML. The language designer does not need to know how tools to support that language are implemented, or be unnecessarily exposed to features in the language which have no purpose other than to support the implementation of tools such as a repository. On the other hand, we recognize the importance of implementation, and hope and expect that MML would provide the kernel of a model specifying one or more tools implementing aspects of MML.
7. MML should support reflection. That is, it should support the treatment of a definition, a model elements, as data, an instance of a model element. It is expected that considerable gains are to be made from building reflection into MML. For example, if one develops a tool that checks instances against model elements, reflection can

be used with this tool to check UML instances against UML models, UML models against the UML meta-model, and the UML meta-model against MML itself (remembering that we should be able to describe MML in itself, i.e. as a model).

In addition, there are some general principles that should apply to any language, including MML:

8. A language should be coherent and consistent.
9. A language should keep the number of concepts used to a minimum. Preferably, there should be a small set of orthogonal core concepts, on top of which other concepts can be defined.
10. A language should not include unmotivated or unnecessary restrictions, that is restrictions which seem to be there on whim.

2.3. An idealised Meta-Modeling Facility (MMF)

It is possible to distil from the preceding discussions some requirements for a meta-modeling facility, which is a meta-modeling language plus its support tools.

2.3.1 Meta-Modeling Language (MML) requirements

In order to define the various components of a language (syntax, concepts, model, instance) and the mappings between them the following two requirements are essential:

Constructs for specifying object structures. Class diagrams are essential to define the vocabulary used to define object structures.

Constructs for expressing well-formedness constraints on object structures. Central to writing out the full definition of syntax, concepts, semantics and the mappings between them is to have a language that is up to the job of writing subtle constraints on object structures. The Object Constraint Language that is part of UML is just such a language; it is essentially a variant of FOPL tuned for writing constraints on *object* structures.

To support (a) the separation of concerns (syntax, concepts, semantics) in a language definition and (b) definitions of language families, such as UML, MML must include:

Constructs for packaging and composing fragments of language definition. Specifically, packages, package generalization, and realization/translation relationships between packages.

So that generic tools can be built, MML requires:

Constructs to support reflection. Something can be viewed as both model and instance. The class `Association` in the UML meta-model is both an instantiable thing - it can have instances - and an instance - it is an instance of the MML `Class` construct. Similarly the UML class `Library` is both instantiable and an instance, in this case of the class `Class` in the UML meta-model.

2.3.2 Tool requirements

Little has been said, so far, about the tools required to support MML. It is our firm belief that a meta-modeling language is little use without the tools to support it. Further, as our approach is to define MML using itself, it is important to provide a method that increases our confidence that the definition of MML is correct.

One approach to this would be to provide a definition of MML in a language which already has a venerable past and in which we (and many others) have confidence, such as the language of mathematics.

Another approach is to define a working model of that language which can be tried and tested with examples. The more examples for which it works as expected, the greater our confidence will be that the definition is correct. Even better, if that working model is generated automatically from the description of MML in itself. And of course, a working model can be used to perform useful tasks like help define new languages using MML. This working model is the set of tools we propose to build. Following is a list of tools we expect such a toolset to include. Of these, we see the first two as most important.

Satisfaction checker

Check that a given instance satisfies a model. A key component of this is to provide a checker that is able to take account of OCL-style constraints: does instance x satisfy constraint c from model m?

Combined with reflective ability, such a tool can be used to perform a number of checking tasks. Using the 4-layer architecture as a reference point, a model can be viewed as an instance of a meta-model, so we can check that a model satisfies its meta-model definition. We can also check that a meta-model satisfies the rules of the MML itself. MML is itself a meta-model, and so meta-models are instances (models) of this meta-model. Finally, MML is an instance of itself, so its well-formedness can be checked against MML. Thus a satisfaction checker is an essential tool in helping us become more confident that MML is correctly defined.

Initially one would implement the checker for MML or some appropriate subset, and this would then work with any language whose concepts could be mapped back to MML concepts. However, eventually it would be desirable to have a satisfaction checker for any language that defined a model part and an instance part. Preferably, this should be generated from the definition of that language in MML, or, alternatively, there should be a generic tool that is able to adapt itself according to the language definition in use (see next tool).

Instance generator

Given a model, generate an instance that satisfies it. Again, the ability to take account of OCL-like constraints is essential.

Such a tool could be used, for example, to generate an instance of a mapping between one language component and another. For example, if one has a model describing the mapping between the syntax and concepts of, say, class diagrams, one could use this tool to generate an expression of the concepts part (specific classes, attributes and associations) from a collection of boxes, lines and strings (the syntax part), by generating a valid mapping.

If this tool has been implemented for some MML, this tool could be used to implement a satisfaction checker for any language whose semantics can defined in MML (the latter condition means that the MML in question must include an OCL-like constraint language). By providing a description of the semantics using MML, and the constraint language in particular, the proposed tool could be used to generate anything that is missing to complete the mapping of a given instance and model. If the mapping can't be completed, then checking fails.

In general, it is expected that an instance generator will never be fully automatic. For example, consider the mapping from concepts to a graphical syntax. That mapping is unlikely to specify the exact layout of diagrams at the syntax end, so will be underdetermined in the sense that many diagrams (with the same boxes and lines, but different layout) would map to the same concepts. To cater for such cases, a mechanism must be provided whereby the additional information can be supplied as required, either from the user of the tool or by virtue of another program (e.g. a layout algorithm).

Graphical editor generation/configuration

The main purpose of MML is to provide a more complete definition of UML organised as a family of languages. Many of the key notations in UML are graphical. It would be desirable to have a means of generating graphical editors for a graphical syntax defined in MML, or, alternatively, of configuring a generic editor for a particular MML-defined language. Research is continuing elsewhere to establish the feasibility of this and how it might be done. An important aspect to keep in mind is the useability of the resulting editor – the editor controls must match the concepts that the diagrams are intended to represent, not the diagrammatic symbols themselves.

Model interchange and XMI

In both the UML and MOF 1.x standards, XMI is used for model interchange. In this section we introduce XMI and argue that it should be treated as "just another syntax".

What is XMI? There are two kinds of XML documents: datatype definitions, files with extension .dtd and XML documents, with extension .xml. XML documents conform (or not) to data type definitions. A data type definition essentially defines the tags and attribute values of tagged elements that can be used in a conforming XML document. XMI is a standard for representing MOF and UML models in XML, and includes various tools for manipulating XMI documents.

There are two key .dtd's included in XMI: mof.dtd and uml.dtd. There is a tool that will generate X.dtd from a document X.xml conforming to mof.dtd. We understand that uml.dtd was generated in this way. There is a document mof.xml conforming to mof.dtd, and changes to mof.dtd could be made by generating it from a new version of mof.xml. Documents conforming to mof.dtd represent MOF-compliant meta-models; documents conforming to uml.dtd represent UML models.

We understand that there is a tool that will take X.xml conforming to uml.dtd, but which only uses a subset of UML – essentially class diagrams – and generate Xasmof.xml conforming to mof.dtd. No doubt this makes use of the close correspondence between (a subset of UML) and MOF.

Currently XMI does not include a .dtd governing xml documents representing *instances* of UML models. This is not a failing of XMI, but rather a consequence of the fact that the UML meta-model does not encode the concept of model instance. Encoding a concept of model instance is a major step to encoding *semantics*, i.e. the relationship between instances.

Similarly, there is also nothing in XMI to represent concrete syntax (e.g. boxes, compartments and lines on a class diagram), including layout information. Again, this can be remedied by including a definition of syntax(es) in the meta-model, and its (their) relationship to the concepts (and to each other). The .dtd defining the tags to carry this information could then be generated using existing tools, allowing XML documents to be written encoding this information.

XMI and MML. The first point to recognize is that XML is just another syntax. When defining MML in itself, we should be able to provide a model of the XML syntax and a mapping of that model into the model concepts part of the definition of MML in itself. The effect of this would be to provide a definition of the XML representation of any meta-model defined using MML. That XML representation could take the form of an .xml or a .dtd document – a model of and mapping to each could both be defined.

As MML is itself a meta-model, this would also serve to define the XML representations for MML.

Thus by treating XML as another syntax we get all the existing features of XMI in the current standards. Combine these with the tools suggested above, we get a more powerful feature set. The satisfaction checker is a much stronger checking tool than the rather weak checking provided by XML, in the guise of checking an .xml document against its corresponding .dtd. Even when XMI is revised to replace .dtd's with XML schemas, the checking will still be weak in comparison. The proposed generation tools will provide all the generation facilities included in XMI, and more.

In addition, as language definitions in MML will include an instance part, as required to define semantics, an XML syntax can br provided for that also. Thus, again be treating XML as just another syntax, it will be possible to produce XML representations of instances of models.

Finally, it is also possible to give a syntax a syntax. So, for example, one can define a mapping between a model of XML and a model of box and line diagrams. In this way the representation of concrete, graphical syntax in XML can also be defined and generated using the tools proposed above.

2.4. Relationship of MMF to MOF

The Meta-Object Facility (MOF) is a language and a set of tools for defining meta-models and meta-data repositories. Currently, UML is defined indirectly using MOF:

there is an encoding of the abstract syntax of UML in MOF which is used to generate IDL and XMI. The OMG has a goal to define UML directly in MOF (i.e. for there only to be one meta-model, not two) and for MOF to be (isomorphic to) a subset of UML.

A key goal of the work described in this report is for UML to be defined in MML and for MML to be a subset of UML, in other words a member of the UML family of languages.

We have not conducted a detailed comparision of MOF and MMF. However, it is worth making the following observations:

- MOF 1.3. does not clearly separate those features targeted on language design and those features targeted on repository generation.
- If the features targeted on language design could be extracted, it seems that there are some problems with the package extension, nesting and reference mechanisms, there is a need to integrate OCL properly, and the reflective aspects could probably be improved.
- If one accepts the argument that UML should be a family of languages, then rearchitecting the definition of MOF so that it is a member of the UML family would be desirable.
- There has been no attempt to separate out the different aspects of a language definition (syntax, concepts, semantics) both in the use of MOF to define languages and in the definition of MOF itself.

Given these observations, it seems that the two goals of

- defining UML directly in MOF and making MOF a subset of UML, and
- incorporating the expressive power of MML in distinguishing between syntax, concepts and semantics,

could only be achieved with major revisions to MOF. An alternative approach would be to define, in MML, a (two way) mapping from MML to MOF and thereby retain MOF in more or less its current form. This is very important given the many existing implementations of MOF. Further work is needed to construct such a mapping.

2.5. Summary

Driven by the needs of language design in general, and the rearchitecting of UML in particular, we have proposed a meta-modeling facility (MMF), which combines a meta-modeling language (MML) with the tools to support that language. The goal is for language designers to define languages as expressions in MML, assisted by tools for editing and managing language definitions, and for users of those languages to be provided, preferably automatically, with language-specific tools based on those definitions. The latter can be via one of two routes: generic tools that work with any language definition, or generation of tool(s) from the language definition.

In the remainder of this report we focus on defining MML itself, as one of the UML family, and then show how MML can be used to define other members and other aspects of that family. A prototype tool, which implements some of the facilities outlined above, is being developed to support MML and should be available shortly.

Chapter 3

An Architecture for UML

This chapter proposes a new architecture for UML, based on the central tenet that UML represents a family of languages. Packages, package generalization and containment are used to provide layering of languages units, and to separate concerns, for example distinguish between model and instance and between syntax and concepts. Some patterns of packages and their relationships, which seem to recur in the language design, are identified.

3.1. Introduction

The architecture proposed in this chapter takes the idea that UML is a family of languages as a central tenet. Packages are used to separate out components of UML in two ways.

- By *subject area*, for example those aspects at the core of static and dynamic modeling, respectively, a constraint language, those aspects dealing with model management, and many more. Language components build on top of more basic components. For example, constraints are built on top of a static core. Package specialization is used to achieve this kind of layering. At some point a language component becomes a language that is recognized as having a particular use, such as MML.
- By *language aspect*, maintaining a separation between syntax and concepts, and between model and instance, as discussed in Chapter 2. Language aspects require mappings between them. Thus syntax needs to be mapped to concepts and models mapped to instances (semantics). Mappings are expressed as packages which specialize the packages representing the two sides being mapped.

The aim is to define the UML family, including MML itself, in MML. At this stage in the report, and when we started out on this work, a definition of MML does/did not exist. In Chapter 2, we suggested an approach where the definition of MML is embodied in a tool which checks conformance of models to meta-models defined in the language. For MML, this essentially encodes a model-theoretic semantics, which states what models are valid instances of any particular meta-model expressed in MML. Then MML (syntax, concepts and semantics) is defined in itself, and put through the tool. There is then a cycle where the model and tool are improved in tandem as one's confidence in the definition of MML increases.

Essentially this is the approach we have taken, though it still remains to detail how we get to the first embodiment of MML in a tool. As indicated in Chapter 2, MML incorporates existing notions in modeling, specifically class diagrams, OCL constraints and packages. The way packages are handled is slightly different to their treament in UML, bearing a much closer resemblance to the ideas presented in Catalysis [D'Souza and Wills 1998, D'Souza et al. 1999]. No precise definition exists which integrates all these notions, or combines them with a rich model of reflection. That is one of our tasks. However, the notions are understood enough to allow a first cut model to be produced using existing notations.

Thus our approach has been to begin by developing a series of class and package diagrams (using Rational Rose) and accompany these with a set of OCL constraints. Then to construct a tool based on our understanding of these paper artefacts. As the tool has taken longer to develop than expected, and because it does not (yet) generate diagrams from the models, it has been convenient to continue to maintain the Rose model, although, inevitably, this has led to some synchronisation issues. Our eventual goal is for there only to be one model (the one in and implemented by the tool), which will be aided by the ability to generate diagrams from the models, according to the syntax mappings that have been defined for the components of MML (see Chapter 5).

3.2. Subject areas

To illustrate how UML can be broken down into subject areas consider Figure 1 and Figure 2.

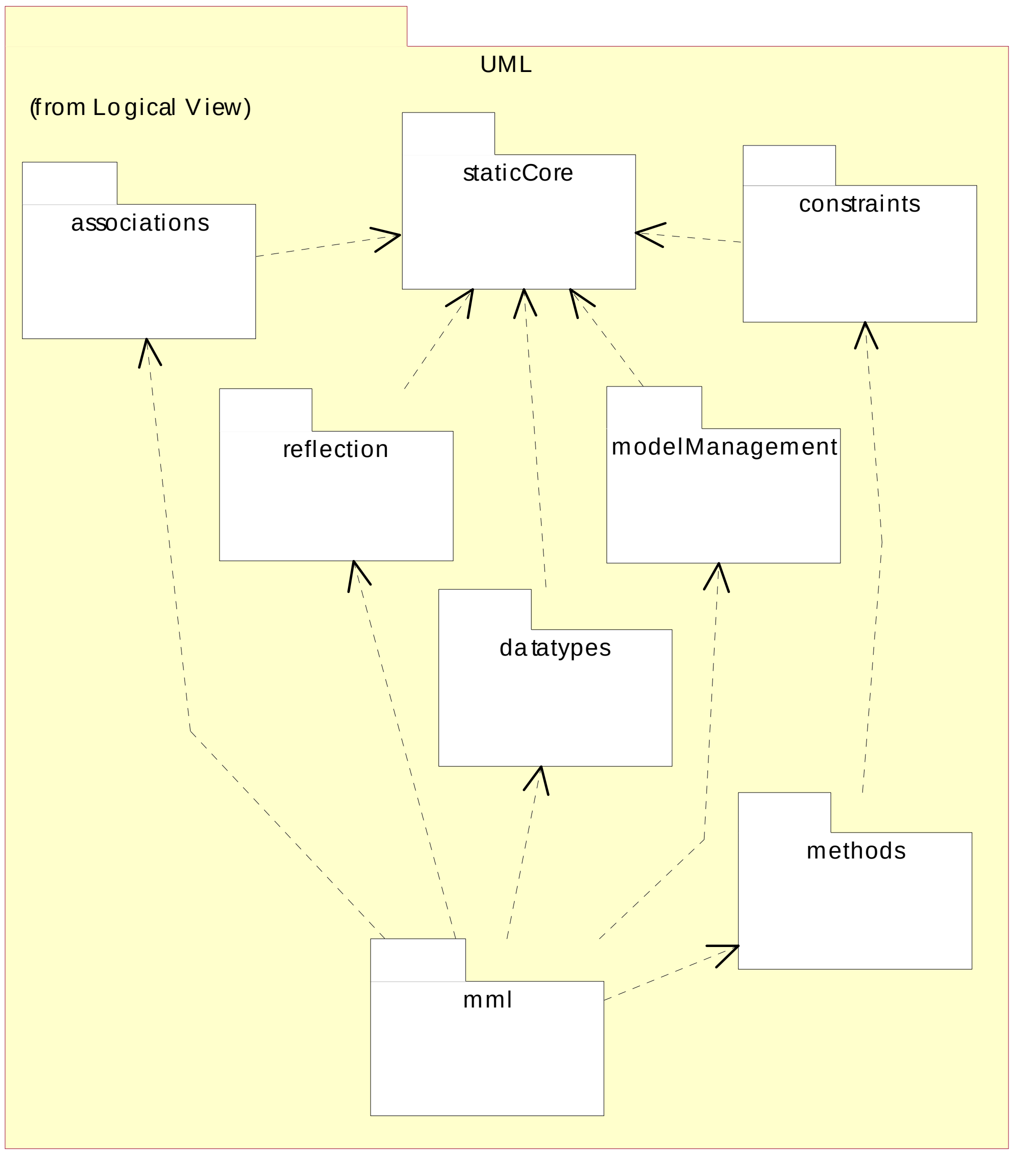

Figure 1. The components of MML

Figure 1 shows the various language components from which MML is constructed. Each of these is described in detail in Chapter 4. The arrows are intended to represent generalisation between packages. *`StaticCore`* generalises *`associations`*, *`reflection`* etc. Our preference would have been to use the normal UML inheritance arrow, but we have been constrained by what is possible in Rational Rose. Generalisation between packages is similar to generalisation between classes. If one thinks of a class as a container, in the sense that it contains attributes, methods and so on, then in a class generalisation relationship, the child must contain everything that the parent contains, but is allowed to specialise any of its components in a behaviourally conformant manner, and include new components. Similarly for package generalisation, except that packages contain classes and associations, instead of attributes and methods. We also allow renaming on generalisation (for packages and classes). Every model element has a name, and a model element can be specialised by changing its name. An example of renaming in use is given in Chapter 5.

All the language components are contained in the package UML. So, as well as containing classes and associations, packages can contain other packages.

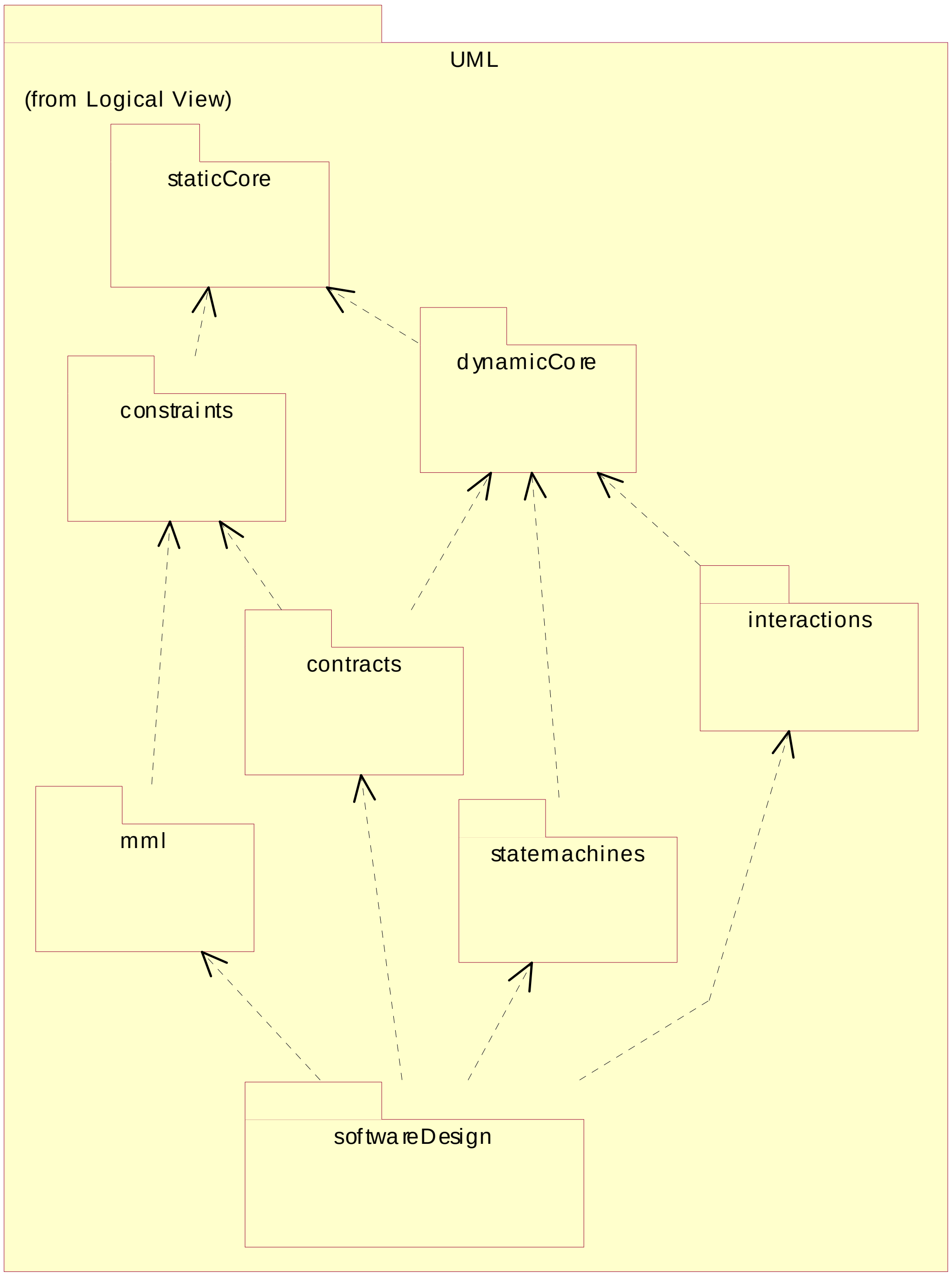

Figure 2. A language for software design

Figure 2 shows how another family member for UML could be constructed, this time a language for software design. One can see how new components to handle the dynamic aspects are introduced, which specialise and extend components used for MML. In addition, the language for software design must include all the features of MML itself, as these provide all the facilities for static modeling.

Of course there are likely to be many other family members: languages for modeling business processes, for modeling software design processes, which may be a specialisation of the language for modeling business processes, for modeling real-time systems; then languages for modeling in specific domains such as telecoms, health, automotive industry, and so on. We expect that all of these can be defined in a similar manner.

3.3. Language aspects

Each language component is further broken down into components according to the model/instance and syntax/concepts divides. For example, the package diagrams defining *`staticCore`* are given by Figure 3, Figure 4 and Figure 5.

Figure 3 illustrates how a language component is broken down into a model and instance aspect, where semantics is a mapping between the conceptual aspect of each.

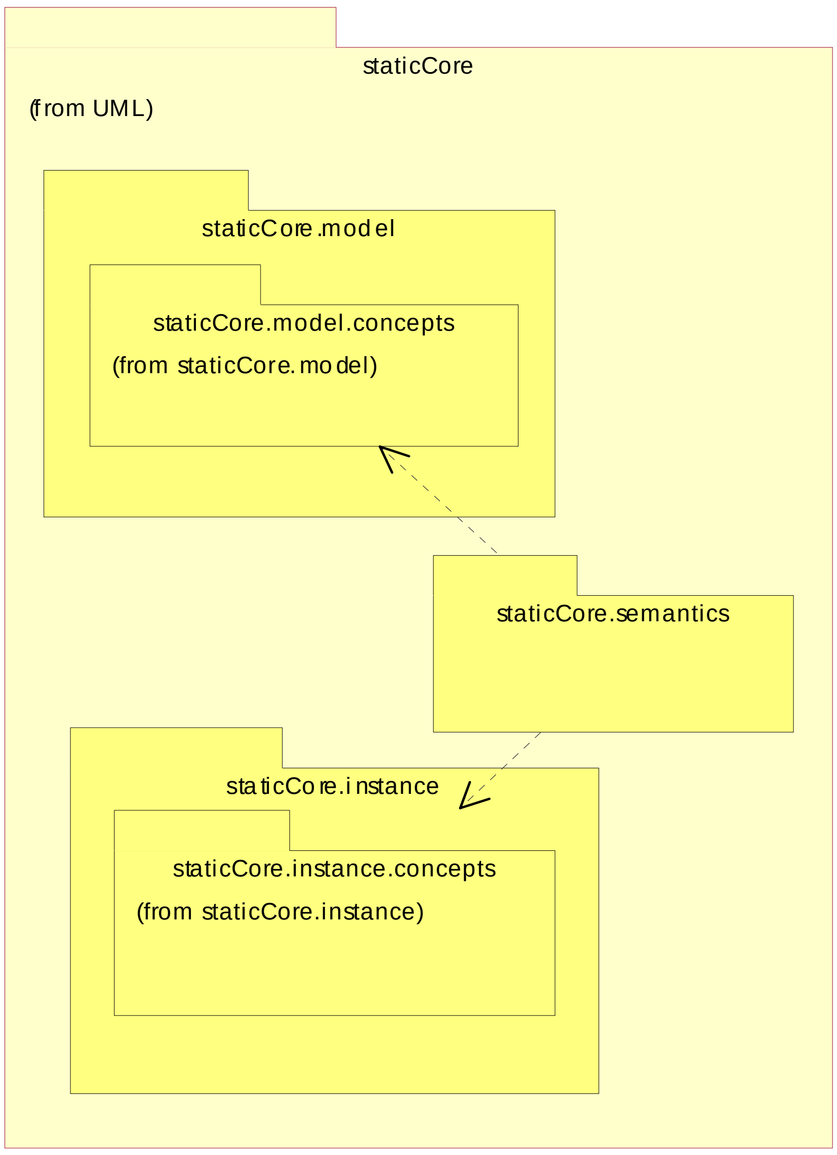

Figure 3. the components of staticCore

Semantics mappings are examined more closely in Chapter 4.

Figure 4 and Figure 5 shows that both model and instance aspects can be broken down into syntax and concepts aspects. This may seem odd for instances, until one considers that an object diagram is just a syntax for the concepts of object, link, etc.

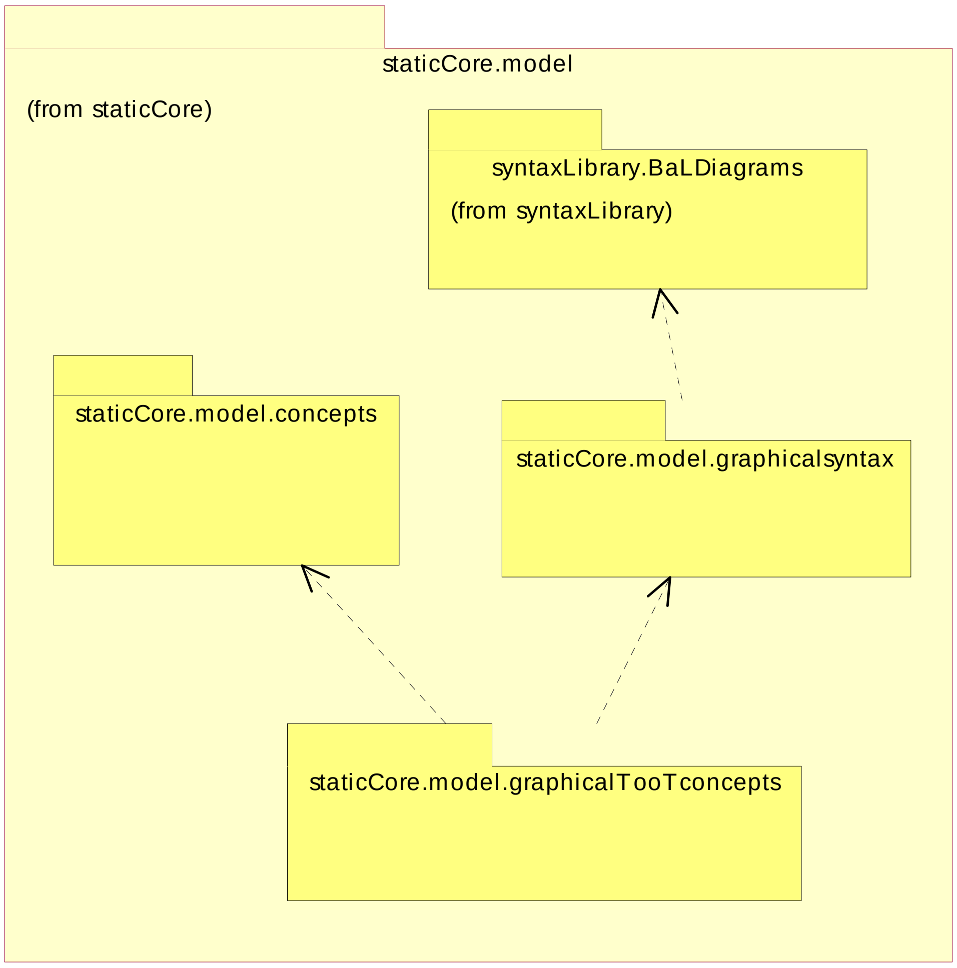
staticCore.model
(from staticCore)
syntaxLibrary.BaLDiagrams
(from syntaxLibrary)
staticCore.model.concepts
staticCore.model.graphicalsyntax
staticCore.model.graphicalTooTconcepts

Figure 4. Components of staticCore.model

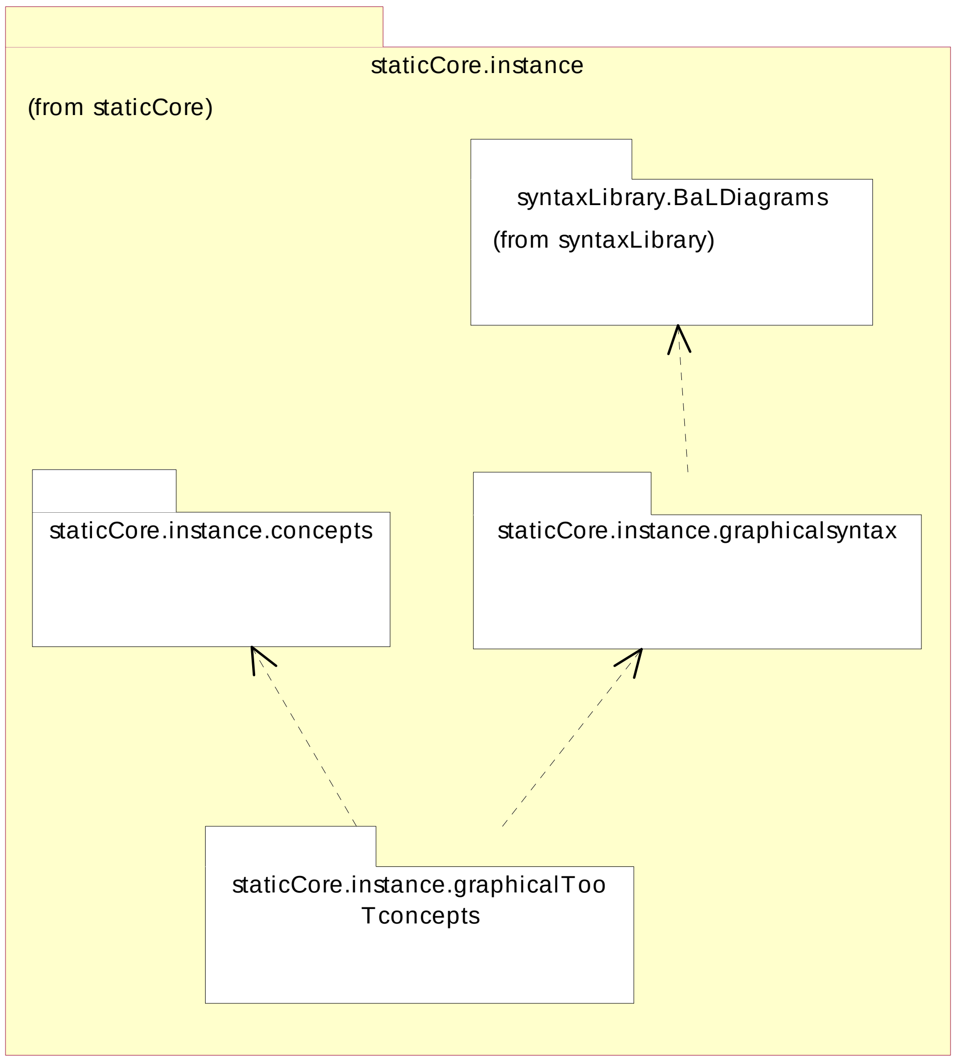
staticCore.instance
(from staticCore)
syntaxLibrary.BaLDiagrams
(from syntaxLibrary)
staticCore.instance.concepts
staticCore.instance.graphicalsyntax
staticCore.instance.graphicalToo Tconcepts

Figure 5. Components of staticCore.instance

The mapping between syntax and concepts is given by a syntaxTooTconcepts package. Where syntax my be a graphical syntax, xml etc. Syntax mappings are examined more closely in Chapter 5.

We haven’t shown the full contents of either staticCore.instance or staticCore.model, as there are likely to be other syntaxes that will be defined. Thus in Chapter 5, both graphical and xml syntaxes are defined for staticCore.model.

3.4. Extending a language component

So what happens to the sub-components when a language component is extended? The definition of generalisation tells us that by Figure 1, for example, *`modelManagement`* must contain the components of *`staticCore`*, that is it must also have model, instance and semantics packages, and, further, that these are still broken down into syntax (possibly more than one), concepts and the mappings between them. In the *`modelManagement`* package we are at liberty to further specialise the components inherited from *`staticCore`*. Numerous examples of language extension can be found in subsequent chapters.

3.5. Summary

This chapter has overviewed a collection of packages representing part of the UML family of languages. It has shown how UML can be constructed incrementally using package generalisation (specialisation) and how a clear separation between the different aspects of a language definition can be maintained. Subsequent chapters add detail to some of the packages presented here.

Further work in the are of language architecture would include identifying useful patterns of package structures (for example the model/instance, syntax/concepts divides) and embody these patterns, including the definition of intuitive syntaxes, in a revision of MML – the UML family member for language definition.

Chapter 4

The Meta-Modeling Language, MML

This chapter presents a precise meta-model description of the meta-modeling language, MML.

4.1. Introduction

The purpose of this chapter is to present a precise description of the semantics of the Meta-Modelling Language (MML). As outlined in the previous chapter, the description consists of a meta-model of the semantics of MML written using the MML itself.

The components of the MML are shown in Figure 6. In order to make the definition of the MML manageable and understandable, a layered, pluggable architecture has been adopted. Packages are used to separate out different concepts of the language. Package extension is then used to combine and extend upon these concepts in a logically consistent way, resulting in the complete MML definition.

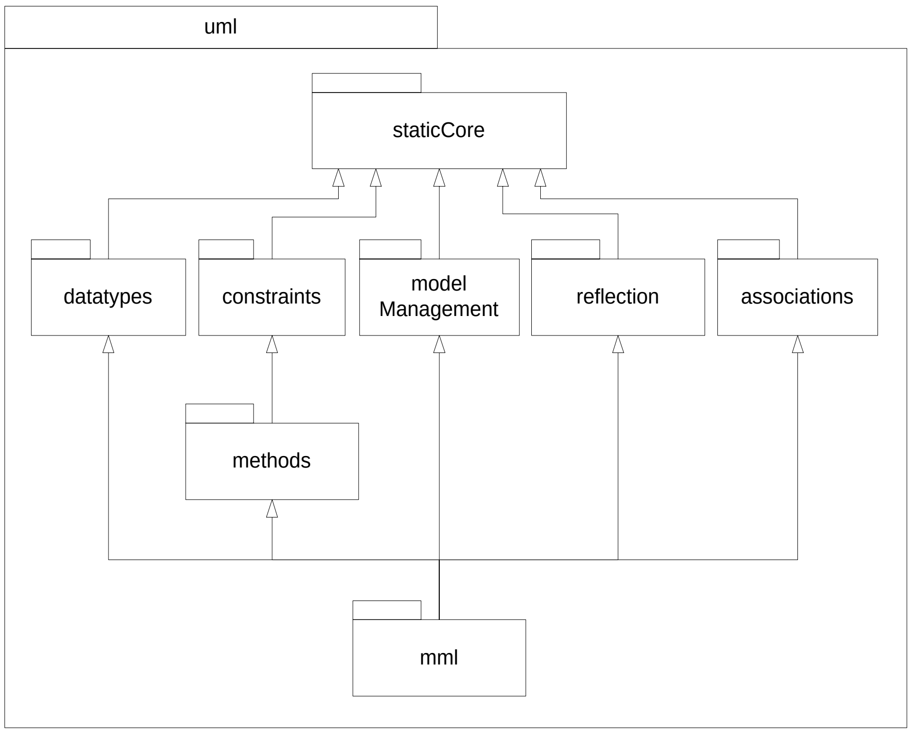

Figure 6. Components of the MML

A brief description of the purpose of each package is described below.

staticCore: fundamental object-oriented constructs for describing the static components of MML. It includes classifiers, classes, attributes, instances, objects and slots.

This package provides the basic semantic structures and concepts from which all other parts of the MML are extended.

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.

constraints: constructs relating to the expression of constraints. It defines a constraint language for MML that is similar to OCL, but which has a precise meta-model semantics.

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

model management: general mechanisms for managing collections of MML constructs. The primary mechanism provided is package extension (package specialisation).

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 allowing reflection in the MML.

associations: contains a definition of MML associations.

mml: merges all the components of MML into a single package.

As described in the previous chapter, each component of the MML is further divided into a strict pattern of syntax, concepts, instance and semantics packages. The concepts package describes the modelling concepts of the component, and the instance package describes the semantic domain of the modelling concepts. A mapping from each concept to its semantic domain is given in the semantics package, which extends both packages. Finally, the syntax package maps syntactical constructs (boxes, lines, etc.) to concepts.

The remainder of this chapter gives a detailed description of the contents of these packages, thus providing a definition of the semantics of MML. Details of the syntax packages are left until a later chapter.

4.2. staticCore

The static core package describes the fundamental static modelling constructs required to build MML models. It provides a general framework for extending MML with new model elements, including plug in-points for extension and useful meta-modelling patterns.

As shown in Figure 7, the staticCore package is decomposed into three sub-packages: model, instance and semantics. These describe the abstract syntax and well formedness rules of the core modelling concepts, the semantic domain and the relationship between abstract syntax and semantic domain concepts.

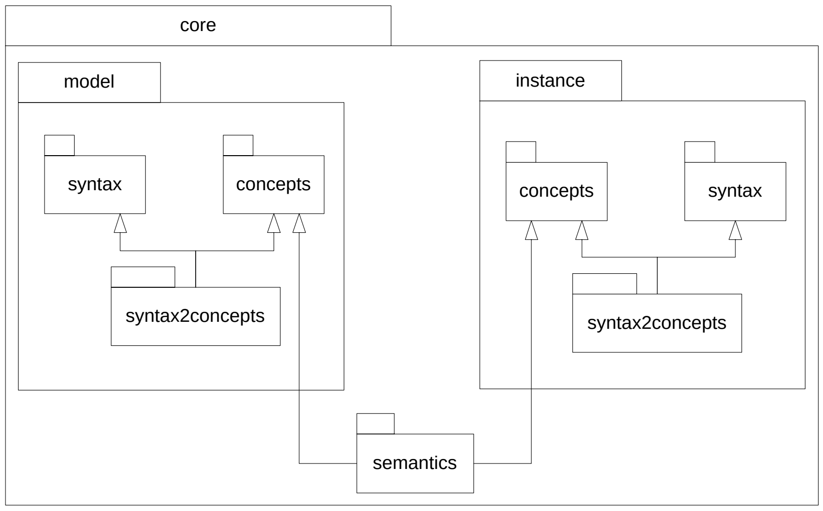

Figure 7. core package

4.2.1 core.model.concepts

The model concepts package of the core is shown in detail in Figure 8. The purpose of this package is to provide an abstract classification hierarchy within which the modelling elements of MML can be ‘plugged’. Briefly, it consists of the following classes:

ModelElement: All components of a model are model elements. Every modelling element has a name.

Container: Many modelling elements contain other modelling elements.This pattern is defined by the class container.

Generalisable: The generalisable class captures the notion of an extension relationship between model elements. A generalisation is a named link between a generalisable element and its parents/children.

Classifier: a generalisable, container of model elements, which captures the notion of a collection of instances. A classifier represents a fundamental pattern of features and properties which are shared by any model element in MML whose purpose is to classify properties of a model.

In addition, two important concrete (i.e. non-abstract) elements of the MML are defined in the staticCore. Classes are denoted as classifiers (they are named and are generalisable and instantiable). Classes also contain attributes. An attribute has a name, and a type, which is a classifier. Attributes are also classifiers, and are viewed as containing their types.

Note, that in the MML, attributes do not have multiplicities. Instead, their type indicates whether they are multi-valued or not (see section 4.3.1). This aims to provide a less restrictive and more coherent way of expressing attribute values, which is more appropriate for OCL.

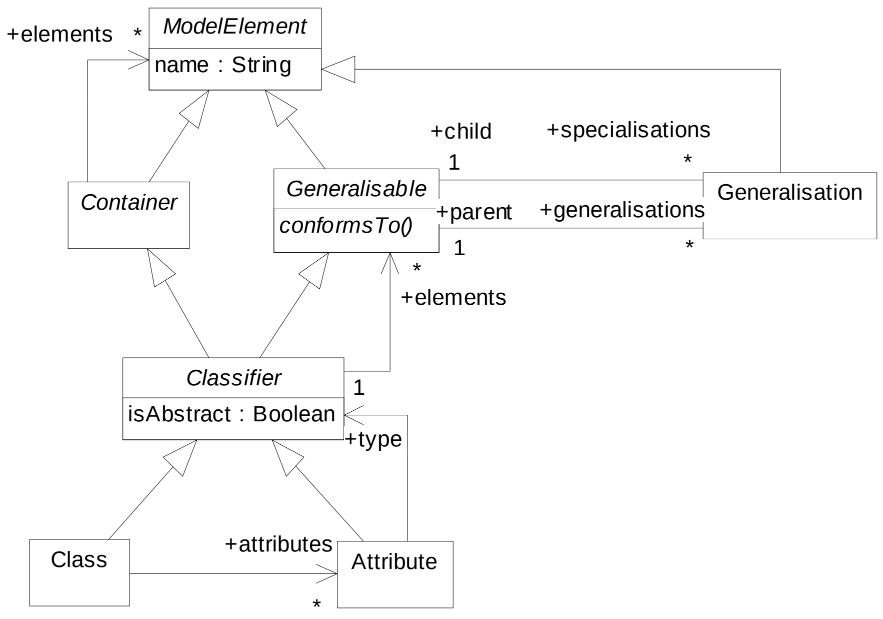

Figure 8: core.model.concepts Package

Well-formedness rules

The following well-formedness rules apply to the static abstract syntax package:

[1] No two elements belonging to a container can have the same name:

```
context uml.staticCore.model.concepts.Container inv:
  elements -> forall (e1 |
    elements -> forall (e2 |
      e1.name <> e2.name implies e1 <> e2))
```

[2] Circular inheritance is not allowed.

```
context uml.staticCore.model.concepts.Generalisable inv:
  not(self.allParents()->includes(self))
```

[3] The parents of a generalisable element must be of the same type.

```
context uml.staticCore.model.concepts.Generalisable inv:
  parents() -> forall(p | p.type() = self.type())
```

Note, the definition of the method type() is given in Section 4.7.4.

[4] A generalisable element must conform to its parents.

```
context uml.staticCore.model.concepts.Generalisable inv:
  parents() -> forall(g | self.conformsTo(g))
```

Generalisable must set up an extensible constraint that must hold between a parent and child. This is done via an abstract method 'conformsTo' that all generalisable elements must provide. This allows specific sub-classes of Generalisable to define their own conformance rules.

[5] The elements of a class contain its attributes.

```
context uml.staticCore.model.concepts.Class inv
  elements->exists(g |
    g.name = "attributes" and
    g.elements = allAttributes())
```

Note that because a class is a container, it is necessary to state an invariant which links each of its contained elements (in this case attributes) with those of the inherited association 'elements'. Furthermore, if it is desirable to add further contained elements, a classifier must be used as a intermediate container for elements of each type. Otherwise, it will not be possible to distinguish between different contained elements which share the same type. For example, a class will be further extended by a variety of features such as methods and constraints. Thus, we distinguish between these by associating a class's elements with named containers, each of which contains the appropriate elements.

[6] The elements of an attribute contain its type.

```
context uml.staticCore.model.concepts.Attribute inv:
  elements->size = 1 and
  elements->exists(g | g.name = "type" and g.elements = Set{type})
```

Methods

The method parents() returns a set containing all the direct parents of a generalisable element.

```
context uml.staticCore.model.concepts.Generalisable
  parents() : Set(Generalisable)
  self.generalisations -> iterate(g set = Set{} | g.parent)
```

The method allParents() returns a set containing all parents of a Classifier, excluding the Classifier itself.

```
context uml.staticCore.model.concepts.Generalisable
  allParents() : Set(Generalisable)
  self.parents()->union(parents()->iterate(parent set = Set{} |
    set->union(parent.allParents())))
```

The method allContents() is an abstract method that is inherited by all classifiers. It returns the set of all elements of the classifier, including those of its parents.

```
context uml.staticCore.model.concepts.Classifier
  allContents() : Set(Classifier)
  elements->union(parents()->iterate(parent contents = Set{} |
     contents->union(parents().allContents())))
```

The method allAttributes() returns the set of all attributes of Class, including those of its parents:

```
uml.staticCore.model.concepts.Class
  allAttributes() : Set(Attribute)
  parents->iterate(p s = attributes |
    s->union(p.allAttributes()->reject(c |
      attributes->exists(c' | c'.name = c.name)))
```

Duplicate attributes from multiply inherited classes are removed non-deterministically.

The methods conformsTo(c) defines what it means for a Classifier to conform to its parents:

```
uml.staticCore.model.concepts.Classifier
  conformsTo(g:Generalizable):Boolean
  if self = g then true
  else
    if g.isKindOf(Classifier)
      then
        if self.allParents()->includes(g)
          then g.elements->forAll(e1 |
            elements->exists(e2 |
              e1.name = e2.name and
              e2.conformsTo(e1)))
          else false
        endif
      else false
    endif
  endif
```

A classifier defines the essential conformance constraint via a definition of the inherited method 'conformsTo'. Two classifiers conform when the sub-classifier defines model elements with the same name as the super-classifier and when the corresponding elements conform. Note that it does not matter whether the elements under test are containers or atomic model elements, as it can be relied upon that the elements of a classifier are generalisable and therefore comparable.

4.2.2 core.instance.concepts

The instance.concepts package (see Figure 9) describes the semantic domain of the concepts in the core.model.concepts package. By semantic domain, it meant the minimal set of concepts that can be used to describe the meaning of the concepts in the model.concepts package.

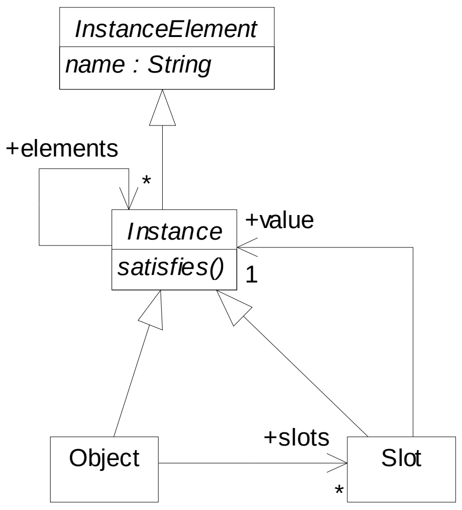

Figure 9: core.instance.concepts Package

An instance is a named instance element. An instance represents a fundamental pattern of features and properties which are shared by any instance in MML whose purpose is to classify properties of a model instances.In particular, it describes the fact that many instances contain other instances, and that instances satisfy the property of their classifiers.

Object and slot are fundamental concrete instances. An object contains slots, which have values.

Well-formedness rules

[1] The elements of an instance have unique names.

```
context uml.staticCore.instance.concepts.Object inv:
  elements -> forall(e1 |
    elements -> forall(e2 |
      e1 <> e2 implies e1.name <> e2.name))
```

[2] An object contains its slots.

```
context uml.staticCore.instance.concepts.Object inv:
  elements->exists(g |
    g.name = "attributes" and
    g.elements = slots)
```

[3] A slot contains is its value.

```
context uml.staticCore.model.concepts.Slot inv:
  elements->size = 1 and
  elements->exists(g | g.name = "type" and g.elements = Set{value})
```

4.2.3 core.semantics

The core.semantics package (Figure 10) imports all the constructs of Figure 8 and Figure 9. It associates classifiers with the valid set of instances that they may classify. Here, the association is further subclassed to restrict slots to being instances of attributes, and objects as instances of classes.

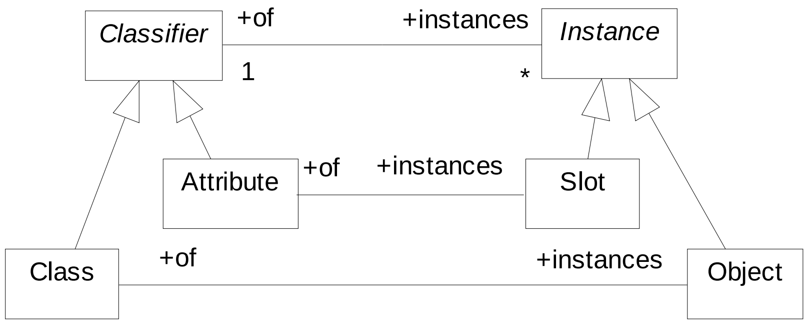

Figure 10: core.semantics Package

Well-formedness rules

The following OCL expressions precisely describe the constraints that relate model to instance concepts.

[1] An instance must satisfy the properties of its classifier.

```
context uml.staticCore.semantics.Instance inv:
  self.satisfies(self.of)
```

[2] Abstract classifiers cannot be instantiated.

```
context uml.staticCore.semantics.Classifier inv:
  isAbstract implies self.instances -> isEmpty
```

Methods

The method satisfies(c) defines what it means for an instance to satisfy the structural properties of a classifier.

```
context uml.staticCore.semantics.Instance inv:
  satisfies(c : Classifier) : Boolean
  if self.of = c then
    of.allContents() -> forall(e1 |
      self.elements -> exists(e2 |
        e1.name = e2.name and
        e2.satisfies(e1)))
  else false
  endif
```

An instance defines the essential satisfaction constraint via a definition of the inherited method 'satisfies'. An instance satisfies the properties of a classifier if, it is an instance of the classifier, and, for every element of the classifier, there exists an element of the instance such that the name of the instance element equals the name of the classifier element, and the instance element satisfies the classifier element.

The staticCore package is designed to ease the definition of new types of modelling element and their instances. In general, new types of element will be generalisable and will act as containers. Modelling elements may be defined as extensions of the existing classes, for example adding class features by extending Class with a new generalisable container. Alternatively, new types of modelling element may be defined, for example state transition machines or collaboration diagrams. Each new type of modelling element may take advantage of the standard patterns by extending Generalisable and/or Container. New sub-types of Generalisable should specialise the 'conformsTo' method so that it defines how to compare two instances of the new type. New sub-types of Container should define an invariant describing how 'elements' is structured. Finally, new sub-types of Instance should specialise the ‘satisfies’ method so that it defines how to compare instances against the model elements that they are an instance of.

It is possible to identify a number of steps that can be followed when adding new model elements to the MML. Briefly, these are as follows:

1. determine whether the model element is a subclass of classifier, i.e. exhibits the property of a generalisable, container;

2. if so, subclass the model element as a classifier in the model.concepts package;
3. constrain the model element's contents to be those of the attribute 'elements';
4. in the instance.concepts package, identify or add a new instance subclass which is an instance of the classifier;
5. in the semantics package, link them by subclassing the 'of/instances' association;
6. for each element of the new model element repeat steps 4-5;
7. determine any dependencies between instances and their elements, and specify these using appropriate constraints.

Note, in future versions of the MML it is intended to generalise the staticCore package further still. This will involve constructing packages of appropriate meta-modelling patterns (container/containable, instantiable/instance, and so on) and then defining concrete MML classes to be 'mix-ins' of the various patterns. As more experience is gained in extending MML with new model elements, an appropriate set of patterns will no doubt become identifiable.

4.3. datatypes

In this section, we extend the core model with data types. The extension is described by the datatypes package (see Figure 11). It extends the model, instance and semantics packages of the core. We follow the guidelines in the previous section to ensure that the extension satisfies the minimal semantic properties of model and instance concepts.

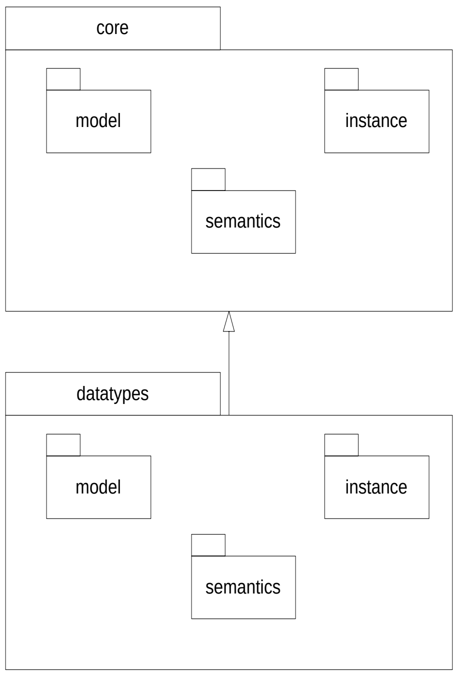

Figure 11: datatypes Package

4.3.1 datatypes.model.concepts

The contents of the datatypes.model.concepts package is shown in Figure 12. It introduces a new classifier, the data type. Datatypes include primitive built-in types (such as integer and string). In addition, a collection datatype is introduced to represent the basic built-in collection types of the MML (sets, sequences and bags).

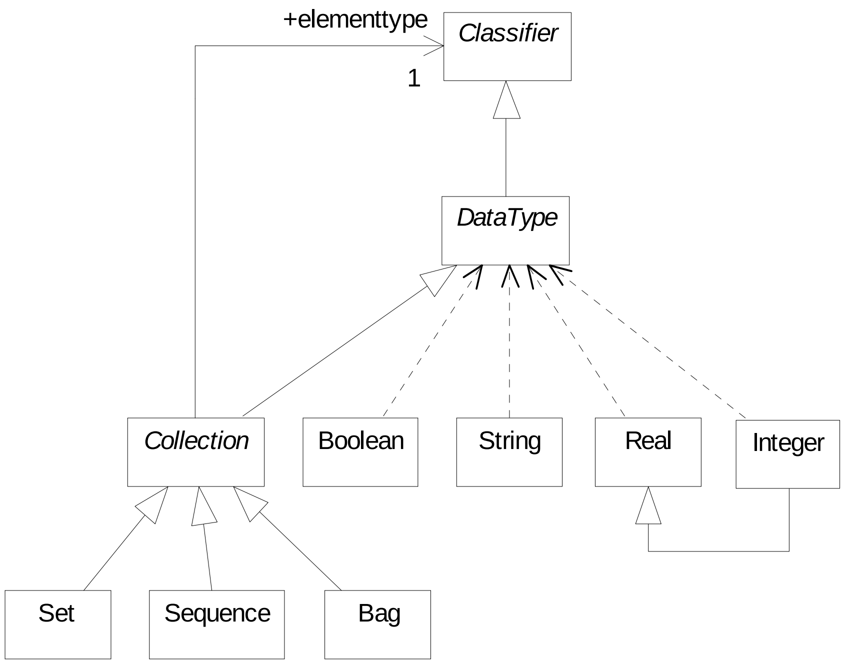

Figure 12: datatypes.model.concepts Package

There can only be one classifier representing a boolean, string, real or integer data type, and this is denoted by defining each data type as being an ‘instance’ of datatype (a definition of what it means for a class to be an instance of another class is given in section 4.7.4). Collections are parameterised by the type of classifier that they can contain. Finally, integers are subclasses of real datatypes, as integer values are also real values.

Well-formedness rules

[1] A collection contains its elementtype

```
context uml.datatypes.model.concepts.Collection inv:
  elements->size = 1 and
  elements->exists(g | g.name = "elementtype" and
    g.elements = Set{elementtype})
```

4.3.2 datatypes.instance.concepts

The Datatypes.instance.concepts package is shown in Figure 13. For each type in the datatype.model.concepts package, a corresponding datatype value is included in the package. For example, a boolean is represented by a boolean value, which may be true or false. Collection values contain elements, i.e. objects or datavalues. In the case of sequence values, an index is associated with each element in the sequence, while a count is associated with each element in a bag value.

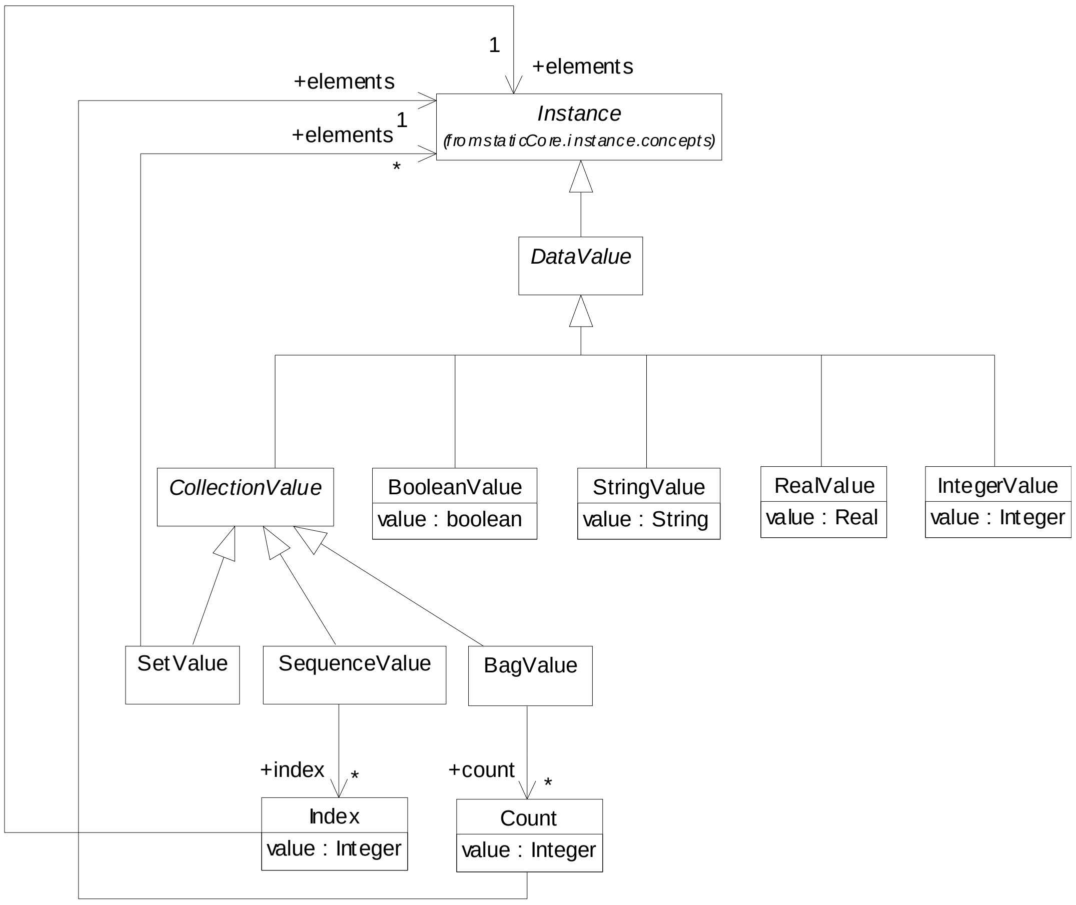

Figure 13: datatypes.instance.concepts Package

Well-formedness rules

[1] A sequence value with n elements, will have 1..n index values.

```
context uml.datatypes.model.instance.SequenceValue inv:
  index -> forall(i1,i2 |
    i1 <> i2 implies i1.value <> i2.value) and
    index -> iterate(x set = Set{} |
      set -> union(x.value)) = 1..(index -> size()))
```

[2] A set value contains are its instances.

```
context uml.datatypes.instance.concepts.SetValue inv:
  elements->exists(g | g.name = "elementtype" and
    g.elements = elements)
```

[3] A sequence value contains its instances.

```
context uml.datatypes.instance.concepts.SetValue inv:
  elements->exists(g | g.name = "elementtype" and
    g.elements = index -> iterate(e set = Set{} |
      set->union(e.elements)))
```

A similar definition is required for a bag value.

The datatypes.semantics package extends the datatypes.model.concepts and datatypes.instance.concepts packages. It associates datavalues with datatypes by sub-classing the instances association (see Figure 14).

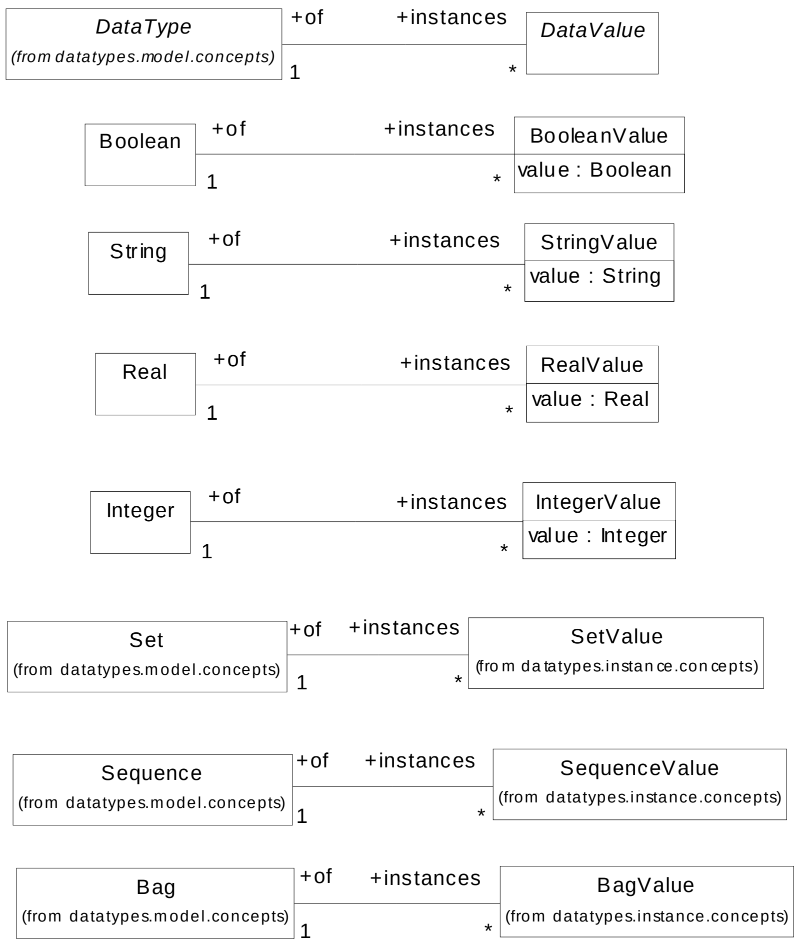
DataType
(from datatypes.model.concepts)
+of
+instances
1
*
DataValue
Boolean
BooleanValue
value : Boolean
String
StringValue
value : String
Real
RealValue
value : Real
Integer
IntegerValue
value : Integer
Set
(from datatypes.model.concepts)
SetValue
(from datatypes.instance.concepts)
Sequence
(from datatypes.model.concepts)
SequenceValue
(from datatypes.instance.concepts)
Bag
(from datatypes.model.concepts)
BagValue
(from datatypes.instance.concepts)

Figure 14. datatypes.semantics Package

Well-formedness Rules

[1] Collection values contain elements that satisfy their collection's element type.

```
    Inherited from Instance
```

Because the elements of both datatypes and data values have been defined above, the satisfies method will ensure that every element of a collection value will be an instance of the type of its collection.

4.4. constraints

In this section, the syntax and semantics of the constraint language provided by the MML are presented. The language is based on the Object Constraint Language (OCL) with the exception of one or two semantic changes that have been added in response to agreed changes at the Kent OCL workshop [uok 2000].

The constraints package extends the data types package in the normal way - each constraints.model.concepts, constraints.instance.concepts and constraints.semantics package extends the corresponding data types package.

4.4.1 constraints.model.concepts

The MML constraints.model.concepts package provides all the basic language expressions of OCL. These include logical expressions (and, not, equals, includes), slot references, variables and iterations. Note that all other constructs of the language, for example or, collect, set union, can be easily defined in terms of the basic expressions defined here. These will be added at a later date.

The abstract syntax of the language is described the constraints.model.concepts package shown in Figure 15.

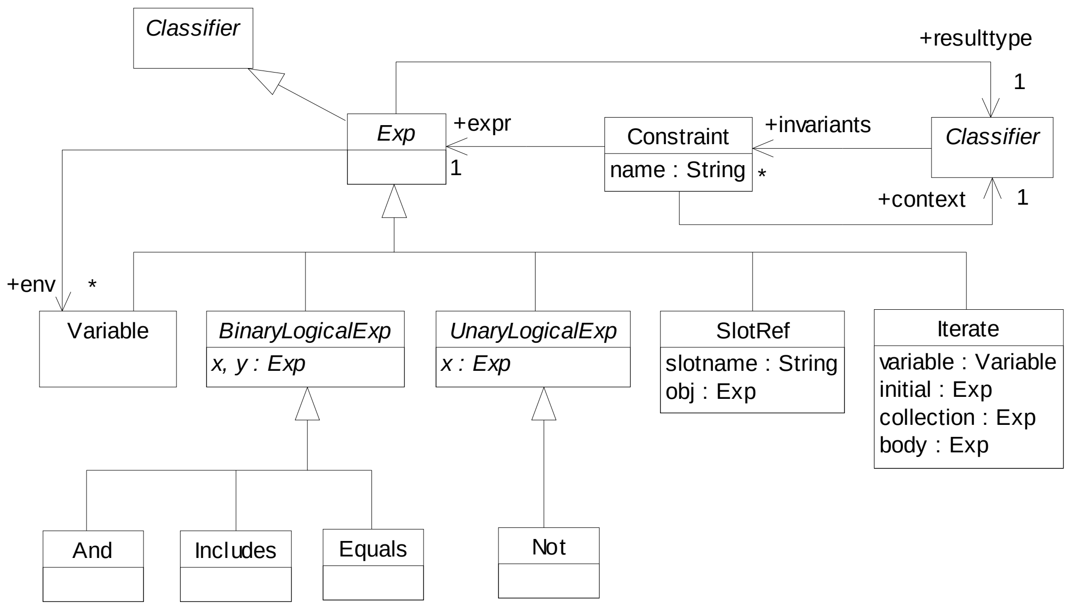

Figure 15: constraints.model.concepts Package

Associated with every classifier is an invariant. An invariant is a set of named constraints. Each constraint is expressed as an expression. An expression has a resulttype and is associated with a set of local variables - its environment.

Well-formedness Rules

[1] A classifier contains its constraints.

```
context uml.constraints.model.concepts.Classifier inv:
 elements -> exists(g |
   g.name = "invariant" and
   g.elements = allConstraints())
```

The following constraints set up the contained elements of expressions to include variables, sub-expressions, and expression type.

[2] An expression contains its variables and resulttype.

```
context uml.constraints.model.concepts.OCLExp inv:
 elements ->
  (exists(g | g.name = "variables" and
    g.elements = vars) and
  exists(g | g.name = "resulttype" and
    g.elements = resulttype))
```

[3] A binary expression contains the expressions, x and y.

```
context uml.constraints.model.concepts.BinaryExp inv:
elements ->
  (exists(g | g.name = "x" and
    g.elements = x) and
  exists(g | g.name = "y" and
    g.elements = y))
```

[4] A unary expression contains the expression, x.

```
context uml.constraints.model.concepts.UnaryExp inv:
elements ->
  exists(g | g.name = "x" and
    g.elements = x)
```

Similarly for SlotRef, Variable and Iterate expressions, etc.

The following constraints ensure that variables are propagated to sub-expressions.

[1] And propagation.

```
context uml.constraints.model.concepts.And inv:
  self.x.env = self.env and self.y.env = self.x
```

[2] Not propagation.

```
context uml.constraints.model.concepts.Not inv:
  self.x.env = self.env
```

[3] Includes propagation.

```
context uml.constraints.model.concepts.Includes inv:
  self.x.env = self.env and self.y.env = env
```

[4] Equals propagation.

```
context uml.constraints.model.concepts.Equals inv:
  self.x.env = self.env and self.y.env = env
```

[5] Slot reference propagation.

```
context uml.constraints.model.concepts.SlotRef inv:
  self.obj.env = self.env
```

Finally, constraints are required to ensure that each expression is associated with an appropriate resulttype.

[1] Binary and Unary expressions return a Boolean Value.

```
context uml.constraints.model.concepts.BinaryExp inv:
  self.resulttype -> isKindOf(Boolean)

context uml.constraints.model.concepts.UnaryExp inv:
  self.resulttype -> isKindOf(Boolean)
```

Similarly, for other expressions.

Methods

The method allConstraints() returns the set of Constraints belonging to a Classifier including those of its parents:

```
Classifier::allConstraints() : Set(Constraint)
 invariant->union(parents->iterate(parent set = Set{} |
   set->union(parent.allConstraints())))
```

4.4.2 constraints.instance.concepts

The semantic domain of the constraint language is described by the notion of a calculation. A calculation is an instance, which associates an expression and an environment (a set of variable bindings) with a value. The value is the result of evaluating the expression in the context of the bound variables. Figure 16 describes how the notion of a calculation is described in the constraints.instance.concepts package.

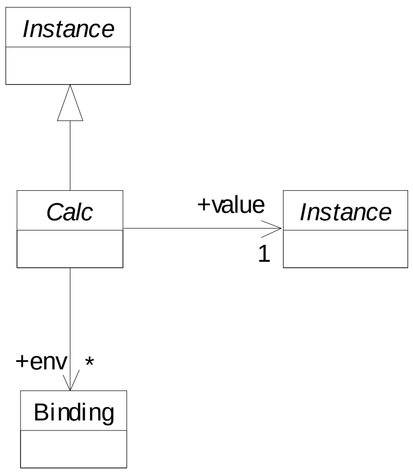

Figure 16: constraints.instance.concepts Package

Calculations exist for all expressions in the constraint language, as shown in Figure 17 (with the exception of iterate, which is described in section 4.4.4).

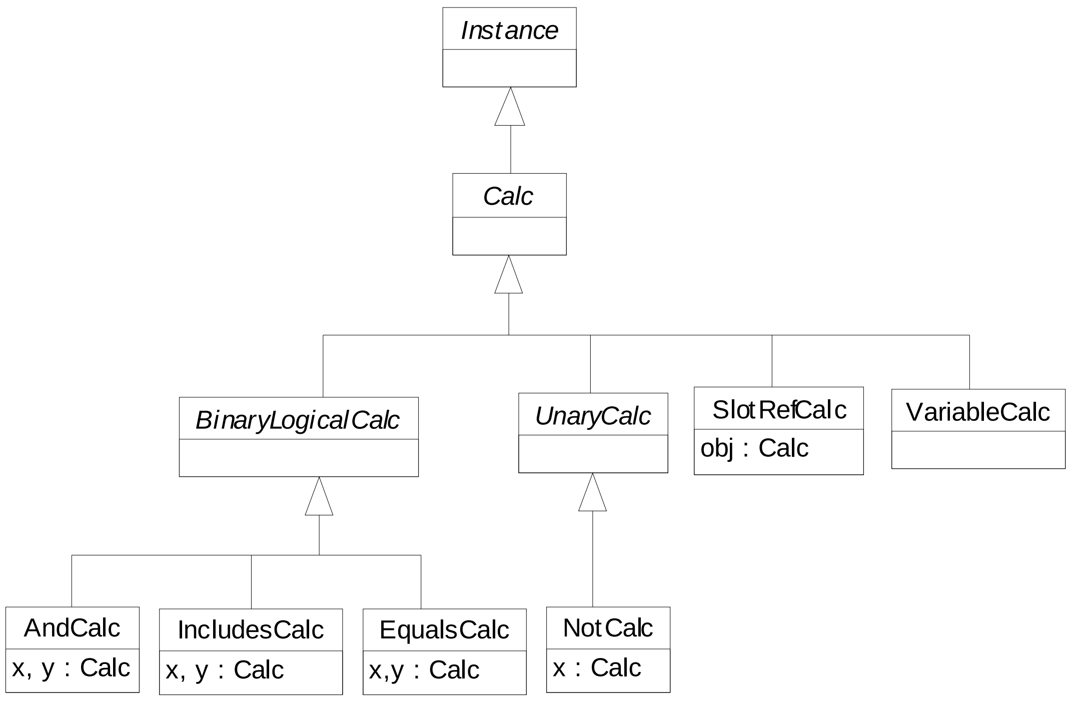

Figure 17: Calculations

Well-formedness Rules

The following constraints set up the contained elements of calculations to include variables, sub-expressions, and expression type.

[2] A calculation contains its variable bindings and result.

```
context uml.constraints.instance.concepts.Calc inv:
 elements ->
  (exists(g | g.name = "variables" and
    g.elements = env) and
  exists(g | g.name = "resulttype" and
    g.elements = Set{value}))
```

[3] A binary calculation contains the expressions, x and y.

```
context uml.constraints.instance.concepts.BinaryCalc inv:
elements ->
  (exists(g | g.name = "x" and
    g.elements = x) and
  exists(g | g.name = "y" and
    g.elements = y))
```

[4] A unary calculation contains the expression, x.

```
context uml.constraints.instance.concepts.UnaryCalc inv:
elements ->
  exists(g | g.name = "x" and
    g.elements = x)
```

Similarly for SlotRef, Variable and Iterate calculations, etc.

4.4.3 constraints.semantics

The constraints.semantics package associates each expression with appropriate calculations (see Figure 18)

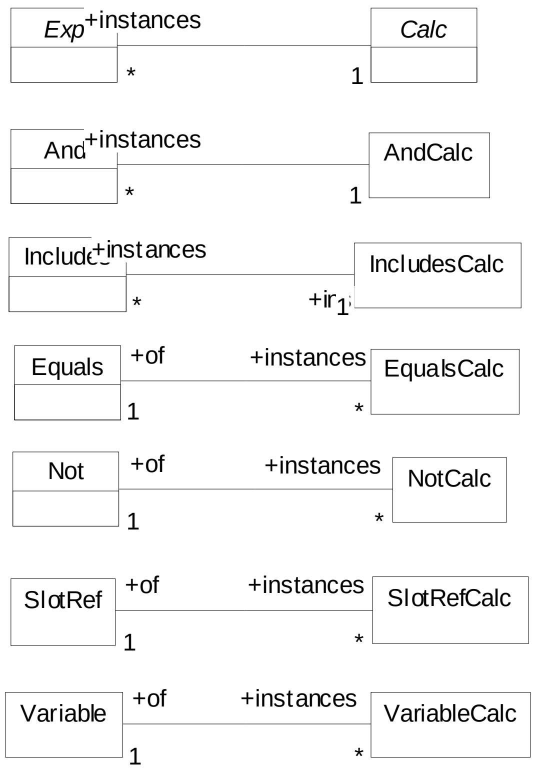

Figure 18. constraints.semantics Package

For each calculation, a constraint is required to describe its evaluation:

[1] The result of an and expression is the conjunction of the value of its x and y expressions.

```
context AndCalc inv:
  self.value = self.x.value and self.y.value
```

[2] The result of a not expression is the negation of the value of its x expression.

```
context NotCalc inv:
  self.value = not self.x.value
```

[3] The result of an includes expression is true if the value of its y expression is a member of its x expression.

```
context Includes inv:
  self.value = self.x.value ->includes(self.y.value)
```

[4] The result of an equals expression is true if the value of its y expression is equal to its x expression.

```
context Equals inv:
  self.value = self.x.value = self.y.value
```

[5] The result of a slot reference calculation is the value of the slot belonging to obj whose slotname is equal to the referenced slot name.

```
context SlotRefCalc inv:
  self.obj.value.slots -> exists(b |
     b.value = self.value and b.name = self.exp.slotname)
```

[6] The result of evaluating a variable expression is the variable’s binding.

```
context VariableCalc inv:
  self.env -> exists(e |
     e.value = self.value and e.variable.name = self.exp.name)
```

[7] Calculations contain sub-calculations that satisfy their expression’s sub-expressions.

```
Inherited from Instance.
```

This follows from the definition of instance satisfaction on contained elements.

4.4.4 .Iterate

The semantics of iterate are described here due to their relative complexity[1]. An iterate expression takes a collection, an initial accumulator value, and a body expression. It iterates through each element of the collection, evaluating the body expression, placing the result in the accumulator, and then iterating again. The iterate terminates when the collection is empty. The value of the accumulator is returned as the result of the iterate.

In order to express the semantics of iterate, a sub-iteration calculation is introduced in order to represent each iteration in the complete calculation. This extension to the constraints.semantics package is shown in Figure 19.

1. Note, this is a preliminary attempt, and has not been plugged into the container/generalisable pattern.

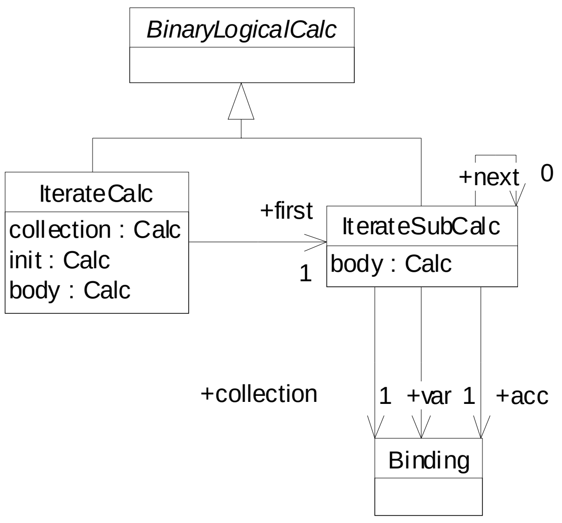

Figure 19. Iterate semantics

Well-formedness Rules

[1] The result of an iterate calculation is the value of the first iterate sub calculation.

```
context uml.constraints.instance.concepts.IterateCalc inv:
  value = self.first.value
```

[2] The first iterate sub calculation has the same body expression as the iterate calculation. Its collection is the value of the iterate calculations collection expression. Its accumulator value is the value of the iterate calculation's init expression. The name of its variable is the same as the expression's variable's name. The name of its accumulator is the same as its expression accumulator's name.

```
context uml.constraints.instance.concepts.IterateCalc inv:
  first.body = top and
  first.collection.value = self.collection.value and
  first.acc.value = init.value and
  first.var.name = exp.variable.name and
  first.acc.name = exp.acc.name
```

[3] An iterate sub calc's bindings contain its collection and acc bindings.

```
context uml.constraints.instance.concepts.IterateSubCalc inv:
  env -> includes(self.collection) and
  env -> includes(self.acc) and
  env -> includes(self.var)
```

[4] Provided that the collection is non-empty, the result of the sub calculation is the value of the next calculation, otherwise, its value is the current accumulator value.

```
context uml.constraints.instance.concepts.IterateSubCalc inv:
  not collection -> isEmpty implies value = next.value and
  collection -> isEmpty implies value = acc.value
```

[5] If the collection is non-empty, there exists a next calculation with the same body, which is applied to an element selected from the current collection and which results in

a new accumulator value calculated by substituting the current value of the accumulator and local variable into the body expression.

```
context uml.constraints.instance.concepts.IterateSubCalc inv:
  not collection -> isEmpty implies
    next.body = self.body and
    next.acc.value = self.body.value and
    collection -> exists(c |
      next.collection.value = self.collection.value - Set(c) and
      next.variable.value = c)
```

4.4.5 Object satisfaction

The following constraint describes the requirements of an object satisfying its constraints:

```
context uml.constraints.instance.concepts.Object inv:
  of.allConstraints() -> forall(i |
    i.expr.calcs -> exists(c |
       c.env -> exists(b |
         b.name = “self” and b.value = self) and
       c.value = true))
```

An object satisfies its class’s constraints, if for every constraint expression, there exists a calculation, whose value when the object is bound to the variable “self”, is true.

4.5. methods

The methods package provides a definition of the syntax and semantics of static methods. It extends the constraints package in the normal way (not shown here) - each model.concepts, instance.concepts and semantics package extends the corresponding constraints package.

A method takes an object, and a set of parameter values, and evaluates them against an OCL expression to obtain a result. Methods are a simplified version of OCL/UML 1.3 operations. Because MML is a static language, MML methods explicitly do not permit side-effects.

4.5.1 methods.model.concepts

The methods.model.concepts package (see Figure 20) associates a set of methods with a class. Methods are named, have a set of named and typed parameters and a body expression. Methods return a value, whose type is given by the method’s returntype.

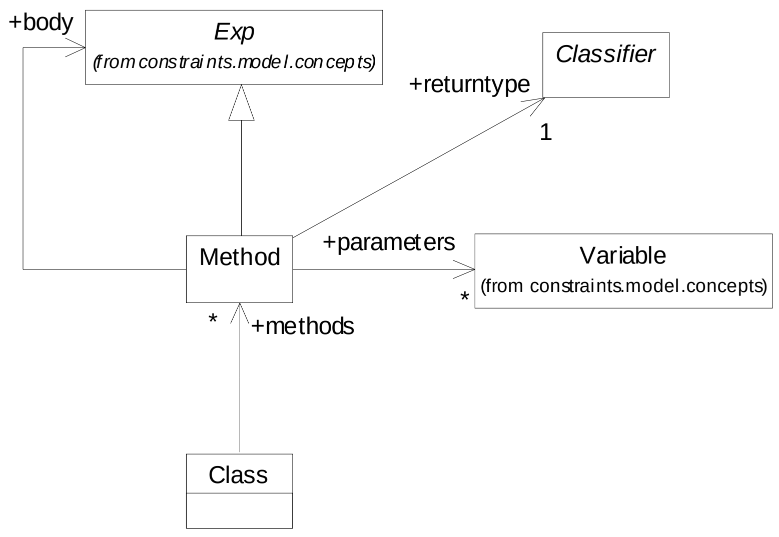

Figure 20. methods.model.concepts Package

Well-formedness rules

The following constraints set up the contained elements of a method expression to include parameter variables and a body sub-expressions.

[1]A method contains its returntype and parameters.

```
context uml.methods.model.concepts.Method inv:
 (elements -> exists(g |
   g.name = "returntype" and
   g.elements = Set{returntype})) and
 (elements -> exists(g |
   g.name = "parameters" and
   g.elements = parameters))
```

[2] A class contains all its methods, including those of its parents.

```
context uml.methods.model.concepts.Class inv:
 elements -> exists(g |
   g.name = "invariant" and
   g.elements = allMethods())
```

Methods

The method allMethods() returns the set of all methods of a class, including those of its parents:

```
uml.staticCore.model.concepts.Classifier
  allMethods() : Set(Methods)
  parents->iterate(p s = methods |
    s->union(p.allMethods()->reject(m |
    methods->exists(m' | m'.name = m.name)))
```

4.5.2 methods.instance.concepts

The meaning of a method is given by a method calculation (see Figure 21).

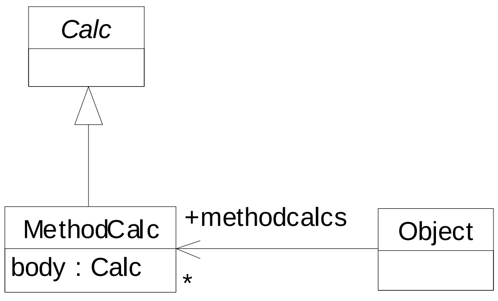

Figure 21. methods.instance.concepts Package

Well-formedness Rules

The following constraints set up the contained elements of a method calculation to include parameter bindings and a body sub-calculation.

```
context uml.methods.instance.concepts.Method inv:
 (elements -> exists(g |
   g.name = "returntype" and
   g.elements = Set{value})) and
 (elements -> exists(g |
   g.name = "parameters" and
   g.elements = env))
(elements -> exists(g |
   g.name = "body" and
   g.elements = body))
```

4.5.3 methods.semantics

The methods.semantics package (shown in Figure 22) associated method calculations with methods.

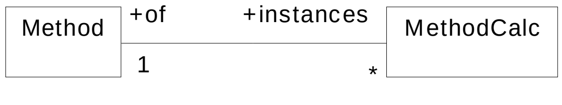

Figure 22. constraints.semantics Package

4.6. modelManagement

The model management package provides a definition of the syntax and semantics of packages. The model management package extends the staticCore package in the normal way (not shown here) - each model.concepts, instance.concepts and semantics package extends the corresponding staticCore package.

In the MML, packages provide a convenient means of grouping together, extending and sharing classifiers. Packages may extend other packages in that one package can be a specialisation of another package. In this case, classifiers in the generalised package are also available in the specialised package, and may be extended. Ownership of classifiers may also be shared through the use of imports. A package may import and classifiers from another package, in which case the classifiers are deemed to be owned by both packages.

The semantics of a package are described by its snapshots (collections of instances conforming to the classifiers owned by the package).

4.6.1 modelManagement.model.concepts

Figure 23 describes the abstract syntax of packages. A package is a classifier, and as such can be generalised and specialised by other packages. A package also contains classifiers.

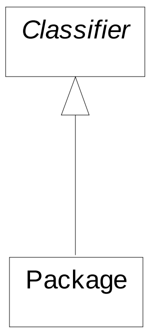

Figure 23: modelManagement.model.concepts Package

Well-formedness Rules

[1] The contents of a package must have unique names.

```
Inherited from Container.
```

[2] A child package extends all the contents of its parent packages in that each classifier in the child package conforms to a classifier with the same name in the parent package.

```
Inherited from Classifier.
```

4.6.2 modelManagement.instance.concepts

Just as classes and data values are denoted by their instances, the semantics domain of a package is the snapshot. A snapshot is a specialisation of an instance, as shown in Figure 24. A snapshot inherits the elements association from instance. The elements of a snapshot denote the instances that are contained in the snapshot.

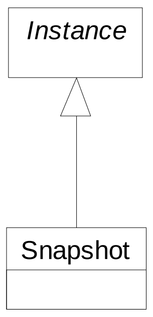

Figure 24. modelManagement.instance.concepts Package

Well-formedness Rules

There are no well-formedness rules.

4.6.3 modelManagement.semantics

The modelManagement semantics package adds the constraint that each instance belonging to a snapshot must be drawn from the contents of its package.

[2] All snapshot's contents are drawn from the contents of its package.

```
Inherited from Instance.
```

4.7. reflection

The reflection package provides a definition of reflection in the MML. In the MML, reflection denotes the ability to view MML model elements as instances. Thus it is possible to describe meta-level architectures in MML. For example, an MML model can be viewed as an instance of the MML meta-model, or as an instance of itself. This ability is also confered upon any extensions to the MML, thereby enabling a family of reflective languages to be developed.

Reflection is also a useful mechanism for representing types in MML, as, in general, the type of a model element will be defined by its meta-classifier.

The reflection package extends the staticCore package in the normal way (not shown here) - each model.concepts, instance.concepts and semantics reflection package extends the corresponding staticCore package.

4.7.1 reflection.model.concepts

No additional concepts.

4.7.2 reflection.instance.concepts

No additional concepts.

4.7.3 reflection.semantics

The reflection.semantics package describes the relationship between instances and model elements.

In the MML, all model elements are objects (see Figure 25). This has two important advantages:

- It provides a universal abstraction of all elements in the MML.
- It provides a simple mechanism for describing and making precise the reflective relationship between model elements and their meta-classifiers. By meta-classifier, it is meant the MML classifier which classifies the type of a model element. For example, a class Person can be viewed as an instance (or meta-instance) of the MML class Class. Class is therefore the meta-classifier of Person.

In general, all model elements are instances of the class Class. Constraints are required to ensure that their slots conform to those of Class, and also capture the essential features of the model element they represent. For example, a class object will contain slots for its name, attributes, etc.

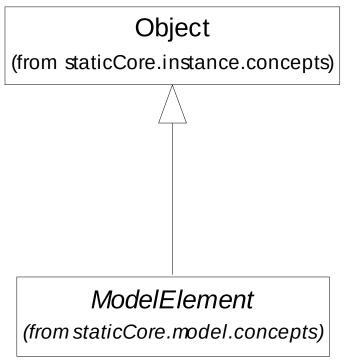
Object
(from staticCore.instance.concepts)
ModelElement
(from staticCore.model.concepts)

Figure 25. reflection.semantics Package

Well-formedness Rules

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

```
context uml.reflection.semantics.Class
  (slots -> exists(s |
    s.name = “name” and s.value = self.name)) and
  (slots -> exists(s |
    s.name = “attributes” and
      s.value -> isKindOf(SetValue) and
        allAttributes() -> forAll(a |
           s.value.elements -> exists(e : Object |
              e.slots -> exists(es |
                 es.name = “name” and
                 es.value = a.name) and
              e.slots -> exists(es |
                 es.name = “type” and
                 es.value = a.type)))) and
```

```
     (slots -> exists(s |
           s.name = “parents” and
           s.value -> isKindOf(SetValue) and
           self.parents -> forAll(p |
              s.value.elements -> exists(e : Object |
                 e = p)))) and
     (slots -> exists(s |
           s.name = “constraint” and
           s.value = self.constraint)) and
     (slots -> exists(s |
           s.name = “of” and
           s.value = self.of and
           s.of.name = “Class”)))
```

When viewed as an instance, each element of the class (name, attributes, parents and constraint) is represented as a slot of the object according to the following rules:

- name is represented by a slot “name” whose value is the class name.
- attributes are represented by a slot “attributes” whose value is a setvalue. Each element of the setvalue is an object with a slot “name” and slot “type” corresponding to each attributes’s name and type.
- parents are represented by a slot “parents” whose value is a setvalue. Each element of the setvalue is an object representation of each parent class.
- constraint is represented by a slot “constraint”, whose value is the object representation of its constraint.

[2] A package is an instance of the class Class.

```
   context uml.reflection.semantics.Package
      (slots -> exists(s |
         s.name = “packagename” and s.value = self.name)) and
      (slots -> exists(s |
         s.name = “contents” and
         s.value.elements = self.contents)) and
      (slots -> exists(s |
           s.name = “of” and
           s.value = self.of and
           s.of.name = “Class”)))
```

A package object contains a slot called “packagename” whose value is the name of the package, and a slot called “contents” whose values is a set containing all the contents of the package.

[3] A datatype is an instance of the class Class.

```
      context uml.reflection.semantics.Boolean
        slots -> exists(s |
          s.name = “name” and s.value = “Boolean”) and
        slots -> exists(s |
           s.name = “of” and
           s.value = self.of and
           s.of.name = “Class”))
```

Similarly for Integer, String, Real, etc.

[4] A set is an instance of the class Class.

```
context uml.reflection.semantics.Set
(slots -> exists(s |
  s.name = “name” and s.value = “Set”)) and
(slots -> exists(s |
  s.name = “elementtype” and s.value = elementtype)) and
slots -> exists(s |
      s.name = “of” and
      s.value = self.of and
      s.of.name = “Class”))
```

and similarly for Sequence and Bag.

Example

Figure 26 gives a small example of an object diagram that satisfies the above constraints for the example of a Dog class instantiating a (MML) class.

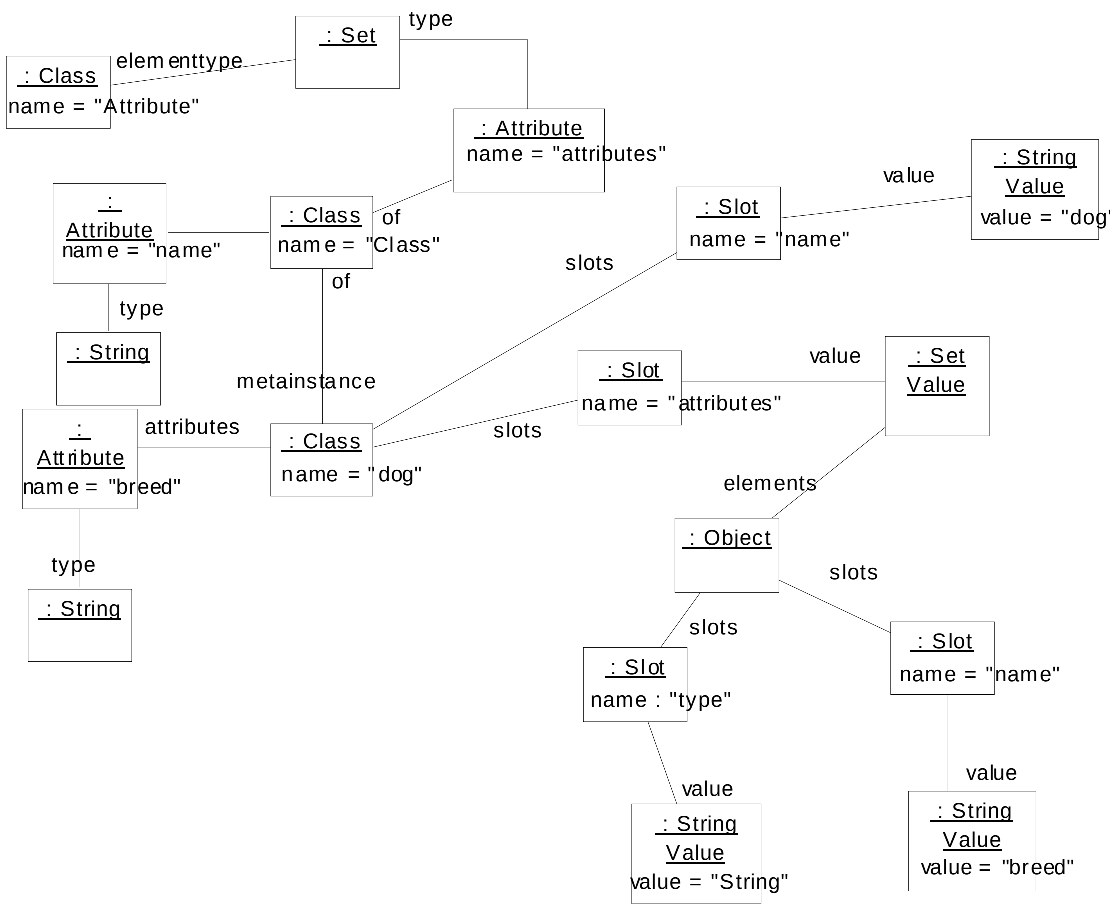

Figure 26. Meta-levels example

4.7.4 Type semantics

By enabling access to meta-classifiers, reflection provides a useful mechanism for determining the type of model elements. The type of a model element is its meta-classifier (for example, a class Person, will have the class Class as its meta-classifier). This property is encapsulated in the following method definitions:

Methods

The method isKindOf(c) returns true if a classifier is an instance of the meta-classifier, c.

```
reflection.semantics.Classifier
  isKindOf(c : Classifier) : Boolean
  self.of = c or
  self.allParents.of -> includes(c)ˆ
```

The method type() returns the meta-classifier that the Classifier is an instance of.

```
reflection.semantics.Classifier
  type(): Instance
  self.of
```

4.8. associations

The associations package provides definitions of binary associations, association classes, etc. These may either extend the concepts or syntax packages of MML, depending on whether they are to be viewed as fundamental concepts of MML, or convenient sugar.

4.8.1 binaryAssociation

Binary associations model bi-directional links between objects. In the MML, binary associations are modelled by pairs of attributes, with appropriate constraints to ensure that the slot values (i.e. links) between objects are of an appropriate multiplicity and are bi-directional. Thus, binary associations do not extend the semantic domain of the MML. Instead they represent a useful modelling constraint on MML models.

associations.model.concepts

Figure 27 shows the necessary extensions to the staticCore.model.concepts package required to model binary associations. A binary association has two association ends. An association end is just a attribute, extended with a multiplicity.

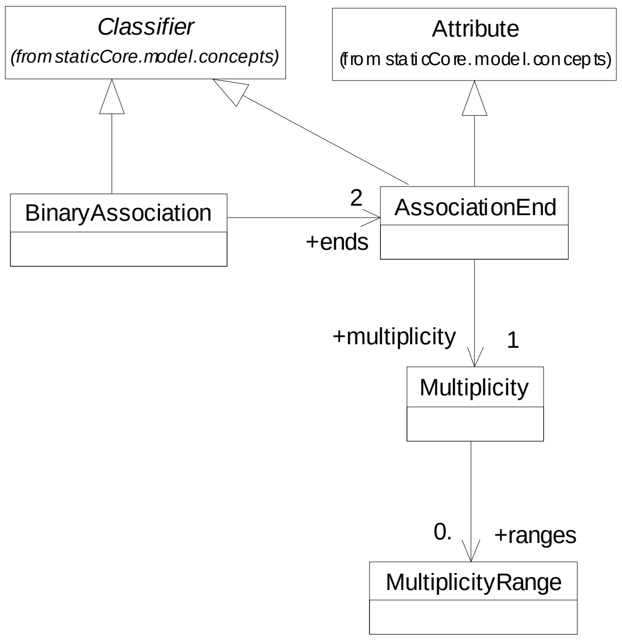

Figure 27. associations.model.concepts Package

Well-formedness Rules

[1] Attribute slots belonging to association ends always have a reverse slot.

```
context uml.association.model.concepts inv:
  end -> forall(e1, e2 | e1.name <> e2.name implies
        e1.instances -> forall(a1 | e2.instances ->
               exists(a2 | a2.value.slots -> includes(a1))))
```

[2] Slots conform to the multiplicity of association ends.

To be added.

4.8.2 association Class

To be defined.

4.9. Conclusions

This chapter has presented a definition of the semantics of the MML language written in itself. A denotational approach was used to assign each construct in the language a precise meaning. This was achieved by describing the relationships that must hold true between elements of the MML, and its instances.

In order to make the semantics more understandable and manageable, packages were used to structure the semantics into separate layers of language constructs. These were effectively combined using package extension to result in the complete definition. Various patterns of constraints have been used to ensure that a consistent approach is taken when defined common properties of subclass conformance and instance satisfaction.

As this is a feasibility study, further work will be required to develop a complete definition of the MML. In particular, the following areas will require further investigation:

- the semantics of the constraints package require further extension and checking to ensure that all the core elements of the constraint language have been defined. The definition of iterate needs checking.
- issues regarding monotonicity need to be investigated further. This relates to the notion of substitutability of classes and their instances. The issue of behavioural substitutability in particular will need further investigation.
- the semantics of MML and those of the tool supporting the language require further merging. Issues regarding the benefits of an operational semantics (currently supported by the tool) versus a declarative semantics (described in this chapter) need to be investigated further.
- a better mechanism is required for plugging classifiers into their containers. Currently, it is quite tedious to write an invariant linking the contents of a classifier with its contained elements. In addition, there is some duplication of the relationship between a classifier’s contents and its slots (when viewed as an object).
- an investigation of the relationship between the MML’s model.concepts package and that of the abstract syntax of MOF1.3. Harmonisation of these two may prove a fruitful route to integrating MML with future versions of MOF (i.e. MOF 2.0). Presently, the main difference between the MOF abstract syntax and MML is our representation of packages and attributes as classifiers. Given the growing acceptance that snapshots denote the instances of a package, and the natural classifier-instance relationship between attributes and slots, this seems a natural conclusion to make.

Chapter 5

Syntax Definitions

This chapter illustrates how MML can be used to define syntax, and the mapping of syntax to concepts. Two kinds of mapping are considered – to/from graphical syntax and to/from XML. We also show how a syntax mapping can be extended as new concepts are introduced. The chapter concludes with some reflection on how to progress this aspect of meta-modeling for UML.

5.1. Introduction

The UML 1.x standard defines the syntax of the various notations employed through examples and English explanation. No where does it provide an complete and unambiguous specification of the syntax, and, perhaps more importantly, of the mapping of that syntax to the underlying concepts, which, in the standard, are represented by the meta-model.

This chapter demonstrates how meta-modeling can be used to define syntax and the mapping of syntax to concepts. In particular it demonstrates:

- the importance of a constraint language, especially when it comes to writing down the mapping to concepts
- the importance of packages and package generalization, in providing a clean separation between syntax, concepts and the mapping between them
- the importance of packages and package generalization, in allowing language units to be constructed by specializing and extending other units

The chapter is divided into three main sections,

5.2. Syntax Packages

The package `syntaxLibrary` represents a set of packages that can be reused and specialised throughout UML for syntax mappings. We have developed three such packages for the purpose of this feasibility study, which are related according to Figure 28.

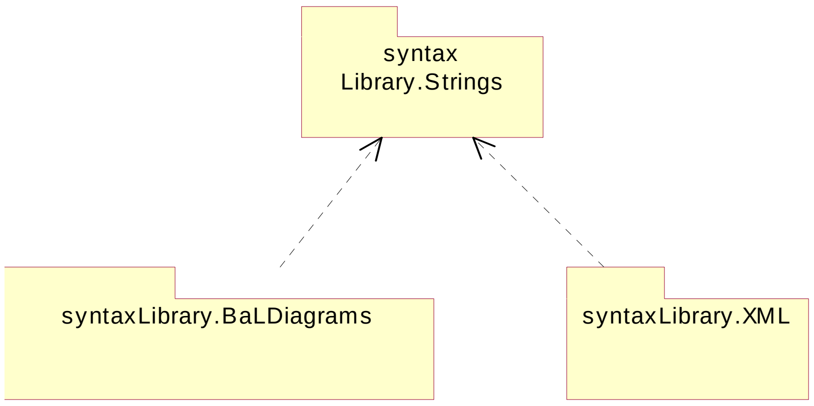

Figure 28. Syntax library packages

The arrows (which are the only arrows one can place between packages in Rational Rose), represent package generalization.

5.2.1 Strings

The contents of the strings package is represented by Figure 29. A piece of text is

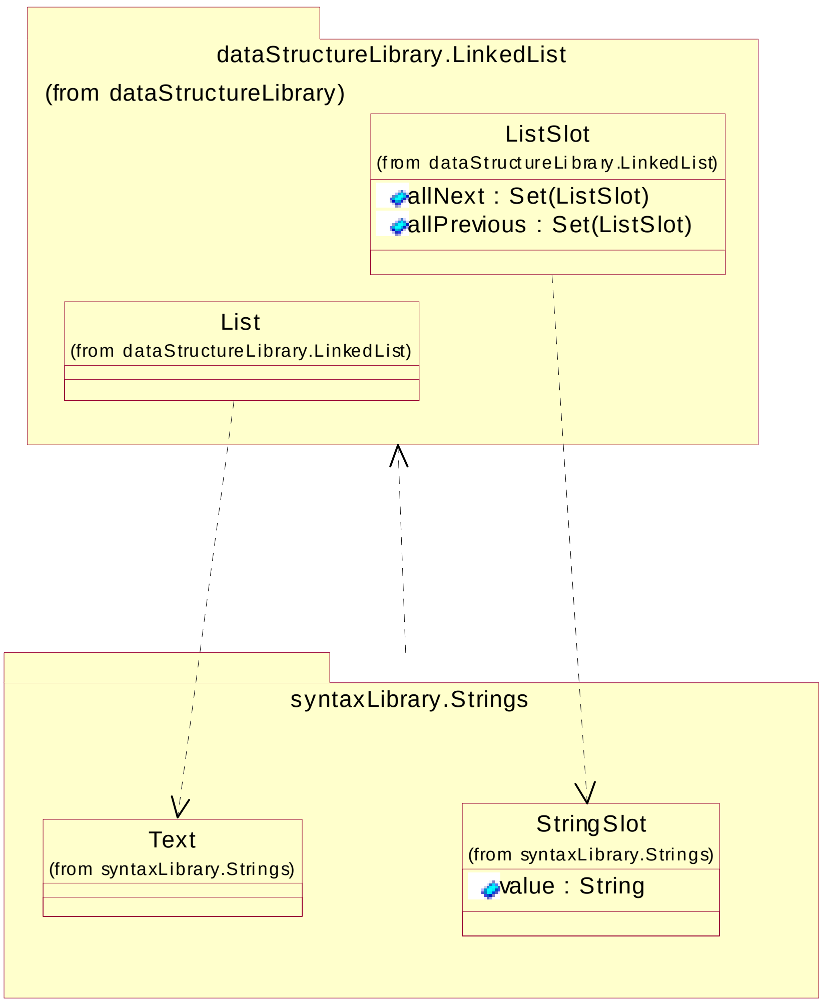
dataStructureLibrary.LinkedList
(from dataStructureLibrary)
ListSlot
(from dataStructureLibrary.LinkedList)
allNext : Set(ListSlot)
allPrevious : Set(ListSlot)
List
(from dataStructureLibrary.LinkedList)
syntaxLibrary.Strings
Text
(from syntaxLibrary.Strings)
StringSlot
(from syntaxLibrary.Strings)
value : String

Figure 29. syntaxLibrary.Strings package

basically a linked list of slots containing strings. This package is built up by inheriting from the package representing the linked list pattern. The arrows down the diagram represent renamings, the arrow up the generalization relationship. The class diagram for *`dataStructureLibrary.LinkedList`* is given in Figure 30.

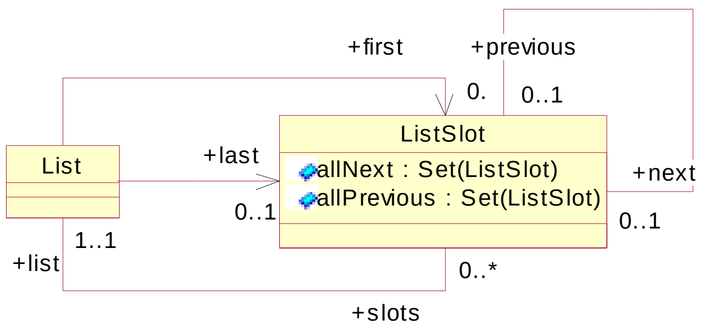

Figure 30. Package defining Linked Lists

`allNext` and *`allPrevious`* have been shown as attributes just to avoid line clutter on the diagram. This also needs to be supported by invariants ensuring that a linked list is a proper linked list, as follows:

Then a series of invariants ensuring that compartments essentially form a linked list:

```
context ListSlot inv:
    "define_allNext"
        self.allNext=self.next->union(self.next.allNext)

context ListSlot inv:
    "define_allPrevious"
        self.allPrevious=
          self.previous->union(self.previous.allPrevious)

context List inv:
    // last can be reached from first
    "last_reachable"
        self.first.allNext->includes(self.last)

context List inv:
    // first has no previous
    "first_is_first"
        self.first.previous->isEmpty

context List inv:
    // last has no next
    "last_is_last"
        self.last.next->isEmpty
```

5.2.2 Box and Line diagrams

In the UML architecture presented in chapter 2, the model of box and line diagrams is given by the package *`syntaxLibrary.BaLDiagrams`*. This model was constructed to support the demonstration of how mappings from concepts to diagrams may be specified in MML. There are many ways in which box and line diagrams could be modelled, and it is likely that a neater approach, where these kinds of diagrams were incorporated into a a much richer structure of packages supporting many different kinds of diagrams, could be adopted.

Figure 31 is a class diagram for the top level classes in the model. Vectors are intended to be more general than mere coordinates, containing an ability to be added to and producted with other vectors. Vectors have their own notion of equals, which circumvents the standard object identity notion – a vector equals another vector if their x and y values are the same, and methods (in the MML sense) for composing vectors. Finally, an association has been defined, allowing access to some standard *directional* vectors, which are stored in an object of a separate class, *`VectorDirections`*. In our model of diagrams, we have chosen a diagram element to have a position and a size, both of which can be represented by a vector (for size, think of the vertical and horizontal distances from the centre of a "bounding box").

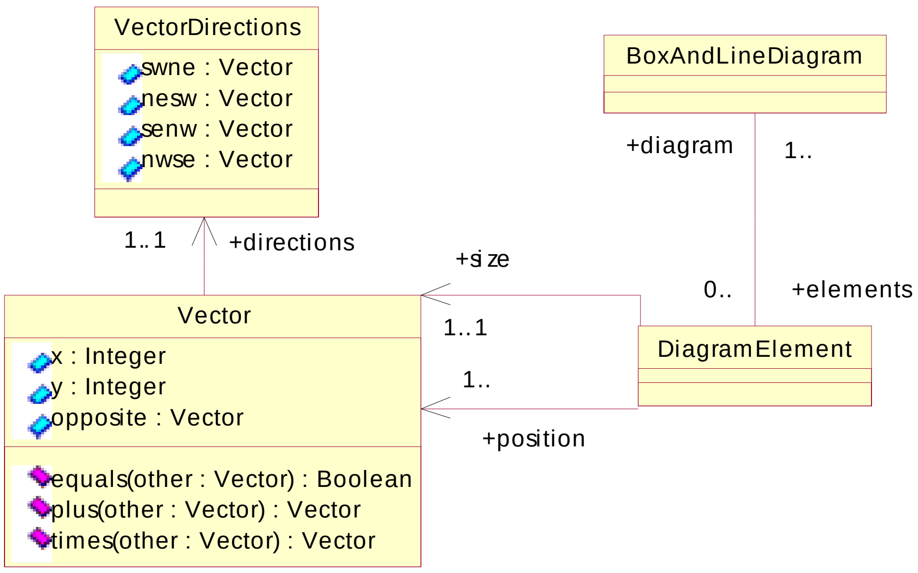

Figure 31. Top level classes for box and line diagrams

A whole series of invariants are required to define the meaning of plus, times, equals, opposite and the directions. First the invariants defining directions:

```
context VectorDirections inv:
    // directions have correct x and y values
    "define_directions"
    self.swne.x=-1 and self.swne.y=1 and
    self.nesw.opposite=self.swne and
    self.senw.x=1 and self.senw.y=1 and
    self.nwse.opposite=self.senw

context Vector inv:
    // stop infinite recursion in directions
    self.directions.nwse.directions=self.directions and
    self.directions.swne.directions=self.directions and
    self.directions.senw.directions=self.directions and
    self.directions.nesw.directions=self.directions
```

The second invariant avoids recursion through the direction vectors. It is being considered whether to have something like static/singleton classes/attributes in a future version of MML. This could be used to improve the model here - for example, the `VectorDirections` class could be a singleton.

Now the invariants defining the methods:

```
context Vector inv:
    "define_opposite"
    self.opposite.equals(self.times(self.directions.nwse))

context Vector inv:
    "define_times"
    self.times(v).x=self.x*v.x and self.times(v).y=self.y*v.y

context Vector inv:
    "define_plus"
    self.plus(v).x=self.x+v.x and self.plus(v).y=self.y+v.y
```

For box and line diagrams there are then two kinds of diagram element, which are, unsurprisingly, boxes and lines. These are defined by the class diagrams in Figure 32 and Figure 34, respectively.

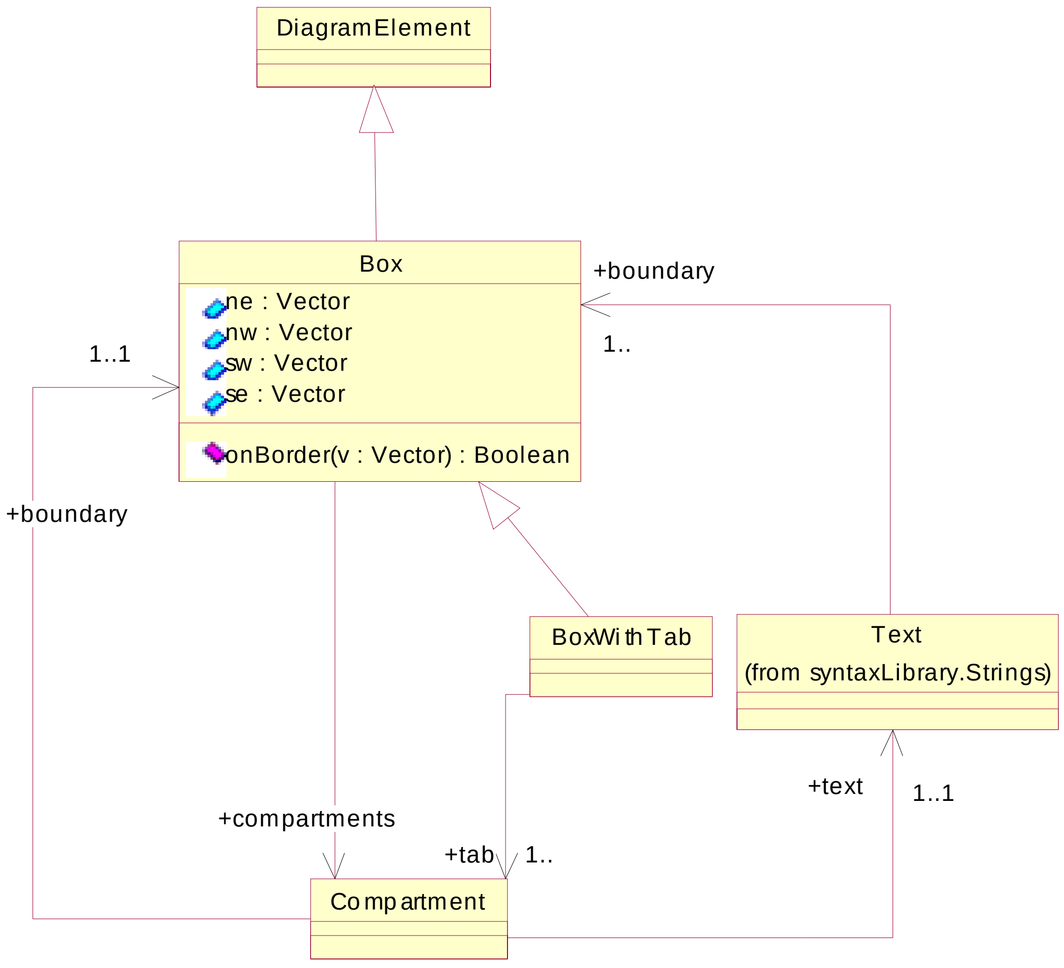

Figure 32. Boxes

Boxes have compartments, and a compartment contains some text. A box with a tab has, unsurprisingly, a tab associated with it (which is also a box). There is also a linked list relationship between Box (the linked list) and Compartment (the list slot). This is obtained simply by specializing the linked list package as in Figure 33.

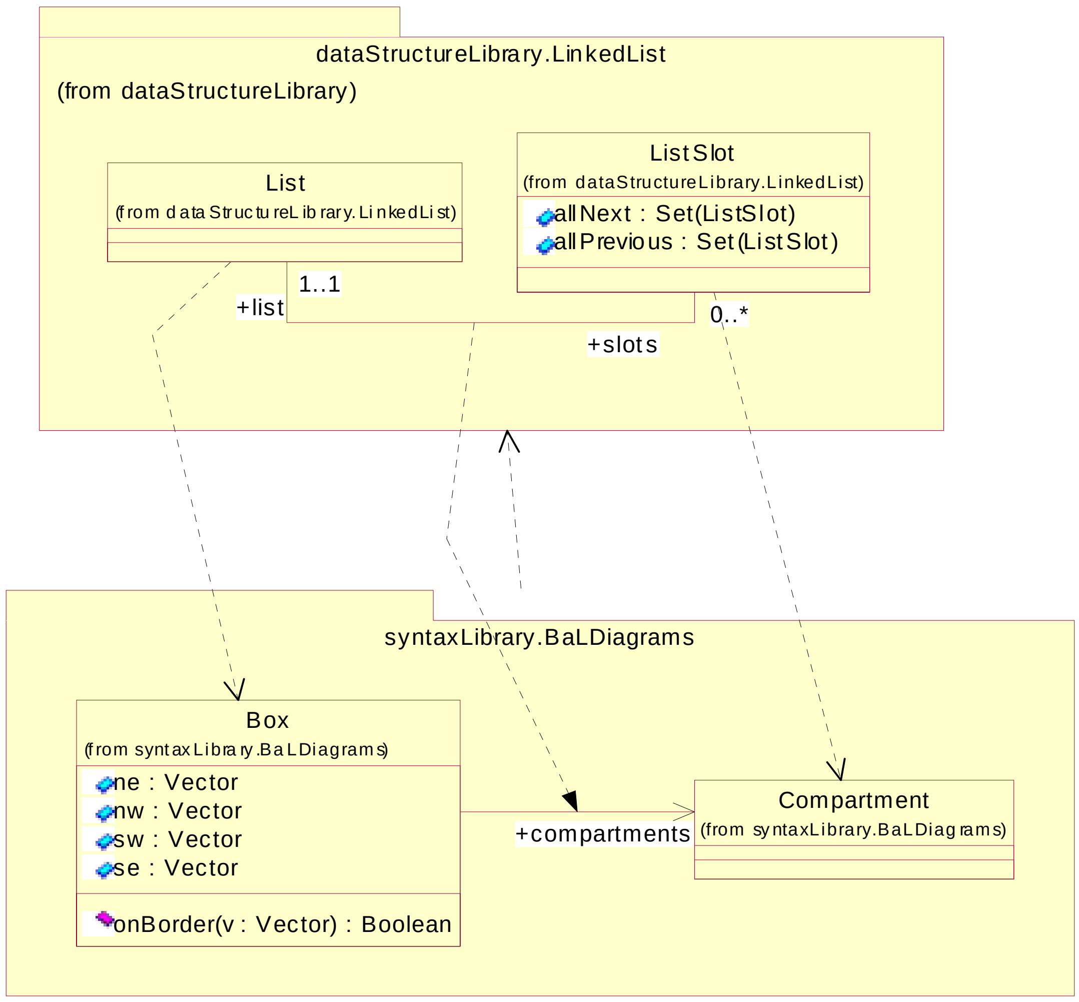
dataStructureLibrary.LinkedList
(from dataStructureLibrary)
List
(from dataStructureLibrary.LinkedList)
ListSlot
(from dataStructureLibrary.LinkedList)
allNext : Set(ListSlot)
allPrevious : Set(ListSlot)
1..1
+list
0..*
+slots
syntaxLibrary.BaLDiagrams
Box
(from syntaxLibrary.BaLDiagrams)
ne : Vector
nw : Vector
sw : Vector
se : Vector
onBorder(v : Vector) : Boolean
+compartments
Compartment
(from syntaxLibrary.BaLDiagrams)

Figure 33. Box/Compartment is List/Slot

Various invariants are required to support this diagram. First the invariants defining the positioning and size of the tab for a box with a tab.

```
context BoxWithTab inv:
    // the boundary of the tab must be at the ne corner of the box
    "tab_location"
        self.tab.boundary.se.equals(self.ne)

context BoxWithTab inv:
    // the tab must not be wider than the box
    "tab_width"
        self.tab.boundary.sw.x<=self.nw.x
```

Now the invariant ensuring that a compartment fully contains the text contained within it:

```
context Compartment inv:
    // a paragraph must appear within the boundary of
    // the compartment
    "text_box_within"
```

```
            self.boundary.contains(self.text.boundary)
```

Then an invariant ensuring compartments in a box are lined up correctly

```
    context Box inv:
        // the compartments of a box must be lined up correctly
        "compartments_placed_correctly"
        self.compartments->notEmpty implies (
            // first_compartment_touches_top
            self.firstCompartment.boundary.ne=self.ne and
             self.firstCompartment.boundary.nw=self.nw and
            // last_compartment_touches_bottom
            self.lastCompartment.boundary.se=self.se and
             self.lastCompartment.boundary.sw=self.sw and
            // compartments_touch
            self.compartments->forAll(c | c.next->isEmpty or
             (c.next.boundary.ne=c.boundary.se and
             c.next.boundary.nw=c.boundary.sw))
            )
```

And then an invariant defining the position of the corners on the box

```
    context Box inv:
        // corners are calculated appropriately from position & size
        "define_corners"
        self.ne=self.position.plus(
         self.size.times(self.size.directions.swne)) and
        self.nw=self.position.plus(
         self.size.times(self.size.directions.senw)) and
        self.se=self.position.plus(
         self.size.times(self.size.directions.nwse)) and
        self.sw=self.position.plus(
         self.size.times(self.size.directions.nesw))
```

Figure 34 deals with lines. A line end may have up to two labels and/or an arrow head,

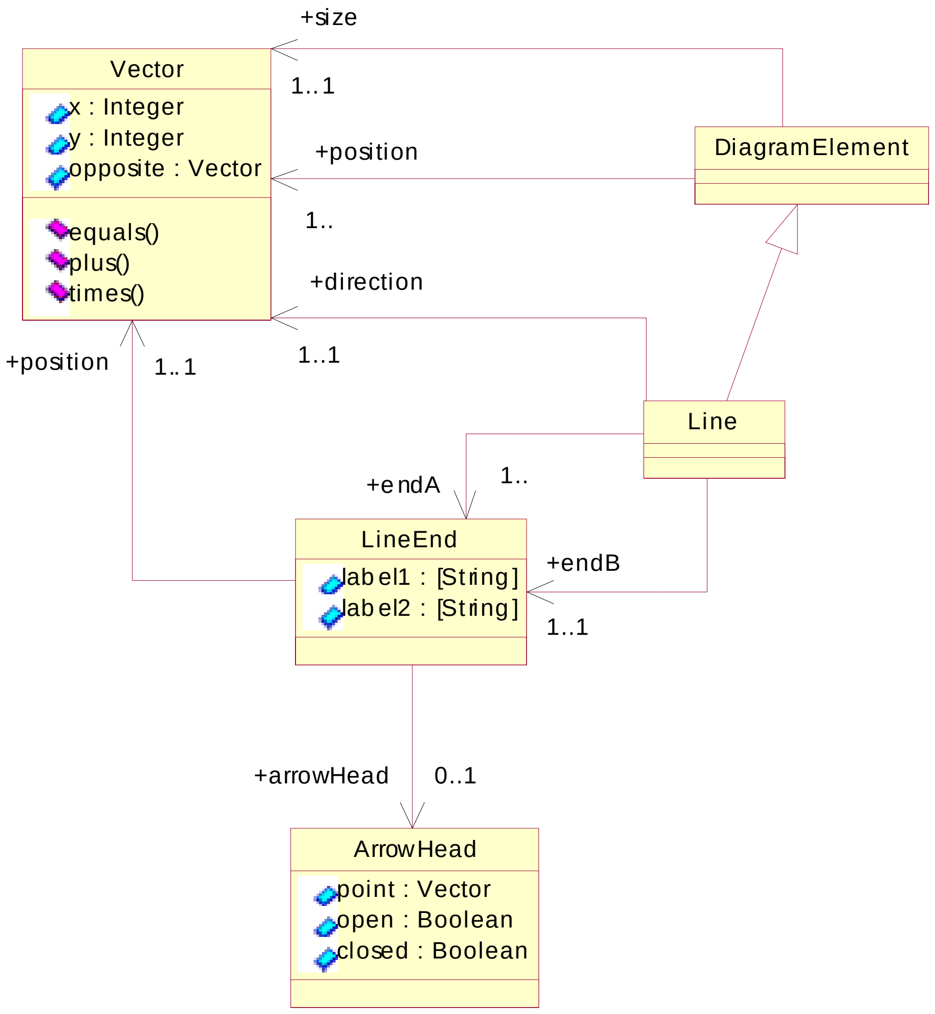

Figure 34. Lines

which will be closed or open. The position of the point of an arrow head is recorded as a vector. The position of the line ends is also recorded. A line has a direction which is also a vector. The direction, size and position of the (centre of the) line can be used to calculate the position of the end points. The diagram is supported by some self-explanatory invariants.

```
context Line inv:
    // direction of a line is one of the pre-determined
    // directions for Vector class
    "direction is a direction"
    self.direction.equals(self.direction.directions.swne) or
    self.direction.equals(self.direction.directions.nesw) or
    self.direction.equals(self.direction.directions.senw) or
    self.direction.equals(self.direction.directions.nwse)

context Line inv:
    // ends of the line are calculated appropriately from size
    // and direction
    "define_ends"
    self.endA.position=self.size.times(self.direction) and
```

```
        self.endB.position=self.size.times(self.direction.opposite)

    context LineEnd inv:
        // the point of the arrow head, if any, is at the position of
        // the line end
        self.arrowHead->isEmpty or
         self.arrowHead.point=self.position

    context ArrowHead inv:
        "open_is_not_closed"
        self.open = not self.closed
```

5.2.3 XML

The package characterising XML is quite simple, involving the single class diagram in Figure 35. This is a simplification of the domain object model for XML, as defined by

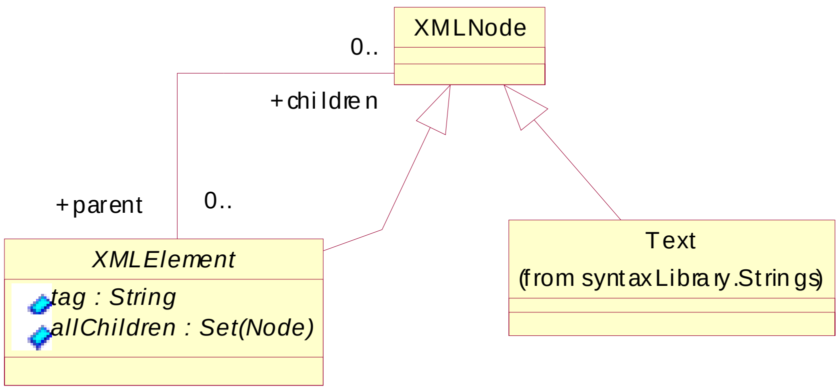

Figure 35. XML

the W3 consortium. It is sufficient to demonstrate the feasibility of specifying the mapping to XML. A string representation of XML could be easily generated from this model (and, indeed, the rules for generating it could also be specified in MML).

Under this model and XML document is an element, which has a tag and has child nodes, which may be other elements or text. The nodes form a tree structure, provided the following invariants are in place:

```
    context Element inv:
      “no_child_sharing_between_nodes”
      self.allChildren->select(c|oclIsKindOf(Element))
        ->forAll(e1,e2|e1<>e2 implies
        e1.children->intersection(e2.children))->isEmpty)

    context Element inv:
      “not_circular”
      self.allChildren->excludes(self)

    context Element inv:
      “define_allChildren”
      self.allChildren=self.children
        ->union(self.children.allChildren)
```

For example an html document would be represented as an `Element` object whose tag is “html”. This might have child elements which are either text or other elements. So it is likely to have a child element with tag “body”, which may in turn have a child element with tag “h1”, which may in turn have a child node that is a text object with a single string slot whose value is “Introduction”, and so on.

5.3. Graphical Syntax Mappings

5.3.1 Mapping of staticCore.model.concepts to box and line diagrams

The mapping of staticCore.model.concepts to box and line diagrams is in the package staticCore.model.graphicalTooTconcepts. TooT is intended to indicate a mapping that goes both ways. This package needs both sides of the mapping, giving rise to the package diagram in Figure 36.

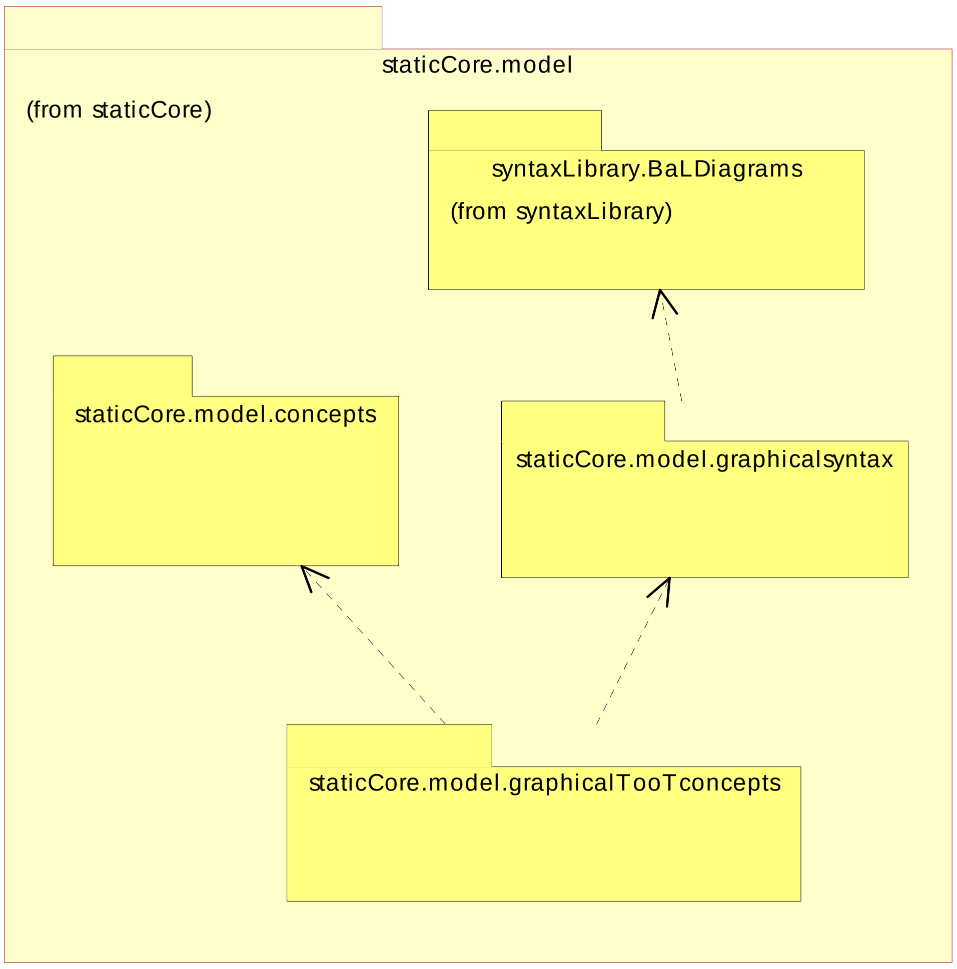
staticCore.model
(from staticCore)
syntaxLibrary.BaLDiagrams
(from syntaxLibrary)
staticCore.model.concepts
staticCore.model.graphicalsyntax
staticCore.model.graphicalTooTconcepts

Figure 36. Graphical syntax for staticCore.model

The details of the mapping (i.e. `staticCore.model.graphicalTooTconcepts`) are given by Figure 37, Figure 38, Figure 39 and accompanying invariants.

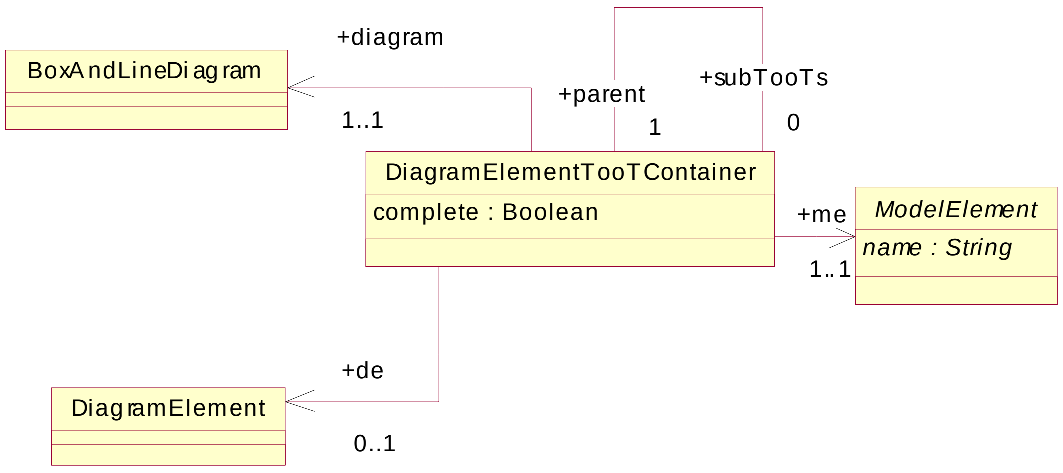

Figure 37. Graphical syntax for staticCore.model.concepts

The basic idea is that each concept element has a mapping class (a TooT class) associated with it, that indicates the diagram element or text that is mapped to that element. TooTs may involve sub-mappings: for example, mapping a class also requires its attributes to be mapped.

The first set of invariants relate to Figure 37 and deal with the general mapping of diagram elements to containers. They ensure that behaviour with respect to submappings (subTooTs) is as expected.

The first invariant ensures that the diagram associated with subTooTs is the same as that associated with the TooT.

```
context DiagramElementTooTContainer inv:
    "pass_diagram_to_element_TooTs"
    self.subTooTs->notEmpty implies
     self.subTooTs.diagram=self.diagram
```

The second invariant ensures that the diagram element being mapped to/from is on the diagram associated with the TooT.

```
context DiagramElementTooTContainer inv:
    "diagram_element_on_diagram"
    self.diagram.elements->includes(self.de)
```

The third and fourth invariants are self-explanatory.

```
context DiagramElementTooTContainer inv:
    "contained_model_elements_not_mapped_twice"
    self.subTooTs->size=self.subTooTs.me->size
    // relies on bags

context DiagramElementTooTContainer inv:
    "only_contained_model_elements_may_be_mapped"
    self.me.oclIsKindOf(Container) implies
     self.me.elements->includesAll(self.subTooTs.me)
```

Whereas, the fourth invariant only ensures that if there are submappings, then these only map elements that are contained in the model element being mapped, the final invariant introduces the concept of a complete mapping, where the submappings are exactly mappings to/from elements contained in the model element being mapped. For example, a complete mapping for a class must map all attributes in the class, whereas an incomplete one may leave some attributes hidden on the resulting diagram.

```
context DiagramElementTooTContainer inv:
    "all_contained_model_elements_mapped_if_complete"
    self.complete =
     (self.me.oclIsKindOf(Container) implies
       self.me.elements=self.toots.me)
```

The second set of invariants relate to Figure 38, and deal with the drawing of generalisation arrows. The first invariant ensures that these arrows are drawn between anything

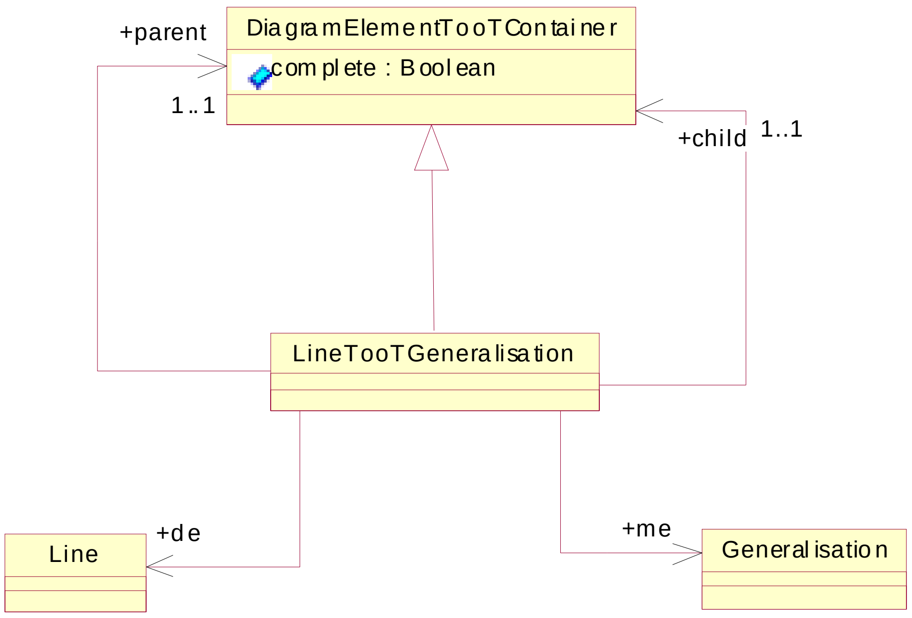

Figure 38. Graphical syntax for generalisations

that maps to a box, otherwise they are omitted. The other invariants are written taking this into account, but make sure when there is a generalisation arrow that it is source/ targeted correctly and has the correct arrow head.

```
context LineTooTGeneralisation inv:
     "generalisations_visible_only_between_boxes"
     (self.child.de->oclIsKindOf(Box) and
       self.parent.de->oclIsKindOf(Box))
     = not self.child.de->isEmpty

context LineTooTGeneralisation inv:
    "parent-child_TooTs_configured_ok"
    self.parent.me=self.generalisation.parent and
     self.child.me=self.generalisation.child

context LineTooTGeneralisation inv:
    // line ends positioned on boundary of parent and child boxes
    "positions_of_line_ends"
```

```
        (self.parent.de.onBorder(self.de.endA.position) and
           self.child.de.onBorder(self.de.endB.position))
         or self.de->isEmpty

    context LineTooTGeneralisation inv:
        "arrow_head_appropriate"
        self.de.endA.arrowHead.closed or self.de->isEmpty
```

The last set of invariants relate to Figure 39. They ensure that a class gets the correct

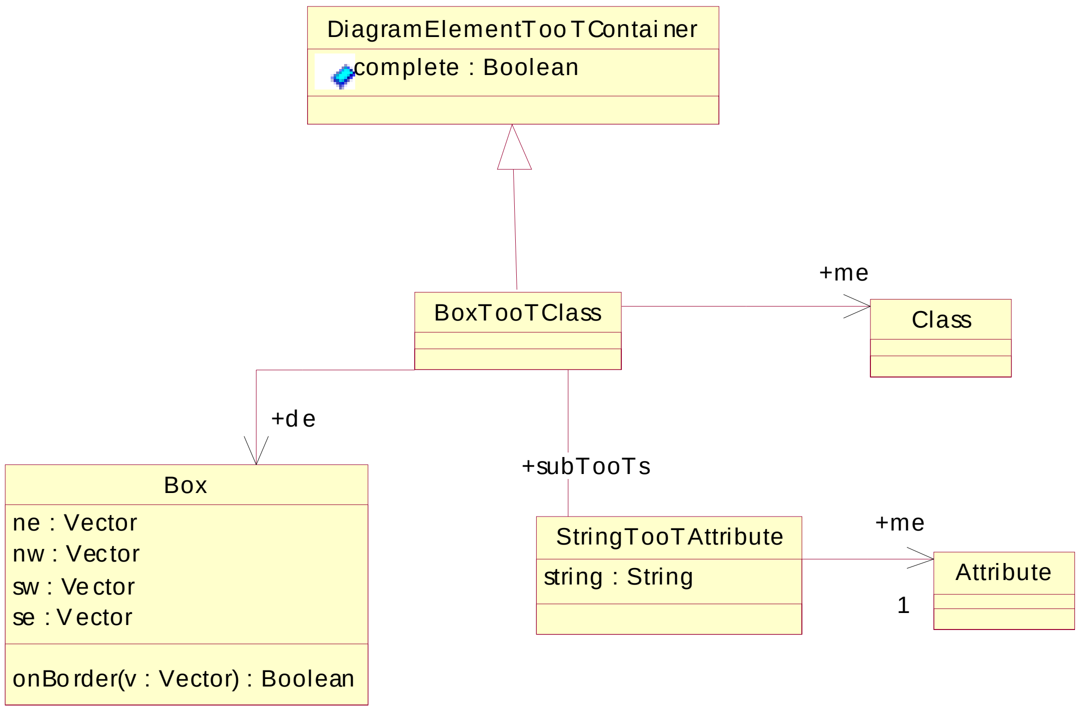

Figure 39. Graphical syntax for classes

number of compartments, including a compartment to hold the name, that attributes get mapped to text appropriately, and that the text is placed appropriately inside the second compartment.

```
    context BoxTooTClass inv:
        "correct_compartments_count"
        self.subTooTs->notEmpty implies
         self.box.allCompartments->size=2 else
           self.box.allCompartments->size=1)

    context BoxTooTClass inv:
        "name_compartment"
        self.box.first.text.slots.value=self.class.name

    context BoxTooTClass inv:
        "attributes"
        self.box.first.next.text.slots->size=self.subTooTs->size and
         self.box.first.next.text.slots.value=self.subTooTs.string

    context StringTooTAttribute inv:
        "define_string"
        self.string=self.attribute.name+":"+self.attribute.type.name
```

5.3.2 Mapping of modelManagement.model.concepts to box and line diagrams

The graphical syntax for packages is an extension of the graphical syntax for the elements in the static core. By stating that modelManagement is an extension of staticCore, modelManagement obtains the expected model/instance and syntax/concepts components. The additional part, which is contained in the package *modelManagement.model.graphicalTooTconcepts* (which must be a specialisation of *staticCore.model.graphicalTooTconcepts*) is given by the class diagram in Figure 40 and accompanying invariants. A package is mapped to a box with a tab. The

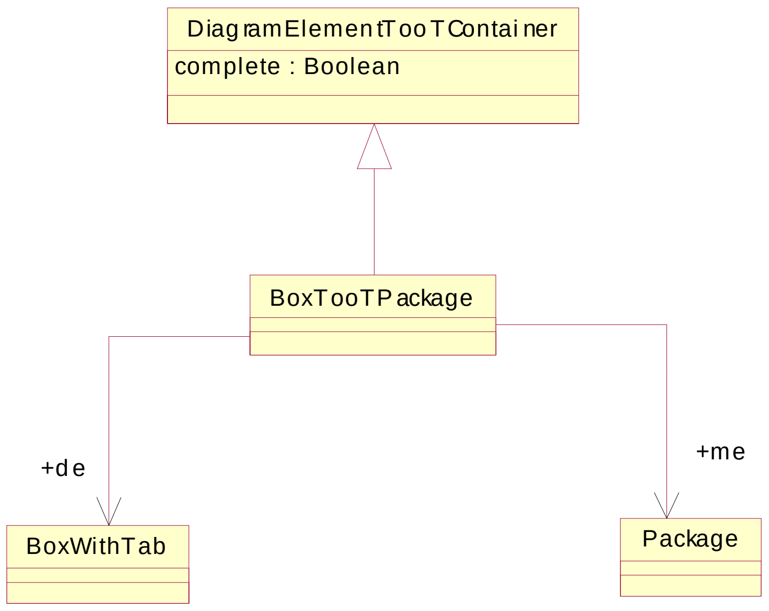

Figure 40. Syntax for packages

first invariant ensures that the name on the tab is correct:

```
context BoxWithTabTooTPackage inv:
    "name_on_tab"
    self.box.tab.text.slots.value=self.package.name
```

Recall that a mapping (a TooT) may contain submappings (subTooTs). The second invariant ensures that these are of the right kind.

```
context BoxWithTabTooTPackage inv:
    // subTooTs for a package TooT are TooTs of classes to boxes
    // and lines to generalisations
    "define_toots"
    self.subTooTs->forAll(t|t.isKindOf(BoxTooTClass) or
     t.isKindOf(BoxTooTPackage) or
     t.isKindOf(LineTooTGeneralization))
```

The final invariant ensures that any boxes representing classes in the package which have been mapped (that is, are being shown on the diagram) are contained in the box representing the package.

```
context BoxWithTabTooTPackage inv:
    "contains_class_boxes"
    self.subTooTs->select(t|t.isKindOf(BoxTooTClass).box
        ->forAll(b|self.box.contains(b))
```

5.4. XMI

5.4.1 Mapping of staticCore.model.concepts to XML

The mapping to XML is defined in a similar way the graphical syntax, but this time replacing box and line diagrams with XML. The mapping package is made up of the class diagram in Figure 41, with some accompanying invariants.

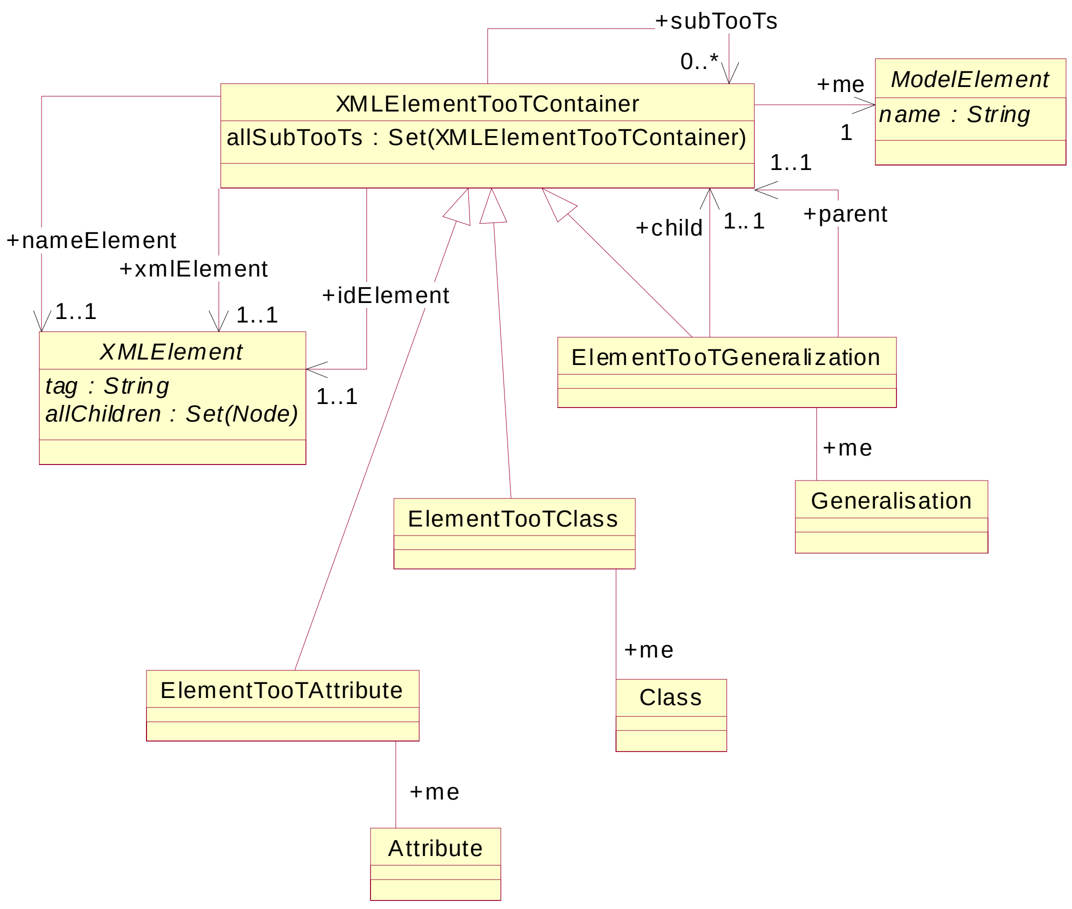

Figure 41. XML mapping for staticCore.model.concepts

The first set of invariants define the constraints on the mapping between any model element and XML elements. It is the case that all model elements map to XML elements. Text nodes come about from textual information accessible from a model element.

```
context XMLElementTooTContainer inv:
    "put_in_header"
    self.parentTooT->isEmpty implies
        (self.xmlElement.parent.tag="xmi" and
        self.xmlElement.parent.children->size=2 and
        (self.xmlElement.parent-self.xmlElement).tag=
         "xmi-header")
```

This invariants just illustrates how the tags wrapping a document, and the header of the document would get set up, which is done only when a mapping (a TooT) is no a sub-TooT of any other TooT. There could be many more constraints to put in other header information (e.g. versioning) if so desired.

The next invariant is the same as in the graphical syntax case.

```
context XMLElementTooTContainer inv:
    "all_subelements_of_model_element_catered_for"
    self.me.oclIsKindOf(Container) implies
self.subTooTs.me=self.me.elements
```

The name element is the XML element which holds the name of a model element.

```
context XMLElementTooTContainer inv:
    "name_element"
    self.nameElement.tag="name" and
    self.nameElement.children.isKindOf(Text) and
    self.nameElement.children.asType(Text).slots.value=
     self.me.name
```

Every model element must be associated with an id in the XML document so that it can be cross-referenced elsewhere in the XML document (see e.g. generalisation).

```
context XMLElementTooTContainer inv:
    "id"
    self.idElement.tag="id" and self.idElement.isKindOf(Text) and
        self.idElement.children.asType(Text).slots.value=self.id
```

To ensure that cross-referencing of id’s works as expected, the id’s must be unique between all children (direct and indirect) of the any XML element that represents a model element. This is achieved by checking the id associated with all subTooTs of a TooT.

```
context XMLElementTooTContainer inv:
    "all_id's_unique"
    self.id->union(self.allsubTooTs.id)->forAll(x,y | not x=y)
```

Define allsubTooTs.

```
context XMLElementTooTContainer inv:
    "allsubTooTs"
    self.allsubTooTs=self.subTooTs->
     union(self.subTooTs.allsubTooTs)
```

Define the children of the xml element associated with a TooT.

```
context XMLElementTooTContainer inv:
    "children"
    self.xmlElement.children=(self.subTooTs.xmlElement
     ->union(self.idElement)->union(self.nameElement))
```

Ensure that a model element is not represented twice an XML document.

```
context XMLElementTooTContainer inv:
    "no_repeats"
    self.allChildren.me->asSet=self.allChildren.me
     // relies on bags
```

The details for each of the different kinds of TooT now amount to stating what the tag of an xml element representing each kind of model element should be.

```
context XMLElementTooTClass inv:
    "tag name"
    self.xmlElement.tag="class"

context XMLElementTooTAttribute inv:
    "tag name"
    self.xmlElement.tag="attribute"
context XMLElementTooTGeneralization inv:
    "tag name"
    self.xmlElement.tag="generalization"
```

Attributes and generalisation need to cross reference other parts of the xml document: the id's identified with the parent and child model elements for a generalisation, and the id of the model element that is the type of an attribute.

```
context XMLElementTooTGeneralization inv:
    "parent/child"
    self.xmlElement.children->size=4
    // i.e. name and id tags plus parent and child tags
    and self.xmlElement.children->exists(el |
        el.tag="parent" and el.children.isKindOf(Text) and
        el.children.asType(Text).slots.value=self.parent.id)
    and self.xmlElement.children->exists(el |
        el.tag="child" and el.children.isKindOf(Text) and
        el.children.asType(Text).slots.value=self.child.id)
    and self.child.me=child
    and self.parent.me=parent

context XMLElementTooTAttribute inv:
    "type"
    self.xmlElement.children->size=3
    // i.e. name and id tags plus this extra type tag
    and self.xmlElement.children->exists(el |
        el.tag="type" and el.children.isKindOf(Text) and
        el.children.asType(Text).slots.value=self.type.id)
    and self.me.type=self.type.me
```

5.4.2 Mapping of modelManagement.model.concepts to XML

As was the case with the graphical syntax mapping (section 5.3.2), this is just a simple extension for the mapping for *`staticCore.model`*. Specifically, the additional component which is part of the package *modelManagement.model.XMLTooTconcepts*, is given by the class diagram in Figure 42, with one accompanying invariant.

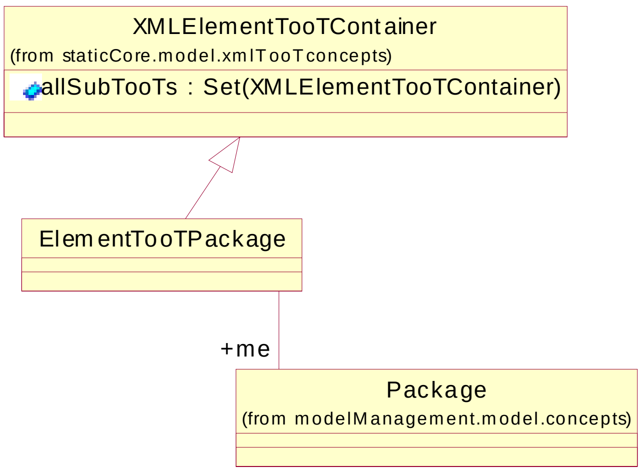
XMLElementTooTContainer
(from staticCore.model.xmlTooTconcepts)
allSubTooTs : Set(XMLElementTooTContainer)
ElementTooTPackage
+me
Package
(from modelManagement.model.concepts)

Figure 42. XML mapping for modelManagement.model.concepts

```
class XMLElementTooTPackage inv:
    "tag name"
    self.xmlElement.tag="package"
```

5.4.3 XML Issues

We have presented an example of mapping to XML just to demonstrate that, in principle, it can be specified using MML. However, we are aware of a number of issues that need to be pursued beyond this feasibility study.

Generalization. On the concepts side, generalisation is currently defined so that all elements inherited from parents are repeated in the child. This makes it easier to allow renaming and the like on specialisation. However, it has inherent redundancy built in, and if this approach is repeated in the XML, then the resulting files will also contain redundant information and therefore be larger than they need to be. The approach to generalization taken in the class diagram mapping, shows that we can identify and ignore redundant information as necessary.

Dual representation of classifiers. If one maps both *instance.concepts* and *model.concepts* to XML, and takes into account that, in *reflection.concepts Classifier*, inherits from *Object*, we see that a *Classifier* could be mapped two ways into XML, via the mapping specified separately for each subclass of *Classifier*, or as the XML representation of an *Object*. This suggests that one could dispense with the XML mapping to classifiers completely and instead only deal with the mapping to *Object*. Our opinion is that dispensing with something just because it can be dispensed with is probably not a good idea. It is certainly true that, for tool builders, processing the XML generated from classifiers is likely to more obvious, hence easier, than dealing with the XML generated from viewing classifiers as objects.

Mapping semantics. It is clear that we can map model.concepts to XML, and the mapping of instance.concepts follows in a similar way. The question is whether there is any need or desire to map semantics. By mapping semantics to XML, one can use XML to preserve and/or exchange the tying together of a particular instance (or instances) to a particular model. For example, one would record in the XML exactly which classifier the element was of. If one just mapped the instance, then one would only record the name of that classifier.

Alternative representation of instances. One way to map instances would be to mimic the approach taken for classifiers, with tags for the subclasses of *`Instance`*, i.e. *`Object`*, *`Slot`* etc. This is the approach assumed so far. A more sophisticated approach would be to leverage the classification mapping in the semantics (the ‘of’ association) to provide the name tags to be used and to provide the element containment structures to be used in the XML.

If this mapping could be specified, then the mapping of classifiers in the way performed in this section would not have to be separately specified. It comes automatically from this mapping of instances, as, through reflection, all classifiers can be viewed as instances that are classified by *`Class`*.

Mapping of other syntaxes, such as diagrams, to XML. The motivation for doing this is to preserve and exchange purely syntactic information, such as diagram layout, which would be lost if only the conceptual layer was mapped to XML. We do not perceive any difficulty in doing this: it is just a case of modeling, for example, a mapping between box and line diagrams and XML.

5.5. Conclusions

This chapter has described how to map diagrams and XML to concepts which have already been elaborated in the description of MML, thereby demonstrating that it is possible to provide a complete and unambiguous description of syntax using MML. A certain style of mapping was developed, which is recursive in nature, and which reflects the containment structures of each side.

By carefully choosing the diagrams to be considered, we have also demonstrated how a syntax mapping may be extended using package generalization. Although not done, it should also be reasonably obvious to the reader that the box and line diagrams could be reused for other syntax mappings, e.g. from staticCore.instance.concepts to object diagrams. We have only mapped one set of concepts to XML, just to demonstrate the feasibility.

To progress the work begun here, we just have to do more modeling and check with those involved in the standardization of UML that the syntaxes we are defining are true to the standard and that the mappings are as expected. There is still room for refactoring and improving the models presented. In particular, bring out the TooT pattern, including subTooTs into a package, as demonstrated in Section 5.2.1 for Lists. We just ran out of time.

One area of possible contention is the degree to which the diagrammatic syntax should be standardized. For example, should it be part of the syntax definition, that compartments in boxes should always be large enough to fully contain the required strings. We have made the decision that position of boxes on the page is not part of the standard, nor are other layout considerations such as colour, thickness of lines, font etc.

With regard to the XML mapping, we have essentially followed an approach similar to the XMI 1.x. Indeed the model is not dissimilar to the specification of the mapping of UML to XML in the XMI standard, except that instead of mapping directly to strings we map to an abstract syntax of XML (assuming that there are already tools that perform the laborious task of parsing/generating strings into this abstract structure). No doubt our model of XML is a little simplistic and the mapping a little naive; we hope to work with those who have worked on XMI 1.x to help specify a more complete and improved version. A number of issues were raised with concern to the XML mapping. In particular, it seems that it may be possible to leverage reflection and the “instance of classifier” relationship to develop a “self-generating” XML mappings.

Chapter 6

Dynamic aspects

This chapter presents some initial work on extending MML with dynamic aspects.

6.1. Overview

In this section, it is shown how a simple dynamic model can be added to the MML as an extension of the staticCore. Its purpose is to provide a core collection of dynamic model and instance concepts that can be imported and reused across all behavioural modelling extensions of MML. For example, interaction diagrams, state machines, and use-cases could all be viewed as extensions of the dynamicCore.

The main focus of the dynamicCore package is on the notion of an Action. An action represents a computational procedure that changes the state of the system. Actions are associated with objects, but can have a global or local effect, i.e. they may change the values of their slots or the slots of other objects. In addition, they may send messages to other objects by creating messages.

6.2. dynamicCore.model.concepts

An action is named element, which has parameters, returns a result of type resulttype, and has a pre- and post-condition expression associated with it.

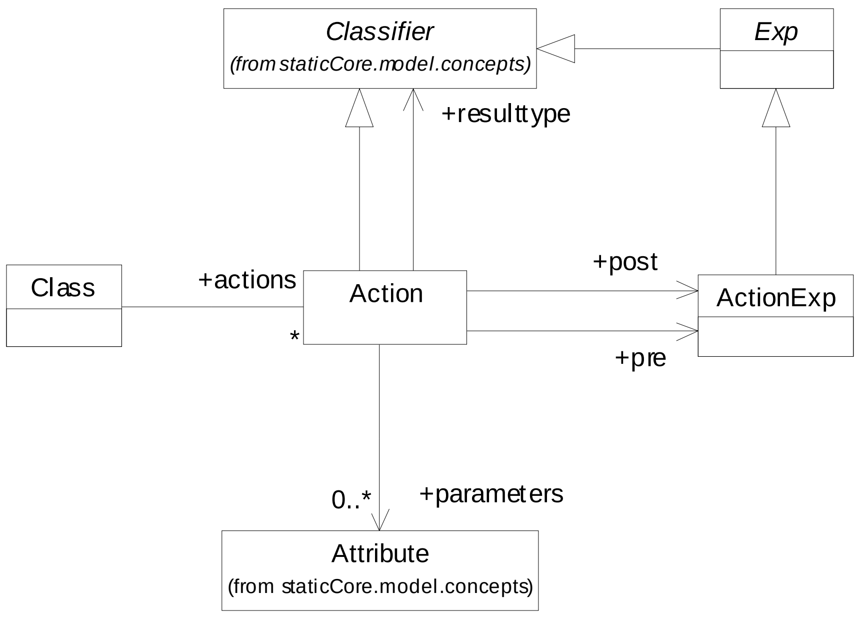

Figure 43. dynamic.core Package

Well-formedness Rules

[1] The elements of a class contain its actions.

```
context uml.dynamicCore.model.concepts.Class inv
  elements->exists(g |
    g.name = "actions" and
      g.elements = allActions())
```

[2] The method allActions() returns the set of all actions of Class, including those of its parents:

```
uml.dynamicCore.model.concepts.Class
  allActions() : Set(Action)
  parents->iterate(p s = actions |
    s->union(p.allActions()->reject(c |
      actions->exists(c' | c'.name = c.name)))
```

[3] The elements of an action contain its parameters, pre- and post- conditions and returntype.

```
context uml.dynamicCore.model.concepts.Class inv
  (elements->exists(g |
    g.name = "parameters" and
      g.elements = parameters)) and
  (elements->exists(g |
    g.name = "pre" and
      g.elements = pre)) and
  (elements->exists(g |
    g.name = "post" and
      g.elements = post))and
  (elements->exists(g |
    g.name = "returntype" and
      g.elements = returntype))
```

Note, as described in the MML chapter, the purpose of these constraints is to set up a general container mechanism for action. Without this mechanism, it would be necessary to re-specify the effect inheritance has on the conformance of an action with those of its parents. Because the rules regarding inheritance of contained elements have already been defined in the MML, these will be automatically applied to actions.

6.3. dynamicCore.instance.concepts

The dynamicCore.instance.concepts package extends the core with the notion of a history (see Figure 44). A history is a sequential record of action invocations and snapshots. As described in the MML specification, a snapshot represents the current objects in the system (instances) and the current links that exist between instances. In addition, the dynamicCore.instance.concepts package extends a snapshot with the current messages being sent between objects. A message is an instance, which has a sender and receiver, a set of argument values and a result.

Changes in state are caused by an action calculation. An action calculation is a special extension of a calculation (described in the MML specification). Given a snapshot that satisfies its pre-condition, an action calculation results in a snapshot that satisfies its post-condition. Thus, the behaviour of any system is viewed in terms of a series of action invocations, in which instances may be created and deleted, messages sent and received, and objects slots are modified.

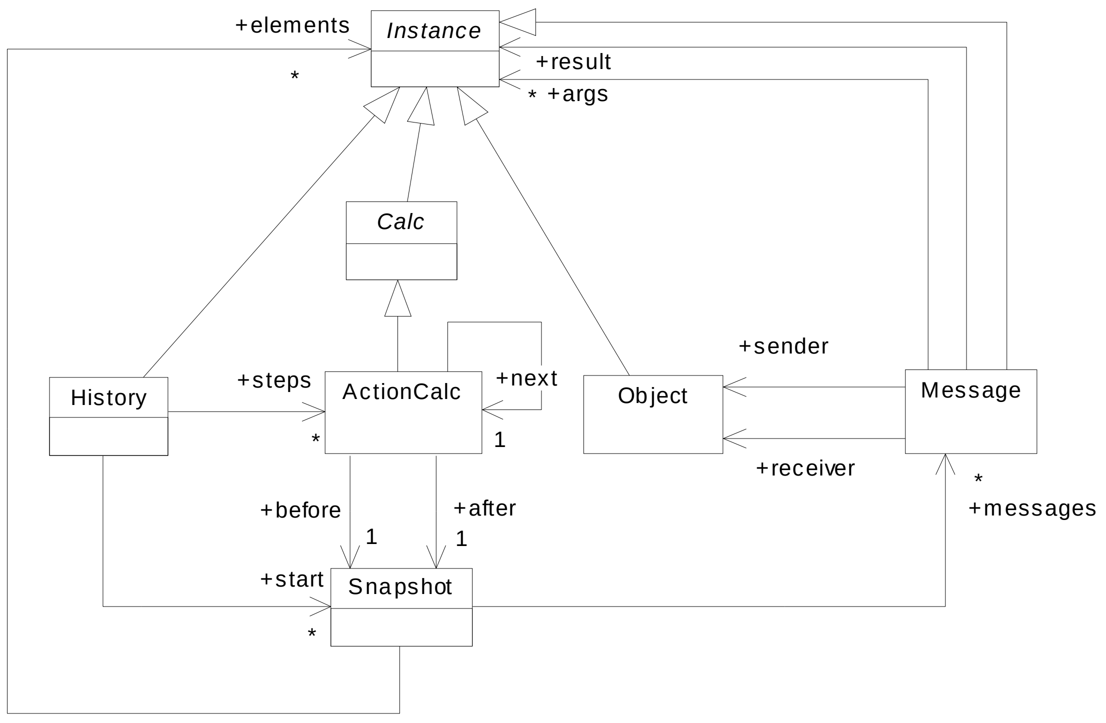

Figure 44. dynamicCore.instance.concept Package

Well-formedness Rules

[1] Messages can only be sent and received between Instances that exist in the current snapshot

```
context dynamicCore.instance.concepts.Snapshot
  message -> forall(m |
    elements -> includesAll(Set{m.sender, m.receiver))
```

[2] A history must start in one of the initial states of the system.

```
context dynamicCore.instance.concepts.History
  start -> includes(steps[1].before)
```

[3] Each step in a history is related to the previous one as a result of an action calculation. Note that it may be the case that there are a number of action calculations that could lead from one state to another. However, it is undetermined (i.e. non-deterministic) which will be chosen.

```
context dynamicCore.instance.concepts.History
    steps -> forall(s1, s2 | s1.next = s2 implies
       s1.after = s2.before)
```

Whilst the above model uses a sequential model of behaviour, it is powerful enough to express both interleaved and non-interleaved models of concurrent execution. This is because independent actions may be thought of as being able to execute in any order thus simulating partial orderings. Partial ordered models of concurrency are a widely and successfully used approach to the description and verification of concurrent systems, (as described by the work of Lamport, Hoare and Milner for example). Furthermore, they may be easily extended to describe both soft and hard real-time systems.

[4] The elements of a History are its steps and starting state; the elements of an action calculation are its pre- and post- snapshots; the elements of a package now includes messages; the elements of a message are it sender and receiver objects.

6.4. dynamicCore.semantics

The dynamicCore.semantics package associates concepts in the instance.concepts package with their classifiers in the model.concepts package. A history is associated with a package. Action calculations are associated with action expressions, while messages are associated with message expressions (see Figure 45). A message expression is a subclass of action expression, and defines the syntax of a method call.

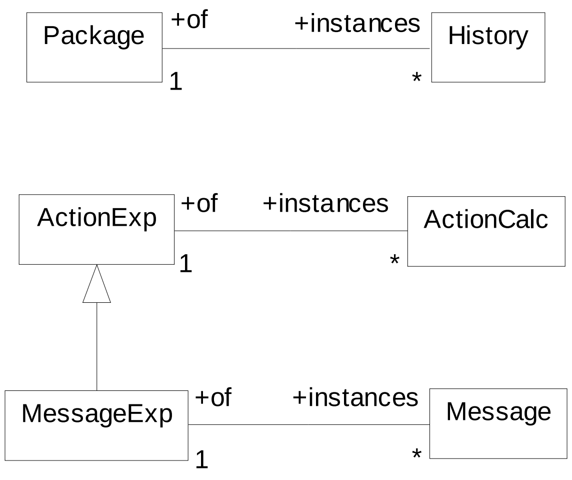

Figure 45. dynamicCore.semantics Package

Note that a definition of action and message expressions is not given here. Essentially, these expressions will form a part of a much larger action language, which will include all the necessary constructs required to describe action specifications[1]. However, whatever language is used, the architecture described above aims to make it easier to add to the dynamicCore, simply by extension of the appropriate action expression and calculation classes.

6.5. Real-time extensions

Once the dynamicCore is defined, extensions to deal with real-time, state charts and other behavioural modelling tools are made simpler. As an example, consider the addi-

1. For example, as described by the UML action semantics submission (www.umlactionsemantics.org).

tion of time. The dynamicCore can be easily extended to deal with timing properties by the extension of snapshot by a timed snapshot. If we assume that the timedCore package extends the dynamicCore package, the contents of the timedCore.instance.concepts package will be as shown in Figure 46.

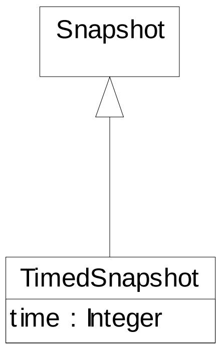

Figure 46. timedCore.instance.concepts Package

A timed snapshot simply records the time at which the snapshot was created.

The following constraint is required to ensure that time keeps increasing during a history:

```
context timedSnapshot.instance.concepts.History
  steps -> forall(s | s.after.time > s.before.time)
```

Again, as in the case of the history model described in section 6.3, timed histories have a sound theoretical foundation, and have been successfully used to model a variety of real-time systems (including soft, hard and distributed real-time properties therein).

6.6. Conclusion

This chapter has shown how the extension mechanisms of MML can be used to extend the language with a dynamicCore. The purpose of the dynamicCore is to provide a pluggable and extensible definition of dynamic behaviour, which can be used as a foundation for specifying further dynamic languages, including real-time, state-based, and interaction based modelling language.

It is envisaged that by providing a single definition of behaviour in this way, greater unification and consistency will be achieved when developing future dynamic components of the UML.

Chapter 7

Conclusions

This chapter brings together summarises the earlier chapters. It then considers how to progress from this feasibility study, paying particular attention to issues of legacy and standardization process, once one has accepted that UML is a family of languages.

7.1. Summary

This report has been a feasibility study in rearchitecting UML as a family of languages, and, at the same time, providing a precise definition of all its aspects, including syntax (both concrete and abstract) and semantics. The study recognizes the audience at which this definition is aimed, and the importance of using an accessible language in which to phrase the definition. In that vein, the study has pinned down the abstract syntax and semantics of a meta-modeling language that is one of the members of the UML family, using familar constructs such as classes, attributes and packages. This language has been defined in terms of itself (Chapter 4) and slotted into the UML family whose architecture has also been defined in MML (Chapter 3).

In addition to abstract syntax and semantics, the study (Chapter 5) has developed mappings between graphical syntax and XML, respectively, and the underlying concepts (abstract syntax) of various language fragments. In addition to those language fragments required to define MML, the study has sketched (Chapter 6) how other areas of UML, in particular notations for dynamic modeling, can be built on top of parts already defined.

During the study, a tool has been prototyped, implementing many aspects of MML. A brief overview of the tool is given in the next section.

7.2. Tool

The importance of implementing a tool to support the work described here was stressed in Chapter 1 of this report. At the time of writing a prototype tool implementing MML has been built, but there has not been the time to write up that work. The approach taken has been to implement a very small subset of MML (essentially objects, slots, classification and constraints), and then bootstrap the rest of MML on top of this. This provides considerable flexibility, as it allows MML to be refactored (within limits) without having to change the program source code. It is believed that, because of the reflective nature of MML, it will be possible to use this tool to provide powerful semantic checking and model execution facilities for meta-models, and their models, built using MML. However, this remains to be fully tested.

For this approach to work, the style in which one writes OCL invariants has to be such that they can be executed in some sense. Thus, at the current time, some conversion of the models specified in this document has had to be undertaken to put them in a form suitable for input into the tool.

The tool will be available from www.puml.org in the near future.

7.3. Further Work

7.3.1 Tool

- Write up the tool work more fully.
- Move away from developing the models using a mixture of Rose diagrams and text, to maintaining the models only in the prototype tool. We hope and expect to be able to using the tool to generate diagrams using by writing a mapping, in MML, to an appropriate XML picture format.

7.3.2 Areas of UML

- Much effort in this feasibility study has focused on trying to tease out patterns to assist the meta-modeler during language design. This has not been completed to our satisfaction. In particular, we feel there is room for moving patterns such as generalization, container/contained, instance/of, which are currently encoded in class hiearchies, into packages that can then be specialized, much in the same way that a `LinkedList` package was used in the definition of Text/StringSlot and Box/Compartment in Chapter 5. Patterns of packages, e.g. model/instance/semantics, syntax/concepts/TooTs, could be treated in a similar way, noting a package can contain other packages.
- One are that did not receive as much attention as we had hopd is the modeling of dynamic aspects. We have hinted at how we would proceed with this, and expect to be able to deal with a large chunk of the dynamic part of UML using our techniques. The separation of syntax from concepts should help here considerably, for example teasing out the specification and execution trace representation uses of sequence diagrams. The biggest issue is how to deal with complex aspects of dynamic behaviour concerned with concurrency and real-time, which have not been fully resolved in UML. We note the recent appearance of the work on Action Semantics, recently submitted for consideration by the OMG, and would hope to align our work in this area with that.
- We are keen to investigate new opportunities for the development of modeling languages that MML and our approach to architecting the definition of UML affords. In particular:
    - languages for expressing constraints visually and prototypically
    - techniques for modeling by example (by prototype), using well defined and integrated syntaxes for viewing traces of behaviour (e.g. filmstrips combined with sequence diagrams to visualise a trace)
    - visual syntax for regularly occurring patterns, such as patterns that occur in language definition when using MML

7.3.3 Backwards compatibility

This feasibility study demonstrates that as soon one adopts a rigorous approach such as that we have proposed, there is trong likelihood that the existing meta-model for UML and, therefore the existing XMI standard, will change in non-trivial ways. This raises a question of backwards compatibility with the 1.x standard.

We believe the approach can accomodate backwards compatibility issues quite cleanly. Much play has been made throughout the report of the representation of mappings as models, for example the semantics mappings between models and instances and the mappings between syntax and concepts. The same technique could be used to solve the bacwards compatibility issue. Thus, for example, one could write a mapping from a model of XMI 1.x (which would be a specialization of the model of XML introduced in Chapter 5) the conceptual layer for all the parts of UML that have been redefined, noting that there may be some (but not much) misalignment, as adopting a rigorous approach, such as ours, will always identify areas of UML 1.x that require refactoring or are simply unwanted. The intention would be to at least ensure that the graphical syntax associated with UML 1.x barely changed, if at all, with the qualification, again, that problems may be discovered when a complete and precise definition of the graphical syntax is attempted.

Once UML has been redefined using the proposed approach, questions of backwards compatibility will be less of an issue, as the whole purpose of this approach is to support the evolution of UML as a family of languages. The next section explores this a little further, as part of considering the standardization process.

7.3.4 Standardization Process

We have said little about the standardization process itself. The theory expounded in the report, and the tool implementing this theory, supports a much more rigorous standardization and signing off procedure. The idea that UML is a family of languages changes our perception of what a standard UML is. In this section we consider, in order:

- the standardization of family members
- acceptance of refactorings of the existing family
- conformance to the standard

Standardizing new family members

We should not think of a single standard language, but rather a set of family members and language components that have, in some sense, been standardized. This suggests that mechanisms need to be in place to standardize family members, that is UML profiles.

When standardizing a family member the following factors need to be taken into account:

- Does the new family member conform to the established pattern for developing UML family members? Is syntax separated from concepts, model from instance, and so on?
- Has the new member introduced concepts that are already included in existing members or language components? It would better if the introduction of new concepts was kept to the minimum.
- Which of the following does the new family member change/specialize:
  - syntax
  - model concepts
  - instance concepts
  - semantics

  The mechanisms for standardization may be tougher depending on nature of the changes/specializations. Specializations are preferred over extensions/changes. Intereference at only the syntax level is preferred over intereference with model concepts which is preferred over intereference with instance concepts and semantics. For example, additional concrete syntaxes for existing concepts might just be accepted with little discussion, perhaps with some syntaxes recommended over others. A new family member introducing a whole host of new concepts and semantics is likely to require much more careful consideration before it is endorsed.
- Is the definition complete? Is it too complete (for example, does it overspecify the syntax part? Has the definition been tested? Does the definition come with a comprehensive set of examples, illustrating the various aspects of the definition? We believe that these questions can only be answered if MML is properly supported by tools.

Refactoring the family

In addition to standardizing new family members, one must also consider proposed refactorings to the existing family. For example, when developing a new family memeber it may be the case that some concepts in an existing member can be reused. These need to be pulled out to a reusable language component. There is a class of refactorings that will preserve existing members (just the way those members are constructed might be different) and a class of refactorings that don't. Should only refactorings in the first class be allowed? If not, what is the procedure for accepting the second refactorings in the second class?

Conformance

It is hard to check whether or not a tool or method conforms to the UML 1.x standard. Because of the way in which the standard is written down, conformance can only be checked by human inspection. The only concrete artefact that can be automatically checked is conformance to the XML interchange format for models, because only this has been specified to the detail required to check that a tool implementing this standard conforms to it. This does not include the mapping from XML to the meta-model, as conformance to the meta-model can only be established through inspection.

Checking conformance to UML, when viewing UML as a family of languages, is more sophisticated. The problem can be split into two:

- Exactly which part of the definition does the tool implment? Is conformance being checked against syntax, concepts, model, instance and/or semantics? A tool might conform to the definition of a graphical syntax say, but might not conform to the mapping of that syntax to concepts. A tool might implement model.concepts, but not instance.concepts and not the mapping between them. A tool might implement the semantics part (e.g. checking instance against model), but not implement any of the syntax.
- Once one has identified the part of the definition being implemented, how does one check conformance? One approach would be to supply a standard set of examples (models, instances of models) that the tool must deal with correctly. This is equivalent to a set of conformance test cases. A published set of examples would not be enough, as one could tune a tool just to work with those examples. Some examples would have to remain hidden, perhaps even be randomly generated. It is unlikely that such an approach will be feasible without significant tool support.

References

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

D'Souza D., Sane A. and Birchenough A. (1999) *First-Class Extensibility for UML - Packaging of Profiles, Stereotypes and Patterns*, in France & Rumpe (1999).

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

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.

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

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