

XMF Bluebook

XMF Bluebook

Table of Contents

Introduction and Tool Overview	1
1. Introduction	3
What is XMF-Mosaic?	3
Using XMF-Mosaic	3
Domain Models	4
Concrete Syntax	4
Behaviour	5
Transformations and Mappings	5
Extensibility	5
Why use XMF-Mosaic?	6
Technical Features	6
Architecture	7
Operating System	7
Virtual Machine	7
Kernel	8
XOCL	9
Base Toolset	9
Tool Definition	10
XMF Clients	10
XMF Walkthroughs	12
2. Creating and Interacting with a Domain Model	16
Example	16
Getting Started	16
Constructing a Domain Model	18
Adding Classes	18
Adding Attributes	21
Adding Associations	23
Using the Property Editor	24
Saving and Loading	25
Interacting with Models	26
Snapshots	26
Creating Standalone Snapshots	29
Dealing with Sequences	30
Synchronisation between Snapshots and Class Models	31
Creating Multiple Views of Snapshots	32
Exporting Snapshots	32
Saving and Loading Diagram Layout	33
Adding Constraints	33
Constraint Reasons	35
Parse Errors	35
Checking Snapshots	36
Exercises	38
Adding Queries	38
Exercises	40
Running Operations	40
Running Operations on Snapshots	40
Operations	42
Exercises	43
Adding Getters and Setters	43
Exercises	44
Adding Constructors	44
Exercises	45
Interacting with Models using the Console	45
Other Console Hints and Tips	46
Editing Values via the Console	47

Using Global Variables	48
Error Reports	48
Using Snapshots	49
Generating Snapshots from the Console	49
Adding Package Operations	49
Adding and Running Package Operations	49
Next Steps	50
3. Working with the Modelling Interface	51
Windows	51
Displaying Multiple Diagrams and Windows	51
Preferences	53
Property Editor	53
4. An Introduction to XOCL	55
Introduction	55
Basic Types	55
Examples	55
Models	55
Using XOCL	57
Context	57
Self	58
Operations	58
Constructors	59
Constraints	59
Variables	60
Types	60
Navigation	61
Collections	61
Collection Operations	62
Logical Expressions	63
Conditional Expressions	63
Imperative Features	64
Object Creation	65
Assignment	65
Sequential Execution	65
Operation Invocation	66
Looping	66
Exceptions	66
Formatting	67
Advanced Features	68
5. Debugging Operations	69
6. Constructing and Running Mappings	70
A Simple Mapping	70
Running the Mapping	73
A Simple Model to C# Mapping	74
The C# Model	74
The Mapping	75
Executing the Mapping	76
Mapping Hints and Tips	77
Error Reporting	77
Constructing a Pretty Printer	78
7. Constructing an XML Parser and Generator	81
First Steps	81
Constructing the Grammar	81
Invoking the Parser	82
Example	83
Debugging the Parser	83
Generating XML	85
8. Using the Programming Interface	88

Getting Started	88
Compiling and Loading	90
Checking the Model	90
Adding Constraints	90
Adding Operations	90
Context	91
Importing Packages	91
9. Toolbar Menus and Initialisation Files	92
Toolbar menus	92
Some Useful Operations	92
Initialisation Files	93
10. Constructing a Diagram Tool for a Model in XTools	95
Domain Model	95
A Candidate Diagram Syntax	95
Constructing an XTool Definition	96
Creating the Tool Definition	96
Adding a Node	96
Adding a Box to the Node	97
Adding an Attribute to the Box	98
Adding a Free Node	99
Adding Tool Bars	101
Running the Tool	101
Adding Edges	102
Adding Edge Toolbar Buttons	104
Adding Menus	104
Re-running the tool	104
Other XTool Capabilities	105
11. Importing XMI	106
12. Constructing a Textual Syntax and Parser	107
Parsing and Synthesising Instances of Models	107
13. Creating a Meta-Profile	109
An Example Profile	109
Adding Constraints	114
14. Generating Code	116
Generating Java	116
15. Using Manifests and Deploying Models	118
Manifest Actions	118
Deploying Manifests	118
Reference	120
16. Namespace, Classes, Packages and MetaClasses	125
Introduction	125
NameSpaces	125
Classes	125
Class Definition	125
Attributes	126
Operations	126
Constraints	127
Inheritance	127
Packages	128
Metaclasses	128
Message Passing	128
Object Creation	129
Slot Access	130
Slot Update	131
Default Parents	131
Example	131
17. Working with Syntax	134
Introduction	134

Grammar and Text Processing	134
Introduction	134
A Simple language Grammar	134
Debugging	140
XBNF Grammar	143
The Grammar Domain Model	143
The XBNF Grammar	145
Tokens	149
XMF Execution Architecture	149
Introduction	150
Performable Elements	150
Syntax Extensions	152
Synthesising Syntax	152
Introduction	153
The OCL Package	153
Examples	156
Quasi-Quotes	161
Introduction	161
Literal Syntax	162
Syntax Templates	162
Splicing	162
Patterns	162
New Performable Elements	162
Introduction	162
Sugar	163
Syntax	168
Exp	168
18. XCore	182
Introduction	182
The XCore Package	182
Attribute	183
Behavioural Feature	183
Bind	183
Classifier	183
Class	183
CodeBox	183
Collection	183
Compiled Operation	183
Constraint	183
Constraint Report	184
Constructor	184
Contained	184
Container	184
Daemon	184
Data Type	184
Dependency	184
Doc	184
DocumentedElement	184
Element	184
Enum	185
Exception	185
ForeignOperation	185
IndexedContainer	185
InterpretedOperation	185
Namespace	185
NamedElement	185
Object	185
Operation	186

OrderedCollection	186
Package	186
Parameter	186
Performable	186
Resource	186
Seq	186
Set	186
Snapshot	186
StructuralFeature	187
Table	187
Thread	187
TypedElement	187
Unordered Collection	187
Vector	187
19. XMap	188
Introduction	188
Language Basics	188
Syntax	189
Diagrammatical Syntax	189
Textual Syntax	190
Clause Syntax	190
Execution	191
Constructing Mappings	191
Creating Mappings via the Modelling Interface	191
Creating Mappings via the Programming Interface	194
Running Mappings	194
Example	195
Class Model to Database	195
Running the Mapping	200
Other Aspects of Mappings	201
Operations	201
Attributes	202
Variable Passing	202
20. XML	204
Introduction	204
XML	204
Parsing XML	205
Introduction	205
Example	205
Debugging XML Grammars	217
The XML Parsing Grammar	220
DOM Input Channel	220
SAX Input Channel	222
XML Output	226
Introduction	226
XML Output Patterns	226
XML Output Channels	233
Raising Events	234
Deploying Java	234
Introduction	234
Deploying Models	234
Deploying Parsers	234
Deploying Factories	234
21. XOCL	235
Introduction	235
Purpose	235
Language Basics	236
Overview of Syntax	237

Basic Data Types	237
Program Constructs	238
Self Evaluating Expressions	238
Variables and Update	239
Calling Operations	240
Infix Operators	240
Prefix Operators	241
Sequencing	241
Special Forms	241
Quasi-Quotes	242
The Meta Character @	242
Documentation	243
Error Handling	243
Control Statements	244
If	244
Case	244
CaseInt	245
TypeCase	246
While	246
For	247
Find	248
Iterators	248
Assignment	250
Pattern Matching	250
Patterns and Pattern Matching	250
Pattern Categories	251
Pattern Contexts	254
Data Type Operations	255
Boolean	255
Channels	255
Clients	256
Daemons	257
Elements	257
Integers	258
Floats	260
Objects	262
Null	262
Operations	263
Strings	265
Sequences	268
Sets	271
Symbols	273
Tables	273
Threads	274
Vectors	275
Debugging	275
Relationship to OCL and ASL	275
XOCL Grammar	276
Pattern Grammar	279
22. XTools	280
Introduction	280
The XTools Architecture	280
Introduction	280
Tool Component	280
Tool Event	281
Tool Definition	284
Tool Deployment	286
Diagram Tools	286

Introduction	286
Diagram Tool Components	287
Nodes and Edges	288
Toolbar	289
Menus	290
Diagram Events	292
An Example Domain Specific XTool	294
Display Elements	309
Diagram Layout	315
Example Tool: Class Diagrams	315
Form Tools	318
Introduction	318
Form Components	319
Menus on Forms	320
Form Events	320
Example: Airports	321
23. Clients	324
Introduction	324
Introduction	324
Message Based Client	324
Eclipse Implementation	324
XMF Implementation	330
Internal Clients	335
External (Socket Based) Clients	335

List of Tables

17.1.	153
21.1.	255
21.2.	257
21.3.	258
21.4.	258
21.5.	259
21.6.	260
21.7.	261
21.8.	262
21.9.	263
21.10.	264
21.11.	266
21.12.	266
21.13.	268
21.14.	269
21.15.	271
21.16.	272
21.17.	273
21.18.	274
21.19.	275
21.20.	276

Introduction and Tool Overview

Table of Contents

1. Introduction	3
What is XMF-Mosaic?	3
Using XMF-Mosaic	3
Domain Models	4
Concrete Syntax	4
Behaviour	5
Transformations and Mappings	5
Extensibility	5
Why use XMF-Mosaic?	6
Technical Features	6
Architecture	7
Operating System	7
Virtual Machine	7
Kernel	8
XOCL	9
Base Toolset	9
Tool Definition	10
XMF Clients	10

Chapter 1. Introduction

This is the definitive guide to XMF-Mosaic. This guide is divided into 3 parts. This part provides an introduction to XMF-Mosaic and its key technical features. The second part gives an overview of XMF-Mosaic's capabilities through a number of example walkthroughs. The third part provides an in-depth technical manual that covers all aspects of XMF and the languages it is based on.

What is XMF-Mosaic?

XMF-Mosaic is a model-driven development platform. Model-driven development (MDD) is an approach to enterprise system and software development where the key drivers of the process are models. By capturing these artefacts as models, significant advantages can be achieved over traditional programming approaches. MDD has the potential to greatly reduce the cost of development by facilitating the automation of many important development processes, including validation of models and generation of code.

XMF-Mosaic enables the rapid development of model-driven solutions that are tailored to the specific needs of a customer or application domain. These tools provide significant help in automating and managing the development process.

Here are some examples of tools that can be built with XMF-Mosaic:

- Tools for designing and inputting a specific type of product, e.g. an aircraft or a financial product.
- Tools for transforming designs into code, for example, generating code from an aircraft model.
- Tools for validating designs, for example validating the correctness of new financial products.

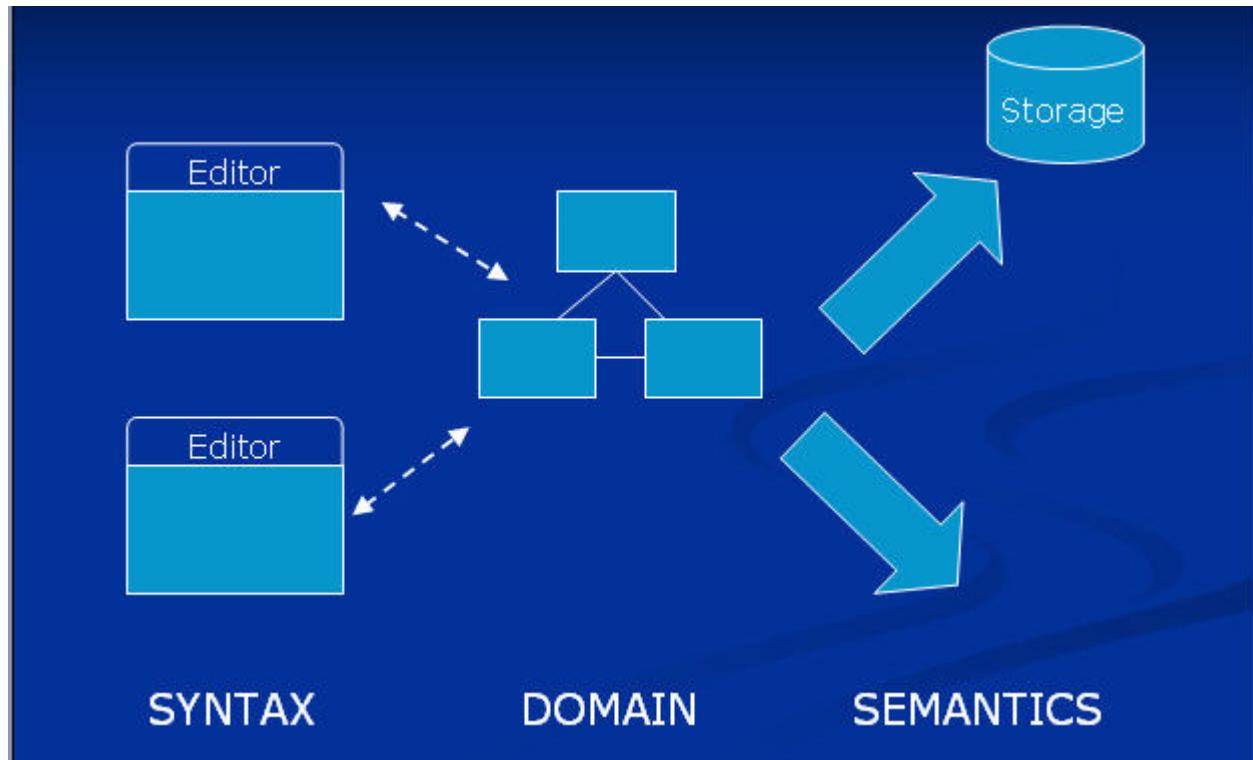
In XMF-Mosaic, you create models that capture the key features of the concepts that are managed by the tool. XMF-Mosaic then provides a rich array of facilities for creating tools based around this model.

A key feature of XMF-Mosaic is that it is completely modelled in itself – in other words, it is used to design itself. Because the user has access to these models, it offers a highly open and extensible platform.

XMF-Mosaic is based on Eclipse (an industrial strength open source IDE) and supports a number of open standards for capturing tool data and behaviour.

Using XMF-Mosaic

XMF-Mosaic provides support for an approach to building tools in which all aspects of the tool is fully modelled. By modelled we mean a 'high-level description'. The following diagram gives an example of the general-architecture of a tool built using XMF-Mosaic:



Domain Models

At the heart of most application specific tools is a domain model. This is a model of the concepts that are manipulated by the tool. For example, a tool for capturing user-interfaces might contain concepts such as 'Button', 'Window' and so on. A domain model can also be viewed as representing the vocabulary of concepts that form the 'language' of user-interface modelling.

Constructing a precise domain model is crucial to building good tools. XMF-Mosaic provides a rich array of tools for this purpose. These allow the concepts and their relationships to be captured using both diagrammatical and textual syntax. In addition, domain models often need to be constrained by what they can represent. For example, we might require that a window cannot contain itself. To support this, XMF-Mosaic provides a rich constraint language based on OCL (the Object Constraint Language) and fully interactive facilities for creating instances of domain models and testing them against their constraints. The constraint language also enables complex queries over models to be conveniently expressed.

Concrete Syntax

While the domain captures the concepts manipulated by the tool, concrete syntax describes the concrete representation of those concepts. There are two main types of concrete syntax:

- Diagrammatical syntax: instances of the domain concepts are represented as diagram elements, e.g. boxes and lines. A diagram editor is used to construct the diagram.
- Textual syntax: instances of the domain concepts are represented as text, which can be parsed and edited in a text editor. This could be programming style syntax or a standard representation such as XML.

XMF-Mosaic provides support for both these styles of syntax.

- It provides a collection of diagram and user-interface languages and tools called XTools. These enable domain specific diagram editors to be rapidly constructed along with customised forms and browsers.

- A generic parser language is provided for building parsers for a wide variety of syntaxes including programming and XML syntaxes.

Behaviour

Most useful tools have executable behaviour (semantics). For instance, a tool for modelling user interfaces may provide rapid prototyping capabilities that enable prototype interfaces to be built and interacted with. A state machine tool might allow state machines to be executed, and so on.

XMF-Mosaic provides a first class programming language for building executable tools, called XOCL - the eXtensible Object Command Language. XOCL:

- Supports all the key object-oriented programming facilities.
- Provides many useful tool programming facilities, including patternmatching and model synchronisation primitives.
- Is fully extensible: new programming constructs can be conveniently embedded in the language.
- Provides full access to all meta-levels in a tool.

XMF-Mosaic enables the semantics of existing languages and tools to be readily reused via specialisation of meta-classes, for example, the semantics of XMF-Mosaic's own tools can be reused and specialised to a specific domain.

Transformations and Mappings

Tools often need to transform and map the data they manage to different representations. This may involve transforming a model into another model, or mapping a model to code written in another language.

XMF-Mosaic provides a variety of mechanisms that support the convenient representations of model to model transformations and model to code mappings. These include:

- XMap: a language for model to model transformations that supports pattern based mapping of input models with output models.
- A powerful text formatting language is provided for flexible pretty-printing of textual code within XOCL programs.

XMF-Mosaic includes a number of transformations out of the box, including the generation of Java from XCore models, and XML serialisation of models. These transformations can be readily modified by the user.

Extensibility

One of the critical features of a good tool platform is the ability to reuse and extend its existing capabilities.

XMF-Mosaic provides a number of approaches to achieving this

- By extending existing meta-classes: Because all aspects of XMF-Mosaic are defined as instances of a meta-data model (XCore), they can be extended through class specialisation. Furthermore, XMF-Mosaic understands when this extension has taken place and automatically provides new tool buttons etc to support the extension.
- By extending existing concrete syntax definition

XMF also provides support for extensibility at the programming level. XOCL provides facilities for extending its syntax. This enables complex patterns of code to be conveniently wrapped up into simple programming statements. Furthermore, new programming languages can be rapidly added by extending existing languages.

Why use XMF-Mosaic?

XMF-Mosaic provides a convenient and fully integrated solution for developers who wish to have complete control over the tools they use in their development processes. In comparison with building tools in general-purpose languages like Java, XMF-Mosaic offers significant reduction in construction effort, better extensibility, and a more intuitive and integrated solution to tool construction. This also applies to IDE's like Eclipse. XMF-Mosaic provides a next level of abstraction over typical IDE facilities. For example, XMF-Mosaic provides domain specific programming languages for building user-interfaces, which are significantly more productive than programming the interfaces from scratch even in an IDE.

There are some strong business cases for developing tool solutions that precisely fit your development processes. Here are some:

- Tools that specifically target a problem domain significantly increase productivity. They achieve this by reducing the design space, enabling developers to focus on what they are developing, rather than how to express the same information in a general purpose language.
- Code and documentation generators are easier to write as the source of the generators is a precisely targetted domain model.
- Creating a domain model is a benefit in its own right as it gives a deeper understanding of the concepts that the business works with.
- Automation becomes a key part of the development processes. Once instilled, this process can lead to large costs savings across many parts of an organisation.

Technical Features

XMF-Mosaic is a rich and powerful modelling tool, which supports all the following capabilities:

- **Domain modelling:** create models of domain concepts using a rich MOF based visual modelling tool.
- **Constraint checking:** write complex constraints and querys and test them on the fly.
- **Model execution:** write complex operations that manipulate models using XOCL - the eXtensible Object Command Language - and test them on the fly.
- **Domain specific modelling:** rapidly create diagram editors, browsers and form editors for specific modelling languages
- **Domain specific programming:** create parsers for new languages and rapidly link them to a domain model.
- **Model to model transformations:** build model to code and model to model transformations.
- **Extensibility:** new languages and tools can be created by extending and reusing existing language and tool definitions.
- **Document generation:** documentation can be generated for all aspects of a tool.
- **Consistent definition:** all aspects of XMF-Mosaic are designed using itself. This facilitates its superior extensibility. New capabilities are added simply by designing them in the tool using high level tool modelling capabilities.
- **Key meta-data standards.** This means that the toolset accessible to developers and is very open. Customers do not worry that their data is going to be 'locked' into the tool.
- **Fully interactive execution:** no compilation phase is required to run XMF-Mosaic models: interactively test tools and models before deploying them to code.

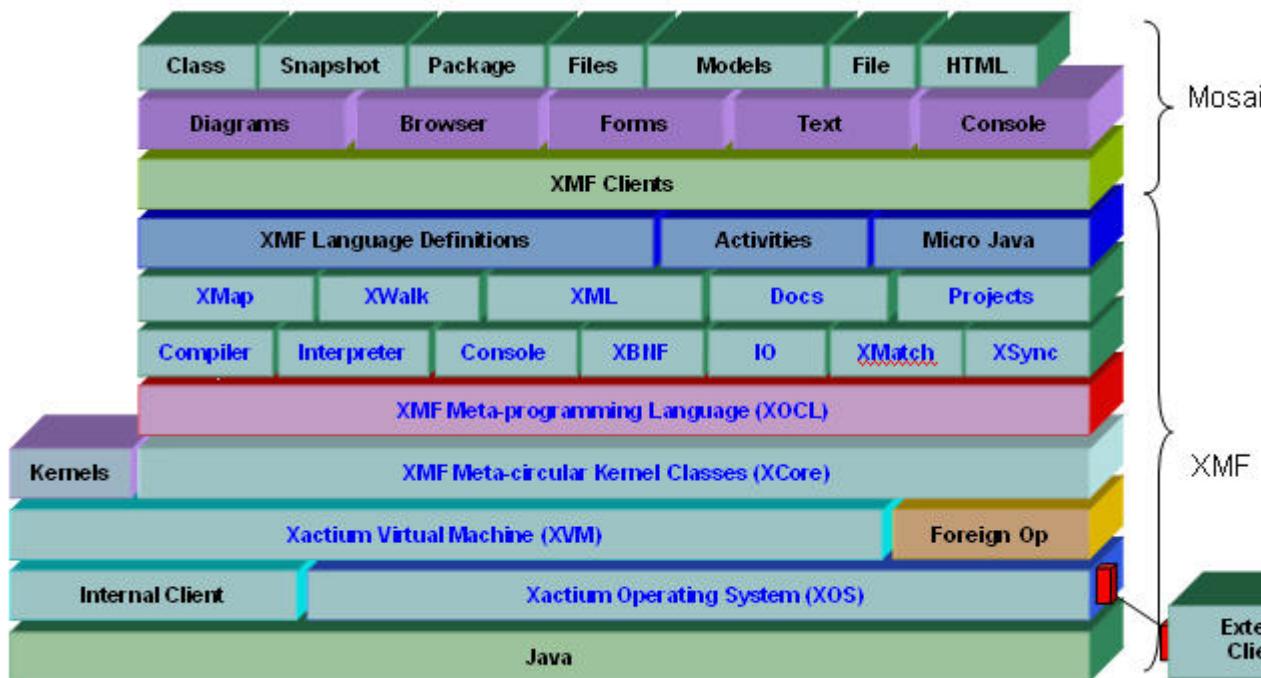
Over time, XMF-Mosaic will be supplied with an increasing array of development capabilities that address both domain specific and general-purpose software development requirements.

Architecture

This section provides a *detailed technical description* of the main components of XMF-Mosaic.

XMF-Mosaic is a layered open modular system that is specifically designed to support the definition a rich array of tool development capabilities. The Mosaic layer provides general-purpose graphical interfaces on top of XMF that support features such as diagram and property editors.

The following diagram is an overview of the architecture of XMF Mosaic V 1.0. Each block is a system component that relies on the components below it. Everything above the Virtual Machine is written in XMF-Mosaic in the form of models. The system is open in the sense that users can browse, modify and extend the system models. XMF can be used as a server where external clients communicate via sockets.



Operating System

The Ceteva Operating System (XOS) provides services that allow XMF to connect to the outside world, to interface to code written in Java and to schedule XMF threads. XOS implements a collection of channel types that are used for input and output including file channels, compressed data channels and XML channels. XOS acts as a server for external clients that connect to XMF using sockets. Inside XMF the connection provides input and output data channels. An external client is any third party software component that can connect using a socket; communication with the client may be synchronous or asynchronous. Internal clients are written in Java and are registered with XOS; they run in the same process space as XMF and communicate with XMF using channels. XOS is written in Java.

Virtual Machine

XMF source code is compiled to a binary format that is performed on the Virtual Machine (XVM). XVM is an object-oriented machine whose instruction set is designed to efficiently support meta-circular language definitions based on XCore. XVM has a rich data set including integers, floats, collections, channels and objects. XVM is multi-threaded and allows multiple threads to run

concurrently. XMF source code can be compiled directly into the XVM heap during a session or can be written to a binary file (.o) and loaded into a subsequent XMF session. XVM supports a number of novel features that supports the definition of XMF tools including meta-object-protocols and object daemons. XVM is also designed to interface to external programming languages, such as Java, through “foreign” operations.

Kernel

XMF provides a collection of classes that form the basis of all XMF-Mosaic defined tools. These classes form the kernel of XMF and are collectively called XCore. The classes are self-describing: all XCore classes are instances of XCore classes. This feature is called meta-circularity and is the key to modularity, uniformity and reusability throughout all system and user defined XMF tools. XCore is based on the MOF (Meta-Object Facility, a meta-data standard developed by the Object Management Group).

XCore includes class definitions for the basic types including Integer, Boolean and String and collection types for sets and sequences of values. XCore is object-oriented, it provides basic notions of Object and Class. XCore is executable; it provides definitions for executable (Performable) entities and Operations.

All tools built using XMF-Mosaic are instances of XCore; therefore tools that are defined to work on instances of XCore can be used on any tool data. For example, general-purpose editors and mappings are provided by XMF that are guaranteed to work across all system and user defined data.

Key classes in XCore are:

- **Class.** XMF-Mosaic is a class-based system. Tools are defined as collections of classes whose instances have state and behaviour. The class Class defines the essential features of a class. Inheritance is used to extend class features in XMF. Since Class is available; user defined tools can extend what it means to be a class. This ability is the basis for meta-programming. For example, Class may be extended with the ability to keep track of all instances or to access instance data from an external database.
- **Daemon.** Daemons monitor the state of objects and perform actions when the object changes state. Daemon technology is the key to implementing a variety of modular reusable tool architectures such as the observer pattern. XMF-Mosaic uses daemons extensively to synchronize data across multiple tools. User defined tools can use daemons to make tools reactive and to ensure data is always consistent.
- **DataType.** Instances of this class are XMF types for basic data values. XMF types include Integer, String, Boolean and Float.
- **Element.** All classes in XMF are extensions of the class Element. Element defines the essential behaviour of all XMF data. For example Element defines features such as being able to produce a printed representation and the ability to handle messages. XMF is a dynamically extensible system; this means that new behaviour can be added to existing classes. This is sometimes referred to as aspect oriented programming. Since Element is available, user defined tools can add system-wide aspects. For example this can be used to add the ability to export any XMF data in any required format (binary encoded, XML, text etc).
- **Exception.** XMF provides exception handling for dealing with exceptional circumstances in running code. The class Exception is the basis for a hierarchy of classes that implement specific types of errors. Exceptions are raised at the point at which they occur and encapsulate data that describes exactly the source of the problem.
- **Operation.** The basis for all XMF execution is the class Operation. An operation has parameters and a body and is equivalent to a standard programming language procedure or function. A significant difference to conventional procedures is that operations are XMF objects that can be created and stored just like any other object. This makes XMF very flexible since behaviour can be encapsulated at the appropriate point in models and data.

- **Package.** XMF supports name spaces that contain collections of named elements. The class Package is used to structure collections of class and sub-package definitions. XMF-Mosaic is structured as a tree of packages containing definitions of all aspects of the system (including XCore). The root name space is called Root; all XMF classes can be referenced via Root.
- **Performable.** XMF provides an environment in which executable languages can be conveniently developed. An executable language implements the interface defined by the abstract class Performable. XOCL is an example of a language that implements this interface.

XCore is an example of a language that can be executed on the Virtual Machine. The Virtual Machine may be initialised with different kernel language definitions although in practice it relies on a small sub-set of XCore being present. This feature allows XMF to deploy embedded systems that run application code without requiring the tools that were used in XMF-Mosaic to develop the application.

XOCL

XMF provides an extensive language for constructing tools; the language is built from XCore and runs on the the Virtual Machine. The language, called XOCL, or eXtensible Object Command Language provides a first class language for manipulating XCore objects.

XOCL provides a large number of language features that address the implementation of real-world scalable tools (the whole of XMF-Mosaic is written in about 100KLOC XOCL). These include side-effects, object creation, exception handling, multi-tasking, pattern matching, first-class types, first-class operations, efficient looping constructs, input output, client-server support, daemon mechanisms and support for dealing with syntax constructs.

XOCL also supports many features of OCL (the Object Constraint Language, mandated by the Object Management Group), and therefore is an excellent foundation for querying models and writing constraints over models.

Base Toolset

XMF provides a basic collection of languages and tools defined in XOCL. These include the following:

- The **XOCL compiler**. XOCL is an example of a language that is constructed on top of XCore and running on the XVM. The XOCL compiler is a fully bootstrapped parser and language translator that converts text (or instances of the XOCL classes) into sequences of XVM instructions. The compiler can be invoked on files, strings or instances of the XOCL model. The compiler is extensible – new language constructs can be introduced that define how to translate themselves to instances of the XVM instruction model
- XMF provides a BNF-style parser language called **XBNF** for defining new textual languages. XCore allows grammars to be defined for classes and XOCL allows language constructs defined by any newly defined grammar to be integrated with any existing language construct. This makes XOCL an infinitely extensible programming language. XOCL takes the form of a grammar for a basic language and then many extra language features are added to XOCL (including fancy looping mechanisms, various definitional constructs and case statements) in terms of separately defined grammars. The extension mechanism is very convenient; once the grammar is defined and loaded, the new language feature is ready to use. Language features can implement new commands or appropriate ways of defining new data constructs (and mixtures of the two).
- XOCL does not need to be compiled in order to run. XMF provides an **XOCL interpreter**. The XOCL class Performable defines an interface for any language construct that is to be executed, including interpreted. Interpretation is often more convenient than compilation since no language translation is involved.
- XMF provides a top-level **console** based command interpreter that can be used as the interactive interface to an XMF or XMF-Mosaic session. The command interpreter reads any valid XOCL syntax typed at the console, evaluates it and then prints the result. The user can access any data in the

XMF system via the console, inspect it and modify it. The command interpreter provides a collection of convenient top-level commands that can be extended to provide productivity accelerators.

- XOCL provides powerful inbuilt support for pattern matching using **XMatch**. XMatch enables patterns of expressions to be matched across XOCL statements thereby facilitating the declarative definition of mappings, constructors, etc.
- The **XSync** language provides a high-level way of synchronising data, where changes in one element cause changes to be automatically propagated to other elements.
- **XMap** is a language that is used to write model-to-model transformations. In XMap, patterns are used to describe how objects of a specific type and structure are mapped to objects of another type and structure. XMap patterns can incorporate arbitrary XOCL, enabling complex mappings to be implemented with ease. XMap is based on the emerging QVT (Query, Views, Transformations) standard.
- XMF provides facilities for parsing and generating **XML** documents. High-level grammar rules can be written which state how a specific XML element pattern can be mapped to a XCore element or trigger the invocation a XOCL action. These rules can be used to generate a parser for a specific XML syntax.
- **XWalk** is an extension to XOCL that provides facilities for efficiently running over large XCore object structures and evaluating its properties, for example running constraints or modifying data.
- In XMF, **documentation** is captured using a special language extension. This enables documentation to be processed and managed at the model level, facilitating flexible production of documentation in XMF.
- **Projects** provide a resource for managing models and for deploying them to code and to files. Projects can be created in the browser, and can be saved and loaded using the XMF (.xar) serialisation format.

Tool Definition

The base toolset provides a powerful collection of tools for the rapid construction of new development tools. Everything from small tools that analyse data to full scale modelling and/or programming language environments can be defined in an open, flexible and transparent way. Currently, the base version of XMF is distributed with some languages built in, including:

- A Java based language called MicroJava, which implements the key features of Java (including statements and expressions). A mapping from XOCL is provided to MicroJava. MicroJava knows how to pretty-print itself to executable code.

XMF Clients

XMF can be connected to a wide variety of external clients (using socket channels) and internal clients (via the XOS). External clients can be any software that supports a well-defined API, for instance other tools or IDE environments. The Mosaic clients are pre-defined internal clients that provide common tool based user interface clients for the languages defined in XMF such as diagram editors, browsers and so on. Each of these clients has two aspects: an XMF client model of the abstractions and functionality provided by client and an external client implementation of the rendering of these abstractions. Because the clients are modelled in a generic fashion, independently of the rendering technology, clients are extremely flexible and adaptable. As a result, new user interface clients can be constructed very rapidly.



Some of the client models that are supported by Mosaic include:

- Diagrams: provides general diagramming abstractions.
- Browser: provides general facilities for creating browsers and attaching menus, etc.
- Forms: provide general form based abstractions.
- Text: provides text editing, generic syntax highlighting and manipulation functions and html functionality.
- Console provides input and output streams to a console.

These clients have been used to construct all the user interface tools provided with XMF, including the following:

- Class diagrams for visualising XCore models
- Mapping diagrams (as an extension to Class diagrams)
- Model browsers
- Snapshot diagrams
- The file browser

The implementations of the clients have been constructed using Eclipse: a generic, open source IDE platform.



XMF Walkthroughs

Table of Contents

2. Creating and Interacting with a Domain Model	16
Example	16
Getting Started	16
Constructing a Domain Model	18
Adding Classes	18
Adding Attributes	21
Adding Associations	23
Using the Property Editor	24
Saving and Loading	25
Interacting with Models	26
Snapshots	26
Creating Standalone Snapshots	29
Dealing with Sequences	30
Synchronisation between Snapshots and Class Models	31
Creating Multiple Views of Snapshots	32
Exporting Snapshots	32
Saving and Loading Diagram Layout	33
Adding Constraints	33
Constraint Reasons	35
Parse Errors	35
Checking Snapshots	36
Exercises	38
Adding Queries	38
Exercises	40
Running Operations	40
Running Operations on Snapshots	40
Operations	42
Exercises	43
Adding Getters and Setters	43
Exercises	44
Adding Constructors	44
Exercises	45
Interacting with Models using the Console	45
Other Console Hints and Tips	46
Editing Values via the Console	47
Using Global Variables	48
Error Reports	48
Using Snapshots	49
Generating Snapshots from the Console	49
Adding Package Operations	49
Adding and Running Package Operations	49
Next Steps	50
3. Working with the Modelling Interface	51
Windows	51
Displaying Multiple Diagrams and Windows	51
Preferences	53
Property Editor	53
4. An Introduction to XOCL	55
Introduction	55
Basic Types	55
Examples	55
Models	55
Using XOCL	57
Context	57
Self	58

Operations	58
Constructors	59
Constraints	59
Variables	60
Types	60
Navigation	61
Collections	61
Collection Operations	62
Logical Expressions	63
Conditional Expressions	63
Imperative Features	64
Object Creation	65
Assignment	65
Sequential Execution	65
Operation Invocation	66
Looping	66
Exceptions	66
Formatting	67
Advanced Features	68
5. Debugging Operations	69
6. Constructing and Running Mappings	70
A Simple Mapping	70
Running the Mapping	73
A Simple Model to C# Mapping	74
The C# Model	74
The Mapping	75
Executing the Mapping	76
Mapping Hints and Tips	77
Error Reporting	77
Constructing a Pretty Printer	78
7. Constructing an XML Parser and Generator	81
First Steps	81
Constructing the Grammar	81
Invoking the Parser	82
Example	83
Debugging the Parser	83
Generating XML	85
8. Using the Programming Interface	88
Getting Started	88
Compiling and Loading	90
Checking the Model	90
Adding Constraints	90
Adding Operations	90
Context	91
Importing Packages	91
9. Toolbar Menus and Initialisation Files	92
Toolbar menus	92
Some Useful Operations	92
Initialisation Files	93
10. Constructing a Diagram Tool for a Model in XTools	95
Domain Model	95
A Candidate Diagram Syntax	95
Constructing an XTool Definition	96
Creating the Tool Definition	96
Adding a Node	96
Adding a Box to the Node	97
Adding an Attribute to the Box	98
Adding a Free Node	99

Adding Tool Bars	101
Running the Tool	101
Adding Edges	102
Adding Edge Toolbar Buttons	104
Adding Menus	104
Re-running the tool	104
Other XTool Capabilities	105
11. Importing XMI	106
12. Constructing a Textual Syntax and Parser	107
Parsing and Synthesising Instances of Models	107
13. Creating a Meta-Profile	109
An Example Profile	109
Adding Constraints	114
14. Generating Code	116
Generating Java	116
15. Using Manifests and Deploying Models	118
Manifest Actions	118
Deploying Manifests	118

Chapter 2. Creating and Interacting with a Domain Model

One of the key steps in building a tool to support a specific application domain, is to precisely capture the domain concepts that the tool will manipulate.

To do this, we need to construct a *domain model*. A domain model captures the key concepts in the domain, along with their properties and relationships. For example, if the domain is business processes, then we would aim to capture all the concepts of a business process (activities, flow, constraints, etc) in the model.

This part describes how to *model* the concepts in a domain. It describes how to capture the domain concepts as class models, and how the model can be enriched with constraints, operations and constructors. It then shows how the powerful dynamic capabilities of XMF-Mosaic can be used to validate the domain model, through the creation of model instances and by executing the model.

Example

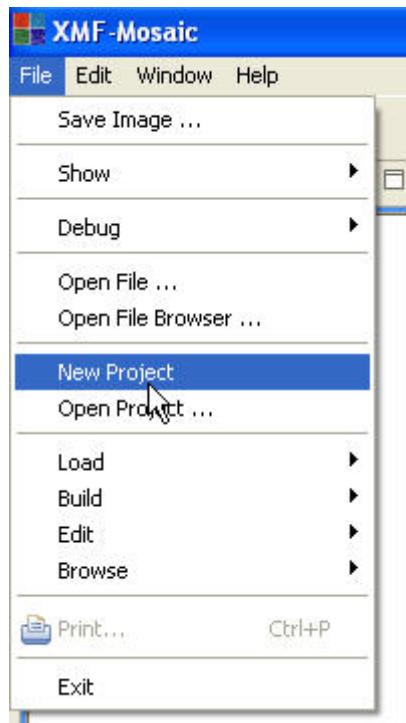
The example we are going to use is a simple component modelling domain. This domain deals with capturing information about the components in a system, their interfaces (ports) and their connections to one another (connectors).

Getting Started

Boot up XMF-Mosaic.

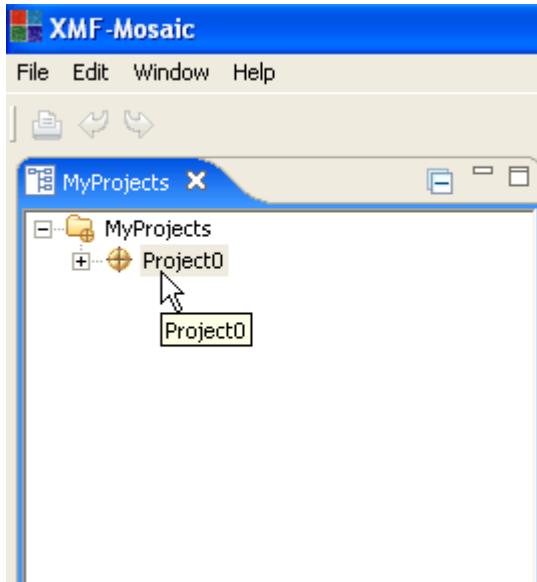
The first step in creating a model is to create a project.

From the file menu, select New Project.

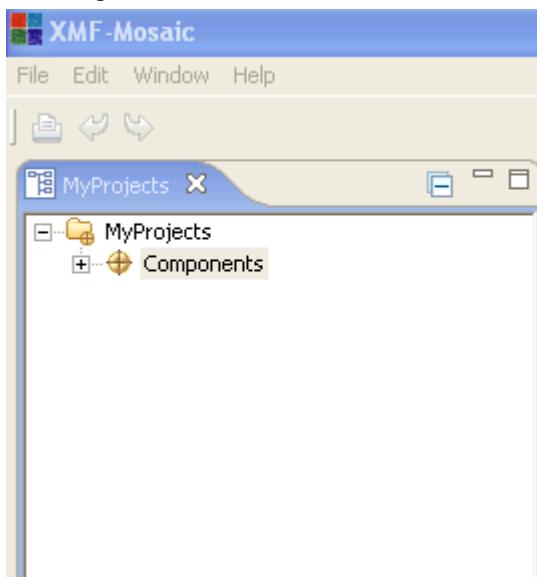


Alternatively, the same operation can be called by right clicking on the User project in the browser.

A new empty project with a default name appears in the MyProjects project tree.

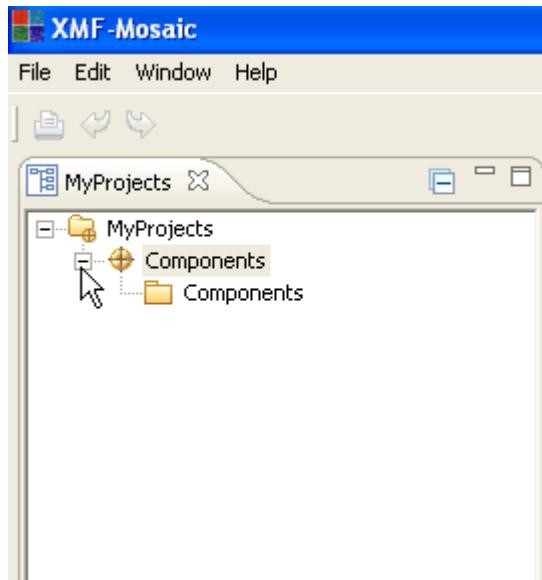


We want the name of the project to reflect what it contains, so the project name is selected and changed to “Components”.



By default, a project contains a *package* - this is where the model will reside.

The package can be viewed by expanding the (Component) project icon by clicking on the cross. By default it has the same name as the project.

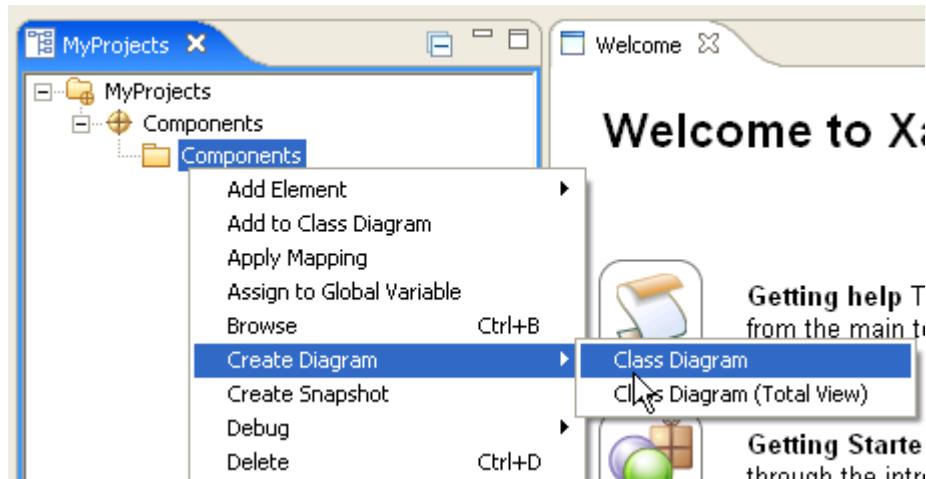


Constructing a Domain Model

We are now going to create a domain model of the concepts in the component modelling domain. We will do this using a class model. This model will focus primarily on describing the concepts in the domain.

The model can be constructed as follows:

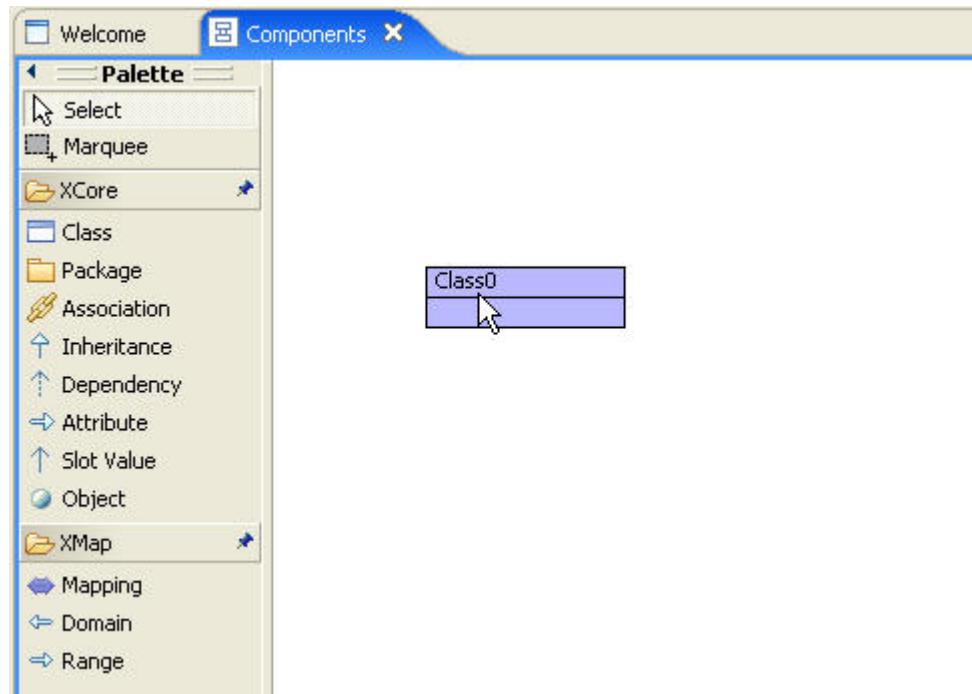
First, right click on the Components package in the browser and select Create Diagram > Class Diagram.



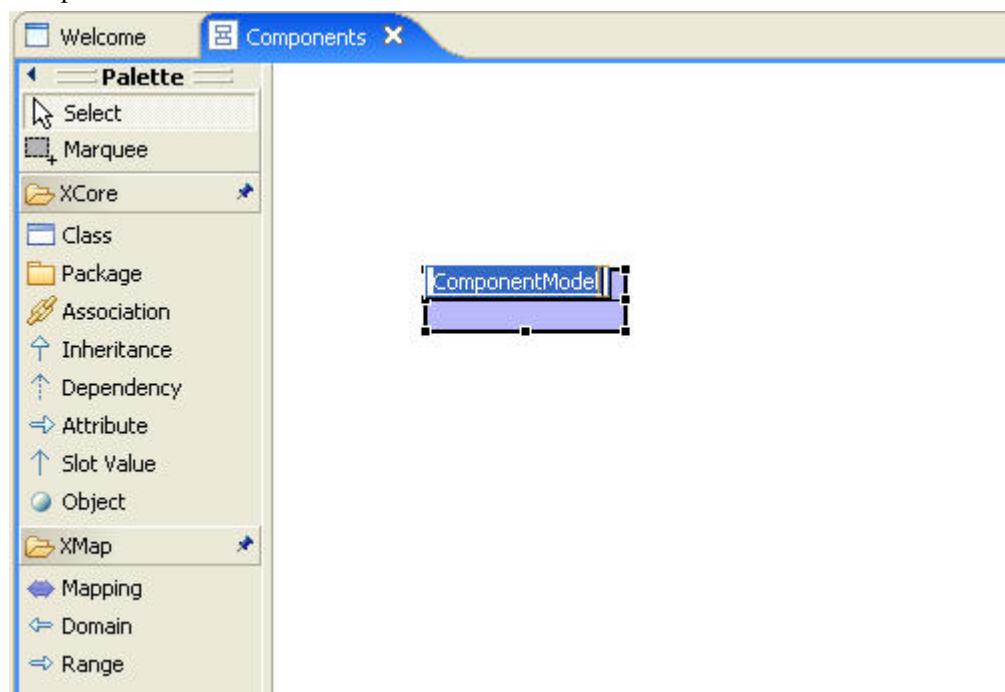
A class diagram editor will be shown.

Adding Classes

In order to add a class to the model select the Class button and click on the diagram. The newly added class is automatically given a default name.



Click on the class name to change or amend it. In this case, we will change the name of Class0 to ComponentModel.



The properties of the class can be edited by right clicking on the class and selecting Edit.

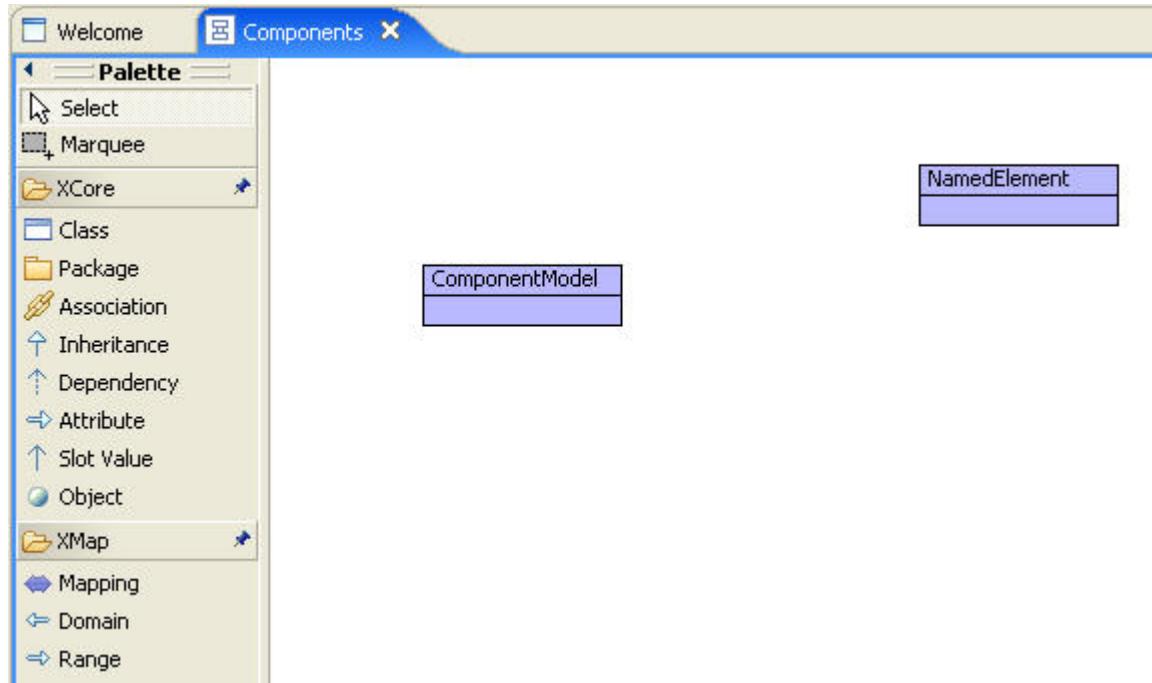


The property editor tells us that:

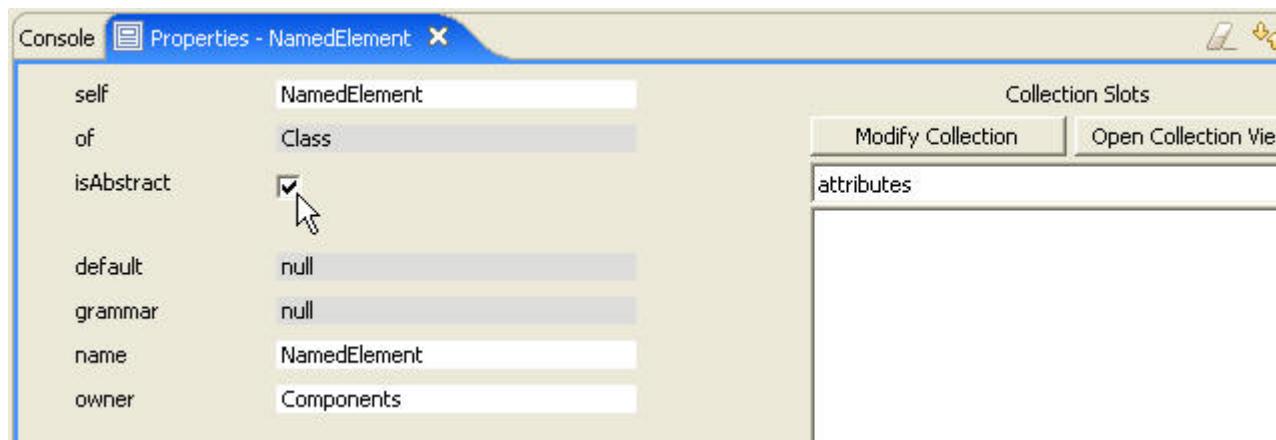
- The element being edited (self) is ComponentModel.
- It is an instance of the class Class.
- It has no attributes (see right hand panel).
- Its name is ComponentModel.
- It is owned by the Components package.

We can ignore the grammar and default properties for now.

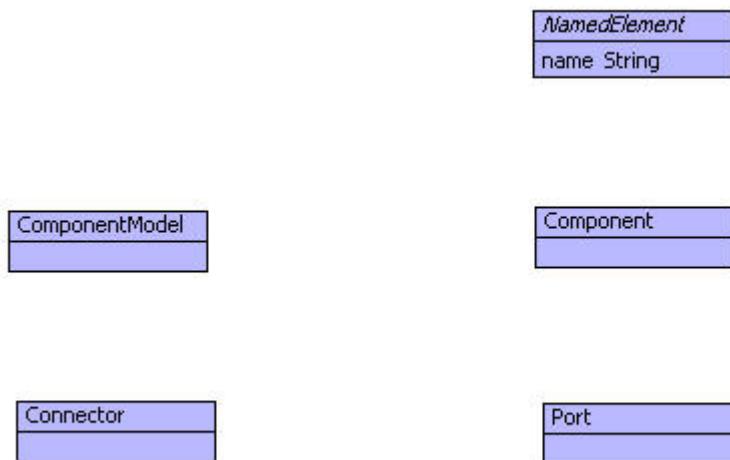
Let's add another class. This time a class called NamedElement.



We want to make this abstract (i.e. non-instantiable). To do this, right click on the class and select Edit. In the class's property editor tick the isAbstract box. The name of the class will become italicised to indicate that it is now an abstract class.



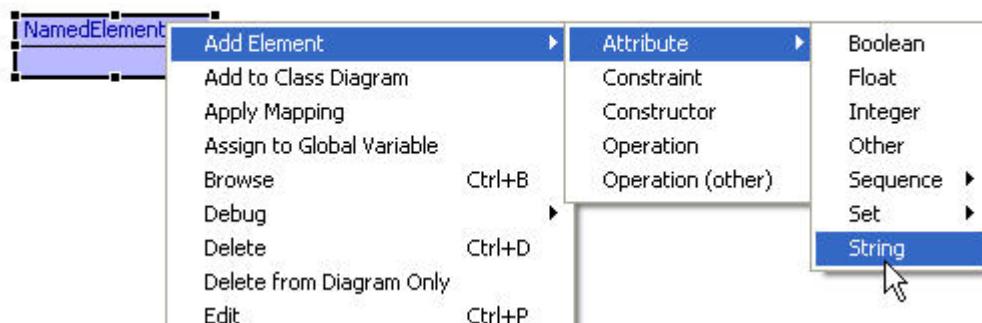
Let's add some more domain concepts to the model.



Adding Attributes

Attributes are variables owned by a class. Attributes are used to represent properties of a class. They have a name and a type.

The class NamedElement requires an attribute 'name' of type String. This is added by left clicking on the selected class. From the Add Element menu select an Attribute of type String.



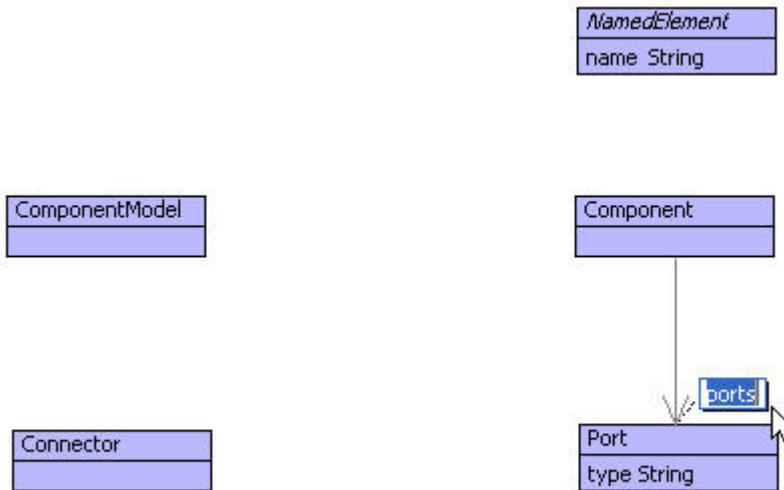
Double click on the attribute name in the class box and change it to 'name'.

Attributes can also have a class as their type. They are represented as named arrows connecting the owner of the attribute to the class that is its type. Note, this use of attributes are similar to uni-directional associations in UML.

The relationship between a Component and a Port can be described by an attribute "ports" whose owner is a Component, and whose type is a Port.

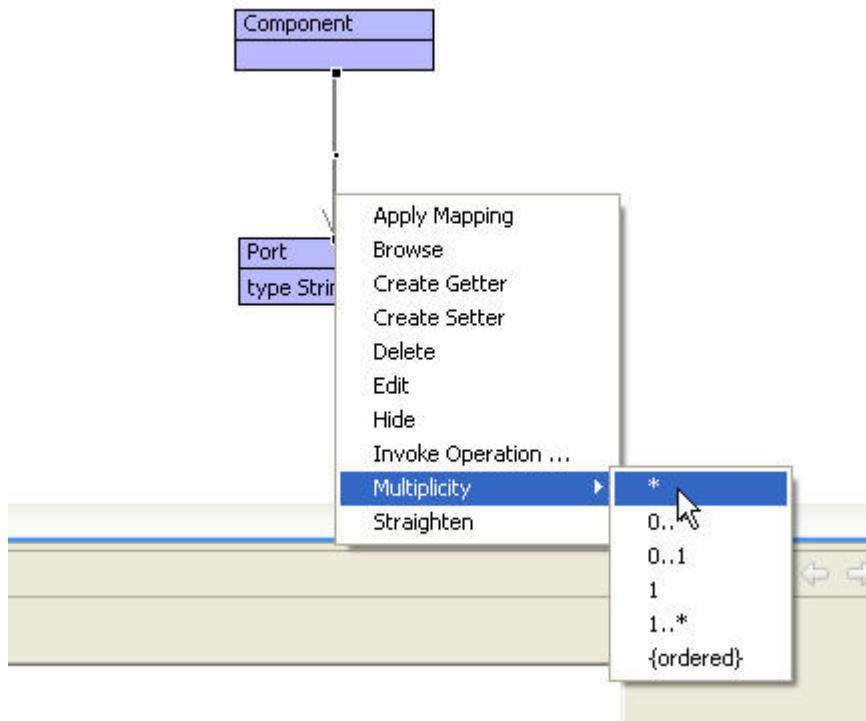
Let's add this attribute to the model. First select the Attribute button in the toolbar. Select the Component class and then drag the edge to the Port class. Edit the name of the attribute by clicking on it. In this case, we are going to change it to "ports".

Note, the name of the attribute can be moved by left clicking on it and dragging it with the mouse.



The attribute edge can be moved by clicking on it, and moving the cursor over the waypoint. A drag icon will appear. Click and drag the edge to the required position. This can also be used to add waypoints to the line.

A default multiplicity of **one** is associated with the newly added attribute. Change the multiplicity by right clicking on the attribute line and selecting the appropriate multiplicity. In this case a Component is associated with **many** Ports, so select ***** (many):



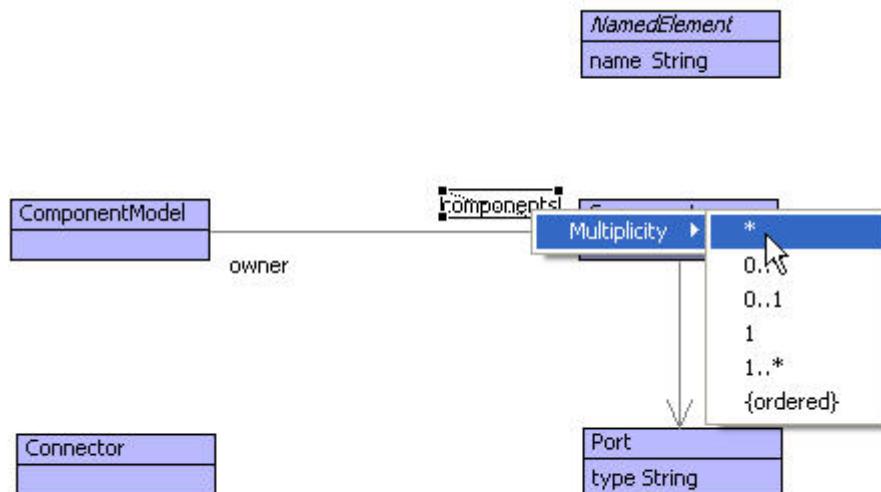
Adding Associations

Associations represent bi-directional relationships between two classes. In XMF-Mosaic, an association is equivalent to a pair of attributes owned by two classes and a constraint which guarantees that there is a round-trip relationship between the attributes.

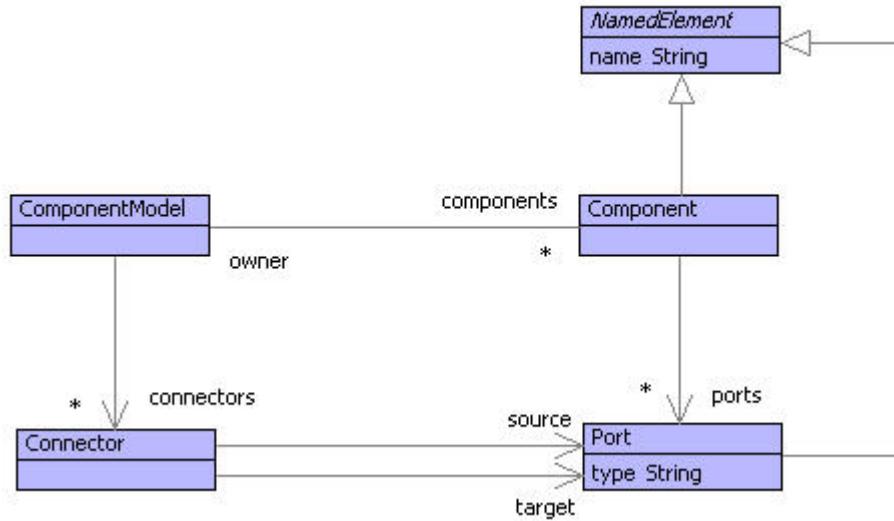
As an example, an association is required between the classes ComponentModel and Component.

The relationship we want to capture is that a ComponentModel contains many Components, and a Component is owned by a ComponentModel.

To create an Association, select the Association tool button and drag the edge from the ComponentModel to the Component. Change the names of the association ends as shown below. The multiplicity of each association end is changed by **right clicking on its role name** and selecting Multiplicity.



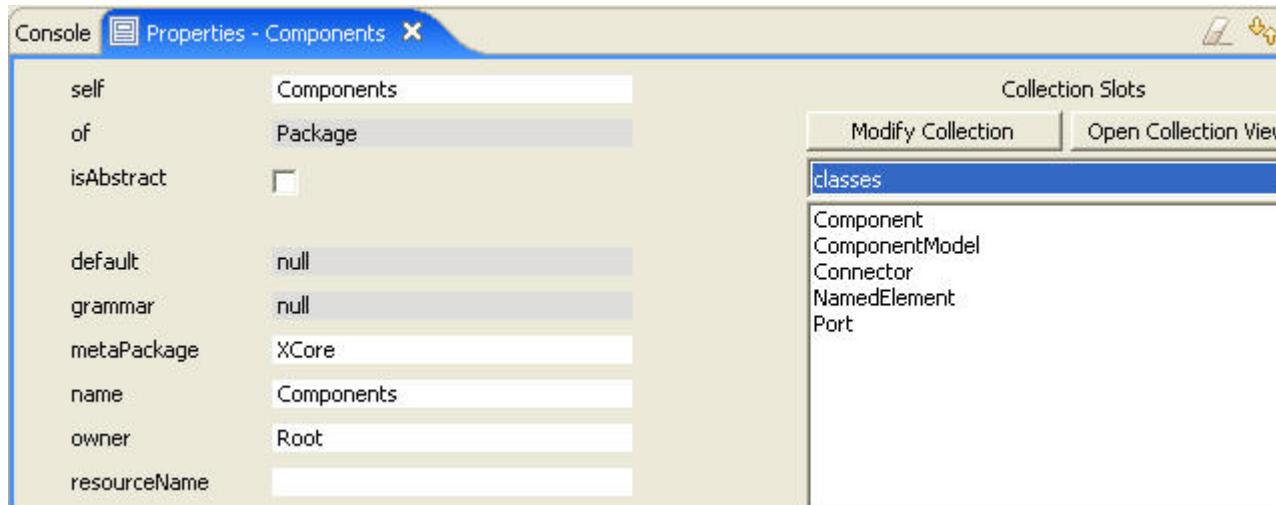
A more comprehensive Components model can now be constructed. Here further attributes have been added, along with two generalizations. The latter ensure that the Component and Port classes inherit the name attribute from the class NamedElement.



Note: right click on an edge and select Straighten to straighten an edge.

Using the Property Editor

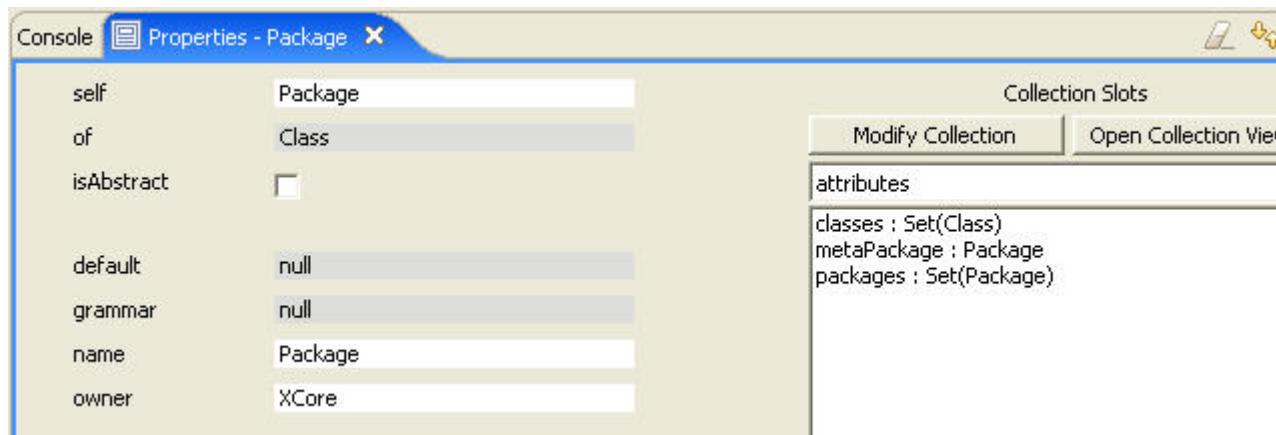
The property editor provides some useful features for viewing and managing the properties of model elements. Right clicking and selecting Edit on an element will display a property editor for that element. For example, editing the package Components (in the browser) will show the properties of the package:



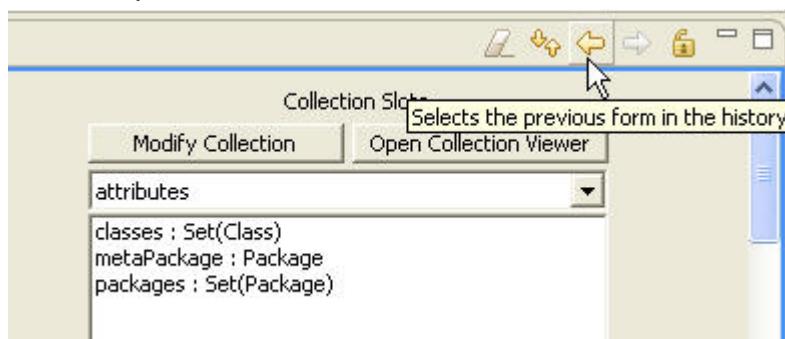
Single value properties (slots) are displayed on the left hand of the property editor. For instance, the package **Components** has a **name** slot whose value is "Component" and an owned slot whose value is the "Root" package. The "of" slot is the class that the element is an instance "of", in this case, a Package.

Double clicking on the value of a slot will display a property editor for that element, for example clicking on the **Component** class in the classes list will show the property editor for the class **Component**.

One of the powerful features of XMF-Mosaic is its uniform representation of its own implementation. To see this, double click on the "of" slot. This will display a property editor for the class Package - the class that defines what a package is.



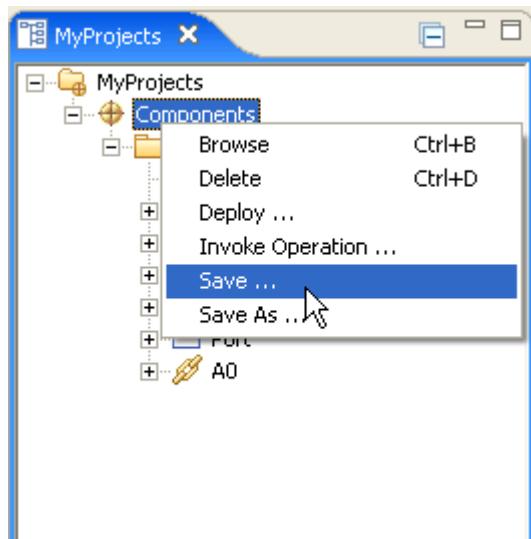
When navigating through the properties of large models, it is often useful to be able to quickly flick back to previous property editors. Rather than displaying hundreds of tabs, XMF-Mosaic provides buttons for managing property editor histories. These can be found in the top right hand bar of the property editor. The left and right hand arrows allow you to move between previous and next editors in the history.



A list of all the editors can be seen by clicking on the up and down arrows. A history can be deleted by clicking on the eraser icon. Finally, property editors can be kept as tabs by clicking on the lock icon - when a new property editor is shown, it will be displayed in a new tab. This is useful when comparing two or more properties.

Saving and Loading

Save the project by right clicking on the project in the browser and selecting Save As or Save. Files are saved in a .xar file.



Saving the project for the first time requires that it is given a name, for example Components.xar.

Subsequent saves will save the project under the same name. Save As can be used to save the project under a different name.

A backup (.xar.bak) file is also be saved.

Projects can be loaded by selecting Open Project ... from the File menu. Choose the project to load from the file chooser.

Interacting with Models

A key difference between XMF-Mosaic and other modelling tools is that its models are fully interactive. In XMF-Mosaic you can create and visualise instances of models in a variety of different ways, and then write operations, transformations, etc. that do things with the models. The ability to interact with models is essential when capturing and validating detailed properties of a complex problem domain. In addition, because XMF-Mosaic is itself completely modelled, all the properties of the tool can be accessed in exactly the same way, resulting in a consistent and highly interoperable approach to tool definition.

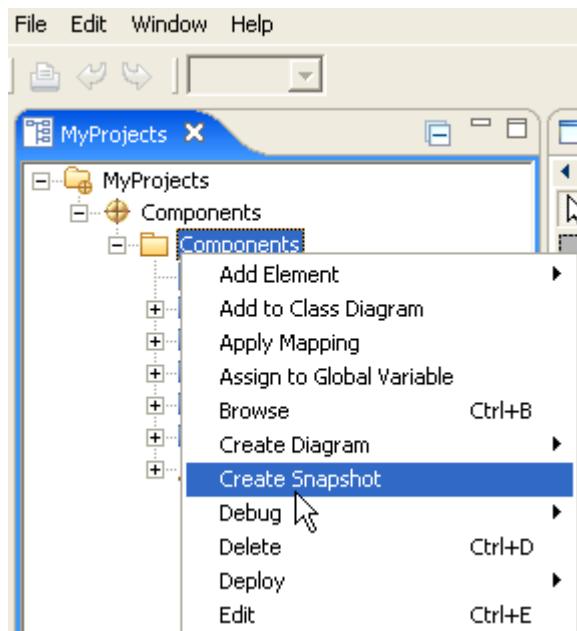
The following sections describe these facilities in detail.

Snapshots

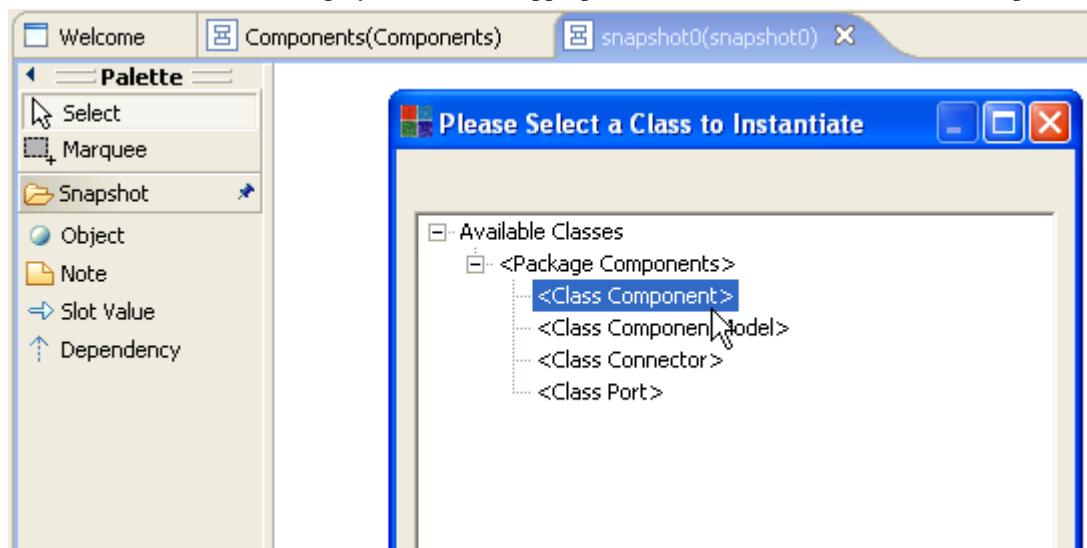
Snapshots provide a visual way of interacting with instances of a model. A snapshot is a diagram that contains instances of classes in package.

Note, snapshots are intended to provide a simple visual way of viewing instances, but they can become difficult to read if they are applied to large numbers of objects.

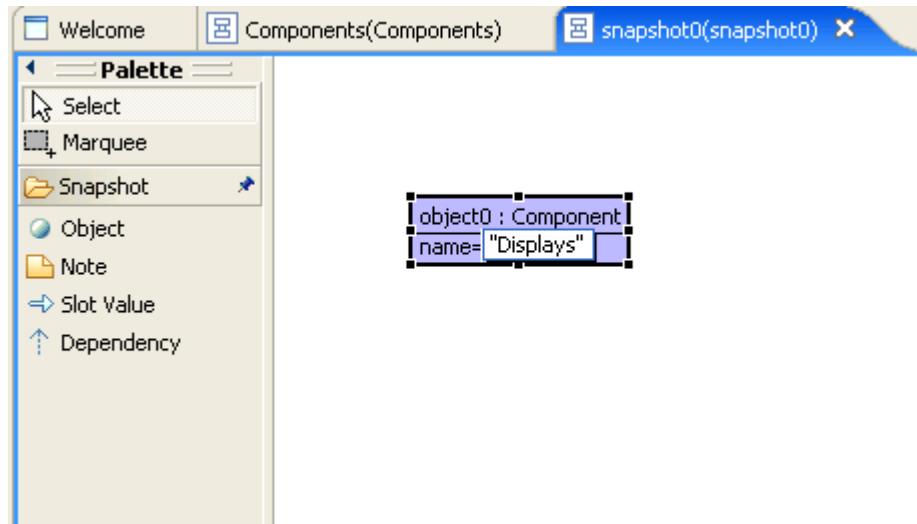
To create a snapshot of a specific package right click on a package in the model browser and select Create Snapshot. Instances of the classes in the package (or its sub-packages) can now be created.



Add objects by selecting the Object button and then clicking onto the diagram. The list of object types that can be created will be displayed. Select the appropriate one and it will be added to the snapshot.

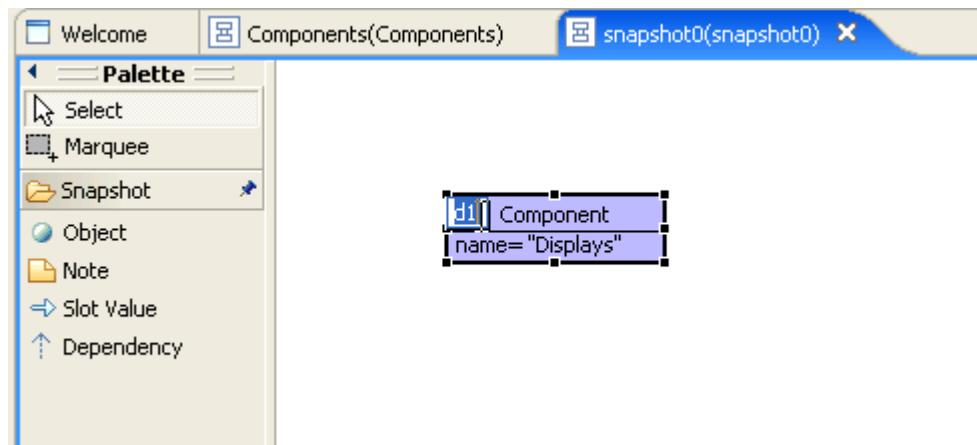


For example, a Component can be added to the diagram.

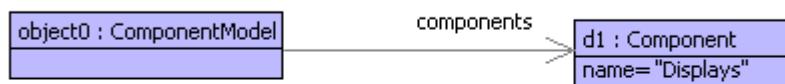


The value of primitive slot (attribute) values can be changed by double clicking on the values in the object.

It is also possible to change the name of the object by clicking on it.

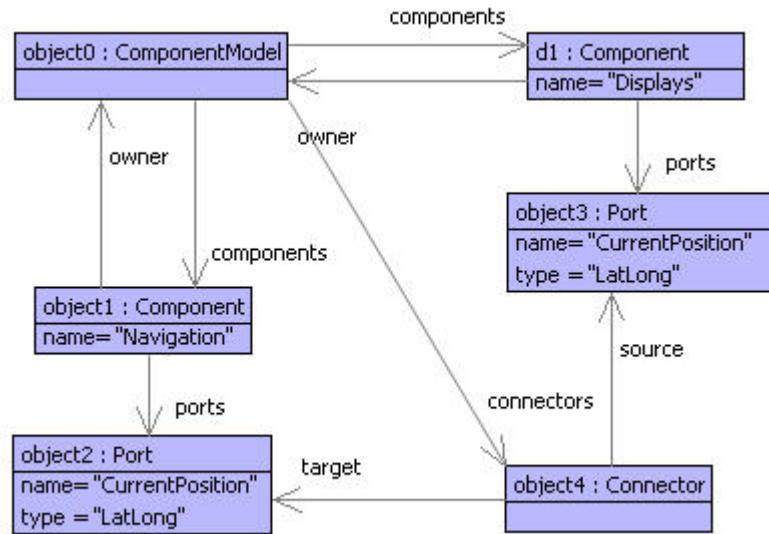


Links can be created between objects by selecting the Slot Value button and connecting the owning object to the target object. As an example, add an instance of the class ComponentModel to the snapshot and create a link between the ComponentModel instance and the Component instance.

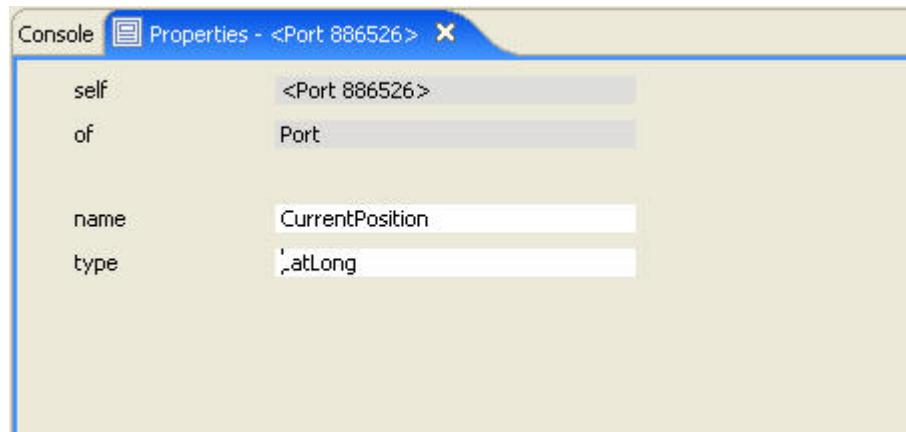


Slot names will be assigned automatically where possible (as shown here), but a choice of slot names will be provided if there is more than one possible attribute that the slot can be an instance of.

A complete snapshot of the model can be created as illustrated below. This represents a simple Component model, in which two components (Displays and Navigation) are connected together. The Displays component expects to receive the current position from the Navigation component.



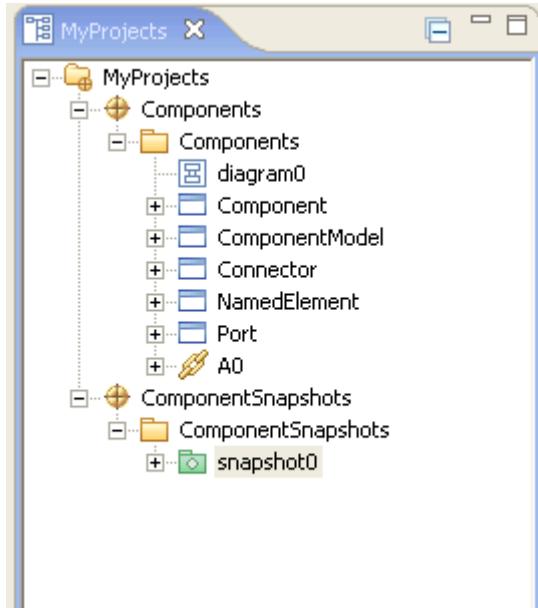
Just like other model elements in XMF-Mosaic, objects can be edited (using right click > Edit). Here is the result of editing object2.



Creating Standalone Snapshots

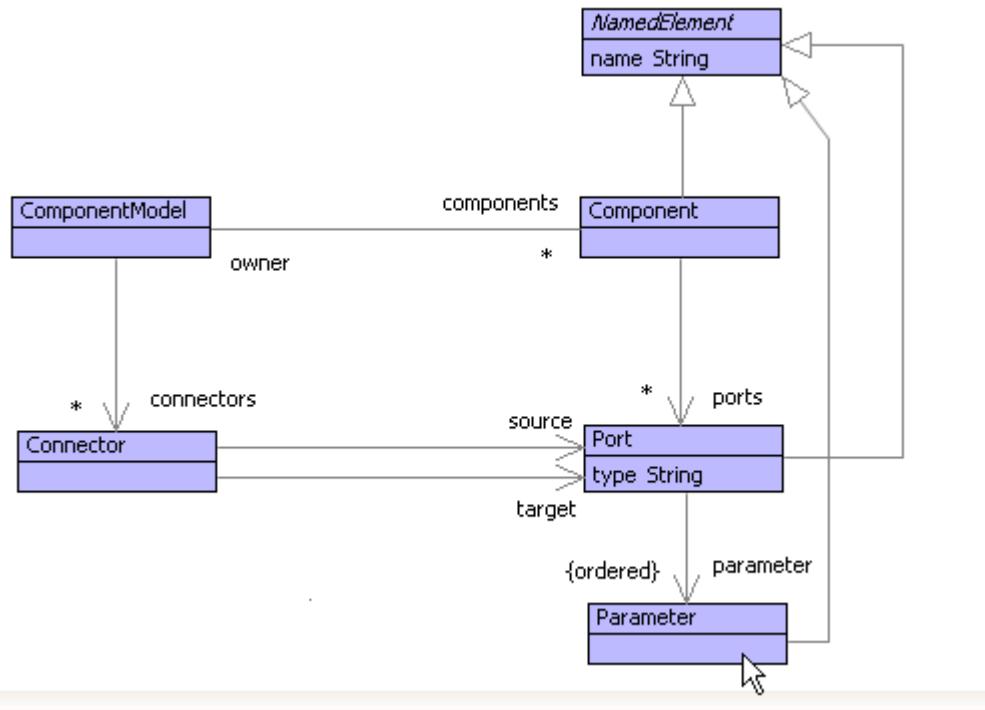
Sometimes it is useful to create instances of elements in another package or project, for example, when we want to manage them separately from the model. To do this, first create a snapshot in the project which the snapshot is to be saved under. Next select the snapshot in the browser and right click > Set Parents. A list of all available packages will be shown. Select the appropriate package, and the snapshot can now be populated with instances of the selected package.

As an example, here is a separate snapshot project that has been created for the Components model:

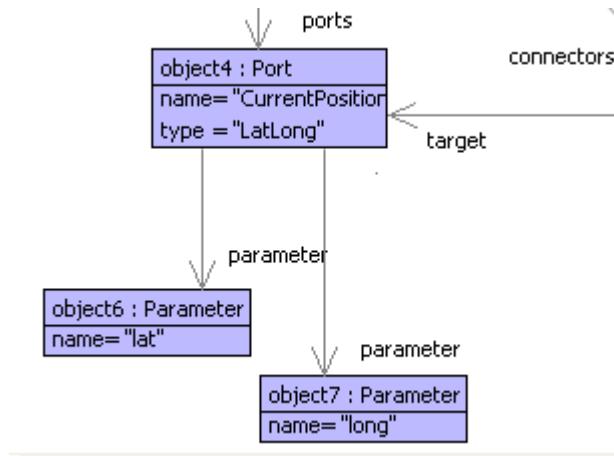


Dealing with Sequences

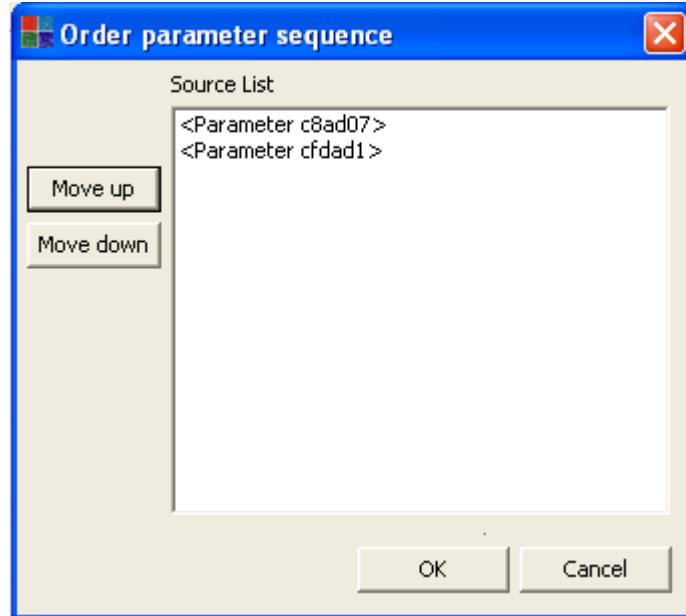
Snapshots can also accommodate the ordering of objects in a collection. As an example, imagine we want to extend the Components model to accommodate ports with sequences of parameters. The extended model is shown below:



Our example snapshot can be extended as follows:



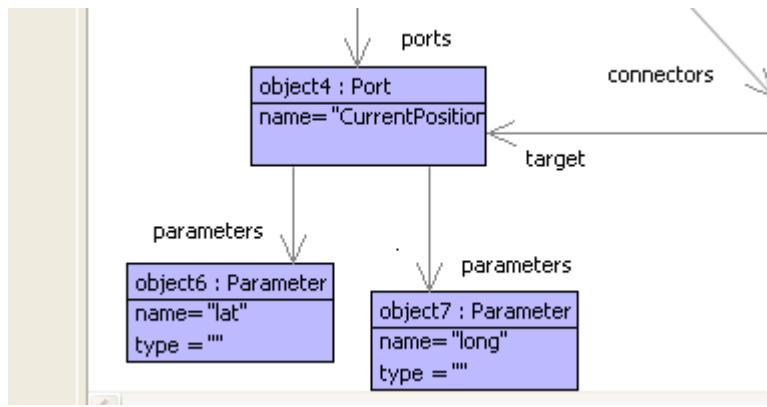
Objects can be re-ordered by right clicking on the owning object, in this case object4, and re-ordering the elements:



Edit the objects in the diagram to find out which objects are associated with each id.

Synchronisation between Snapshots and Class Models

In XMF-Mosaic, snapshots are designed to stay in step with class models, even if changes are made to the class model. As an example, let's imagine that we want to delete the attribute, type, from a Port as we will defer it to a parameter. We can delete type from Port and add it to the Parameter instead. The snapshot remains in sync, the type slot has been removed from the Port objects and has been added to the Parameter objects:



Creating Multiple Views of Snapshots

Just as with class diagrams, multiple views of the same snapshot can be created in XMF-Mosaic. This is very useful when you wish to capture different aspects of a model instance, and can considerably reduce the size of the snapshots you create.

To create a new snapshot, simply right click on the snapshot and select Create > Snapshot Diagram. This will create a partial (empty view) of the snapshot. Adding more objects to it will add new objects to the snapshot.

To create a total (synchronised view), select Create > Snapshot Diagram (Total View). This will create a diagram containing all the objects in the snapshot.

Exporting Snapshots

There are two main ways that snapshot (instance data) can be exported: as XOCL and as XML.

To export a snapshot as XOCL, right click on a the snapshot and select Deploy > Snapshot.

Choose where to put the file. A .xmlf file will be generated containing XOCL code. Here is the XOCL for the Components snapshot:

```

parserImport XOCL;

context Root
@Snapshot snapshot0

object0 = Root::Components::ComponentModel[
    connectors = Set{object5};
    components = Set{object1,object3} ]
object5 = Root::Components::Connector[
    source = object2;
    target = object4 ]
object4 = Root::Components::Port[
    name = "CurrentPosition";
    type = "LatLong" ]
object1 = Root::Components::Component[
    ports = Set{object2};
    name = "Displays";
    owner = object0 ]
object2 = Root::Components::Port[
    name = "CurrentPosition";
    type = "LatLong" ]
  
```

```
object3 = Root::Components::Component[  
    ports = Set{object4};  
    name = "Navigation";  
    owner = object0 ]  
  
end
```

The code contains the XOCL constructors and data necessary to compile and then load the snapshot back into the tool.

To re-load, simply open the file in XMF-Mosaic and then Compile and Load it. Note, the snapshot is saved by default as owned by the Root package. This can be changed to an appropriate project package, e.g. Components or ComponentSnapshots, by editing the package name.

To view the contents of a loaded snapshot, a diagram needs to be created. Select Create Diagram > Snapshot Diagram (Total View).

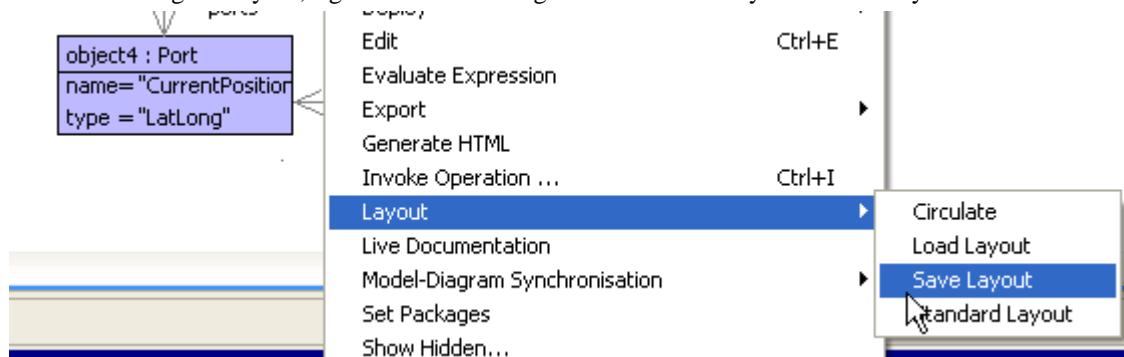
Note, this will not save the layout of the diagram. To do this, follow the instructions for saving and loading diagram layout below.

To export as XML, select the Deploy > XML option.

Saving and Loading Diagram Layout

It is possible to save and load the layout of a diagram, so that when it is imported in an un-formatted way, its layout can be reconstructed. This is useful when re-constructing snapshots saved as XML or XOCL (see above), or when working with class models written in code.

To save the diagram layout, right click on the diagram and choose Layout > Save Layout:

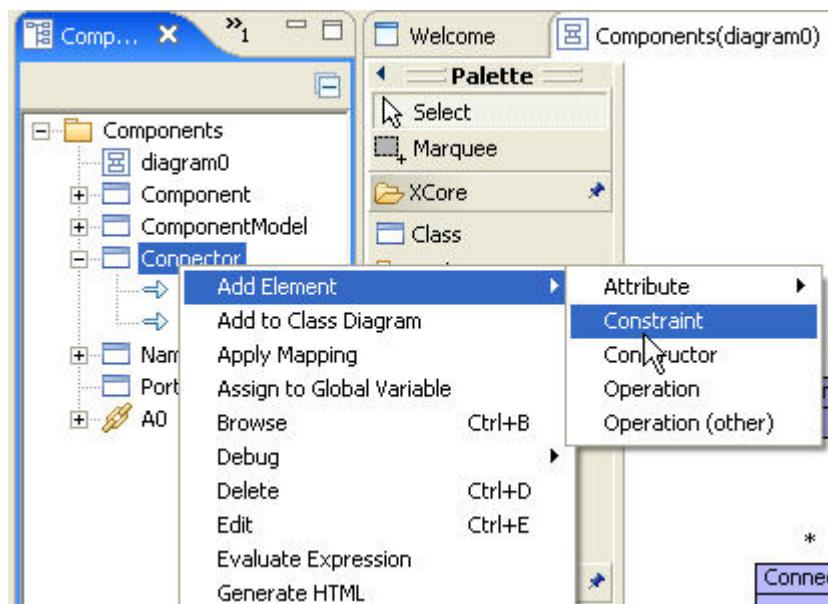


To load, right click on the diagram you wish to lay out, then choose Layout > Load Layout and load the layout file. Note, only diagram elements whose layout has been saved will be laid out.

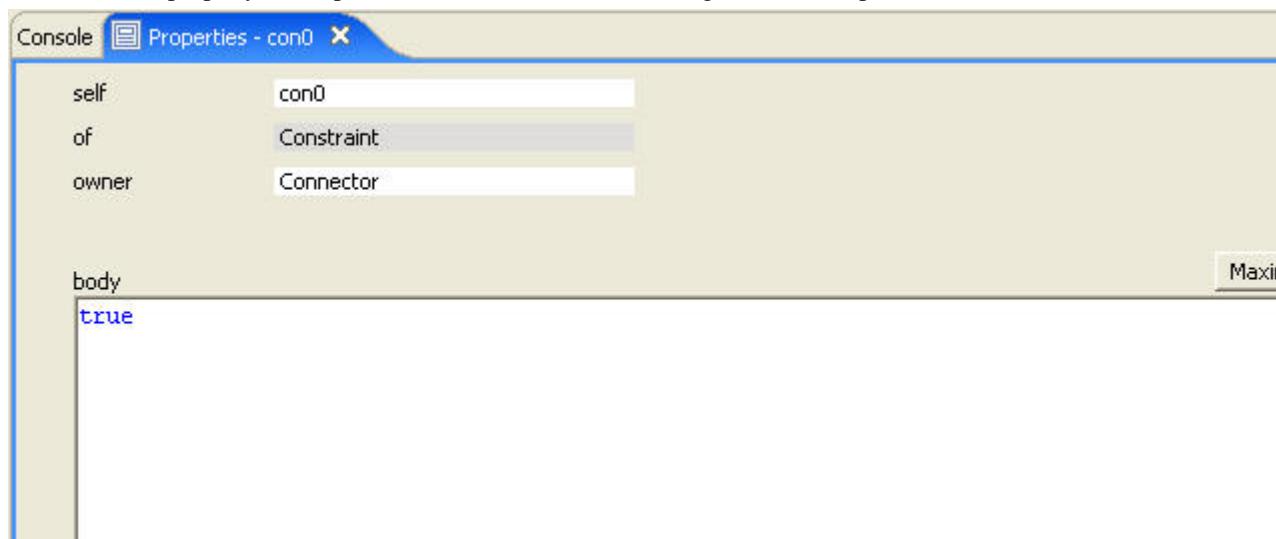
Adding Constraints

A class diagram can only capture certain information about the concepts and relationships in a domain. To add extra information, XMF-Mosaic provides a language for conveniently writing constraints and operations on models. This language is called **XOCL** (eXtensible Object Command Language). XOCL is based on OCL (the Object Constraint Language). See the XOCL walkthrough for more details.

To add a constraint, right click on a class in the class diagram or in the browser and choose New>Constraint. A new constraint is added to the browser under the class. Click on it to show the constraint property editor.

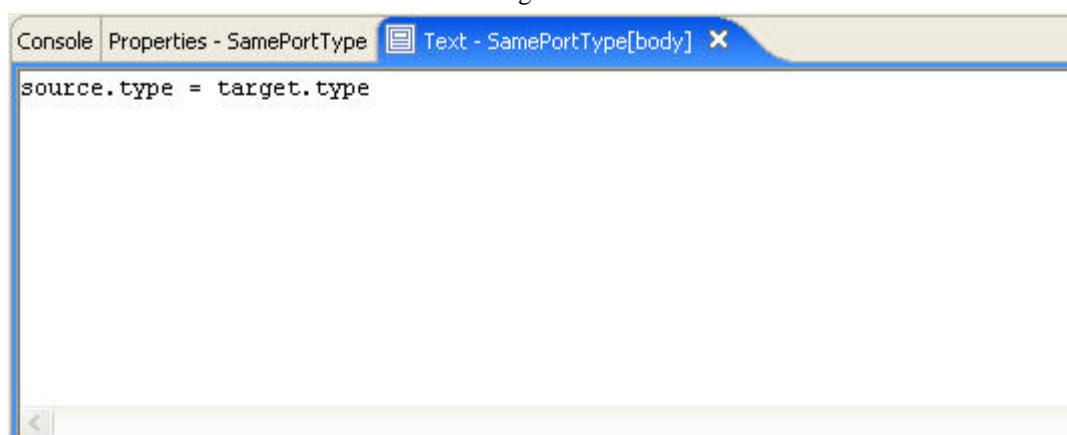


The constraint property editor provides a text window for entering constraint expressions.



The maximise button can be used to expand the window if required.

The following expression is added to the constraint. Right click > Commit Changes to update the model with the new constraint. Edit self to change the constraint name.



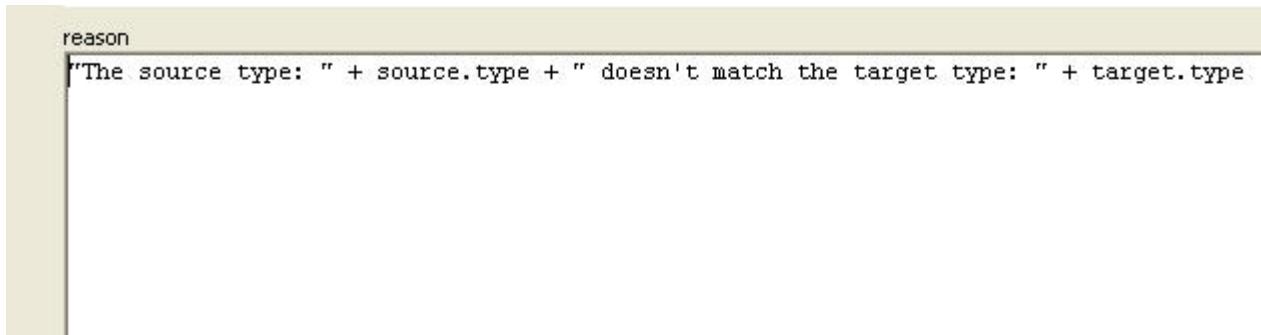
This constraint states that a connector should only connect two port if they are of the same type.

Note the code box will turn a tasteful shade of pink when the code changes. This will clear when the code is successfully committed.

Constraint Reasons

When a constraint fails, it is often useful to generate a report that describes the reason for the failure.

A reason can be added to a constraint via the reason editor (scroll down to view it). Add an Xocl expression that generates an appropriate string report. For example, the following report generates a reason for the failure of the SamePortType constraint:

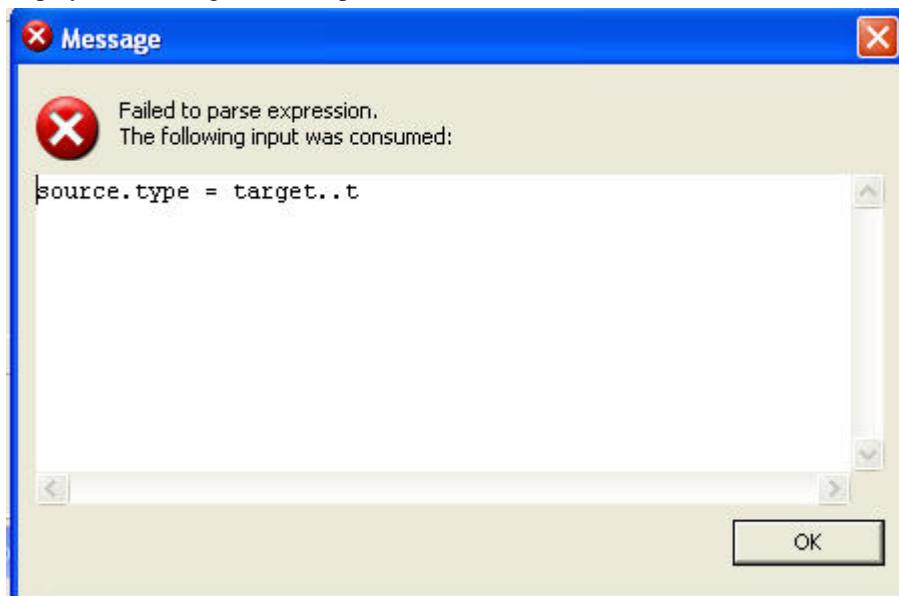


The screenshot shows a window titled "reason" with a single line of text: "The source type: " + source.type + " doesn't match the target type: " + target.type".

Parse Errors

When entering code into the bodies of constraints, operations, etc. parse errors will be displayed if an attempt is made to commit code that does not parse.

For example, try adding an additional "." to the SamePortType expression. A parse error will be displayed indicating where the parser reached before it failed.



In this case, the code box will remain pink until the mistake is removed, or, the right click > Cancel Changes option is chosen. This will return the code box to its previous (valid) state.

Many other constraints can be added to the model. As an example, the following constraint on ComponentModel ensures that no two components can have the same name:

The screenshot shows the Rational Rose UML console with a properties view for a constraint named "NoDuplicateNames". The properties are:

- self: NoDuplicateNames
- of: Constraint
- owner: ComponentModel

The body of the constraint contains the following UML expression:

```
components->forAll(c1 |
  components->forAll(c2 |
    c1 <> c2 implies c1.name <> c2.name))
```

Checking Snapshots

Objects in a snapshot can be checked against their constraints in the following ways:

Via the Object menu in the browser:

- Right click on the object, select Invoke Operation ..., then choose checkConstraints()

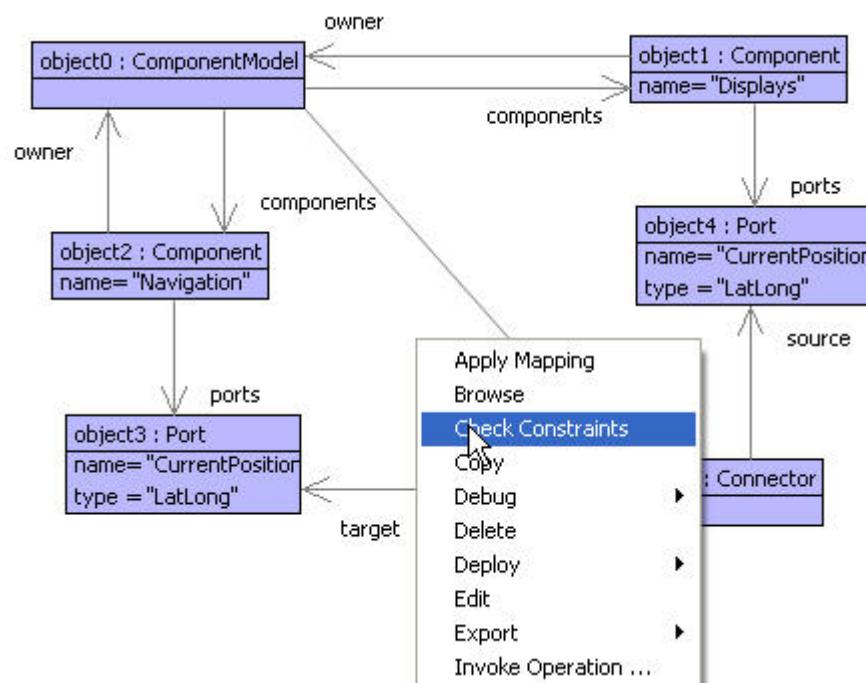
Via the Snapshot menu in the browser or diagram:

- Right click on the snapshot icon in the browser, or the background of the snapshot diagram, and select Check Constraints. This will check the constraints on all the objects in the snapshot. Note, a choice of whether to check only the objects in the diagram, or all the objects in the snapshot (if using a partial view) can be made at this point.

Via the Console:

- Invoke the checkConstraints() operation on the relevant object. The details of the ConstraintReport can be edited by appending .edit() to checkConstraints().

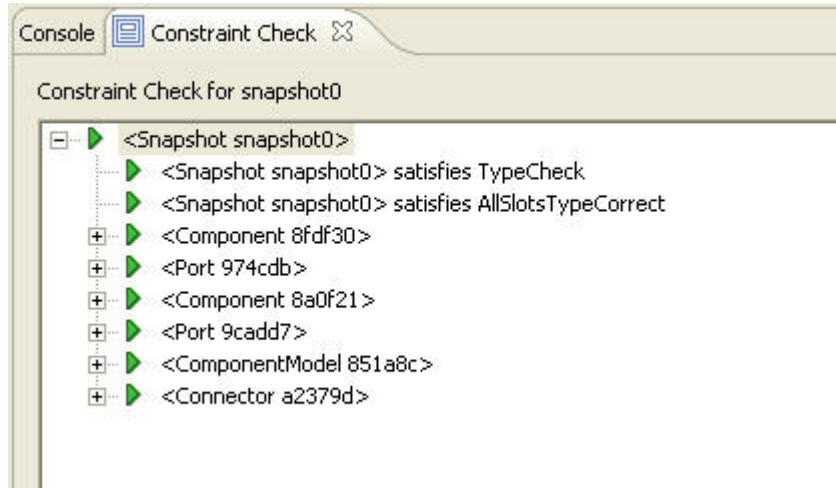
Here is the constraint checker being invoked via the snapshot diagram:



The result of checking an object or snapshot is to display a *ConstraintReport*: a tree of object constraint reports.

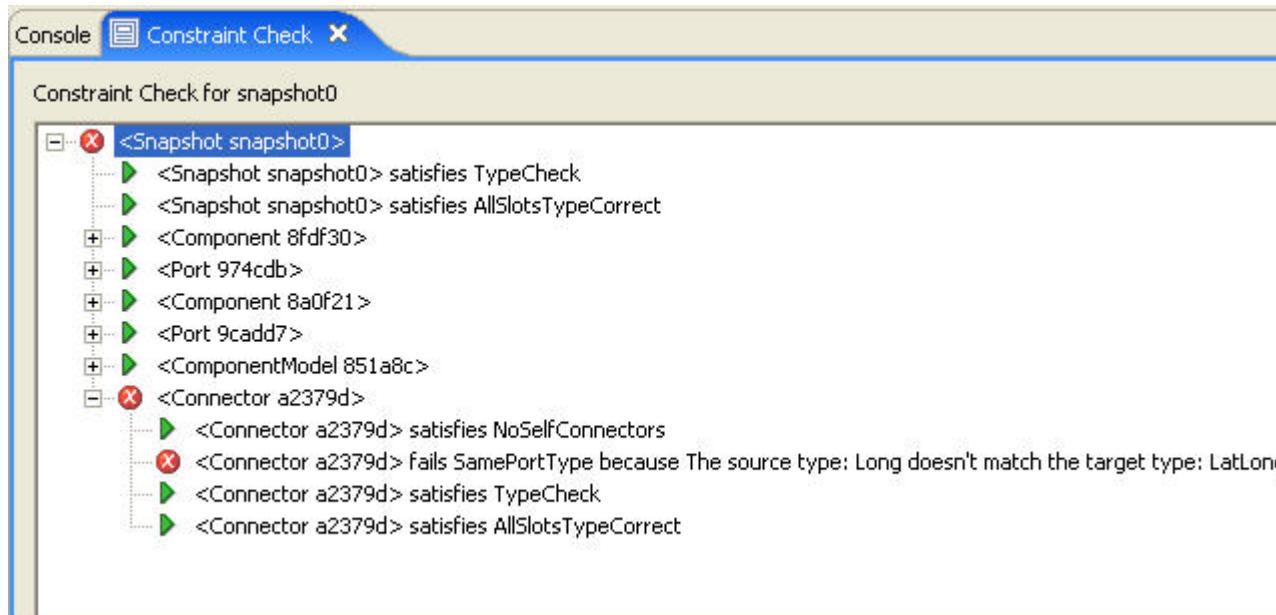
Interpreting the results

A ConstraintReport is displayed as a tree containing the result of each constraint check:



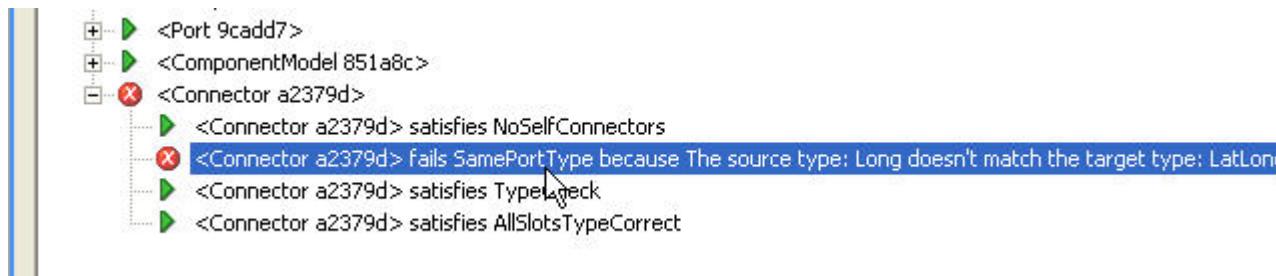
In this example, every object has passed its constraints.

Let's change the snapshot so that it fails a constraint, for example by setting the type of one of the ports to be different from a port it is connected to. Do this by editing the value of type in the object. Running the constraint checker produces the following:



In this case, the snapshot fails the constraint check, and if we expand it to see why, it is because the types don't match. Furthermore, because we added a Constraint Reason when we wrote the constraint, it has printed out some helpful diagnostics.

The failed constraint can be edited by double clicking on it.



Finally, a constraint report can be exported in HTML. Right click on the background of the report and choose Export > HTML, or View > HTML.

Constraint Report (Thu Sep 22 14:47:00 BST 2005)

Checks for <Snapshot snapshot0>

Dependent checks

Constraint TypeCheck

Constraint AllSlotsTypeCorrect

Checks for <Component 8fdf30>

Checks for <Port 974cdb>

Checks for <Component 8a0f21>

Checks for <Port 9cadd7>

Checks for <ComponentModel 851a8c>

Checks for <Connector a2379d>

Exercises

Here are some other constraints to try adding to the domain model:

- A connector cannot connect the same port.
- A component cannot contain two or more ports with the same name.
- The component models connectors can only connect ports belonging to its components.

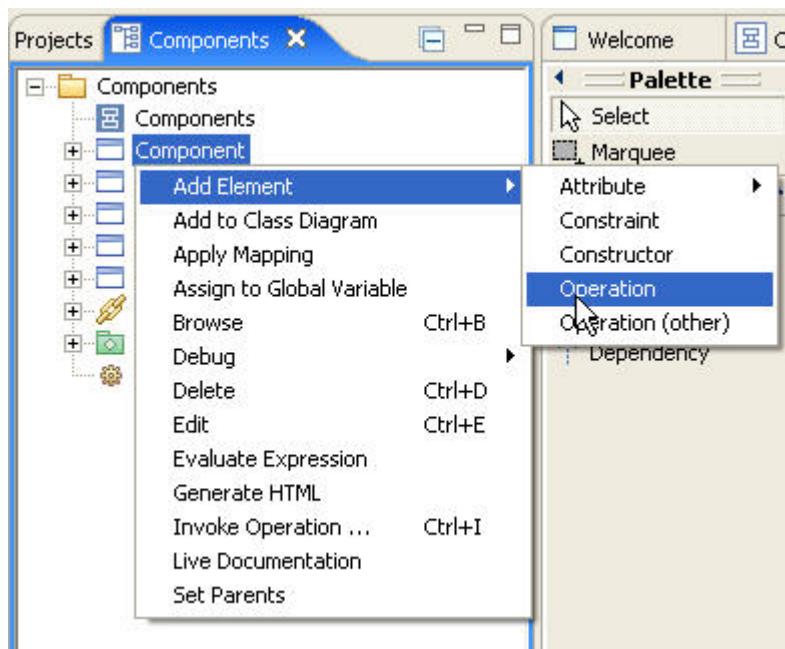
Write constraint reasons for the above.

Adding Queries

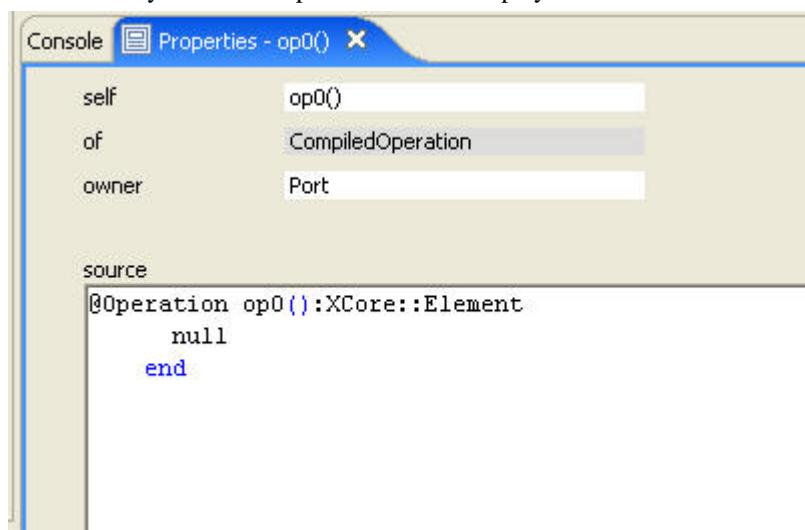
Queries are **operations** that do not change the state of the model. They are a very useful means of navigating around a model, filtering and collecting information about its properties.

Operations can be added to classes in a similar way to adding constraints and queries.

To add an operation, right click on a class in the class diagram or in the browser and choose Add Element>Operation. A new operation will be added to the browser under the class. Click on it to show the operation editor.



The default syntax for an operation will be displayed.



The name of the operation, its parameters, return type and body can all be entered as part of the code.
Right click > Commit Changes to update the model.

You should not change the "@Operation" and "end" parts of the expression.

An example, here is a query on the class ComponentModel that returns all the connectros whose source and target port types don't match.

The screenshot shows a console window titled "Properties - dontMatch()". The "source" section contains the following XQuery code:

```

@Operation dontMatch():XCore::Element
    connectors->select(c |
        c.source.type <> c.target.type)
end

```

Queries may also take parameter values. Here is a query on the class ComponentModel that returns all the connectors that are connected to a port, p.

The screenshot shows a console window titled "Properties - connectedTo()". The "source" section contains the following XQuery code:

```

@Operation connectedTo(p : Port):XCore::Element
    connectors->select(c |
        c.target = p)
end

```

Exercises

Add the following queries to the domain model:

- A query on the class ComponentModel that returns the set of components that have more than x ports.
- A query on the class Component that returns all the components whose ports it is connected to.

Running Operations

Operations and queries can be run in a number of different ways in XMF-Mosaic. The following describes the most common ways.

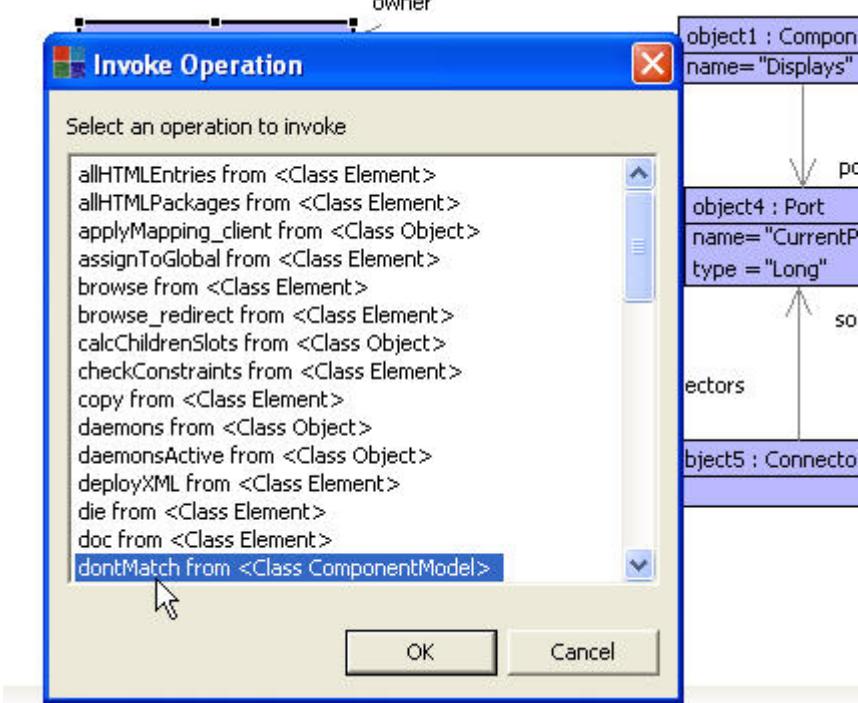
Running Operations on Snapshots

Operations and queries can be run on objects in a snapshot.

Right click on the object in the browser and select Invoke Operation. This will display the operations that can be invoked on the object. Because all classes inherit from Object, this list will include all operations that have been defined for an Object. In addition, operations on the class will be displayed.

Only operations with zero parameters are displayed in this list (operations that require parameters should be accessed via the console or via the expression evaluator - see later).

As an example, we can call the dontMatch() query via Invoke Operation.

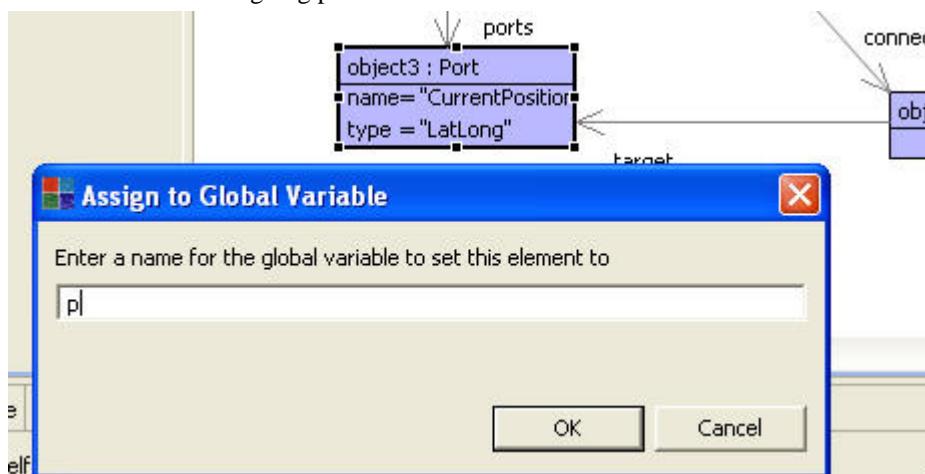


The result of running the operation will be displayed in the property editor.

Note that operations are immediately available via the menu once they have been successfully committed.

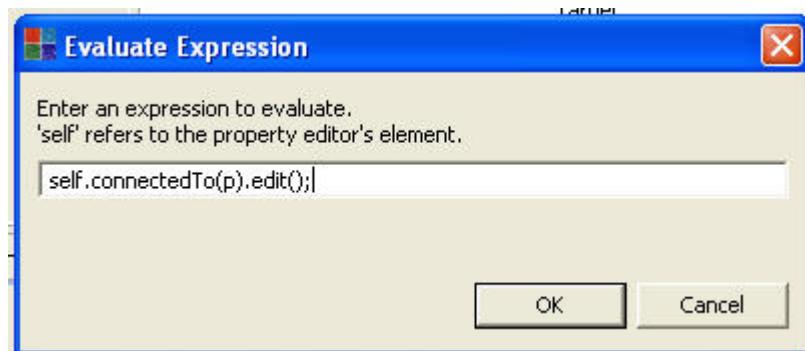
In the case of operations with parameters, we must be able to pass the parameters into the operation in order to call it. This can be achieved using the console (see later). However, there is menu based support for this as follows:

First assign any objects that will be used as parameter values to global variables. To do this right click on the object in a property editor and then select Assign To Global Variable. Give the object a name, in this case we are assigning p to a Port.



Next, select the object that the operation is to be run on, and select right click > Evaluate Expression.

A simple expression editor will appear. Type the operation in, followed by an edit() operation to show the results. Press OK to run the operation.



Operations

In addition to writing queries, operations also enable instances of models to be manipulated (just as you would manipulate data in a programming language). Xocl provides a number of imperative (programming style) features for the purpose of writing operations. These include standard "for", "while", and "case" constructs among others. Full details can be found in part 3 of the Bluebook.

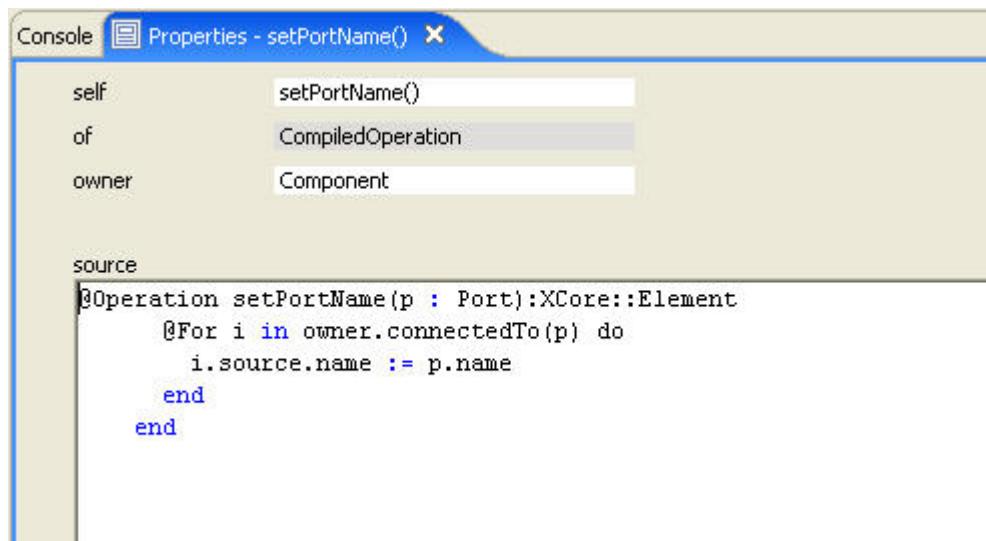
As an example, the following operation sets the name of a Port to be the value passed by the parameter of the operation.



Right click > Commit Changes to update the model with the new operation. Edit self to change the operation name.

Again, if the code is not syntactically correct a parse error will be raised.

Operations can make use of queries to access properties of a model before they manipulate them. For example, the following operation uses a query to calculate all the connections that target the port, p, and then sets the source port names to be the same as p's name:



The screenshot shows the Xtext Properties view for the operation `setPortName()`. It displays the following information:

- self**: `setPortName()`
- of**: `CompiledOperation`
- owner**: `Component`
- source**:

```
@Operation setPortName(p : Port):XCore::Element
    @For i in owner.connectedTo(p) do
        i.source.name := p.name
    end
end
```

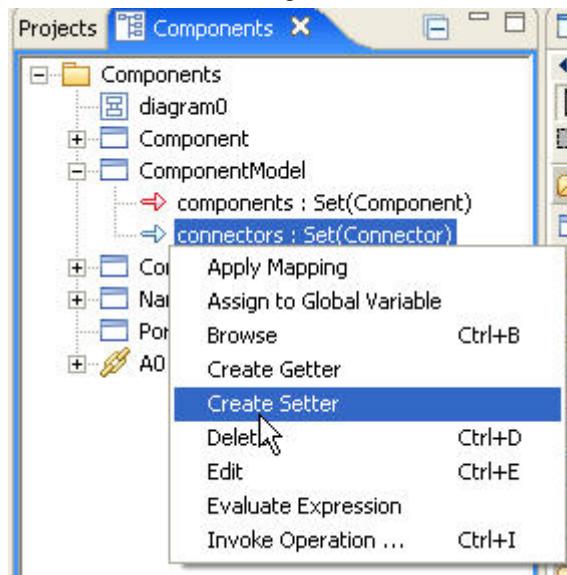
Exercises

Try adding the following operations to the domain model:

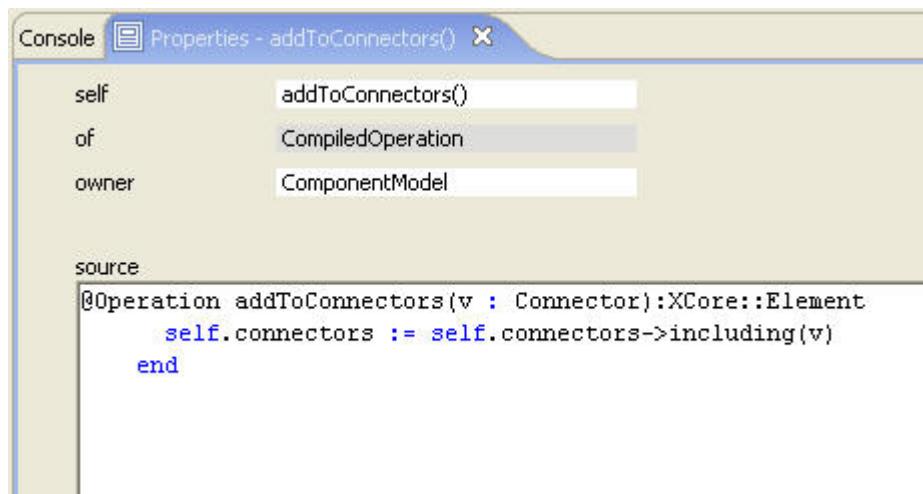
- An operation on the class Component that sets its owner to be a passed ComponentModel.
- An operation on the class Component that sets the name of the Component to be a passed String.

Adding Getters and Setters

Getter and Setter operations for specific attributes can be conveniently added to a class. Right click on the attribute (in the diagram or browser) and select Create Getter or Create Setter.



Creating a Setter will add operations for adding new instances to the attribute (as shown below), while adding a Getter will add an operation to query the value of an attribute.



The screenshot shows the Xtext Properties view for the 'addToConnectors()' operation. It displays the following information:

- self**: addToConnectors()
- of**: CompiledOperation
- owner**: ComponentModel
- source**:

```
@Operation addToConnectors(v : Connector):XCore::Element
    self.connectors := self.connectors->including(v)
end
```

Exercises

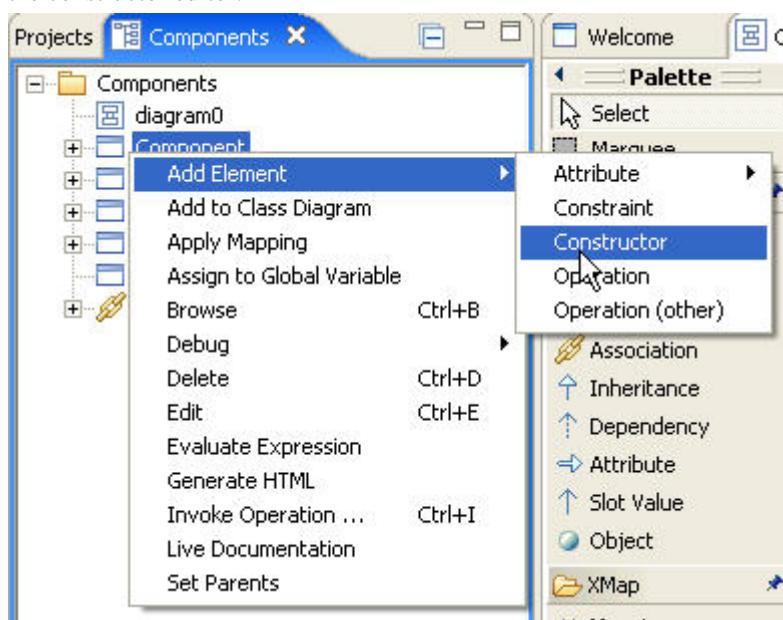
Create a Setter for the attribute "components" of the class ComponentModel.

Adding Constructors

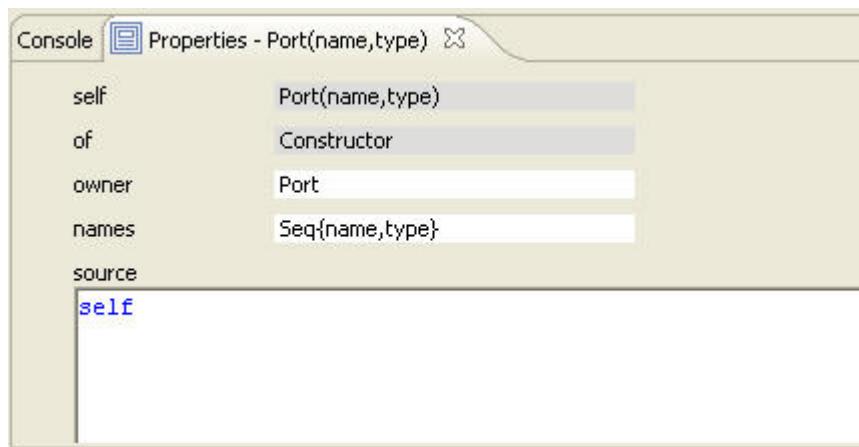
Constructors are operations on classes which are called when creating a new instance of the class.

Note by default all classes have the empty constructor and this need not be defined.

To add a constructor right click on a class in the class diagram or in the browser and choose New>Constructor. A new constructor will be added to the browser under the class. Click on it to show the constructor editor.



Parameters can be added to a constructor. To do this, add the name of each parameter in the desired order within the comma separated sequence of names. For example, the following constructor for a Port has 'name' and 'type' as its parameters.

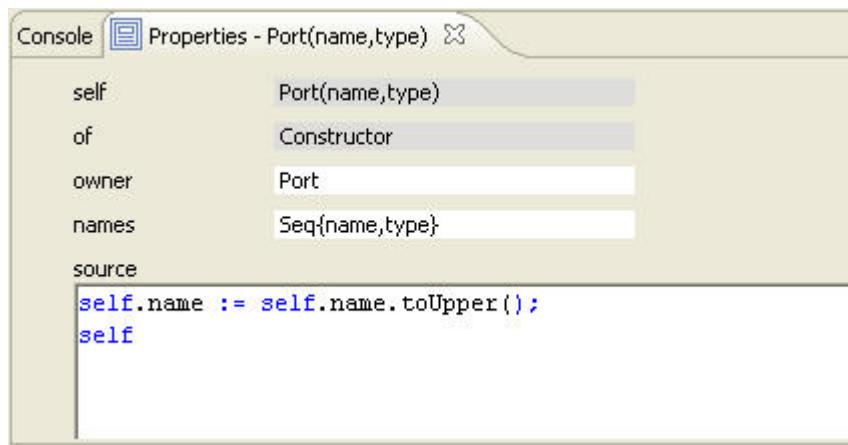


Note each name should correspond to a name of an attribute of the class that is to be instantiated when the constructor is called.

Any number of constructors may be defined for a class. This enables different tasks to be performed per type of instantiation depending on the number of parameters that are passed.

In addition to supporting simple parameter passing, XOCL code can be entered in the body of a constructor to perform a specific task when the constructor is called.

Here is an example of an action that sets the name of a Port to be uppercase when it is created.



After this action is performed, an action to return the object (self) is called. If this were not added the constructor would only return the value of self.name.

Exercises

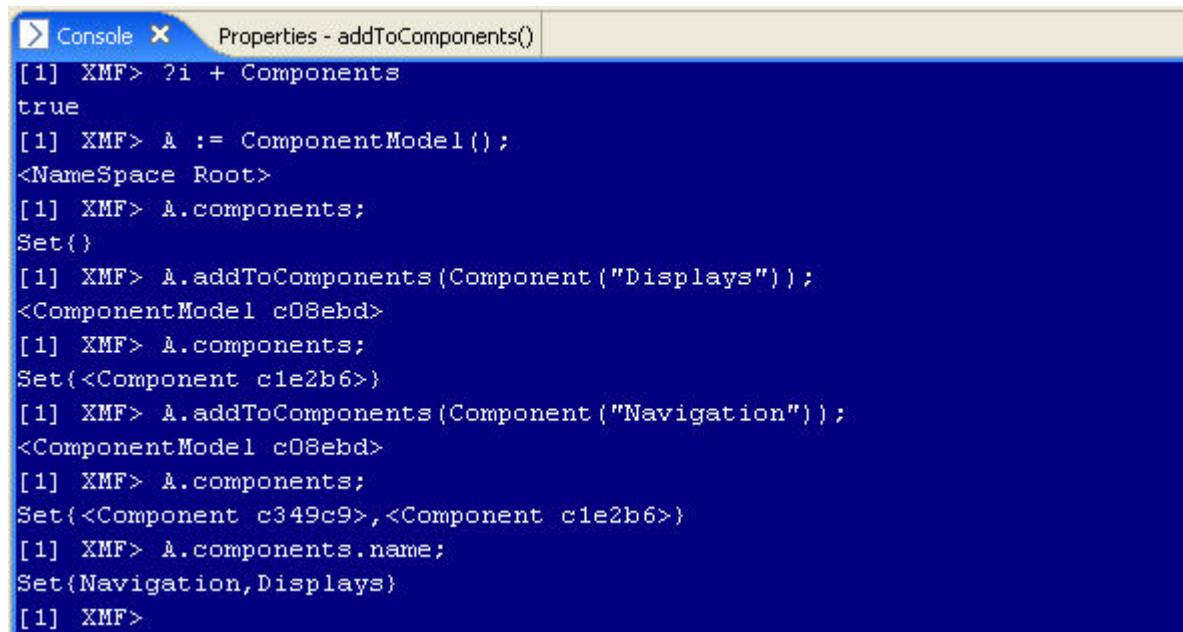
Try adding a constructor for a Component takes its name as a parameter.

Interacting with Models using the Console

Once a model has been built, it is useful to validate that it is accurate. One way to do this is by interacting with the model via the console. The console is an interpreter for evaluating XOCL expressions.

Switch to the console by clicking on the Console View tab on property editor. XOCL expressions can be entered that evaluate to create instances of models, modify them and invoke operations on them.

For example, the following shows the use of the console to interact with and test the Components model:



```
Console X Properties - addToComponents()
[1] XMF> ?i + Components
true
[1] XMF> A := ComponentModel();
<NameSpace Root>
[1] XMF> A.components;
Set()
[1] XMF> A.addToComponents(Component("Displays"));
<ComponentModel c08ebd>
[1] XMF> A.components;
Set(<Component c1e2b6>)
[1] XMF> A.addToComponents(Component("Navigation"));
<ComponentModel c08ebd>
[1] XMF> A.components;
Set(<Component c349c9>, <Component c1e2b6>)
[1] XMF> A.components.name;
Set(Navigation, Displays)
[1] XMF>
```

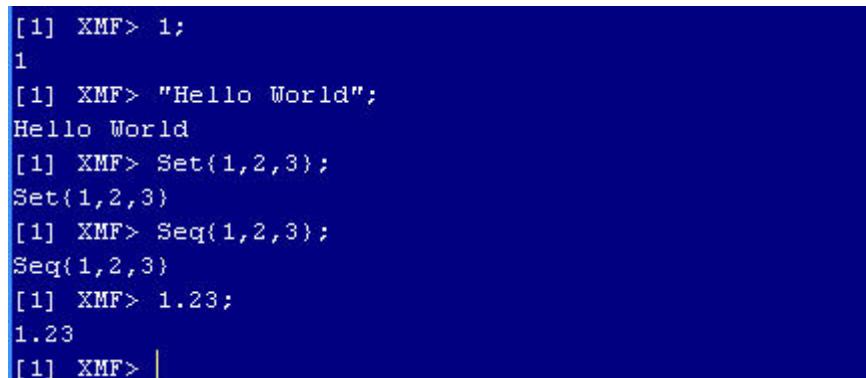
Here is what is happening:

- The `?i` command is used to import the `Components` package so that all its contents are visible. Note, otherwise we would have to give the full path names to each element, e.g. `Components::Component`.
- `A` is the result of creating an instance of the class `ComponentModel` In XOCL, the use of brackets after a class name denotes the invocation of a constructor. Note, we will have needed to create a constructor for `Component` beforehand.
- The result of navigating down the components of `A` is evaluated. Because no components have been added to `A` it returns the empty set `Set{}`.
- A new Component is added to `A` by invoking the `addToComponents()` operation on `A`. The string "Displays" is passed as a parameter to the constructor.
- We enter "`A.components`" again and a Set containing the new Component is returned.
- The process is repeated by adding a new Component named 'Navigation'.
- Finally, the names of the components are returned by performing the expression `A.components.name`

Other Console Hints and Tips

The console is a very flexible tool for interacting with models. The following are some examples.

Basic variable values can be created simply by typing them into the console. In addition, the console is a useful place to test out XOCL operations and expressions.



```
[1] XMF> 1;
1
[1] XMF> "Hello World";
Hello World
[1] XMF> Set(1,2,3);
Set(1,2,3)
[1] XMF> Seq(1,2,3);
Seq(1,2,3)
[1] XMF> 1.23;
1.23
[1] XMF> |
```

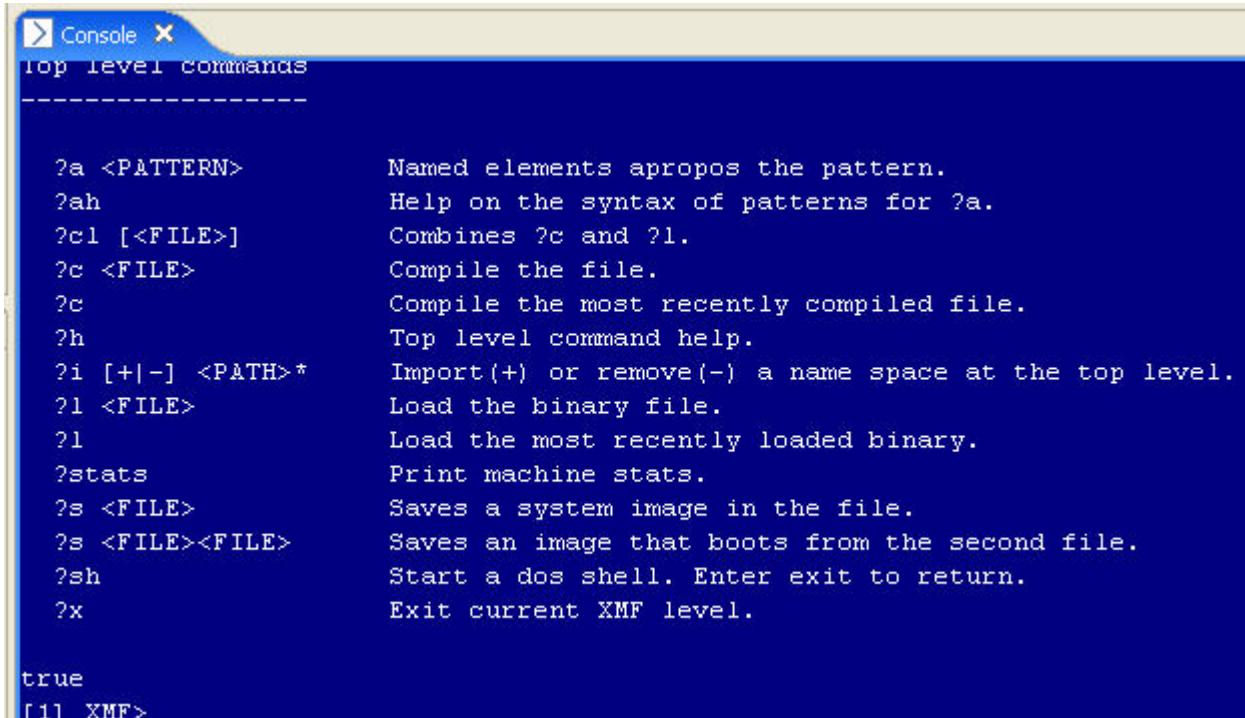
All values can be assigned to variables.

The console is also useful for creating instances of classes. As described in the XOCL manual, there are two ways that instances of classes can be created in XOCL. The first is to use a constructor as described above. The second approach is to use a *pattern constructor*. The following uses a pattern constructor to create an instance of a component and match its name to the string literal "Navigation". The advantage of a pattern constructor is that it allows values to be explicitly assigned to local variables.

```
[1] XMF> x := Component[name = "Navigation"];
<NameSpace Root>
[1] XMF> x;
<Component d54784>
[1] XMF>
```

When using the console, the up and down cursor keys can be used to move between previous commands.

The console provides a number of "hot keys". These can be accessed by typing ?h (no semi-colon is required).



A screenshot of a terminal window titled "Console". The title bar has a blue header with the word "Console" and a close button. Below the title bar, the text "top level commands" is displayed in white on a dark background. A horizontal dashed line follows. Below this, a list of commands is shown in white text:

?a <PATTERN>	Named elements apropos the pattern.
?ah	Help on the syntax of patterns for ?a.
?cl [<FILE>]	Combines ?c and ?l.
?c <FILE>	Compile the file.
?c	Compile the most recently compiled file.
?h	Top level command help.
?i [+ -] <PATH>*	Import(+) or remove(-) a name space at the top level.
?l <FILE>	Load the binary file.
?l	Load the most recently loaded binary.
?stats	Print machine stats.
?s <FILE>	Saves a system image in the file.
?s <FILE><FILE>	Saves an image that boots from the second file.
?sh	Start a dos shell. Enter exit to return.
?x	Exit current XMF level.

At the bottom of the list, the word "true" is followed by a cursor. At the very bottom of the window, the text "[1] XMF>" is visible.

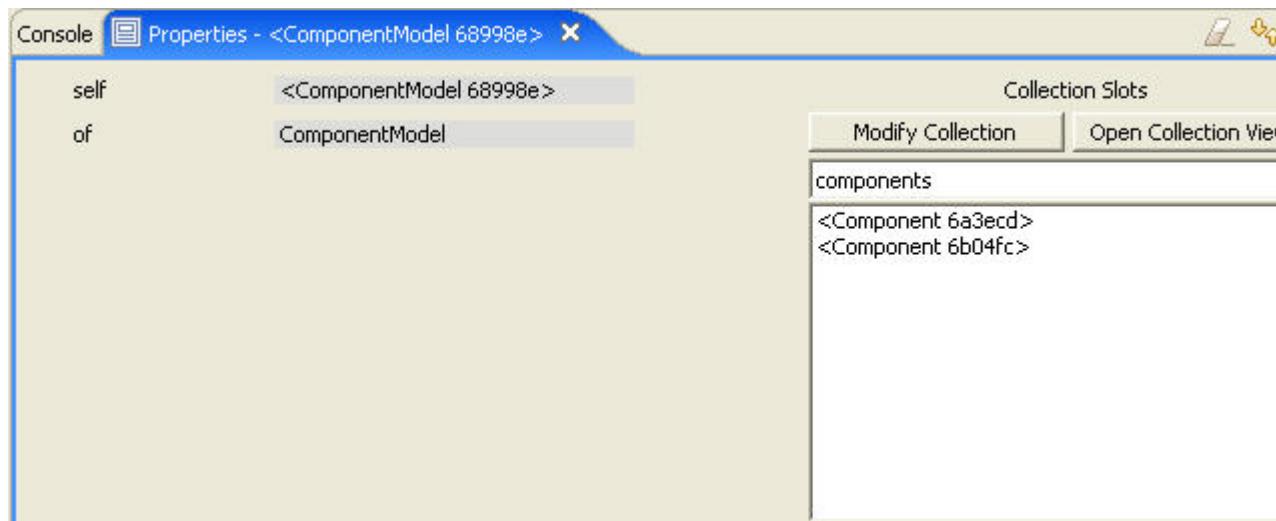
Any operation running in the console can be interrupted by pressing the escape key.

Editing Values via the Console

At any point we can edit/view the properties of an object in a property editor by running edit() against it. For example, let's edit the ComponentModel instance which was assigned to the variable A (above):

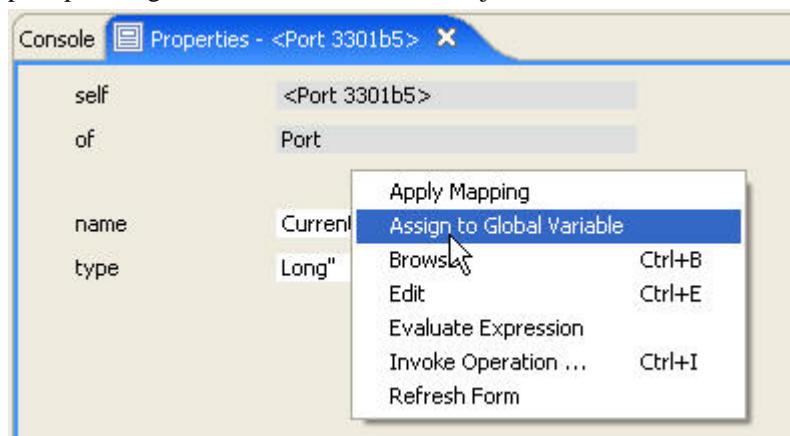
```
[1] XMF> A.edit();|
```

A collection editor will be displayed containing the instance.



Using Global Variables

Global variables can be assigned to elements which can then be used in the console. For example, edit the properties of a Port object, then select right click > Assign to Global Variable. The tool will prompt for a global variable name for the object.



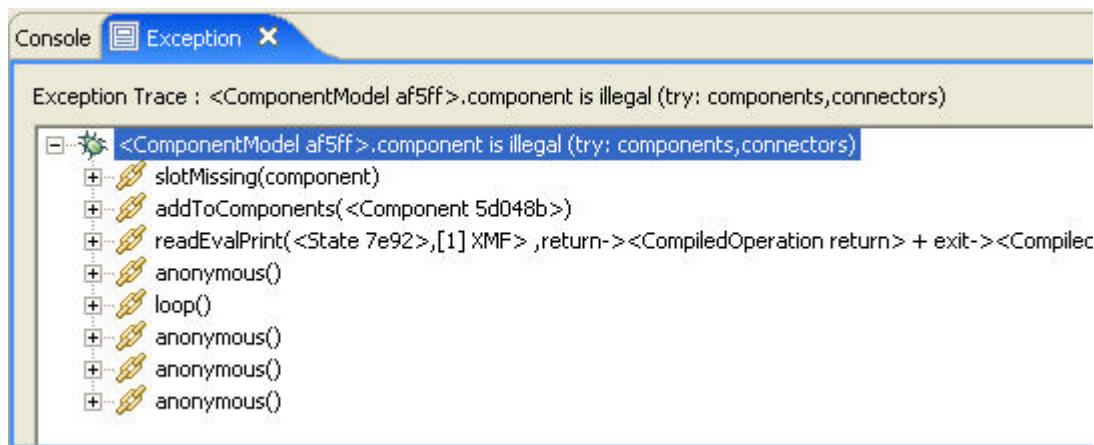
The port object is then available in the console:

```
p;  
<Port d8c2>  
[1] XMF> |
```

Error Reports

If errors occur when building the model, they will be shown as an ErrorReport.

Try modifying addToComponents() so that it attempts to update an invalid slot, e.g. change "components" to "component".

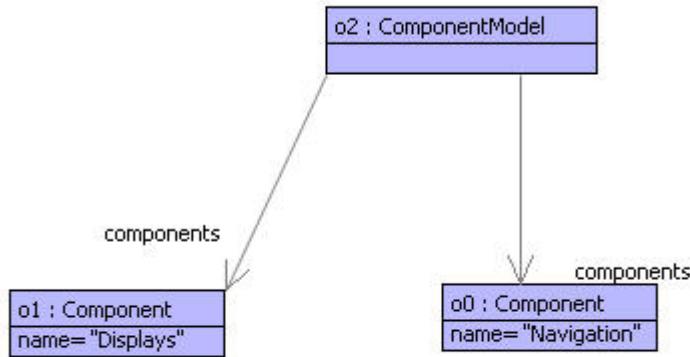


Error reports can be expanded to show the sequence of calls that resulted in the error. These go from the most specific call right down to the engine calls that handled the error.

Using Snapshots

Generating Snapshots from the Console

Snapshots can also be generated from an object by running `toSnapshot()` followed by `showDiagram()` on the object. Select TotalView (a partial view will by default not be populated with objects). For example, running `A.toSnapshot().showDiagram()` in the console produces the following snapshot:



Adding Package Operations

In XMF-Mosaic, packages have class like features. They can have operations and attributes, can be instantiated and inherit from other packages.

Package operations in particular are useful for capturing global operations that apply across an entire model.

Adding and Running Package Operations

To add an operation to a package, right click New > Operation. Edit the operation in the browser. The following operation creates an instance of a `ComponentModel` and populates it with values.

```
Console Properties - factory() X
self      factory()
of       CompiledOperation
owner    Components

source
@Operation factory():XCore::Element
    let c1 = Component[name = "Displays"];
        c2 = Component[name = "Navigation"]
    in let nl = Connector[source = "c1",target = "c2"]
        in ComponentModel[components = Set(c1,c2),connectors = Set(nl)]
    end
end
end
```

The operation can be invoked via the console using the special package operation invocation syntax (Package::Op()). The result of running the operation can be edited by adding .edit() after the operation call.

Next Steps

Although more can be added, most of the basic concepts in the domain have been captured. In the remainder of this document, different ways of manipulating models of languages are described and illustrated through this example.

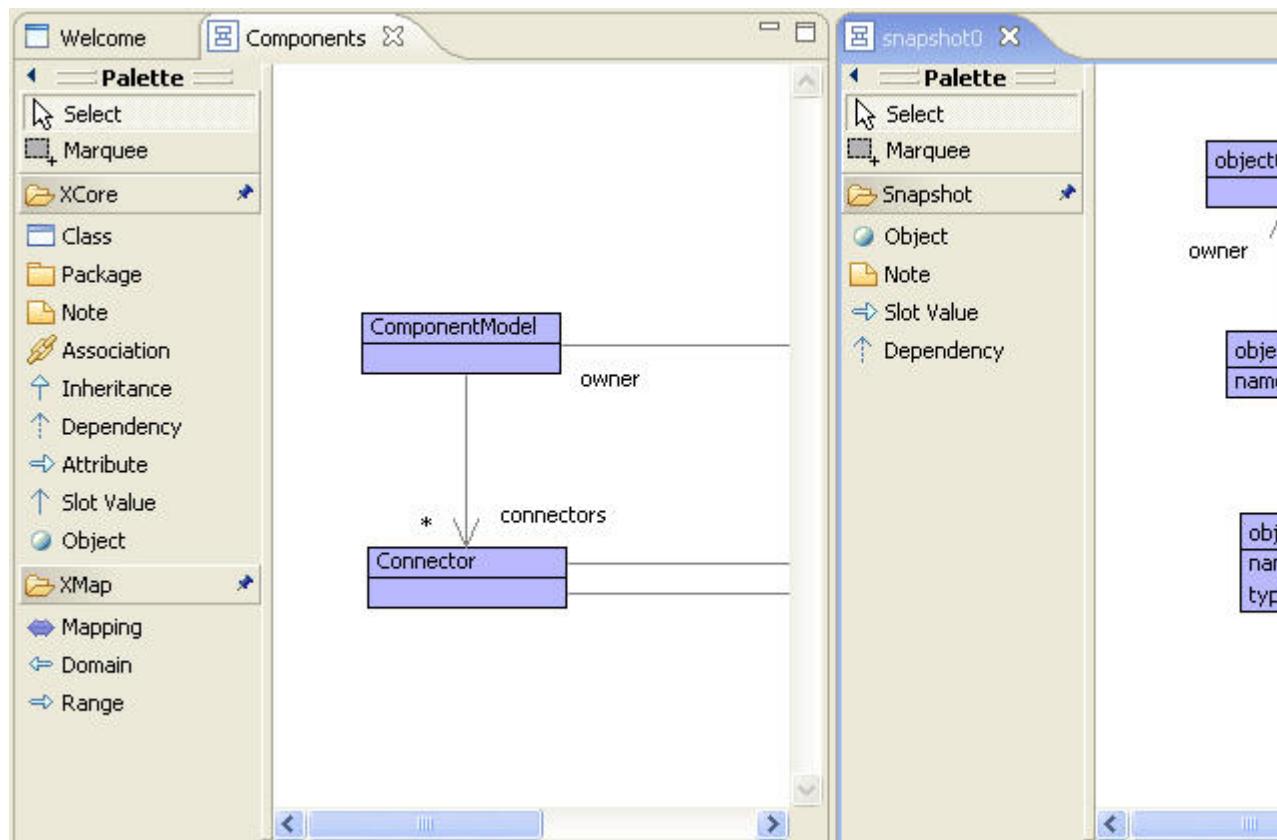
Chapter 3. Working with the Modelling Interface

The previous chapter showed how models can be constructed in XMF-Mosaic. This chapter gives some general hints and tips for getting the most from the modelling interface.

Windows

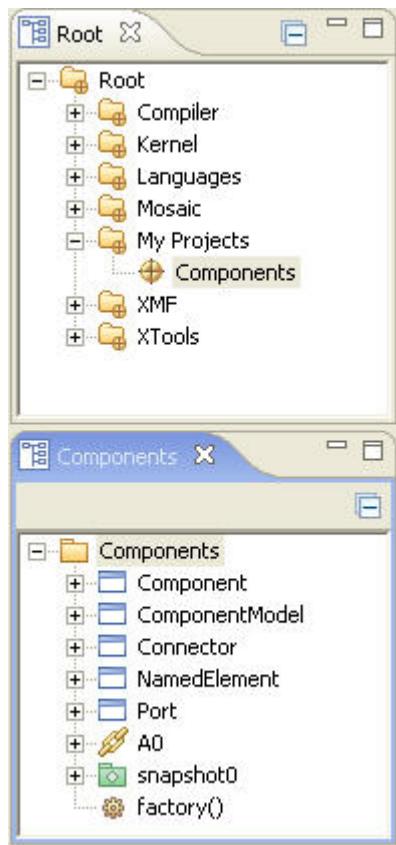
Displaying Multiple Diagrams and Windows

Any number of diagram windows can be displayed at once by dragging a diagram tab to one of the edges of another diagram. For example, a snapshot diagram can be placed next to a class diagram:

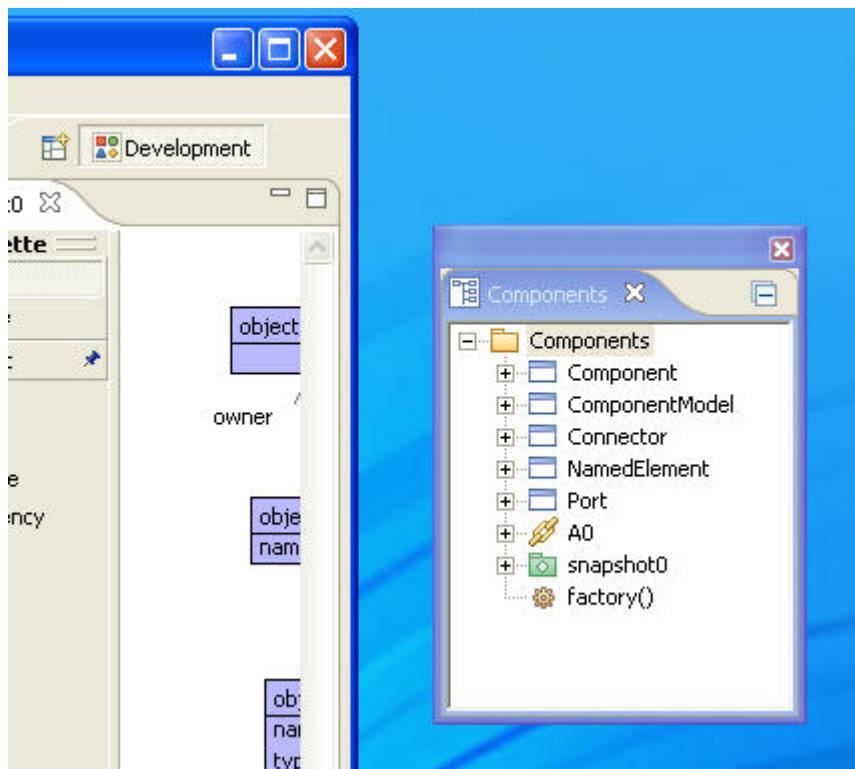


Diagrams can be tiled both horizontally and vertically.

Browser and property editors can be split in exactly the same way. For example, the Root browser and a Project browser can be split:

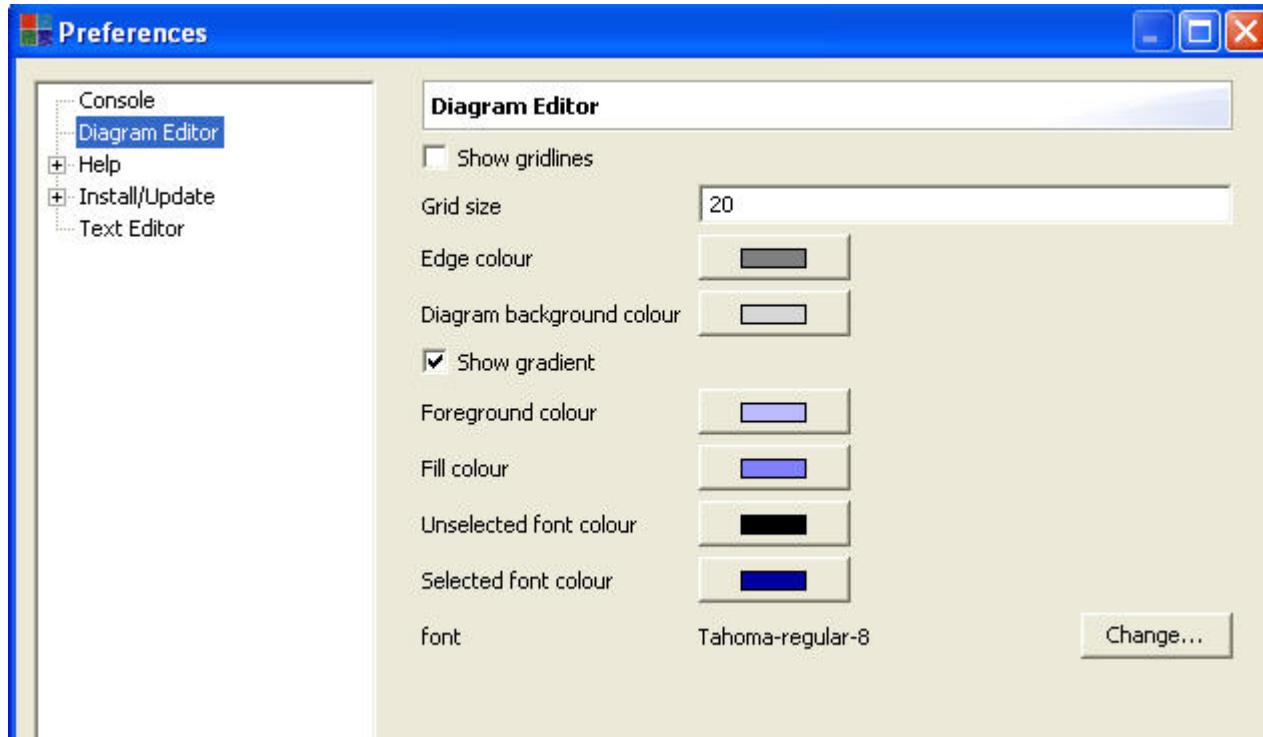


It is also possible to move windows outside of XMF-Mosaic. Both the browser and console can be moved in this way.



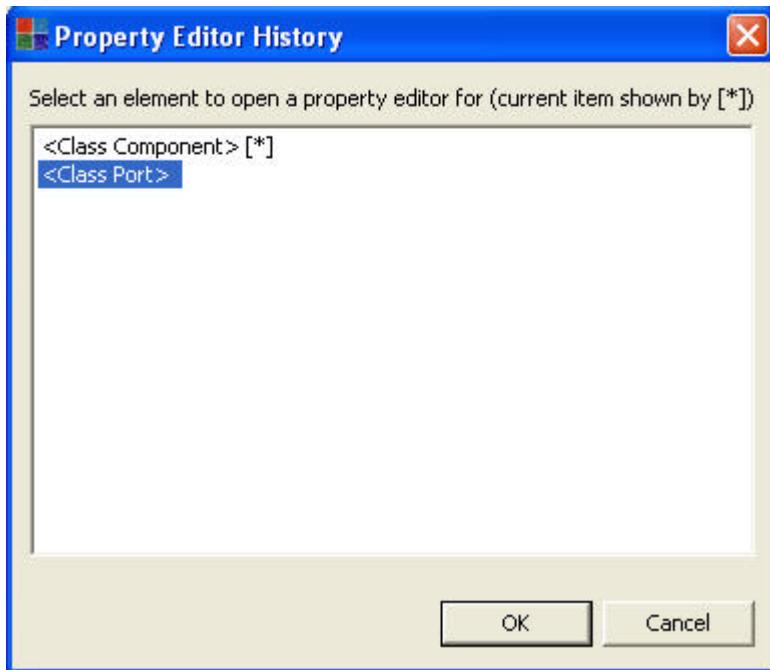
Preferences

Many different preferences can be set for different aspects of XMFMosaic, including the diagram editor, console and text editors. Preferences can be changed via the Window > Preferences menu and are automatically loaded at boot. Preferences can also be imported and exported.



Property Editor

Property editors provide facilities for managing property histories. Everytime a new property is displayed, the old property is overwritten in the property editor window. However, the old property editor can still be viewed by looking at its history. For instance, if we edit the class Port followed by the class Component, the properties of the Airport will be the last to be displayed in the property editor. To view the history right, click on the history icon in the top right hand of the property editor. Select the property to view: We can also flip backwards and forwards between histories using the left and right arrow buttons (like a browser).



If you don't want a property to be overwritten, it can be locked by clicking on the padlock icon in the top right hand of the property editor window:



Clicking on the eraser icon clears the history.

Chapter 4. An Introduction to XOCL

Introduction

XOCL (the eXtensible Object Command Language) is a flexible and powerful language for writing programs and constraints on models. XOCL is based on OCL (the Object Constraint Language) but extends it with programming features that make it much more productive.

XOCL includes many useful features, including facilities for conveniently navigating around models, and for filtering and manipulating model data. XOCL is based on standard OO programming principles, which means it is easy to learn. XOCL can be used in a wide variety of places in a model; wherever there is a need to evaluate a property of a model or modify it in some way. This means that once you have learnt XOCL it can be used in many different contexts.

This chapter provides a walkthrough of the key features of XOCL. It includes an introduction to its basic types, the basic features offered by XOCL for manipulating values and navigating around models, and its imperative programming features.

For an in-depth description of all the features of XOCL, see Part 3 of the Bluebook.

Basic Types

XOCL provides four basic types

- Boolean
- String
- Integer
- Float (64 bit real numbers)

All the usual operators, e.g. arithmetic operators, are provided (see the Bluebook for a detailed list).

XOCL also provides the inbuilt value "null" for situations when the result of an expression cannot be evaluated.

Examples

Type the following into the console:

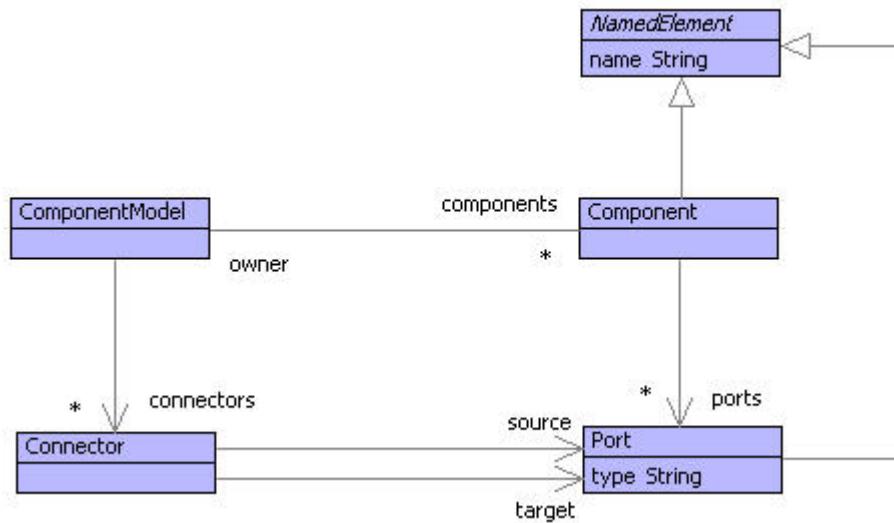
```
1 + 2;
10.mod(3);
true and false;
true implies false;
"Hello" + "World";
"Hello".toUpperCase();
10.1.cos();
3.14.sqrt();
null;
```

Models

XOCL is used to add important detail to models. One of the most important types of models is a class model, where XOCL is typically used to write constraints, operations and constructors on classes..

Classes can be declared in two main ways: via UML style class diagrams or via code.

Here is an example of a class diagram:



Here is the equivalent model represented as code:

```

parserImport Xocl;

context Root
    @Package Components

    @Class ComponentModel

        @Attribute components : Set(Component) (?,!)
        @Attribute connectors : Set(Connector) (?,!)

    end

    @Class Component extends NamedElement

        @Attribute owner : ComponentModel (?,!)
        @Attribute ports : Set(Port) (?,!)

    end

    @Class NamedElement isAbstract

        @Attribute name : String (?,!)

    end

    @Class Connector

        @Attribute target : Port (?,!)
        @Attribute source : Port (?,!)

    end

    @Class Port extends NamedElement

        @Attribute type : String (?,!)

    end
  
```

```
    end
end
```

Using Xocl

In a class model, Xocl is used in the bodies of constraints, operations and constructors. The way that Xocl is added depends on the interface being used.

For details on how to use Xocl in a class diagram, see chapter Chapter 2, *Creating and Interacting with a Domain Model*.

For details on how to use Xocl in code, see chapter Chapter 8, *Using the Programming Interface*.

Xocl is also used in the body of XMap mappings, see chapter Chapter 6, *Constructing and Running Mappings*.

Context

The context of an Xocl expression is the model element whose instances will be referenced by the expression. In the case of an operation or constraint, this might be the class that owns the operation/constraint.

When writing code via the text editor, the context of an expression can be defined in two ways. Firstly, by embedding the operation or constraint in the body of the relevant code definition, for example as follows:

```
context Components
@Class Port extends NamedElement
  @Attribute type : String (?,!)
  @Operation setPortName(name : String)
    self.type := name
  end
end
```

Alternatively, the context keyword can be used to separate the definitions, e.g.

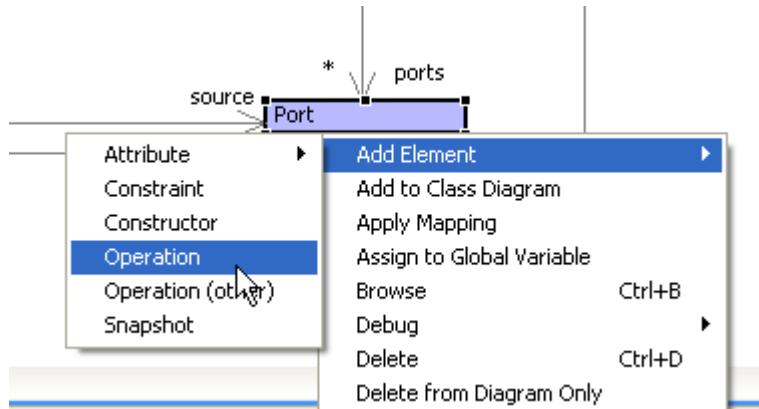
```
context Components
@Class Port extends NamedElement
  @Attribute type : String (?,!)
end

context Port
@Operation setPortName(name : String)
  self.type := name
end
```

Contexts are a useful way to separate different aspects of a model into different files.

In the diagram editor, context is assigned automatically depending on the model element being edited (see the creating a domain model walkthrough).

For example, if we want to add the operation setPortName() to the class Port, we would use the Add Element > Operation menu option on the class Port:



A new operation will be added to the class in the browser - the class Port is now the context of the operation.

Double clicking on the operation will launch the operation editor and the code for the operation can be entered:

```

Console Properties - setPortName() X
self      setPortName()
of        CompiledOperation
owner    Port

source
@Operation setPortName(name : String):XCore::Element
    self.type := name
end

```

From now on, we will assume that the code examples in this chapter can be added to their context as follows: ..

- Via the text editor (by explicitly writing their context or embedding them in the appropriate context element declaration)
- Via the diagram editor (by selecting the context element and using the appropriate Add Element menu)

Self

The variable "self" is a reserved word used to refer to instances of the context model element. In the case of the operation setPortName() above, self refers to an instance of the Port class.

Self is important when referencing the context object as a value, and is required when assigning values to a slot.

Operations

Operations capture the behaviour of classes. Operations can be added to classes either via the diagram editor, or by declaring them in code. Operations can change the state of objects, or simply return information without causing a state change (i.e. perform a query).

The syntax of an operation is as follows:

```
context model_element
  @Operation name(parameter_list):return_type
    body
  end
```

Here is an example of an operation that sets the name of a Port object:

```
context Port
  @Operation setPortName(name : String)
    self.type := name
  end
```

Operations have a context, which is the model element whose instances will be referenced by the operation. In the case of an operation added to a class in the diagram editor, this context is set automatically.

Constructors

Constructors are operations that are called when a class is instantiated. Constructors may have parameters that contain values that are passed to the attribute values (slots) of an object when it is created.

The syntax of a Constructor is as follows:

```
context model_element
  @Constructor(parameter_list)
    body
  end
```

The body of constructor is code that is run on initialisation of the object.

As an example, the following is a constructor for a Port. A port has two attributes, its name and type. By default, if the names of the parameters match the attribute names, their values will be assigned automatically when the class is instantiated. Note, any number of constructors can be created for a specific class - the constructor that is invoked is determined by the number of parameters passed.

```
context Port
  @Constructor(name,type)
  end
```

Here is an example where the body of the constructor is used to assign a value to a slot:

```
context Port
  @Constructor(name)
    self.type := name + "_Type"
  end
```

Note, the use of self in the assignment is required.

Constraints

Constraints are Xocl expressions which evaluate to true or false. Constraints are typically added to classes.

The syntax of a constraint is as follows:

```
context model_element
  @Constraint name
    boolean_expression
```

```
end
```

Constraints have a context and a name. The body of a constraint should be a boolean expression.

The following constraint states that the source and target ports of a connector should have the same type.

```
context Connector
  @Constraint SamePortType
    source.type = target.type
  end
```

Variables

Variables can be declared in a number of different ways in XOCL:

- As attributes on classes (and packages)
- As local variables
- Using a binding (not described here)

We have already seen attribute declarations.

Local variables are declared using the let expression. The syntax of a let expression is as follows:

```
let var_name_1 = expression_1;
  var_name_2 = expression _2
in
  body
end
```

Variable declarations are separated by semicolons. They can be referenced by any expression in the body of the let expression. As an example, the following let expression introduces three variables x,y,z into an expression:

```
let x = 1;
  y = 2;
  z = 3
in
  x + y + z
end
```

Variable declarations cannot normally reference each other unless they are in the body of another let expression. However, the "then" expression provides a convenient way of permitting this:

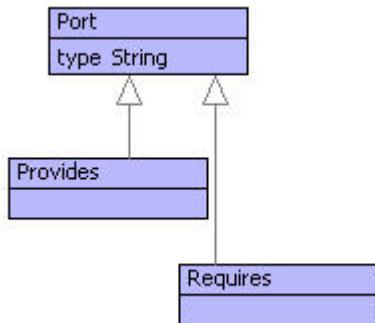
```
let x = 1;
  y = 2 then
    c = x + y
in
  c + z
end
```

Types

XOCL provides full access to the type of an object via the `of()` operation. For instance, if the object `o` is of type `Connector`, then `o.of()` will return the class `Connector`.

Because XOCL treats all elements as objects, the `of()` operation can be used to determine the type of all model elements. For example, `Component.of()` will return the class `Class` because a `Component` class is an instance of the class `Class`.

XOCL also provides facilities for testing type conformance. If we extend the Components model by specialising the class Port like so:



the operation **isKindOf()** can be used to determine the type of a Port object.

If p is a port, then the expression

```
p.isKindOf(Provides)
```

will return true if p is an instance of the class Provides.

Navigation

The dot notation “.” is used to navigate down attribute values. It has the form:

```
obj.attribute_name
```

The result of the navigation depends on the type of the attribute. If the attribute type is a Set, then a set of elements will be returned. If the type is a Sequence, a sequence of elements will be returned.

In the case of a class model, this means that navigating down attributes and association ends of multiplicity * and {ordered} will return a set or a sequence of objects respectively. For example, the following operation will return the result of navigating down the components attribute of a ComponentModel, and will return a set of components. Note the use of self is optional when navigating down attributes and associations.

```

context ComponentModel
  @Operation components()
    self.components
  end
  
```

The dot notation can be applied multiple times. In XOCL, the result of navigating down multiple attributes whose types are sets or sequences is not sets of sequences, etc, but the result of flattening the resultant values. For example, the following expression will return the set of Ports associated with all the components belonging to the component model:

```

context ComponentModel
  @Operation components()
    self.components.ports
  end
  
```

Collections

XOCL provides two main types for managing collections of elements: Sets and Sequences.

The value of a Sequence and Set are represented by the following literals:

```
Set{v1,v2,...} // The set containing the elements v1, v2, ...
Seq{v1,v2,...} // The sequence containing the ordered elements v1, v2, ...
```

As an example, try entering the following into the console:

```
Set{"bill","ed","hilda"};
Seq{1,2,3,4};
```

Sets and Sequences can also contain objects and other values, including other sets, sequences, etc.

Collection Operations

XOCL provides a number of OCL based operations for dealing with collections of objects. These are important when dealing with the results of navigating over models.

The three primary operations are select, collect and iterate.

XOCL uses the "->" notation to denote the application of an operation on a collection, e.g. collection->select(c | expression)

Select

Select filters a collection. It has the following syntax:

```
collection->select(var | expression_with_var)
```

Selects just those elements of the collection that satisfy the expression. The variable var is bound to each element in turn.

The following operation makes use of select to return only those connectors whose source and target ports don't match:

```
context ComponentModel
@Operation dontMatch()
  connectors->select(c |
    c.source.type <> c.target.type)
end
```

Collect

Collect builds a collection of values. It has the following syntax:

```
collection->collect(var | expression_with_var)
```

Collects the result of evaluating the expression. The variable var is bound to each element in turn.

The following operation makes use of collect to return the set of port names associated with all components:

```
context ComponentModel
@Operation portNames()
  components.ports->collect(p | p.name())
end
```

Iterate

Accumulate a collection of values. It has the following syntax:

```
collection->iterate(var acc=expression | expression_with_var)
```

Accumulates the result of evaluating an expression in the variable acc. The variable var is bound to each element in turn. The initial value of acc is expression.

The following operation makes use of iterate to accumulate the total number of ports of type t:

```
context ComponentModel
@Operation numberOfPorts(t:String)
    components->iterate(c tot=0 |
        tot + (c.ports->select(p | p.type = t)->size))
    end
```

Other Operations

XOCL has many other useful collection operations. Here are a few of the most commonly used.

```
collection->size() // Returns the number of elements in the collection
collection->includes(x) // Returns true if the collection includes x
collection->including(x) // The result of including x in the collection
```

Some operations relate specifically to Sequences or Sets. The following are some common Sequence operations:

```
sequence->head() // Returns the value at the head of the sequence
sequence->tail() // Returns the tail of the sequence
sequence + sequence // The concatenation of two sequences
sequence->asSet // Turns a sequence into a set
```

Logical Expressions

XOCL provides all the usual Boolean expressions including: and, or, not and equals.

In addition, it provides "exists" and "forAll" for evaluating Boolean expressions over collections. Their syntax is as follows:

```
collection->forAll(c | boolean_expression_with_c) // returns true if the expression
                                                    // for all elements of the collection
collection->exists(c | boolean_expression_with_c) // returns true if the expression
                                                    // for at least one element of the collection
```

As an example, the following operation returns true if all connectors match ports of the same type:

```
context ComponentModel
@Operation validConnections : Boolean
    connections->forAll(c | c.source.type = c.target.type)
end
```

The following operation returns true if there is at least one invalid connection:

```
context ComponentModel
@Operation invalidConnection : Boolean
    connections->exists(c | c.source.type <> c.target.type)
end
```

Conditional Expressions

The standard if and case expressions are provided in XOCL.

The syntax of an "if" expression is shown below. Note, both the else and elseif expressions are optional.

```
if condition then
    expression
elseif condition
    expression
else
    expression
end
```

Here is an example that makes use of an "if" expression to determine whether a new component can be added to a component model.

```
context ComponentModel
@Operation addToComponents(v : Component) : XCore::Element
    if components->exists(c |
        c.name = v.name)
    then
        self.error("Cant add a component with the same name as an existing component")
    else
        self.components := self.components->including(v)
    end
end
```

The syntax of a "case" expression is shown below.

```
@Case var of
    [ pattern1 ] do expression end
    [ pattern2 ] do expression end
end
```

Here is an example that makes use of a "case" expression to determine the type of a Port and to print out a report.

```
context Port
@Operation testPort()
let p = self.type
in @Case p of
    [ "Integer" ] do
        "An Integer Port".println()
    end
    [ "Complex" ] do
        "A Complex Port".println()
    end
end
end
```

An alternative form of a case expression is a @TypeCase. This selects a do expression depending on the type of the element being passed to it. For example, the following code will return a different string depending on the type of element it is supplied.

```
@TypeCase(element)
Component do "Component" end
Port do "Port" end
else "Failed"
end
```

Imperative Features

XOCL extends OCL with a number of imperative programming features, which turn it into a powerful programming language.

Object Creation

Objects are instances of classes. Objects are created by calling a class constructor. The syntax is as follows:

```
class_name(parameter_values)
```

Assumng that the Components model example is in scope, the following can be typed into the console:

```
Port();
```

This will create an instance of the class Port using the default (empty) constructor.

Note, if we define a constructor for the class then we could call this by passing the appropriate parameter values. For example, if the following constructor has been created:

```
context Port
  @Constructor(name, type)
end
```

We can pass the name and type of the Port to the instance like so:

```
Port("x", "Integer");
```

An alternative instantiation syntax is also provided, which enables values to be bound to slots using name bindings. The following binds the values to the slots by their name:

```
Port[name = "x", type = "Integer"];
```

Assignment

Values can be assigned to attributes (slot update) using the assignment operator.

```
var := expression // assigns the result of evaluating the expression to var
```

The following operation gives an example of setting the owner of a Component to be the ComponentModel, c.

```
context Component
  @Operation setOwner(c : ComponentModel)
    self.owner := c
  end
```

Note, that self is required when setting slot values.

A common assignment pattern is adding a new element to a collection. This can be easily achieved using the including() operation:

```
context ComponentModel
  @Operation addToComponents(v : Component)
    self.components := self.components->including(v)
  end
```

Sequential Execution

A sequential operator (";") is provided for sequencing different operations. For example, adding a component to a component model and setting it to be owned by the component model.

```
context ComponentModel
  @Operation addToComponents(c : Component)
    self.components := self.components->including(v);
```

```
c.owner := self  
end
```

Operation Invocation

The dot notation (".") is used to invoke operations on objects. The following operation invokes the setOwner() operation on the component, c.

```
context ComponentModel  
    @Operation addToComponents(c : Component)  
        self.components := self.components->including(v);  
        c.setOwner(self)  
    end
```

Looping

XOCL provides a number of looping constructs, including while loops and for loops.

The syntax for a "for" loop is as follows:

```
@For var in collection do  
    expression  
end
```

The reAssign() operation uses a for loop to reassign any connections that target the Port, old, to the Port, new.

```
context ComponentModel  
    @Operation reAssign(old:Port,new:Port)  
        @For c in connectors do  
            if c.target = old then  
                c.target := new  
            end  
        end  
    end
```

The syntax for a "while" loop is as follows:

```
@While condition do  
    expression  
end
```

Here is an example of a simple counter that makes use of a while loop.

```
context Root  
    @Operation count()  
        let x = 0  
        in @While x < 1000000 do  
            x := x + 1  
        end;  
        x.println()  
    end  
end
```

Exceptions

XOCL provides the operation **error()** to raise an error which can be caught by the XMF-Mosaic Virtual Machine. Here is an example of its use to raise an error when an invalid Component is about to be added to a ComponentModel.

```

context ComponentModel
@Operation addToComponents(v : Component):XCore::Element
    if components->exists(c |
        c.name = v.name)
    then
        self.error("Cant add a component with the same name as an existing component")
    else
        self.components := self.components->including(v)
    end
end

```

XOCL also provides a **try catch** mechanism for handling exceptions. For full details see the XOCL guide in the Bluebook.

Formatting

XOCL provides a powerful facility called "format" for formatting and outputting data to a variety of output streams. The general form a format expression is:

```
format(OUTPUT,CONTROLSTRING,SEQOFARGS)
```

where

- OUTPUT is an output channel (often stdout).
- CONTROLSTRING is a string of chars and control chars (see below).
- SEQOFARGS is a sequence of args consumed by the control string. This is optional if no args are required by the control string.

The control string a little program. The simplest form is just a string containing no control characters:

```
format(stdout,"Hello world")
```

sends the chars to the output channel. Control characters start with a ~, for example:

```
format(stdout,"Hello~%World~%")
```

prints a newline (~% meaning print a newline char) after each word. Control characters can consume args:

```
format(stdout,"Hello ~S, how are you ~S~%",Seq{ "Fred" , "diddling" })
```

the ~S control consumes the next arg, turns it to a string using .toString() and then prints it. Some of the control characters take args:

```
format(stdout,"Hello~<?,x>S~%",Seq{10 , "Fred" })
```

prints Fred in a column width of 10 characters padding the extra spaces out with 'x' like so:

```
HelloFredxxxxxx
```

A useful control char is ~{ which is used to loop through a sequence:

```
format(stdout,"~{~S~%~} ",Seq{names})
```

prints each element of the sequence names on a new line. The ~{ can include separator chars:

```
format(stdout,"~{,~%~;~S~} ",Seq{names})
```

prints each element iof the sequence names with a ',' followed by a new line char between each pair. The ~; is used to terminate the separator. This is particularly useful when generating comma separated code in a programming language.

```
format(stdout, "~S~^~S", Seq{1,2,3});
```

The ~^ control character enables you to repeat a print (~S) statement. For example, the above prints, 1 followed by 1. This is useful when printing a string at both the start and end of an expression, e.g. proc A end A.

Advanced Features

XOCL provides a number of powerful programming features which for brevity are not described here. These include:

- Pattern matching.
- Continuations.
- An extensible grammar (the @For and @Case constructs are examples of extensions to the language).

See the XOCL guide for more information.

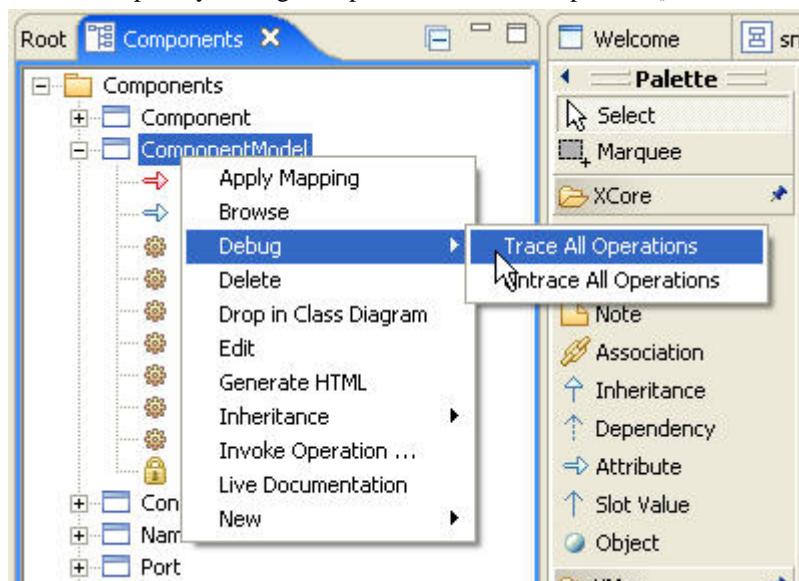
Chapter 5. Debugging Operations

Operations can be debugged by setting a trace on them. Calls to the operation will be displayed in the console.

A trace can be set in a number of ways:

- By right clicking on an operation and selecting Trace from its Debug menu.
- By right clicking on a container, e.g. a Class, and selecting Trace from its Debug menu. All operations belonging to the container (and transitively belonging to all its containers) will be traced.
- Via the console, by typing the path to the operation or container and calling trace(). Note, calling untrace() will stop the trace.

As an example, try tracing the operation, addToComponents() on the ComponentModel class:



Running the operation will print the result of tracing the operation:

```
[1] XMF> ?i + Components
true
[1] XMF> A := ComponentModel();
<NameSpace Root>
[1] XMF> A.addToComponents(Component());
Enter <ComponentModel 6d5557>.addToComponents(<Component 6e0a47>)
Exit <ComponentModel 6d5557>.addToComponents = <ComponentModel 6d5557>
<ComponentModel 6d5557>
[1] XMF>
```

Note, you should not run trace on System classes, e.g. XCore or Root.

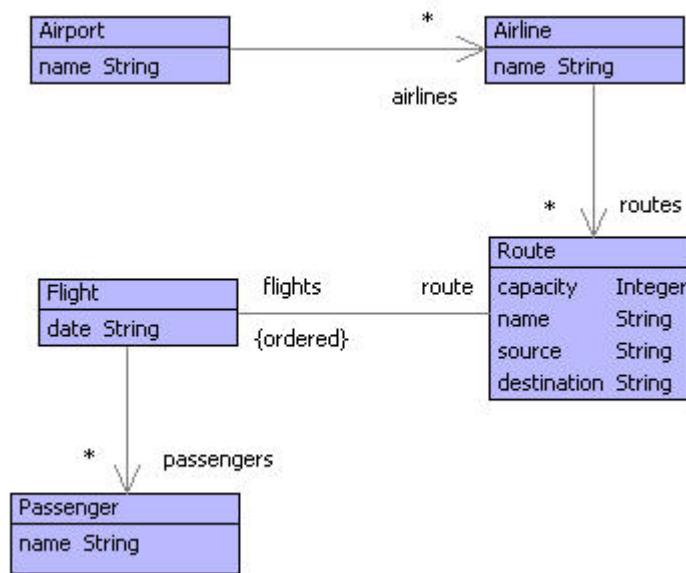
Chapter 6. Constructing and Running Mappings

Mappings are an essential part of being able to transform models written in one language to models written in another language. This part describes how to construct model to model mappings in XMap: a declarative pattern matching mapping language. Full details of the XMap language can be found in part 3 of the Bluebook.

A Simple Mapping

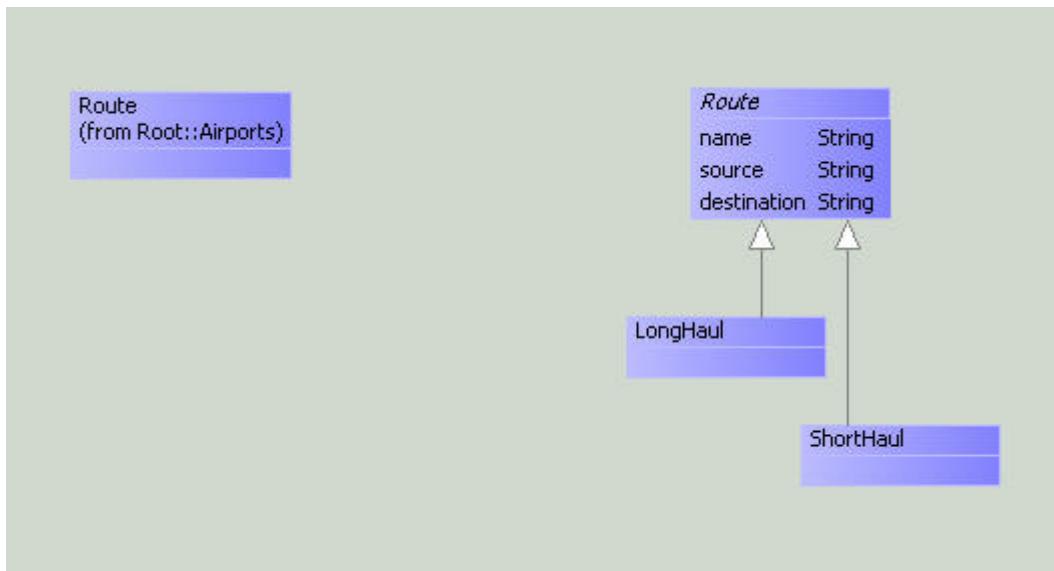
We want to create a simple mapping from one class to another class.

The source model is a simple domain model of Airports:



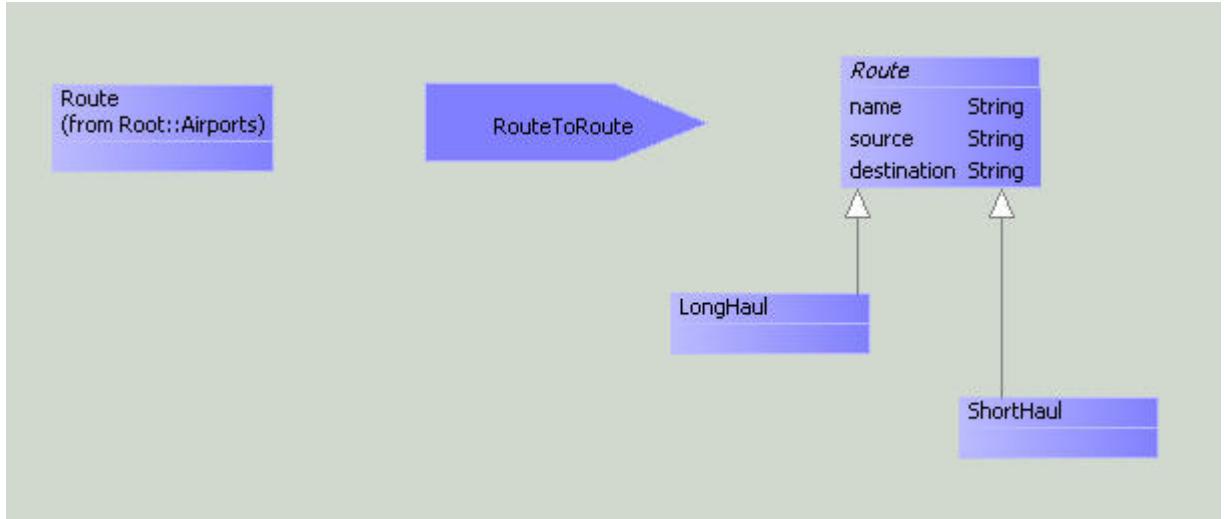
Create or load the Airports model.

Next, create a new project Example1 containing the classes that we want to map between.



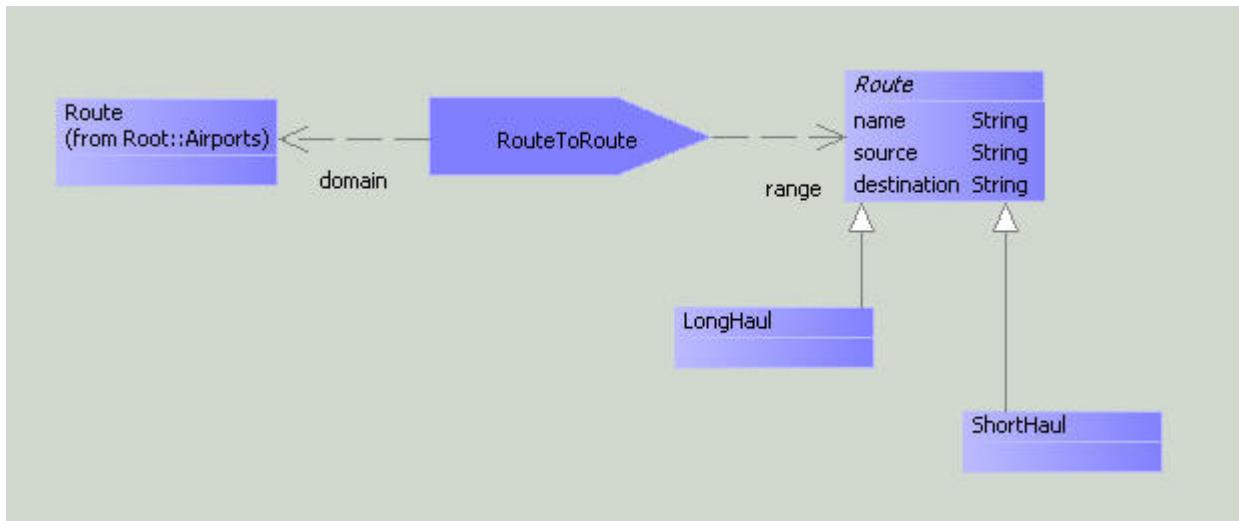
Note, the Route class has been copied from the Airports model. This is achieved by right clicking on the class in the Airports model, selecting Drop in Class Diagram and then choosing the Example1 package.

Next, a mapping is added to the model. Select the Mapping from the icons in the diagram editor and place it on the diagram and give it a name.



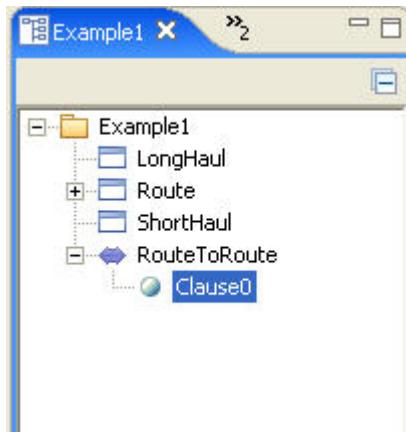
A mapping must have at least one domain and a single range.

Select these from the icons underneath the mapping icon.



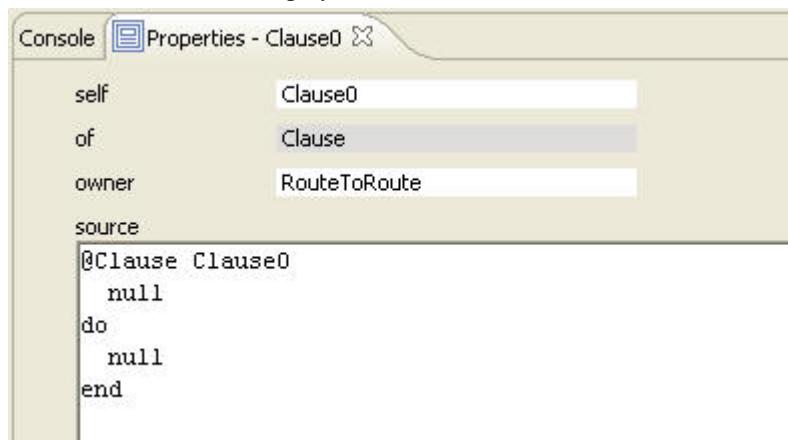
The domain is the input to the mapping and the range is its output. Although compulsory, it is important to think of the domain and range as indicating a dependency between a mapping and its constituents. In practice, the actual values passed to the mapping need not necessarily be of the domain type. Similarly, the range need not be of the target type.

The rules and actions performed by a mapping are described by *clauses*. Add a new clause to a mapping by right clicking on the mapping in the diagram editor or browser and choosing New > Clause.



Now, edit the clause by double clicking on it in the browser.

A clause editor will be displayed.



A clause has a name, and a default clause expression followed by a “do” action.

The expression is a pattern that must be matched before the clause will fire.

When the clause is matched, the action after the 'do' will be performed.

We wish to capture the rule that a Route with a capacity less than 250 will be mapped to a ShortHaul Route with the same name. In addition, we want to map a Route with a capacity ≥ 250 to a LongHaul Route.

To achieve this we need to write two clauses. The first clause matches with a Route whose capacity is less than 250 and generates a Short Haul Route.

```
@Clause Short
Airports::Route[
    name = N,
    source = S,
    destination = D,
    capacity = C]
when C < 250
do
    ShortHaul[
        name = N,
        source = S,
        destination = D]
end
```

Here:

- The name of the clause has been changed to Short
- The clause expression is a Route, whose name equals the variable 'N', whose source and destination equals the variables 'S' and 'D' and whose capacity equals the variable C.
- The when condition states that this clause can only fire if the capacity is < 250.
- The 'do' action creates a ShortHaul Route whose name is equal to the variable 'N', and which has the same source and destination.

Note because the variables are global they are matched across the clause, ensuring for example that the name of the source Route matches the target Route name.

To enter the clause, enter the text and right click > Commit it.

To deal with the other possibility (that the capacity ≥ 250) we need to create another clause and add it to the mapping.

Again, right click on the mapping in the diagram editor or browser and choose New > Clause. Edit and commit the clause as before.

```
@Clause Long
Airports::Route[
    name = N,
    source = S,
    destination = D,
    capacity = C]
when C >= 250
do
    LongHaul[
        name = N,
        source = S,
        destination = D]
end
```

The code is similar to that of the previous clause, but the “when” condition requires that the capacity of the route is ≥ 250 . The resulting action creates an instance of a LongHaul Route.

Running the Mapping

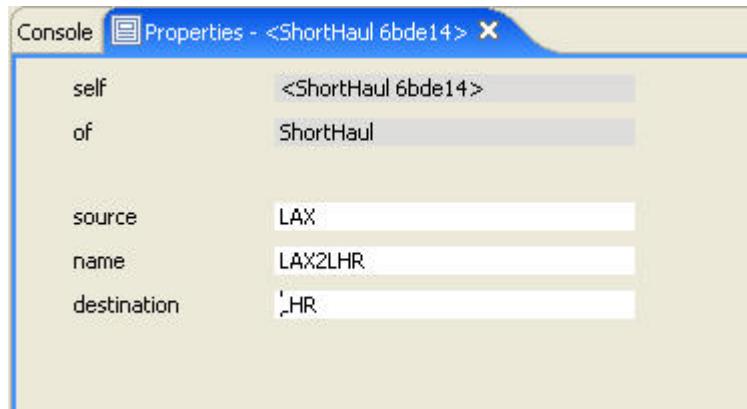
Mappings can be run in two ways: via the console or by passing the object to the mapping.

Here's an example of running a mapping via the console. Let's create an instance of a Route, x, named "LAX2LHR", with a capacity of 200 and source and destination "LAX" and "LHR" respectively..

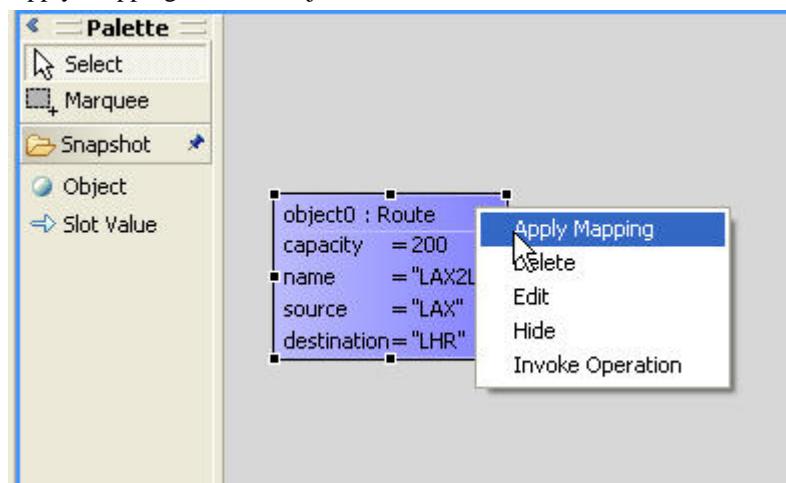
The mapping can be run by calling the mapping and passing it the Route, x.

```
[1] XMF> x := Airports::Route[name = "LAX2LHR", capacity = 200, source = "LAX",
    "LHR"];
<NameSpace Root>
[1] XMF> Example1::RouteToRoute() (x);
<ShortHaul 6b815c>
[1] XMF> |
```

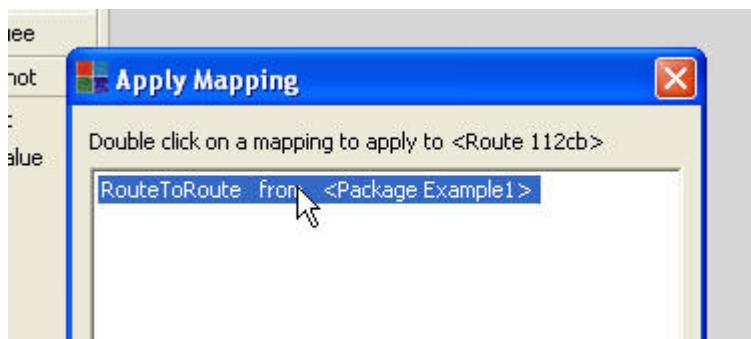
The result of the mapping is a ShortHaul Route whose name is "LAX2LHR" as we would expect.



The same result can be achieved by creating the same instance of a Route in a Snapshot and selecting Apply Mapping from the object's menu:



A choice of available mappings will be displayed. Select the mapping that the object is to be passed to, and it will be invoked.

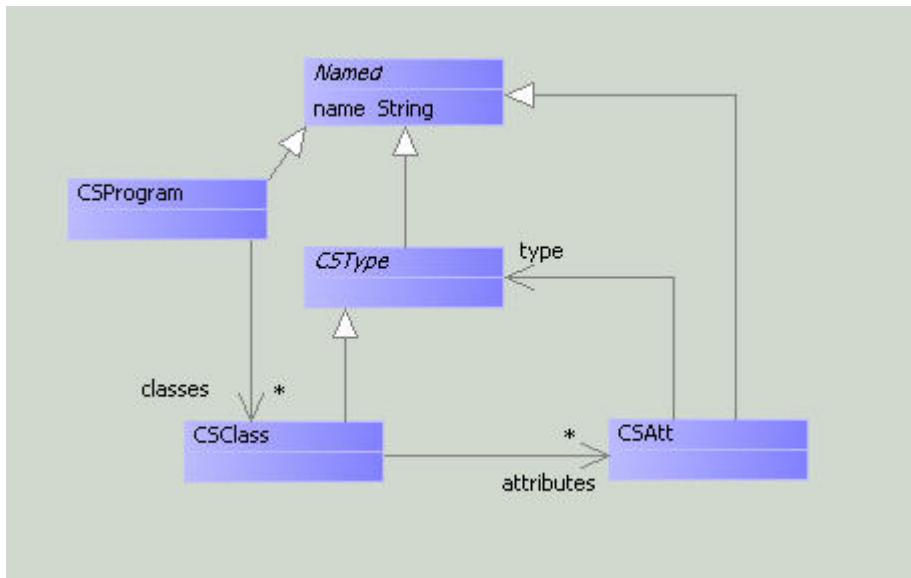


A Simple Model to C# Mapping

This section describes the construction of a mapping between model (XCore) packages and classes to a small model of the C# programming language.

The C# Model

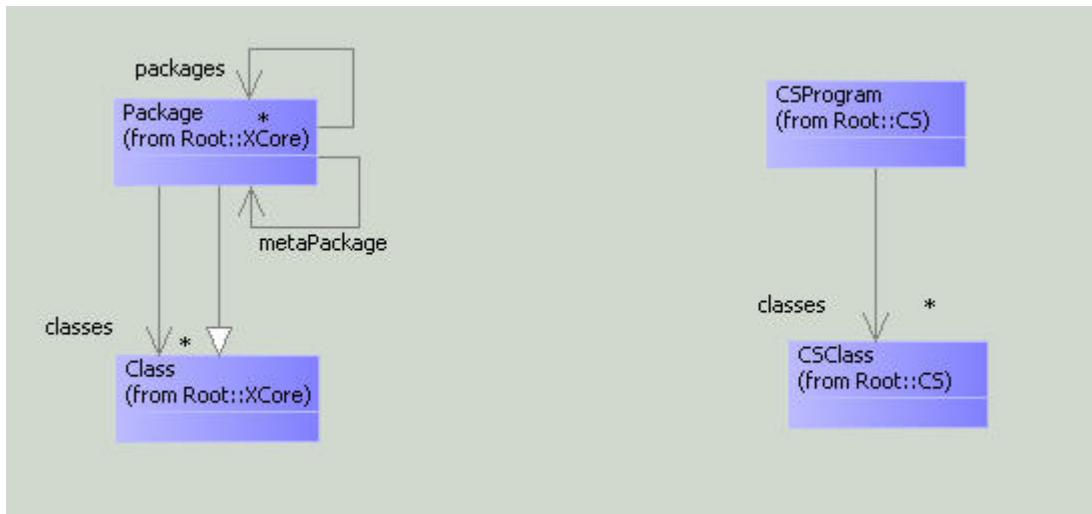
The following is a simple model of the abstract syntax of C# that will be used in the mapping. A C# model consists of a program containing a number of classes, which have attributes.



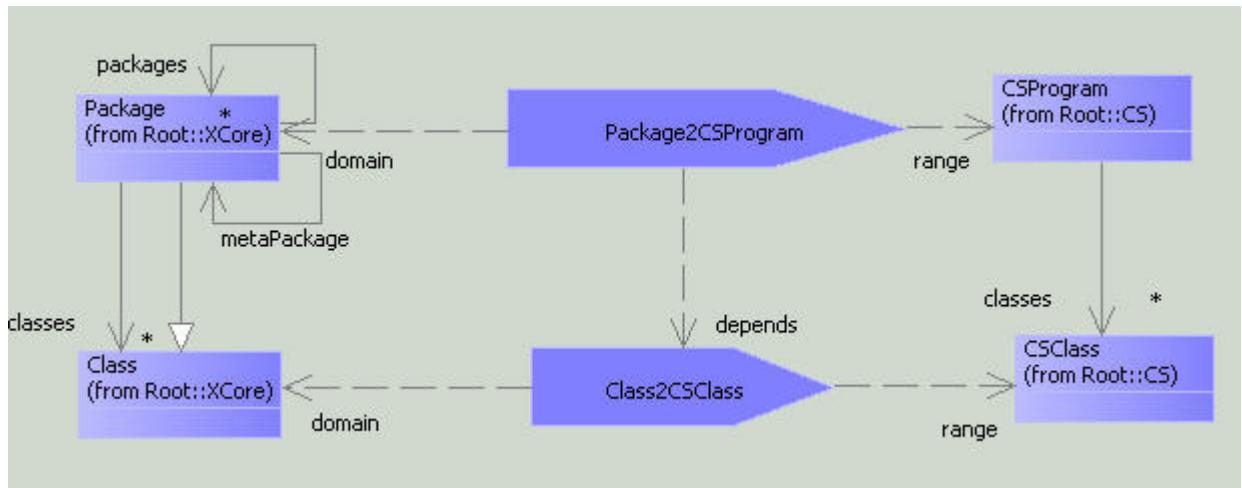
The Mapping

We'll construct two mappings. The first mapping (Package2CSProgram) turns a Package into a C# Program . This mapping then calls the Class2CSClass mapping, which iterates over the classes in the Package and maps them to CSClasses. Of course, further mappings can be constructed to map Class operations and attributes, etc, but we won't deal with those here.

First, we need to import the Package and Class classes from the XCore package. To do this, open the Root::Kernel::XCore package in the browser. Select the classes Package and Class and then select Drop in Class Diagram to drop them into the mapping package. Note, we also need to import the C# classes CSProgram and CSClass by dropping them into the model from the CS model.



Next, the mappings are added. Remember that the domain and range arrows are added by selecting the domain and range buttons respectively. A dependency arrow is also added to indicate that the Package2CSProgram mapping is dependent on the Class2CSClass mapping.



Let's add the mapping clause by starting with the simplest mapping. The Class2CSClass mapping transforms an instance of a Class into an instance of a CSClass.

Its clause says that whenever the mapping is given a Class it returns a CSClass with the same name.

```

@Clause Clause0
    XCore::Class[name = N]
do
    CS::CSClass[name = N]
end

```

The Package2CSProgram maps a Package to a CSProgram. In doing so, it maps each class in the package to a CSClass by calling the Class2CSClass mapping.

```

@Clause Clause0
    XCore::Package[name = N, classes = C]
do
    CS::CSProgram[name = N, classes = Cs]
where
    Cs = C->collect(c | Class2CSClass()(c))
end

```

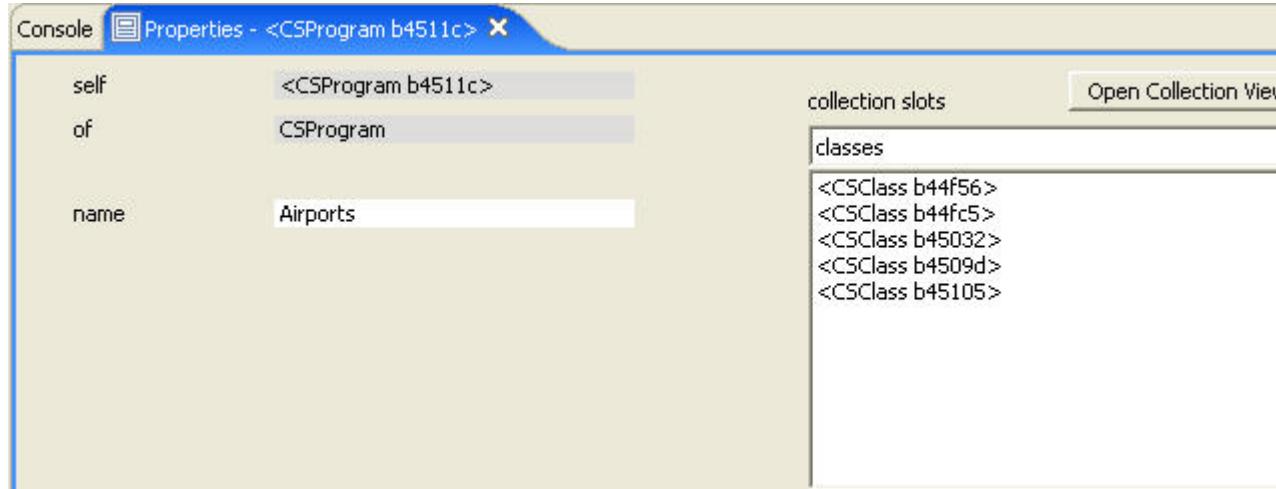
Executing the Mapping

Just as in the previous example, the mapping can be executed via the console.

Because we are mapping from XCore packages, we can actually pass a package from a model into the mapping and see what it produces. Let's map the Airports package

```
[1] XMF> ClassMapping::Package2CSProgram()(Airports);
<CSProgram ac7c5c>
```

As expected, the mapping has generated a C# Program. If we edit it, we can see it has produced a CSClass for each class in the Airports package.

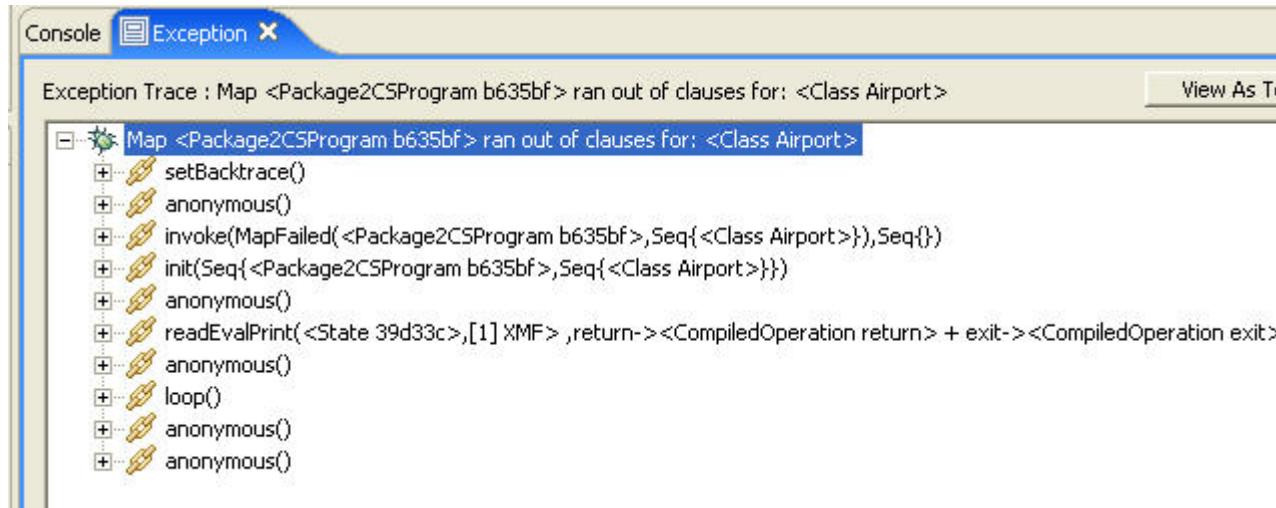


Mapping Hints and Tips

Error Reporting

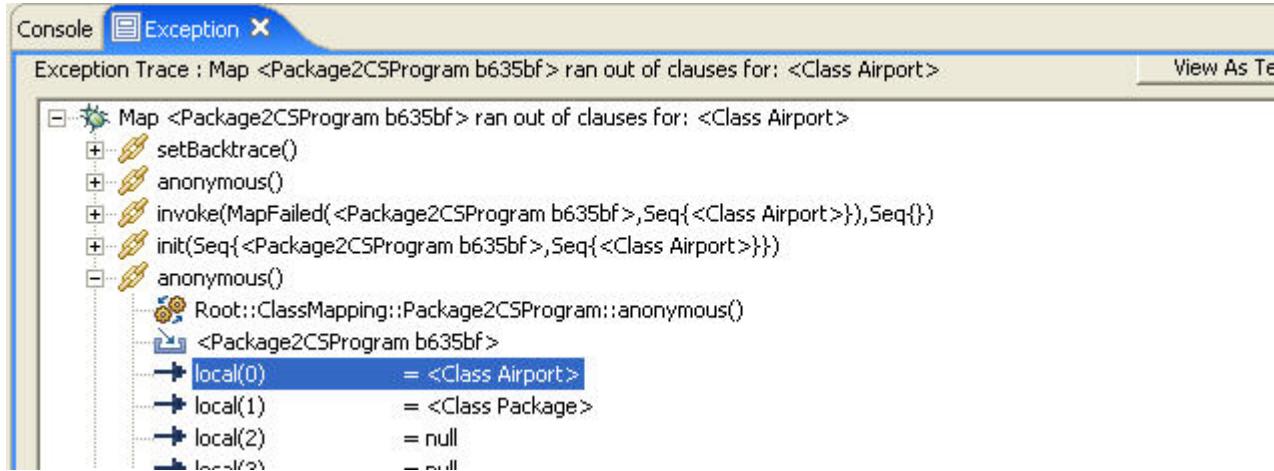
When running a mapping, all the usual exception reports will be displayed if for example, the mapping attempts to set an invalid slot or create an invalid instance. In addition, a MapFailed exception is raised if mapping cannot find a clause that matches the input to the mapping.

As an example, try running the mapping by passing an invalid object, for instance a Class to the Package2CSProgram mapping. Because the Package2CSProgram mapping is expecting a Package it will fail and generate the following exception.



Additional information on the cause of the exception can be found by expanding the tree nodes.

Here, the value being passed to the mapping can be seen in local(0).



Clearly, it is of the wrong type.

If we wanted to examine the passed object in more detail it can be navigated to by double clicking on local(0).

Constructing a Pretty Printer

In the previous section, a mapping from packages and classes to C# was constructed. In practice however, the objective will be to generate concrete code (not just an instance of the C# model). For example, we might expect to generate some code of the following form:

```
using System;

class Route
{
    integer capacity;
    string name;
    string source;
    string destination;
}
class Passenger
{
    string name;
}
class Airline
{
    string name;
}
class Airport
{
    string name;
}
class Flight
{
    string date;
}
```

The basic approach is to add operations to the language model to pretty print it. The first step is to load your language model and plan where the appropriate operations need to be added. In general, it is best to start at the leaf classes where there is minimum dependency on other elements in the model.

As an example, a pretty print operation can be added to the class CPPAtt (a C# attribute). First, add a pretty print operation to the class and give it a name, e.g. pprint().

Now, edit the operation. Two parameters are required by the operation: an output type, out, which will be used to direct the output of the operation to an editor or to a file, and an indent, which is the current indent of the code.

We can now construct the body of the operation. An attribute will be pretty printed as the type name and the attribute name, terminated with a semicolon.

```
@Operation pprint(out : XCore::Element, indent : XCore::Element):XCore::Element
    format(out, "~V~S~%", Seq{indent, type.name + " " + name + ";"})
end
```

The format command is used to generate the text. The sequence of control characters dictates the format of elements that occur in the sequence of elements that are to be formatted. For example, ~V denotes a vertical space, ~S is a string and ~% is a line return. Note, full details of the format command can be found in Part 3 of this guide.

In this case an indent is inserted before the string is printed, and a character return is inserted after the semicolon.

The definition of pprint on CSClass is as follows:

```
@Operation pprint(out : XCore::Element, indent : XCore::Element):XCore::Element
    format(out, "~V~S~%", Seq{indent, "class " + name + " ", indent, "{"));
    attributes->collect(a |
        a pprint(out, indent + 2));
    format(out, "~V~S~%", Seq{indent, "}"})
end
```

The class signature and name of the class is printed followed by the attributes and operations of the class.

Finally, we need to construct the pretty printer for a CSProgram.

```
@Operation pprint(out : XCore::Element, indent : XCore::Element):XCore::Element
    format(out, "~V~S~%", Seq{indent, "using System;"});
    classes->collect(c |
        a pprint(out, indent + 2))
end
```

If we run the pprint operation on the result of mapping the Airports package, the following code is generated:

```
using System;

class Route
{
}
class Passenger
{
}
class Airline
{
}
class Airport
{
}
class Flight
{
}
```

A lot more needs to be done to complete the mapping. In particular mappings from Attribute and Operation to C# Attributes and Operations need to be defined. However, it is simply a case of following the same overall approach.

Chapter 7. Constructing an XML Parser and Generator

It is often useful to be able to interchange models or programs as XML data. For example, there may be a third party tool that does some pre- or post-processing tasks on the data, or we may want to maintain models and instances of models as XML files.

This section describes how an XML parser and generator can be constructed in XMFMosaic enabling any format of XML to be parsed and exported to a from an instance of a model. It is based on (XXML) a generic, high-level grammar language that can be used to capture XML grammars and the rules for mapping them to models.

The following sections give an example of its application to the Components model, enabling instances of the model to be represented as and parsed as XML.

First Steps

The first step in constructing an XML grammar for a language is to create a file to contain the grammar definition. This can be done via the file browser (see Part 1 of this guide).

Open the file browser by right clicking on a project in the browser and selecting Create File Browser.

Select the directory that the file will be created in, and create the file, e.g. AircraftsGrammar.xmf.

Constructing the Grammar

First we need to add the relevant imports to the file so that it can be compiled and loaded by XMFMosaic. We import the parsers for XOCL and XML.

We import the Components and XML packages.

```
parserImport XOCL;
parserImport XML::Parser;

import Components;
import XML::Parser; // Get ParserChannel and Ref
```

A grammar can now be constructed using XXML.

In the context of the ComponentModel class, we declare a Grammar definition, called ComponentModel.

```
context ComponentModel
    @Grammar ComponentModel
```

Associated with the Grammar are a number of grammar rules which state how the XML should be parsed.

```
ComponentModel ::= 
    <ComponentModel>
        C = Component*
        N = Connector*
    </ComponentModel>
    { ComponentModel[components=C,connectors=N] } .
Component ::= 
    <Component n=name>
        P = Port*
    </Component>
```

```

{ Component[name=n,ports=P] }.
Port ::= 
    <Port n=name t=type i=id/>
    i := { Port[name=n,type=t] }.
Connector ::= 
    <Connector sid=source tid=target/>
    { Connector[source=Ref(sid),target=Ref(tid)] }.
end

```

Each grammar rule describes a rule for recognizing the next bit of XML. It checks the next element tag and binds the variables to the attribute values. The child elements must match the rules defined by the child grammars and binds the variables to the result. If the match is successful then the XOCL action occurs and returns a value.

The ComponentModel grammar is defined to be the XML tag <ComponentModel> followed by some child grammars (it has no attributes). The child grammars bind the result of parsing a sequence of Components followed by a sequence of Connectors.

The rule is terminated by the tag </ComponentModel>

Every time the grammar rule is parsed, its effect is to unify the variables with the values that are parsed.

These values are then used to perform an XOCL action, which is written in curly brackets after the grammar expression.

Here, the action instantiates the class ComponentModel with the variable values.

Similar grammar rules are written for the other elements in the model.

The grammar for Component is straightforward - it returns a Component instance.

The grammar for a Port illustrates the use of an "id" which is associated with each new Port instance.

The grammar for a Connector uses this "id" to look up the values of the source and target ports of the Connector. This is performed using the Ref() operation. This is how cross-references are managed in XXML.

Invoking the Parser

The following code provides operations for reading in XML files and parsing them using the parser.

```

context Components
@Operation parseFile()
let file = xmf.openFile(xmf.projDir(),"*.xml")
in if file <> ""
then
    @WithOpenFile(fout <- file)
    Components::parse(fout).edit()
end
else xmf.message("Parse Cancelled")
end
end
end

context Components
@Operation parse(inch:InputChannel)
let grammar = ComponentModel::ComponentModel.compile()
in let xin = ParserChannel(inch,grammar)
in xin.debug := true;
xin.parse("ComponentModel");

```

```

        // Get the result and resolve the references.
        xin.result(true)
    end
end
end

```

The package operation `parseFile()` first opens a browser to select the XML to be imported. If the file exists. This calls the `parse()` operation, which opens an `XMLInputChannel` and the XML is read into variable `xin`. The result (an instance of the `ComponentsModel`) is returned.

Example

As an example we can construct a small XML file such as below and save it in an XML file, e.g `example.xml`.

```

<ComponentModel>
    <Component name="Displays">
        <Port id="Displays:CurrentPosition" type="Long" name="CurrentPosition" />
    </Component>
    <Component name="Navigation">
        <Port id="Navigation:CurrentPosition" type="LatLong" name="CurrentPosition" />
    </Component>
    <Connector target="Navigation:CurrentPosition" source="Displays:CurrentPosition" />
</ComponentModel>

```

To run the parser, run the `parseFile()` package operation from the console by typing `Components::parseFile();`

The result will be to parse the XML and create an instance of the `Components` model:



Debugging the Parser

If `debug` has been set to true for a grammar, the console will display a trace of pattern matches that have taken place during the parsing process:

```

Components::parseFile();
<ComponentModel>
    <Component name='Displays'>
        BIND n = Displays
        <Port id='Displays:CurrentPosition' type='Long' name='CurrentPosition'>
            BIND n = CurrentPosition
            BIND t = Long
            BIND i = Displays:CurrentPosition
        </Port>
        PUSH(<Port d69cc1>)
    </Component>
</ComponentModel>

```

```

        BIND x = <Port d69cc1>
    </Component>
    PUSH(Seq{})
    BIND xs = Seq{}
    PUSH(Seq{})
    PUSH(<Port d69cc1>)
    PUSH(Seq{<Port d69cc1>})
    BIND P = Seq{<Port d69cc1>}
    PUSH(Seq{<Port d69cc1>})
    BIND P = Seq{<Port d69cc1>}
    PUSH(<Component d6f173>)
    BIND x = <Component d6f173>
    <Component name='Navigation'>
        BIND n = Navigation
        <Port id='Navigation:CurrentPosition' type='LatLong' name='CurrentPosition'>
            BIND n = CurrentPosition
            BIND t = LatLong
            BIND i = Navigation:CurrentPosition
        </Port>
        PUSH(<Port d76f09>)
        BIND x = <Port d76f09>
    </Component>
    PUSH(Seq{})
    BIND xs = Seq{}
    PUSH(Seq{})
    PUSH(<Port d76f09>)
    PUSH(Seq{<Port d76f09>})
    BIND P = Seq{<Port d76f09>}
    PUSH(Seq{<Port d76f09>})
    BIND P = Seq{<Port d76f09>}
    PUSH(<Component d7c2d5>)
    BIND x = <Component d7c2d5>
    <Connector target='Navigation:CurrentPosition' source='Displays:CurrentPosition'>
        PUSH(Seq{})
        BIND xs = Seq{}
        PUSH(Seq{})
        PUSH(<Component d7c2d5>)
        PUSH(Seq{<Component d7c2d5>})
        BIND xs = Seq{<Component d7c2d5>}
        PUSH(Seq{<Component d7c2d5>})
        PUSH(<Component d6f173>)
        PUSH(Seq{<Component d6f173>,<Component d7c2d5>})
        BIND C = Seq{<Component d6f173>,<Component d7c2d5>}
        BIND sid = Displays:CurrentPosition
        BIND tid = Navigation:CurrentPosition
    </Connector>
    PUSH(<Connector d86f69>)
    BIND x = <Connector d86f69>
</ComponentModel>
PUSH(Seq{})
BIND xs = Seq{}
PUSH(Seq{})
PUSH(<Connector d86f69>)
PUSH(Seq{<Connector d86f69>})
BIND N = Seq{<Connector d86f69>}
PUSH(Seq{<Connector d86f69>})
PUSH(Seq{<Component d6f173>,<Component d7c2d5>})
BIND C = Seq{<Component d6f173>,<Component d7c2d5>}
```

```
BIND N = Seq{<Connector d86f69>}
PUSH(<ComponentModel d8a323>)
```

The parser starts at the root of the grammar tree and calls each element in turn. Whenever an XML element is encountered that matches a grammar element, the values of variables are bound with the attributes of the element. Once successfully bound, the action associated with the grammar expression is performed to create the appropriate model instance/s.

Once you are happy with the grammar, the debug option should be switched off for efficient execution of the parser.

Generating XML

A similar approach can be used to construct an XML generator. A simple domain specific language is used to capture the rules for generating XML. Here are the rules for generating XML for instances of the Component model:

```
context ComponentModel
@Operation exportXML(out:OutputChannel)
@XML(out)
<ComponentModel>
    @For component in components do
        component.exportXML(out)
    end;
    @For connector in connectors do
        connector.exportXML(out, self)
    end
</ComponentModel>
end
end

context ComponentModel
@Operation portId(port:Port):String
@Find(component,components)
    when component.ports->includes(port)
    do component.name + ":" + port.name
    else self.error("Cannot find port " + port.toString())
end
end

context Component
@Operation exportXML(out:OutputChannel)
@XML(out)
<Component name=name>
    @For port in ports do
        port.exportXML(out, self)
    end
</Component>
end
end

context Port
@Operation exportXML(out:OutputChannel,component:Component)
let portId = component.name + ":" + name
in @XML(out)
    <Port name=name type=type id=portId/>
end
end
```

```

    end

context Connector
@Operation exportXML(out:OutputChannel,model:ComponentModel)
let sourceId = model.portId(source);
targetId = model.portId(target)
in @XML(out)
    <Connector source=sourceId target=targetId/>
end
end
end

```

The structure of each rule is as follows:

```

@XML(outputChannel)
<Tag name=exp name=exp ...>
    body
</Tag>
end

```

An @XML command writes the XML data to the supplied output channel. The XML elements and their attributes are then written. The body is XOCL code that writes the child elements.

As an example, the exportXML() operation generates a ComponentModel tag, followed by the results of calling the exportXML() operation on the ComponentModel's components and connectors.

To generate the XML, the following operation is defined:

```

parserImport XOCL;
parserImport XML::PrintXML;

import Components;
import IO;

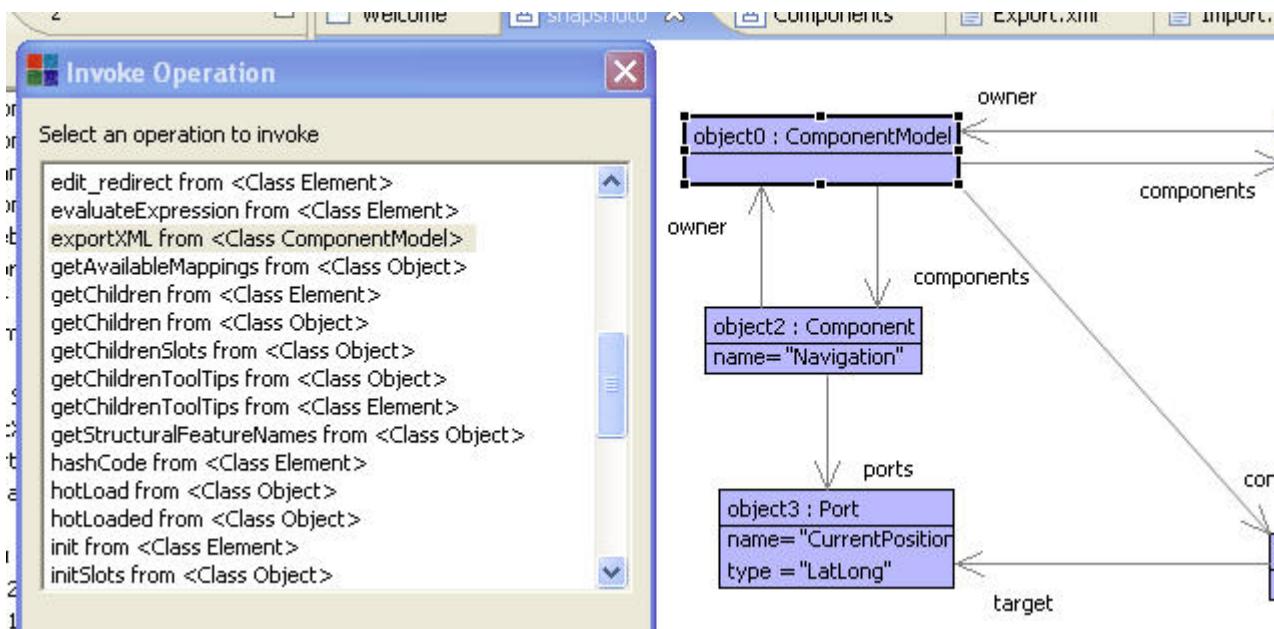
context ComponentModel

@Operation exportXML()
let file = xmfc.writeFile(xmfc.projDir(),"*.xml")
in if file <> ""
then
    if not file.fileExists() orelse xmfc.question("Overwrite " + file)
    then
        @WithOpenFile(fout -> file)
            self.exportXML(fout);
            xmfc.message("written " + self.toString() + " to " + file)
        end
    else xmfc.message("Deployment Cancelled")
    end
    else xmfc.message("Deployment Cancelled")
    end
end
end

//Rest of rules

```

This can be run on any ComponentModel instance, for example:



The following XML is produced:

```

<ComponentModel>
  <Component name="Displays">
    <Port id="Displays:CurrentPosition" type="Long" name="CurrentPosition" />
  </Component>
  <Component name="Navigation">
    <Port id="Navigation:CurrentPosition" type="LatLong" name="CurrentPosition" />
  </Component>
  <Connector target="Navigation:CurrentPosition" source="Displays:CurrentPosition" />
</ComponentModel>
  
```

This can now be parsed back into the tool if required using the XML parser!

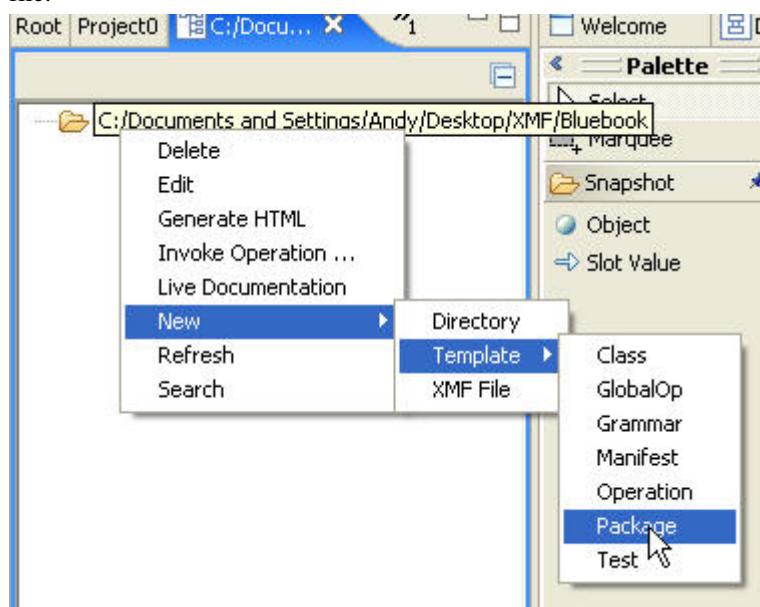
Chapter 8. Using the Programming Interface

This section shows how to use the programming interface to construct some of the models described above. We will add use the programming interface (see Part 1 of this guide) to construct the code.

Getting Started

Select File Browser .. from the File menu. Choose a directory (or create one), where you want the code to reside.

Right click on the directory and choose New XMF File From Template > Package to create an XMF file.



The file will contain the default declarations needed to construct an XMF package definition. Change the name of the file to Components.xmf.

We will be creating a package in the context of the Root package, so make sure that the context is set to Root. There are no imports.

```
parserImport XOCL;

context Root
@Package Components

@Doc The Components model end

end
```

The screenshot shows the code editor with the file 'Components.xmf' open. The content of the file is as follows:

```
parserImport XOCL;

context Root
@Package Components

@Doc The Components model end

end
```

The '@Doc' line is highlighted in red, indicating it is a comment or a placeholder.

Code can now be entered for each of the classes in the model. For a detailed guide to XOCL and XMF classes and package see Part 3 of this guide.

The following code constructs the classes used in the Airports model. Note that constructors have been added for each class, and that attributes have getters (?) and setters (!), and in the case of Set and Sequence types, updators (+) and removers (-).

```
parserImport XOCL;

context Root
  @Package Components

    @Doc The Components model end

    @Class ComponentModel

      @Attribute components : Set(Component) (?,!,+,-) end
      @Attribute connectors : Set(Connector) (?,!,+,-) end

      @Constructor(components,connectors) end

    end

    @Class Component extends NamedElement

      @Attribute owner : ComponentModel (?,!,+,-) end
      @Attribute ports : Set(Port) (?,!,+,-) end

      @Constructor(ports,owner,name) end

    end

    @Class NamedElement isabstract

      @Attribute name : String (?,! ) end

      @Constructor(name) ! end

    end

    @Class Connector

      @Attribute target : Port (?,! ) end
      @Attribute source : Port (?,! ) end

      @Constructor(target,source) ! end

    end

    @Class Port extends NamedElement

      @Attribute type : String (?,! ) end

      @Constructor(type,name) ! end

    end
end
```

Compiling and Loading

At any point, right click on the editor and select Save, Compile and Load to save, compile and load the model.

Checking the Model

The contents of the model can be checked in the property editor by editing the package via the console.

```
[1] XMF> Components;
<Package Components>
[1] XMF> Components.edit();
com.xactium.forms
[1] XMF> Components.browse();
com.xactium.modelBrowser
[1] XMF>
```

To view the package in the browser, enter the package name, Airports, followed by .browse().

Adding Constraints

Constraints can be added by using the @Constraint end declaration.

As an example, let's add a constraint that checks for source and target port types:

```
@Class Connector

    @Attribute target : Port (?,! ) end
    @Attribute source : Port (?,! ) end

    @Constructor(target,source) ! end

    @Constraint ValidPortTypes
        source.type = target.type
    end

end
```

Adding Operations

Operations can be added in exactly the same way: This operation returns all the connectors whose port types don't match

```
@Class ComponentModel

    @Attribute components : Set(Component) (?,! ,+,-) end
    @Attribute connectors : Set(Connector) (?,! ,+,-) end

    @Constructor(components,connectors) end

    @Operation dontMatch()
        connectors->select(c |
            c.source.type <> c.target.type)
    end
end
```

Context

A context is a powerful device for adding new elements to a class or package. It enables different aspects of an element to be separated out and declared elsewhere.

To use a context, first declare the context, and then the element that is to be added to it. For example, we can add the ValidPortTypes constraint using a context.

```
context Connector
    @Constraint ValidPortTypes
        source.type = target.type
    end
```

Note, this must be declared outside of the Airports package declaration to be syntactically valid.

In general, any element that has an owner can make use of context, including attributes, operations, classes and packages.

Importing Packages

Import declarations enable the contents of other packages to be imported so that they can be referenced by other elements. For instance, let's say we wanted to separate out the constraints of the Components model from the class. We can create a new file called ComponentsCons.xmf that contains the constraints:

Provided that we have loaded the Components package, we can compile and run the new file and the constraints will be added to the appropriate classes.

```
parserImport Xocl;

import Components;

context Connector

    @Constraint ValidPortTypes
        source.type = target.type
    end
```

Imports are transitive, and can be used to any depth.

Chapter 9. Toolbar Menus and Initialisation Files

This section describes how XMF-Mosaic can support user customised menus and initialisation files.

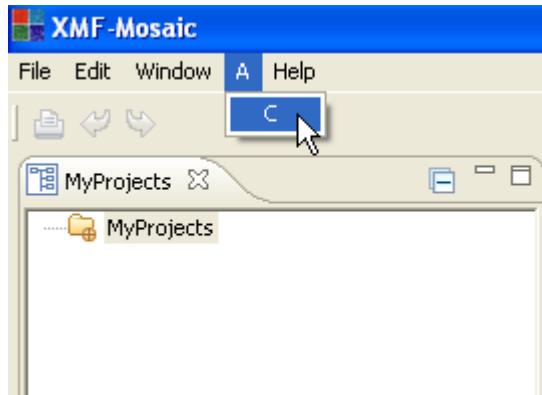
Toolbar menus

XMF-Mosaic provides facilities for rapidly building customised menus that can be used to perform repetitive tasks such as loading projects, or running operations.

You can add a new menu, or a new menu item to an existing menu, on the toolbar by sending a message to the xmf object: xmf.addDropDownMenuItem(menuPath,handler) where menu path is a sequence of strings that identifies the menu item, and handler is a 0-arity operation that is called when the new menu item is selected. For example, the following can be run in the console:

```
xmf.addDropDownMenuItem(Seq{ "&A", "B", "&C" }, @Operation() "Selected".println() end)
```

This will produce the following:



Selecting C will run the operation and generate the text "Selected" in the console.

Note, the menu item "B" denotes a menu category. If other menus are created using "A" and "B" they will be shown as belonging to the same category of sub-menus.

Also note, if the menu item is preceded by an "&" it represents a key-binding. In this case, "Alt-A" followed by "Alt-C" will run the operation.

Commands to add menus can be loaded on startup of the tool using the Init file (see below).

Some Useful Operations

The following are examples of useful operations that are typically used in menus.

This is an example of a menu that is used to load one or more .xar files in your projects directory, MyProjects, via a Demo menu:

```
xmf.addDropDownMenuItem(Seq{ "&Demo", "Main", "Load Model" },
@Operation()
@Busy("Loading Model")
    xmf.projectManager().getElement("MyProjects").loadMosaicProject(xmf.projD
    end
end);
end);
```

This is an example of a menu that is used to load one or more compiled .o files:

```
xmf.addDropDownMenuItem(Seq{ "&Demo", "Main", "Load Model" },  
    @Operation()  
    @Busy("Loading Model")  
    xmf.projectManager().getElement("MyProjects").loadMosaicProject(xmf.projD  
    end  
end);
```

This example enables a specific file to be edited after selecting it using a file chooser. Note, the following two examples need to import the XmfFileTree (as shown in the next section).

```
xmf.addDropDownMenuItem(Seq{ "&Demo", "Main", "Edit File" },  
    @Operation()  
    let file = xmf.openFile(xmf.projDir(), "*.xmf") in  
        if file.fileExists() then  
            XmfFile(null, file).editText()  
        end  
    end  
end);
```

This example enables a specific file to be selected and then compiles and loads it: Note the use of the string operation splitBy. This splits a file name around the "." into a sequence, the head of which is the preceding part of the file name.

```
xmf.addDropDownMenuItem(Seq{ "&Demo", "Main", "Compile and Load File" },  
    @Operation()  
    let file = xmf.openFile(xmf.projDir(), "*.xmf") in  
        if file.fileExists() then  
            @Busy("Compiling and loading "+file)  
            let filename = file->splitBy(".", 0, 0)->head in  
                Compiler::compileFile(file, true, true);  
                (filename + ".o").loadBin()  
            end  
        end  
    end  
end);
```

Initialisation Files

To make the task of setting up menus and other general-purpose features easier, XMF-Mosaic provides an init file facility. This file, init.o, is automatically loaded on start-up of the tool provided that it is placed in the directory referenced by the MOSAICINIT environment variable.

The following init file (init.xmf) is an example of a simple init file. To use it, compile the file in the MOSAICINIT directory. It contains a menu item which enables the init.xmf file to be conveniently edited via a File > Edit > Init menu.

```
// The init file is loaded on startup. It is used to automatically add user cus  
// menus and operation to XMF-Mosaic. These can be used for all sorts of  
// purposes, including setting up menus for specific projects, or running  
// any code you wish. To run the init file, it must be saved in the XMFINIT  
// directory (this must be available under your windows system environment  
// variables). Compile, save and load the init file before launching XMF-Mosaic  
// to run it.  
  
parserImport XOCL;
```

```
import Clients;
import XmfFileTree;
import IO;

// Add your customised menus and operations here.

// Provides a drop down menu for editing the Init file. Note the sequence
// elements denotes the menu hierarchy starting with the root File menu.
// Use & to add shortcut key binding, e.g. Alt-F, Alt-E in this case to
// get to the Edit menu.

xmf.addDropDownMenuItem(Seq{ "&File", "Extras", "&Edit", "Main", "Init" },
@Operation()
    XmfFile(null,xmf.initFile()).editText()
end);

// Provides a drop down menu for browsing the files in your project directory
xmf.addDropDownMenuItem(Seq{ "&File", "Extras", "&Browse", "Main", "Projects" },
@Operation()
    Directory(xmf.projDir(),Seq{ ".*.xmf" }).browse()
end);

// An example of a menu that is used to load one or more .xar files in your pro
// MyProjects

xmf.addDropDownMenuItem(Seq{ "&Demo", "Main", "Load Model" },
@Operation()
    @Busy("Loading Model")
        xmf.projectManager().getElement("MyProjects").loadMosaicProject(xmf.projD
            end
        end);

// An example of a menu that is used to load one or more compiled .o files

xmf.addDropDownMenuItem(Seq{ "&Demo", "Main", "Load File" },
@Operation()
    @Busy("Loading File")
        (xmf.projDir() + "/Dir/Code.o").loadBin()
    end
end);

// An example of defining an operation that will be added automatically to Root
context Root
@Operation hello()
    format(stdout,"HelloWorld")
end
```

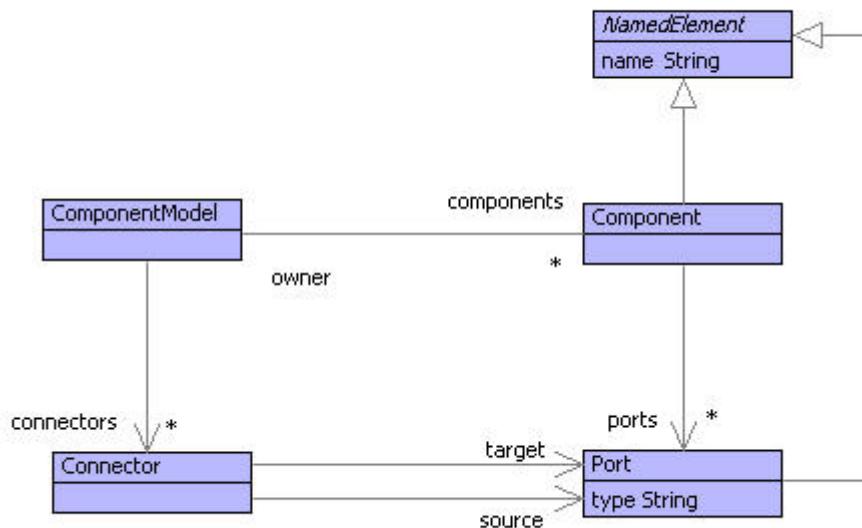
Chapter 10. Constructing a Diagram Tool for a Model in XTools

XTools provides a suite of languages for modelling and deploying user-interfaces. This chapter describes how diagram editors can be modelled and deployed in XTools.

Domain Model

We want to create a diagram tool which manipulates an underlying domain model.

As an example, let's construct a diagram editor for the component modelling language. Here is it's domain model:



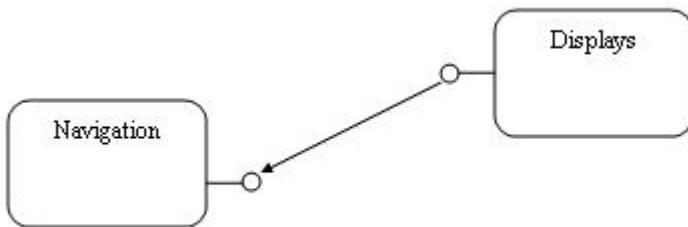
When we construct a diagram in the tool, we want it to manipulate instances of this model.

A Candidate Diagram Syntax

We want to construct an editor that:

- represents Components as rounded boxes with names
- represents Ports as circles.
- represents Connectors as arrows between Ports.
- allows Ports to be connected to Components

In other words, something like the follows:



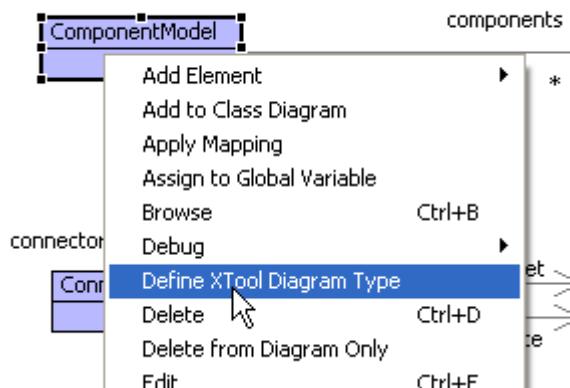
Constructing an XTool Definition

Let's go through the process of constructing an XTool definition.

Creating the Tool Definition

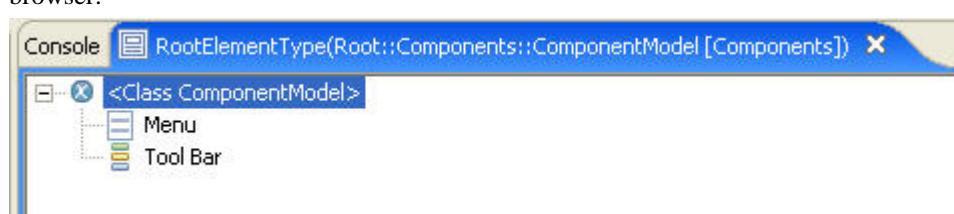
First, we need to create an XTool definition. This will contain all the information needed to support the XTool..

To do this, simply right click on the class that represents the root type of the model to be XTooled: In this case, it is the class ComponentModel:



Enter a name for the XTool, e.g. Components.

A new XTool definition will be created. This will be displayed in the form editor and in the XTool browser.

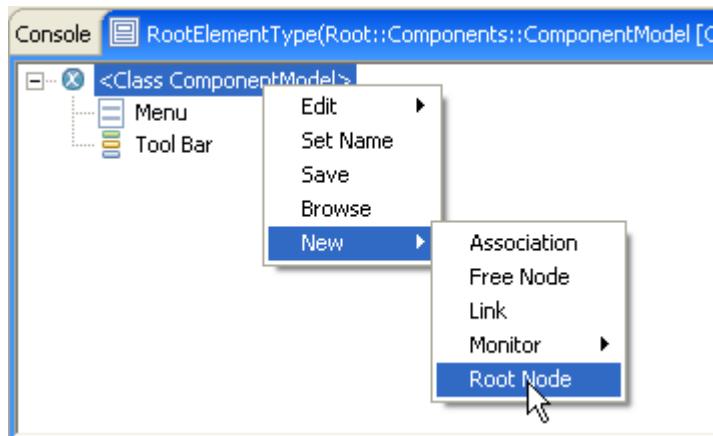


Adding a Node

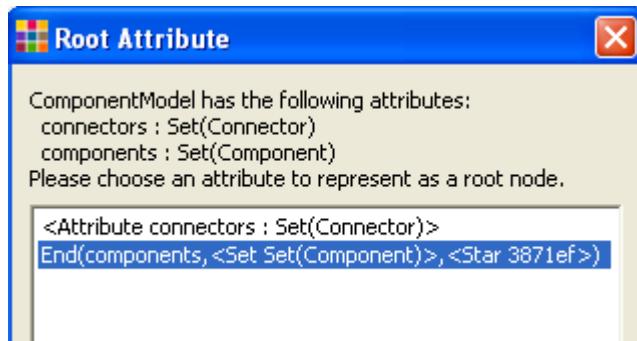
Two main types of diagram element can be added to the tool: nodes and edges.

Let's create a node definition that represents a Component.

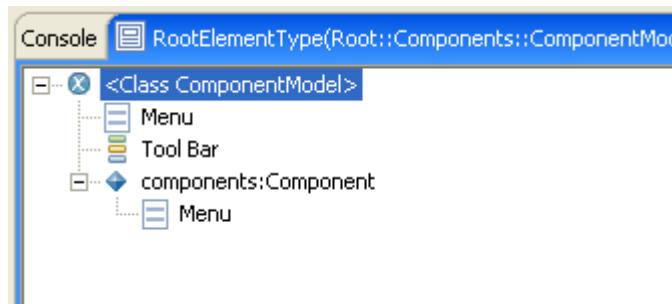
To do this, right click on the root type and select New > Root Node.



A choice of elements to be displayed as nodes will be shown. In this case, we want to select the Component element.



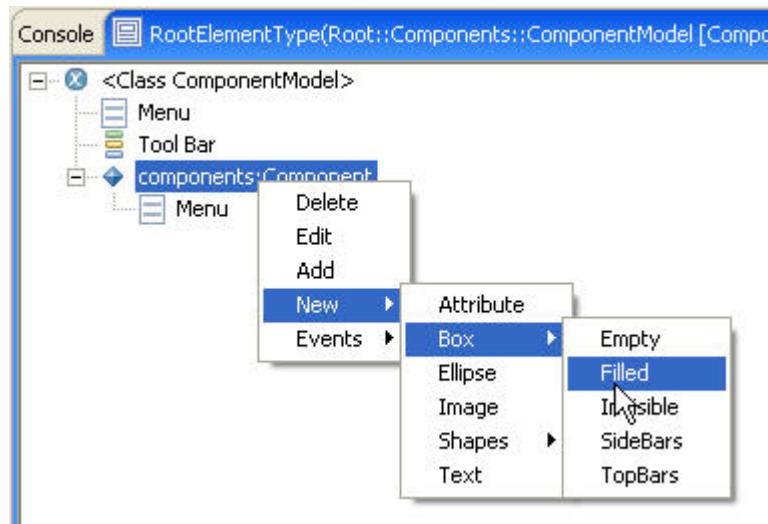
The Component node is then added to the XTool:



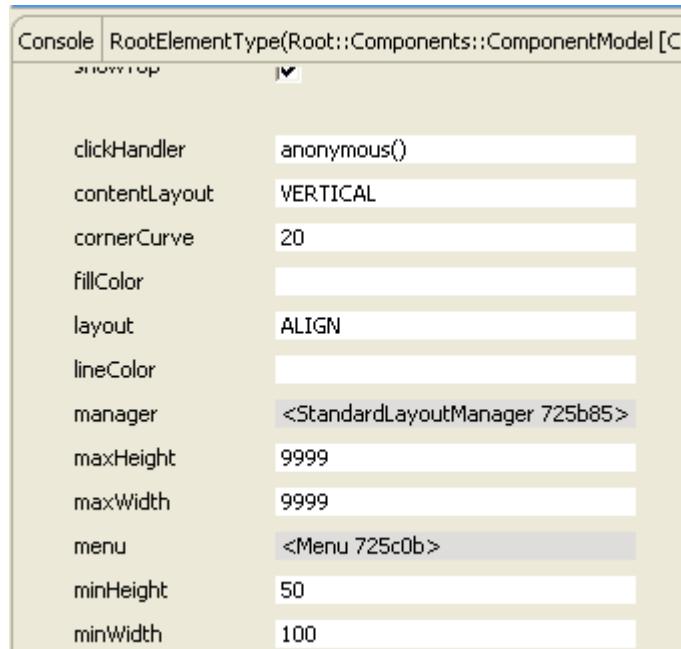
Adding a Box to the Node

A Node on its own will not have a diagrammatical representation. To enable this, we need to add a shape to it.

An Entity will be represented as a Box, so we'll add one to the Component node. To do this right click on the Component node and create a new box:



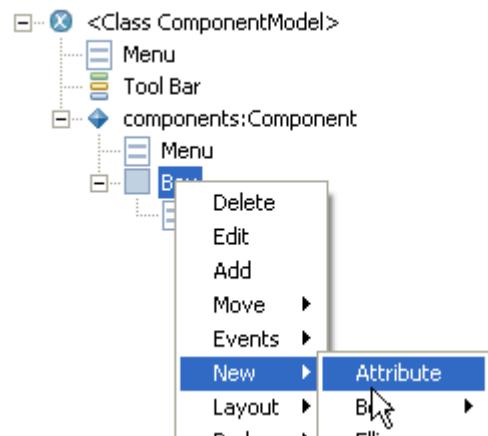
Boxes can be parameterised in a wide variety of ways. In this case, we want the box to have a minimum width of 100, a minimum heights of 50 and a corner curve of 30 degrees. To change this, right click on the create box and select Edit. Change the properties as required:



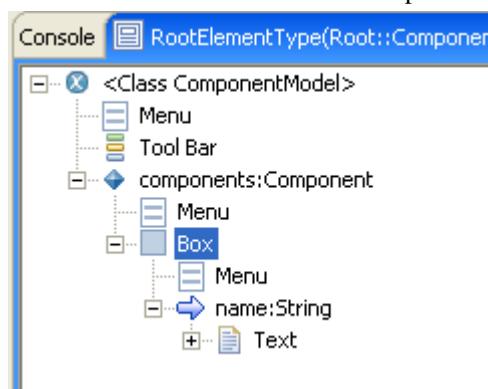
Adding an Attribute to the Box

In order to be able to edit the name of the Component, we need an Attribute text field. This can be added to the Box as follows:

Right click on the box and select New > Attribute. Sele

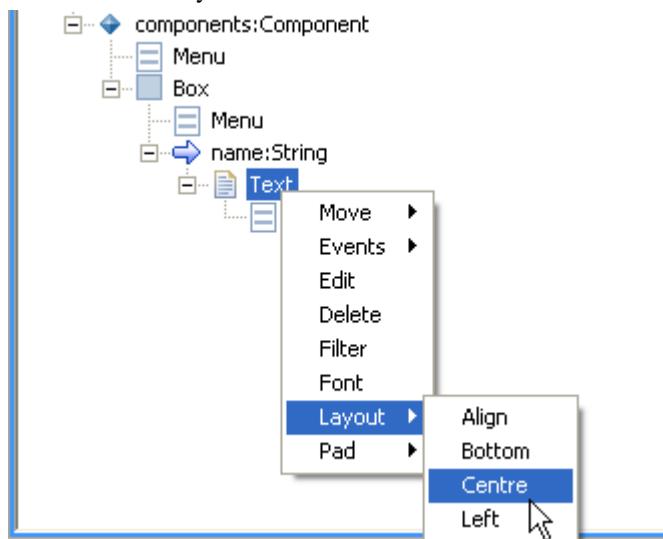


Select the name attribute from the drop down list. The new attribute has been added to the box:



A Text box is automatically added to the Attribute to display its value. It can be edited to view its properties.

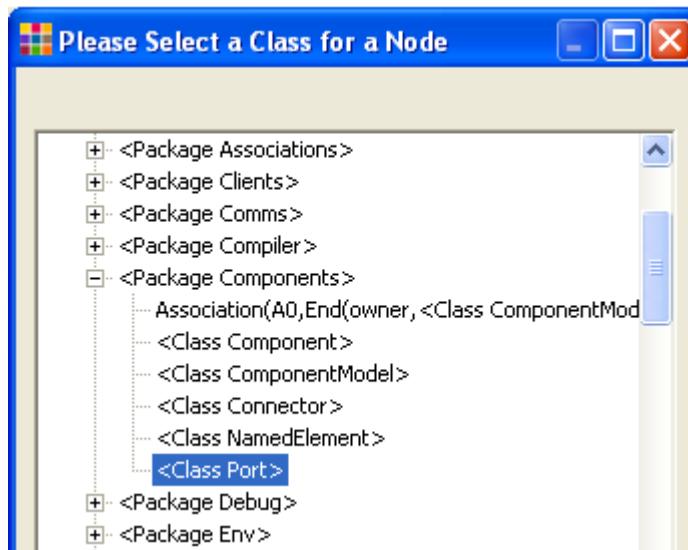
We want to ensure that the text for the Attribute is centred in the box. To do this, right click on the box and set its Layout to Centre:



Adding a Free Node

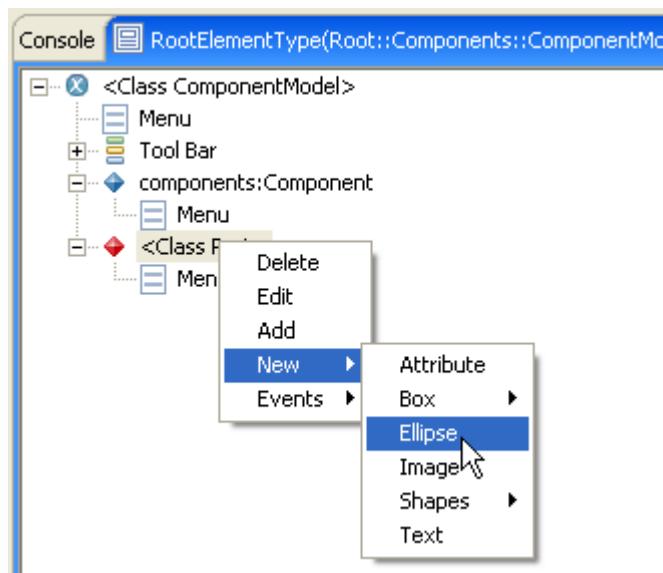
A Free Node is a node whose class is not directly accessible from the root class. In this example, Port is a free node.

To add a Free node, right click on the root element type and select New > Free Node. A list of all the packages available in the system will be displayed. Select the Port class from the package Components.

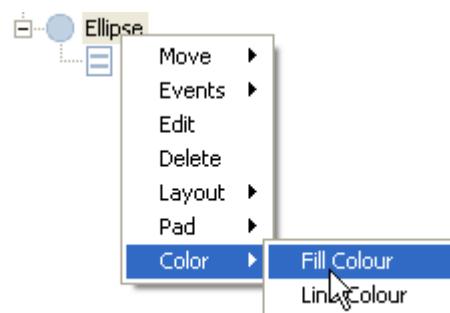


The node will be added to the browser.

We want to represent a Port as an Ellipse. An ellipse can be added in the same way as a box:



We will need to adjust the size and colour of the ellipse. To change its fill colour, right click and select Colour > Fill Colour.

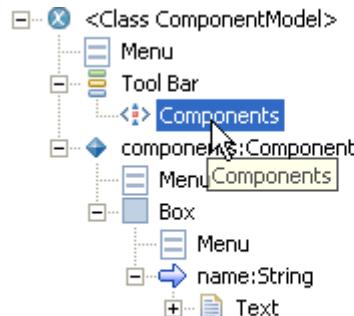


The size of the ellipse can be changed by editing the properties of the Ellipse. In this case, we want its height and width properties to be 10 points, however, this is the default size, and we therefore don't need to make any changes.

Adding Tool Bars

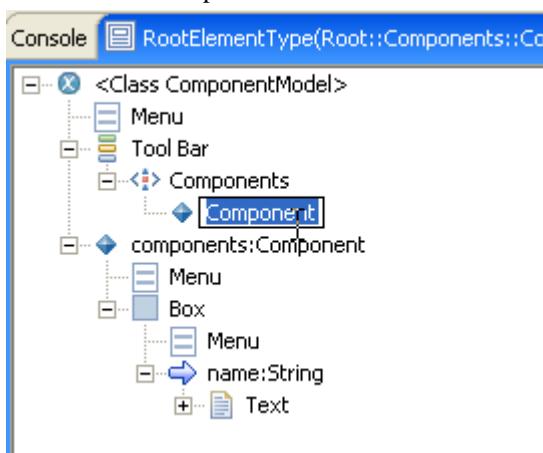
Toolbar groups and buttons need to be created to enable the selection of a creation tool for the element.

To do this, right click on the Tool Bar node and select New Group. Give the Group a name, e.g. Components:



This provides a tool bar group, under which a number of tool bar buttons can be added.

Let's create a button for adding Component nodes. Right click on the Component group and select New > Node Button. Choose Root(component:Component) from the list. Change the name of the button to be "Component".



Follow exactly the same steps to add a button for Port nodes.

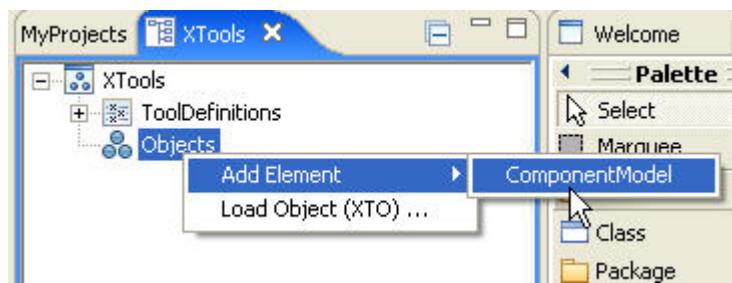
Running the Tool

One of the powerful features of XTools is that there is no compile cycle involved in running a tool. This makes testing a very dynamic and iterative process.

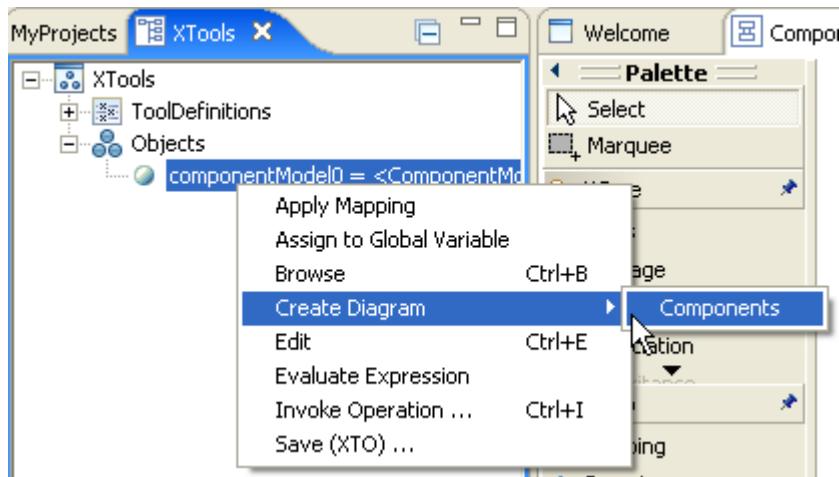
Let's run the tool we have built so far.

First, Save the XTool (right click on the Root Node and select Save) so that it is backed up.

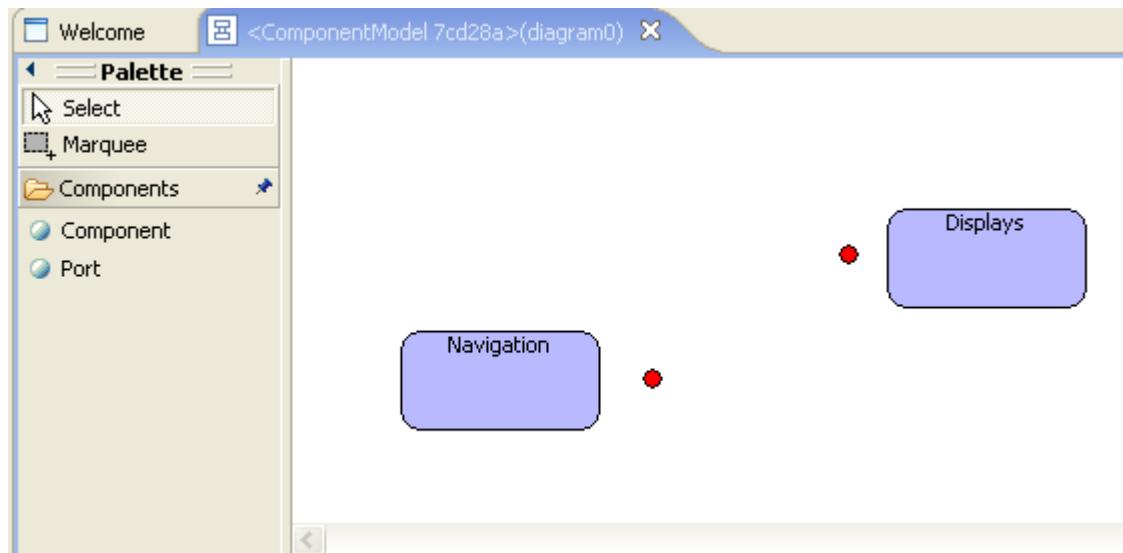
Next right click on the Object node in the XTool browser (select Browse > XTools Browser if you cannot see it in the browser) and select Add Element > ComponentModel to create an instance of the ComponentModel class.



A ComponentModel object will be created. To show its diagram select it and right click Create Diagram > Component.



A new diagram editor for the tool will be displayed. Note that the relevant tool bar and buttons are now available for adding Component and Port nodes to the tool.



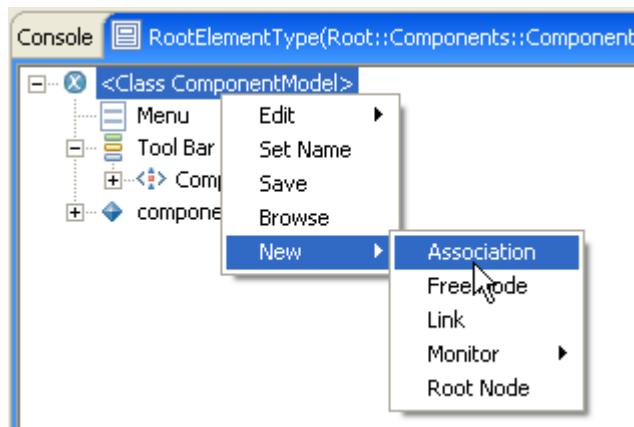
Adding Edges

Another type of diagram element is an Edge (a connection between Nodes). Edge definitions can be added to the tool in a similar way to Node definitions.

Edges come in two types:

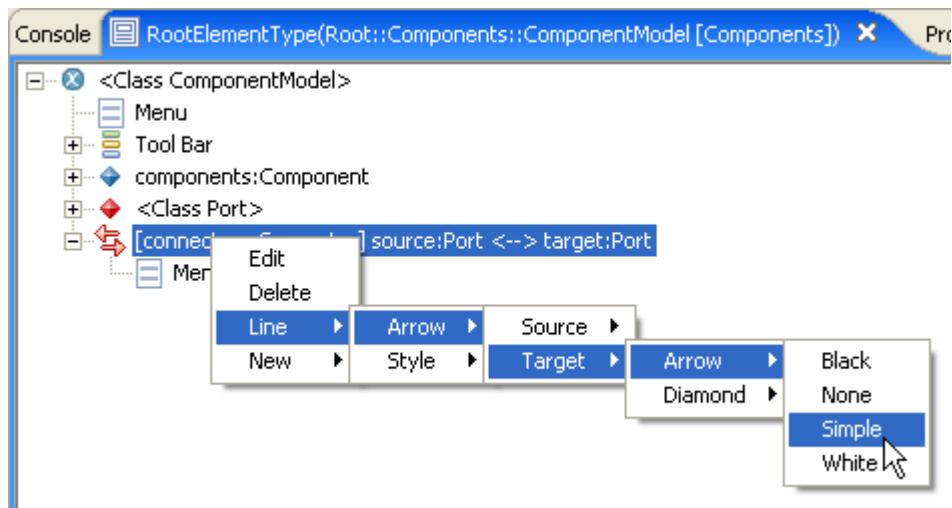
- Associations, which map to a class that acts as a relationship between elements.
- Links, which map to an attribute of a class.

We want to represent a Connector as an Association between Ports. To do this, we add an Association to the tool definition as follows:



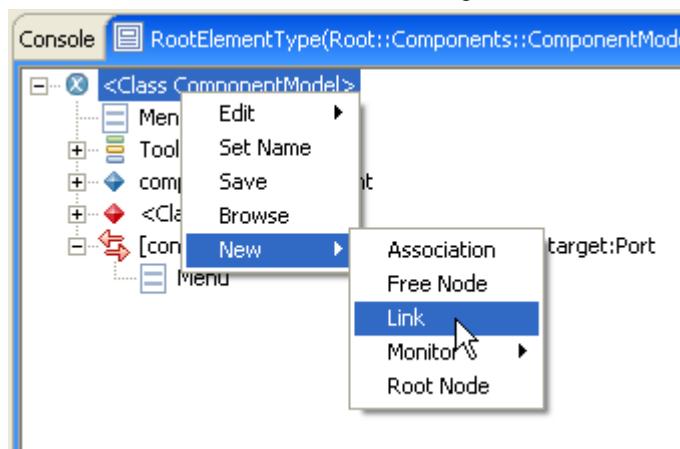
A choice of collections that the Association will map to will be displayed. Select the connectors collection.

An Association may define the shape of its ends, and in this case we can chose to make its target end an arrowhead by setting the target arrow of the edge to be a simple arrow.

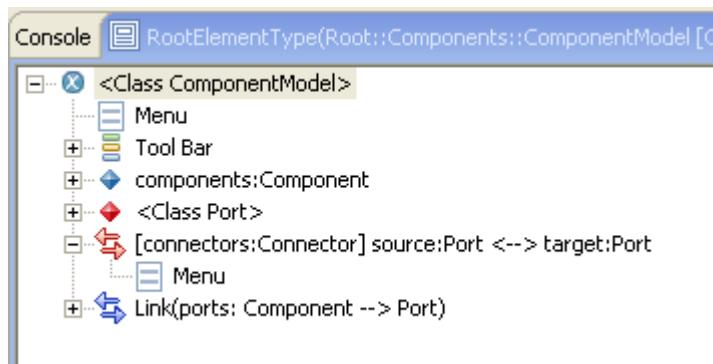


The second edge to add to the tool is the link between a Component and a Port which corresponds to the ports attribute of the Component class.

To do this, add a new Link to the tool using the New menu.

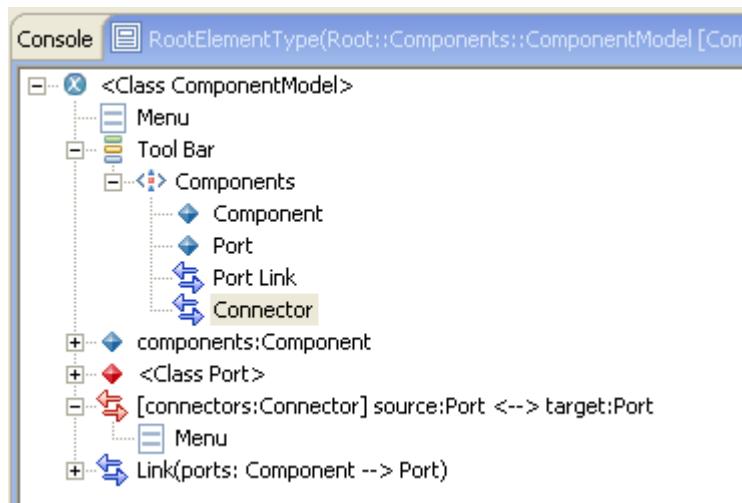


Select the source of the Link (in this case the class Component), then select the attribute that will map to the link (in this case the attribute ports). The resulting tool is shown below:



Adding Edge Toolbar Buttons

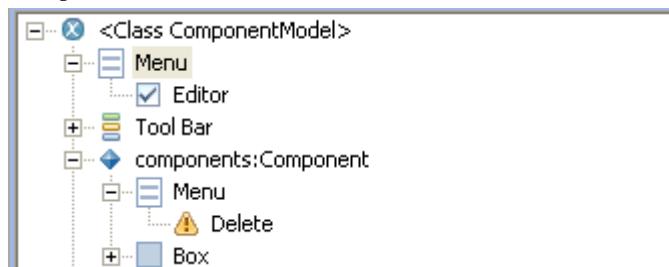
Edge tools bar buttons can be added in exactly the same as node buttons. Here the two edge buttons corresponding to the Port Link and Connector Association have been added.



Adding Menus

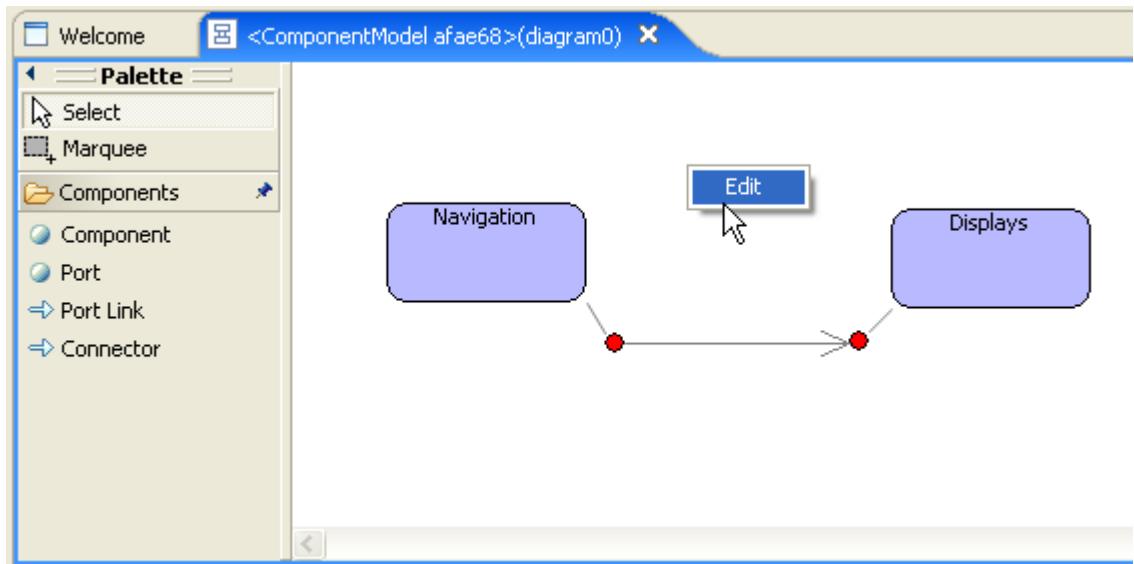
It is possible to add menus to all diagram elements for performing a wide variety tasks. For example, Edit and Delete menu items, or menu items that perform specific XOCL commands.

To add a menu item, select an element and right click on its Menu to add a New menu item. Here two menus have been added. An Edit menu item to the tool diagram and a Delete menu item to the Component node.



Re-running the tool

We can re-run the tool to test out its functionality.



Running Edit, will show the property editor for the ComponentModel instance being managed by the tool.

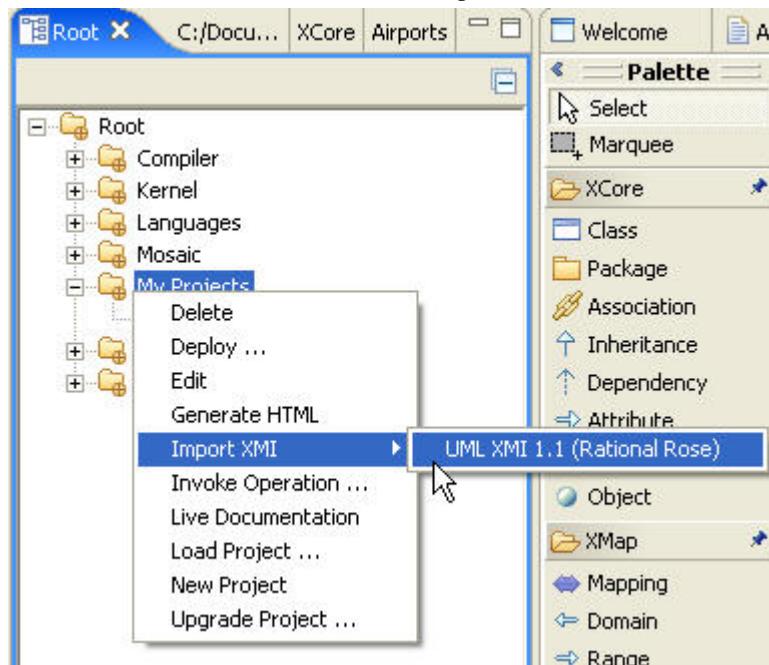
This is just the start of developing a fully blown tool. Additional menus and diagram elements can be added very flexibly and tested interactively.

Other XTool Capabilities

Many other XTool capabilities are supported by XMF-Mosaic, including support for modelling browsers and property editors. Walkthroughs for these capabilities will be available in the next release of the tool.

Chapter 11. Importing XMI

XMI can be imported into XMFMosaic by selecting the import XMI option off a project. The list of currently supported XMI versions will be shown in the menu. Select the required version and use the file chooser to select the .xml file to be imported.



Chapter 12. Constructing a Textual Syntax and Parser

This chapter demonstrates how a textual syntax can be constructed in XBNF: a language for defining grammars and mappings of grammars to models.

Here we show how this can enable us to parse code and synthesise instances of models. However, XBNF supports many other powerful capabilities, including capabilities for extending the grammars of existing languages and for conveniently synthesising code.

Parsing and Synthesising Instances of Models

We want to synthesise instances of the following model by parsing in a textual representation:

```
parserImport XOCL;

context Root

@Package EntityModels

@Doc
    A Simple Entity Model
end

@class Named extends XOCL::Syntax
    @Attribute name : String end
end

@class EntityModel extends Named
    @Attribute entities : Set(Entity) end
end

@class Entity extends Named
    @Attribute relationships : Set(Relationship) end
end

@class Relationship extends Named
    @Attribute type : String end
end

end
```

Note that the class `Named` extends the class `XOCL::Syntax`. It is necessary for all classes we want to synthesise to inherit from this class.

The code for a grammar for this language is shown below:

```
parserImport XOCL;
parserImport Parser::BNF;

import EntityModels;

context EntityModel
```

```

@Grammar extends OCL::OCL.grammar

EntityModel ::= 
    name = Name
    entities = Entity*
    {EntityModel[name = name, entities = entities]}.

Entity ::= 'entity' name = Name rels = Relationship*
    {Entity[name = name, relationships = rels]}.

Relationship ::= 'rel' name = Name '->' type = Name
    {Relationship[name = name, type = type]}.

end

```

Note, just as with the XML grammar in the previous chapter, we define a series of grammar rules that synthesise instances of the model.

Here are some points to note:

- The grammar first imports the parsers for XOML and BNF. It also imports the EntityModels package (which must be loaded). The grammar of the EntityModel class is defined as follows:
 - The root of the grammar is the rule EntityModel.
 - A rule is sequences of pattern declarations followed by an action (in curly brackets) which is called after the parsing has occurred.
 - In the case of the EntityModel rule, it is defined to be a name, followed by a sequence of Entity (which is bound to the variable entities).
 - The result of parsing an EntityModel is to create an instance of a EntityModel class with the variables name and entities populated with the above values.
 - An Entity is the string ‘name’ and a sequence of Relationships (bound to the variable rels). The result of parsing it is to create an instance of a Event class with the variable name passed as a parameter.
 - A similar rule is used for Relationships.

The following is a small example of a model written in the syntax:

```

parserImport EntityModels;

import EntityModels;

Root::p :=
    @EntityModel Customers
        entity Customer
            rel Owns -> Account
        entity Account
    end;

```

Compiling and loading the above three files in order will load the model, the parser definition and the example. The result will be stored in the Root variable p, which can be edited in the console using p.edit().

Chapter 13. Creating a Meta-Profile

One common use of modelling tools is to tailor them to support a specific modelling domain. This typically involves specialising modelling concepts already provided by the tool using stereotypes and tagged values. The advantage of doing this is that the tool's existing editing and drawing capabilities, etc, do not need to be physically changed. By combining a number of different stereotypes together, a "profile" can be constructed for a specific modelling requirement.

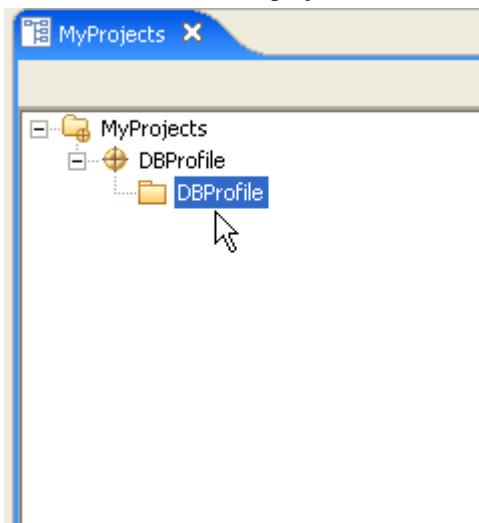
The disadvantage of traditional stereotype and tagged value mechanisms is that they are essentially just a way of annotating existing model elements – little in the way of semantic or well-formedness information can be added to them.

XMF supports stereotypes and tagged values, but in a way that is significantly more controllable and powerful than traditional tools. XMF enables "meta profiles" to be constructed, in which stereotyped elements are true instances of specialised language concepts.

An Example Profile

Let's design a data modelling profile, which allows us to construct a data model as an instance of a datamodel metamodel. The data modelling profile provides three key modelling concepts: a data model, a data entity and a key attribute.

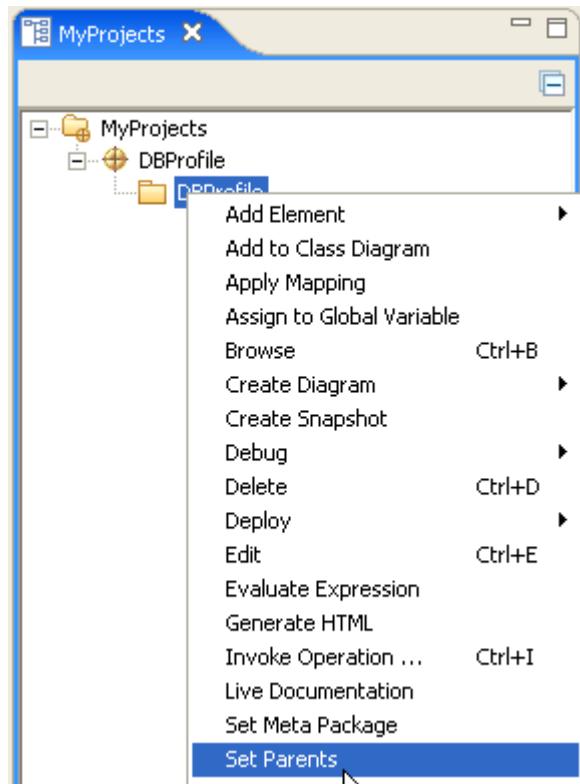
First, create a DBProfile project.



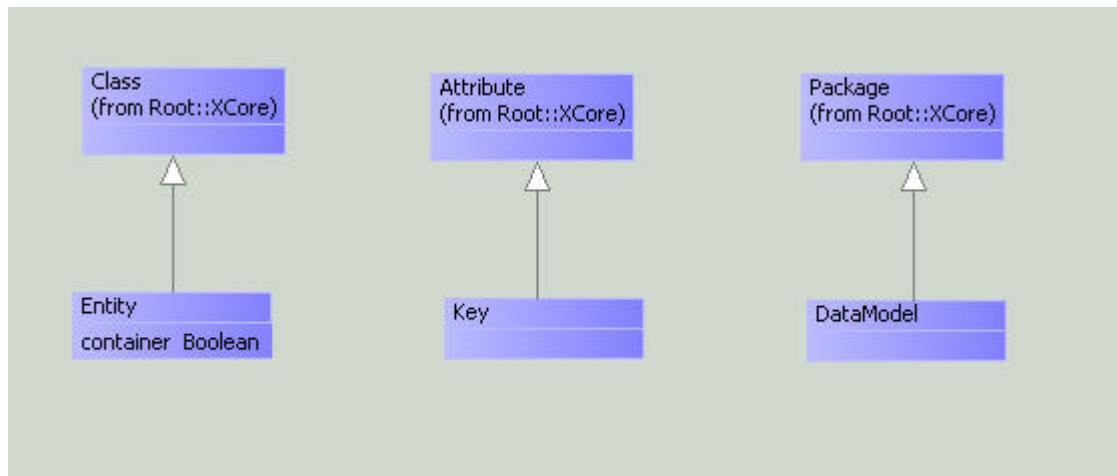
Next, browse and show the diagram for the project.

A key part of the profile is that it extends the XCore metamodel. This enables it to inherit the modelling capabilities of XCore - the language used to describe class models.

Right click on the DBProfile package and select Set Parents, and tick the package XCore from the list.



At this point we can open the DBProfile package and begin constructing the profile. We want to be able to extend existing modelling elements so that we can reuse their editing capabilities. Here is the model for the DBProfile:



Here a DataModel specialises the class Package, so we can now model with DataModels rather than Packages. A Key is a specialisation of an Attribute, and an Entity specialises a Class. Finally, an additional attribute is added to Entity to denote that some Entities are also containers.

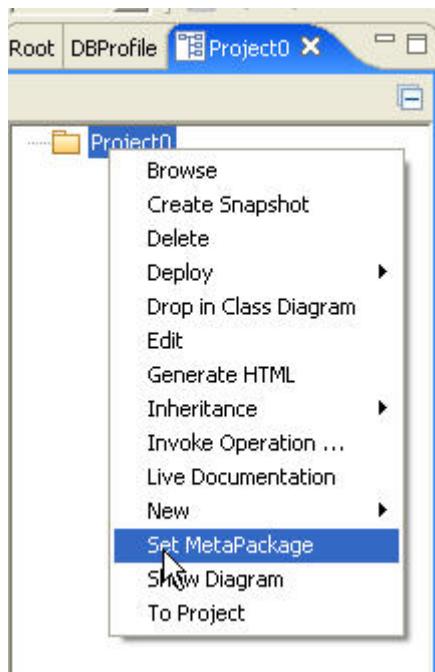
Note that in order to specialise the XCore classes, we simply right clicked on each class and selected Inheritance > Add Parent.

We can now create a model that makes use of this profile.

First create a new project and browse the package.

Rather than editing this package as an XCore package, we want to edit it as an instance of the profile.

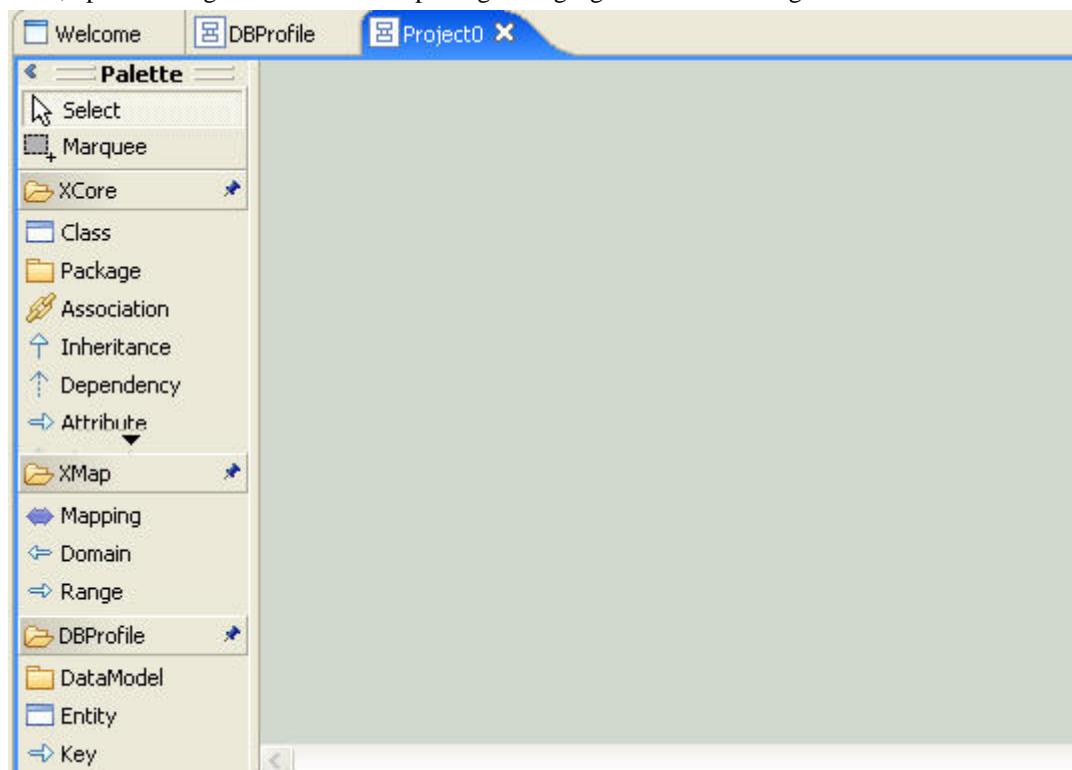
To do this, right click on the model package and select Set MetaPackage.



A list of meta packages will be displayed.

Choose DBProfile.

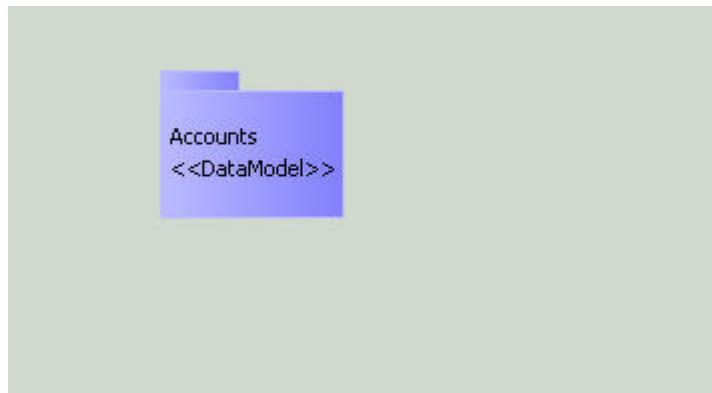
Now, open the diagram for the model package using right click ShowDiagram.



Notice that in addition to the usual tool buttons, a new collection of buttons is displayed at the bottom.

We can select these elements and start creating models using the new model elements that are provided by the profile.

For instance, a new DataModel can be created, called Accounts.



The contents of the Accounts can be viewed by showing its diagram (right click Show Diagram). However, when this is done, the profile buttons are no longer there. This is because we need to set the metaPackage of the model to be the profile package.

We could do this by hand, but manually setting its metaPackage is time consuming. A more general solution is to extend the initialisation operation of the DataModel class so that this is done automatically. This is done as follows:

Right click on the class and add an init() operation.

```

Console Properties - pprint() Properties - init()
self           init()
of             CompiledOperation
owner          DataModel

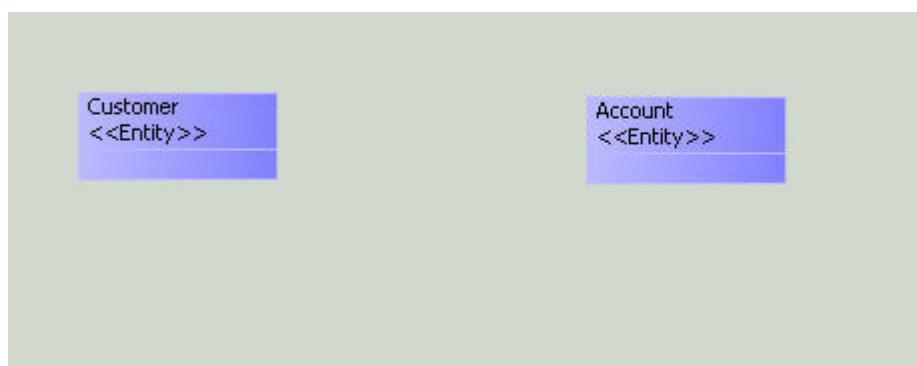
source
@Operation init(args : XCore::Element):XCore::Element
    super();
    self.metaPackage := DBProfile
end

```

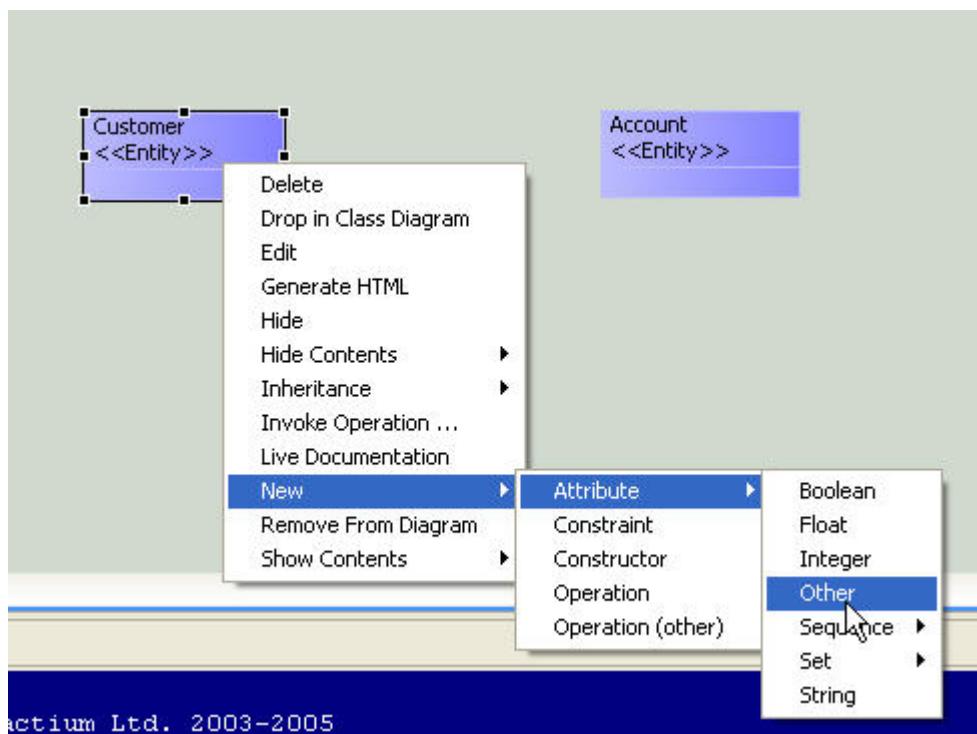
The init() operation requires a single parameter args. Because it extends the init() operation on the class Package, the operation calls its super class's operation's body and then sets the metaPackage.

Now when we view the diagram for a DataModel, the appropriate buttons are available, and a model can be created.

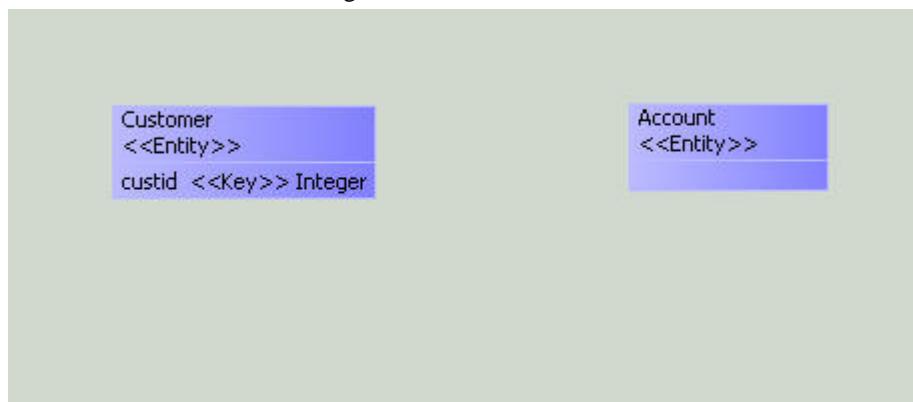
Let's create some entities:



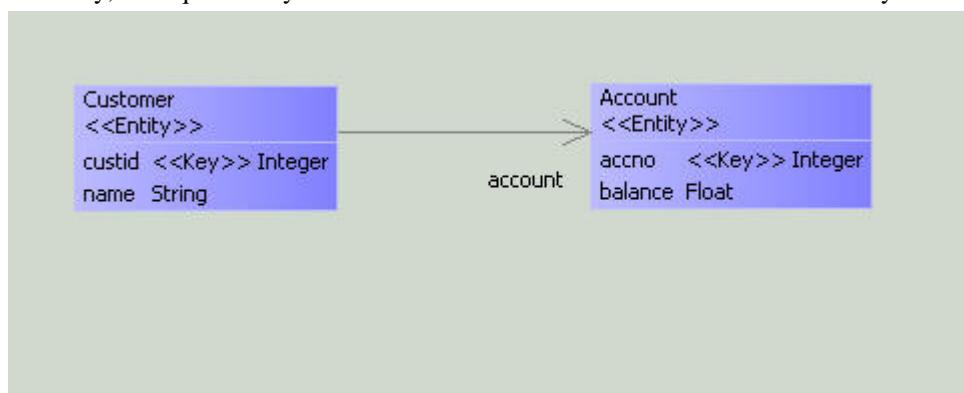
We now want to add an attribute to the customer class. However, in this case we want the attribute to be a Key. To select this, Other is chosen from the New > Attribute menu.



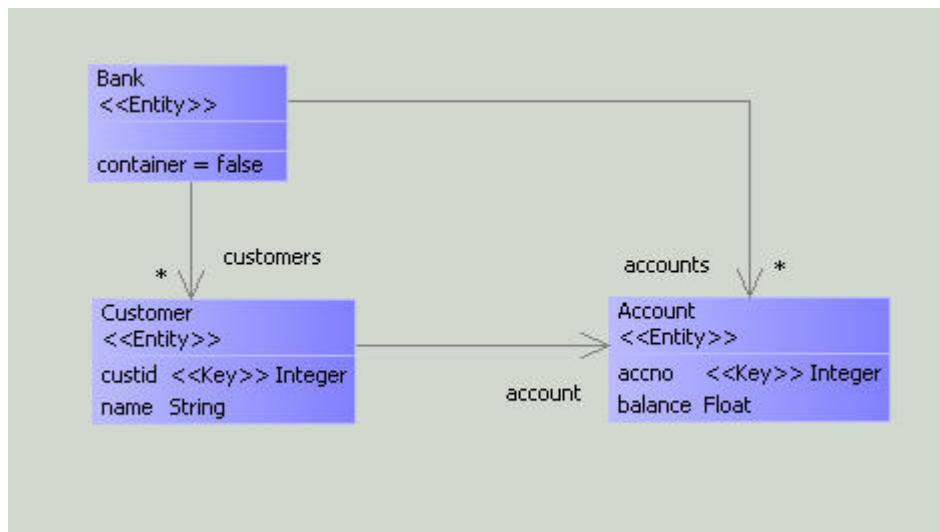
The attribute name is then changed to custid.



Similarly, we require a Key for the Account class. We'll also add some other non-key attributes as well:



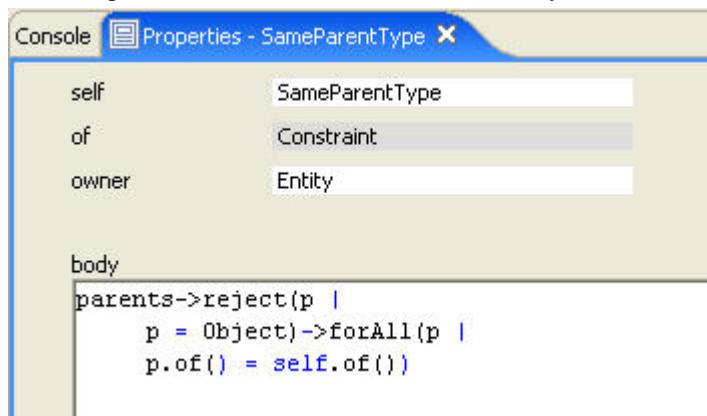
Finally, let's add a Bank entity. In this case, we want to capture the fact that the Bank entity is a container. If you remember, the class Entity extended the attribute of the class Class, with an attribute called 'contents'. Because the Bank is an instance of the class Entity, we can now set this value to be true. To do this, right click on the Bank and choose Show Contents > Slot Values. The result is to add a new compartment to the class in which the Slot Value is displayed.



The Slot Value can be set to true by clicking on the value and editing it.

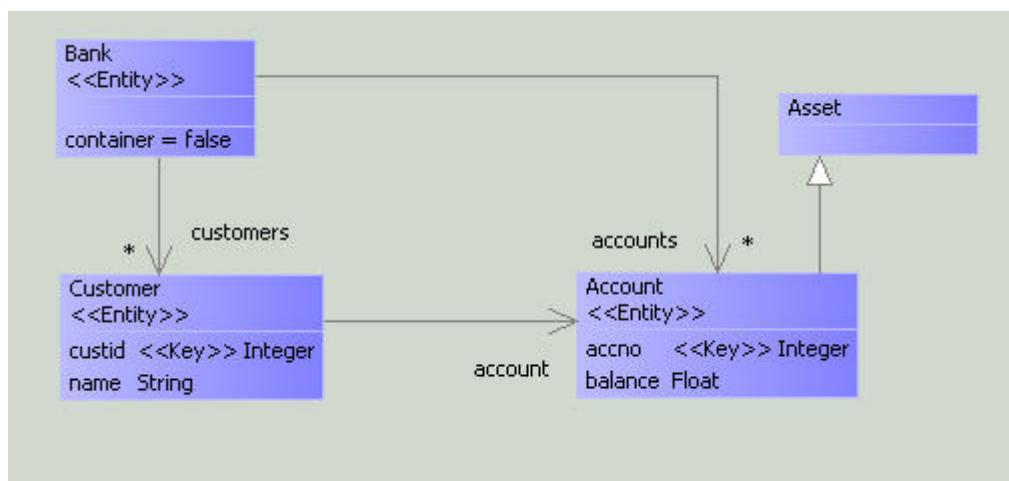
Adding Constraints

Constraints can also be added to the profile model to rule out specific relationships between elements. For example, let's add a constraint to the class Entity that ensures it can only specialise another Entity.

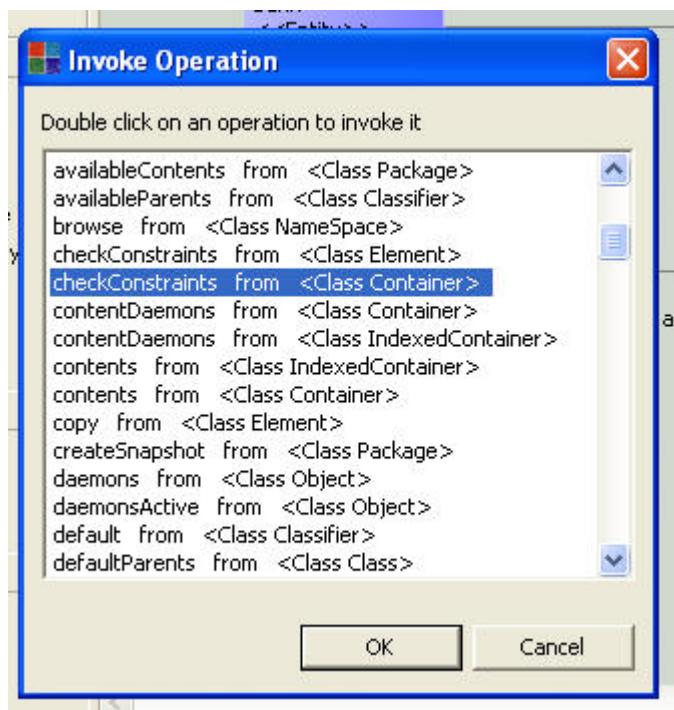


The constraint rejects parents of type Object, as all elements will inherit from this class. It ensures that the type of the parent must be the same as the child.

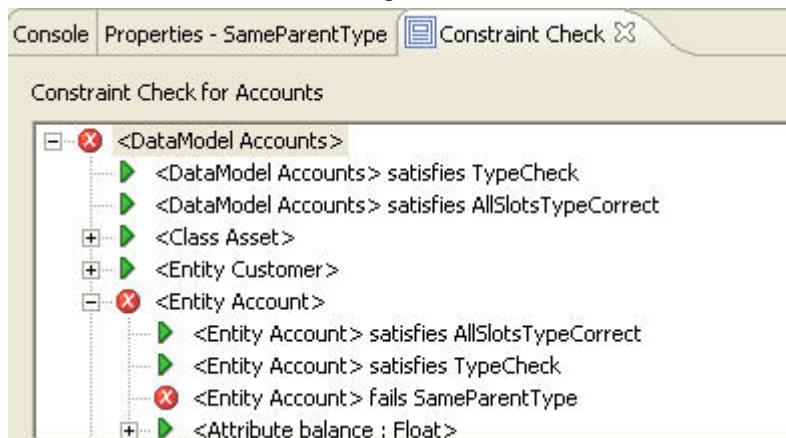
A model can be checked to see whether it conforms to its constraints. Here's a model that should fail:



To check the constraints on the model, we need to run checkConstraints on the contents of the package. To do this, right click on the package in the browser and select Invoke Constraints > checkConstraints() from Container.



The constraint fails as we would expect.



Chapter 14. Generating Code

XMF-Mosaic provides a number of off the shelf routes for generating code from models. These are available off the right click Deploy menu on a package.

Generating Java

Selecting Java from the Deploy menu will generate a new directory containing a Java classes for each of the model classes. The Java classes include the attributes and appropriate accessor and updator operations. In addition, a toXML operation is provided for serialising instances of Java classes as XML.

The following Java code was generated from the Components model.

```
package Components1;

public class ComponentModel {

    // Attributes...

    private java.util.Vector components;
    private java.util.Vector connectors;

    // Accessors...

    public java.util.Vector components() { return components; }
    public java.util.Vector connectors() { return connectors; }

    // Updaters...

    public void setComponents(java.util.Vector value) { this.components = value;
    public void setConnectors(java.util.Vector value) { this.connectors = value;

    // Display...

    public String toString() {
        String s = "ComponentModel[";
        return s + "]";
    }

    // Operation stubs...

    public int deleteFromComponents(int v) {
        return null;
    }

    public int addToComponents(int v) {
        return null;
    }

    public int connectedTo(int p) {
        return null;
    }

    public int dontMatch() {
        return null;
    }
}
```

```
}

public int componentForPort(int p) {
    return null;
}

public int addToConnectors(int v) {
    return null;
}

public int deleteFromConnectors(int v) {
    return null;
}
// XML Serialization...

public void writeXML(java.io.PrintStream out, java.util.Hashtable idTable) {
    if(idTable.containsKey(this))
        out.print("<Ref id=' " + idTable.get(this) + "'/>");
    else {
        String id = Integer.toHexString(this.hashCode()).toUpperCase();
        out.print("<Object id=' " + id + "'>");
        out.print(">");
        out.print("<Slot name='components'>");
        out.print("<Set>");
        for(int i = 0; i < components.size(); i++) {
            Component o = (Component)components.elementAt(i);
            o.writeXML(out,idTable);
        }
        out.print("</Set>");
        out.print("</Slot>");
        out.print("<Slot name='connectors'>");
        out.print("<Set>");
        for(int i = 0; i < connectors.size(); i++) {
            Connector o = (Connector)connectors.elementAt(i);
            o.writeXML(out,idTable);
        }
        out.print("</Set>");
        out.print("</Slot>");
        out.print("</Object>");
    }
}
```

Chapter 15. Using Manifests and Deploying Models

In order to manage the compilation and loading of multiple files, XMF supports a convenient project management abstraction called a manifest. A manifest provides a convenient way of managing collections of files that have dependencies on each other.

In version 1.0 of XMF, manifests must be declared in a file. The following file imports the Manifest facility and implements a manifest that compiles Airports.xmf file followed by AirportsCons.xmf:

```
parserImport Manifests;

@Manifest Airports
  p = @File Airports end
  @File AirportsCons end
end;
```

A manifest is declared using `@Manifest`. A manifest has a name (in this case `Airports`) and contains a list of files.

Save, Compile and Load the Manifest (right click > Save, Compile and Load). A manifest icon will appear in the file browser which contains some additional menu options:

The two additional options are:

Build Manifest: This will compile all the files in the manifest.

Load Manifest: This will load the binary of the files.

Compile, Build and Load: Compiles the Manifest, then Builds and Loads the files.

These can be used to compile and load as many files as required in a specific development project.

Manifest Actions

Actions can be added to a manifest to perform specific tasks once the manifest has compiled and loaded its list of files. For example, we might want to edit the results, or we might want to post process the loaded model in some way.

In the following example, `p` is the package that results after loading two files: Network and NetworkCons. The action that is called after the “do” launches an editor for `p`.

```
parserImport Manifests;

@Manifest Networks
  p = @File Network end
  @File NetworkCons end do
    p.edit()
  end;
```

Building and loading the manifest results in the appropriate property editor being launched.

Deploying Manifests

Often, we don’t want to create manifests for a model by hand. XMF provides a deployment capability that enables manifests (and the files they manage) to be automatically generated from a model in the model editor.

As an example, let's imagine we want to generate a manifest and associated files for the Airports model:

First right click on the package and choose deploy.

This will offer a number of deployment types.

Select XOCL.

Now choose the directory that you want to save the deployed code to.

Now create a file browser (right click on any Project in the browser) for the created directory.

Each of the classes in the model will have been deployed in code, along with a Manifest file.

The manifest file will contain all the relevant files:

```
parserImport XOCL;
parserImport Manifests;

// Manifest deployed by Andy on Sat Sep 24 17:22:04 BST 2005

@Manifest Components
  p = @File Components end
  @File Component end
  @File ComponentModel end
  @File Port end
  @File NamedElement end
  @File Connector end
do p
end;
```

Right click Compile, Build and Load to compile, build and load the manifest.

Reference

Table of Contents

16. Namespace, Classes, Packages and MetaClasses	125
Introduction	125
NameSpaces	125
Classes	125
Class Definition	125
Attributes	126
Operations	126
Constraints	127
Inheritance	127
Packages	128
Metaclasses	128
Message Passing	128
Object Creation	129
Slot Access	130
Slot Update	131
Default Parents	131
Example	131
17. Working with Syntax	134
Introduction	134
Grammar and Text Processing	134
Introduction	134
A Simple language Grammar	134
Debugging	140
XBNF Grammar	143
The Grammar Domain Model	143
The XBNF Grammar	145
Tokens	149
XMF Execution Architecture	149
Introduction	150
Performable Elements	150
Syntax Extensions	152
Synthesising Syntax	152
Introduction	153
The OCL Package	153
Examples	156
Quasi-Quotes	161
Introduction	161
Literal Syntax	162
Syntax Templates	162
Splicing	162
Patterns	162
New Performable Elements	162
Introduction	162
Sugar	163
Syntax	168
Exp	168
18. XCore	182
Introduction	182
The XCore Package	182
Attribute	183
Behavioural Feature	183
Bind	183
Classifier	183
Class	183
CodeBox	183

Collection	183
Compiled Operation	183
Constraint	183
Constraint Report	184
Constructor	184
Contained	184
Container	184
Daemon	184
Data Type	184
Dependency	184
Doc	184
DocumentedElement	184
Element	184
Enum	185
Exception	185
ForeignOperation	185
IndexedContainer	185
InterpretedOperation	185
Namespace	185
NamedElement	185
Object	185
Operation	186
OrderedCollection	186
Package	186
Parameter	186
Performable	186
Resource	186
Seq	186
Set	186
Snapshot	186
StructuralFeature	187
Table	187
Thread	187
TypedElement	187
Unordered Collection	187
Vector	187
19. XMap	188
Introduction	188
Language Basics	188
Syntax	189
Diagrammatical Syntax	189
Textual Syntax	190
Clause Syntax	190
Execution	191
Constructing Mappings	191
Creating Mappings via the Modelling Interface	191
Creating Mappings via the Programming Interface	194
Running Mappings	194
Example	195
Class Model to Database	195
Running the Mapping	200
Other Aspects of Mappings	201
Operations	201
Attributes	202
Variable Passing	202
20. XML	204
Introduction	204
XML	204

Parsing XML	205
Introduction	205
Example	205
Debugging XML Grammars	217
The XML Parsing Grammar	220
DOM Input Channel	220
SAX Input Channel	222
XML Output	226
Introduction	226
XML Output Patterns	226
XML Output Channels	233
Raising Events	234
Deploying Java	234
Introduction	234
Deploying Models	234
Deploying Parsers	234
Deploying Factories	234
21. Xocl	235
Introduction	235
Purpose	235
Language Basics	236
Overview of Syntax	237
Basic Data Types	237
Program Constructs	238
Self Evaluating Expressions	238
Variables and Update	239
Calling Operations	240
Infix Operators	240
Prefix Operators	241
Sequencing	241
Special Forms	241
Quasi-Quotes	242
The Meta Character @	242
Documentation	243
Error Handling	243
Control Statements	244
If	244
Case	244
CaseInt	245
TypeCase	246
While	246
For	247
Find	248
Iterators	248
Assignment	250
Pattern Matching	250
Patterns and Pattern Matching	250
Pattern Categories	251
Pattern Contexts	254
Data Type Operations	255
Boolean	255
Channels	255
Clients	256
Daemons	257
Elements	257
Integers	258
Floats	260
Objects	262

Null	262
Operations	263
Strings	265
Sequences	268
Sets	271
Symbols	273
Tables	273
Threads	274
Vectors	275
Debugging	275
Relationship to OCL and ASL	275
XOCL Grammar	276
Pattern Grammar	279
22. XTools	280
Introduction	280
The XTools Architecture	280
Introduction	280
Tool Component	280
Tool Event	281
Tool Definition	284
Tool Deployment	286
Diagram Tools	286
Introduction	286
Diagram Tool Components	287
Nodes and Edges	288
Toolbar	289
Menus	290
Diagram Events	292
An Example Domain Specific XTool	294
Display Elements	309
Diagram Layout	315
Example Tool: Class Diagrams	315
Form Tools	318
Introduction	318
Form Components	319
Menus on Forms	320
Form Events	320
Example: Airports	321
23. Clients	324
Introduction	324
Introduction	324
Message Based Client	324
Eclipse Implementation	324
XMF Implementation	330
Internal Clients	335
External (Socket Based) Clients	335

Chapter 16. Namespace, Classes, Packages and MetaClasses

Introduction

NameSpaces

Classes

XMF is a class-based object-oriented modelling environment. Each value in XMF has a type or classifier that describes its structure and behaviour. Values in XMF are divided into objects and non-objects. Object types are called classes and non-object types are called classifiers. If a value v is of a type c then we say that v is an instance of c. XMF is provided with a large number of classes and classifiers; XMF developers can define their own classes and classifiers as extensions of those provided.

Classes and classifiers classify their instances by running constraints. A constraint is a boolean valued expression that runs in the context of the current state of the candidate instance. The outcome of constraint checking is a constraint report containing details of the constraints that were performed, the candidates, the outcome and a reason for any constraints that failed. Constraint checking is a powerful mechanism for checking whether a model or a model scenario is correctly formed.

Class Definition

A class describes the structure and behaviour of its instances. A class has a name and lives in a name-space. The following is a basic class that lives in the name-space Root:

```
context Root  
  @Class EmptyClass  
end
```

By default, the class EmptyClass specializes the XMF class XCore::Object and provides a single constructor for creating instances: EmptyClass(). If we perform the following expression:

```
EmptyClass().isKindOf(EmptyClass)
```

then the result is true since the newly created instance is directly an instance of EmptyClass. In addition, the expression:

```
EmptyClass().isKindOf(Object) and EmptyClass().isKindOf(Element)
```

returns true since EmptyClass inherits (by default) from Object and Object inherits from Element (the class Element does not inherit from anywhere). The class EmptyClass is itself a value:

```
EmptyClass.isKindOf(Class) and  
EmptyClass.isKindOf(Classifier) and  
EmptyClass.isKindOf(NamedElement) and  
EmptyClass.isKindOf(Object) and  
EmptyClass.isKindOf(Element)
```

Attributes

A class typically defines some attributes that correspond to slots in the instances of the class. Each attribute has a name and a type and may optionally have some modifiers and an initial value. The following is a simple example of a class with attributes:

```
context Root
@class Point
@Attribute x : Integer end
@Attribute y : Integer end
@Constructor(x,y) ! end
end
```

A new point is created using the constructor defined by Point. A class may define any number of constructors; each constructor must have a different number of arguments. Each constructor argument corresponds to one of the attributes in the class. The optional modifier ! declares that the printed representation for a point is defined by that constructor.

A new Point is constructed:

```
Point(100,200)
```

where 100 is the value of the slot x and 200 is the value of the slot y. Accessor and updater operations are automatically produced by including attribute modifiers:

```
context Root
@class Point
@Attribute x : Integer (?,!) end
@Attribute y : Integer (?,!) end
@Constructor(x,y) ! end
end
```

The modifier ? defines that operations getX and getY are automatically provided; modifier ! defines that operations setX and setY are automatically provided:

```
let p = Point(100,200)
in p.setX(p.getX() - 1);
   p.setY(p.getY() - 1)
end
```

Operations

Object-oriented execution proceeds by message passing; a message consists of a name and some argument values. When a message is sent to an object, the name is looked up in the class of the object (and its parents); if an operation is found then it is invoked otherwise an error is reported. Operations may be defined as part of a class or a package definition or added to existing classes and packages via a context definition.

The following defines a class of stacks and adds the definition to a package named Stacks. The example shows the complete contents of a file containing the definition. XMF can be used in a file-based mode where files contain source code that is compiled using the XMF compiler. The compiled binary is then loaded into XMF.

```
parserImport XOCL;

context Stacks

@class Stack
```

```

@Attribute elements : Seq(Element) end
@Attribute index : Integer end

@Operation isEmpty():Boolean
    self.elements->isEmpty
end

@Operation pop()
    if self.isEmpty()
    then throw StackUnderflow(self)
    else
        let head = elements->head
        in self.elements := elements->tail;
            head
        end
    end
end

@Operation push(e:Element)
    self.elements := Seq{e | elements}
end

@Operation top()
    if self.isEmpty()
    then throw StackUnderflow(self)
    else elements->head
    end
end

```

Constraints

Inheritance

All classes are specializations or sub-classes of at least one other class. By default a class is a sub-class of XCore::Object. A class inherits constructors, attributes, operations and constraints from its parent. For example the following class specializes Point with a z co-ordinate:

```

context Root
@class Point3D extends Point
    @Attribute z : Integer (?,! ) end
    @Constructor(x,y,z) ! end
end

```

An instance of Point3D may be constructed using a two-place constructor or a three-place constructor (the three-place is probably more useful). An instance of the class Point3D is also an instance of the parent class Point:

```

let p = Point3D(1,2,3)
in p.isKindOf(Point)
end

returns true.

```

Multiple

A class may inherit from more than one super-class in which case the sequence of parents is given by comma separated paths after the extends keyword. The order of the parents is used to determine

the lookup order for operations, although it is considered bad practice to rely on the ordering when handling messages.

Multiple inheritance is most useful when inheriting orthogonal behaviour and structure that is then extended in the new sub-class.

Run-Super

When a sub-class extends a super-class it may shadow an existing operation by defining a new operation in the sub-class with the same name and arity as an inherited class. Sending a message to an instance of the sub-class will cause the shadowing operation definition to be performed.

Sometimes it is useful to be able to reference a shadowed operation from a shadowing operation body. This occurs when the shadowing class extends the behaviour of the shadowed class. XMF provides the keyword super that is used to invoke the shadowed operation from the shadowing operation.

This section provides a simple example of how super is used.

Packages

Metaclasses

XMF is an environment for language design and deployment. Languages control the structure and behaviour of the values that they denote. In this sense, language design is a meta-activity and requires a meta-language that represents the structure and behaviour of the language components.

XMF provides a meta-circular object-oriented kernel language called XOCL. The meta-circular property means that XOCL is defined completely in itself. This property validates XOCL as a meta-language. Object-orientation provides a basis for application extension and reuse through inheritance and modularity through encapsulation.

XOCL is both meta-circular and object-oriented, it is suitable for language definition where languages can be easily constructed as modular extensions of the basic XOCL language. Instantiations of XOCL are languages and extensions of XOCL are meta-languages.

All languages have key semantic features that can be represented as an interface in the definition of the language. Consider a language with an operational semantics. In this case programs written in the language may be viewed as controlling a machine that contains the state of the execution at any given snapshot in time. The key semantic features of the language form the API of the machine.

Given a language L, we would like to construct a new language that is L-like. If L is defined using object-oriented principles then it is attractive to construct the new language as an extension of L using inheritance. Syntax structures and values of the new language can be defined by extending the appropriate features of L. We would like to construct the semantics of the new language using the same approach. If we have constructed the semantics of L by encapsulating the key features as an implementation of the API as described above, then the new language semantics can be defined by inheriting and extending these operations as appropriate.

Where the semantics of a language has been constructed using object-oriented principles, the resulting collection of classes and operations is referred to as a meta-object-protocol. This section describes the XOCL MOP.

Message Passing

XOCL performs computation in terms of messages between elements. A message consists of a name and some data. A message is sent from a source element to a target element. The target element receives the message, performs appropriate computation and returns a result. Messages between elements

aresynchronous: the source element halts computation and waits the return value from the target element.

Message passing occurs when the source element performs an expression of the form: o.m(x,y,z,...) where o is the target element, m is the message name and x, y, z etc. is the data, or parameters, of the message.

Message passing is defined by the MOP component referred to as the message passing protocol. The protocol is defined by the meta-class of the target element and is called sendInstance:

```
context Element
  @Operation send(message,args):Element
    self.of().sendInstance(self,message,args)
  end
```

A default protocol is provided by the Kernel meta-class Classifier:

```
context Classifier
  @Operation sendInstance(element,name,args)
    // Get all the operations of element with the
    // correct name and arity. Select the most
    // specific and invoke it.
    let arity = args->size then
      operations = element.of().allOperations()
      ->asSeq
      ->select(o | o.name = name and o.arity() = arity)
    in if operations->isEmpty
      then
        element.error("Cannot handle "+message+"/"+arity)
      else operations->head.invoke(element,args)
      end
    end
  end
```

Since Class is a sub-class of Classifier, any sub-class of Class that defines a new sendInstance operation will provide a specialized message passing protocol for the instances of its instances. This can be used to implement specialized operation lookup mechanisms, to facilitate debugging information and to change the basic message passing mechanisms (for example by defining a class of objects with message queues).

Object Creation

Objects are created by sending a new message to a class together with some initialization data. The preferred way of invoking the new operation of a class is to apply the class as an operator to the initialization arguments. This is preferred because it is succinct and because the compiler and XMF VM can handle class instantiation more efficiently in this form:

```
context Classifier
  @Operation invoke(target:Element,args:Seq(Element)):Element
    self.new(args)
  end
```

The new operation is defined by the meta-class of the receiving class. It constitutes the instantiation protocol for a collection of classes. The class Classifier defines the default instantiation protocol:

```
context Classifier
  @Operation new(args:Seq(Element)):Element
    self.new().init(args)
  end
```

where the operation new creates an empty new instance of the receiver and init initializes the new instance.

Sub-classes of Classifier can define their own instantiation protocol. Typically this will use super to create an instance using the default protocol and then perform some extra computation to initialize the new instance; however, in principle the instantiation protocol can by-pass the default protocol altogether. To create a raw instance of a class C and add a single slot named "x" with initial value 10 you can do the following:

```
let o = Kernel_mkObj()
in Kernel_setOf(o,C);
   Kernel_addAtt(o,Symbol("x",100));
o
end
```

Using the kernel-level operations, you can create a completely bespoke instantiation protocol.

Slot Access

Objects have internal storage in the form of named slots. Access to a slot value is via the object and the name of the slot. Slots are named using symbols. Access is defined by the object's slot access protocol. The slot access protocol is used when an expression of the form o.a is performed. Access involves checking that the slot exists and then accessing the value of the slot.

The existence of a slot can be checked using the hasSlot operation defined by Object:

```
context Object
@Operation hasSlot(name):Boolean
  self.of().hasInstanceSlot(self,name)
end
```

The hasSlot operation invokes the hasInstanceSlot operation of the object's class. hasInstanceSlot forms part of the slot access protocol for the object; the operation is defined by the object's meta-class. The default definition is provided by Class and uses the kernel operation Kernel_hasSlot to directly check whether there is a machine-level slot:

```
context Class
@Operation hasInstanceSlot(object,name)
  Kernel_hasSlot(object,name)
end
```

Access to a slot's value is provided by the operation get defined by Object:

```
context Object
@Operation get(name:String):Element
  self.of().getInstanceSlot(self,name)
end
```

The operation getInstanceSlot is defined by an object's meta-class and describes how to access the storage associated with an object and a slot name. The default protocol is provided by Class and uses the kernel operation Kernel_getSlotValue to access the machine-level slot (as added using Kernel_addAtt):

```
context Class
@Operation getInstanceSlot(object,name)
  Kernel_getSlotValue(object,name)
end
```

Typically a new slot access protocol is required because a collection of classes implement object storage in a non-standard way (for example using a table, in a data base or distributed over a network).

Slot Update

The value of an object's slot can be updated using the object's slot update protocol. A slot is updated when an expression of the form `o.a := e` is performed. The class `Object` provides an operation used to set the value of a slot:

```
context Object
@Operation set(name:String,value:Element):Element
    self.of().setInstanceSlot(self,name,value);
    self
end
```

The object's meta-class defines an operation `setInstanceSlot` that forms the update protocol. The default update protocol is defined by `Class` and uses the kernel-level operation `Kernel_setSlotValue` to update the machine-level slot and to invoke any daemons that are defined on the object:

```
context Class
@Operation setInstanceSlot(object,name,value)
    Kernel_setSlotValue(object,name,value)
end
```

A new slot update protocol is used to circumvent the default storage. For example the storage for a slot may be in a database or accessed over a network.

Default Parents

A class is created as an instance of a meta-class. When a class is created it must have some parents. The meta-class defines an operation `defaultParents` that produces a set of classes that are the default parents for its instances. The basic definition for `defaultParents` is provided by `Classifier`:

```
context Classifier
@Operation defaultParents():Set(Classifier)
    Set{Element}
end
```

Most classes are instances of the class `Class`, that overrides the definition as follows:

```
context Class
@Operation defaultParents():Set(Classifier)
    Set{Object}
end
```

Example

Suppose that we want to define a class of objects that can have standard attribute defined slots in addition to dynamic slots. Attribute defined slots are defined at the class level. Dynamic slots are defined at the object level and can be added and removed dynamically. Both types of slot can be accessed and updated via the standard protocols using `o.a` and `o.a := e` expressions.

In order to implement these objects we require a new slot access and update protocol. The protocol is defined at the meta-level and is to be called the `Elastic` protocol. We require two new classes: `ElasticObject` that is the super-class of all user-defined elastic classes; and, `ElasticClass` that defines the elastic protocol.

The class `ElasticObject` uses a table to contain the dynamic slots:

```
context Root
@Class ElasticObject
```

```
    @Attribute slots : Table = Table(100) end
end
```

An elastic object provides operations to add and remove the dynamic slots:

```
context ElasticObject
@Operation addSlot(name:String,value)
  slots.put(Symbol(name),value)
end

context ElasticObject
@Operation removeSlot(name)
  slots.remove(Symbol(name))
end
```

An elastic object can remove all the dynamic slots:

```
context ElasticObject
@Operation removeAll()
  @For key inTableKeys slots do
    self.removeSlot(key.toString())
  end
end
```

An elastic object can increment the values of all the dynamic slots. Note that incAll uses the slot access and update protocol for the object to change the value of the dynamic slots:

```
context ElasticObject
@Operation incAll()
  @For key inTableKeys slots do
    self.set(key,self.get(key) + 1)
  end
end
```

The class ElasticClass defines the elastic MOP:

```
context Root
@Class ElasticClass extends Class
end
```

ElasticObject must be a parent of any elastic class:

```
context ElasticClass
@Operation defaultParents()
  Set{ElasticObject}
end
```

The elastic slot access protocol inspects the slots table to see if the required slot is defined there. If not then the protocol uses super to revert to the default protocol inherited from Class:

```
context ElasticClass
@Operation getInstanceSlot(object,name)
  if Kernel_getSlotValue(object,Symbol("slots")).hasKey(name)
    then Kernel_getSlotValue(object,Symbol("slots")).get(name)
    else super(object,name)
  end
end

context ElasticClass
@Operation setInstanceSlot(object,name,value)
```

```

if Kernel_getSlotValue(object,Symbol("slots")).hasKey(name)
then Kernel_getSlotValue(object,Symbol("slots")).put(name,value)
else super(object,name,value)
end
end

context ElasticClass
@Operation hasInstanceSlot(object,name)
if Kernel_getSlotValue(object,Symbol("slots")).hasKey(name)
then true
else super(object,name)
end
end

```

There is no specific need for sendInstance in the elastic protocol, however it is defined for completeness and simply prints a message before reverting to the default protocol:

```

context ElasticClass
@Operation sendInstance(element,message,args)
format(stdout,"Sending message ~S(~{,~;~S~})~%",Seq{message,args});
super(element,message,args)
end

```

There is no specific need for new in the elastic protocol, however it is defined for completeness and simply prints a message before reverting to the default protocol:

```

context ElasticClass
@Operation new(args)
format(stdout,"Creating a new instance of an elastic class~%");
super(args)
end

```

The following is an example class definition that specifies its meta-class as ElasticClass:

```

context Root
@Class C metaclass ElasticClass
@Attribute s : Element end
@Operation test()
self.addSlot("x",100);
self.addSlot("y",200);
self.addSlot("z",300);
self.x := self.x + 1;
self.y := self.y + 1;
self.z := self.z + 1;
self.incAll();
self.s := self.x + self.y + self.z;
self.removeAll();
s
end
end

```

Chapter 17. Working with Syntax

Introduction

This chapter describes XMF-Mosaic's powerful facilities for capturing and parsing textual syntaxes. This is important when you wish to develop using a specialised textual programming language, or create parsers for existing programming languages.

This document describes how to use XMF-Mosaic to parse textual languages and consequently synthesize elements. A textual language is parsed when it is consumed character-by-character and checked against rules (a grammar) governing legal sequences of characters. As text is parsed, the grammar rules can perform actions that construct (or synthesize) new elements.

A textual language can synthesize anything. For example, if we define a domain model representing two-dimensional tables then a textual language can be defined that, when parsed, synthesizes instances of the class `Table2D`.

All textual interaction with XMF-Mosaic (for example file based or console based) is governed using a parser based on a grammar for XOCL. In this case the parser synthesizes XOCL program code that is subsequently passed to the XOCL compiler or the command interpreter.

XOCL provides a powerful mechanism that allows the XOCL grammar to be arbitrarily extended with new language features (becoming an extensible language). User defined grammars can be added incrementally as modules to the XOCL grammar. Providing the new extension synthesizes valid program code then the new language construct is assimilated into XOCL.

Working with grammars that synthesize program code occurs frequently when defining textual languages. The synthesizing actions are often required to produce large amounts of program code from a small amount of text (after all this is the point of defining a language). XMF provides technology that vastly reduces the amount of work necessary to define the synthesizing actions; the technology is called quasi-quotes.

Quasi-quotes are used to define code templates. A code template is a mapping from code fragments to a program by inserting the code fragments into a code pattern. The result of applying a template can be supplied as an argument to subsequent templates. Very large programs can be synthesized from humble beginnings.

Grammar and Text Processing

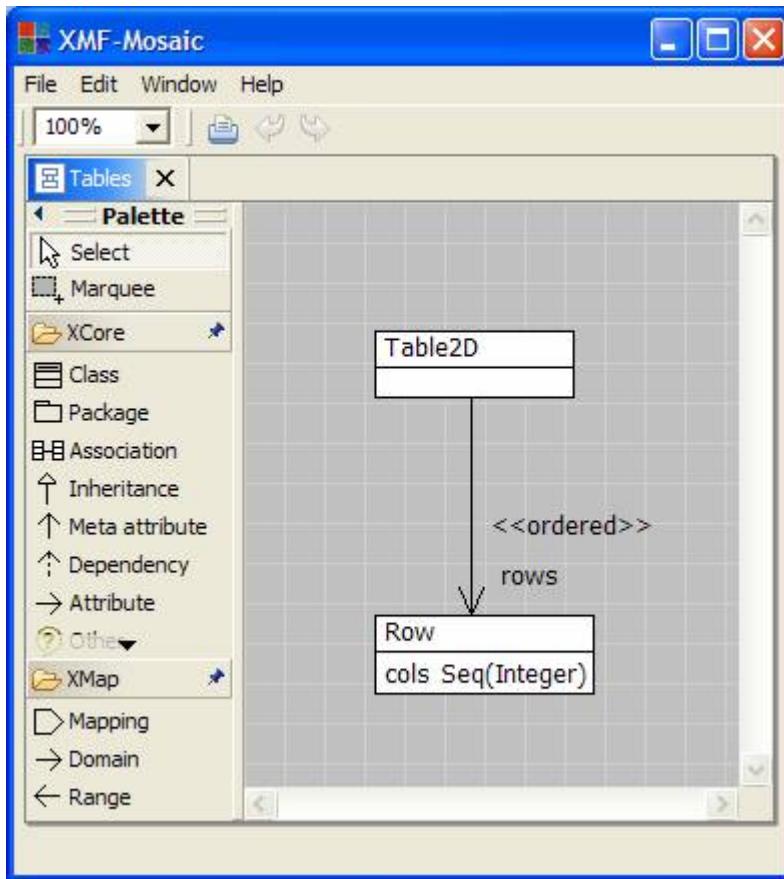
Introduction

A Simple language Grammar

To construct a minimal textual language we must define a domain model and decide how instances of the model will look when rendered in text. We must then define how to produce the textual representation from an instance and how to synthesize an instance from a textual representation.

This section shows how all aspects of this process are achieved in XMF-Mosaic. We will assume that we are defining a language from a domain model rather than reverse engineering a domain model from a textual language. All aspects apply to both of these activities.

Our domain model is that for two-dimensional tables of integers. A table consists of a collection of rows; each row is a sequence of column values. The model is shown below:



Once we have decided on a domain model we can define a textual representation for instances of the model. There may be more than one representation (when dealing with legacy languages this is likely to be the case). If instances can be synthesized from more than one textual representation then this can easily be accommodated by defining more than one grammar. In this example we will keep it simple by defining a single representation as follows:

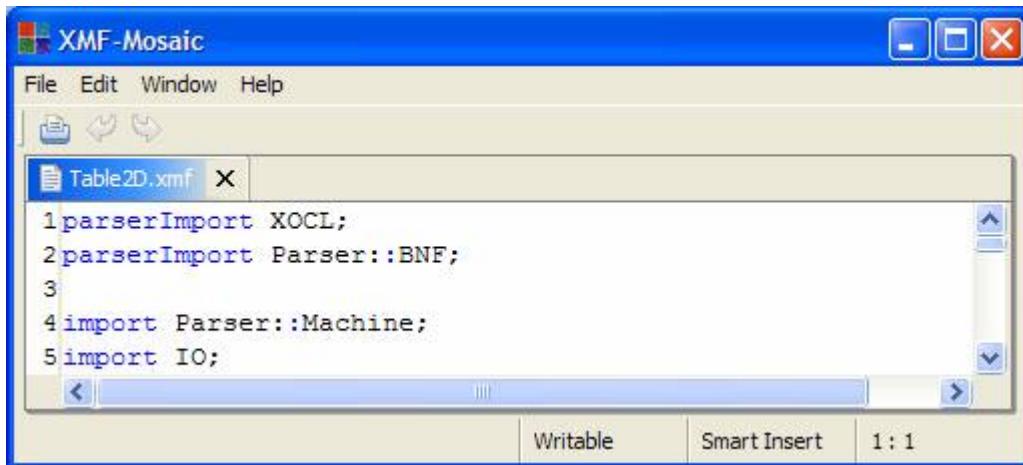
```

1table
2 row 1 2 3 end
3 row 2 3 4 end
4 row 5 6 7 end
5 end

```

The task is now to define how to translate from instances of the model to the text and back again. The rest of this section steps through the complete definition of the model, its mapping to text and its synthesis from text. The example is given as a complete XOCL program that can be compiled and loaded.

The header of the file must import all the name spaces necessary to define the mappings:



```

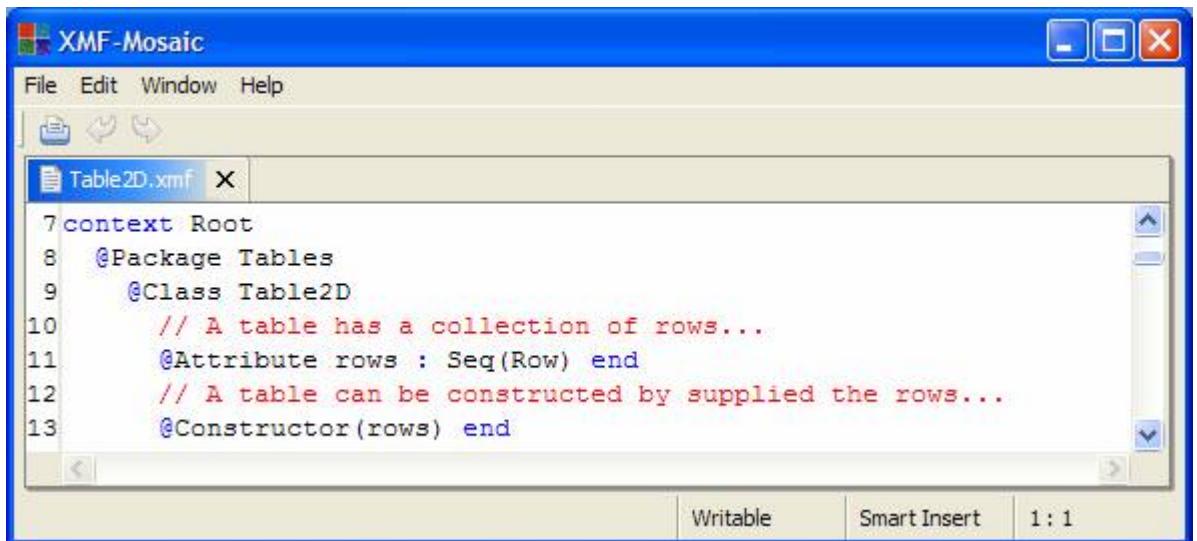
1 parserImport XOCL;
2 parserImport Parser::BNF;
3
4 import Parser::Machine;
5 import IO;

```

Lines 1 and 2 import grammars that define parsing rules used in the rest of the file. Line 1 imports the XOCL grammar that allows us to write XOCL code. Virtually all program files start with this line. Line 2 imports the grammar that defines the textual languages for writing grammars (the parsing and synthesis rules for writing grammars are written in themselves).

Lines 4 and 5 import name spaces that define names referenced in the rest of the file. Line 4 imports the XMF parsing machinery that allows us to create a parse machine state and set it running. Line 5 imports the IO package that allows us to open files, read and write channels.

We have already seen the domain model. It was originally defined in a text file and then displayed as a diagram. The structure of tables is defined below:

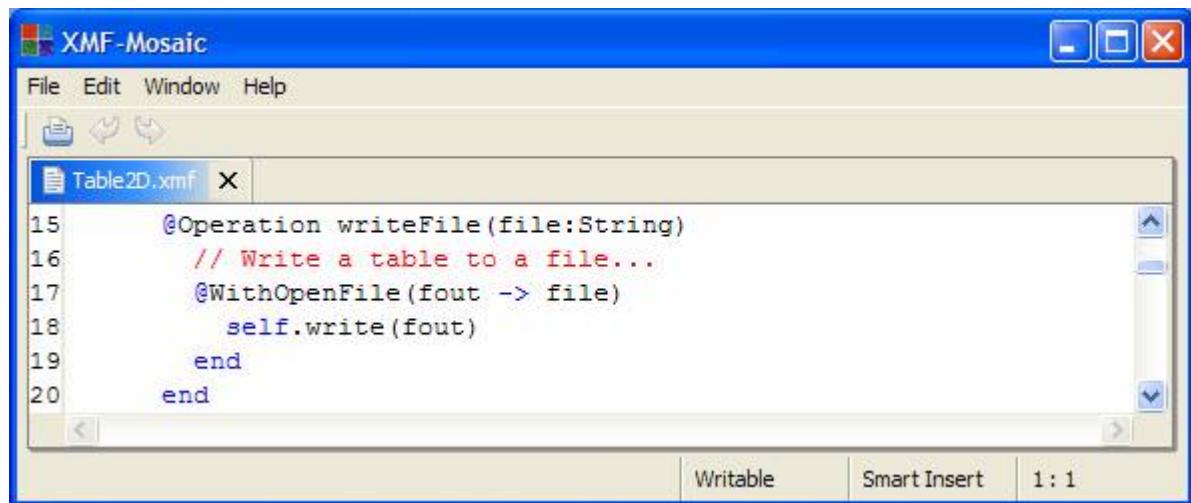


```

7 context Root
8   @Package Tables
9     @Class Table2D
10    // A table has a collection of rows...
11    @Attribute rows : Seq(Row) end
12    // A table can be constructed by supplied the rows...
13    @Constructor(rows) end

```

We would like to be able to store tables in files. The following operation opens an output file channel and supplies it to the write operation for the table. By separating out the file name and the output operation we allow tales to be sent to other types of output channel (such as stdout):

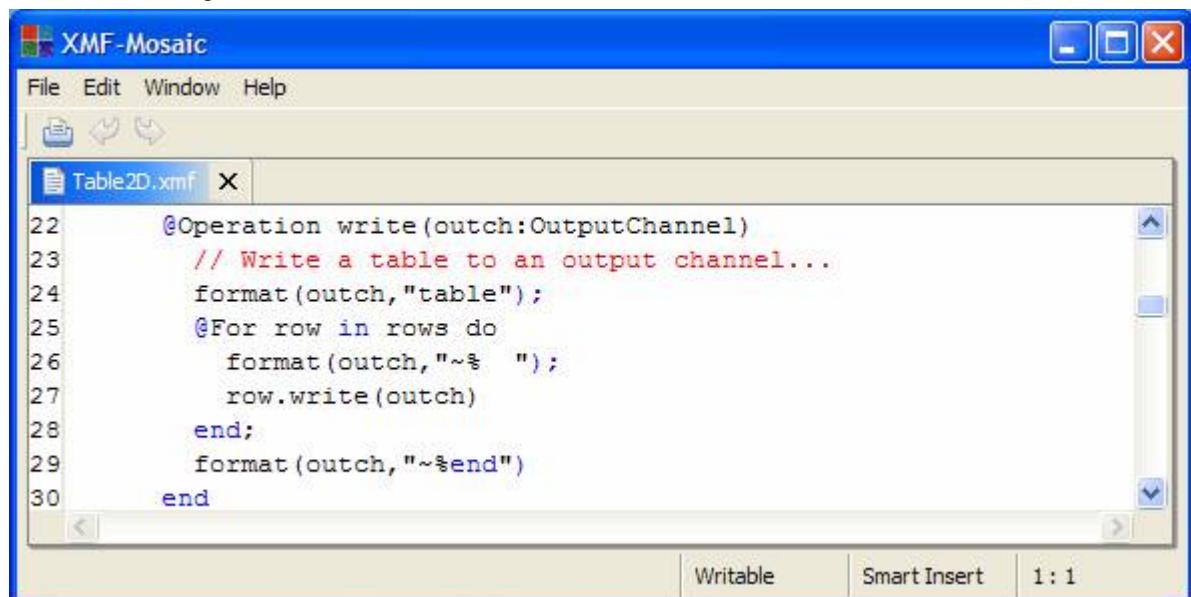


The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. A central window titled "Table2D.xmf" contains the following code:

```
15     @Operation writeFile(file:String)
16         // Write a table to a file...
17         @WithOpenFile(fout -> file)
18             self.write(fout)
19         end
20     end
```

The code uses XML-like syntax with annotations (@Operation, @WithOpenFile) and self-references (self.write). The status bar at the bottom right shows "Writable", "Smart Insert", and "1:1".

The output operation write is defined below. It formats the table to the output channel and sends each row a write message.

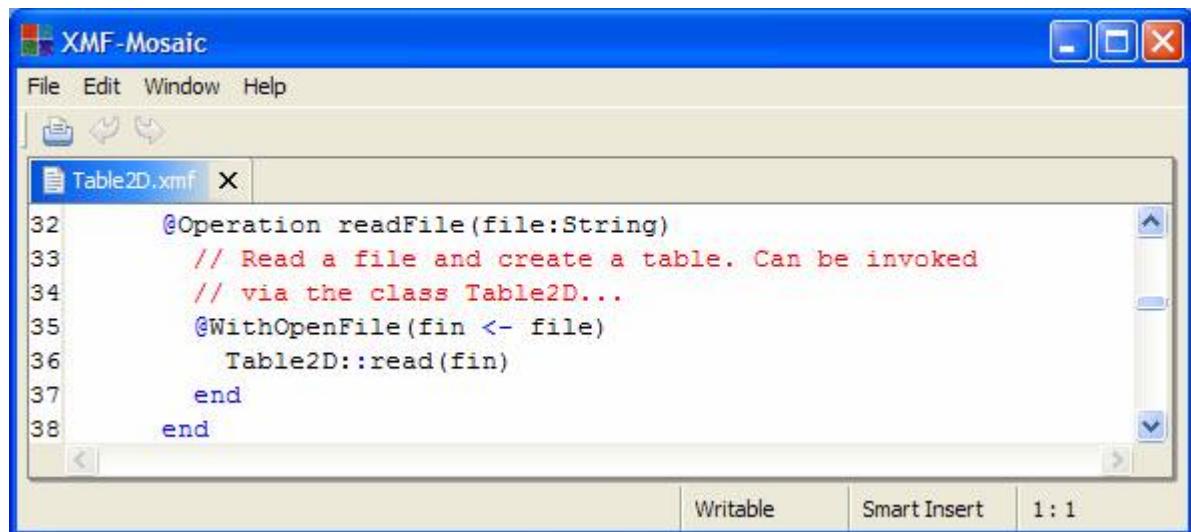


The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. A central window titled "Table2D.xmf" contains the following code:

```
22     @Operation write(outch:OutputChannel)
23         // Write a table to an output channel...
24         format(outch,"table");
25         @For row in rows do
26             format(outch,"~% ");
27             row.write(outch)
28         end;
29         format(outch,"~%end")
30     end
```

This code defines an operation "write" that takes an output channel as a parameter. It formats the table as "table", then iterates over rows, sending each row to the output channel. The status bar at the bottom right shows "Writable", "Smart Insert", and "1:1".

We want to be able to read formatted tables from text files. We do this to create a table and are unlikely to read a table by sending an existing table a message. XMF allows operations to be defined in classes and called via the class (much like static methods in Java). Of course the value of self is not defined in operations that are used this way.



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". The toolbar has icons for new file, open file, save file, and close file. The main window displays a code editor with the file "Table2D.xmf" open. The code is as follows:

```

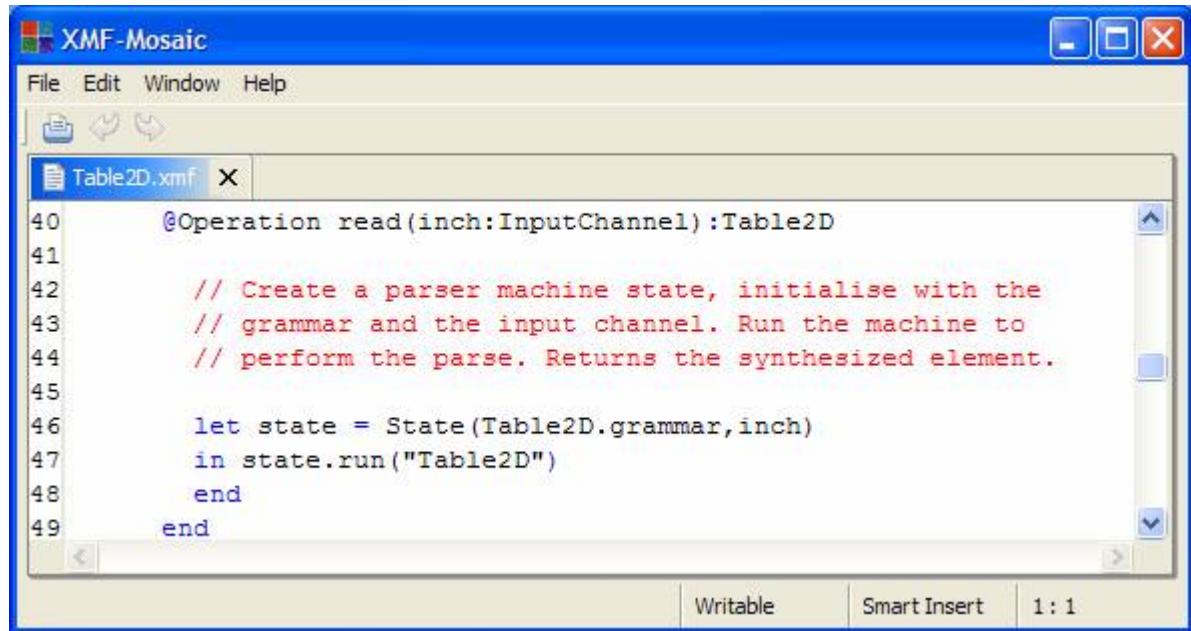
32     @Operation readFile(file:String)
33         // Read a file and create a table. Can be invoked
34         // via the class Table2D...
35         @WithOpenFile(fin <- file)
36             Table2D::read(fin)
37         end
38     end

```

The code editor has status bars at the bottom labeled "Writable", "Smart Insert", and "1:1".

Note that in line 36 we reference the operation read via the containing class Table2D.

The read operation must parse the text from the supplied input channel and synthesize a table (assuming that the input is syntactically correct). To parse an input source we must create an instance of a parsing machine initialised with a grammar and the input source. The package Parser::Machine defines the class State that is an initial parsing machine state:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". The toolbar has icons for new file, open file, save file, and close file. The main window displays a code editor with the file "Table2D.xmf" open. The code is as follows:

```

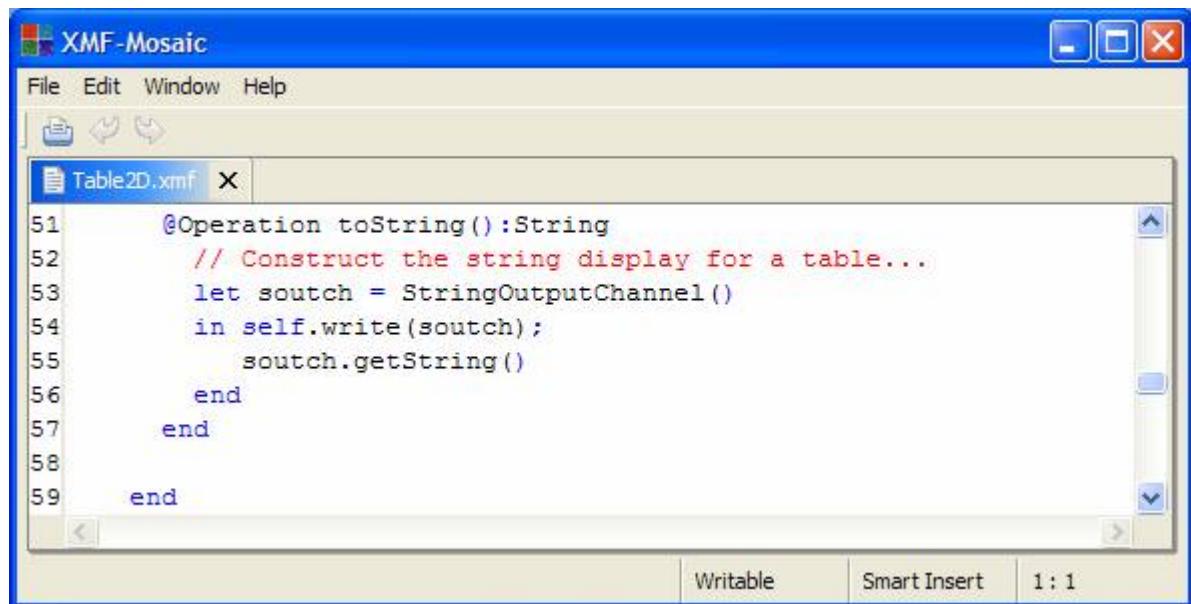
40     @Operation read(inch:InputChannel) :Table2D
41
42         // Create a parser machine state, initialise with the
43         // grammar and the input channel. Run the machine to
44         // perform the parse. Returns the synthesized element.
45
46         let state = State(Table2D.grammar,inch)
47         in state.run("Table2D")
48         end
49     end

```

The code editor has status bars at the bottom labeled "Writable", "Smart Insert", and "1:1".

A parsing machine state provides an operation run that is supplied with a starting non-terminal name. The machine will use the rule with the supplied name to start the parse. If the parse is successful then the synthesized value is returned, otherwise an exception is raised describing the parse error.

We can take advantage of the operation that writes a table to an output channel to produce a convenient string representation for a table. The XMF command interpreter always uses the `toString` operation to display values at the console. The following operation uses a string output channel to capture the string representation of a table:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". The toolbar has icons for file operations. A window titled "Table2D.xmf" contains the following code:

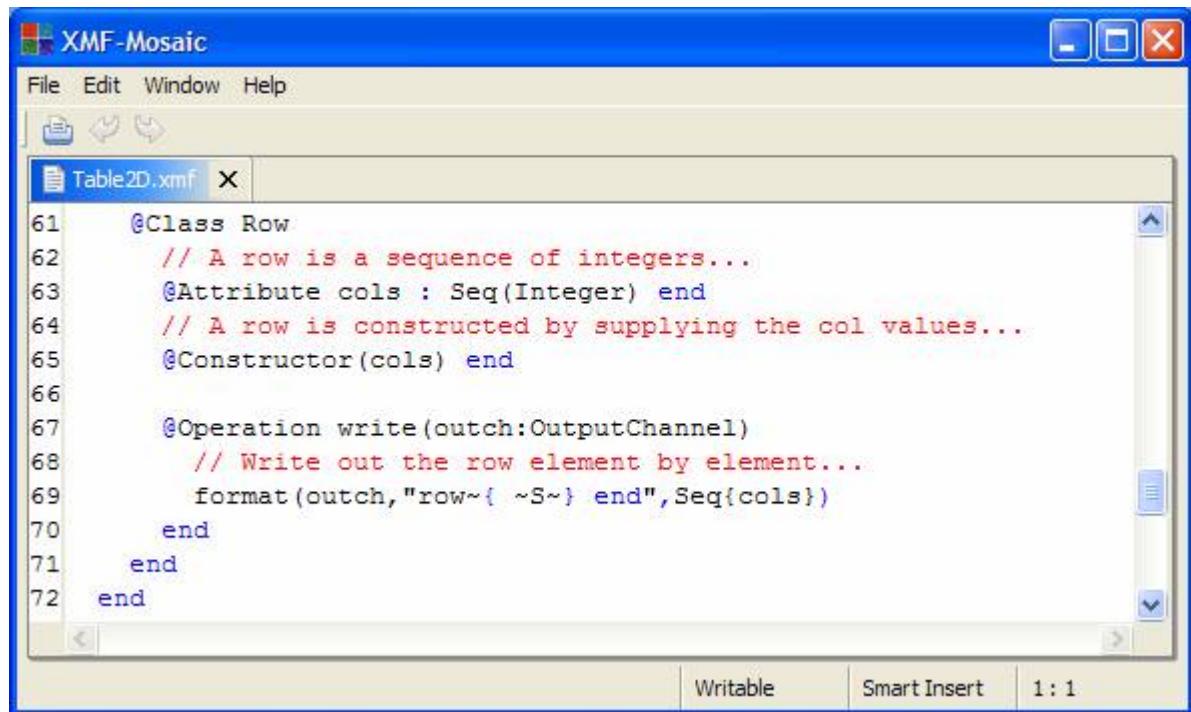
```

51     @Operation toString():String
52         // Construct the string display for a table...
53         let soutch = StringOutputChannel()
54         in self.write(soutch);
55             soutch.getString()
56         end
57     end
58
59 end

```

At the bottom of the window are buttons for "Writable", "Smart Insert", and a ratio "1:1".

The class Row is defined simply as follows:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". The toolbar has icons for file operations. A window titled "Table2D.xmf" contains the following code:

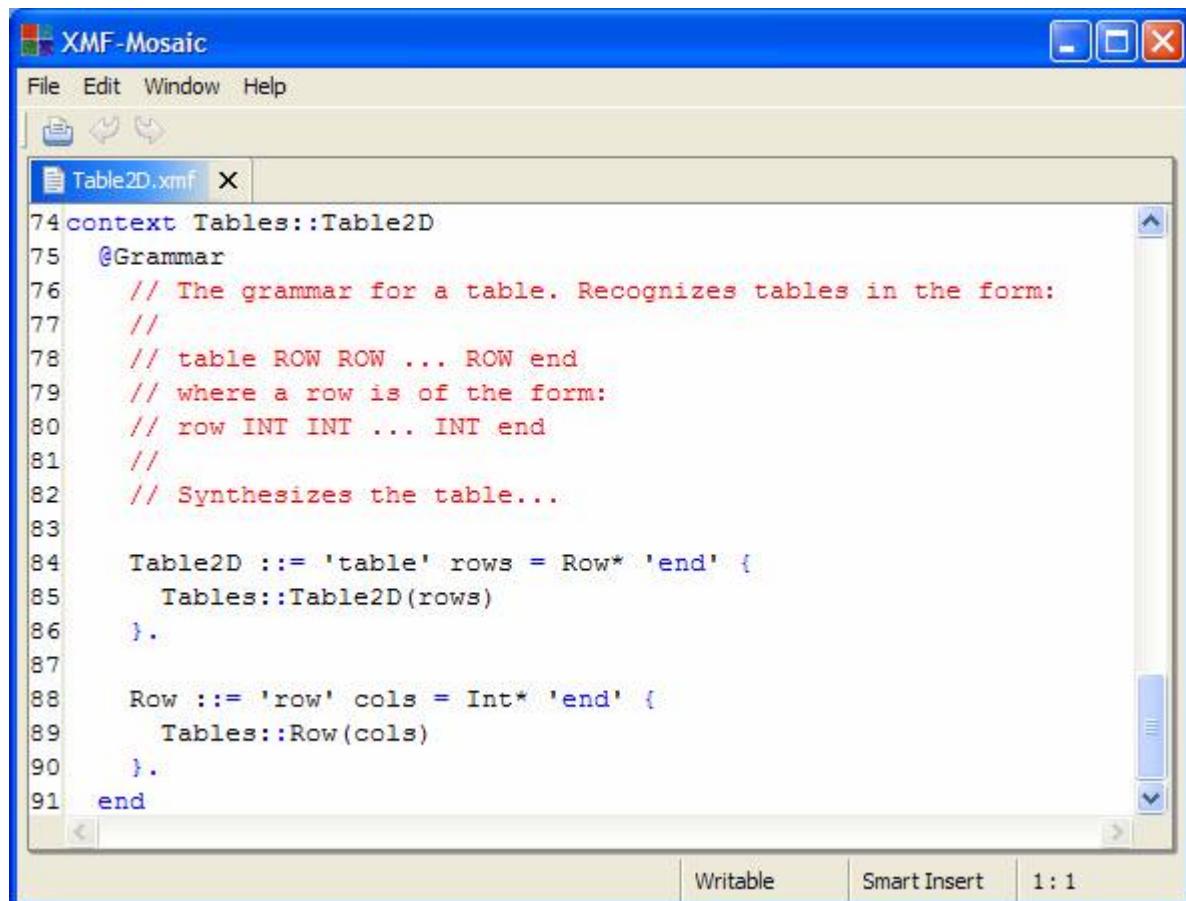
```

61     @Class Row
62         // A row is a sequence of integers...
63         @Attribute cols : Seq(Integer) end
64         // A row is constructed by supplying the col values...
65         @Constructor(cols) end
66
67         @Operation write(outch:OutputChannel)
68             // Write out the row element by element...
69             format(outch,"row~{ ~S~} end",Seq{cols})
70         end
71     end
72 end

```

At the bottom of the window are buttons for "Writable", "Smart Insert", and a ratio "1:1".

The language is complete except for the definition of a grammar. A grammar definition can occur within a class definition or can be added to a class using a context definition. It is somewhat a matter of taste; in general it is better to organise XMF source code as a collection of small files that are composed using a manifest file. We have separated out the grammar definition in this example to keep things simple:



```

File Edit Window Help
Table2D.xmf X
74 context Tables::Table2D
75   @Grammar
76   // The grammar for a table. Recognizes tables in the form:
77   //
78   // table ROW ROW ... ROW end
79   // where a row is of the form:
80   // row INT INT ... INT end
81   //
82   // Synthesizes the table...
83
84   Table2D ::= 'table' rows = Row* 'end' {
85     Tables::Table2D(rows)
86   }.
87
88   Row ::= 'row' cols = Int* 'end' {
89     Tables::Row(cols)
90   }.
91 end

```

Writable Smart Insert 1:1

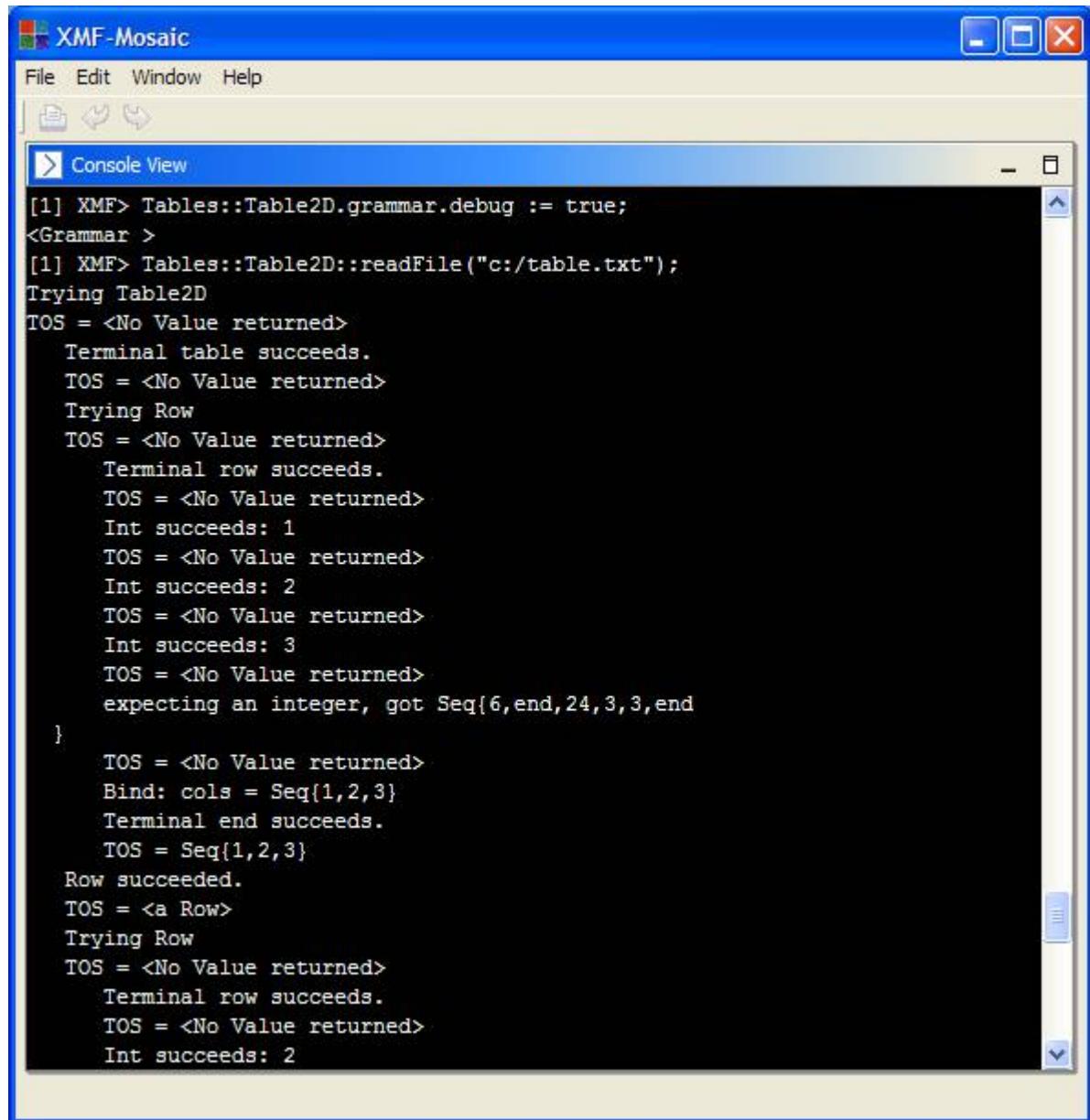
A grammar consists of rules or clauses that define how to parse and synthesize individual syntax groups. In the example above we have the group of rows (lines 88 – 90) and the group of tables (lines 84 – 86).

A table is defined to be the keyword table followed by a sequence of rows, followed by the keyword end. If this is successfully recognized then the action (contained within { and }) synthesizes an instance of the class Table2D (a restriction on grammars is that absolute paths should be used to reference named elements). The variable rows is bound to the sequence of rows recognized in line 84. Variables bound in this way can be referenced in actions; rows is used to initialize the new instance of the table in line 85.

A row is defined to be the keyword row followed by a sequence of integers, followed by the keyword end. The name Int is built in to XMF grammars and defines the syntax of integers.

Debugging

When parsing input it is possible that there is a bug in the grammar. This can lead to incorrectly parsed input or an erroneously reported parse error. XMF provides a trace mechanism that allows you to see the steps of the parsing machine as it proceeds. Each grammar has a boolean slot called debug that can be used to toggle the trace information. The following example shows part of the trace output for the table language:



The screenshot shows the XMF-Mosaic application window with a "Console View" tab selected. The console output displays a parse trace for a grammar named "Tables::Table2D". The trace shows the parser trying different clauses like "Table2D", "Row", and "Int". It uses "TOS" to represent the top of the stack. The trace indicates successful matches ("Terminal succeeds."), stack updates ("TOS = <No Value returned>"), and failed expectations ("expecting an integer, got Seq{6,end,24,3,3,end}"). The parser successfully constructs a "Seq{1,2,3}" object from the input.

```
[1] XMF> Tables::Table2D.grammar.debug := true;
<Grammar >
[1] XMF> Tables::Table2D::readFile("c:/table.txt");
Trying Table2D
TOS = <No Value returned>
Terminal table succeeds.
TOS = <No Value returned>
Trying Row
TOS = <No Value returned>
Terminal row succeeds.
TOS = <No Value returned>
Int succeeds: 1
TOS = <No Value returned>
Int succeeds: 2
TOS = <No Value returned>
Int succeeds: 3
TOS = <No Value returned>
expecting an integer, got Seq{6,end,24,3,3,end}
}
TOS = <No Value returned>
Bind: cols = Seq{1,2,3}
Terminal end succeeds.
TOS = Seq{1,2,3}
Row succeeded.
TOS = <a Row>
Trying Row
TOS = <No Value returned>
Terminal row succeeds.
TOS = <No Value returned>
Int succeeds: 2
```

As the parse proceeds the trace shows the names of the clauses that are tried. At each step the parse state has a stack that maintains the values synthesized by parse actions. The top value on this stack is displayed in the trace (TOS =). The trace displays when terminals are successfully matched. The trace also displays when expectations are not matched by the input (not necessarily indicating an error). For example, the grammar entry Int* is terminated when an integer is expected but the terminating keyword end is encountered.

The following tool snapshot shows a very simple language definition for performing arithmetic calculations. The example shows that grammars can be defined as a stand-alone global variable (Calculator), although they are more often used in conjunction with classes. The console shows the result after compiling and loading the file and invoking the calculator operation:

The screenshot shows the XMF-Mosaic tool interface. The main window displays the file `Calc.xmf` containing a BNF-like grammar definition. The grammar defines a `Root::Calculator` with rules for `Mult`, `Add`, and `Int` elements, along with a context block for `Root` that includes an `@Operation calc()`. The console view at the bottom shows the output of running the `calc()` operation, which performs the calculation $100 * 2 + 3$ and outputs the result `500`.

```

1 parserImport Parser::BNF;
2
3 import Parser::Machine;
4
5 Root::Calculator :=
6   @Grammar
7     Calc ::= Mult '='.
8
9     Mult ::= n1 = Add (
10       '*' n2 = Mult { n1 * n2 } |
11       '/' n2 = Mult { n1 / n2 } |
12       { n1 }
13     ).
14
15     Add ::= n1 = Int (
16       '+' n2 = Add { n1 + n2 } |
17       '-' n2 = Add { n1 - n2 } |
18       { n1 }
19     ).
20
21   end;
22
23 context Root
24   @Operation calc()
25     let state = State(Calculator,stdin)
26     in state.run("Calc")
27   end
28 end

```

Console View:

```

> Console View
calc();
100 * 2 + 3 =
500
[1] XMF> calc();

```

Tool status:

- Writable
- Smart Insert
- 21:3

The following tool snapshot shows a grammar for a tree language (based on XML). The language synthesizes a nested collection of sequences. It is interesting because it shows the use of predicates in the grammar (line 15) where values bound in from actions in a clause are used to perform checks during the parse. The start and end tag of a composite element must be the same for the parse to succeed.

The screenshot shows the XMF-Mosaic IDE interface. The main window displays the XBNF grammar file `SimpleXML.xmf`. The code defines a parser import, imports Parser::Machine and Parser::BNF, and defines a Root::Tree grammar. It includes rules for Element, SingleElement, CompositeElement, and Seq. The context is set to Root, and an operation tree() is defined. The bottom panel, titled "Console View", shows the execution of the grammar. The output shows the creation of a root node with two children, each having a child of its own, resulting in a total structure of four nodes. The console also displays GC collection information.

```

1 parserImport Parser::BNF;
2
3 import Parser::Machine;
4
5 Root::Tree :=
6   @Grammar
7     Element ::= 
8       SingleElement | CompositeElement.
9     SingleElement ::= 
10      '<' tag = Name '>' { Seq{tag} }.
11     CompositeElement ::= 
12      '<' tag1 = Name '>'
13        children = Element*
14      '</' tag2 = Name '>'
15      ? tag1 = tag2
16        { Seq{tag1 | children} }.
17      end;
18
19 context Root
20   @Operation tree()
21     let state = State(Tree,stdin)
22     in state.run("Element")
23   end
24 end

[1] XMF> tree();
[GC 72% collected, 16MB used, 42MB available.]
<root>
<child1/>
<child2>
<grandchild/>
</child2>
</root>
Seq{root,Seq{child1},Seq{child2,Seq{grandchild}}}
[1] XMF>

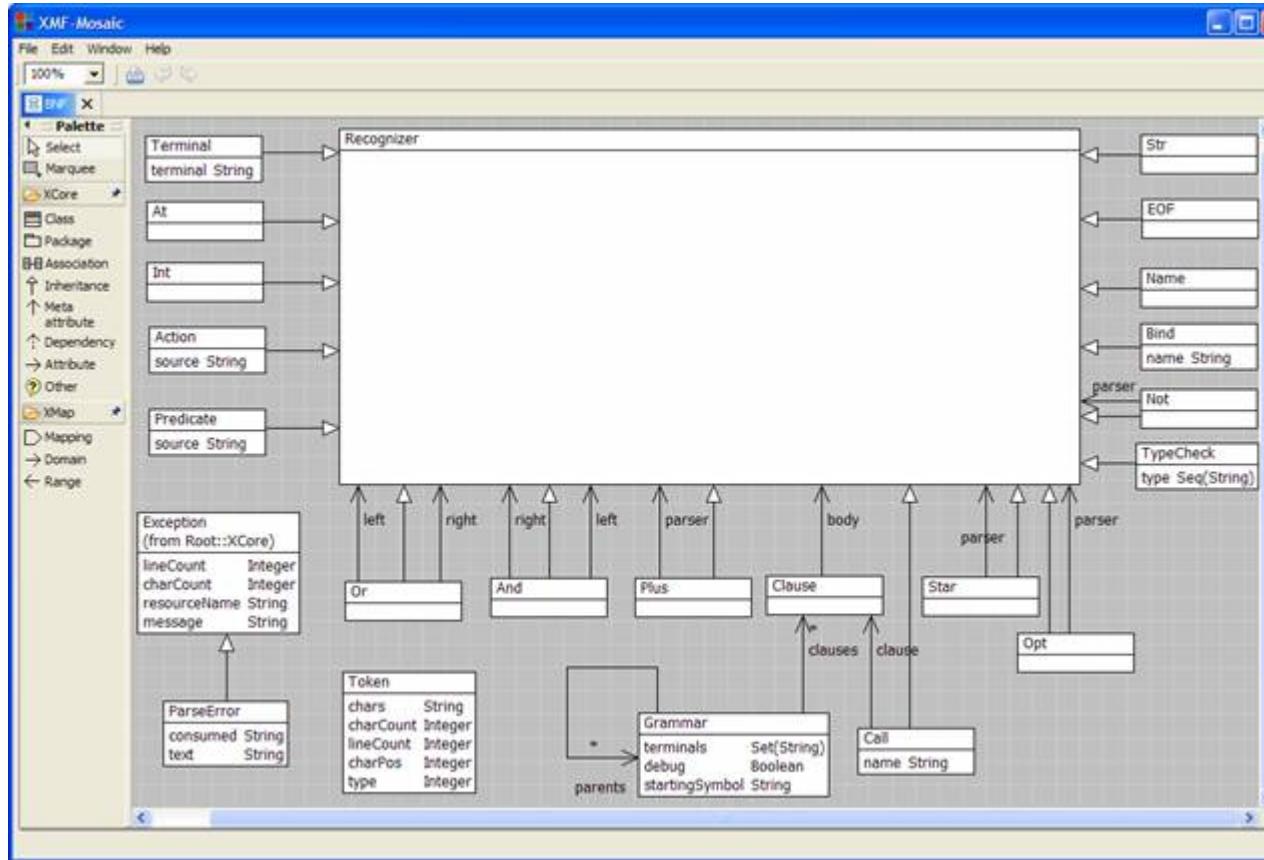
```

XBNF Grammar

XMF grammars are written in a textual language called XBNF. This language is very like extended BNF with actions written in XOCL. XBNF itself can be considered a language that supports language definition; it has a domain model and a grammar.

The Grammar Domain Model

An instance of the following model is synthesized when you define an XBNF grammar:

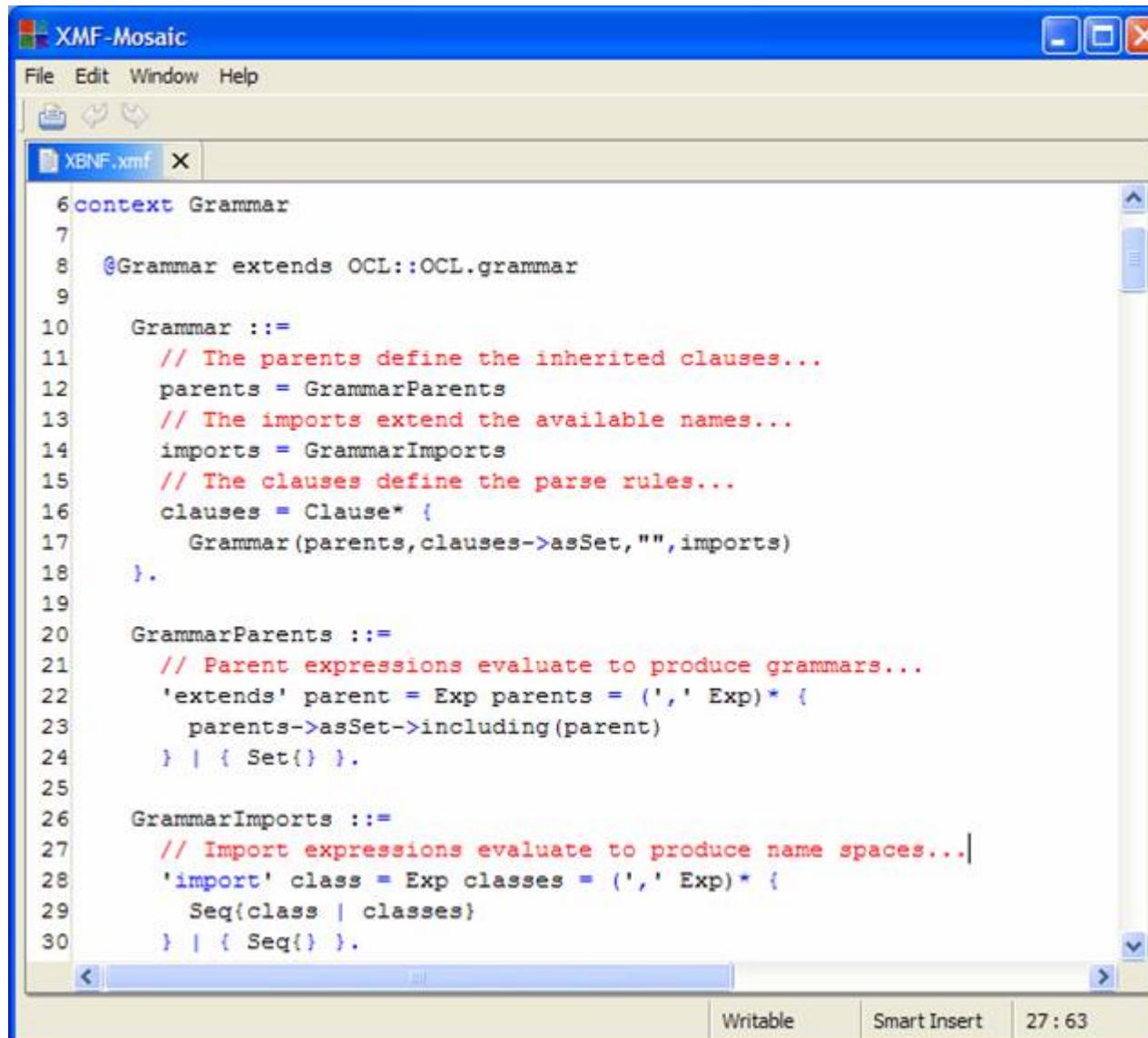


The rest of this section provides an overview of the components of the XBNF model.

- **Grammar:** A grammar defines a number of clauses that are used to parse input and synthesize elements. A grammar may have a collection of parent grammars from which it inherits clauses. If a grammar inherits multiple clauses with the same name then they are merged using disjunction.
- **Clause:** A clause is a parse rule. The name of the clause is a non-terminal of the grammar and can be called from other clauses in the grammar. The body of a clause is a recognizer which defines how to parse input and synthesize elements.
- **Recognizer:**
- **Action:** An action consumes no input and cannot cause the parse to succeed or fail. When an action is encountered in the parse it is evaluated to synthesize a value. The value produced by the action is pushed onto the parse stack.
- **And:** A conjunction of recognizers is used to sequence the left then the right components when encountered during a parse. This succeeds when the left then the right components succeed and synthesizes the value produced by the right component.
- **At:** A meta-character used to define how to escape from the current grammar and switch to another grammar.
- **Bind:** A bind has a name and a component recognizer. It succeeds when the component succeeds and binds the name to the synthesized value.
- **Call:** A call references a clause by name. Control switches to the named clause and succeeds when the named clause succeeds. The value synthesized by the parse is that produced by the called clause.
- **EOF:** Succeeds when the end of input is encountered.

- Int : Succeeds when the next input token is an integer; synthesizes the integer.
- Name : Succeeds when the next input token is a name; synthesizes a string.
- Not : Succeeds when the component recognizer fails. Note that this does not synthesize anything and bindings occurring within the component recognizer are not available after the negation succeeds.
- Opt : An optional recognizer succeeds and will consume input if the component recognizer matches the input. No value is synthesized.
- Or : A disjunction succeeds when either of the component recognizers succeed. Bindings established within the components are not available outside the disjunction. No value is synthesized. When a disjunction is encountered, the left recognizer is tried and the right recognizer is recorded as a choice point. If the parse subsequently fails then the parser backtracks to the most recently established choice point and tries the alternative.
- ParseError : Exception raised when an error occurs during parsing. An error occurs when the input cannot match the current recognizer and there are no alternative recognizers (arising from disjunctions) left to try.
- Plus : Succeeds when the component recognizer has been applied at least once to the input. Synthesizes a sequence of values produced by each application of the component.
- Predicate : Succeeds when the evaluation of the predicate returns true otherwise causes the parse to fail.
- Star : Succeeds when the component recognizer has been applied 0 or more times to the input. Synthesizes a sequence of values produced by each application of the component.
- Str : Succeeds when the next input token is a string; synthesizes the string.
- Terminal : Succeeds when the next input token is a named terminal string. No value is synthesized.
- Token : The parser uses a tokenizer to process the raw stream of input characters into a stream of input tokens.
- TypeCheck : The value synthesized by the component recognizer is checked against a type referenced as a path. If the value is of the specified type then the parse proceeds otherwise it fails.

The XBNF Grammar



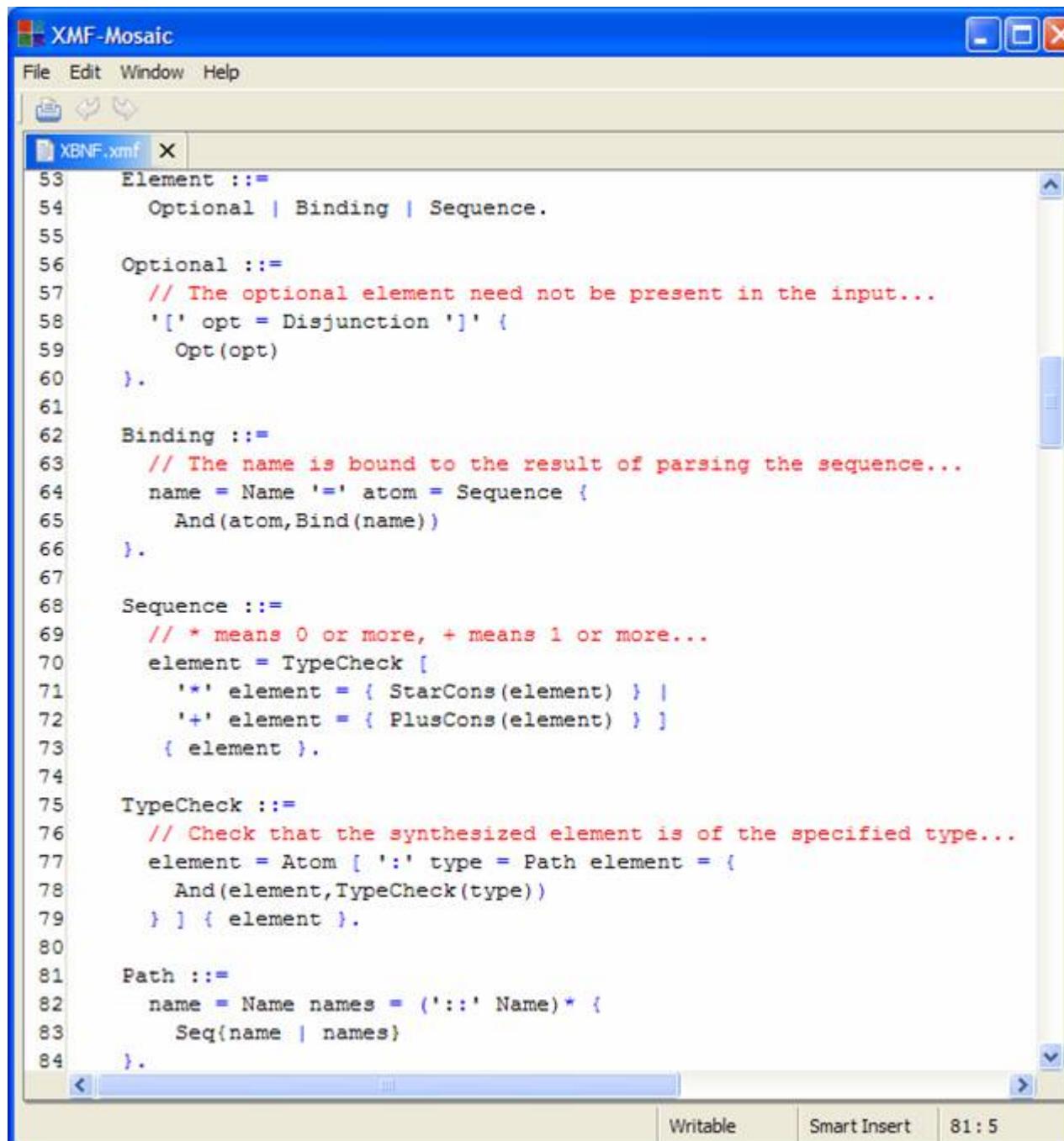
The screenshot shows the XMF-Mosaic editor interface with a blue header bar. The menu bar includes File, Edit, Window, and Help. Below the menu is a toolbar with icons for file operations. The main window displays a code editor with the file name XBNF.xmf. The code is an XBNF grammar definition:

```
6 context Grammar
7
8 @Grammar extends OCL::OCL.grammar
9
10 Grammar ::= 
11     // The parents define the inherited clauses...
12     parents = GrammarParents
13     // The imports extend the available names...
14     imports = GrammarImports
15     // The clauses define the parse rules...
16     clauses = Clause* {
17         Grammar(parents, clauses->asSet,"",imports)
18     }.
19
20 GrammarParents ::= 
21     // Parent expressions evaluate to produce grammars...
22     'extends' parent = Exp parents = (',' Exp)* {
23         parents->asSet->including(parent)
24     } | { Set{} }.
25
26 GrammarImports ::= 
27     // Import expressions evaluate to produce name spaces...
28     'import' class = Exp classes = (',' Exp)* {
29         Seq(class | classes)
30     } | { Seq{} }.
```

The status bar at the bottom right shows "Writable", "Smart Insert", and the time "27:63".

The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main window displays a code editor for the file "XBNF.xmf". The code defines three non-terminal symbols: Clause, Disjunction, and Conjunction. The Clause definition uses a Disjunction operator ('.') to combine a name and a body. The Disjunction definition uses an Or operator ('|') to combine elements. The Conjunction definition iterates over elements to produce an And expression. The code is annotated with comments explaining the binding priority and ordering rules for operators.

```
31 Clause ::=  
32     // Note binding priority for operators defined by NT ordering.  
33     // Use parentheses in clause body to override ordering...  
34     name = Name '::::' body = Disjunction '.' {  
35         Clause(name,body)  
36     }.  
37  
38  
39 Disjunction ::=  
40     // Use | to define alternative parses...  
41     element = Conjunction [  
42         '|' rest = Disjunction element = { Or(element,rest) }  
43     ]  
44     { element }.  
45  
46 Conjunction ::=  
47     // Element ordering dictates parse ordering...  
48     elements = Element+ {  
49         elements->tail->iterate(e conj = elements->head |  
50             And(conj,e))  
51     }.
```



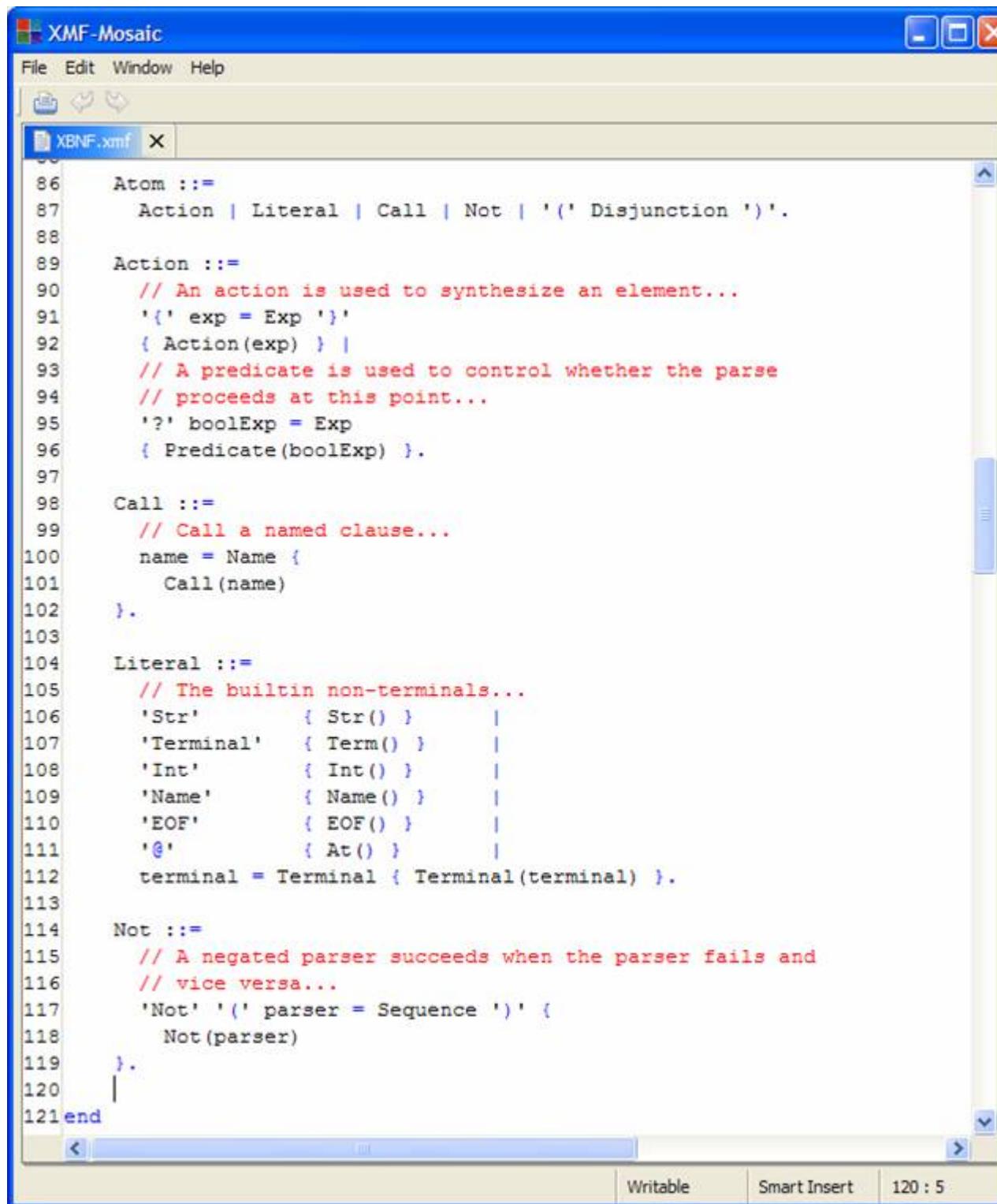
The screenshot shows the XMF-Mosaic editor interface with a blue header bar. The title bar displays "XMF-Mosaic". Below it is a menu bar with "File", "Edit", "Window", and "Help". A toolbar with icons for file operations is visible above the main editor area. The main window contains a code editor titled "XBNF.xmf". The code defines various grammar elements:

```

53 Element ::= 
54     Optional | Binding | Sequence.
55 
56 Optional ::= 
57     // The optional element need not be present in the input...
58     '[' opt = Disjunction ']' {
59         Opt(opt)
60     }.
61 
62 Binding ::= 
63     // The name is bound to the result of parsing the sequence...
64     name = Name '=' atom = Sequence {
65         And(atom,Bind(name))
66     }.
67 
68 Sequence ::= 
69     // * means 0 or more, + means 1 or more...
70     element = TypeCheck [
71         '*' element = { StarCons(element) } |
72         '+' element = { PlusCons(element) } ]
73     { element }.
74 
75 TypeCheck ::= 
76     // Check that the synthesized element is of the specified type...
77     element = Atom [ ':' type = Path element = {
78         And(element,TypeCheck(type))
79     } ] { element }.
80 
81 Path ::= 
82     name = Name names = ('::' Name)* {
83         Seq{name | names}
84     }.

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "81:5".



The screenshot shows the XMF-Mosaic IDE interface with the file 'XBNF.xmf' open. The code defines a grammar for XBNF (eXtensible BNF) with the following rules:

```

86 Atom ::= Action | Literal | Call | Not | '(' Disjunction ')'.
87
88 Action ::=
89   // An action is used to synthesize an element...
90   '{' exp = Exp '}'
91   { Action(exp) } |
92   // A predicate is used to control whether the parse
93   // proceeds at this point...
94   '?' boolExp = Exp
95   { Predicate(boolExp) }.
96
97 Call ::=
98   // Call a named clause...
99   name = Name {
100     Call(name)
101   }.
102
103 Literal ::=
104   // The builtin non-terminals...
105   'Str'      { Str() } |
106   'Terminal' { Term() } |
107   'Int'      { Int() } |
108   'Name'     { Name() } |
109   'EOF'      { EOF() } |
110   '@'        { At() } |
111
112   terminal = Terminal { Terminal(terminal) }.
113
114 Not ::=
115   // A negated parser succeeds when the parser fails and
116   // vice versa...
117   'Not' '(' parser = Sequence ')' {
118     Not(parser)
119   }.
120
121 end

```

The status bar at the bottom right indicates the file is 'Writable' and has 120 lines with 5 changes.

Tokens

XMF Execution Architecture

Introduction

Execution of XOCL programs or console commands conforms to a standard execution architecture. This architecture is extensible; new language constructs can be added dynamically. This section describes the basic features of the XMF execution architecture and how the extension mechanism works so that new languages can be embedded within XOCL.

The basic execution process parses an input source and synthesizes a performable element. XMF provides a large number of performable elements in XOCL and its various language extensions. You can easily define your own type of performable elements in terms of how they are parsed and executed.

Performable Elements

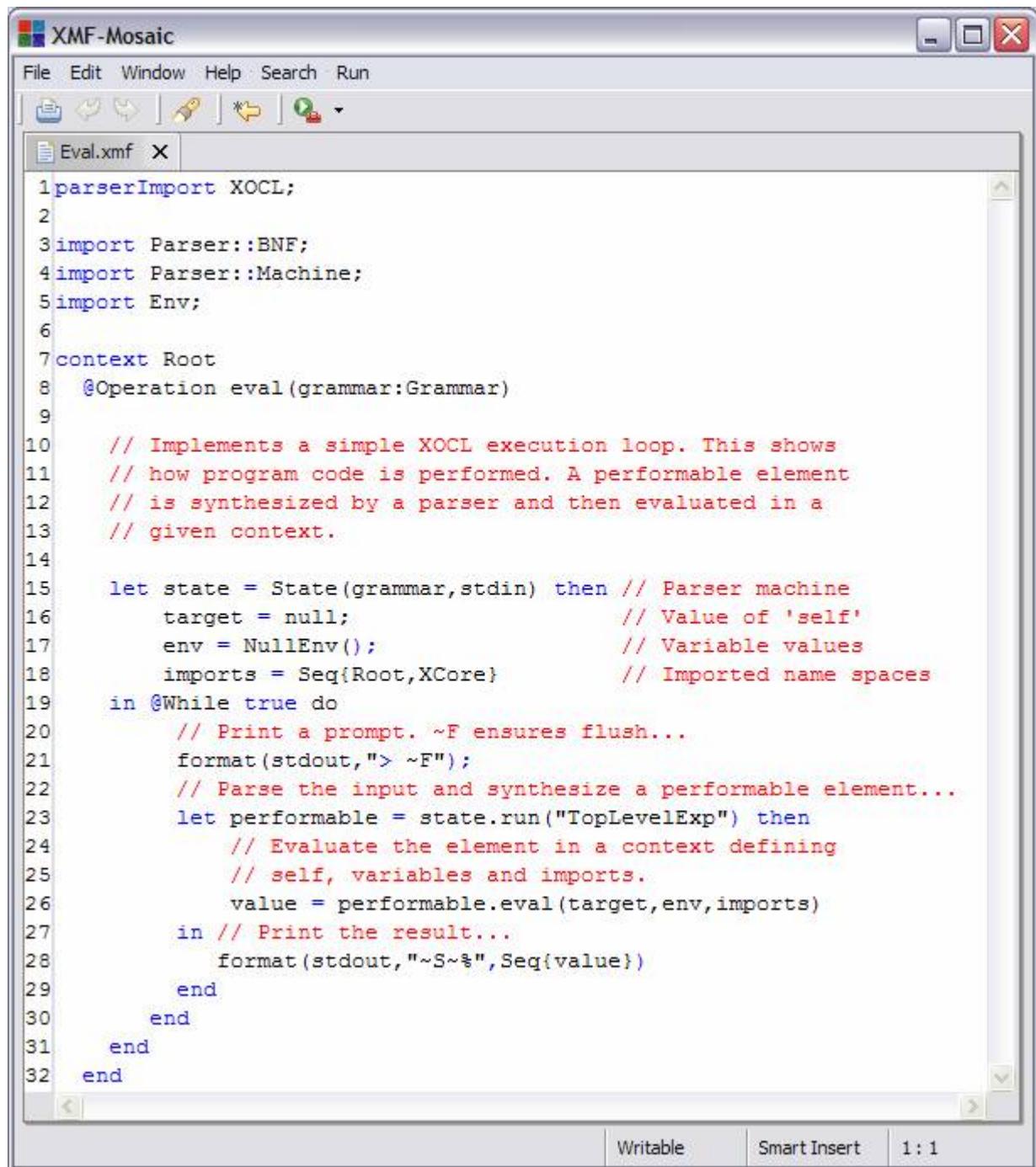
An XMF performable element implements an interface that supports its evaluation. Evaluation can occur in one of two ways:

- Interpretation. This occurs via the element's eval operation and invokes an interpreter that inspects the structure of the element and returns its value. The eval operation takes arguments that define the context of the interpretation. The context defines the value of self, the value of any variables and the name spaces that are imported at the point of evaluation.
- Compilation. This occurs via the element's compile operation and invokes a compiler that translates the receiver into a sequence of XMF-VM instructions. The compilation operation takes a number of arguments that define the compilation context.

In both cases, as far as the user of XMF is concerned, the key factor is that the target of evaluation is an instance of XCore::Performable. Therefore, any user interface that executes interpretation or compilation must translate from input to a performable element.

The majority of user interfaces to this process accept text as input. The text is parsed using a grammar and a performable instance is synthesized. Therefore, to hook into the evaluation process of XMF you need to understand how to write grammars that recognize new language constructs and synthesize performable elements. Fortunately, there are a number of XMF technologies that ease this activity.

The following example shows how a grammar that synthesizes performable elements is linked into the XMF evaluation architecture. The example is a simplified version of the XMF top level command interpreter:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for new, open, save, cut, copy, paste, and search. The main window shows a code editor with the file "Eval.xmf" open. The code is written in a domain-specific language (DSL) for XMF. It defines a parser import for XOCL, imports various parser components, and defines a context Root with an eval operation. The eval operation implements a simple execution loop that reads from stdin, runs a TopLevelExp, evaluates the result, and prints it to stdout. The code uses annotations like @Operation, @While, and @For, and variables like state, target, env, and imports.

```

1 parserImport XOCL;
2
3 import Parser::BNF;
4 import Parser::Machine;
5 import Env;
6
7 context Root
8   @Operation eval(grammar:Grammar)
9
10  // Implements a simple XOCL execution loop. This shows
11  // how program code is performed. A performable element
12  // is synthesized by a parser and then evaluated in a
13  // given context.
14
15  let state = State(grammar,stdin) then // Parser machine
16    target = null;                      // Value of 'self'
17    env = NullEnv();                   // Variable values
18    imports = Seq{Root,XCore}        // Imported name spaces
19  in @While true do
20    // Print a prompt. ~F ensures flush...
21    format(stdout,> ~F");
22    // Parse the input and synthesize a performable element...
23    let performable = state.run("TopLevelExp") then
24      // Evaluate the element in a context defining
25      // self, variables and imports.
26      value = performable.eval(target,env,imports)
27    in // Print the result...
28      format(stdout,"~S~%",Seq{value})
29    end
30  end
31 end
32 end

```

The eval operation accepts a grammar argument in line 8. The grammar should synthesize a performable instance that is subsequently evaluated in line 26. If OCL::OCL.grammar is supplied to eval then the operation behaves like the XMF-Mosaic top level command interpreter.

Compilation of performable elements occurs when you compile an XMF source file. The compiler reads the input source, synthesizes a performable element, compiles the element to a sequence of machine instructions and then writes the instructions to a binary file.

Instead of writing to a binary file, the instructions can be transformed to a compiled operation using the operation Compiler::compileToFun as shown below:

```

1 parserImport XOCL;
2
3 import Parser::BNF;
4 import Parser::Machine;
5 import Compiler;
6
7 context Root
8   @Operation compile(grammar:Grammar)
9
10  // Implements a simple compiler. A performable element
11  // is read from stdin and then compiled to produce
12  // an operation. The free variables referenced in the
13  // element are turned into arguments for the operation.
14
15  let state = State(grammar,stdin) then
16    performable = state.run("TopLevelExp") then
17      // Calculate the free variables...
18      FV = performable.FV() -> asSeq;
19      // Dynamics are the imported name spaces...
20      dynamics = Seq{}
21      in // Compiler::compileToFun takes a name, the performable
22      // element, the sequence of arg names, the dynamics and
23      // whether or not to save the source code. It returns an
24      // operation whose body performs the performable element.
25      compileToFun("",performable,FV,dynamics,true)
26    end
27  end

```

The free variables of the element are calculated in line 18 and used as the argument names for the operation constructed in line 25.

The key feature of operations eval and compile is that the XML evaluation architecture reads sources of XMF program code, parses the input using a grammar, synthesizes an instance of XCore::Performable and then interprets or compiles the element.

XOCL allows new language constructs to be added as described in the following section. The new constructs are defined as grammars that are integrated into the XOCL grammar. Language extensions that are added in this way should therefore synthesize performable elements.

Syntax Extensions

Synthesising Syntax

Introduction

Language definitions in XMF-Mosaic are defined as grammars that synthesize performable elements. The package OCL defines a complete language of performable classes. A convenient way of defining a new language feature is to define a grammar that synthesizes instances of OCL classes. This section describes the classes in the OCL package and how to use them to define new language features.

The OCL Package

The OCL package defines a language whose elements are extensions of XCore::Performable. The language has a concrete syntax that is defined by the grammar OCL::OCL.grammar. This grammar synthesizes instances of the OCL classes. All of XMF-Mosaic is written in this language.

Each class in the OCL package has one or more constructors. These constructors can be used to synthesize performable elements when defining new language constructs. This section defines the OCL class constructors.

The table below lists all of the main OCL class constructors. The constructors are defined in the first column and the second column describes what the class implements and the types of the arguments. When specifying the types of the arguments we refer to performable elements as ‘exp’.

Table 17.1.

Addp(left,right)	Creates an add pattern. An example is the argument pattern in the following operation: @Operation(l + r) l end.
Apply(operator,args)	The operator expression is applied to the sequence of argument expressions, for example f(1,2,3).
BinExp(left,binOp,right)	A binary expression. The binary operator is a string. An example is: x + 1.
BoolExp(value)	A boolean value. The value argument should be either true or false.
CollExp(collection,collOp,args)	A collection expression. The collOp is a string naming a standard collection expression. For example S->including(x).
Condp(pattern,condition)	A condition pattern. The condition is a boolean values expression. For example the following pattern in a case statement: @Case c of Class(name) when name <> “C” do name end end
ConsExp(head,tail)	A pair constructing expression. For example Seq{1 s}.
Consp(head,tail)	A pair pattern. For example in the following operation argument pattern: @Operation(Seq{h t}) h end.
Constp(const)	A constant pattern. The value of the constant should be an integer, string, boolean or float. For example in the following case statement: @Case x of 10 do “10” end end
ContextDef(path,element)	A context definition as occurring at the top level of a source file. Both the path and the element are expressions. It is equivalent to path.add(element).
ContextDef(path,element,isForward)	As for ContextDef except that the isForward boolean argument controls whether or not the element is initialised after it is added to the container. This is useful when loading multiple

	definitions from files where the definition contain mutual dependencies.
Dot(target,name)	A slot reference. The target is an expression and the name is a string.
FloatExp(prePoint,postPoint)	A slot expression. The pre and post points are string representations of the numbers before and after the decimal point.
HeadUpdate(seq,value)	Update the head of a sequence as in S->head := e.
If(test,conseq,alt)	An if expression. Each of the arguments are expressions.
ImportIn(nameSpace,body)	A local import. Both arguments are expressions. For example: import namespace in MyClass(YourClass(1,2,3)) end
Includingp(set,element)	An including pattern. Both the set and element are patterns. For example the element selection pattern occurring as the argument in the following operation: @Operation(S->including(x)) x end
Instantiate(class,args)	A keyword instantiation expression. The class is an expression and the arguments are key args. For example: Class[name="C"].
IntExp(value)	An integer constant expression. The value is an integer.
IterExp(collection,iterOp,name,body)	An iteration expression where the collection and body are expressions and the iterOp and name are strings. The iterOp should be one of the strings: "select", "collect", "reject". For example: S->collect(x x + 1).
Iterate(collection,name,accumulator,value,body)	An iterate expression where the collection, value and body are all expressions and the name and accumulator are strings. For example: S->iterate(x y = 100 x + y)
KeyArg(name,value)	A key arg is a value used in an Instantiate expression.
Keyp(name,pattern)	A keyword pattern occurring in an instance of the Keywordp pattern. The name is a string.
Keywordp(class,names,keys)	A keyword constructor pattern. The class is a string and the names are a sequence of strings. To represent the class P::Q::C the class arg is "P" and the names are Seq{"Q","C"}. The keys are a sequence of Keyp instances.
Let(bindings,body)	A let expression. The bindings argument is a sequence of instances of ValueBinding and the body is an expression. For example: let x = 10; y = 20 in x + y end
NamedType()	A type expression. With no arguments the type represents XCore::Element.
NamedType(path)	A type expression where the argument is a sequence of strings determining the path (relative to current imports) of the type.
Negate(exp)	A not expression.

Objectp(class,names,slots)	A positional constructor pattern. The class is a string and the names are a sequence of strings. To represent the class P::Q::C the class arg is “P” and the names are Seq{“Q”,“C”}. The slots is a sequence of patterns. For example the argument pattern in the following operation: @Operation(C(x,10)) x end
OpType(domains,range)	The type of an operation. The domains are a sequence of type expressions and the range is a type expression. For example the type of the argument in the following operation: @Operation(f: (Integer,Integer)->Integer):Integer f(1,2) end
Operation(name,parameters,type)	An operation expression. The name is a string, the parameters are a sequence of patterns, the type is a type expression.
Operation(name,parameters,type,performable)	As above where the performable argument is an expression which is the body of the operation.
Operation(name,parameters,type,performable,documentation)	As above where the documentation is a string.
Operation(name,parameters,type,performable,documentation,withMultiArgs)	As above with MultiArgs is a boolean that determines whether the operation can accept a variable number of arguments.
Order(first,second)	A sequenced expression where the arguments are expressions. For example: x.m(1); y.n(2)
ParametricType(constructor,args)	A type expression of the form C(T).
Parentheses(exp)	Parentheses in source code are recorded in the abstract syntax using an instance of Parentheses. For evaluation purposes this behaves exactly like the expression argument.
ParserImport(names,exp)	Not supported.
Path(root,names)	A path expression consists of an expression root and a sequence of names. For example Root::XCore::Class.
PathUpdate(path,value)	The update of a name in a name space. The path is a an instance of the class Path and the value is an expression. For example: P::Q::X := 100
Self()	A reference to the current value of self.
Send(target,message,args)	Send a message to an element. The target is an expression. The message is a string and the args is a sequence of expressions. For example: o.m(a,b,c)
SetExp(collType,elements)	A set expression constructs sets and sequences. The collType argument is either “Set” or “Seq” and the elements is a sequence of expressions. For example Set{1,2,3} or Seq{1,2,3}.
SlotUpdate(target,name,value)	Update the slot of an object. The target and value are expressions and the name is a string. For example: o.n := 100
StrExp(value)	A string constant. The value argument is a string.
Syntaxp(exp)	A syntax pattern.
TailUpdate(seq,value)	Update the tail of a sequence as in S->tail := e.

Throw(exp)	Throw the value of the expression as in: throw Error("no handler")
Try(body,name,handler)	A try expression. The body and handler are expressions and the name is a string. For example: try body catch(e) handler end
ValueBinding(name,value)	A value binding is used in let expressions to bind names to values. The name is a string and the value is an expression.
Var(name)	A variable reference. The name is a string.
Var(name,lineCount,charCount)	A variable reference where the occurrence of the variable in source code has been recorded. The line and char count are both integers.
VarUpdate(name,value)	Update the value of a local variable. The name is a string and the value is an expression. For example: x := 10
Varp(name)	A variable pattern. The name is a string. Arguments to operations are patterns. An argument that is just a name is represented as a variable pattern.
Varp(name,pattern,type)	A variable pattern which binds a value to name providing it matches pattern. The type is a type expression recording the declared type of the variable. For example: @Operation(x = C(10) : P::C) x.m(100) end
Varp(name,type)	A variable pattern declaring just the name and the type.
XOCL::While(test,body)	A while loop. Defined in the package XOCL. The test argument is a boolean values expression and the body is an expression. For example: @While x > 10 do y := y + x; x := x - 1 end

Examples

Suppose that we want to write operations together with their specifications. The specification of an operation consists of a pre-condition and a post-condition. A pre-condition states what must be true on entry to the operation and a post-condition states what must be true on exit from the operation. The context for the conditions includes the arguments of the operation.

A specification serves two important purposes: firstly we can specify an operation without knowing how the operation is to be implemented. The pre-condition and post-condition are used to restrict the possible legal implementations.

Secondly, the conditions can be performed when the operation is called. The conditions provide a form of dynamic run-time checking.

This section shows how a simple specification construct can be defined. A grammar for the new construct is defined that synthesizes an operation definition.

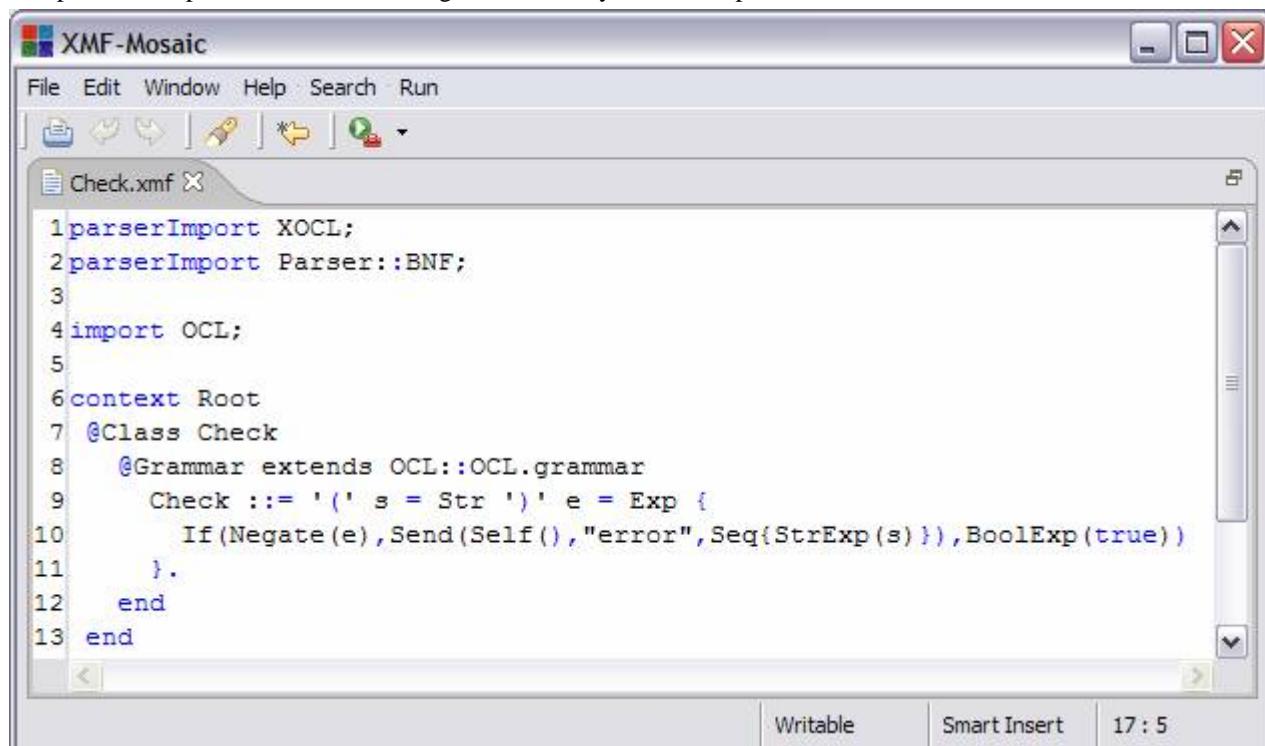
A specification will be defined as follows:

```
@Spec incx(dx:Integer)
  pre dx > 0
    self.x := x + dx
  post x = preSelf.x + dx
end
```

The specification above requires that the value of dx is greater than 0 before it is used to increase the value of the slot x. The post condition specifies that the state of the x slot must have changed appropriately. The state of the receiver is accessible in the post-condition via the variable preSelf. The class definition for the Spec construct is shown below:

Checking Conditions

Suppose that we wish to implement a new language construct that checks a condition and raises an error if the condition does not hold. Essentially this is just an if-expression used in a particular pattern. The pattern is captured as a class with a grammar that synthesizes a performable instance:



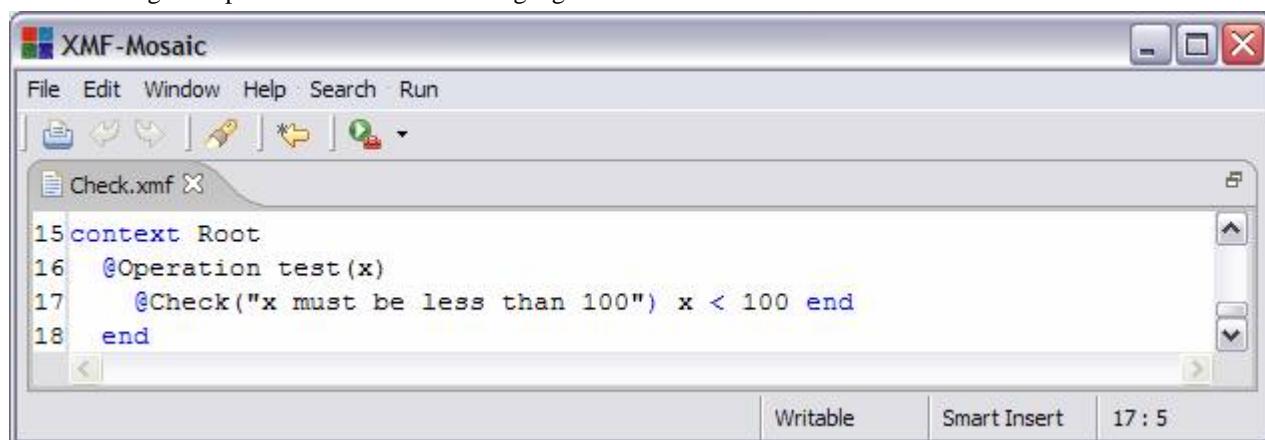
The screenshot shows the XMF-Mosaic IDE interface with a file named "Check.xmf" open. The code defines a grammar for a "Check" class that extends OCL's OCL.grammar. It includes a rule for "Check" which takes a string "s" and an expression "e", and performs an action based on whether "e" is negated or not. The code is as follows:

```

1 parserImport XOCL;
2 parserImport Parser::BNF;
3
4 import OCL;
5
6 context Root
7 @Class Check
8   @Grammar extends OCL::OCL.grammar
9     Check ::= '(' s = Str ')' e = Exp {
10       If(Negate(e), Send(Self(), "error", Seq(StrExp(s))), BoolExp(true))
11     }.
12   end
13 end

```

The following example shows how the new language construct is used:



The screenshot shows the XMF-Mosaic IDE interface with the same "Check.xmf" file open, but with additional code added. This new code defines an operation "test(x)" with a precondition that "x" must be less than 100. The code is as follows:

```

15 context Root
16   @Operation test(x)
17     @Check("x must be less than 100") x < 100 end
18   end

```

Specification

Suppose that we want to write operations together with their specifications. The specification of an operation consists of a pre-condition and a post-condition. A pre-condition states what must be true on entry to the operation and a post-condition states what must be true on exit from the operation. The context for the conditions includes the arguments of the operation.

A specification serves two important purposes: firstly we can specify an operation without knowing how the operation is to be implemented. The pre-condition and post-condition are used to restrict the possible legal implementations.

Secondly, the conditions can be performed when the operation is called. The conditions provide a form of dynamic run-time checking.

This section shows how a simple specification construct can be defined. A grammar for the new construct is defined that synthesizes an operation definition.

A specification will be defined as follows:

```
@Spec incx(dx:Integer)
  pre dx > 0
    self.x := x + dx
  post x = preSelf.x + dx
end
```

The specification above requires that the value of `dx` is greater than 0 before it is used to increase the value of the slot `x`. The post condition specifies that the state of the `x` slot must have changed appropriately. The state of the receiver is accessible in the post-condition via the variable `preSelf`.

The class definition for the `Spec` construct is shown below:

XMF-Mosaic

File Edit Window Help

Spec.xmf

```

1 parserImport Xocl;
2 parserImport Parser::BNF;
3
4 import OCL;
5
6 context Root
7 @Class Spec
8     // Implements a simple specification as an operation
9     // with a pre and post condition. The conditions are
10    // checked on entry and exit to the operation.
11    @Grammar extends OCL::OCL.grammar
12    // Careful when extending an existing grammar not to define
13    // clauses with overlapping names. For example the OCL grammar
14    // defines a clause with the name Args- use SpecArgs here.
15 Spec ::= n = Name a = SpecArgs pre = Pre body = Exp post = Post {
16
17     // Change:
18     // @Spec n(args)
19     //   pre p1
20     //   e
21     //   post p2
22     // end
23     // to
24     // @Operation n(args)
25     //   if p1
26     //   then
27     //     let preSelf = self.copy()
28     //     in let result = e
29     //       in if p2
30     //         then result
31     //         else self.error("post fails"
32     //           end
33     //         end
34     //       end
35     //     else self.error("pre fails")
36     //   end
37     // end

```

Writable Smart Insert 59 : 13

```

39     Operation(n,a,NamedType()),
40         If(pre,
41             Let(Seq{ValueBinding("preSelf",Send(Self(),"copy",Seq{}))),,
42                 Let(Seq{ValueBinding("result",body)},
43                     If(post,
44                         Var("result"),
45                         Send(Self(),"error",Seq{StrExp("post fails")})}}},
46                     Send(Self(),"error",Seq{StrExp("pre fails")})})
47     }.
48     SpecArgs ::= NoArgs | SomeArgs.
49     NoArgs ::= ' ' { Seq{} } .
50     SomeArgs ::= '(' n = SpecArg ns = (, ' SpecArg)* ' ) {
51         Seq{n | ns}
52     }.
53     SpecArg ::= n = Name { Varp(n) } .
54     Pre ::= 'pre' Exp.
55     Post ::= 'post' Exp.
56 end
57 end

```

```

59 context Root
60 @Class Test
61     @Attribute x : Integer end
62     @Spec incx(dx)
63         pre dx > 0
64             self.x := x + dx
65         post x = preSelf.x + dx
66     end
67 end

```

A For Loop

```

1 parserImport Xocl;
2 parserImport Parser::BNF;
3
4 import OCL;
5 import Xocl;
6
7 context Root
8 @Class For
9     @Grammar extends OCL::OCL.grammar
10    For ::= n = Name 'in' c = Exp 'do' b = Exp {
11        // let c = c->asSeq
12        // in @While not c->isEmpty do
13        //     let n = c->head
14        //     in b;
15        //     c := c->tail
16        //     end
17        // end
18        // end
19        Let(
20            Seq{ValueBinding("c", CollExp(c, "asSeq", Seq{}))},
21            While(
22                Negate(CollExp(Var("c"), "isEmpty", Seq{})),
23                Order(
24                    Let(
25                        Seq{ValueBinding(n, CollExp(Var("c"), "head", Seq{})),
26                            b),
27                        VarUpdate("c", CollExp(Var("c"), "tail", Seq{}))))))
28            .
29        end
30    end|

```

Writable Smart Insert 30 : 6

Quasi-Quotes

Introduction

When designing languages and language constructs it is usual to construct grammars that synthesize syntax elements. As we have seen in earlier sections, grammars can perform any commands in their actions; in particular they can construct and return instances of classes. If the classes inherit from XCore::Performable then the grammar synthesizes syntax.

One of the characterising features of grammars that synthesize syntax is that they map from relatively simple constructs to relatively complex syntax trees (after all this is the key reason for defining new language features – they are simpler to use than the corresponding expanded form).

Syntax comes in two key formats: abstract syntax and concrete syntax. Abstract syntax is oriented towards machines and is essentially the data structure that the machine represents when supplied with syntax. Concrete syntax is oriented towards humans and is character based. For example, the following is concrete syntax:

```
x + 1
```

and equivalently in abstract syntax:

```
BinExp(Var("x"), "+", IntExp(1))
```

Working with abstract syntax in the actions of grammars that synthesize syntax can be cumbersome. The size of the instantiation expressions can become large and you must remember all of the arguments to the constructors. Fortunately, XMF provides a key technology that allows you to work with concrete syntax when constructing instances of performable classes. This technology is called quasi-quotes.

Quasi-quotes allow you to write concrete syntax that constructs a performable element. In addition, quasi-quote expressions can be used to construct syntax templates by leaving holes where syntax can be dropped in to a surrounding expression. Quasi-quotes with holes can be used to capture syntax patterns.

This section describes how quasi-quotes work and how to construct syntax templates.

Literal Syntax

Syntax is constructed in XMF by instantiating a sub-class of XCore::Performable. A performable class is one that implements the API for evaluation, in terms of eval, compile and operations that they need to do their tasks. You will rarely need to implement the evaluation operations yourself since you can implement virtually all new constructs, however complex, in terms of new syntax constructs that expand into configurations of existing syntax classes.

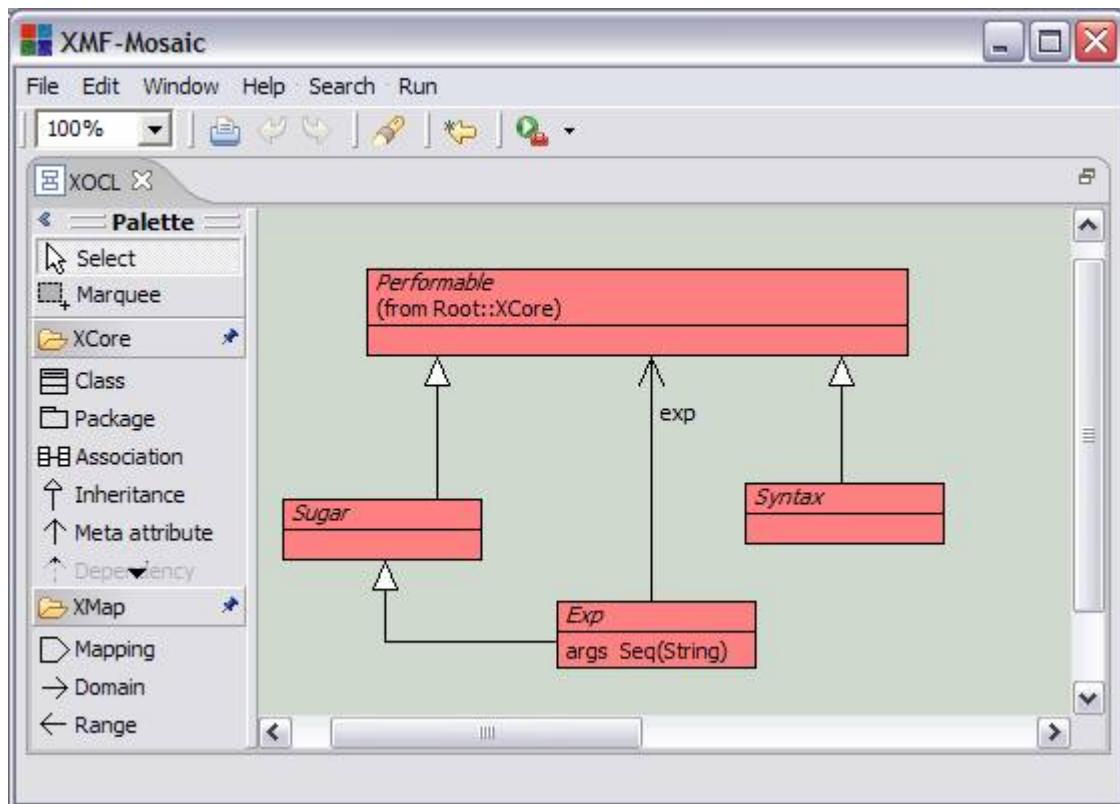
Syntax Templates

Splicing

Patterns

New Performable Elements

Introduction



Sugar

Introduction

It is usually the case that new syntax constructs can be implemented by translating to existing syntax constructs. The translation is often referred to as desugaring. It is convenient to create an instance of a new syntax class and then to desugar when the construct when XMF performs an evaluation or compilation request on the instance. By leaving the desugaring step to the last possible moment, the structure of the original syntax construct is retained for as long as possible.

Grammars that synthesize instances of new syntax classes can then use operations defined on the classes to implement the desugaring step. The classes can be as complex or as simple as required. In the limit, a sugared syntax class can implement a complete compiler for a complex new syntax construct, where the compiler produces performable elements.

XMF provides a sub-class of XCore::Performable called XOCL::Sugar that implements evaluation and compilation in terms of a call to an operation desugar. The desugaring operation performs a mapping from the receiver to a new performable object that already implements the evaluation or compilation. This section describes how to use XOCL::Sugar.

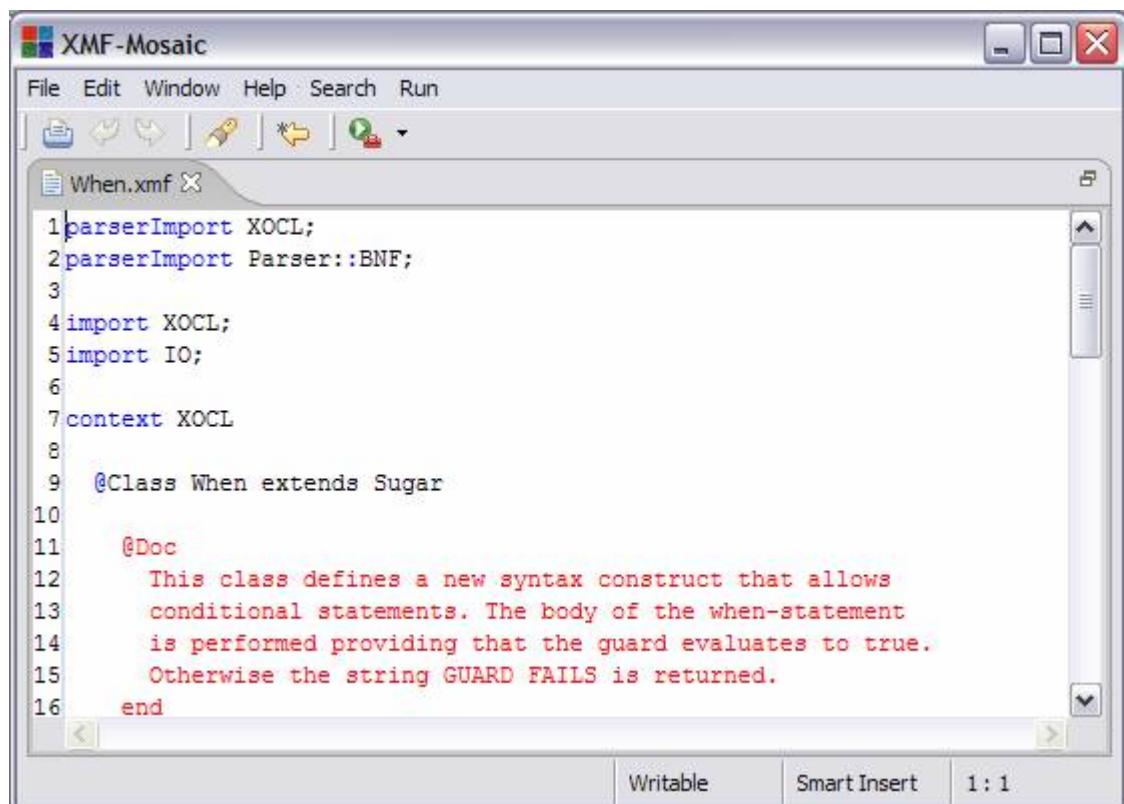
Guarded Statements

Consider a new language feature that implements a guarded statement. A guarded statement is to be implemented as an if expression with no else clause. The following is an example of its use:

```

@When passengers > 63 do
    format(stdout, "Too many people on the bus.%")
end
  
```

The conditional statement is implemented as a sugared syntax class whose desugar operation produces the equivalent if expression:

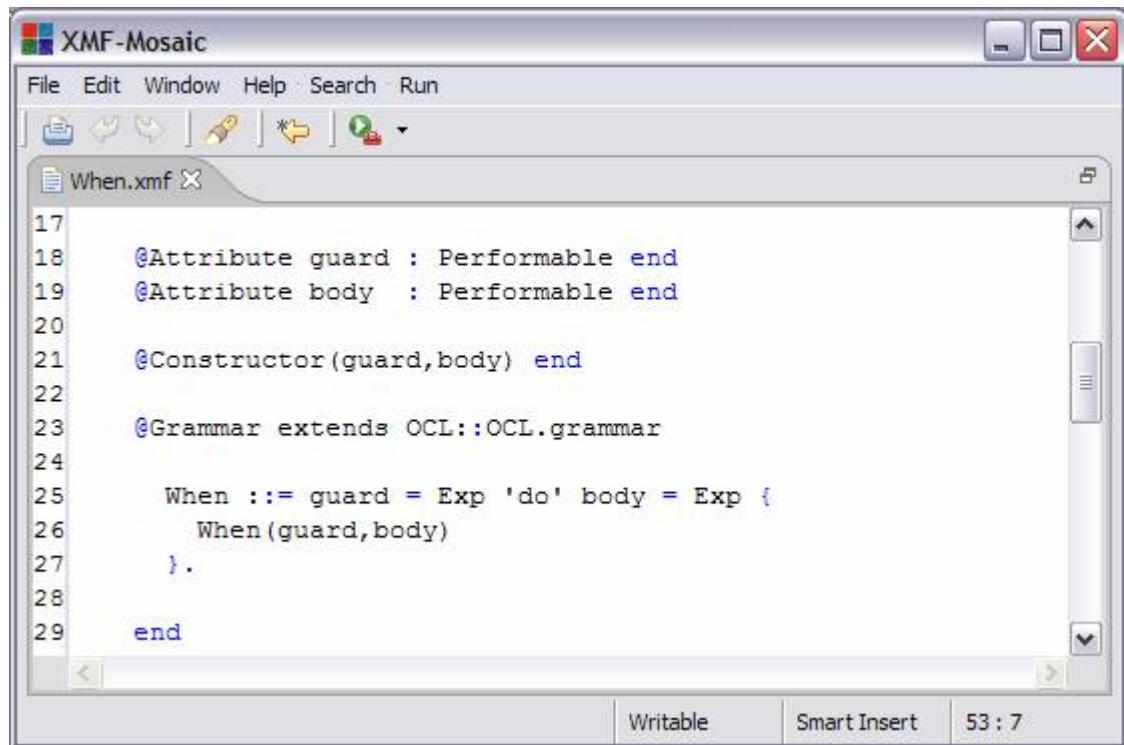


```

1 parserImport XOCL;
2 parserImport Parser::BNF;
3
4 import XOCL;
5 import IO;
6
7 context XOCL
8
9 @Class When extends Sugar
10
11 @Doc
12   This class defines a new syntax construct that allows
13   conditional statements. The body of the when-statement
14   is performed providing that the guard evaluates to true.
15   Otherwise the string GUARD FAILS is returned.
16 end

```

The grammar for a when-statement synthesizes an instance of the When class:

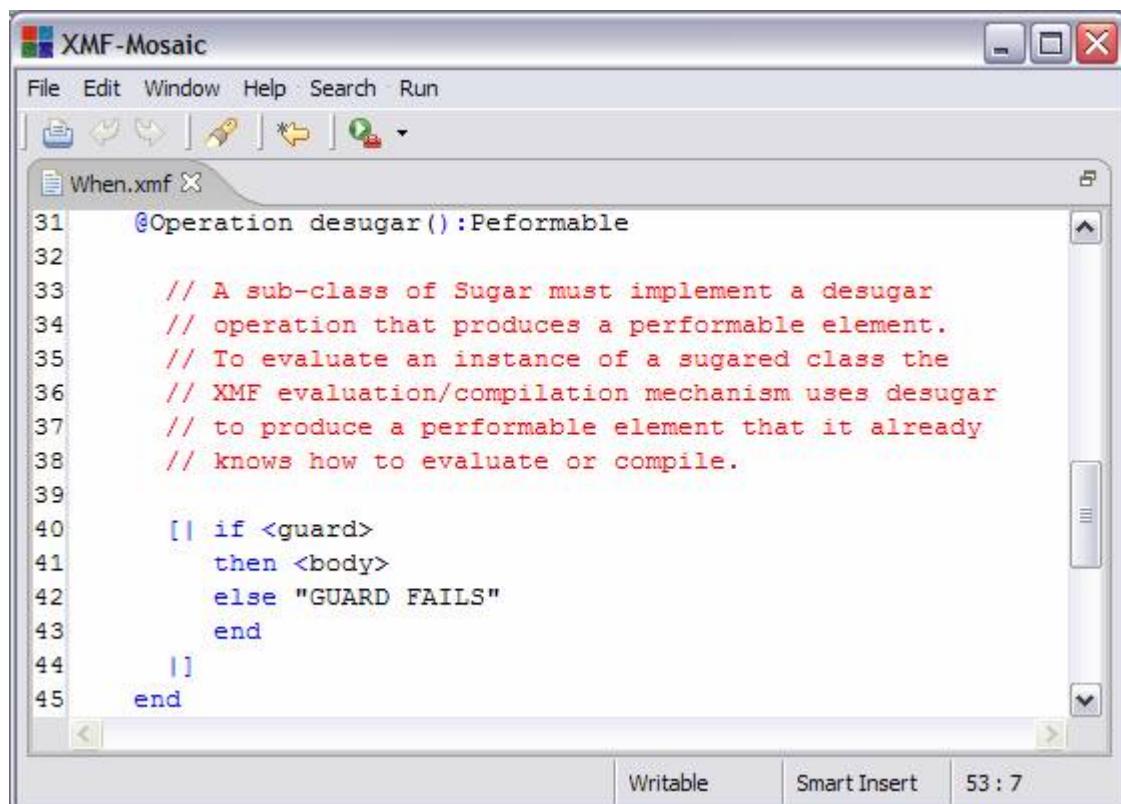


```

17
18   @Attribute guard : Performable end
19   @Attribute body   : Performable end
20
21   @Constructor(guard,body) end
22
23   @Grammar extends OCL::OCL.grammar
24
25   When ::= guard = Exp 'do' body = Exp {
26     When(guard,body)
27   }.
28
29 end

```

The desugar operation produces an instance of performable for a lower-level construct. In this case we produce an if expression and lose the fact that the then-part was guarded and the else-part was inserted.



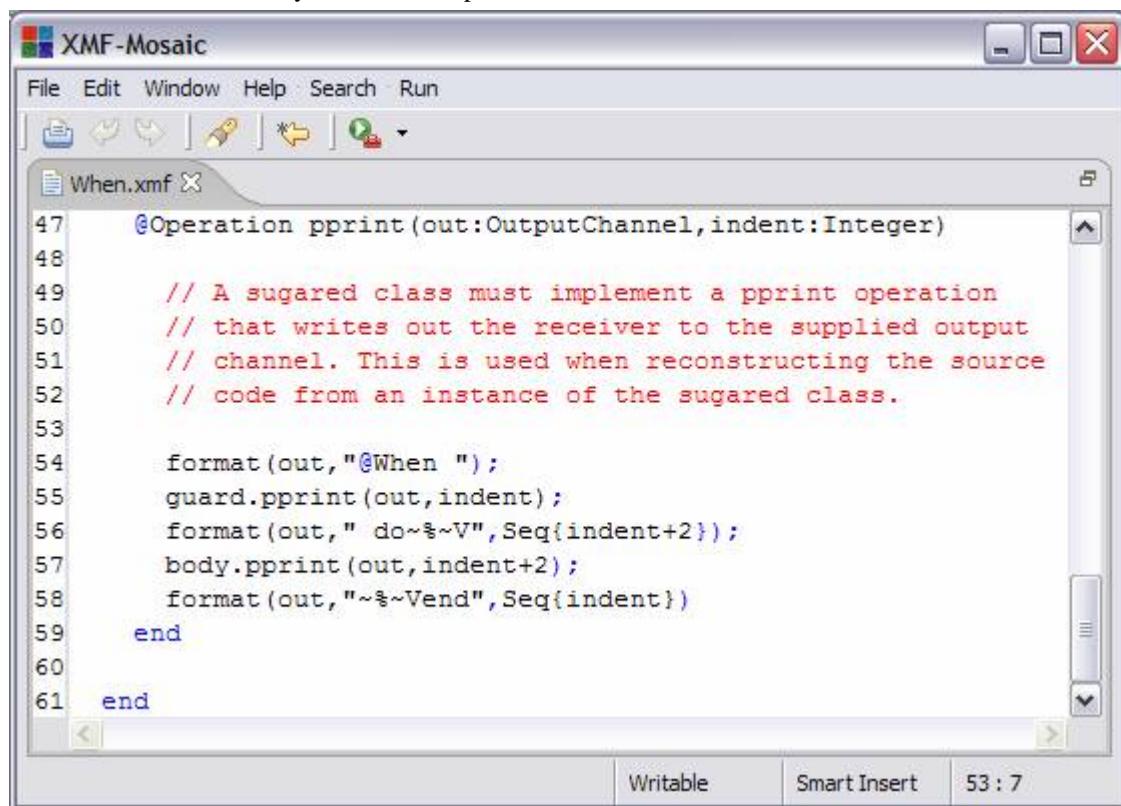
The screenshot shows the XMF-Mosaic IDE interface with the file 'When.xmf' open. The code defines an operation 'desugar()' for a class 'Performable'. It includes comments explaining the purpose of the operation and its implementation. The code uses standard programming constructs like if-then-else and end statements.

```

31     @Operation desugar():Performable
32
33     // A sub-class of Sugar must implement a desugar
34     // operation that produces a performable element.
35     // To evaluate an instance of a sugared class the
36     // XMF evaluation/compilation mechanism uses desugar
37     // to produce a performable element that it already
38     // knows how to evaluate or compile.
39
40     [| if <guard>
41       then <body>
42       else "GUARD FAILS"
43       end
44     |]
45   end

```

By providing a pprint operation in the syntax class for When we will see the original statement in any source code that is saved by the XMF compiler:



The screenshot shows the XMF-Mosaic IDE interface with the file 'When.xmf' open. The code now includes a pprint operation that formats the receiver into an output channel. The operation uses various XMF-specific functions like format and Seq to construct the output string.

```

47     @Operation pprint(out:OutputChannel, indent:Integer)
48
49     // A sugared class must implement a pprint operation
50     // that writes out the receiver to the supplied output
51     // channel. This is used when reconstructing the source
52     // code from an instance of the sugared class.
53
54     format(out, "@When ");
55     guard pprint(out, indent);
56     format(out, " do~%~V", Seq(indent+2));
57     body pprint(out, indent+2);
58     format(out, "~%~Vend", Seq(indent))
59   end
60
61 end

```

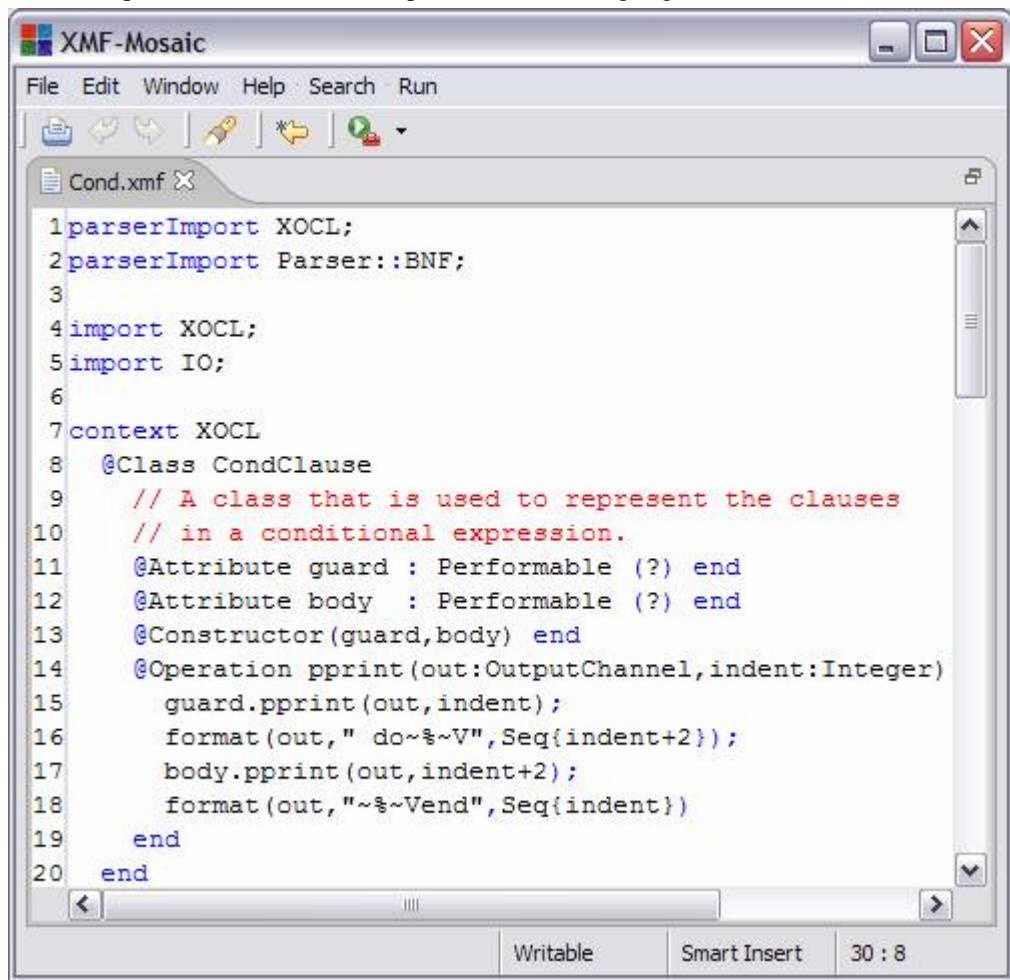
Conditional Expressions

Suppose we are developing a system that involves testing a large number of conditions. The conditions can be tested using nested if-expressions, however this gets difficult to read as the number of nested expressions increases. Suppose that XMF did not provide an elseif keyword as part of an if-expression and that, therefore, there is a requirement for a tabular conditional expression of the following form:

```
@Cond
  self.inRange(x,0,m) do
    self.tooLow(x)
  end
  self.inRange(x,m,n) do
    self.justRight(x)
  end
  self.inRange(x,n,infinity) do
    self.tooHigh(x)
  end
end
```

where each clause is tried in turn until a test evaluates to true whereupon the corresponding action is performed.

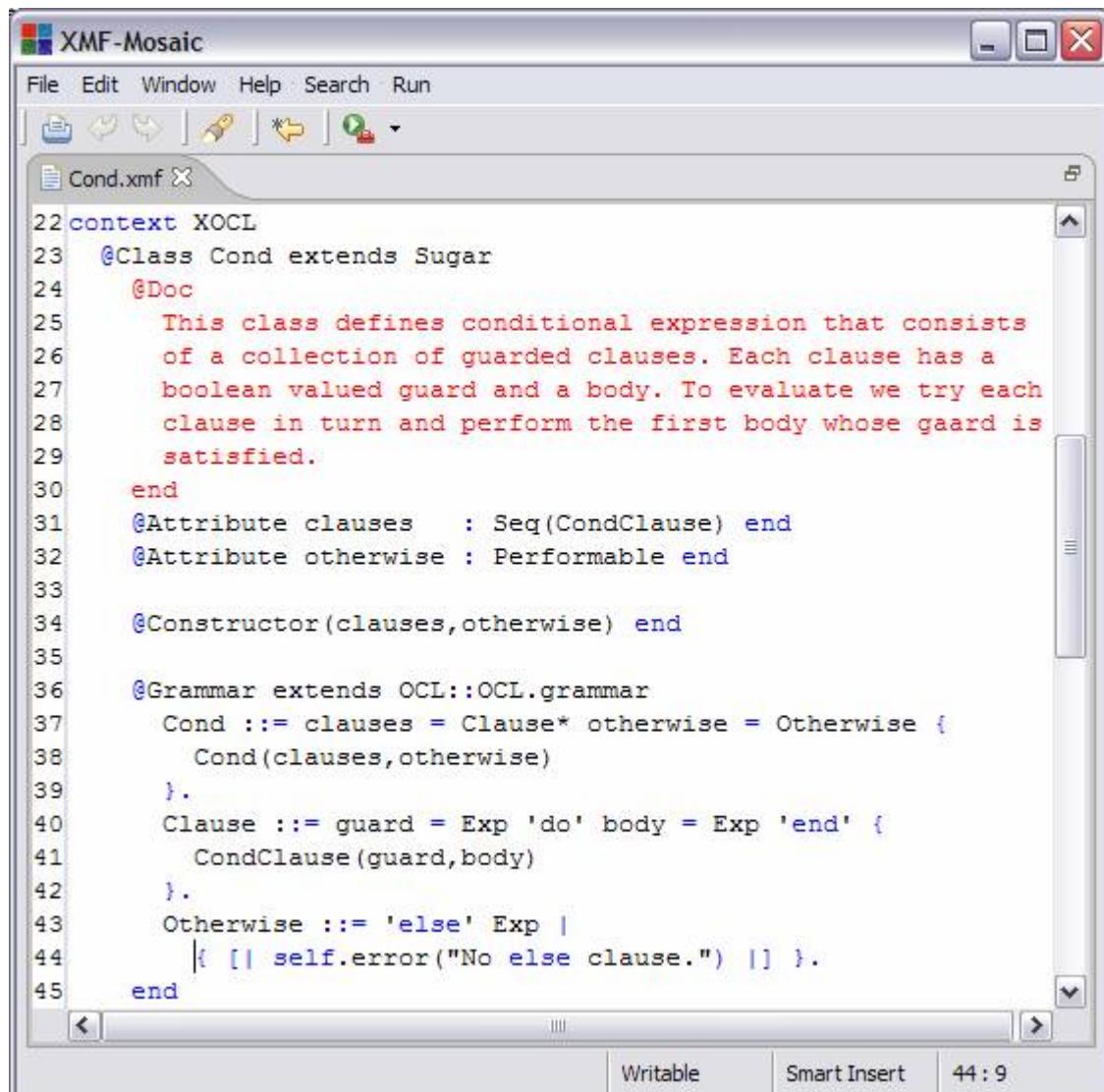
We can implement the conditional expression above using sugar as follows:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations like Open, Save, and Run. The main window displays a code editor with the file "Cond.xmf" open. The code is as follows:

```
1 parserImport XOCL;
2 parserImport Parser::BNF;
3
4 import XOCL;
5 import IO;
6
7 context XOCL
8   @Class CondClause
9     // A class that is used to represent the clauses
10    // in a conditional expression.
11    @Attribute guard : Performable (?) end
12    @Attribute body   : Performable (?) end
13    @Constructor(guard,body) end
14    @Operation pprint(out:OutputChannel,indent:Integer)
15      guard pprint(out,indent);
16      format(out," do~%~V",Seq(indent+2));
17      body pprint(out,indent+2);
18      format(out,"~%~Vend",Seq(indent))
19    end
20  end
```

The code editor has scroll bars on the right and bottom. At the bottom, there are buttons for Writable, SmartInsert, and a status bar showing "30 : 8".



The screenshot shows the XMF-Mosaic editor interface with a file named 'Cond.xmf' open. The code is written in OCL (Object Constraint Language) and defines a class 'Cond' that extends 'Sugar'. The class has attributes 'clauses' (a sequence of 'CondClause') and 'otherwise' (a 'Performable'). It also has a constructor that takes 'clauses' and 'otherwise'. The grammar section defines 'Cond' as a sequence of clauses followed by an 'otherwise' clause, which is further defined as a 'CondClause'. The 'Otherwise' clause is defined as an 'else' expression that either succeeds or fails with an error message. The code is annotated with documentation comments (@Doc) explaining the purpose of each part.

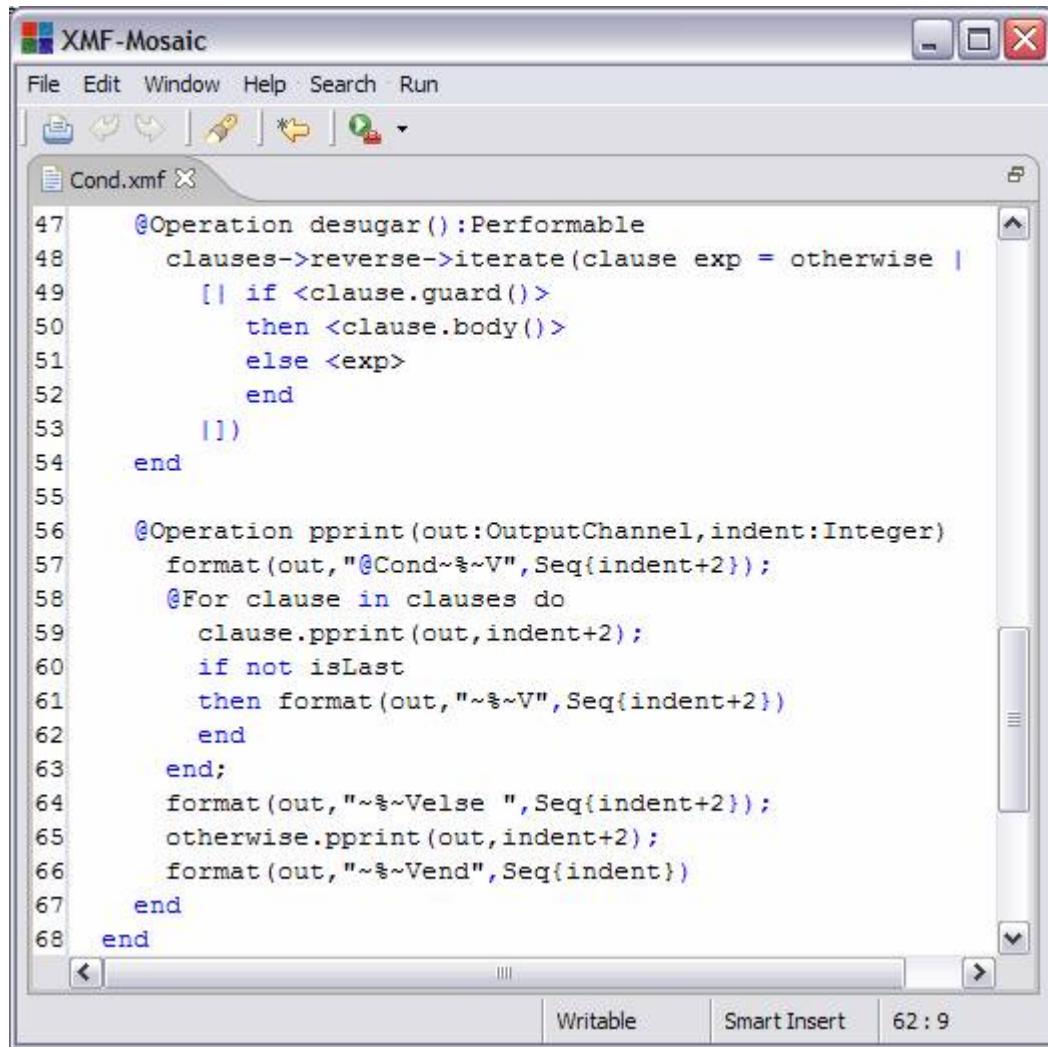
```

22 context XOCL
23 @Class Cond extends Sugar
24 @Doc
25     This class defines conditional expression that consists
26     of a collection of guarded clauses. Each clause has a
27     boolean valued guard and a body. To evaluate we try each
28     clause in turn and perform the first body whose gaard is
29     satisfied.
30 end
31 @Attribute clauses : Seq(CondClause) end
32 @Attribute otherwise : Performable end
33
34 @Constructor(clauses,otherwise) end
35
36 @Grammar extends OCL::OCL.grammar
37 Cond ::= clauses = Clause* otherwise = Otherwise {
38     Cond(clauses,otherwise)
39 }.
40 Clause ::= guard = Exp 'do' body = Exp 'end' {
41     CondClause(guard,body)
42 }.
43 Otherwise ::= 'else' Exp |
44     [|| self.error("No else clause.") || ] .
45 end

```

Notice that the cond-expression synthesizes instances of CondClause in line 41 as part of the overall synthesis of a Cond. This is typical of a sugared construct which needs to represent internal structure prior to the desugaring process. The instances of CondClause are transient and exist only long enough to process the containing Cond. In general, many such transient objects may be created when processing sophisticated sugared constructs.

A cond-expression is desugared into a nested if-expression:



```

47     @Operation desugar():Performable
48         clauses->reverse->iterate(clause exp = otherwise |
49             [| if <clause.guard()>
50                 then <clause.body()>
51                 else <exp>
52                     end
53             []])
54     end
55
56     @Operation pprint(out:OutputChannel, indent:Integer)
57         format(out, "@Cond~%~V", Seq{indent+2});
58         @For clause in clauses do
59             clause pprint(out, indent+2);
60             if not isLast
61                 then format(out, "~%~V", Seq{indent+2})
62                 end
63             end;
64             format(out, "~%~Velse ", Seq{indent+2});
65             otherwise pprint(out, indent+2);
66             format(out, "~%~Vend", Seq{indent})
67         end
68     end

```

Syntax

Exp

Introduction

State Machines

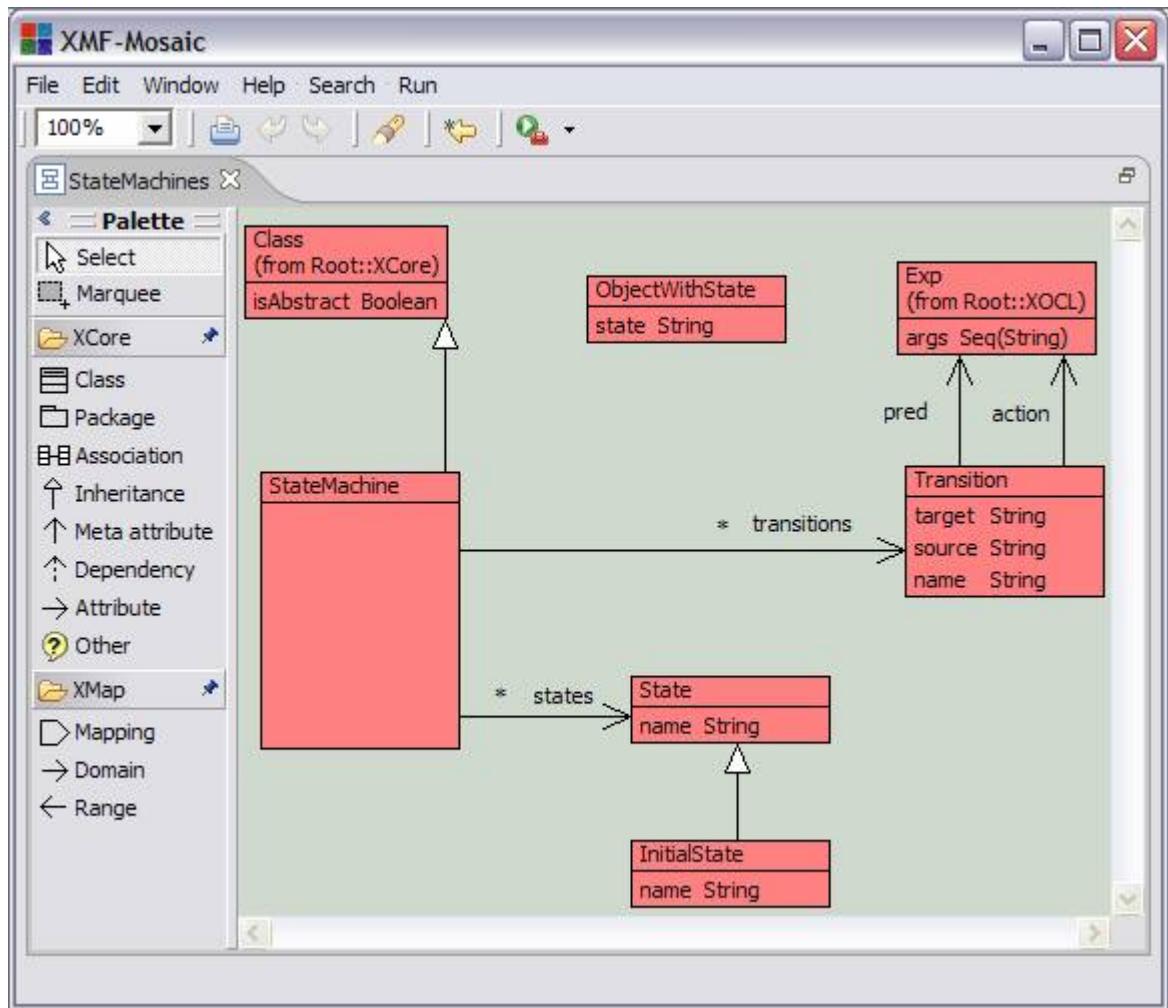
State machines are ubiquitous in Software Engineering. Whilst the basic notion of a state machine is very simple (the idea that components have a number of states and execution occurs when the component changes from one state to another) there are a huge number of ways in which state machines are implemented and deployed.

Fortunately, XMF-Mosaic makes it very easy to construct a state machine model and to define its semantics. This section shows how XCore::Class can be extended to provide states and transitions. The example shows a number of technologies:

- A new language for defining state machines. The language provides constructs for state machines; states; initial states and transitions. Transitions have guards and actions both of which are implemented using the XOCL::Exp.

- Meta-classes. The class StateMachine is an extension of XCore::Class and as such supports class-contents such as attributes and operations. The new meta-class defines attributes for states and transitions and a number of meta-operations for running the state machine against objects with state.
- Daemons. An instance of a state machine class is an object with state. A state machine monitors changes in the state of its instances by placing a daemon on each instance. When the daemon fires, the state machine determines whether any of its transitions are enabled and performs any enabled actions.

The following model shows the classes in the state machine language:

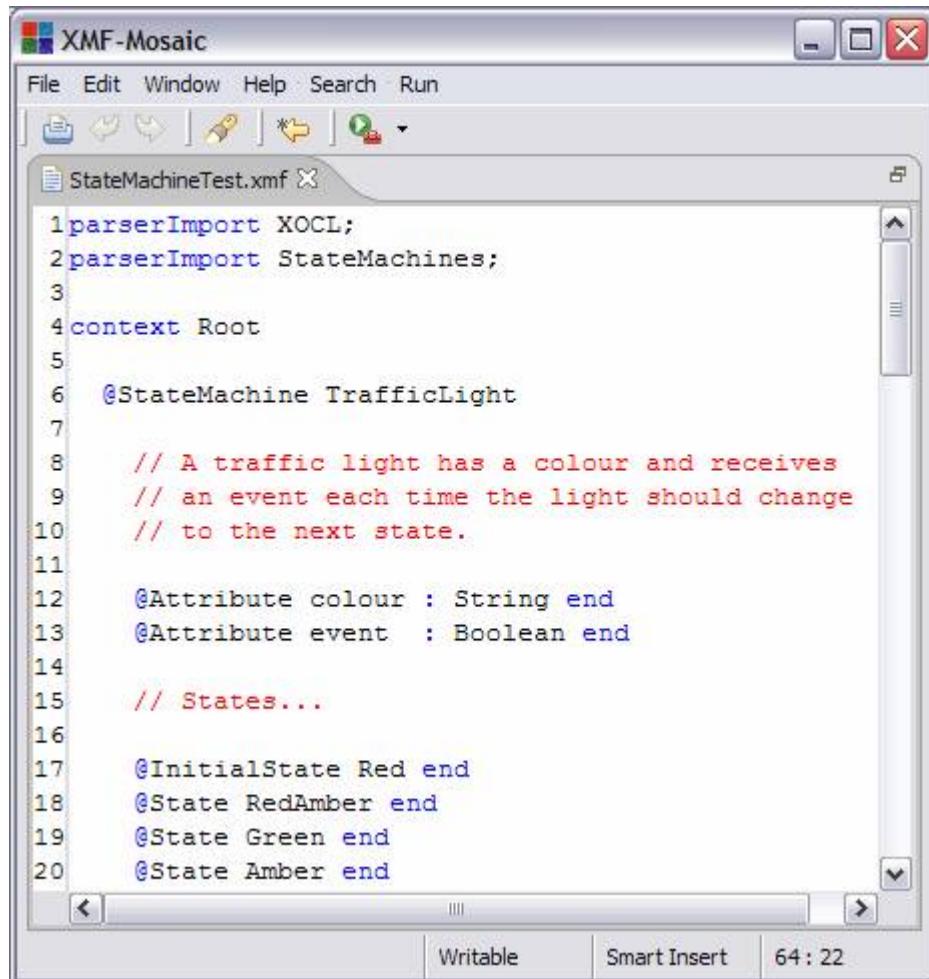


The key features of the model are as follows:

- Instances of StateMachine are classes with states and transitions. Therefore we can have instances of state machines.
- The class ObjectWithState is defined to be a parent of any state machine. This class defines an attribute state that records the current state of the state machine instance.
- Transitions have predicates and actions that are implemented as expressions. The expressions will be run against an object with state when it changes.

The model above defines a domain specific language for representing state machines. In addition to defining the structure of state machines, the model defines the executable semantics for state machines. Before showing how the semantics is defined, we give two examples of how state machines are used to implement simple applications. The first example shows traffic lights and the second example shows a vending machine.

The following state machine implements a simple traffic light system:

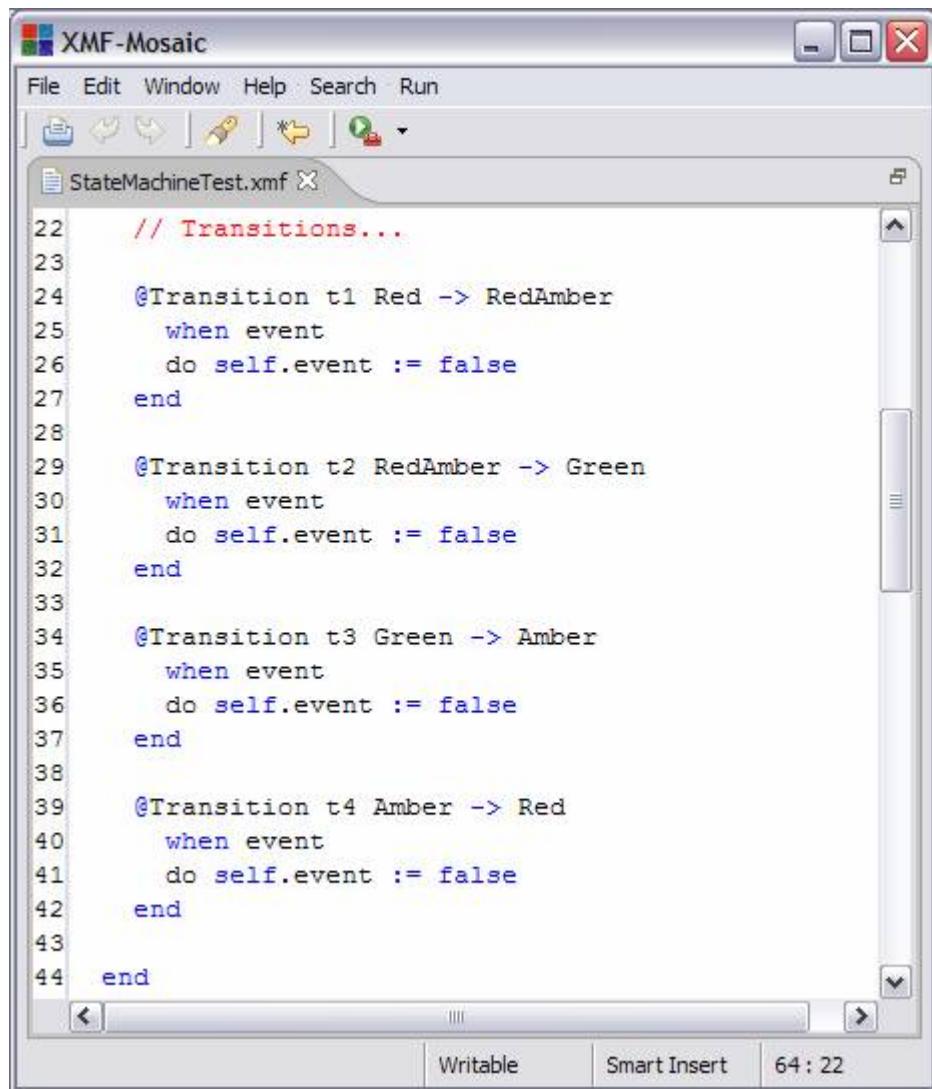


The screenshot shows the XMF-Mosaic editor window with the file "StateMachineTest.xmf" open. The code defines a state machine named "TrafficLight" with four states: Red, RedAmber, Green, and Amber. It includes imports for Xocl and StateMachines, and defines attributes for colour and event.

```
1 parserImport Xocl;
2 parserImport StateMachines;
3
4 context Root
5
6 @StateMachine TrafficLight
7
8     // A traffic light has a colour and receives
9     // an event each time the light should change
10    // to the next state.
11
12    @Attribute colour : String end
13    @Attribute event : Boolean end
14
15    // States...
16
17    @InitialState Red end
18    @State RedAmber end
19    @State Green end
20    @State Amber end
```

Line 6 introduces a state machine and names it. State machines may contain the same elements as classes and line 12 and 13 define the state that is manipulated by the state machine. The event attribute is monitored by the state machine and corresponds to events received from an external source such as a timer.

Lines 17 – 20 define the states for the machine. At any time an instance of TrafficLight must be in one of the states: Red, RedAmber, Green or Amber. The transitions that control how a traffic light changes between these states are defined below:

A screenshot of the XMF-Mosaic editor window. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations. The main area shows a code editor with the file "StateMachineTest.xmf" open. The code is a state machine definition:

```
22 // Transitions...
23
24 @Transition t1 Red -> RedAmber
25   when event
26     do self.event := false
27   end
28
29 @Transition t2 RedAmber -> Green
30   when event
31     do self.event := false
32   end
33
34 @Transition t3 Green -> Amber
35   when event
36     do self.event := false
37   end
38
39 @Transition t4 Amber -> Red
40   when event
41     do self.event := false
42   end
43
44 end
```

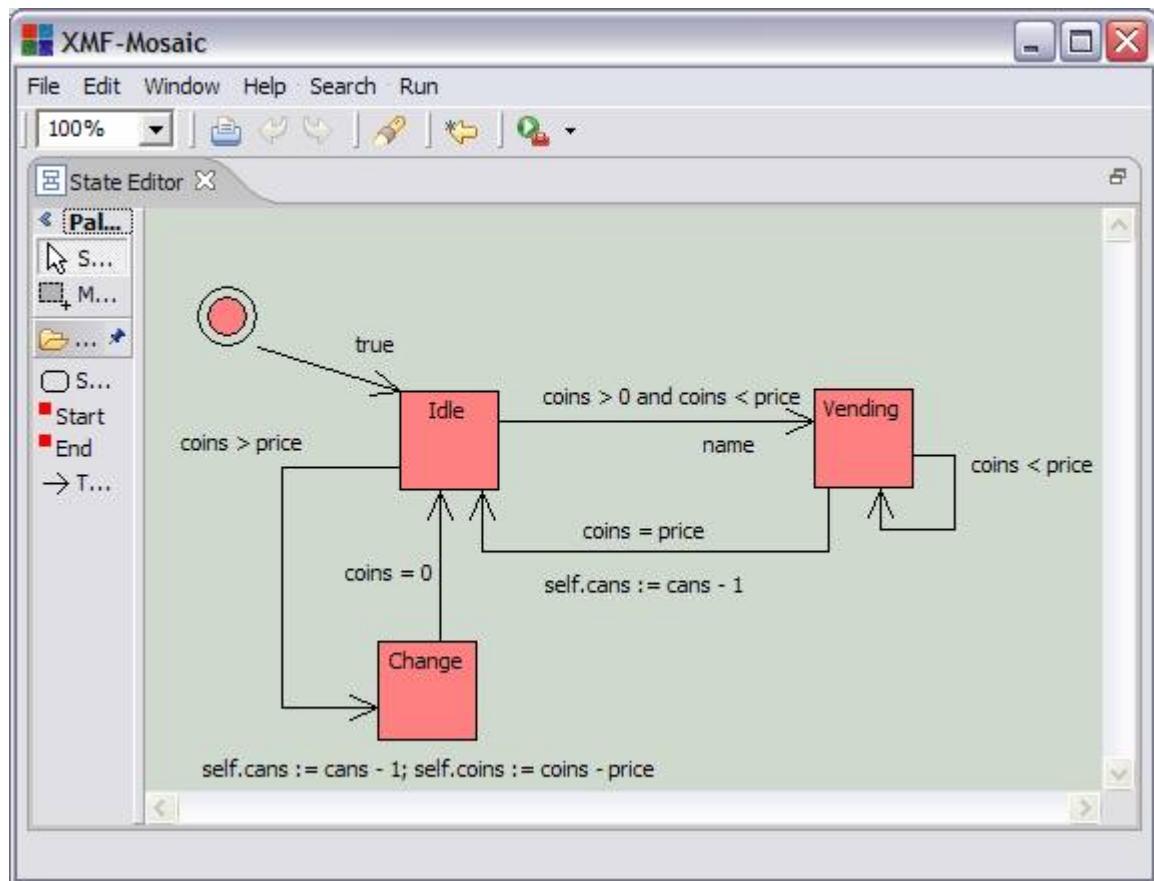
The status bar at the bottom shows "Writable", "Smart Insert", and the time "64: 22".

A transition has a name, a source and target state name, a guard and an action. For example the transition defined in line 24 is named t1, changes between state Red and RedAmber, occurs when the traffic light state changes so that event is true and updates the value of the event slot to be false.

A traffic light is defined to change state when it receives an event (the value of the event slot changes to be true). The state is changed and the event flag is reset.

A vending machine dispenses cans of drink. Each can has a price; a customer enters coins into the machine until the value of the coins exceeds the price of a can. A can is dispensed and the machine waits for any change to be withdrawn before waiting for the next customer.

A vending machine is either idle (waiting for a customer), vending (accepting coins up to the price of a can), or waiting for change to be removed. The vending machine state machine is defined below:



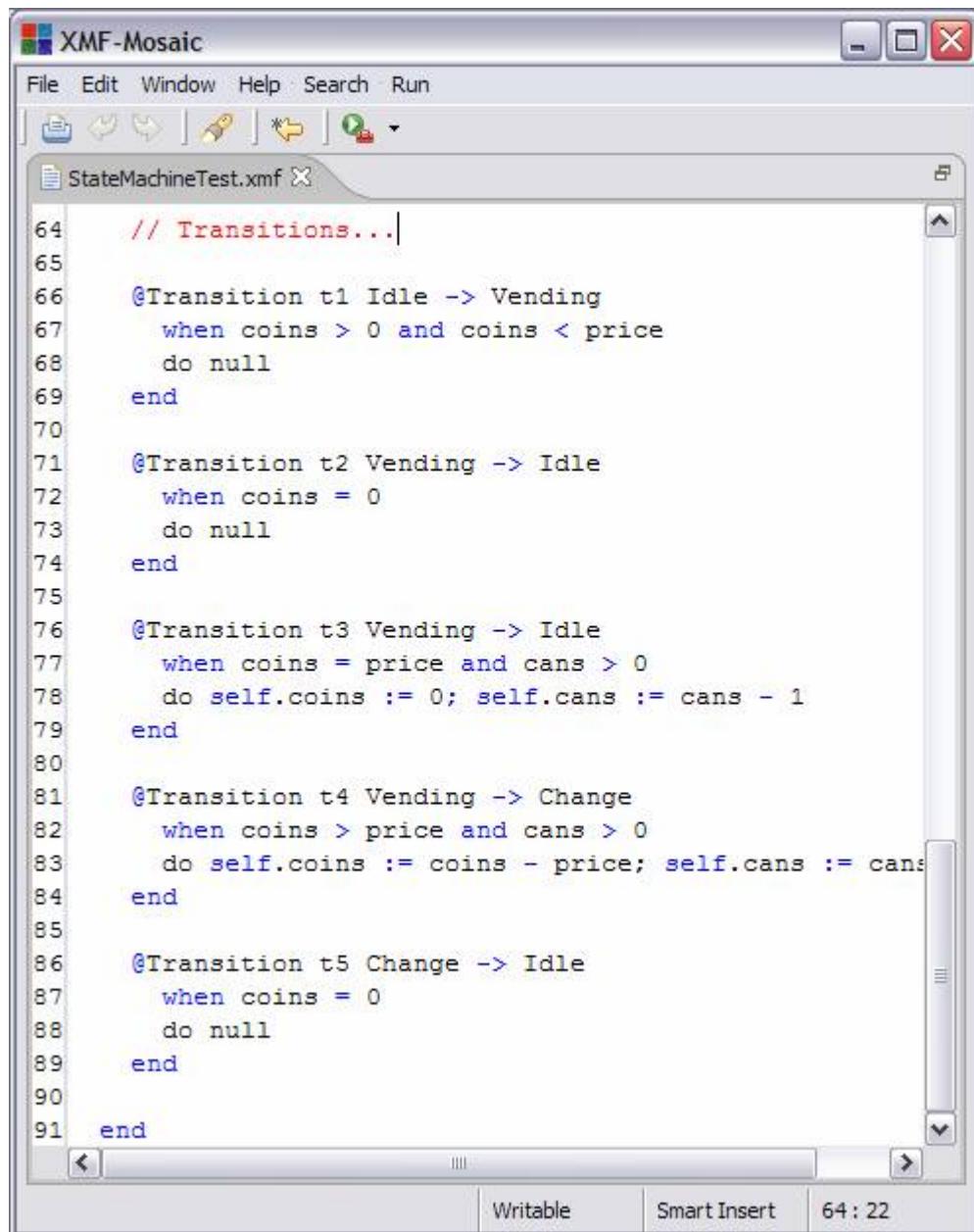
As a textual definition:

```

46 context Root
47 @StateMachine VendingMachine
48
49 // A vending machine accepts coins and dispenses
50 // cans at a predefined price. If more coins than
51 // necessary are inserted then the change must be
52 // released before the next can can be dispensed.
53
54 @Attribute coins : Integer end
55 @Attribute cans : Integer end
56 @Attribute price : Integer end
57
58 // States...
59
60 @InitialState Idle end
61 @State Vending end
62 @State Change end

```

The transitions for a vending machine are defined below:



The screenshot shows the XMF-Mosaic editor interface with the file "StateMachineTest.xmf" open. The code defines five transitions for a vending machine state machine:

```
64 // Transitions...
65
66 @Transition t1 Idle -> Vending
67     when coins > 0 and coins < price
68     do null
69 end
70
71 @Transition t2 Vending -> Idle
72     when coins = 0
73     do null
74 end
75
76 @Transition t3 Vending -> Idle
77     when coins = price and cans > 0
78     do self.coins := 0; self.cans := cans - 1
79 end
80
81 @Transition t4 Vending -> Change
82     when coins > price and cans > 0
83     do self.coins := coins - price; self.cans := cans - 1
84 end
85
86 @Transition t5 Change -> Idle
87     when coins = 0
88     do null
89 end
90
91 end
```

The editor has a toolbar with icons for file operations, a search bar, and a status bar at the bottom indicating "Writable", "Smart Insert", and the time "64:22".

The semantics of state machines is defined as a package below:

```

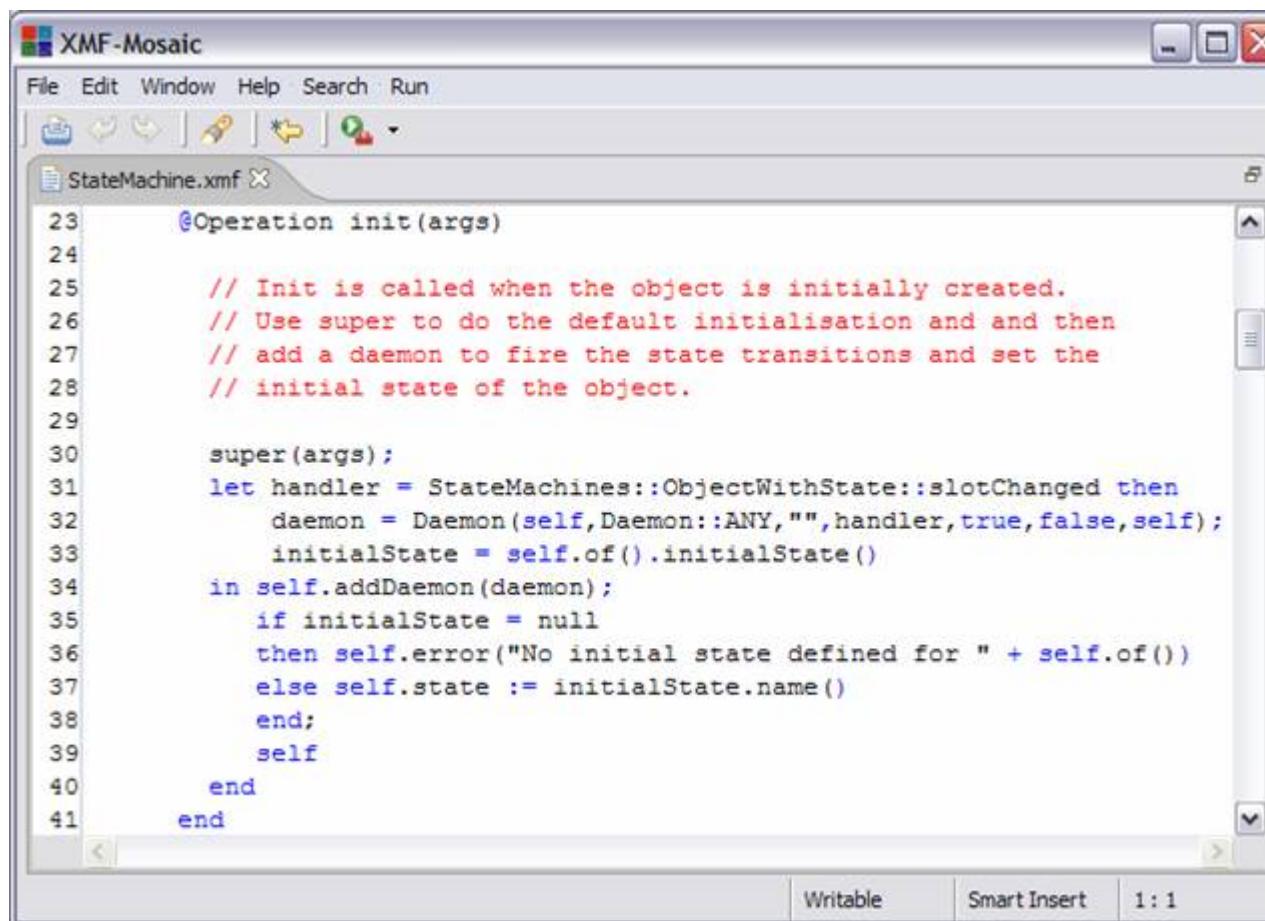
1 parserImport Xocl;
2 parserImport Parser::BNF;
3
4 import OCL;
5
6 context Root
7
8 @Package StateMachines
9
10 // A package of classes that implement an event driven
11 // collection of state machine classes.
12
13 @Class ObjectWithState
14
15 // An object with state must be an instance of a state
16 // machine. The object maintains the name of the state
17 // and monitors state changes via a daemon. When the
18 // daemon fires, the transitions of the state machine
19 // control the actions that take place.
20
21 @Attribute state : String (?,! ) end

```

The class ObjectWithState is typical of a pattern occurring when defining new meta-classes. The new meta-class StateMachine (below) requires that its instances have a support system. In this case the support system is the attribute state and associated operations. When instances of StateMachine are created, the class ObjectWithState is added as a parent of the resulting class. This ensures that instances of instance of StateMachine have the correct basic slots and supporting operations.

When an XMF object is created via a constructor, the init operation is called with the constructor arguments. An object with state needs to install a daemon that monitors state changes (any slot modification). The daemon inspects the state machine controlling the object and fires any enabled state transition. In addition an object with state must initialise the state slot.

The init operation for ObjectWithState is defined below:



```

23     @Operation init(args)
24
25         // Init is called when the object is initially created.
26         // Use super to do the default initialisation and then
27         // add a daemon to fire the state transitions and set the
28         // initial state of the object.
29
30     super(args);
31     let handler = StateMachines::ObjectWithState::slotChanged then
32         daemon = Daemon(self,Daemon::ANY,"",handler,true,false,self);
33         initialState = self.of().initialState()
34     in self.addDaemon(daemon);
35     if initialState == null
36     then self.error("No initial state defined for " + self.of())
37     else self.state := initialState.name()
38     end;
39     self
40   end
41 end

```

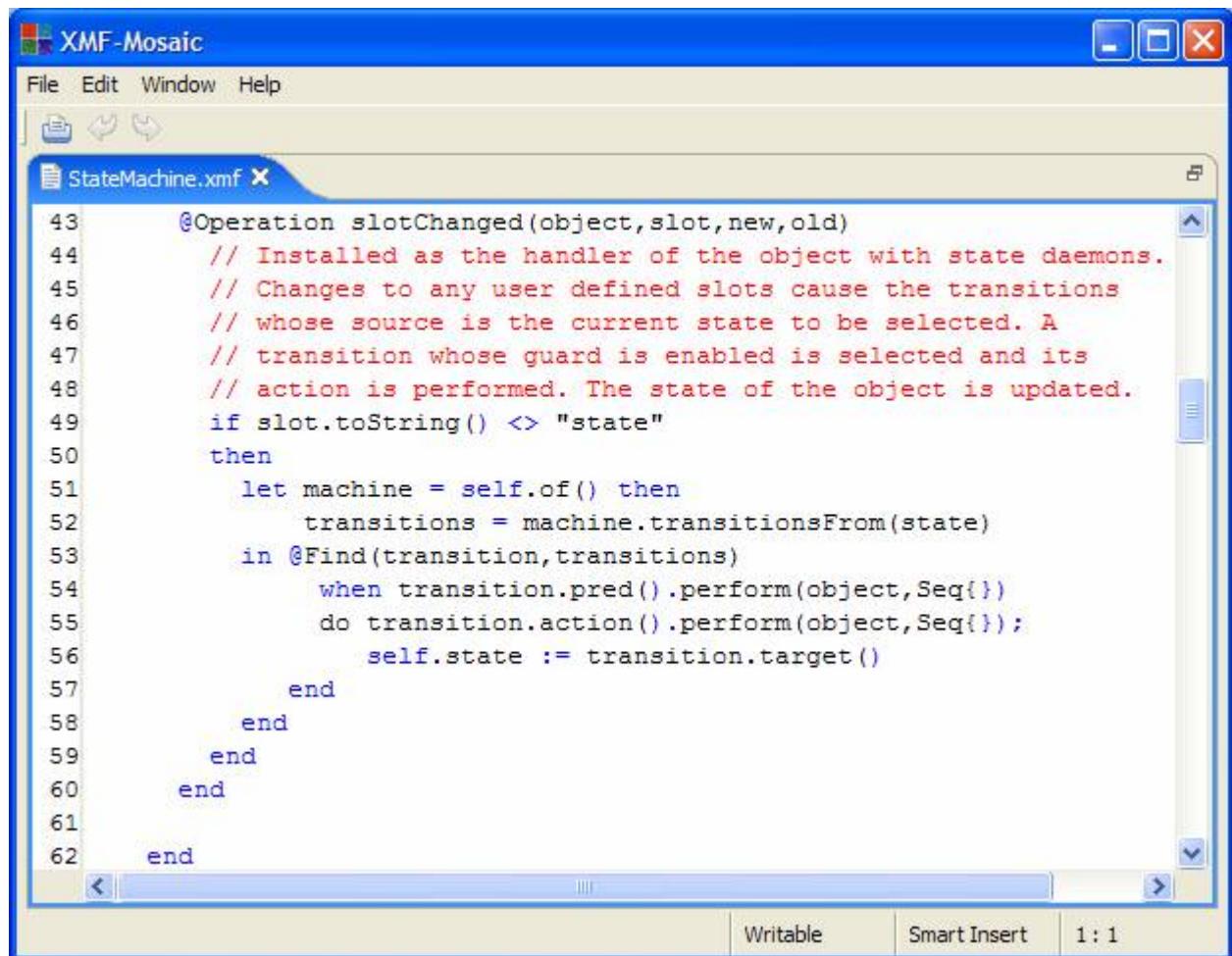
Writable Smart Insert 1:1

The daemon added to an object with state uses a handler slotChanged. This handler is called whenever a slot of the object monitored by the daemon changes. A daemon may be added to more than one object (here the daemon is created on a per-object basis). The type of the daemon determines which slot it monitors. In this case the type Daemon::ANY defines that the handler is called when any slot of a monitored object changes.

The handler used by the daemon is defined by the class ObjectWithState. A daemon handler is passed four arguments:

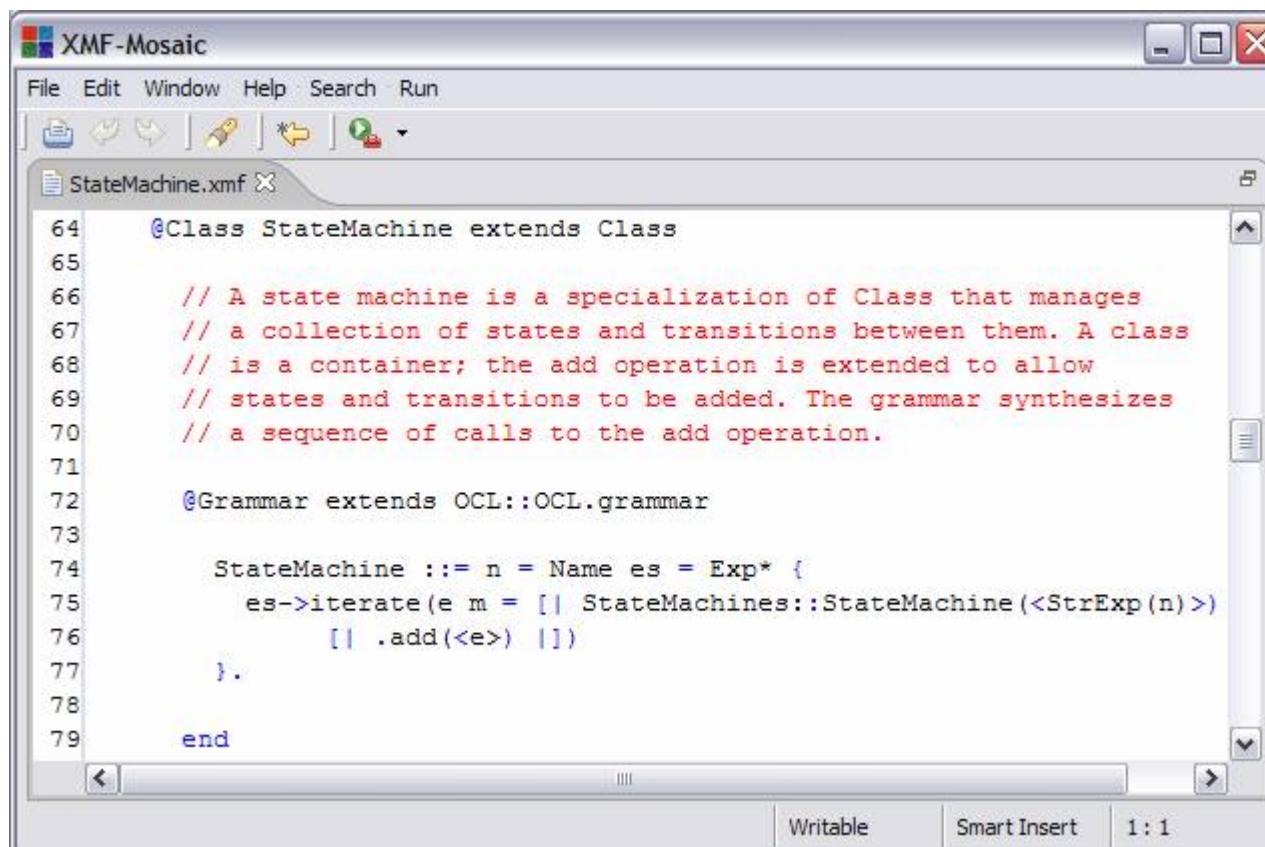
- The object whose state has changed.
- The name of the slot that has changed.
- The new value of the slot.
- The old value of the slot.

In the following we check that the slot is not the state slot and then calculate the transitions that are enabled. If any transitions are enabled then one is selected at random and its action is performed. Both the predicate and the action are expressions. An expression has an operation perform that is used to evaluate the expression. The perform operation is supplied with a value for self in the expression and a sequence of local variable values. The target argument is used in the same way as that to Operation::invoke where unqualified slot references are resolved with respect to the supplied target object.

A screenshot of the XMF-Mosaic editor window. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main area is a code editor titled "StateMachine.xmf" with the ".xmf" extension. The code is written in a domain-specific language, likely Ecore, with syntax highlighting. The code defines an operation slotChanged that handles state transitions based on guard conditions and actions. The code is numbered from 43 to 62. At the bottom of the editor are buttons for "Writable", "Smart Insert", and a status bar showing "1:1".

```
43     @Operation slotChanged(object,slot,new,old)
44         // Installed as the handler of the object with state daemons.
45         // Changes to any user defined slots cause the transitions
46         // whose source is the current state to be selected. A
47         // transition whose guard is enabled is selected and its
48         // action is performed. The state of the object is updated.
49         if slot.toString() <> "state"
50             then
51                 let machine = self.of() then
52                     transitions = machine.transitionsFrom(state)
53                     in @Find(transition,transitions)
54                         when transition.pred().perform(object,Seq{})
55                         do transition.action().perform(object,Seq{});
56                         self.state := transition.target()
57                 end
58             end
59         end
60     end
61
62 end
```

The class StateMachine is defined as a meta-class that creates sub-classes of ObjectWithState:



The screenshot shows the XMF-Mosaic editor interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", "Help", "Search", and "Run". Below the menu is a toolbar with icons for file operations like Open, Save, and Find. The main window displays a file named "StateMachine.xmf" with the following content:

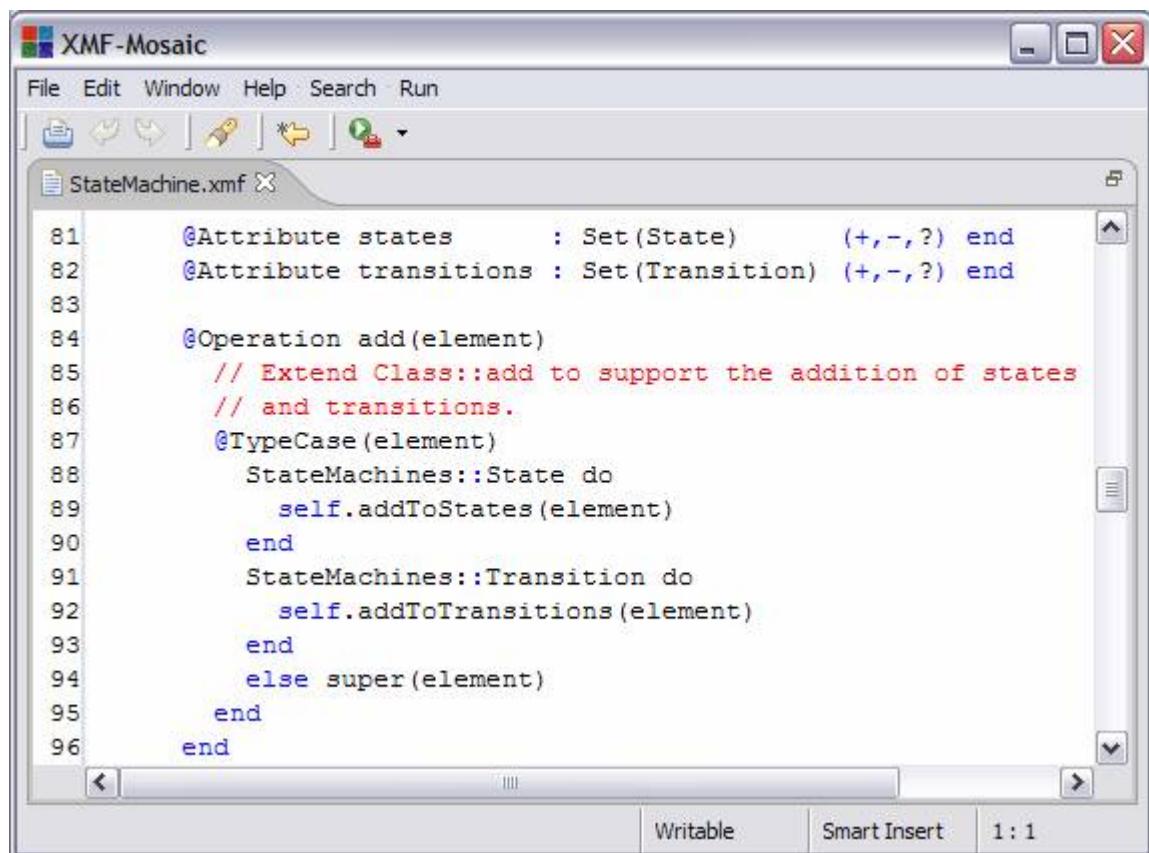
```

64  @Class StateMachine extends Class
65
66      // A state machine is a specialization of Class that manages
67      // a collection of states and transitions between them. A class
68      // is a container; the add operation is extended to allow
69      // states and transitions to be added. The grammar synthesizes
70      // a sequence of calls to the add operation.
71
72  @Grammar extends OCL::OCL.grammar
73
74  StateMachine ::= n = Name es = Exp* {
75      es->iterate(e m = [] StateMachines::StateMachine(<StrExp(n)>)
76          [| .add(<e>) |])
77      }.
78
79  end

```

The code defines a new meta-class "StateMachine" that extends "Class". It includes a detailed comment explaining how it handles state transitions. The grammar part defines the syntax for creating state machines with a name and a collection of expressions. The implementation part shows how to iterate over these expressions and add them to a state machine.

A state machine is a new language feature. The grammar is defined on lines 72 – 79 and is a simple example of how to define a new syntax feature for a new type of meta-class. The syntax definition for Class (as in @Class X ... end) expands to a collection of calls to add for each contained definition (for example attributes and operations). A class knows how to add the supplied elements. This is exploited above, by just allowing state machines to contain arbitrary expressions each of which will be evaluated and the resulting element supplied as an argument to add on the state machine. We then leave it to an extended definition of add in StateMachine to handle any new types of element that we wish to add to a state machine, leaving the definition of Class::add to handle the rest:



The screenshot shows the XMF-Mosaic IDE interface with the title bar "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations. The main window displays the file "StateMachine.xmf" with the following code:

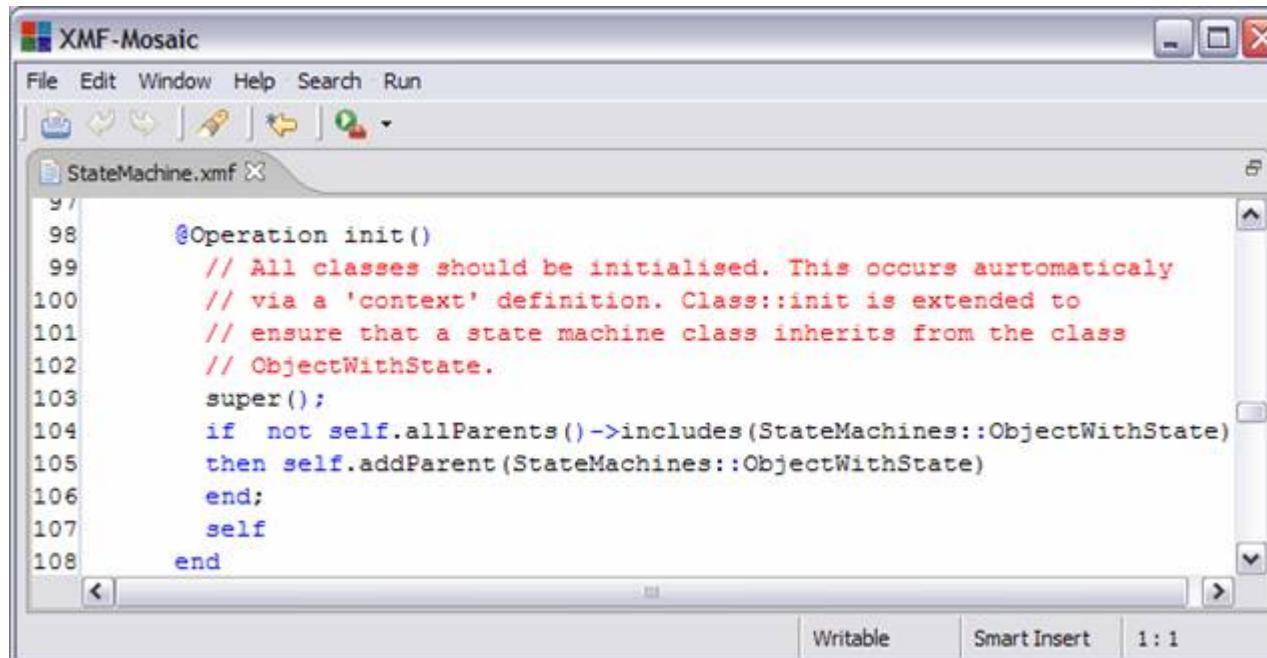
```

81     @Attribute states      : Set(State)      (+,-,?) end
82     @Attribute transitions : Set(Transition) (+,-,?) end
83
84     @Operation add(element)
85         // Extend Class::add to support the addition of states
86         // and transitions.
87         @TypeCase(element)
88             StateMachines::State do
89                 self.addToStates(element)
90             end
91             StateMachines::Transition do
92                 self.addToTransitions(element)
93             end
94             else super(element)
95         end
96     end

```

The status bar at the bottom shows "Writable", "Smart Insert", and "1:1".

When a new state machine is created we must arrange for it to inherit from `ObjectWithState` so that its instances get the state slot and associated initialisation support. All name spaces must be initialised via the `init()` operation prior to use. Normally this occurs automatically and you are not aware of the call to `init` (for example defining context `P @Class C ... end` automatically calls `init` when `C` is added to `P`). We extend the initialisation below to add in the required super-class:



The screenshot shows the XMF-Mosaic IDE interface with the title bar "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations. The main window displays the file "StateMachine.xmf" with the following code:

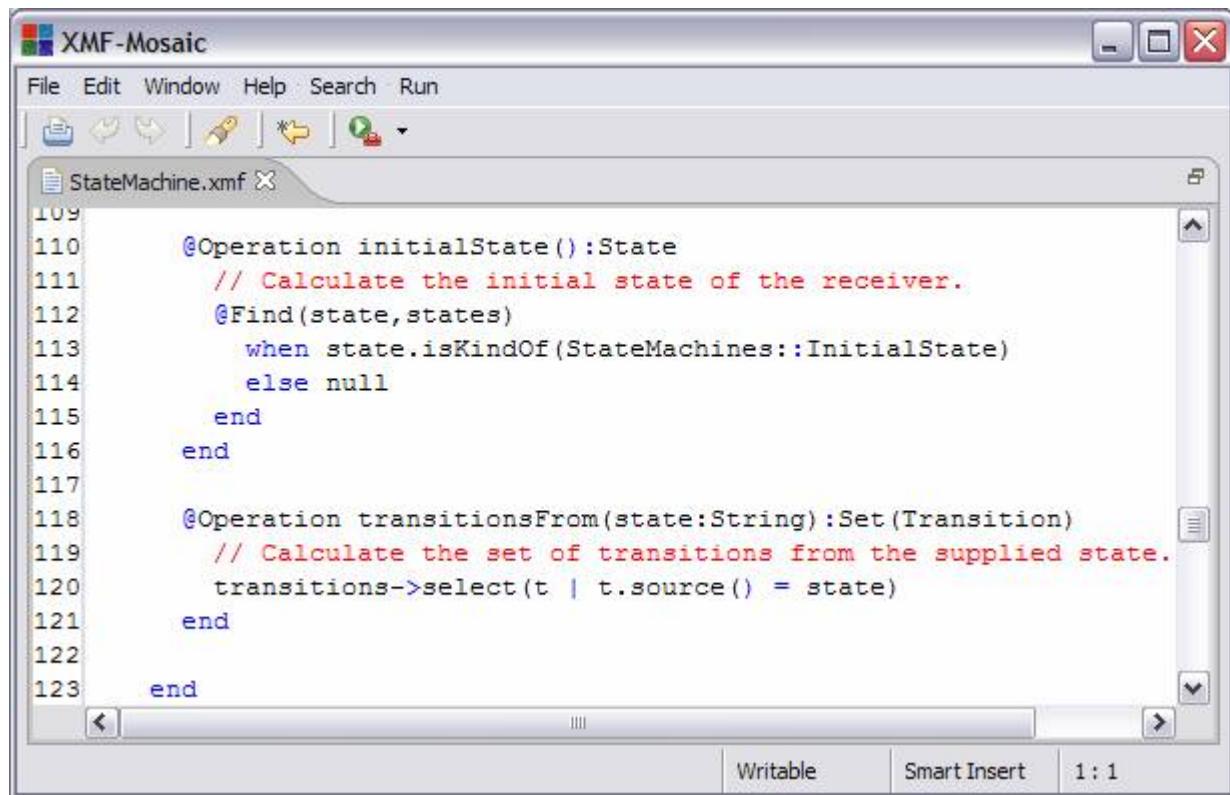
```

97
98     @Operation init()
99         // All classes should be initialised. This occurs automatically
100        // via a 'context' definition. Class::init is extended to
101        // ensure that a state machine class inherits from the class
102        // ObjectWithState.
103        super();
104        if not self.allParents()->includes(StateMachines::ObjectWithState)
105        then self.addParent(StateMachines::ObjectWithState)
106        end;
107        self
108    end

```

The status bar at the bottom shows "Writable", "Smart Insert", and "1:1".

Operations used via `ObjectWithState`:



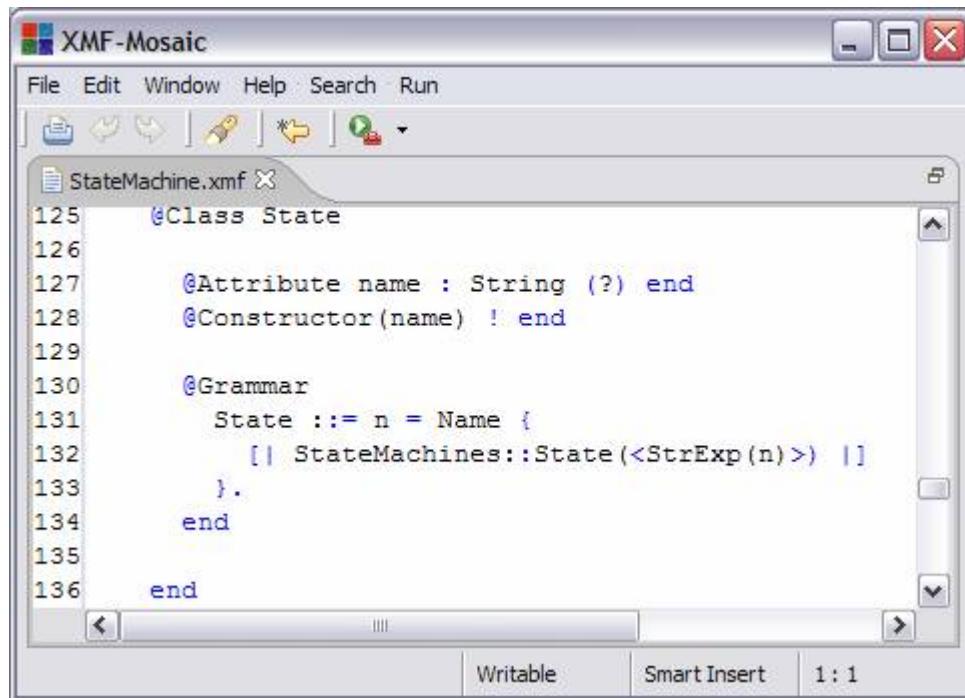
The screenshot shows the XMF-Mosaic editor window with the file "StateMachine.xmf" open. The code defines two operations: `initialState()` and `transitionsFrom(state)`. The `initialState()` operation calculates the initial state of the receiver by finding a state that is kind of `InitialState`. The `transitionsFrom(state)` operation calculates the set of transitions from the supplied state by selecting transitions where the source is the state.

```

109
110     @Operation initialState():State
111         // Calculate the initial state of the receiver.
112         @Find(state,states)
113             when state.isKindOf(StateMachines::InitialState)
114             else null
115         end
116     end
117
118     @Operation transitionsFrom(state:String):Set(Transition)
119         // Calculate the set of transitions from the supplied state.
120         transitions->select(t | t.source() = state)
121     end
122
123 end

```

In this example of state machines, a state is just a name. We define a new syntax construct for defining a state. A more sophisticated implementation of state machines might, for example, introduce enter and exit actions for states or allow nested state machines:



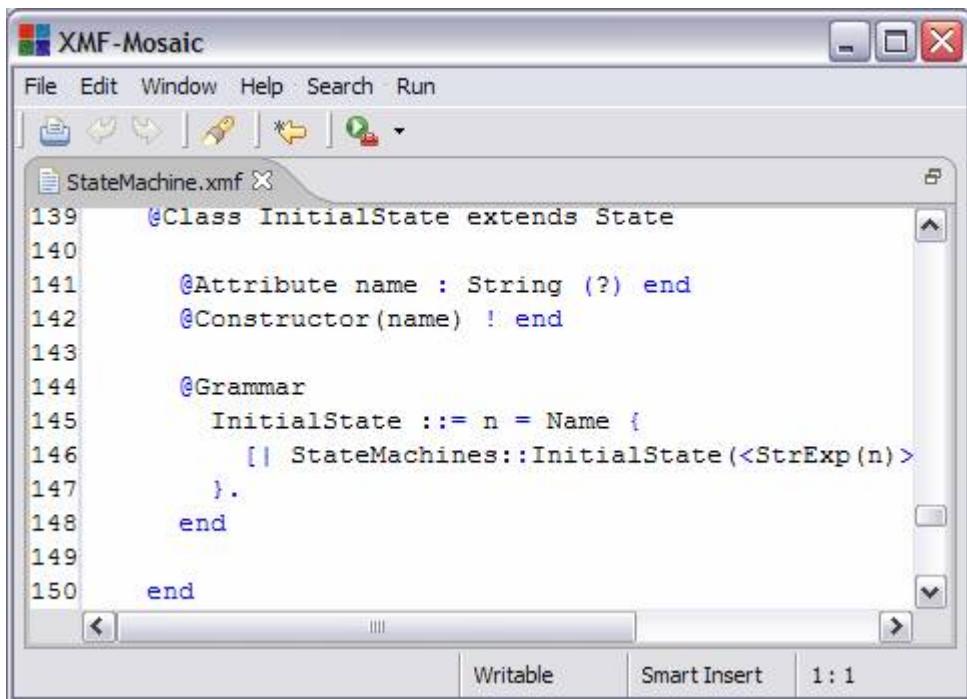
The screenshot shows the XMF-Mosaic editor window with the file "StateMachine.xmf" open. The code defines a class `State` with an attribute `name` and a constructor `(name)`. It also defines a grammar for `State` which matches a name and creates a `StateMachines::State` object with it.

```

125     @Class State
126
127         @Attribute name : String (?) end
128         @Constructor(name) ! end
129
130         @Grammar
131             State ::= n = Name {
132                 []| StateMachines::State(<StrExp(n)>) []
133             }.
134         end
135
136     end

```

An initial state is just a state. We use the type of an initial state to select the initial state of a state machine:

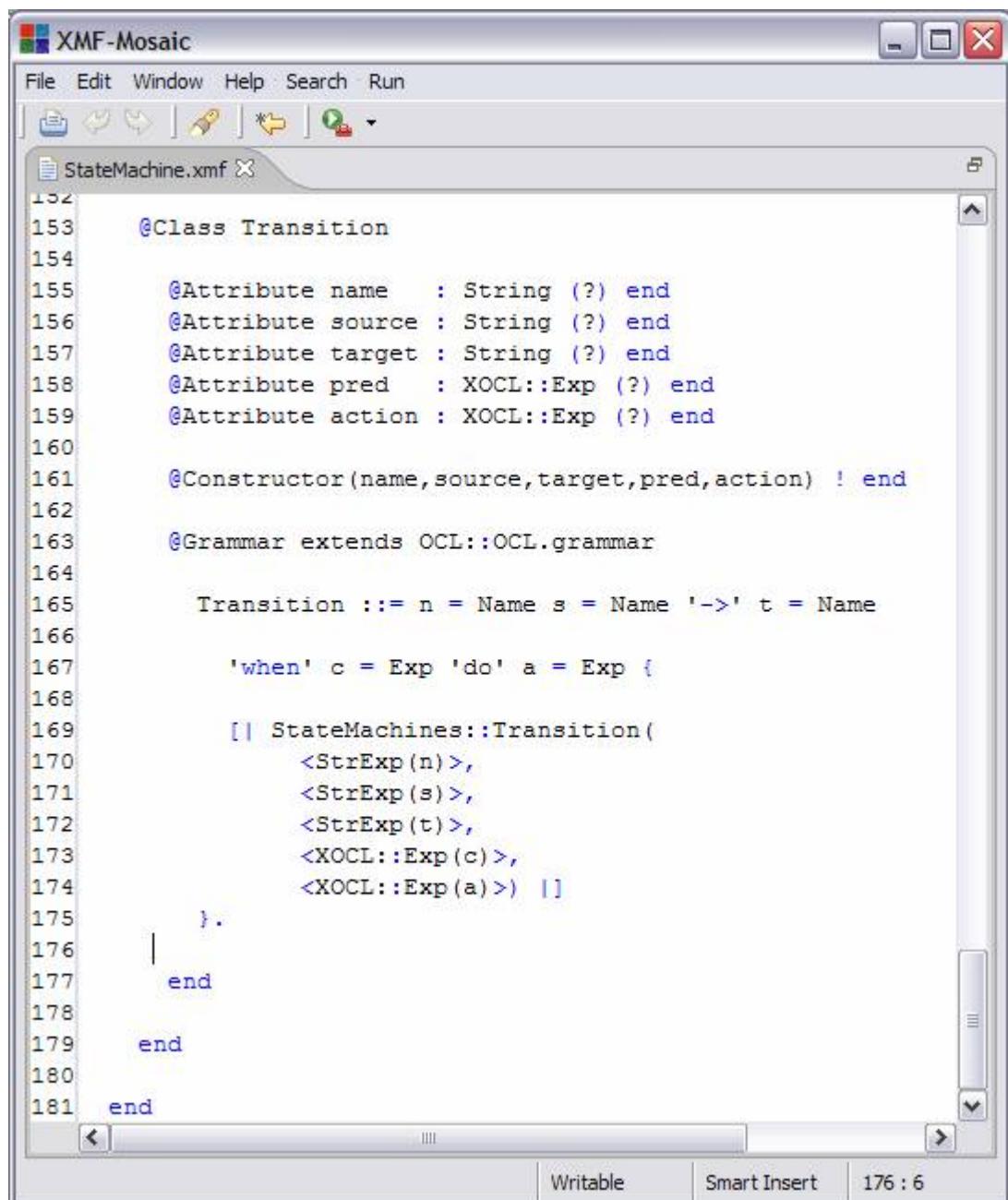


The screenshot shows the XMF-Mosaic editor window with the title "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for new, open, save, cut, copy, paste, and search. The main editor area displays the file "StateMachine.xmf" with the following code:

```
139     @Class InitialState extends State
140
141         @Attribute name : String (?) end
142         @Constructor(name) ! end
143
144         @Grammar
145             InitialState ::= n = Name {
146                 []| StateMachines::InitialState(<StrExp(n)>
147                 ).
148             end
149
150         end
```

The status bar at the bottom shows "Writable", "Smart Insert", and "1:1".

A transition is implemented as a new syntax construct as follows. The guard and action on a transition are implemented as an expression (XOCL::Exp) that is performed in order to check whether the transition is enabled and when the transition fires respectively:



The screenshot shows the XMF-Mosaic editor window with the file "StateMachine.xmf" open. The code defines a class Transition with attributes name, source, target, pred, and action, and a constructor that takes name, source, target, pred, and action. It also extends OCL::OCL.grammar and defines a Transition rule with parameters n, s, t, c, and a, and a body that creates a StateMachines::Transition object with those values.

```
152
153     @Class Transition
154
155         @Attribute name    : String (?) end
156         @Attribute source  : String (?) end
157         @Attribute target  : String (?) end
158         @Attribute pred   : XOCL::Exp (?) end
159         @Attribute action  : XOCL::Exp (?) end
160
161     @Constructor(name,source,target,pred,action) ! end
162
163     @Grammar extends OCL::OCL.grammar
164
165     Transition ::= n = Name s = Name '-'> t = Name
166
167     'when' c = Exp 'do' a = Exp {
168
169         [ | StateMachines::Transition(
170             <StrExp(n)>,
171             <StrExp(s)>,
172             <StrExp(t)>,
173             <XOCL::Exp(c)>,
174             <XOCL::Exp(a)>) | ]
175     } .
176
177     end
178
179 end
180
181 end
```

Chapter 18. XCore

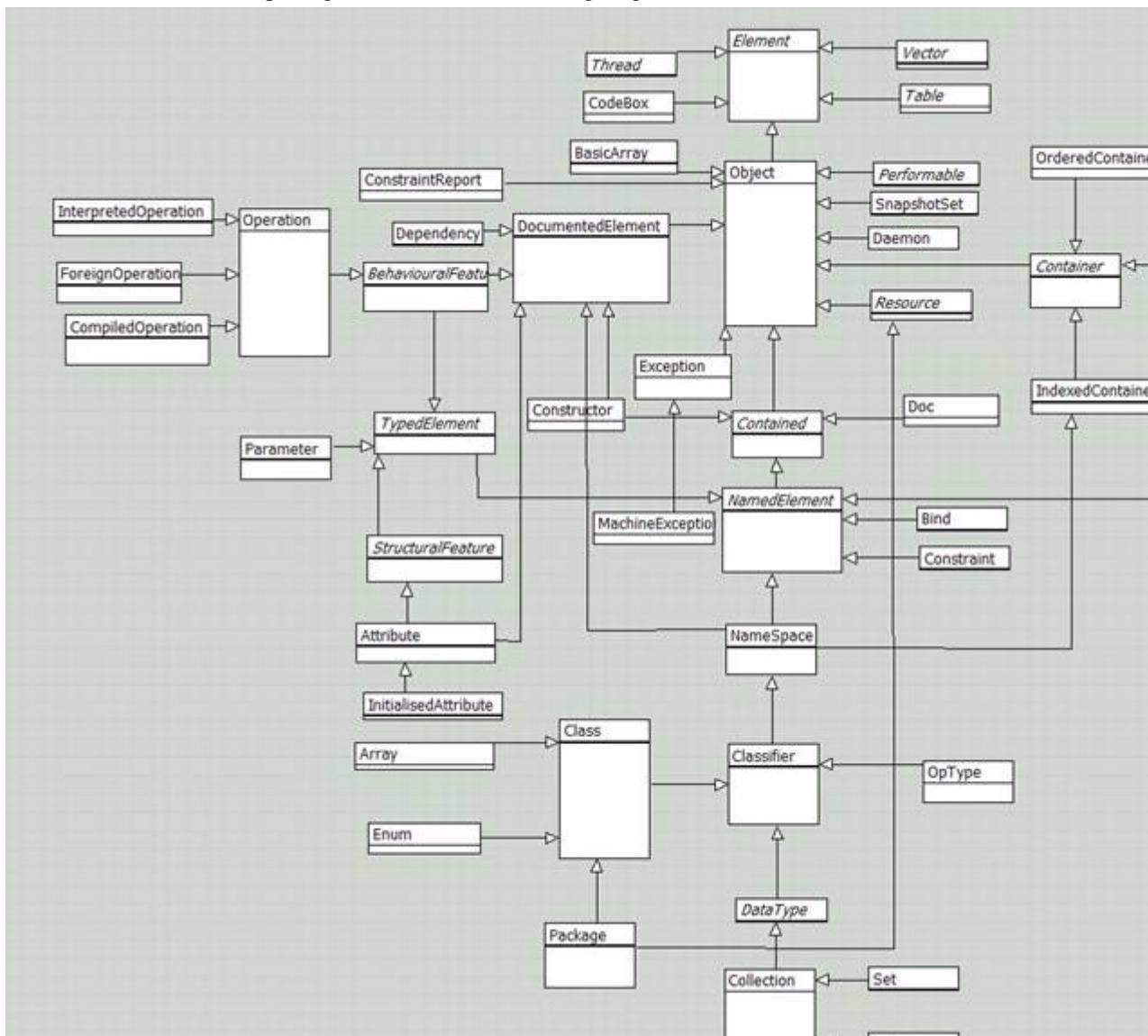
Introduction

XMF provides a collection of classes that form the basis of all XMF-Mosaic defined tools. These classes form the kernel of XMF and are collectively called XCore. The classes are self-describing: all XCore classes are instances of XCore classes. This feature is called meta-circularity and is the key to modularity, uniformity and reusability throughout all system and user defined XMF tools.

XCore includes class definitions for the basic types including Integer, Boolean and String and collection types for sets and sequences of values. XCore is object-oriented, it provides basic notions of Object and Class. XCore is executable; it provides definitions for Performable entities and Operations. All system defined tools in XMF-Mosaic are instances of XCore; therefore tools that are defined to work on instances of XCore can be used on any XMF data. For example, general-purpose editors and mappings are provided by XMF that are guaranteed to work across all system and user defined data.

The XCore Package

The contents of the XCore package is shown in the following diagram:



The following sections provide a brief summary of each class. Note, the XCore model can be explored by clicking on Kernel > XCore in the XMF browser.

Attribute

An attribute is a structural feature of a class. It defines the name and type of a slot of the instances of the class. When the class is instantiated, a new object is created and a slot is added for each attribute defined and inherited by the class. Each slot is initialised to contain the default value for the type of the corresponding attribute. Constructor: Attribute(name,type) The name is a string and the type is a classifier.

Behavioural Feature

A behavioural feature is a typed element that can be invoked. Typically a behavioural feature is an operation.

Bind

A binding associates a name with a value. Constructor: Bind(name,value) Constructs a binding, the name is a string and the value is any element.

Classifier

A classifier is a name space for operations and constraints. A classifier is generalizable and has parents from which it inherits operations and constraints. A classifier can be instantiated via new/0 and new/1. In both cases the default behaviour is to return a default value as an instance. If the classifier is a datatype then the basic value for the datatype is returned otherwise null is returned as the default value. A classifier can also be applied to arguments (0 or more) in order to instantiate it. Typically you will not create a Classifier directly, but create a class or an instance of a sub-class of Class.

Class

XMF-Mosaic is a class-based system. Tools are defined as collections of classes whose instances have state and behaviour. The XCore class Class defines the essential features of a class. Inheritance is used to extend class features in XMF. Since Class is available; user defined tools can extend what it means to be a class. This ability is the basis for meta-programming. For example, Class may be extended with the ability to keep track of all instances or to access instance data from an external database.

CodeBox

A CodeBox contains compiled code that can be executed on the XMF VM.

Collection

The root class for all collection types.

Compiled Operation

A CompiledOperation is the type of all XMF compiled operations. A compiled operation can be invoked using invoke/2 or by applying it to its arguments. A compiled operation consists of machine code instructions. A compiled operation may be associated with its source code to aid debugging.

Constraint

A constraint is a named Boolean expression owned by a classifier. Constraints are defined by classifiers to be performed with respect to their instances and as such any occurrences of self in a constraint will refer to the instance that is being checked.

Constraint Report

A constraint report is produced by sending a classify message to an element or a check constraints message to a classifier. A report is a tree structured element describing the constraints that were performed and their outcomes. Note that internal nodes of the tree may have dummy constraint reports used as containers of sub-constraint reports. Such dummies have a null constraint and an empty reason, but the satisfied Boolean is the conjunction of the sub-reports.

Constructor

A constructor contains a code body that is invoked on instantiation of a classifier.

Contained

A contained element has an owner. The owner is set when the contained element is added to a container. Removing an owned element from a container and adding it to another container will change the value of owner in the contained element.

Container

A container has a slot contents that is a table. The table maintains the contained elements indexed by keys. By default the keys for the elements in the table are the elements themselves, but sub-classes of container will modify this feature accordingly. Container provides operations for accessing and managing its contents.

Daemon

Daemons monitor the state of objects and perform actions when the object changes state. Daemon technology is the key to implementing a variety of modular reusable tool architectures such as the observer pattern. XMF-Mosaic uses daemons extensively to synchronize data across multiple tools. User defined tools can use daemons to make tools reactive and to ensure data is always consistent.

Data Type

Instances of this meta-class are XMF types for basic data values. XMF types include Integer, String, Boolean and Float.

Dependency

A dependency occurs between a source and a target model element. When a dependency is created, the attach operation is performed and when it is removed the detach operation is performed. These operations allow sub-classes of Dependency to have semantics.

Doc

A class used to represent documentation.

DocumentedElement

A documented element has an attribute doc:Doc, which is used to store documentation relating to the element. Any class that can be documented should specialize this class.

Element

All classes in XMF are extensions of the XCore class Element. Element defines the essential behaviour of all XMF data. For example Element defines features such as being able to produce a printed

representation and the ability to handle messages. XMF is a dynamically extensible system; this means that new behaviour can be added to existing classes. This is sometimes referred to as aspect oriented programming. Since Element is available, user defined tools can add system-wide aspects. For example this can be used to add the ability to export any XMF data in any required format (binary encoded, XML, text etc).

Enum

The enumerated type.

Exception

XMF provides exception handling for dealing with exceptional circumstances in running code. The XCore class Exception is the basis for a hierarchy of classes that implement specific types of errors. Exceptions are raised at the point at which they occur and encapsulate data that describes exactly the source of the problem.

ForeignOperation

Provides an interface to operations written in external programming languages, such as Java.

IndexedContainer

An indexed container is a class that manages a hashtable associating keys with values. The Container::add/1 operation is implemented by IndexedContainer to add the argument as both an index and a value. The class IndexedContainer provides a 2-place operation for 'add' that allows the index to be different from the value. Note that 'remove/1' expects to be supplied with the index.

InterpretedOperation

An interpreted operation is created when we evaluate an operation definition.

Namespace

A name space is a container of named elements. A name space defines two operations getElement/1 and hasElement/1 that are used to get an element by name and check for an element by name. Typically a name space will contain different categories of elements in which case the name space will place the contained elements in its contents table and in a type specific collection. For example, a class is a container for operations, attributes and constraints. Each of these elements are placed in the contents table for the class and in a slot containing a collection with the names operations, attributes; and constraints respectively. The special syntax :: is used to invoke the getElement/1 operation on a name space.

NamedElement

A named element is an owned element with a name. The name may be a string or a symbol. typically we use symbols where the lookup of the name needs to be efficient.

Object

Object is the super-class of all classes with structural features in XMF. Object provides access to slots via the get/1 and set/2 operations. Object is the default super-class for a class definition - if you do not specify a super-class then Object is assumed.

Operation

The basis for all XMF execution is the XCore class Operation. An operation has parameters and a body and is equivalent to a standard programming language procedure or function. A significant difference to conventional procedures is that operations are XMF objects that can be created and stored just like any other object. This makes XMF very flexible since behaviour can be encapsulated at the appropriate point in models and data.

OrderedCollection

A container that wraps a collections of ordered elements.

Package

XMF supports name spaces that contain collections of named elements. The XCore class Package is used to structure collections of class and sub-package definitions. XMF-Mosaic is structured as a tree of packages containing definitions of all aspects of the system (including XCore). The root name space is called Root; all XMF classes can be referenced via Root. Unlike UML and MOF, XCore packages are subclasses of Class. They can therefore be instantiated and can have operations, attributes and constraints.

Parameter

A parameter is a typed element that occurs in operations.

Performable

XMF provides an environment in which executable languages can be conveniently developed. An executable language implements the interface defined by the abstract XCore class Performable. XOCL is an example of a language that implements this interface.

XCore is an example of a language that can be performed by XVM. XVM may be initialised with different kernel language definitions although in practice, XVM relies on a small sub-set of XCore being present. This feature allows XMF to deploy embedded systems that run application code without requiring the tools that were used in XMF-Mosaic to develop the application.

Resource

A resource records where the resource originated via a resource name. For example a definition is a resource that records the file where it was loaded from.

Seq

Seq is a sub-class of DataType. All sequence data types are an instance of Seq. Seq defines an attribute elementType that is used to record the type of the elements in a sequence data type.

Set

Set is a sub-class of DataType. All set data types are an instance of Set. Set defines an attribute elementType that is used to record the type of the elements in a set data type.

Snapshot

A snapshot is collection of objects. A snapshot is an instanceOf another element (typically a package), which contains elements which classify the contents of the snapshot.

StructuralFeature

This is an abstract class that is the super-class of all classes that describe structural features. For example Attribute is a sub-class of StructuralFeature. Other types of structural feature are possible by managing the internal structure of objects via a MOP.

Table

A table associates keys with values. Any element can be used as a key. A table has an initial size and can support any number of values. Use hasKey/1 to determine whether a table contains a key. Use get/1 to access a table via a key and put/2 to update a table given a key and a value. Use keys/0 to access the set of keys for a table.

Thread

A thread is a unit of concurrent execution. When a thread is created, it continues processing on the XVM until either it performs a read operation that blocks on input or when it explicitly calls yield. All XOCL values implement the yield operation. In both cases the thread is said to yield control. When a thread yields control, the XOS schedules another thread that is waiting. The scheduling algorithm aims to ensure that all waiting threads get scheduled providing that they yield.

TypedElement

A typed element is a named element with an associated type. The type is a classifier. This is an abstract class and is used (for example) to define Attribute.

Unordered Collection

A container that wraps a collections of unordered elements.

Vector

A vector is a fixed length array of elements. They are created using the constructor Vector(). Vectors provide very efficient insert (put/2) and lookup operations (ref/1).

Chapter 19. XMap

Introduction

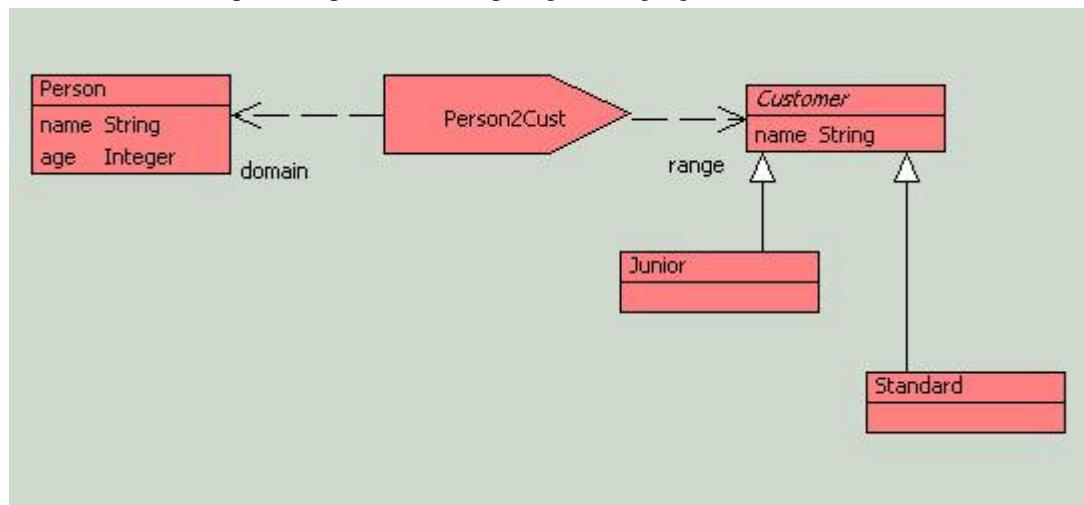
This document provides a definitive guide to the XMap language. XMap is a language for constructing mappings between models. Using XMap you can define mappings in a highly declarative way, enabling the essence of what the mapping does to be captured succinctly and precisely (as opposed to having to describe how the mapping is performed).

Here are some other benefits of XMap:

- It enables system designers to realise a strong separation of concerns between mappings and the models that they relate. Aspects such as pretty printing do not need to be mixed with the detail of the mapping itself.
- XMap can also make use of XOCL expressions to provide the extra expressiveness required to capture complex mappings
- The XMap language is fully modelled. This means that mappings expressed in XMap can be easily transformed into other data formats.

Language Basics

XMap provides both a textual and diagrammatical language for expressing mappings. The model below contains a simple example of the XMap diagram language:



The example contains a mapping called Person2Cust, which takes a Person object and generates a Customer object.

Mappings contain rules, known as clauses. These state how one or more objects (the domain) are mapped to a target object (the range). Here is an example of a clause belonging to the Person2Cust mapping:

```

self      Child
of       Clause
owner    Person2Cust
source

@Clause Child
  Person[name = N, age = A]
  when A < 16
do
  Junior[name = N]
end

```

A clause has a name, followed by a pattern declaration, a when expression and a do action. Provided that the value supplied to the clause matches the pattern declaration, and the when expression is true, then the do action will execute. In this example, provided that a Person object is supplied whose age is < 16, a Junior customer object will be created with the same name as the Person.

A mapping may contain more than one clause. In this example, a clause named Adult is also defined. It maps a Person who is older than 16 to a Standard customer object.

```

self      Adult
of       Clause
owner    Person2Cust
source

@Clause Adult
  Person[name = N, age = A]
  when A >= 16
do
  Standard[name = N]
end

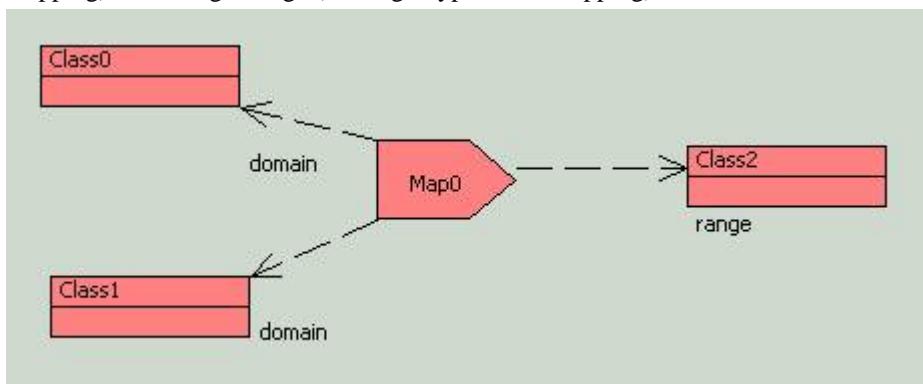
```

The rest of this document will dig deeper into the XMap language, starting with its syntax.

Syntax

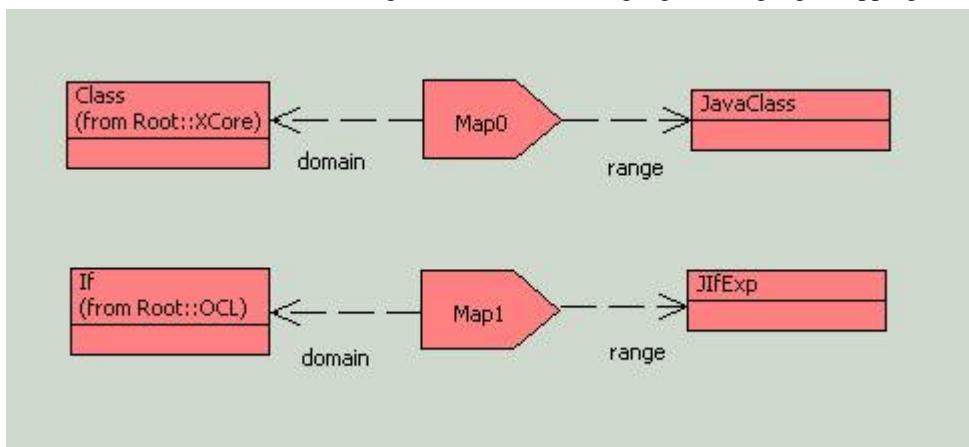
Diagrammatical Syntax

A mapping is diagrammatically shown as an arrow with one or more domains (the source types of the mapping) and a single range (the target type of the mapping):



The domain and range of a mapping can be any (instantiable) Classifier type, including data types, packages and other mappings.

Because XMF-Mosaic provides full access to its own definition, mappings can be defined between meta-classes. This is crucial to being able to construct language-to-language mappings.



Textual Syntax

A mapping also has a textual syntax of the form:

```

@Map <MapName>(<DomainType>, ...) -> <RangeType>

@Clause <ClauseName>
end
...
end

```

Clause Syntax

A Clause has the following syntax:

```

@Clause <ClauseName>
<V1> = <PatternDecl>,
...
when <BooleanExpression>
do <XOCLExpression>
where <X1> = <XOCLExpression>;
...
end

```

A clause consists of a comma-separated list of pattern declarations, which may optionally be equated with a variable. This is followed by an optional when part, containing a Boolean expression, and a do action, which contains an XOCL expression. The optional where clause contains a list of semicolon separated variable introductions, where each variable is equated with an XOCL expression.

Pattern Syntax

Full details of the pattern declaration syntax can be found in the XOCL manual, but there are some specific forms that are most generally used in XMap:

- A keyword constructor: <ClassName>[<att1> = <x1>, ...]

Here, the variable/s att refer to named attributes of the class, and x is the variable/s they are matched against.

- A normal constructor : <ClassName>(<v1>, ...)

Where v1 is a constructor variable.

- A set pattern: <X>->including(<PatternDecl>)

Where X represents a collection of values, which include an element that matches the pattern declaration.

Execution

XMap is fully executable. The execution of an XMap mapping is as follows:

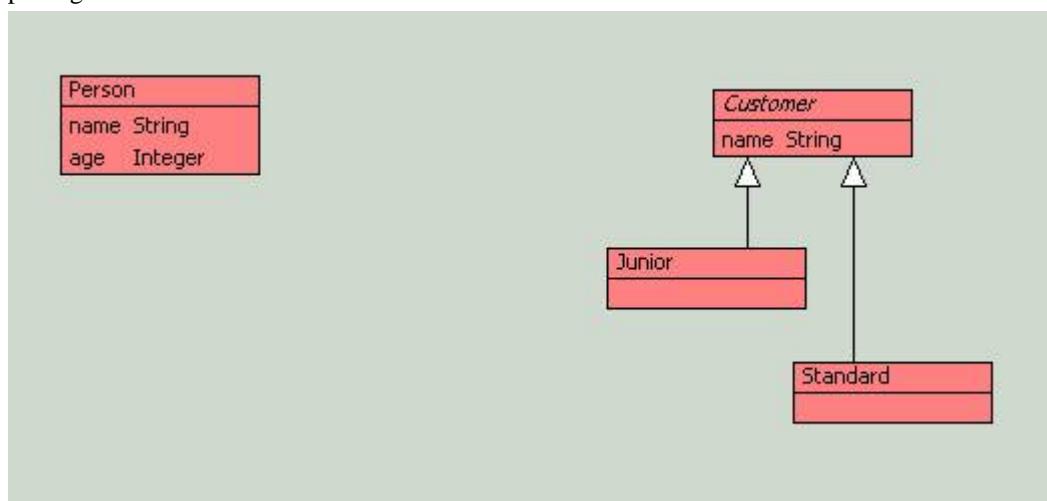
- Iterate through each clause, in order of definition.
- Attempt to match the input values of the mapping with a clause. A clause matches if:
 - The input to the clause matches the pattern declarations.
 - The when condition is true.
- If a clause matches, the body of the do action is executed (substituting any variables values bound in the pattern declarations and where part). The mapping terminates.
- If no clauses match the input values, the mapping terminates and generates an exception.

Constructing Mappings

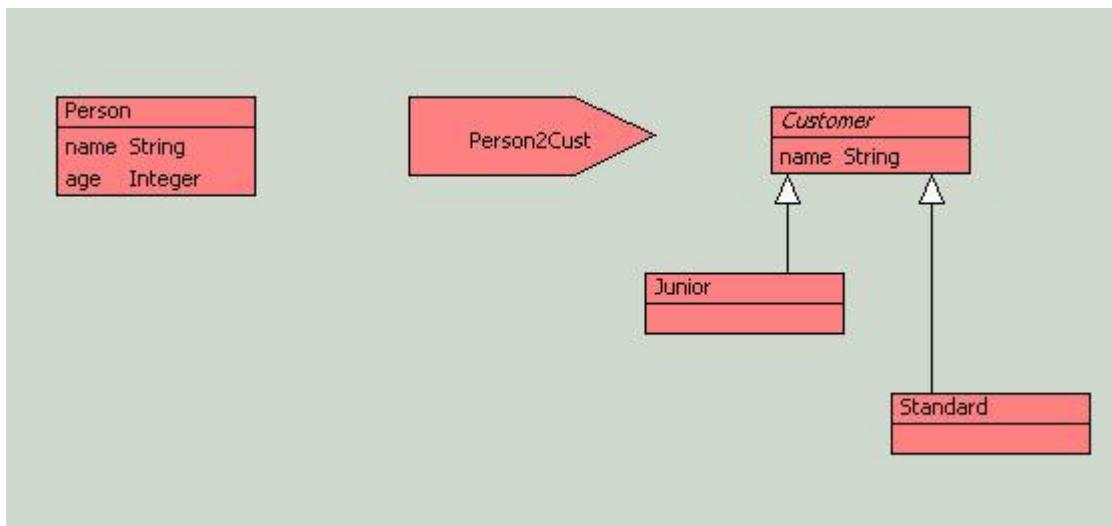
Mappings can be constructed using XMF-Mosaic's modelling interface and via the programming interface as follows.

Creating Mappings via the Modelling Interface

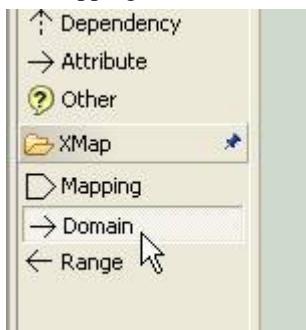
Let's create a simple mapping from one class to another class. First, create a project Example1 containing the classes that we want to map between: Alternatively, we could import classes from other packages into the model.



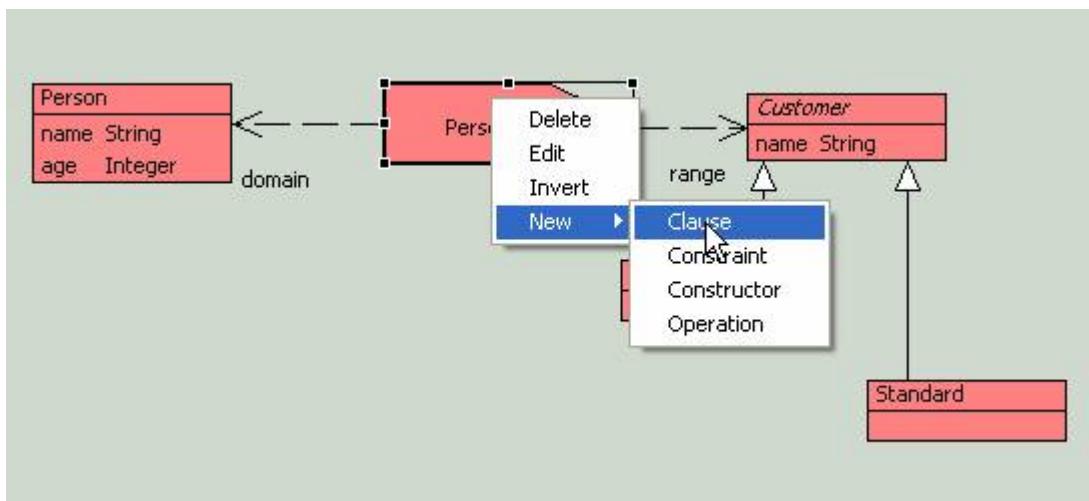
Next, a mapping is added to the model. Select the Mapping from the icons in the diagram editor and place it on the diagram and give it a name.



A mapping must have at least one domain and a single range. Select these from the icons underneath the mapping icon.



The domain is the source class or classes of the mapping and the range is the target class. Although compulsory, it is important to think of the domain and range as indicating a dependency between a mapping and its constituents. In practice, the actual values passed to the mapping need not necessarily be instances of the domain class.



To add a clause to a mapping, right click on the mapping in the diagram editor or browser and choosing New > Clause.

Now, edit the clause by double clicking on it in the browser.

A clause editor will be displayed.

Forms View

Clause0

self	Clause0
of	Clause
owner	Person2Cust
source	<pre>@Clause Clause0 null do null end</pre>

To enter the clause, enter the text and right click > Commit.

Child

self	Child
of	Clause
owner	Person2Cust
source	<pre>@Clause Child Person[name = name, age = A] when A < 16 do Junior[name = name] end</pre>

Other clauses can be created in exactly the same way.

Forms View

Child Adult

self	Adult
of	Clause
owner	Person2Cust
source	<pre>@Clause Adult Person[name = name, age = A] when A >= 16 do Standard[name = name] end</pre>

Creating Mappings via the Programming Interface

Creating a mapping via the programming interface is straightforward. Just create a file (select File Browser from the File menu) and create a new File. Provided that XOCL has been imported, a model (along with mappings) can be entered as normal. For example, the following text is equivalent to the above model:

```
parserImport XOCL;

context Root

@Package Example1

@class Person
  @Attribute name : String end
  @Attribute age : Integer end
end

@class Customer
  @Attribute name : String end
end

@class Junior extends Customer
end

@class Standard extends Customer
end

@Map Person2Cust(Person) -> Customer

@Clause Child
  Person[name = N, age = A]
  when A < 16
  do
    Junior[name = N]
  end
@Clause Parent
  Person[name = N, age = A]
  when A >= 16
  do
    Standard[name = N]
  end
end
end
```

Save, compile and load the file to compile and load the model and the mapping.

Running Mappings

Mappings are run by instantiating the mapping and passing it the appropriate object/s to be mapped. The syntax of a call is as follows:

```
<Mapping>()(<Value>)
```

Note that if there is more than one domain, this must be reflected by the list of values passed to the mapping, e.g.

```
<Mapping>()(<Value1>, <Value2>, ..., ValueN)
```

The following example shows a mapping being executed in the console. Of course, a mapping can be run by calling it from within another mapping or operation.

```
[1] XMF> p := Example1::Person[name = "Bob", age = 15];
<NameSpace Root>
[1] XMF> c := Example1::Person2Cust()(p);
<NameSpace Root>
[1] XMF> c.name;
Bob
[1] XMF> c;
<a Junior>
[1] XMF> |
```

First, an instance of a person, p, whose name is “Bob” and whose age is 15 is created.

The mapping is run by calling the mapping and passing it the person, p.

The result of the mapping is a Junior customer whose name is “Bob” as we would expect.

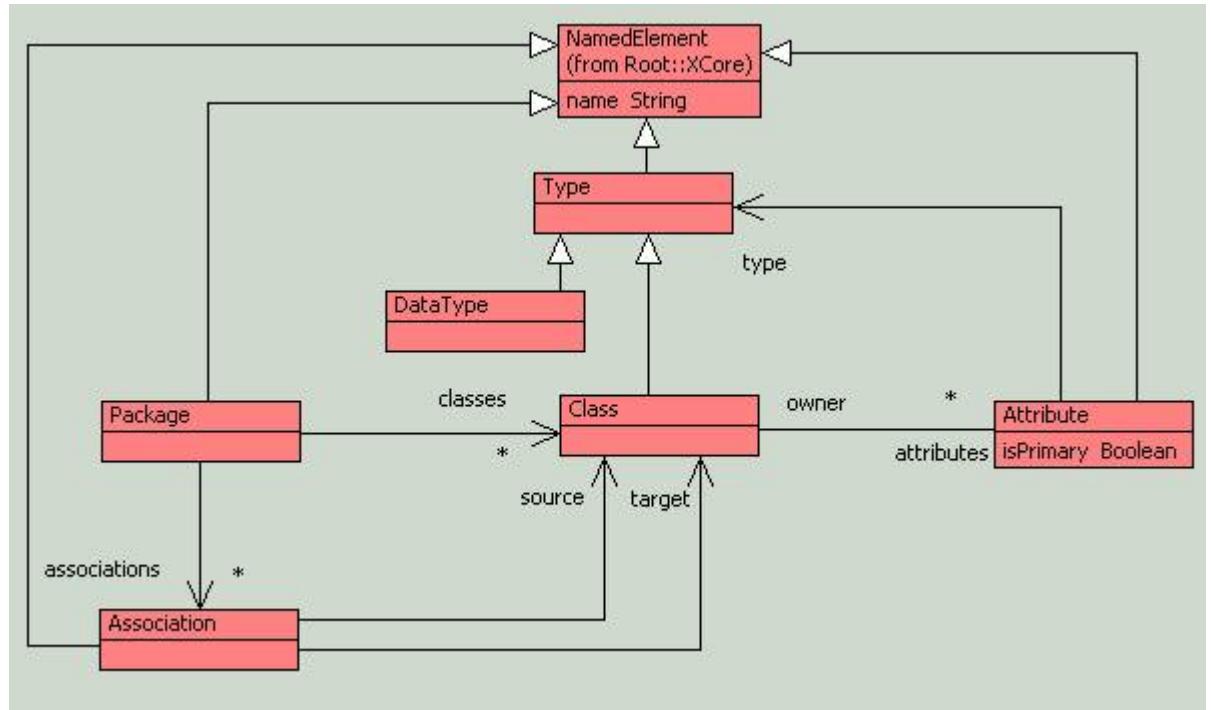
Example

Class Model to Database

This example defines a mapping between a class model and a model of a database. In doing so, it exercises a number of key XMap features, including pattern nesting, variable passing and multiple domain inputs to mappings.

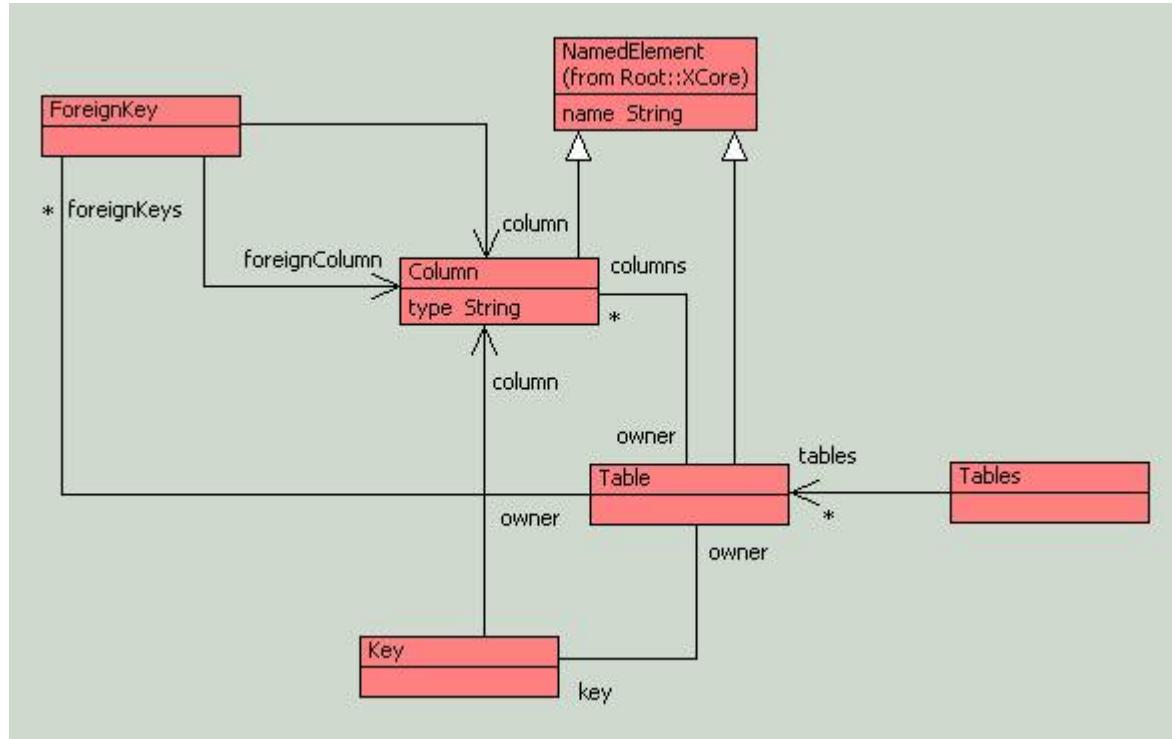
Class Model

The following model describes the concepts in the class model. It includes most of the concepts that are typically found in a class model, including classes, packages, associations and generalization.

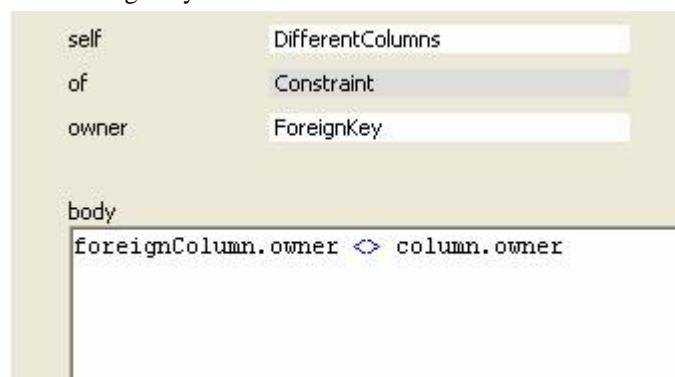


Database Model

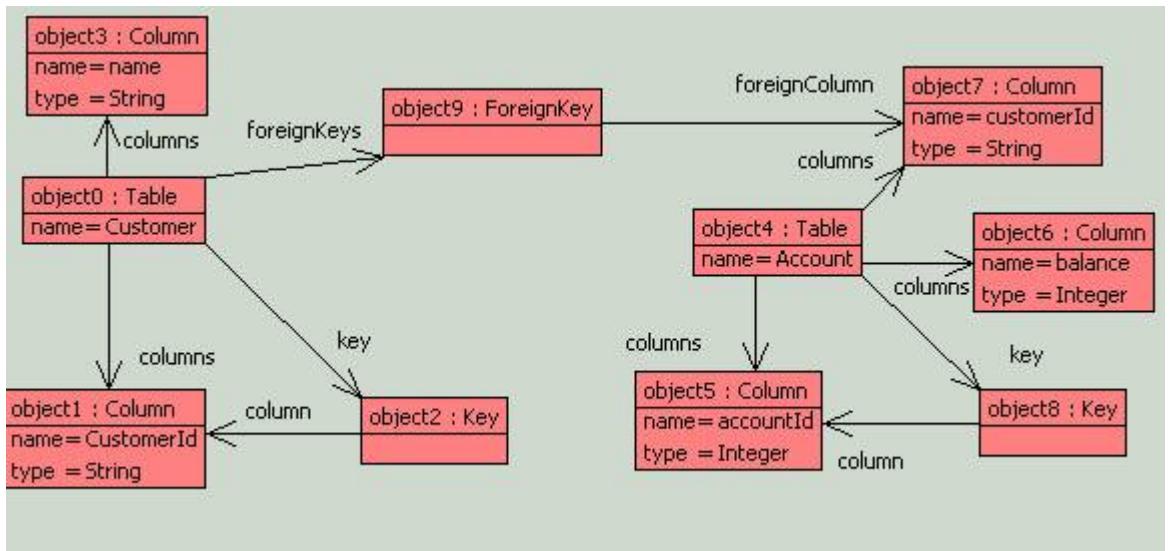
The database model contains a collection of typical relational database concepts. A relational database contains a collection of tables, which have names. Each table contains a collection of columns, which are named. A Column has a type. A table always contains a Key, which is a pointer to the Column that uniquely identifies an instance of a Table. A table may also contain ForeignKey's. These relate a Column to the Column in another table that it is the foreign key for.



There will be a significant number of constraints on this model. For example, it must never be the case that a foreign key refers to a column in the same table as the foreign key column:



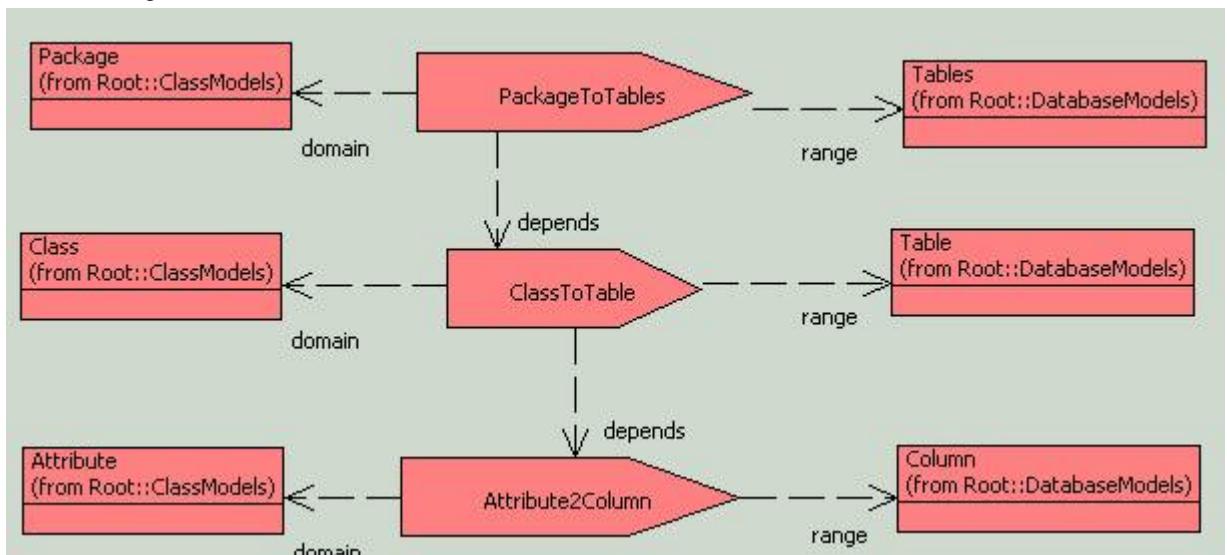
As an example of a database model that could be represented by this model, consider the tables Account and Customer. The table Account has three columns, customerId, accountNo and balance. The column customerId is a foreign key – it ties each instance of an account to a specific customer. The column accountNo is a key, as it uniquely identifies the account. The column ‘balance’ just represents some information about the account. Similarly, the table customer has three columns, one of which, customerId, is its key.



The following diagram shows the same information as it would be represented by an instance of the Database model:

Database Mapping

The following model describes how class models are transformed into database models:



A package is mapped to an instance of Tables. This contains the results of mapping each Class in the package into a Table. Each Table contains a collection of columns, each of which is the result of mapping the attributes of the Class into a Column. In the case of an Attribute that is marked as a key, a Key is created which points to the corresponding Column.

Let's take each mapping in turn and explain how it works:

PackageToTables

The PackageToTables mapping contains the following clause:

```

self          Clause0
of            Clause
owner        PackageToTables
source
@Clause Clause0
  ClassModels::Package[
    classes = C,
    associations = A]
do
  DatabaseModels::Tables[
    tables = T]
where
  T = C->collect(c |
    ClassToTable()(c))
end

```

Here, the input to the mapping requires a Package containing some classes C and some associations A. If there is a match then the action after the ‘do’ part is executed. In this case, it creates an instance of the class Tables, and sets its tables to be T. T is defined in the ‘where’ part. T is calculated as the result of mapping each class in C to a table.

Note Associations will be dealt with in future versions of this example.

ClassToTable

The first part of the ClassToTable mapping is as follows:

```

@Clause Clause0
  ClassModels::Class[
    name = className,
    attributes =
      A->including(
        ClassModels::Attribute[
          name = attName,
          isPrimary = true,
          type =
            ClassModels::DataType[
              name = typeName
            ]
        ]
      )
    ]

```

This pattern expects a Class, and matches the name of the Class with the variable className.

In addition, this pattern illustrates the use of a collection pattern to match with an element in a collection. In this case, it matches if the attributes of the class bind with a collection of attributes A provided that it includes a primary attribute. This matches provided that the attribute is a primary Attribute, and binds the name of the attribute to the variable attName.

A similar type of pattern is used to return the name of the type of the attribute. Here the attribute type is matched against a DataType, whose name is then bound to the variable typeName.

Provided there is a match, the following action is invoked:

```

do
  DatabaseModels::Table[
    name = className,

```

```
key = DatabaseModels::Key[
    column = c],
columns = C->including(c)].setOwnership()
where
    c = DatabaseModels::Column[
        name = attName,
        type = typeName];
    C = Attribute2Column()(A)
end
```

This creates an instance of a Table, whose name is the class name, and which has a Key that points to a Column that has the same name and type as the primary Attribute.

In addition, the columns of the Table are calculated by mapping each of the remaining attributes in A to columns and including the key column.

Finally, the operation setOwnership is called on the generated table. This sets the owner attribute of all the columns of the table, its foreign keys and key to be the table. It is defined as follows:

```
@Operation setOwnership():Element
@For c in columns do
    c.owner := self
end;
@For f in foreignKeys do
    f.owner := self
end;
key.owner := self;
self
end
```

The modified table is returned by the final self.

Attribute2Column

The attribute to column mapping takes a collection of attributes and maps them to columns. There are three clauses in this mapping.

The first one just returns the empty set if there are no attributes:

```
@Clause Clause1
Set{}
do
Set{}
end
```

The second clause matches if the attributes match the collection A that includes a attribute whose type is a datatype. If so, it creates a collection of columns, C, which includes a new column whose name and type is equal to the name and type of the attribute. However, C is calculated by passing the remaining attributes A to self, which in this case is the Attribute2Column mapping. Thus, it recurses through all the attributes applying the appropriate Attribute2Column clauses until they are consumed.

```
@Clause Clause0
A->including(ClassModels::Attribute[
    name = attName,
    type = ClassModels::DataType[
        name = typeName] ])
do
C->including(DatabaseModels::Column[
    name = attName,
    type = typeName])
```

```

where
  C = self(A)
end

```

The third clause performs a similar function to the above, but in this case it only matches if the type of the attribute is a class. In this case, it runs through the attributes in the type applying the Attribute2Column mapping to those elements and adding them to the collection. Once this is complete it modifies the resulting column names so that they reflect the name of the type.

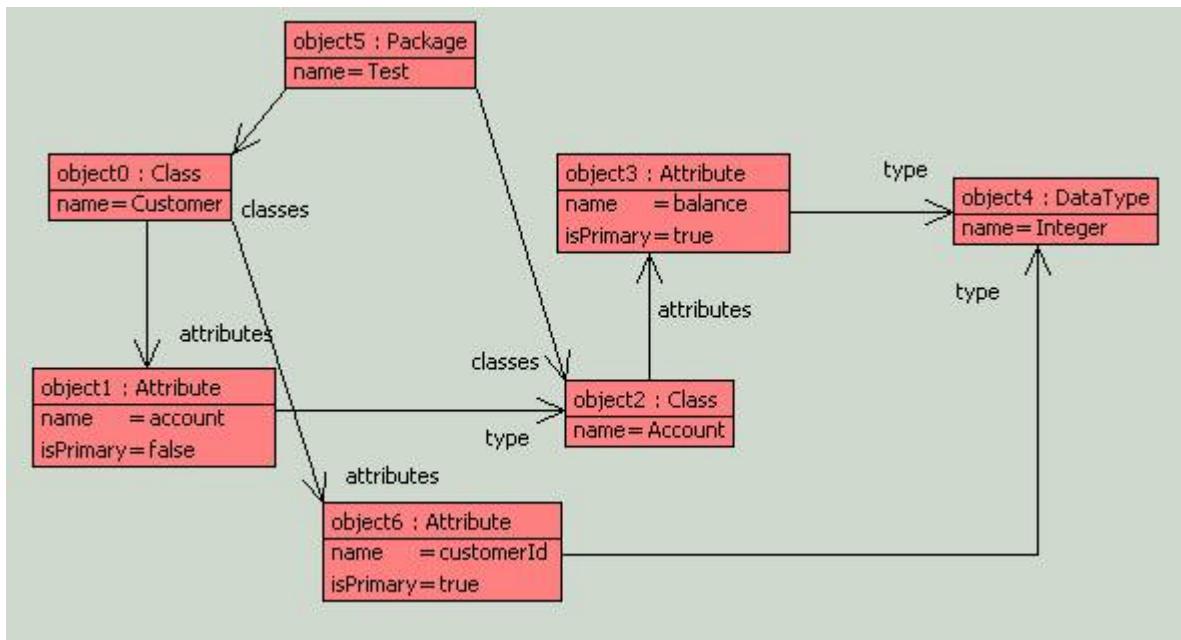
```

@Clause Clause2
A->including(ClassModels::Attribute[
  name = attName,
  type = class])
when class.isKindOf(ClassModels::Class)
do
  C->union(self(class.attributes)->collect(col |
    col.name := class.name + "_" + attName + "_" + col.name))
  where C = self(A)
end

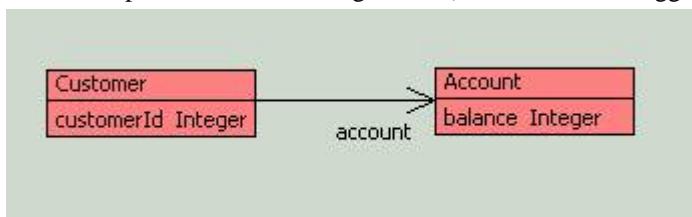
```

Running the Mapping

As an example, the following snapshot shows an instance of the class model.



This corresponds to the following model (with customerId tagged as a primary attribute):



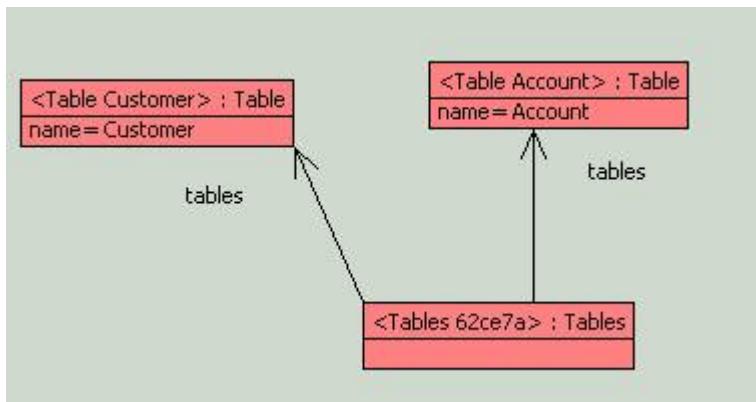
This can be run through the mapping as follows:

```

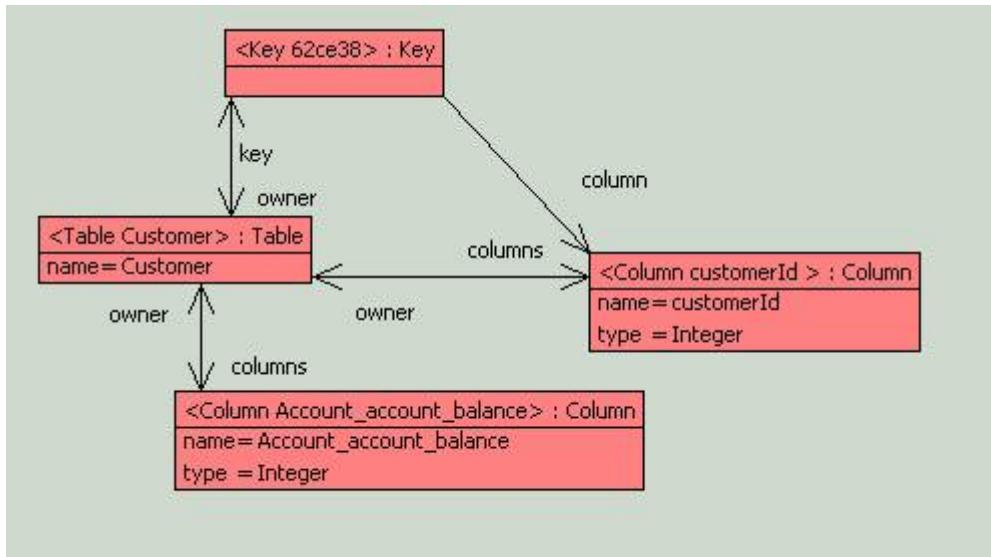
[1] XMF> DatabaseMapping::PackageToTables() (ClassModels::snapshot0::object5).edit()
<Tables dc6025>
[1] XMF> |

```

And returns the following snapshot (run toSnapshot() and then showDiagram()) on the result of the mapping:



Individual snapshots for specific objects can be shown, again by running toSnapshot() and showDiagram() on the object. For instance, here is the snapshot for the Customer table:



As expected, a new Key has been created for the customerId column, and a column resulting from turning the associated Account attributes into columns has also been created.

Other Aspects of Mappings

Operations

XMap mappings can have operations. As an example, let's add an operation to the Attribute2Column mapping that updates the name of a column.

An operation is added as usual by right clicking on the mapping (in the browser or the diagram) and selecting New > Operation. An operation is added to the mapping. Let's call this setColumnName():



The body of the operation is as follows:

```

self      setColumnName()
of       CompiledOperation
owner    Attribute2Column

source
@Operation setColumnName(col : XCore::Element,attName : XCore::Element,class :
    col.name := class.name + "_" + attName + "_" + col.name;
    col
end

```

The Attribute to Column mapping can be modified to use this operation:

```

self      Clause2
of       Clause
owner    Attribute2Column

source
@Clause Clause2
A->including(ClassModels::Attribute[
    name = attName,
    type = class])
when class.isKindOf(ClassModels::Class)
do
C->union(self(class.attributes)->collect(col |
    self.setColumnName(col,attName,class)))
where C = self(A)
end

```

Note self is required as it is referring to the instance of the mapping.

Attributes

Just like classes, mappings may have attributes. These are a useful way of recording information about a mapping. To add an attribute, right click on the mapping and select New > Attribute (for a primitive typed attribute) or draw an attribute line from the mapping to the class. The following example shows how an Integer attribute can be used to store a count of the classes that were mapped by a specific mapping:

```

@Clause Clause0
p = ClassModels::Package[
    classes = C,
    associations = A]
do
    self.count := classes.size;
    DatabaseModels::Tables[
        tables = T]
where

```

Variable Passing

Variables can be bound with patterns as follows:

```

@Clause Clause0
p = ClassModels::Package[
    classes = C,

```

```
    associations = A]
do
  DatabaseModels::Tables[
    tables = T]
where
  T = C->collect(c |
    ClassToTable()(p,c))
end
```

The variable p is bound to the Package that is passed to the mapping. This can be used to pass the package to any other part of the mapping. In this case, it is passed to the ClassToTable mapping as a second argument. As this example shows, variable passing is useful when values have to be passed through mappings. For instance, the following ClassToTable mapping could utilize the passed package to prefix the name of the generated Table: Note that the clause now expects two patterns as input, a Package and a Class, and that these must have a comma to separate them.

```
@Clause Clause0
  ClassModels::Package[name = N],
  ClassModels::Class[
    name = className,
    attributes =
    ...
  do
    DatabaseModels::Table[
      name = P.name + ":" + className,
      key = DatabaseModels::Key[
        column = c],
      columns = C->including(c)].setOwnership()
  where
    c = DatabaseModels::Column[
      name = attName,
      type = "Integer"];
    C = Attribute2Column()(A)
  end
```

Chapter 20. XML

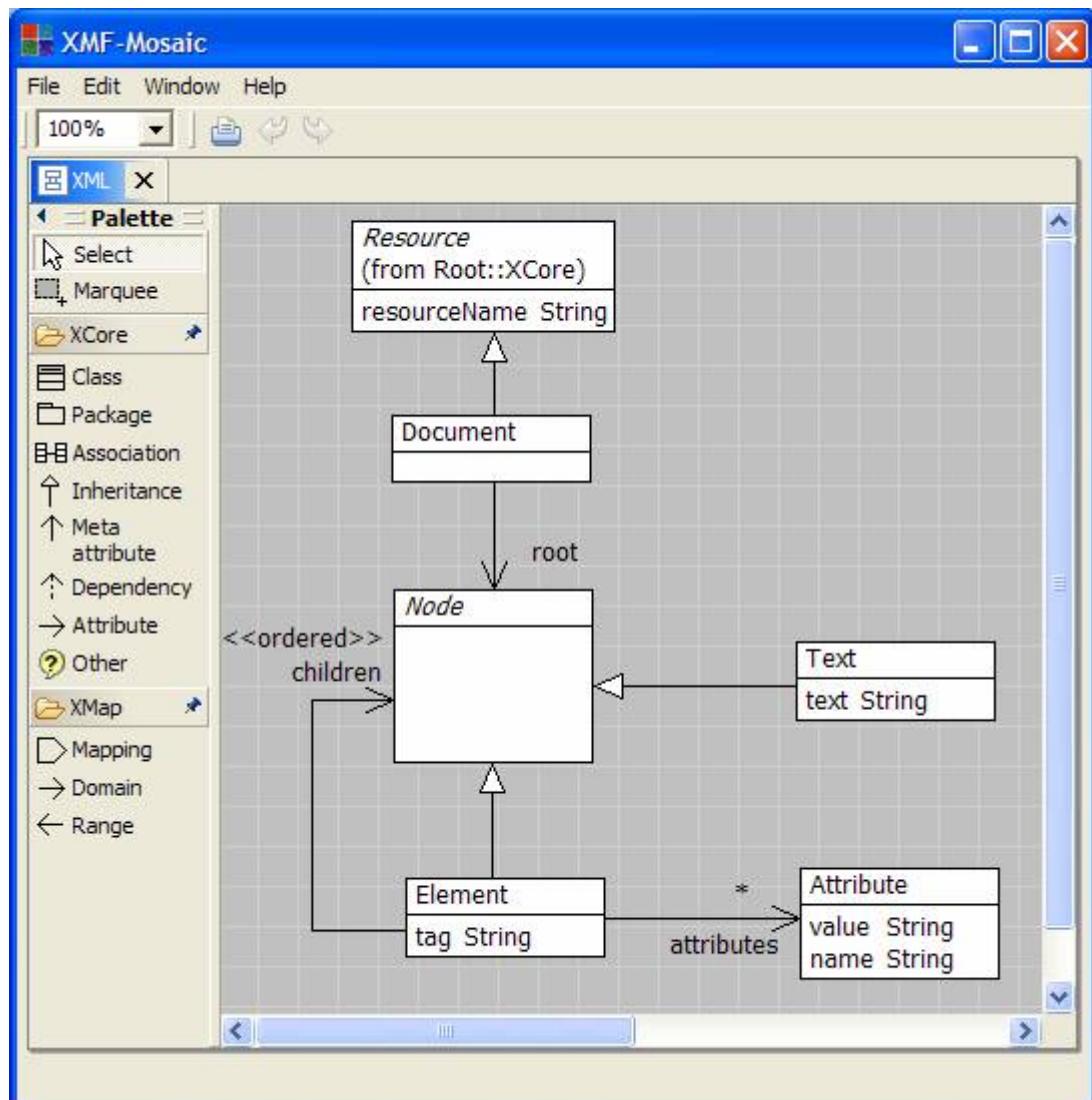
Introduction

XML is a standard data format and as such is an essential tool for saving XMF data and allowing XMF to interact with other tools. XMF provides extensive support for working with XML and in particular provides a declarative parser for reading sources of XML and synthesizing XMF data.

This document describes how to work with XML in XMF. A number of standard formats exist for XML (such as XMI for models). This document does not describe these formats.

XML

XML is a markup language for character based information. An XML document consists of a hierarchically organised collection of elements. Each element has a tag, some named attributes and a body. The body of an element is either text or is a sequence of child elements. The following model shows the basic structure of XML:



Parsing XML

Introduction

XMF provides a number of technologies for reading sources of XML. In most cases, an application is required to translate the XML input into XMF data elements. This can be achieved by reading the XML source character by character, but fortunately XMF provides XML parsing technology that interprets declarative rules specifying how to match XML input and synthesize XMF elements.

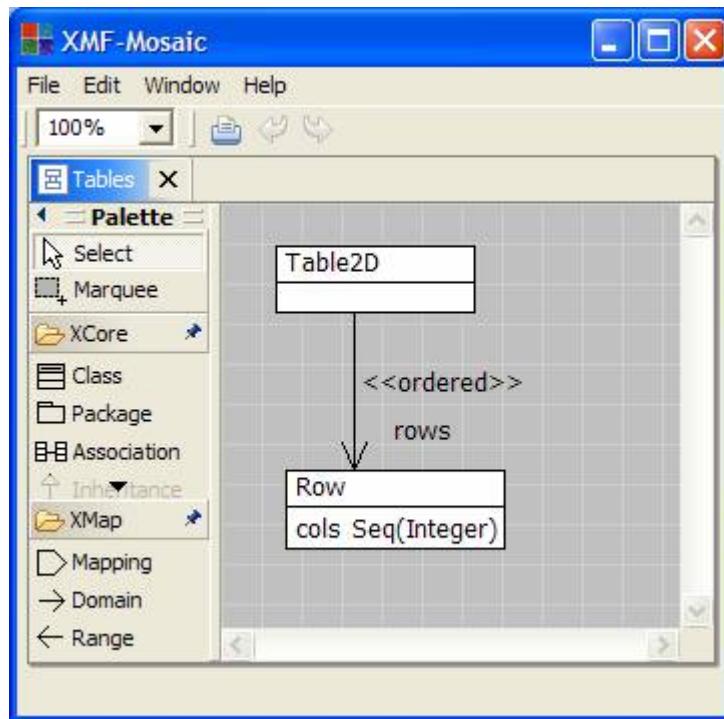
An XMF parser reads an XML input source and interprets an XML grammar. An XML grammar consists of a collection of clauses. Each clause defines how to match a portion of the input and how to translate it into XMF data. An XML grammar can be viewed as the type of an XML document in the same way as a DTD. Unlike DTDs, grammar components are associated with actions that are performed when the appropriate portion of the input has been successfully consumed.

This section shows how to define XML grammars and how to run XML parsers. It starts with a number of examples that contain key technology features. Finally the XML grammar model is defined in full with a complete definition of the components.

Example

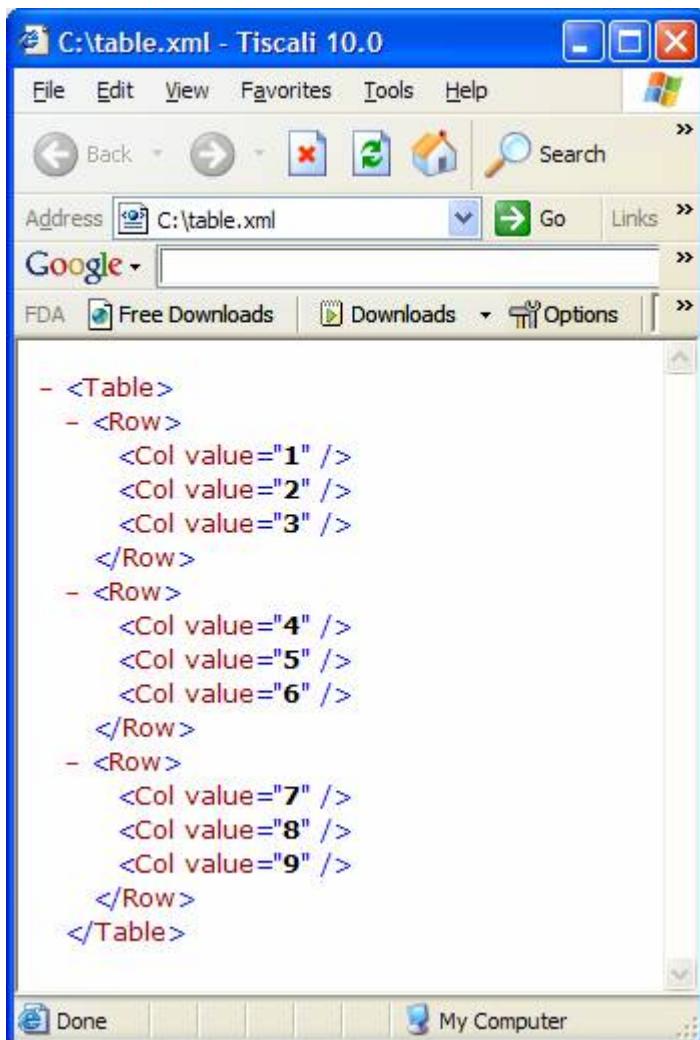
Two Dimensional Tables

This section gives an example of how to read the XML representation of a simple domain model and synthesize an instance of the model. The following model represents two dimensional integer tables:



Suppose that we have been supplied with an XML document that contains a two-dimensional integer table. We want to read the XML and produce an instance of the domain model. This situation occurs frequently in system development when we are supplied with data that has been exported from a third party tool and we want to process the information using XMF. Alternatively, it may occur when we have used XML as the format for storing data from a previous use of XMF-Mosaic.

The following screen shot shows how the data could be expressed in XML:



The XML is to be parsed using a grammar that defines how to match tables, rows and columns. An XML grammar consists of a collection of clauses each of which defines how to match elements, their attributes and their children.

An XML source (such as a file) is read and each element is matched against the clauses in the grammar. When a clause matches against the current XML input element, its action is performed causing an XMF element to be synthesized.

Matching an element involves matching its tag, attributes and its children. All of these components must match in order for the match to be successful. Attribute values can be extracted as part of the matching process and passed to the synthesizing actions. Values synthesized by matching children can be used in the actions of a parent. This way information can be extracted from the XML elements as they are consumed and synthesized elements can be passed up the tree as it is consumed. Eventually the result of the parse is the value synthesized by the root element of the XML document. The following tool screenshot shows the content of a source file that defines an XML grammar for parsing two-dimensional tables. The file imports the XML::Parser package in line 1 so that the grammar for parsing XML grammars is available in the rest of the file. Line 4 imports the same package so that names can be referenced in code (for example ParserChannel in line 40).

The screenshot shows the XMF-Mosaic application window. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for new, open, save, and close. The main area is a code editor with the file "Tables.xmf" open. The code is an XML grammar definition:

```

1 parserImport XML::Parser;
2 parserImport XOCL;
3
4 import XML::Parser;
5 import IO;
6
7 context Root
8   @Operation grammar()
9     // XML Grammars can be created anywhere. This example
10    // produces a grammar as the result of an operation...
11    @Grammar Table
12      // A table consists of a sequence of rows. The
13      // clause synthesizes an instance of the class
14      // Tables::Table2D...
15      Table ::= 
16        <Table>
17          rows = Row*
18        </Table> {
19          Tables::Table2D(rows)
20        }.
21      // A row is a sequence of columns...
22      Row ::= 
23        <Row>
24          cols = Col*
25        </Row> {
26          Tables::Row(cols)
27        }.
28      // A column is just an integer value...
29      Col ::= 
30        <Col value/> {
31          value.toInt()
32        }.
33    end
34  end

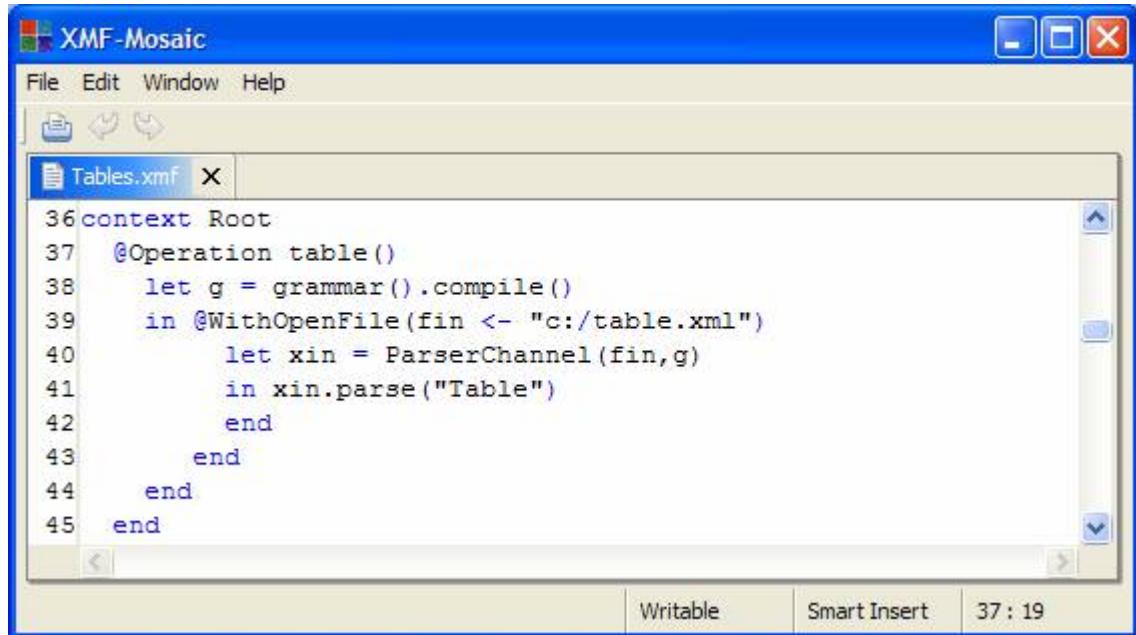
```

The status bar at the bottom shows "Writable", "Smart Insert", and "28 : 46".

In order to use an XML grammar to parse and synthesize an XML source, the grammar must be compiled. Compilation processes the grammar, checks it for any errors and then translates it into an efficient parsing table. The table is then used as the input to a general XML parsing engine. Compilation checks for a number of errors. In particular it checks that the grammar is LL(1). An LL(1) grammar allows the parser to proceed based on one token lookahead. In this case, tokens are XML elements. Fortunately, LL(1) grammars can be easily checked and this is done automatically for you. If you specify a non-LL(1) grammar, XMF will report a warning and indicate where you have gone wrong.

To use a compiled grammar, you create an instance of XML::Parser::ParserChannel. The constructor for this class accepts an XML grammar and an XML input source. A parser channel has an operation

parse that is used to consume input from the XML source, to run the parser and return the synthesized result:



```

36 context Root
37   @Operation table()
38     let g = grammar().compile()
39     in @WithOpenFile(fin <- "c:/table.xml")
40       let xin = ParserChannel(fin,g)
41         in xin.parse("Table")
42       end
43     end
44   end
45 end

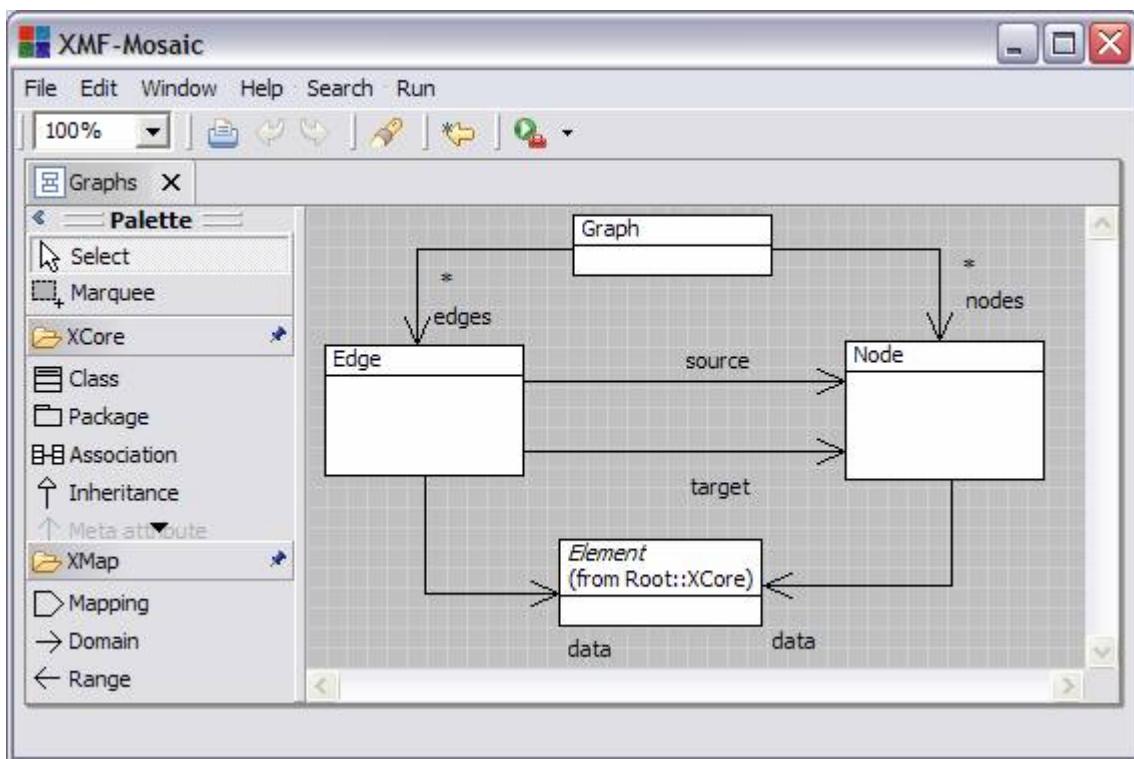
```

Dealing with References

XML data is tree-structured. This means that data elements cannot be shared by multiple paths from the root element. This is not true of XMF data elements, and computer data structures in general. General data elements are graphs where individual elements may be shared (or pointed to) by multiple parents.

This restriction on XML data representation is usually handled using data identifiers and references. Any element that can be referenced by multiple sources is allocated a unique identifier. One of the occurrences of the element is translated into an XML tree and all other occurrences are encoded in the reference source using the identifier. It is then up to the interpretation of the XML data to faithfully interpret the identifiers and their references. XMF manages identifiers and references automatically providing they are appropriately declared in the XML grammar. This section shows how this is achieved using a simple example of encoding graphs using XML.

The following model is used to represent graphs:

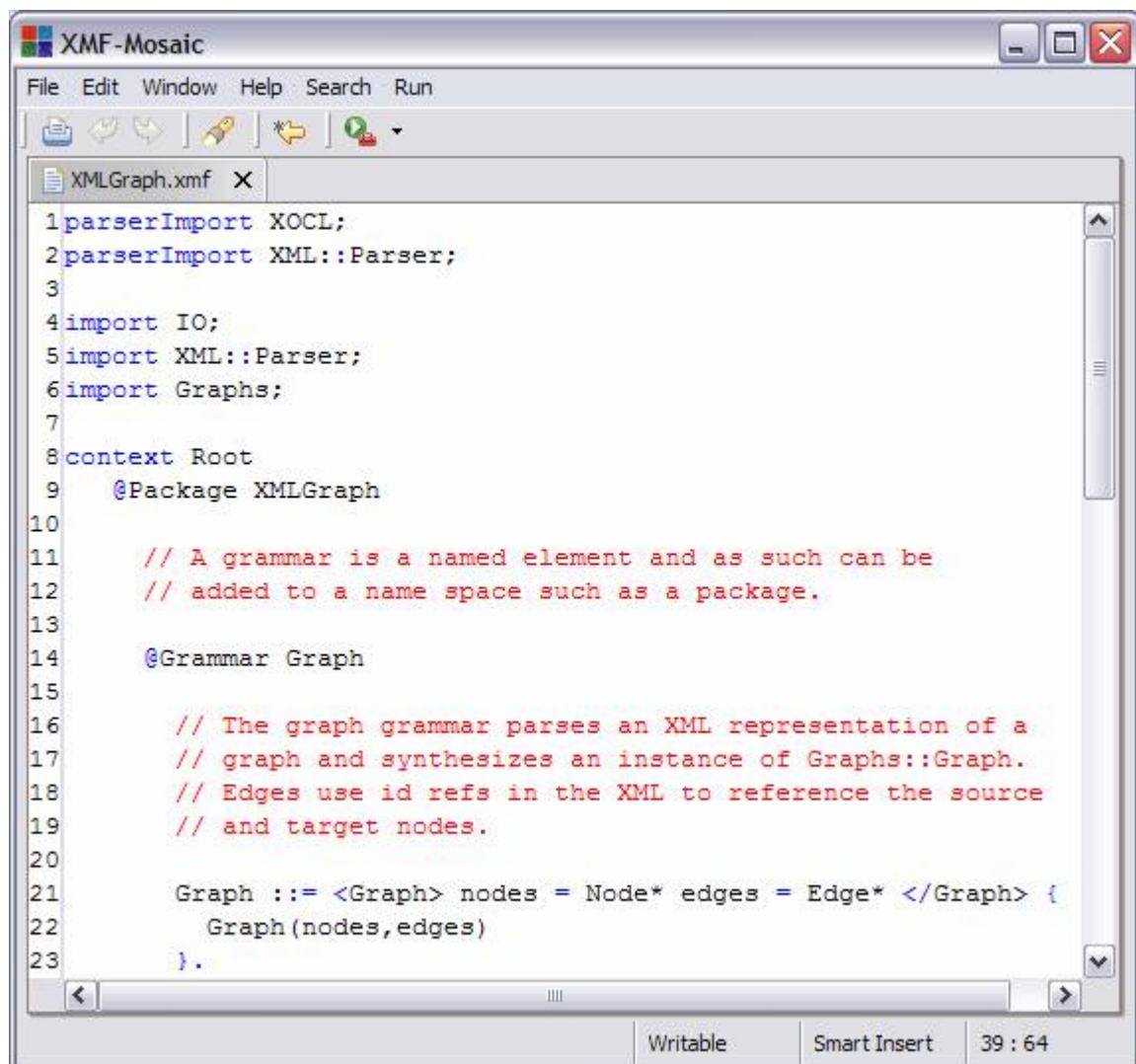


A graph can be represented in XML as follows. A node can be accessed by the containing graph through a number of routes: via the nodes attribute or via the source or target attributes of edges. Sharing of a node through multiple routes is encoded in the XML by allocating a unique identifier to each node. Nodes are represented using an XML element with the unique tag within the graph element and referenced using their identifier via edges:

```

<?xml version="1.0" encoding="UTF-8"?>
<Graph>
    <Node data="node1" id="1" />
    <Node data="node2" id="2" />
    <Node data="node3" id="3" />
    <Node data="node4" id="4" />
    <Edge data="edge1" source="1" target="2" />
    <Edge data="edge2" source="2" target="1" />
    <Edge data="edge3" source="1" target="3" />
    <Edge data="edge4" source="3" target="1" />
    <Edge data="edge5" source="1" target="4" />
    <Edge data="edge6" source="4" target="1" />
    <Edge data="edge7" source="1" target="1" />
</Graph>
  
```

We will synthesize an instance of the graph model using an XML grammar. The grammar is defined as a named element in a package as shown below:



The screenshot shows the XMF-Mosaic IDE interface with the title bar "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations. The main window displays an XML grammar file named "XMLGraph.xmf". The code is as follows:

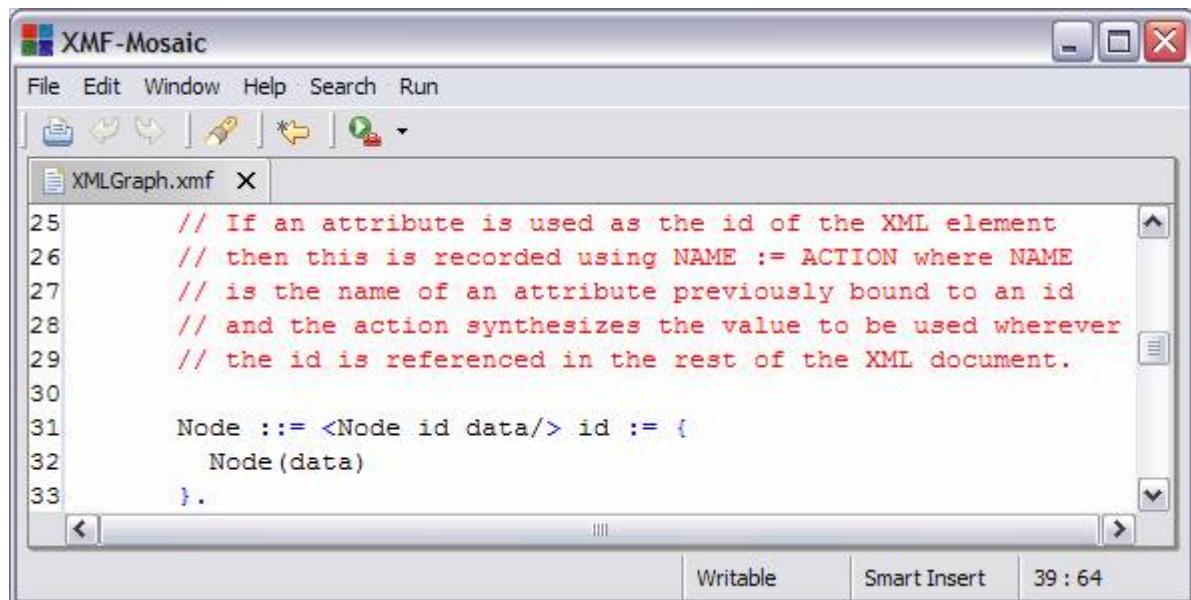
```

1 parserImport XOCL;
2 parserImport XML::Parser;
3
4 import IO;
5 import XML::Parser;
6 import Graphs;
7
8 context Root
9   @Package XMLGraph
10
11   // A grammar is a named element and as such can be
12   // added to a name space such as a package.
13
14   @Grammar Graph
15
16   // The graph grammar parses an XML representation of a
17   // graph and synthesizes an instance of Graphs::Graph.
18   // Edges use id refs in the XML to reference the source
19   // and target nodes.
20
21   Graph ::= <Graph> nodes = Node* edges = Edge* </Graph> {
22     Graph(nodes,edges)
23   }.

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "39 : 64".

An XML element representing a node is parsed and an instance of the class Graph::Node is synthesized by the clause defined below. A node has attributes id and data which are matched and bound to clause variables in line 31. The clause synthesizes a node which is associated with the identifier bound to the variable id:



The screenshot shows the XMF-Mosaic IDE interface with the title bar "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations. The main window displays an XML grammar file named "XMLGraph.xmf". The code continues from the previous snippet:

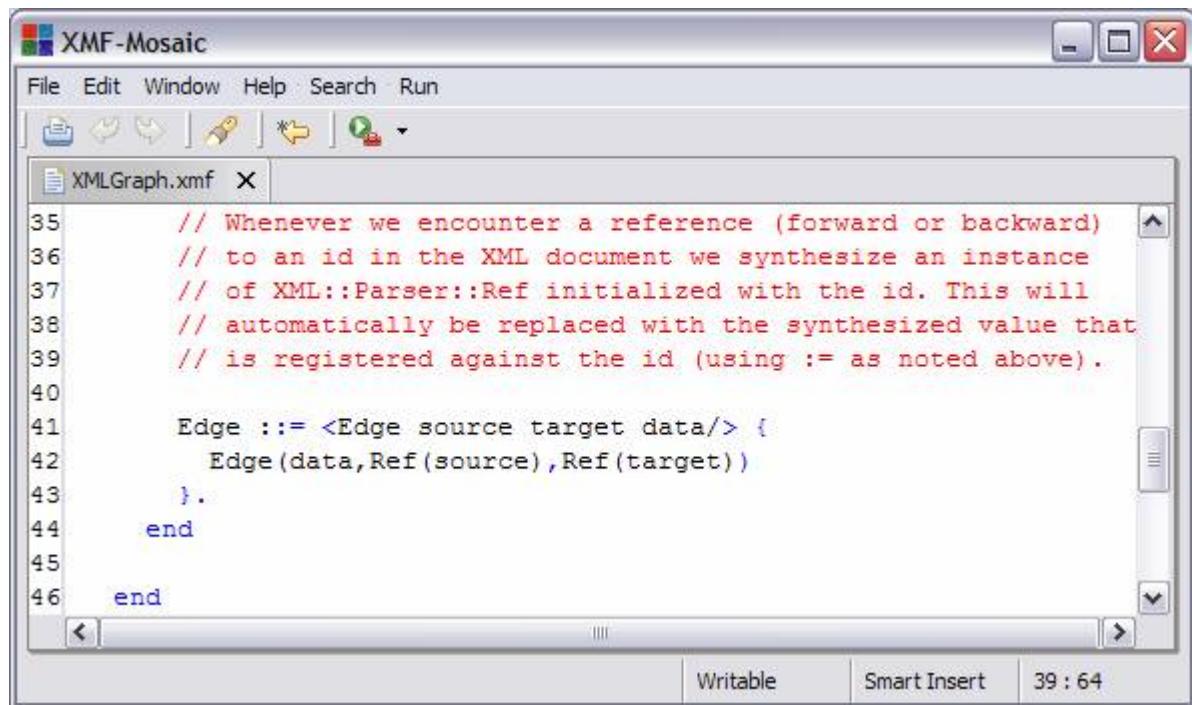
```

25   // If an attribute is used as the id of the XML element
26   // then this is recorded using NAME := ACTION where NAME
27   // is the name of an attribute previously bound to an id
28   // and the action synthesizes the value to be used wherever
29   // the id is referenced in the rest of the XML document.
30
31   Node ::= <Node id data/> id := {
32     Node(data)
33   }.

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "39 : 64".

Edges are parsed and synthesized by the clause defined below. Instances of the class Graph::Edge refer to the graph nodes; but, nodes are not available at this point in the parse since the XML elements refer to the source and target nodes via their ids. An XML parser can indicate that an identifier reference should be replaced with the associated value that is synthesized elsewhere by including a reference in the result of the clause as shown below:



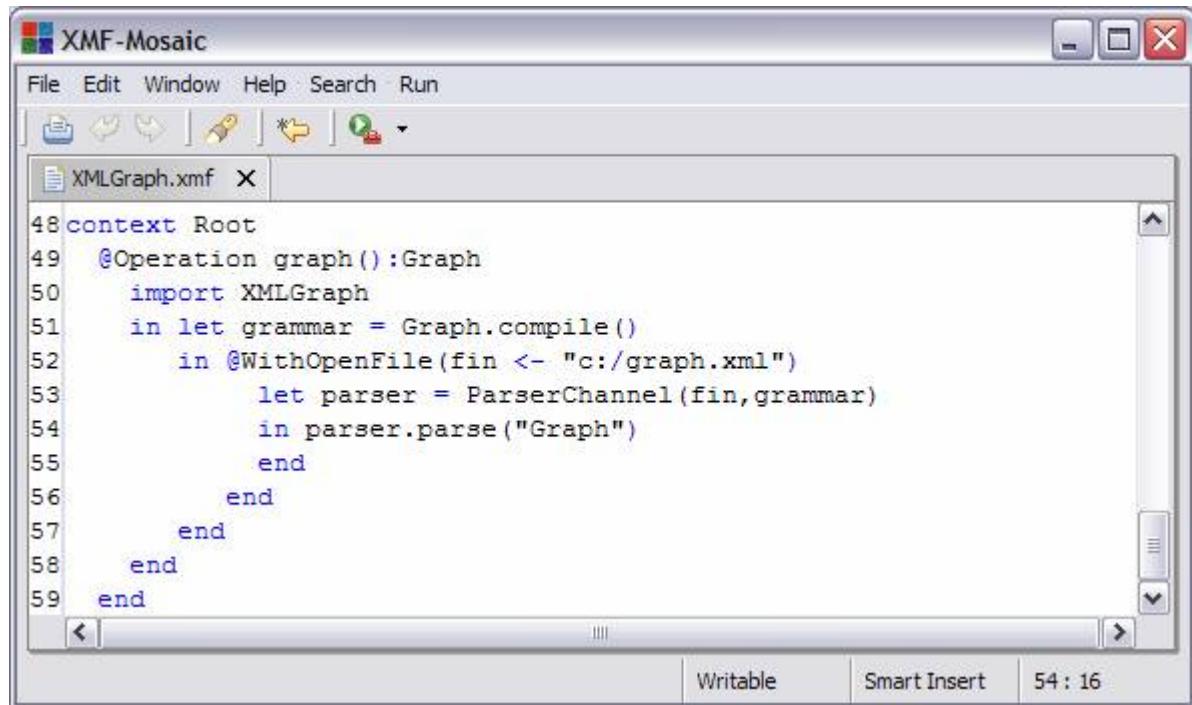
```

XMF-Mosaic
File Edit Window Help Search Run
XMLGraph.xmf X
35     // Whenever we encounter a reference (forward or backward)
36     // to an id in the XML document we synthesize an instance
37     // of XML::Parser::Ref initialized with the id. This will
38     // automatically be replaced with the synthesized value that
39     // is registered against the id (using := as noted above).
40
41     Edge ::= <Edge source target data/> {
42         Edge(data,Ref(source),Ref(target))
43     }.
44 end
45
46 end

```

The screenshot shows the XMF-Mosaic IDE interface with a file named "XMLGraph.xmf" open. The code editor displays a Graph grammar rule for "Edge". The rule uses a synthesized instance of "XML::Parser::Ref" initialized with an id from the XML document. The code is color-coded for syntax highlighting.

The following operation shows how the Graph grammar is used to parse a file containing a graph encoded in XML.



```

XMF-Mosaic
File Edit Window Help Search Run
XMLGraph.xmf X
48 context Root
49     @Operation graph():Graph
50     import XMLGraph
51     in let grammar = Graph.compile()
52         in @WithOpenFile(fin <- "c:/graph.xml")
53             let parser = ParserChannel(fin,grammar)
54                 in parser.parse("Graph")
55             end
56         end
57     end
58 end
59 end

```

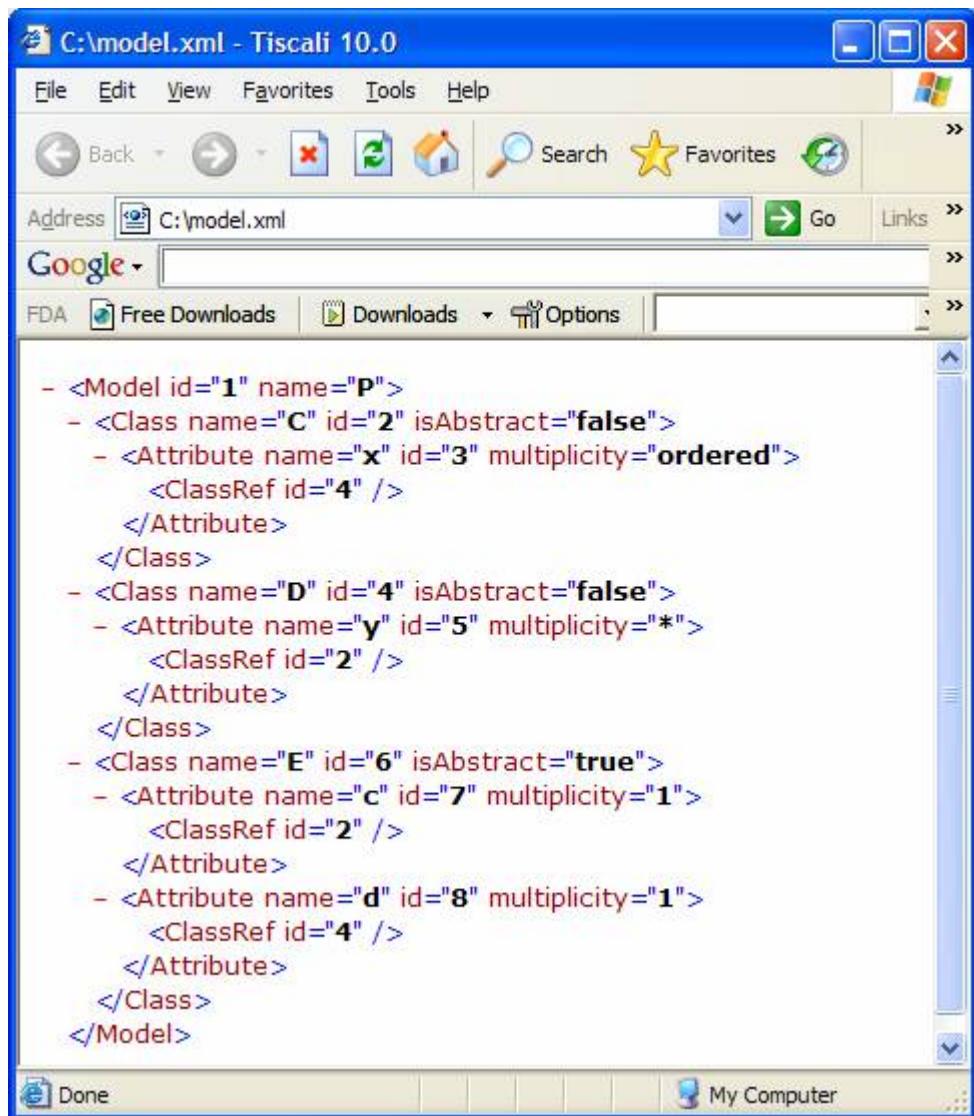
The screenshot shows the XMF-Mosaic IDE interface with the same "XMLGraph.xmf" file open. This time, the code editor displays a Graph grammar rule for the "context Root" block. It defines an operation "graph()" that imports "XMLGraph", compiles a grammar, and then parses an XML file named "graph.xml" using a "ParserChannel". The code is color-coded for syntax highlighting.

Representing Models in XML

Most modelling tools allow you to encode models using XML; often this is the only way of saving and loading models to persistent storage. The various versions of XMI for UML is an example of this kind

of model encoding. XMF-Mosaic allows models to be encoded in a variety of ways and in particular allows you to develop your own encoding that suits the domain with which you are modelling. This section provides an overview of how models can be encoded using an XMI-style XML format.

The following XML document shows a simple model encoding:



The screenshot shows a Windows-style application window titled "C:\model.xml - Tiscali 10.0". The window has a standard menu bar (File, Edit, View, Favorites, Tools, Help) and a toolbar with icons for Back, Forward, Stop, Refresh, Home, Search, and Favorites. Below the toolbar is an address bar showing "C:\model.xml". The main content area displays the XML code for a model. The XML structure includes a root element <Model id="1" name="P">, which contains three class definitions: <Class name="C" id="2" isAbstract="false">, <Class name="D" id="4" isAbstract="false">, and <Class name="E" id="6" isAbstract="true">. Each class has one attribute defined: <Attribute name="x" id="3" multiplicity="ordered"> with a reference to class <ClassRef id="4"/>; <Attribute name="y" id="5" multiplicity="*> with a reference to class <ClassRef id="2"/>; and <Attribute name="c" id="7" multiplicity="1"> with a reference to class <ClassRef id="2"/> and <Attribute name="d" id="8" multiplicity="1"> with a reference to class <ClassRef id="4"/>. The XML code is color-coded for readability. At the bottom of the window, there are buttons for Done and My Computer.

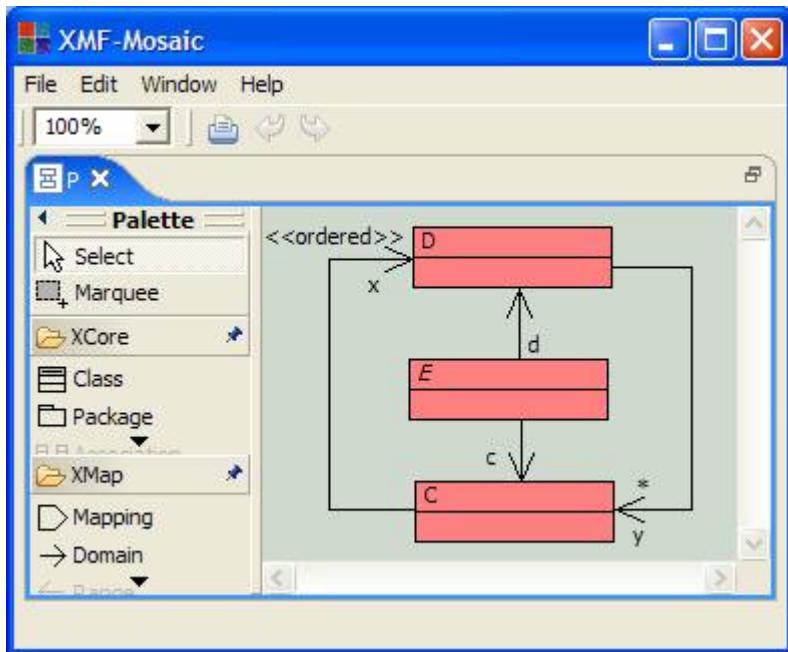
```

<- <Model id="1" name="P">
  - <Class name="C" id="2" isAbstract="false">
    - <Attribute name="x" id="3" multiplicity="ordered">
        <ClassRef id="4" />
      </Attribute>
    </Class>
  - <Class name="D" id="4" isAbstract="false">
    - <Attribute name="y" id="5" multiplicity="*>
        <ClassRef id="2" />
      </Attribute>
    </Class>
  - <Class name="E" id="6" isAbstract="true">
    - <Attribute name="c" id="7" multiplicity="1">
        <ClassRef id="2" />
      </Attribute>
    - <Attribute name="d" id="8" multiplicity="1">
        <ClassRef id="4" />
      </Attribute>
    </Class>
  </Model>

```

Models are just top-level packages containing standard elements such as classes and sub-packages. Each model element is allocated a unique identifier to facilitate cross model references. An example of a reference occurs in the attributes of the model where the type of the attribute is encoded as a reference to a class that is represented in full elsewhere in the document.

The result of loading the XML model is shown below:



The model is synthesized by parsing an XML grammar. The grammar is shown below. Each identifier is registered against the associated model element during the parse using :=:

The screenshot shows the XMF-Mosaic editor interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for new, open, save, and close. The main window displays an Xtext grammar file named "Models.xtext". The code is as follows:

```

parserImport Xocl;
parserImport XML::Parser;

import XML::Parser;

context Root

@Grammar Model

// A grammar that reads an XML representation of a model.
// Each model element is allocated a unique identifier
// to allow multiple occurrences of the element to be
// resolved via references.

Model ::=

    // A model is just an outermost package...
    <Model id name = name>
        elements = Element*
    </Model> id := {
        elements->iterate(e p = Package(name) | p.add(e))
    }.

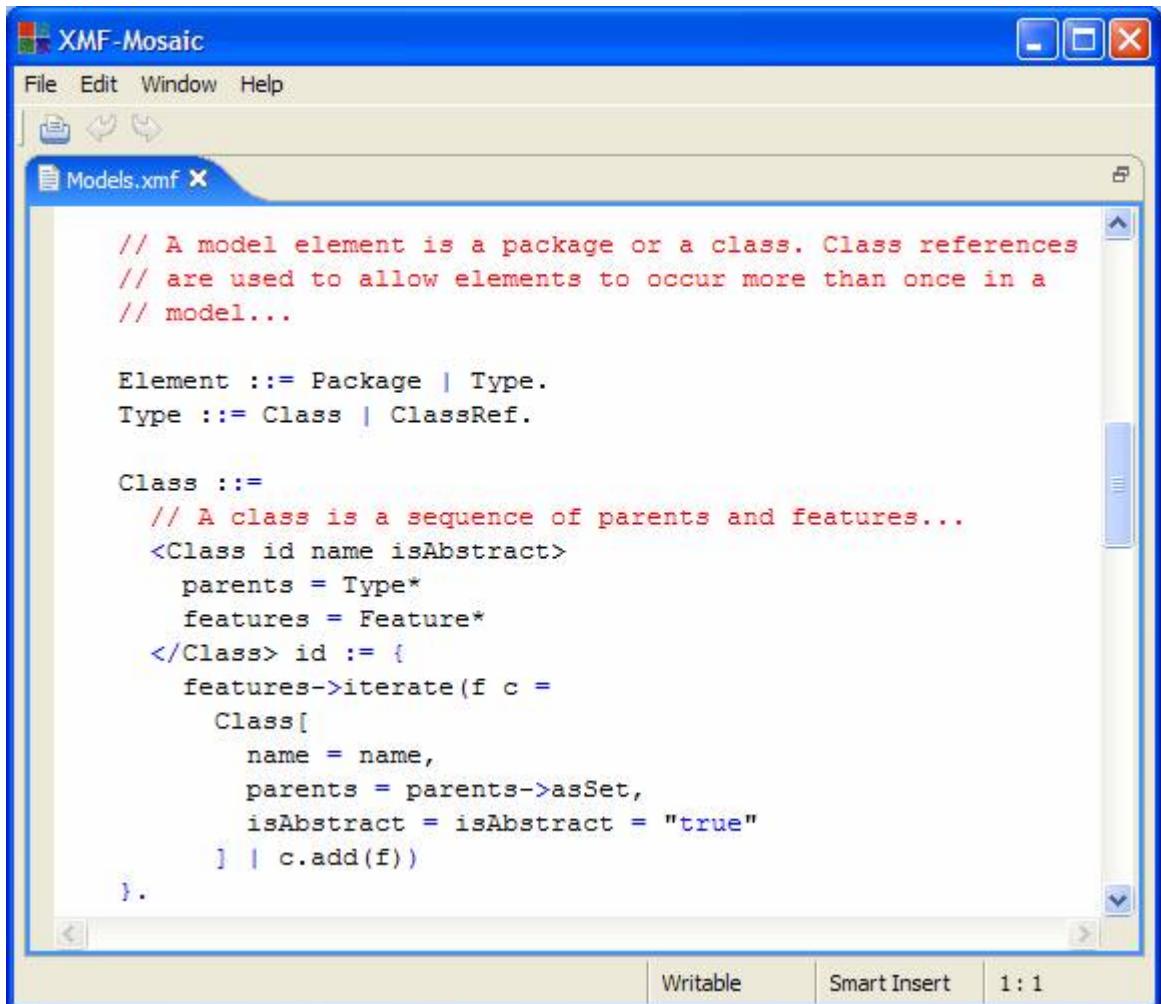
Package ::=

    // A package occurs as a nested element...
    <Package id name = name>
        elements = Element*
    </Package> id := {
        elements->iterate(e p = Package(name) | p.add(e))
    }.

```

At the bottom of the editor, there are buttons for "Writable", "Smart Insert", and a ratio "1:1".

You should be careful when using references (instances of XML::Parser::Ref) as initialization arguments in class instantiation because class instantiation may process the arguments and expect them to be data values of the referenced type rather than the reference itself. In the following construction of a class we use a keyword constructor since the parents may be references (in which case addParent will fail since it expects a classifier rather than a reference).



The screenshot shows the XMF-Mosaic editor interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. A central window titled "Models.xmf" displays the following XML code:

```

// A model element is a package or a class. Class references
// are used to allow elements to occur more than once in a
// model...

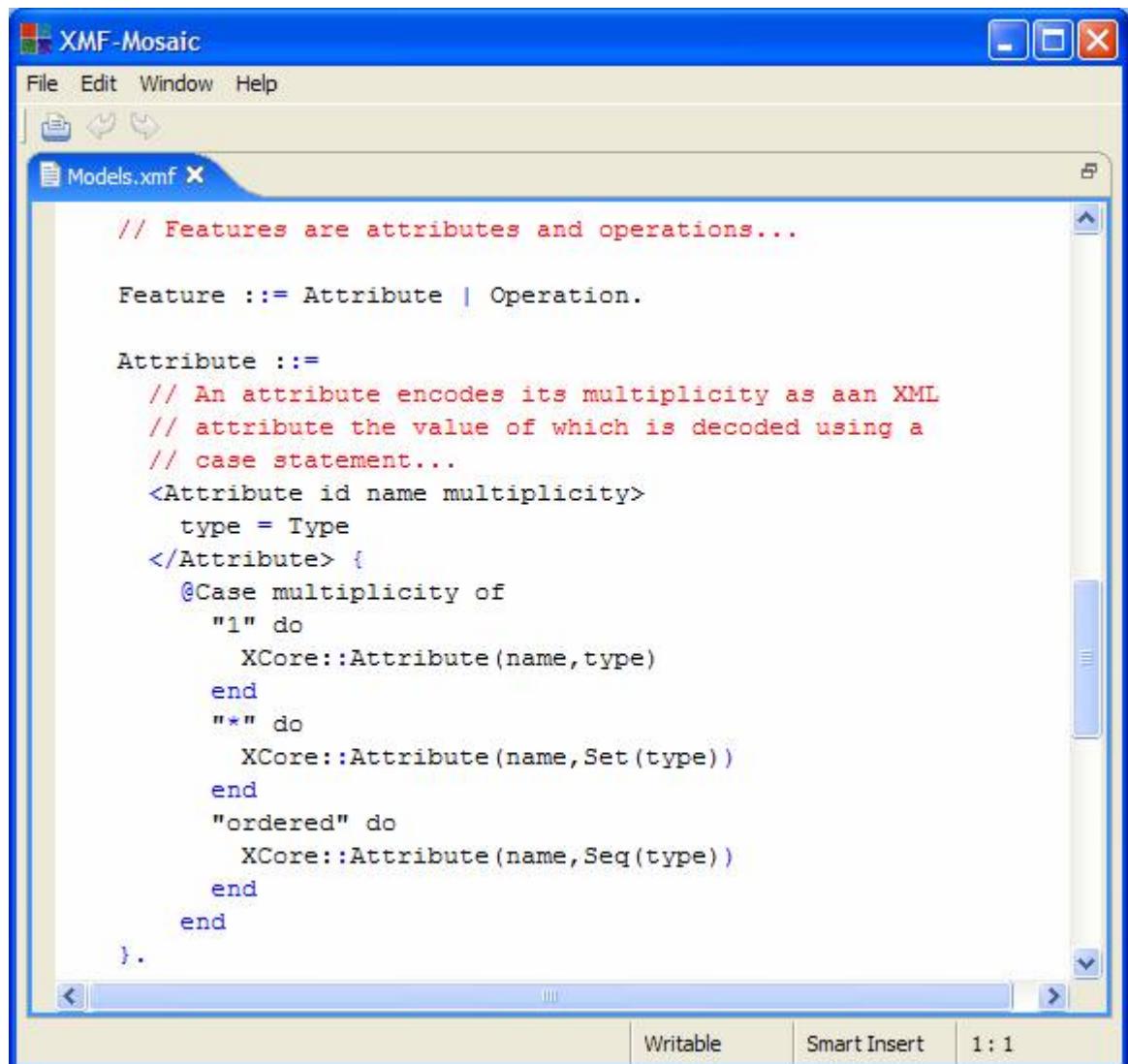
Element ::= Package | Type.
Type ::= Class | ClassRef.

Class ::=
// A class is a sequence of parents and features...
<Class id name isAbstract>
  parents = Type*
  features = Feature*
</Class> id := {
  features->iterate(f c =
    Class[
      name = name,
      parents = parents->asSet,
      isAbstract = isAbstract = "true"
    ] | c.add(f))
}.

```

At the bottom of the editor window, there are buttons for "Writable", "Smart Insert", and a ratio "1:1".

The definition of Attribute below shows a typical scenario where XML attributes encode information that must be processed in order to synthesize the model element. In this case the multiplicity of the attribute (whether it is atomic, a set type or a sequence type) is encoded as an enumerated type that is processed in order to decide what the type of the attribute will be.



The screenshot shows the XMF-Mosaic editor interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. A window titled "Models.xmf" is open, displaying XML code. The code defines a feature as an attribute or operation, and then defines an attribute with its multiplicity encoded as an XML attribute. It uses a case statement to handle three cases: "1", "*", and "ordered". For each case, it creates an XCore::Attribute object with the appropriate name and type.

```
// Features are attributes and operations...

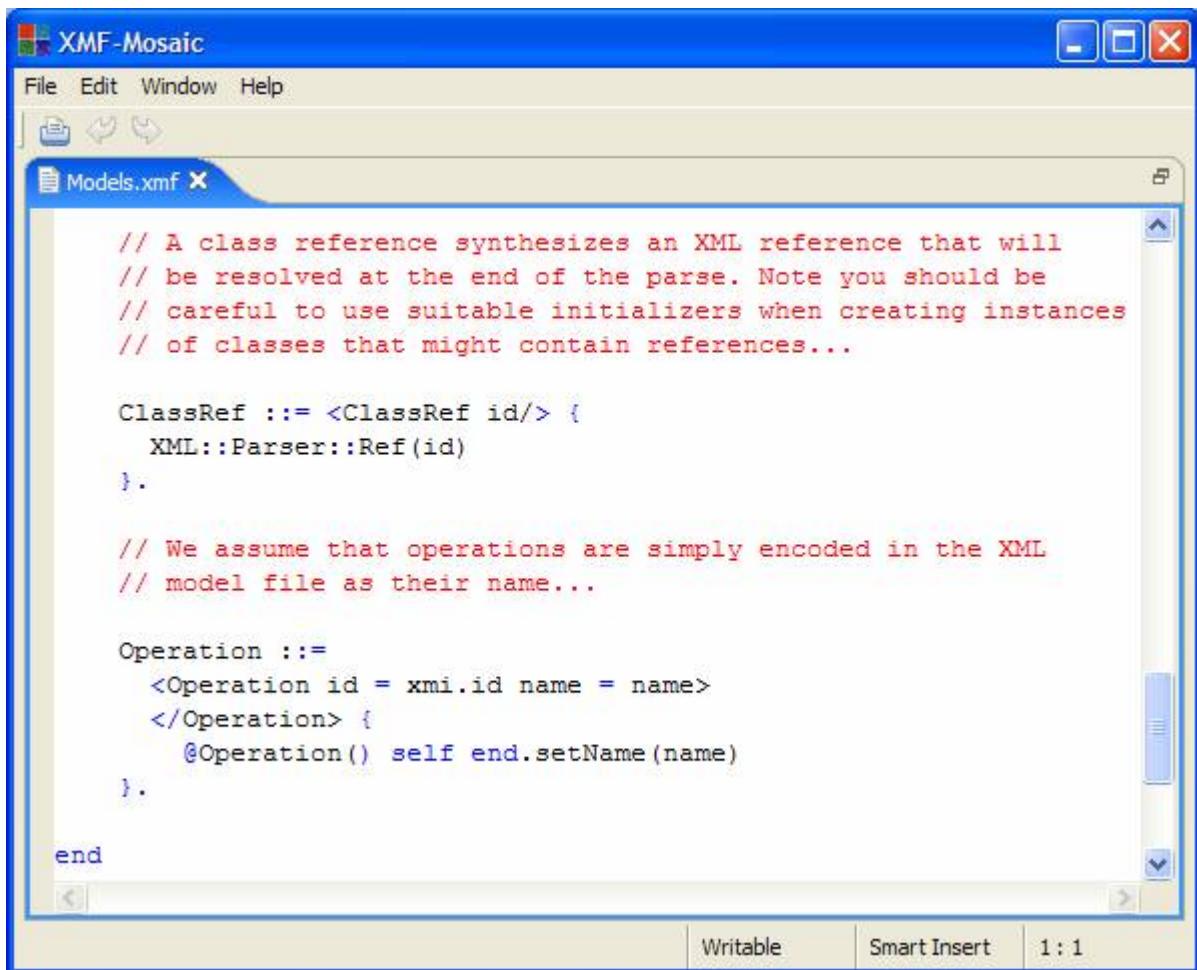
Feature ::= Attribute | Operation.

Attribute ::=

// An attribute encodes its multiplicity as an XML
// attribute the value of which is decoded using a
// case statement...
<Attribute id name multiplicity>
    type = Type
</Attribute> {
    @Case multiplicity of
        "1" do
            XCore::Attribute(name,type)
        end
        "*" do
            XCore::Attribute(name,Set(type))
        end
        "ordered" do
            XCore::Attribute(name,Seq(type))
        end
    end
}.

```

The following ClassRef definition shows how references are synthesized:



The screenshot shows the XMF-Mosaic interface with a window titled "Models.xmf". The code editor contains the following XML grammar definition:

```

// A class reference synthesizes an XML reference that will
// be resolved at the end of the parse. Note you should be
// careful to use suitable initializers when creating instances
// of classes that might contain references...

ClassRef ::= <ClassRef id/> {
    XML::Parser::Ref(id)
}.

// We assume that operations are simply encoded in the XML
// model file as their name...

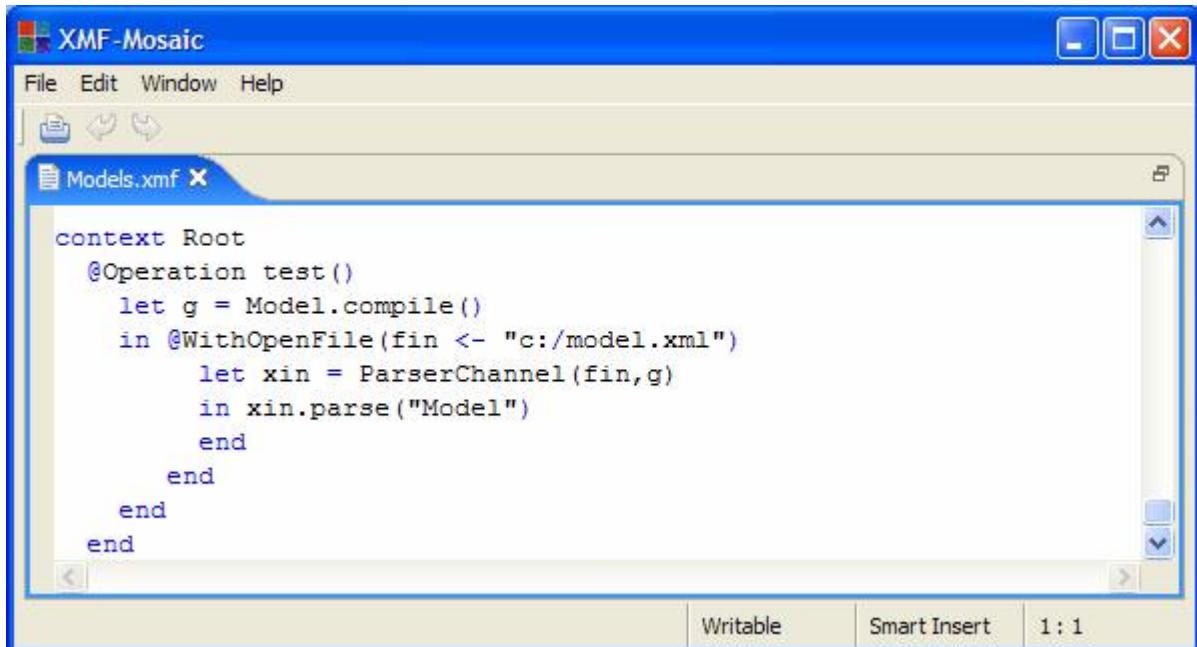
Operation ::= 
    <Operation id = xmi.id name = name>
    </Operation> {
        @Operation() self end.setName(name)
    }.

end

```

The status bar at the bottom right indicates "Writable", "Smart Insert", and a ratio of "1 : 1".

Finally, the following operation can be used to read a file containing a model that is encoded in XML:



The screenshot shows the XMF-Mosaic interface with a window titled "Models.xmf". The code editor contains the following XML grammar definition:

```

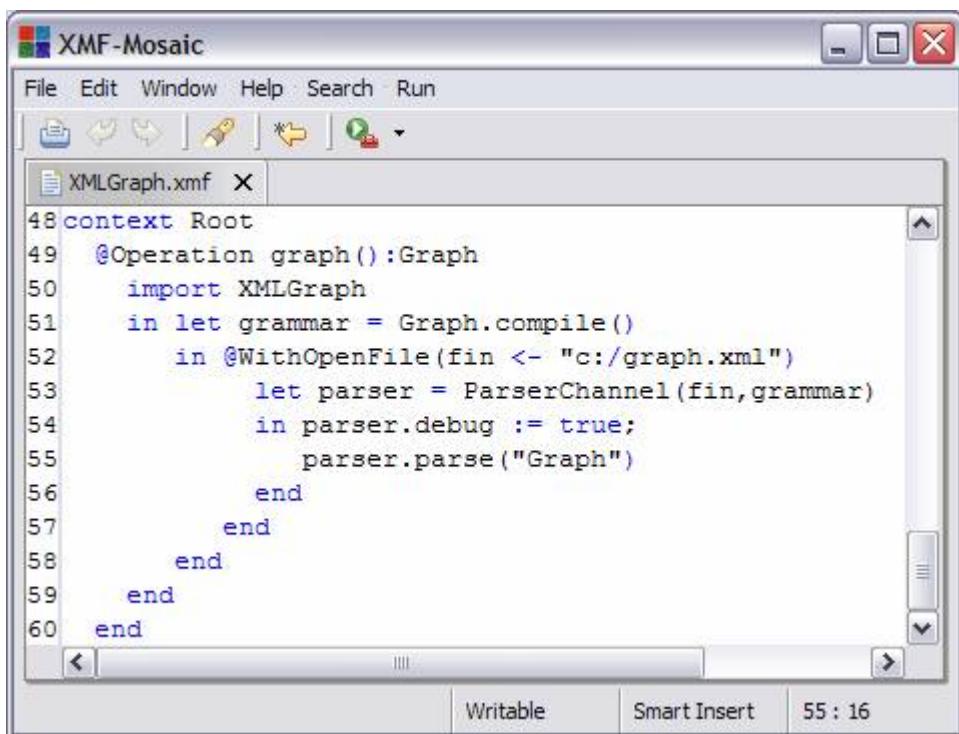
context Root
@Operation test()
let g = Model.compile()
in @WithOpenFile(fin <- "c:/model.xml")
    let xin = ParserChannel(fin,g)
    in xin.parse("Model")
    end
end
end

```

The status bar at the bottom right indicates "Writable", "Smart Insert", and a ratio of "1 : 1".

Debugging XML Grammars

An XML parser channel has an attribute debug that can be used to provide a trace of execution. If the flag is set:

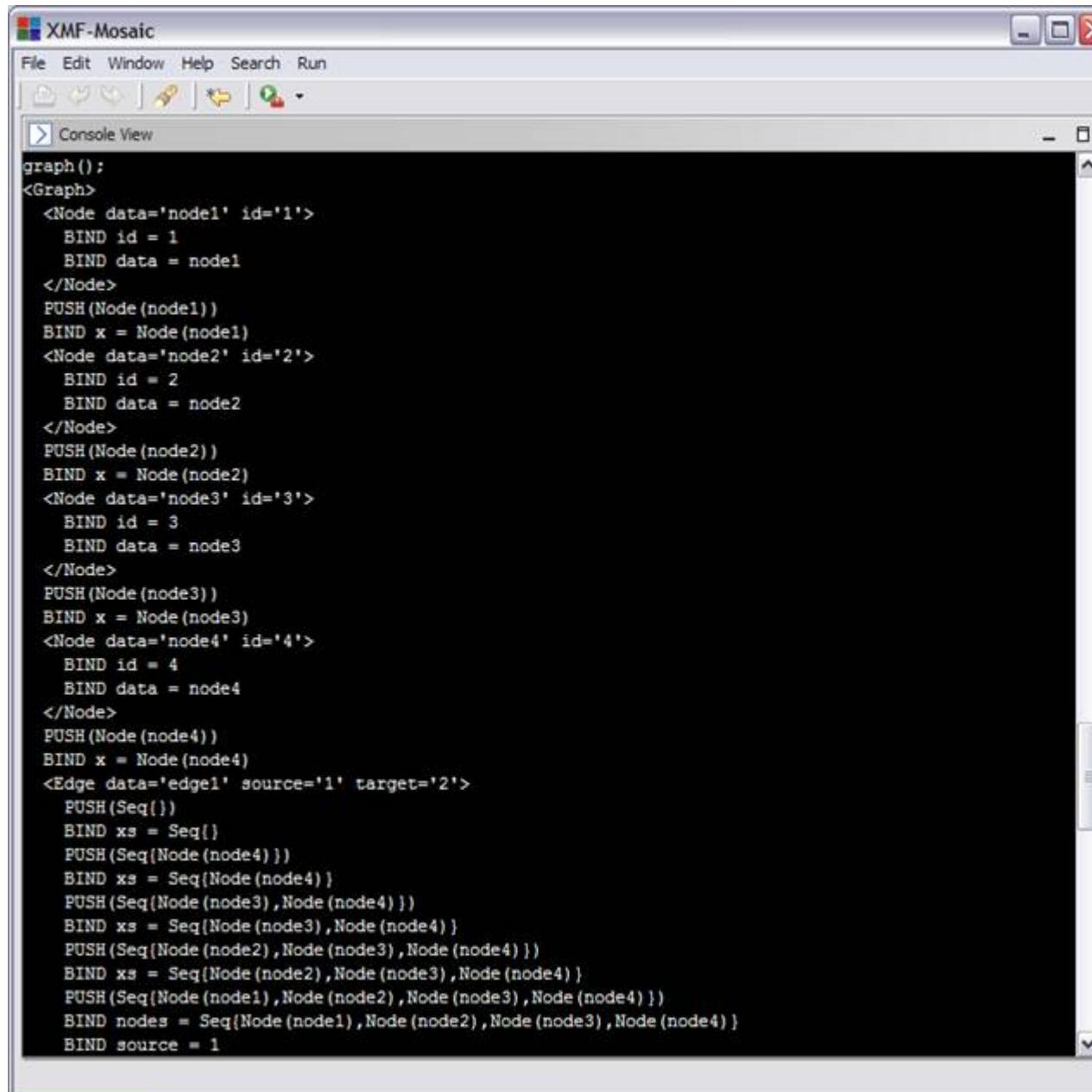


The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations like Open, Save, and Run. The main window displays a code editor with the file "XMLGraph.xmf" open. The code is as follows:

```
48 context Root
49   @Operation graph():Graph
50     import XMLGraph
51     in let grammar = Graph.compile()
52       in @WithOpenFile(fin <- "c:/graph.xml")
53         let parser = ParserChannel(fin,grammar)
54         in parser.debug := true;
55           parser.parse("Graph")
56         end
57       end
58     end
59   end
60 end
```

At the bottom of the code editor, there are buttons for Writable, Smart Insert, and a status bar showing "55 : 16".

then output is produced that shows the XML input as it is consumed, shows the result of actions as they are performed and shows the clause variables as they are bound. The following is a partial trace produced by reading a graph with debug set:

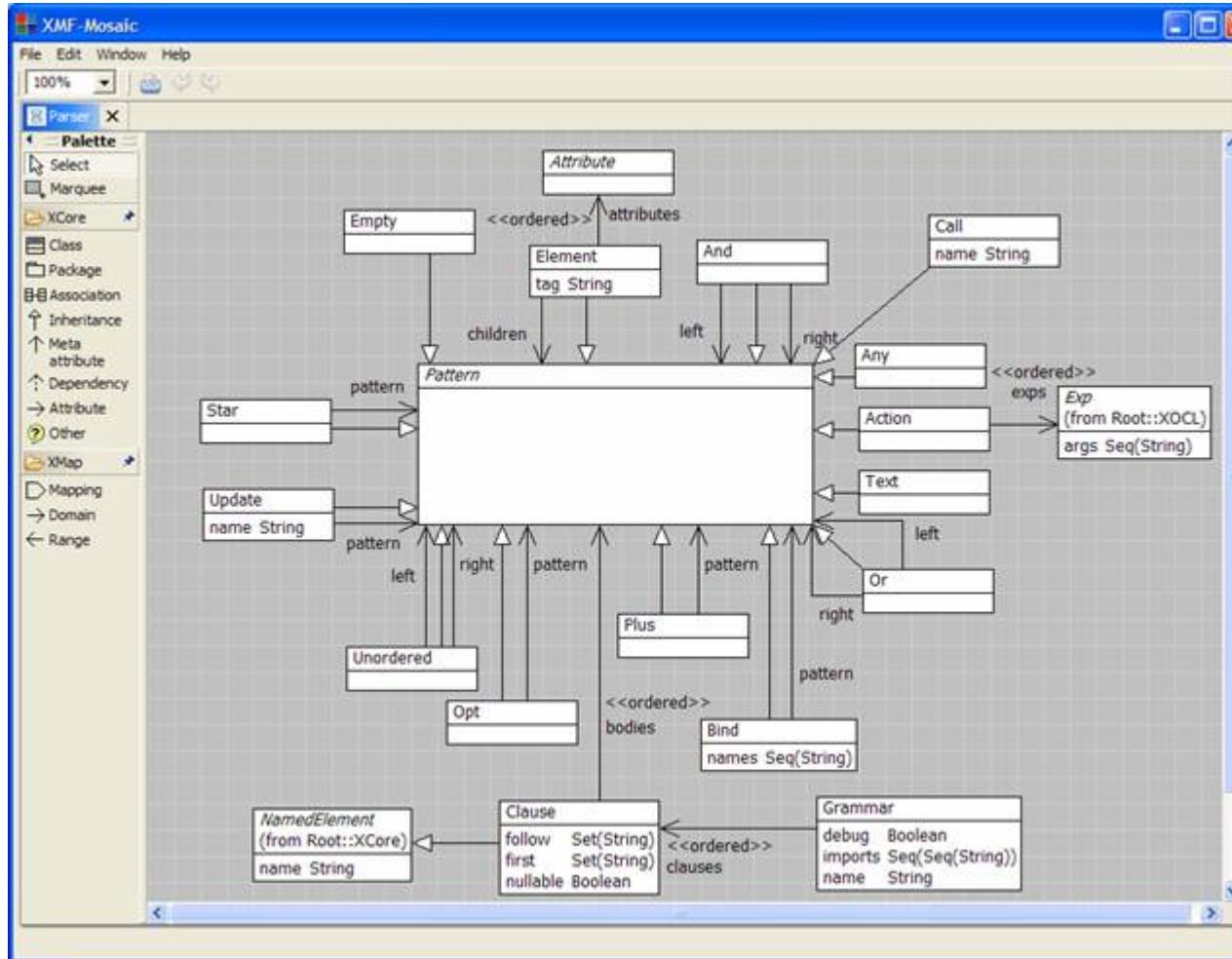


The screenshot shows the XMF-Mosaic application window. The title bar reads "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", "Help", "Search", and "Run". Below the menu is a toolbar with various icons. The main area is titled "Console View" and contains the following XML code:

```
graph();
<Graph>
    <Node data='node1' id='1'>
        BIND id = 1
        BIND data = node1
    </Node>
    PUSH(Node(node1))
    BIND x = Node(node1)
    <Node data='node2' id='2'>
        BIND id = 2
        BIND data = node2
    </Node>
    PUSH(Node(node2))
    BIND x = Node(node2)
    <Node data='node3' id='3'>
        BIND id = 3
        BIND data = node3
    </Node>
    PUSH(Node(node3))
    BIND x = Node(node3)
    <Node data='node4' id='4'>
        BIND id = 4
        BIND data = node4
    </Node>
    PUSH(Node(node4))
    BIND x = Node(node4)
    <Edge data='edge1' source='1' target='2'>
        PUSH(Seq())
        BIND xs = Seq()
        PUSH(Seq(Node(node4)))
        BIND xs = Seq(Node(node4))
        PUSH(Seq(Node(node3),Node(node4)))
        BIND xs = Seq(Node(node3),Node(node4))
        PUSH(Seq(Node(node2),Node(node3),Node(node4)))
        BIND xs = Seq(Node(node2),Node(node3),Node(node4))
        PUSH(Seq(Node(node1),Node(node2),Node(node3),Node(node4)))
        BIND nodes = Seq(Node(node1),Node(node2),Node(node3),Node(node4))
        BIND source = 1

```

The XML Parsing Grammar



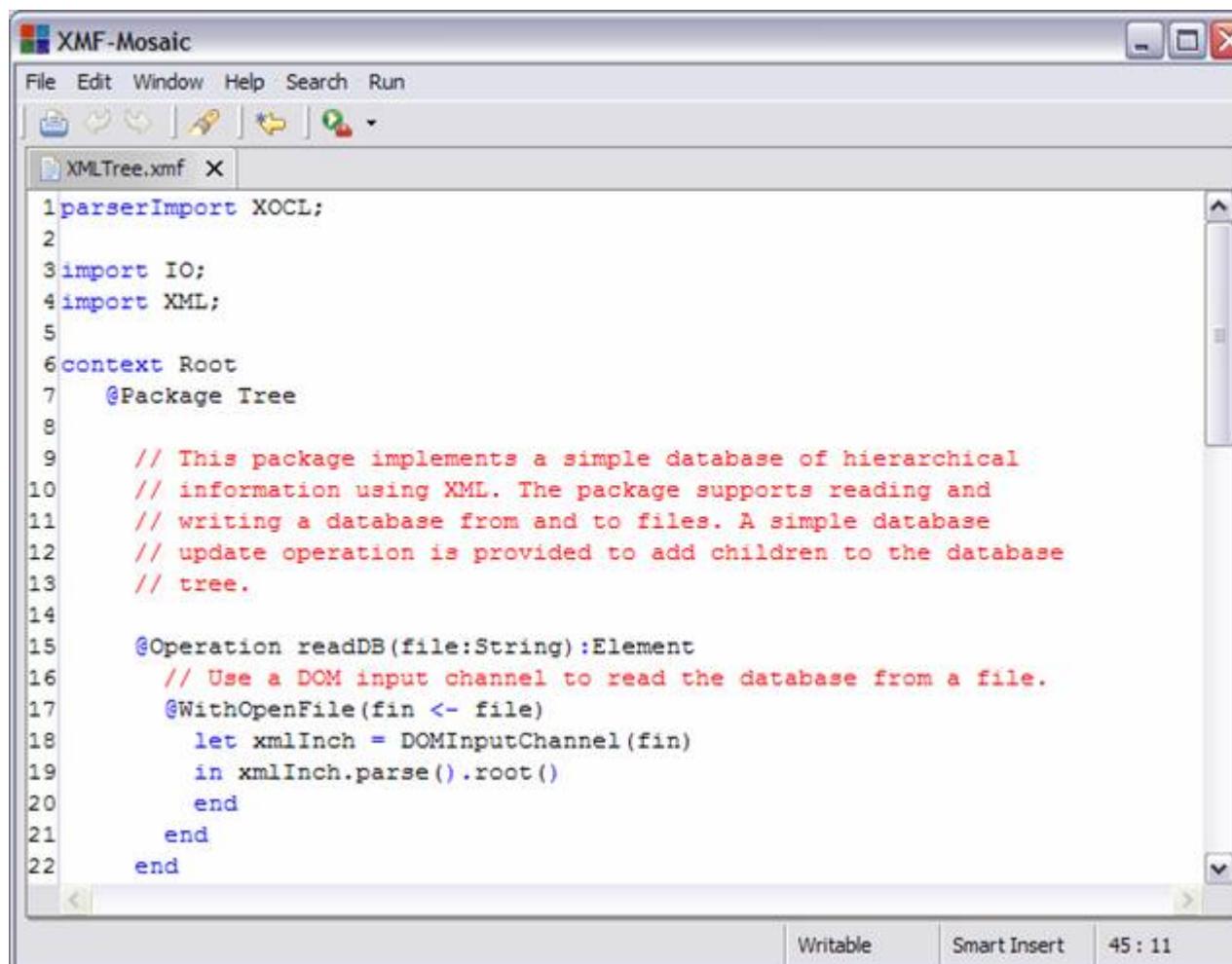
DOM Input Channel

The XML parser provided by XMF does not give direct access to the XML elements in a document. The parser matches against the XML as it is read and the parser actions are used to synthesize any XMF element.

Certain applications require access to the XML representation of the data. For example, the XML model may be the domain model. In this case you can write an XML parser that synthesizes an instance of the XML model. This use case is supported directly in XMF by DOMInputChannel. The use of this class is equivalent to writing a bespoke parser that synthesizes XML; however, it is much more efficient because general parsing machinery is not required to recognize the XML elements as they are encountered.

This section shows how DOMInputChannel works by providing a simple example of its use.

Consider the management of some hierarchical data such as classification trees, component descriptions, and organisation structures. The data can be maintained as a database in XML format. Database queries and update can be implemented as operations that work directly on the XML data. The following Tree package defines operations that partially implement such a database:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, Run. The toolbar has icons for Open, Save, Undo, Redo, Cut, Copy, Paste, Find, and Delete. A toolbar button for "File" is also present. The main window shows a file named "XMLTree.xmf" with the following code:

```

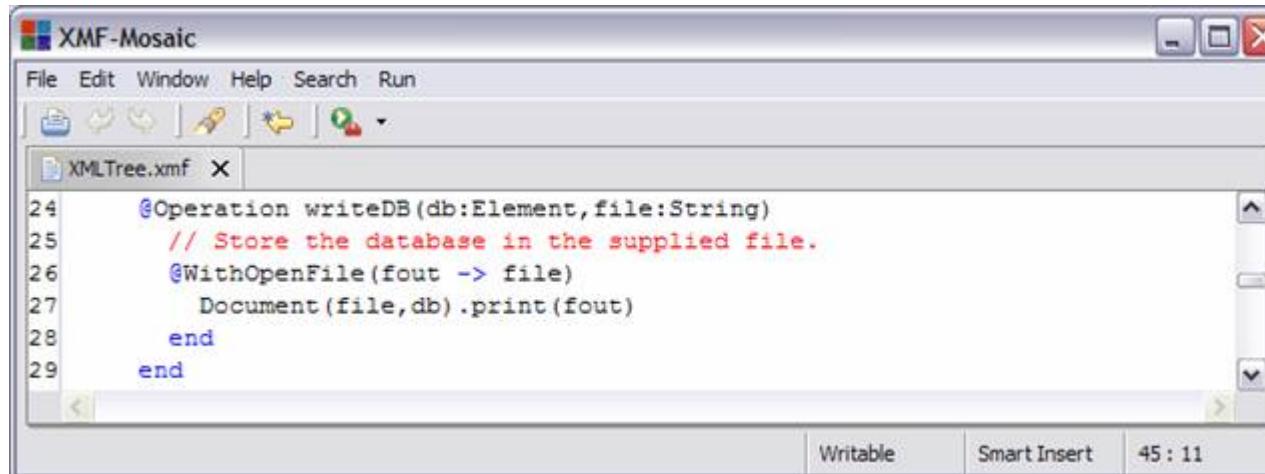
1 parserImport XOCL;
2
3 import IO;
4 import XML;
5
6 context Root
7     @Package Tree
8
9     // This package implements a simple database of hierarchical
10    // information using XML. The package supports reading and
11    // writing a database from and to files. A simple database
12    // update operation is provided to add children to the database
13    // tree.
14
15    @Operation readDB(file:String):Element
16        // Use a DOM input channel to read the database from a file.
17        @WithOpenFile(fin <- file)
18            let xmlInch = DOMInputChannel(fin)
19            in xmlInch.parse().root()
20            end
21        end
22    end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "45 : 11".

The `readDB` operation uses a DOM input channel to read the contents of a standard XML file. The result of performing the parse operation is an XML document whose root element is the database. The constructor for the class `DOMInputChannel` accepts an input channel that produces XML characters when read.

The following operation uses the `print` operation of `XML::Document` to write the database to a text file:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, Run. The toolbar has icons for Open, Save, Undo, Redo, Cut, Copy, Paste, Find, and Delete. A toolbar button for "File" is also present. The main window shows a file named "XMLTree.xmf" with the following code:

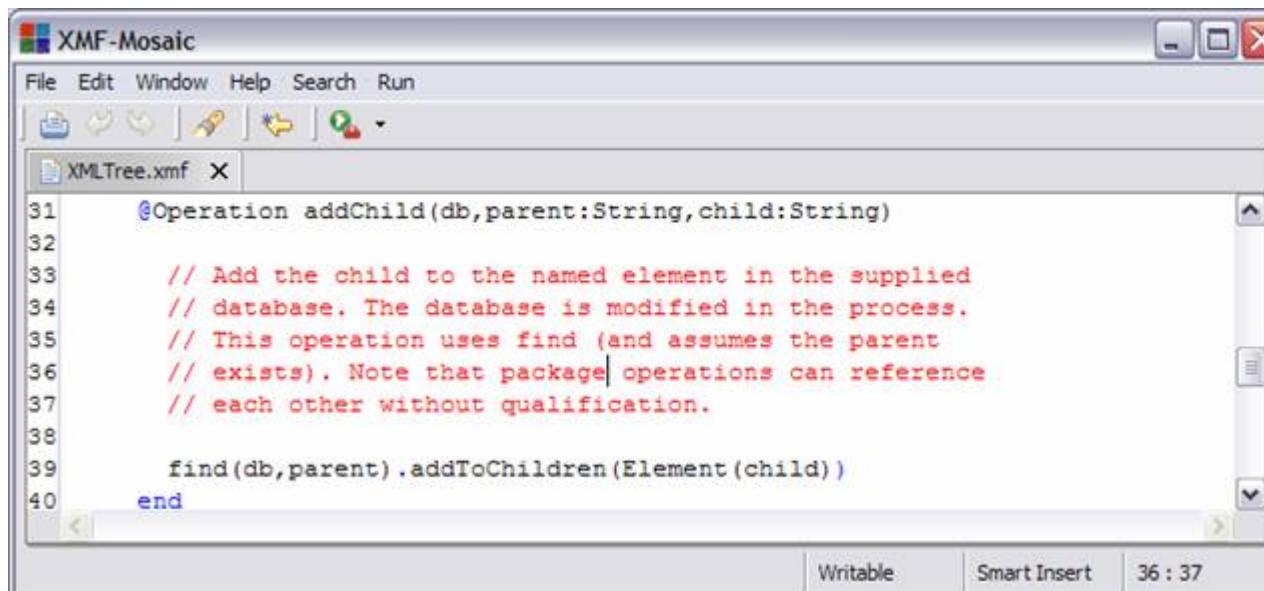
```

24    @Operation writeDB(db:Element,file:String)
25        // Store the database in the supplied file.
26        @WithOpenFile(fout -> file)
27            Document(file,db).print(fout)
28        end
29    end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "45 : 11".

Once the database has been read (or constructed by directly creating instances of the `XMLmodel`). It can be manipulated using XOCL. The following operation adds a child to an element in the database:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, Run. The toolbar has icons for New, Open, Save, Cut, Copy, Paste, Find, and Run. The main window shows a file named "XMLTree.xmf" with the following code:

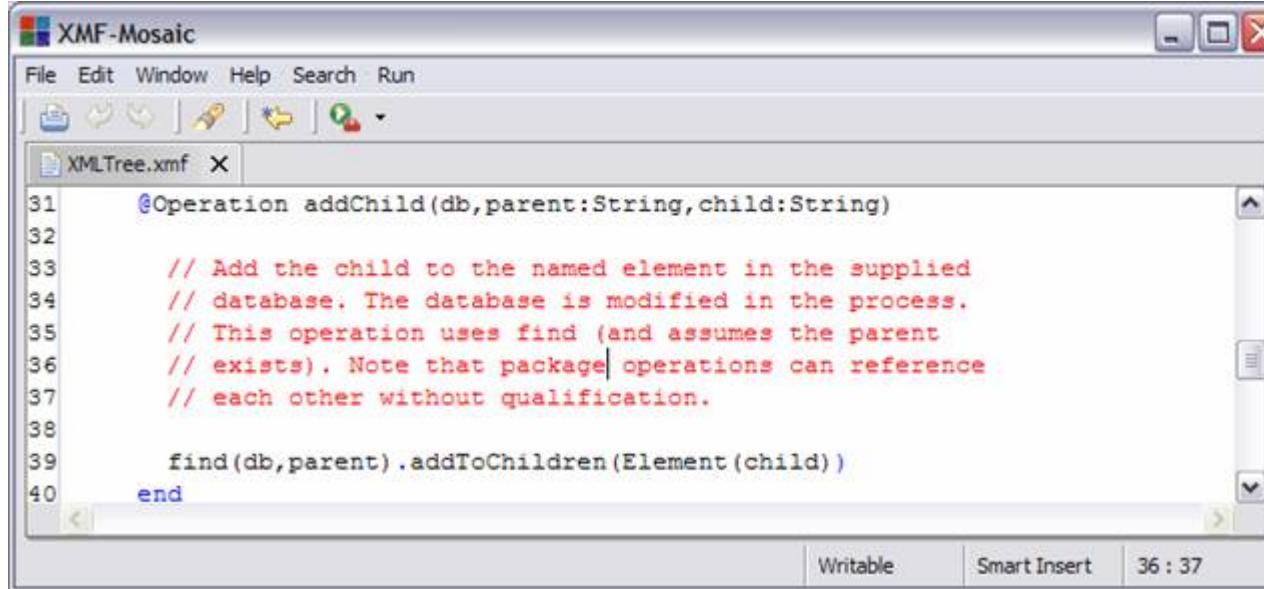
```

31     @Operation addChild(db,parent:String,child:String)
32
33     // Add the child to the named element in the supplied
34     // database. The database is modified in the process.
35     // This operation uses find (and assumes the parent
36     // exists). Note that package| operations can reference
37     // each other without qualification.
38
39     find(db,parent).addToChildren(Element(child))
40   end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "36 : 37".

The following query operation uses pattern matching to select an element:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, Run. The toolbar has icons for New, Open, Save, Cut, Copy, Paste, Find, and Run. The main window shows a file named "XMLTree.xmf" with the same code as the previous screenshot, but with a different line 39:

```

31     @Operation addChild(db,parent:String,child:String)
32
33     // Add the child to the named element in the supplied
34     // database. The database is modified in the process.
35     // This operation uses find (and assumes the parent
36     // exists). Note that package| operations can reference
37     // each other without qualification.
38
39     find(db,parent).addChildren(Element(child))
40   end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and "36 : 37".

SAX Input Channel

XML input can be processed using an XML grammar or using a DOM input channel. The former provides a convenient way of declaring XML patterns and synthesizing XMF data. The latter provides a convenient way of translating an XML input source into an instance of the XML model. Occasionally, the XML instance is not required and the overhead of using general parsing machinery seems unnecessary. For example, if we wanted to count the number of elements in the input or check whether an element with a given tag exists in the input.

For occasions where parsing is unnecessary and DOM is not required, XMF provides a SAX input channel. A SAX input channel processes the XML input by generating events each time an XML node is encountered in the input. The channel has handlers that are called when the events are fired. XMF works hard to ensure that the input is processed as efficiently as possible and therefore, SAX input channels are appropriate for processing very large XML data sources.

The key operations called on an instance of SAXInputChannel are as follows:

```
startElement(tag:Buffer,attributes:Buffer)
```

This operation is called when each element is encountered in the input. The tag of the element is supplied as a string buffer and the attributes are supplied as a buffer containing instances of XML::IO::SAXAttribute with attributes name and value both of type string buffer. All buffers are reused in order to ensure the input is processed as efficiently as possible. Therefore if you wish to retain the tag, or the attribute names or attribute values then the buffers should be transformed into the equivalent strings using `toString`.

```
endElement(tag:Buffer)
```

This operation is called when the end tag of each element is encountered in the input. The tag is a string buffer that is subsequently reused as above.

```
characters(text:Buffer)
```

This operation is called when text is encountered in the input. The text is supplied as a string buffer that is subsequently reused as above.

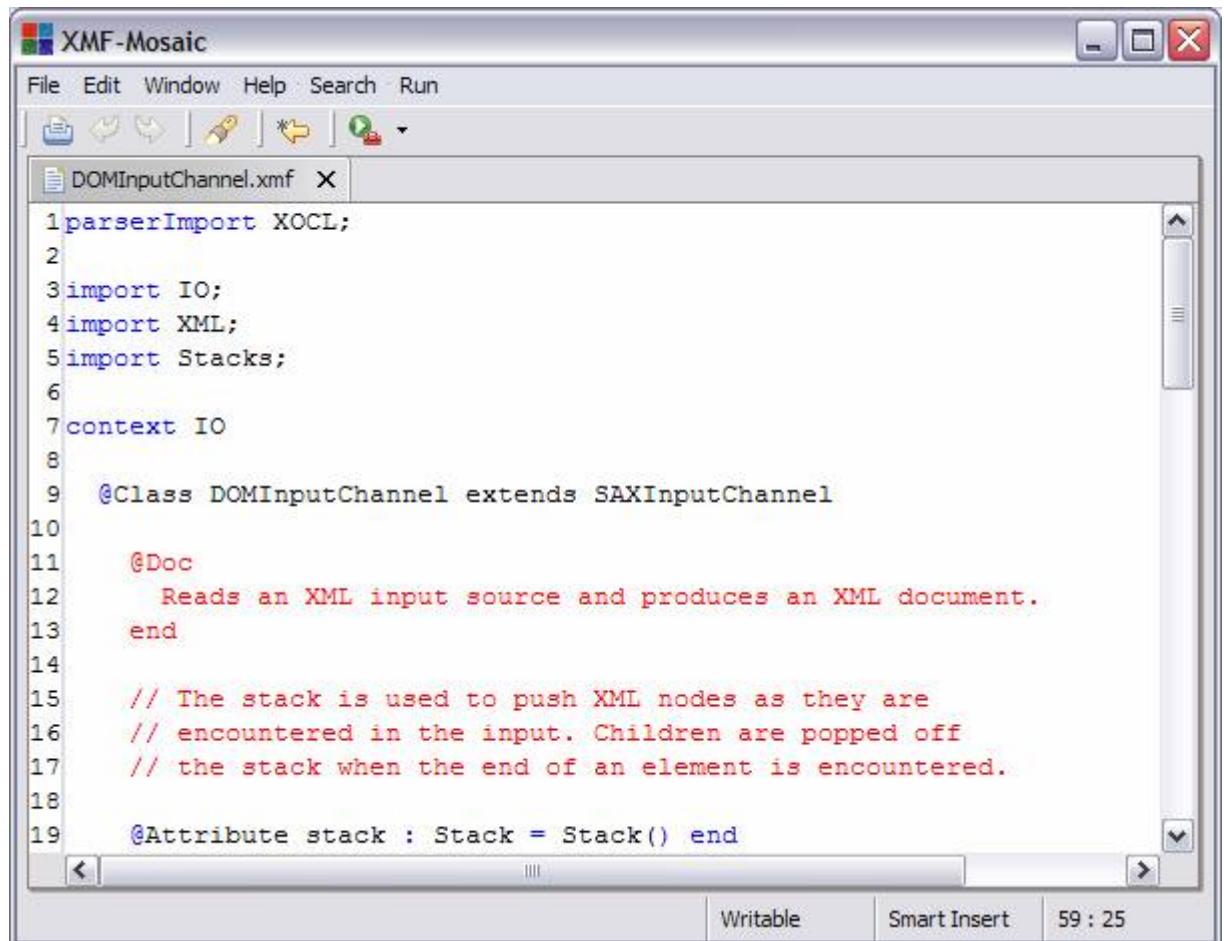
```
parse()
```

Call this operation to start processing the input. No value is returned by default. Extend this operation in sub-classes of SAXInputChannel to construct and return values based on the operations defined above.

```
SAXInputChannel(inch:InputChannel)
```

A constructor for SAX input channels. The argument is an input channel that produces XML text.

This section shows how SAX input channels work in XMF by implementing a DOM input channel in terms of a SAX input channel. The following tool screenshot shows the class uses a stack to manage the XML nodes as they are read and synthesized by the operations defined above:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes File, Edit, Window, Help, Search, and Run. Below the menu is a toolbar with icons for file operations. A central window displays the code for "DOMInputChannel.xmf". The code is as follows:

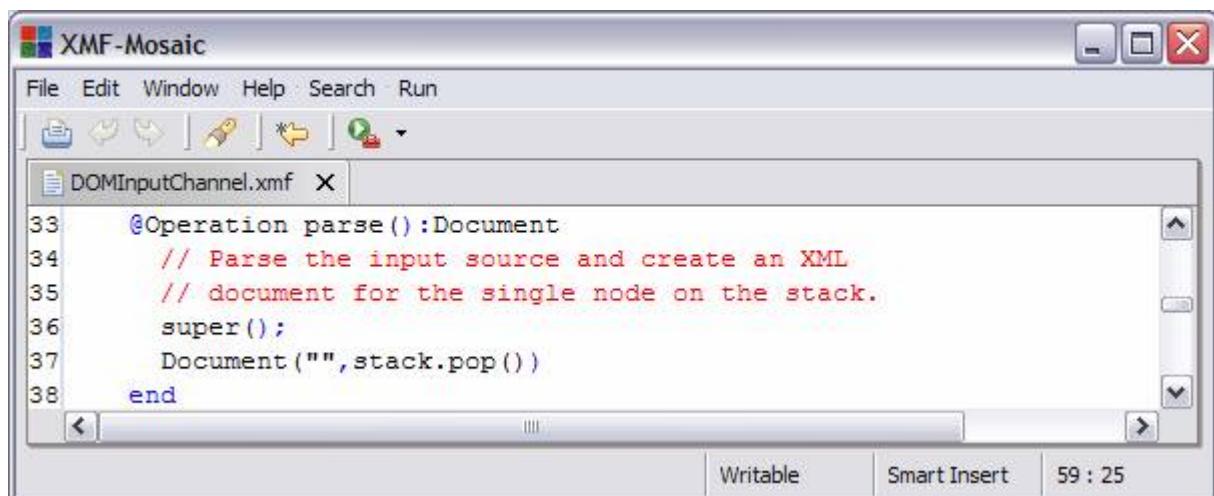
```

1 parserImport XOCL;
2
3 import IO;
4 import XML;
5 import Stacks;
6
7 context IO
8
9 @Class DOMInputChannel extends SAXInputChannel
10
11     @Doc
12         Reads an XML input source and produces an XML document.
13     end
14
15     // The stack is used to push XML nodes as they are
16     // encountered in the input. Children are popped off
17     // the stack when the end of an element is encountered.
18
19     @Attribute stack : Stack = Stack() end

```

The code is syntax-highlighted, with keywords in blue and comments in red. The status bar at the bottom right shows "Writable", "Smart Insert", and "59 : 25".

When the input source is parsed, the intermediate XML nodes are pushed and popped on the stack. Child nodes are popped when the end element event occurs and added to the currently open element on the stack. Eventually, the root node is left as the single stack element. The parse just pops the root node, creates and returns an XML document:



The screenshot shows the XMF-Mosaic IDE interface. The menu bar includes File, Edit, Window, Help, Search, and Run. The toolbar contains icons for file operations like Open, Save, and Cut/Paste. A tab bar at the top shows 'DOMInputChannel.xmf X'. The main code editor area contains the following Java-like pseudocode:

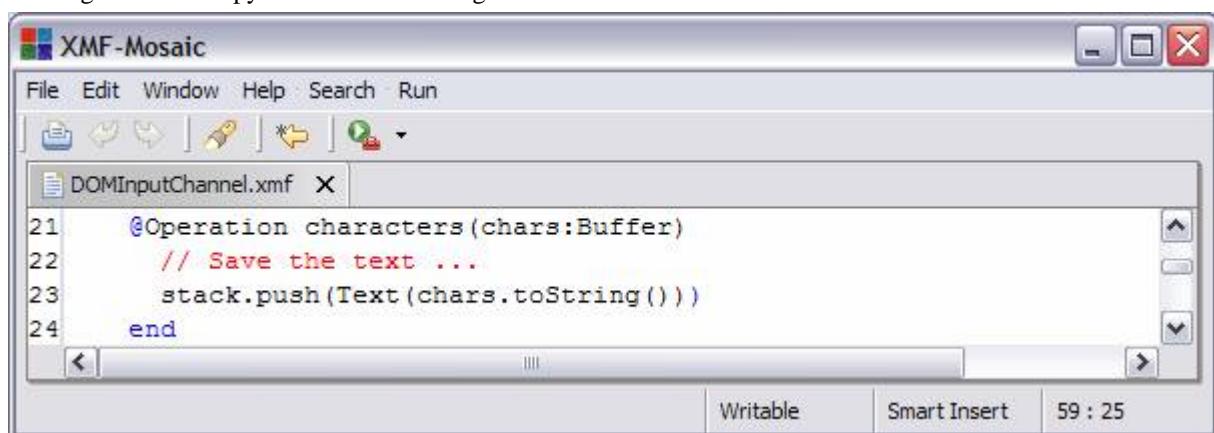
```

33     @Operation parse():Document
34         // Parse the input source and create an XML
35         // document for the single node on the stack.
36         super();
37         Document("",stack.pop())
38     end

```

The status bar at the bottom right shows 'Writable', 'Smart Insert', and '59 : 25'.

When the characters operation is called, a Text node is created and pushed on the stack. Note how `toString` is used to copy the text on the string buffer:



The screenshot shows the XMF-Mosaic IDE interface. The menu bar includes File, Edit, Window, Help, Search, and Run. The toolbar contains icons for file operations like Open, Save, and Cut/Paste. A tab bar at the top shows 'DOMInputChannel.xmf X'. The main code editor area contains the following Java-like pseudocode:

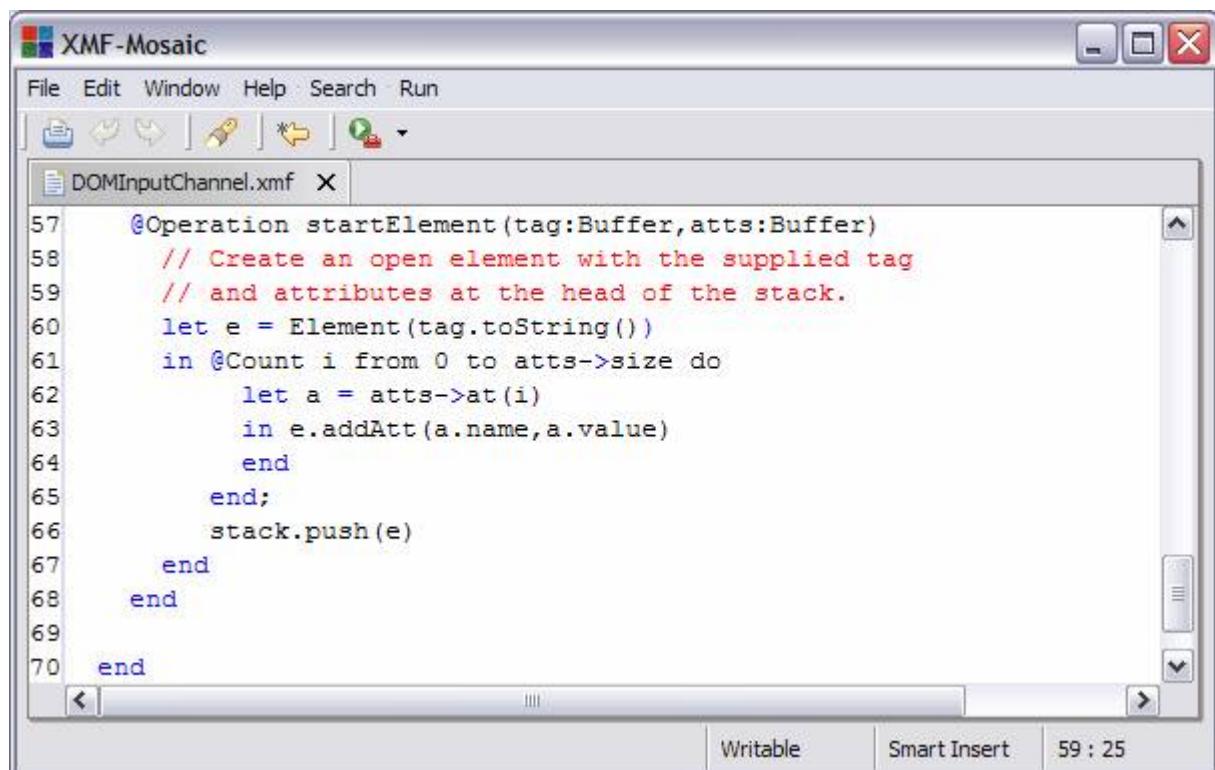
```

21     @Operation characters(chars:Buffer)
22         // Save the text ...
23         stack.push(Text(chars.toString()))
24     end

```

The status bar at the bottom right shows 'Writable', 'Smart Insert', and '59 : 25'.

When a new element node is encountered in the input we create a new XML element object and push it on the stack. Between the start element event and the associated end element event there will be any number of events that create the children nodes. The children nodes will be pushed on the stack. When the end element event occurs it is straightforward to pop the child elements from the stack up to the parent element and modify the parent by inserting the children:



The screenshot shows the XMF-Mosaic IDE interface with the file `DOMInputChannel.xmf` open. The code implements the `startElement` operation:

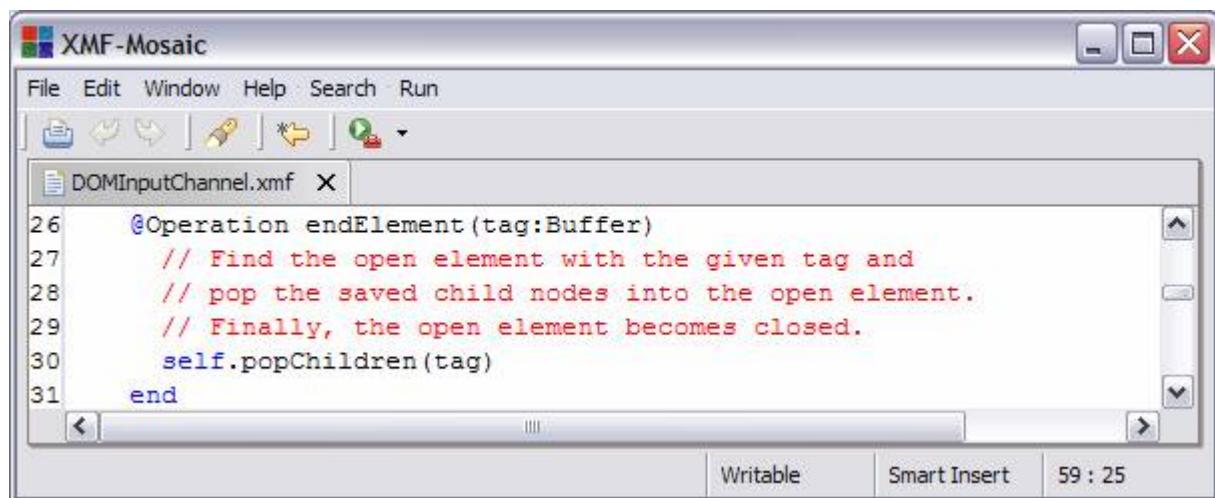
```

57     @Operation startElement(tag:Buffer,atts:Buffer)
58         // Create an open element with the supplied tag
59         // and attributes at the head of the stack.
60         let e = Element(tag.toString())
61         in @Count i from 0 to atts->size do
62             let a = atts->at(i)
63             in e.addAtt(a.name,a.value)
64             end
65         end;
66         stack.push(e)
67     end
68 end
69
70 end

```

The code uses a stack to keep track of elements and their attributes as they are parsed.

The end element event causes all the children that have been pushed since the corresponding parent element node to be popped and inserted:



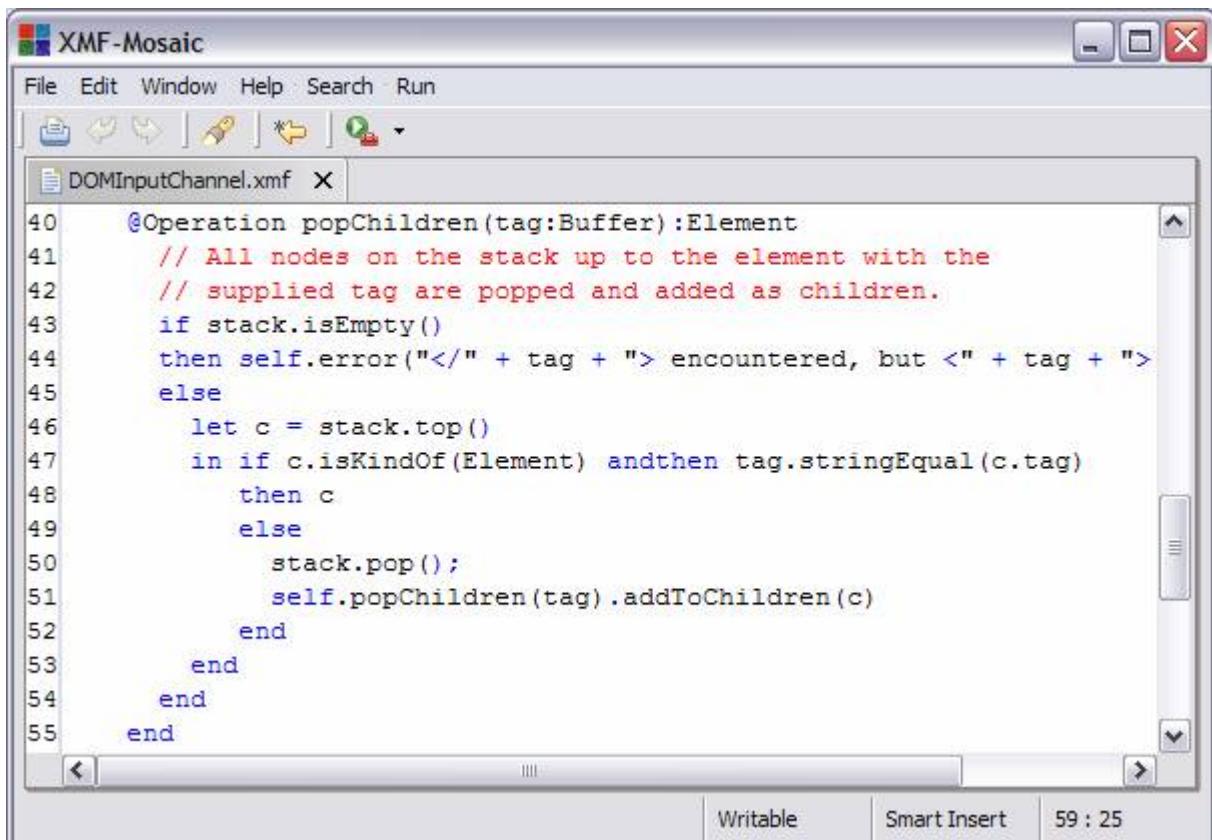
The screenshot shows the XMF-Mosaic IDE interface with the file `DOMInputChannel.xmf` open. The code implements the `endElement` operation:

```

26     @Operation endElement(tag:Buffer)
27         // Find the open element with the given tag and
28         // pop the saved child nodes into the open element.
29         // Finally, the open element becomes closed.
30         self.popChildren(tag)
31     end

```

This part of the code handles the closing of elements by popping the stack and inserting the children back into the open element.



```

40     @Operation popChildren(tag:Buffer):Element
41         // All nodes on the stack up to the element with the
42         // supplied tag are popped and added as children.
43         if stack.isEmpty()
44             then self.error("</" + tag + "> encountered, but <" + tag + ">")
45         else
46             let c = stack.top()
47             in if c.isKindOf(Element) andthen tag.stringEqual(c.tag)
48                 then c
49                 else
50                     stack.pop();
51                     self.popChildren(tag).addToChildren(c)
52                 end
53             end
54         end
55     end

```

XML Output

Introduction

When developing XMF applications it is likely that you will want to generate XML from XMF data. This can be achieved in a number of ways depending on the level of control and the amount of reuse that you want to have over the output. This section describes approaches to XML output. The most direct way of producing an XML document is to print the characters, for example:

```
format(out, "<Class name='~S' id='~S' />", Seq{c.name(), c.path()})
```

This approach to output provides complete control over the text that is produced, but suffers from a lack of XML semantics – the format command does not know it is processing an XML element.

There are two strategies for producing XML output that are more attractive than directly writing characters:

- Generating instances of the XML model and then sending the instance a print message.
- Using the XML::PrintXML::XML construct to specify output patterns.

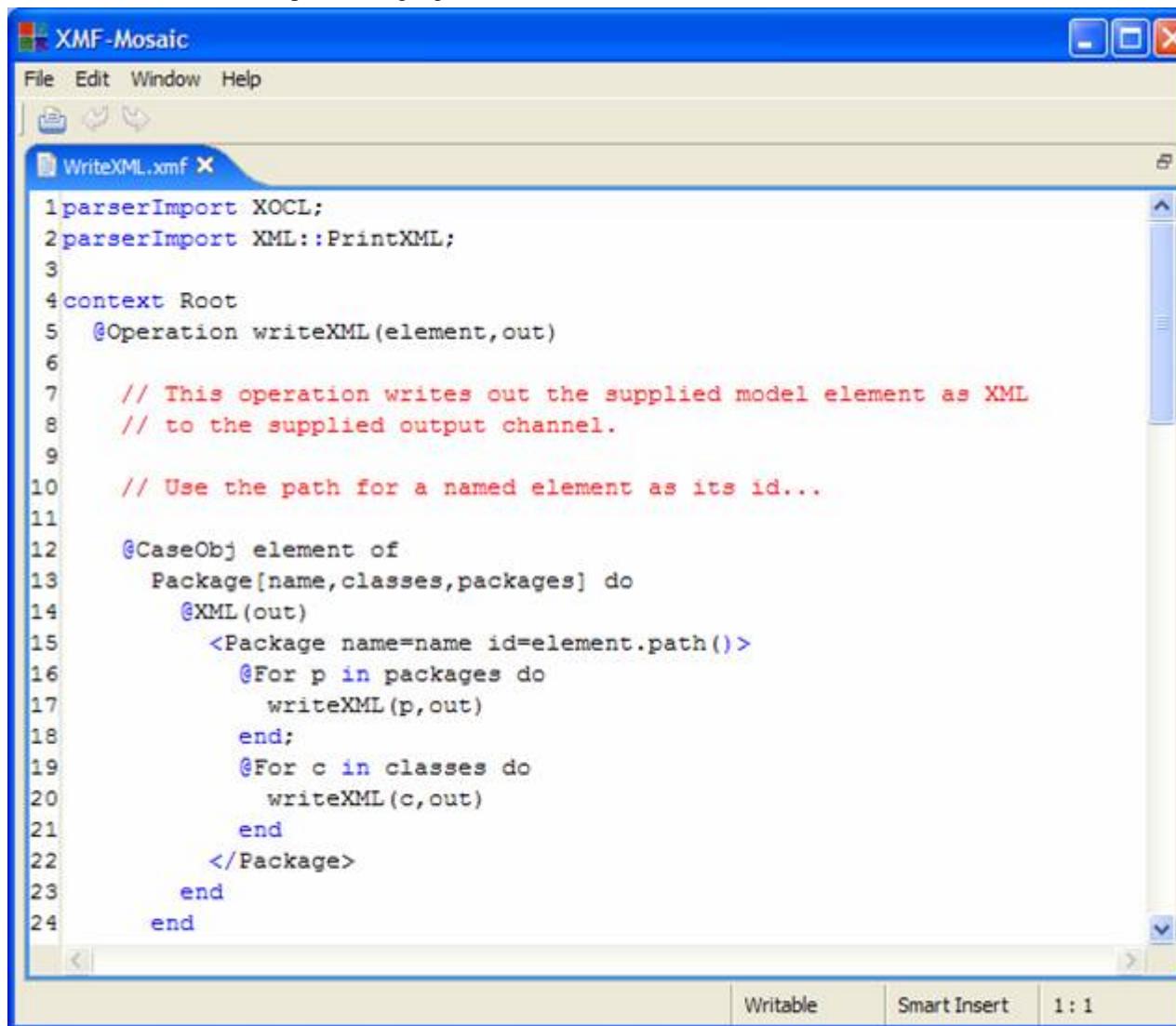
The rest of this section describe the second of these approaches.

XML Output Patterns

Consider the task of translating a model to an XML document. The model consists of packages, classes, attributes and operations. The XML tags reflect the type of model elements. Models contain multiple

references to the same model elements, this is encoded in the XML document using attributes whose values are unique identifiers.

The following operation defines a mapping that writes a model element to an XML document. It uses the XML::PrintXML::XML pattern language:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main window contains a code editor titled "WriteXML.xmf". The code is written in XML::PrintXML::XML pattern language:

```

1 parserImport XOCL;
2 parserImport XML::PrintXML;
3
4 context Root
5 @Operation writeXML(element,out)
6
7 // This operation writes out the supplied model element as XML
8 // to the supplied output channel.
9
10 // Use the path for a named element as its id...
11
12 @CaseObj element of
13   Package[name,classes,packages] do
14     @XML(out)
15       <Package name=name id=element.path()>
16         @For p in packages do
17           writeXML(p,out)
18         end;
19         @For c in classes do
20           writeXML(c,out)
21         end
22       </Package>
23     end
24   end

```

The code editor has status bars at the bottom indicating "Writable", "Smart Insert", and "1:1".

Line 12 starts a case analysis on the supplied element. The CaseObj construct matches the supplied value (element) against patterns consisting of the path to the direct classifier of the value followed by a number of slot names defined by the classifier. If the value is a direct instance of the classifier (i.e. its of operation returns the named classifier) then the body of the matching CaseObj clause is performed with the named slots bound to the corresponding slot values.

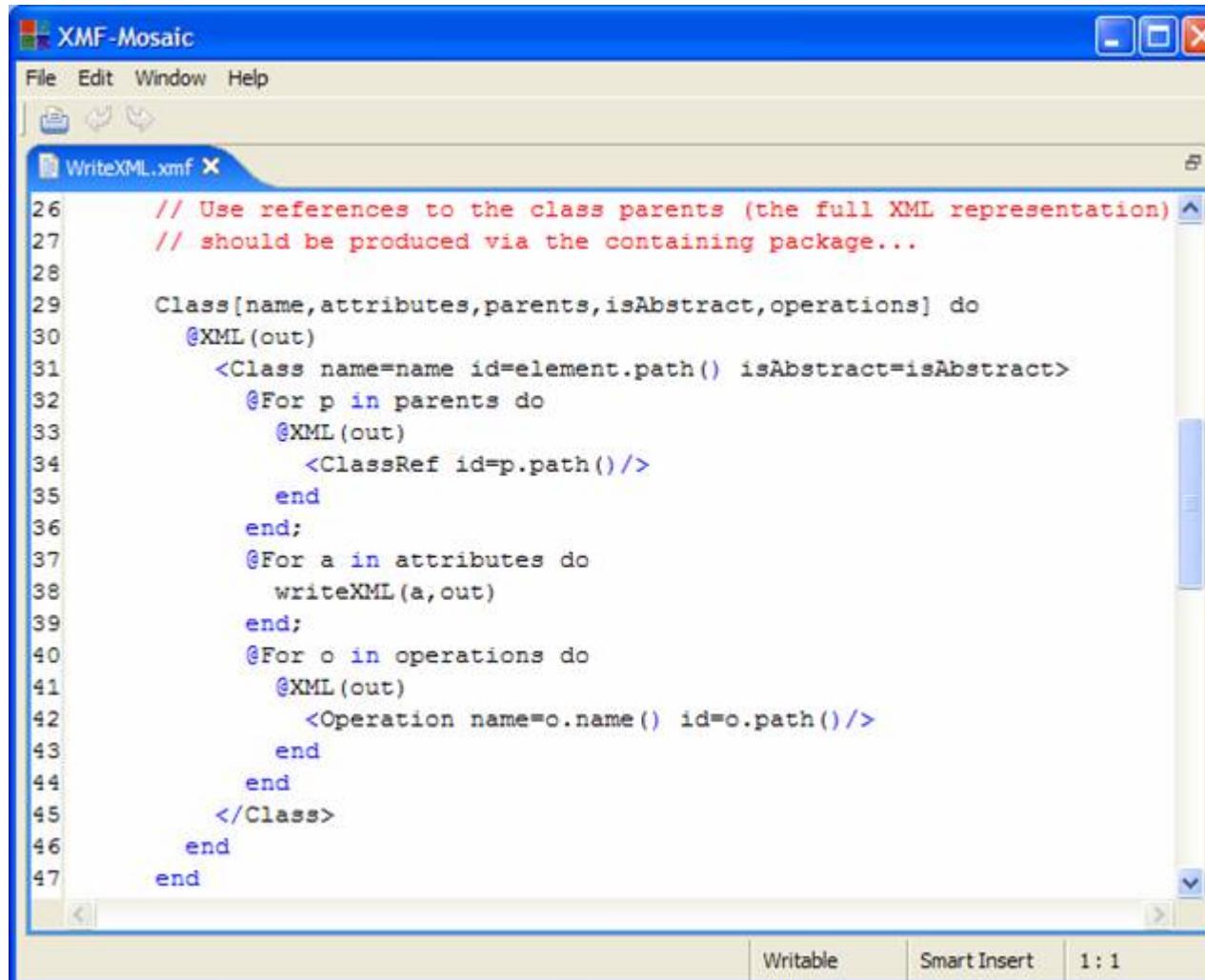
The first pattern in line 13 matches against a package. The XML pattern starting in line 14 specifies some XML to be sent to the supplied output channel (out).

Line 15 defines some XML output. The output pattern is specified in the same format as an XML element containing a tag and some attributes. The tag name and the attribute names are all literal names (you can also use strings or expressions in parentheses). The values of attributes are expressions. When a pattern is evaluated, the character are written to the supplied output channel; any expressions are evaluated and the corresponding values are transformed to strings.

The body of the element pattern ranges over lines 16 – 21. Element bodies can be nested elements or program code. In 16 – 21 there is program code that loops through the packages and classes bound in line 13 and calls writeXML to produce the appropriate output.

The package pattern is complete at line 22 where the terminating tag is output.

Classes are output as follows:



The screenshot shows the XMF-Mosaic editor interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". A toolbar with icons for file operations is visible. The main window displays a file named "WriteXML.xmf" with the following XML code:

```

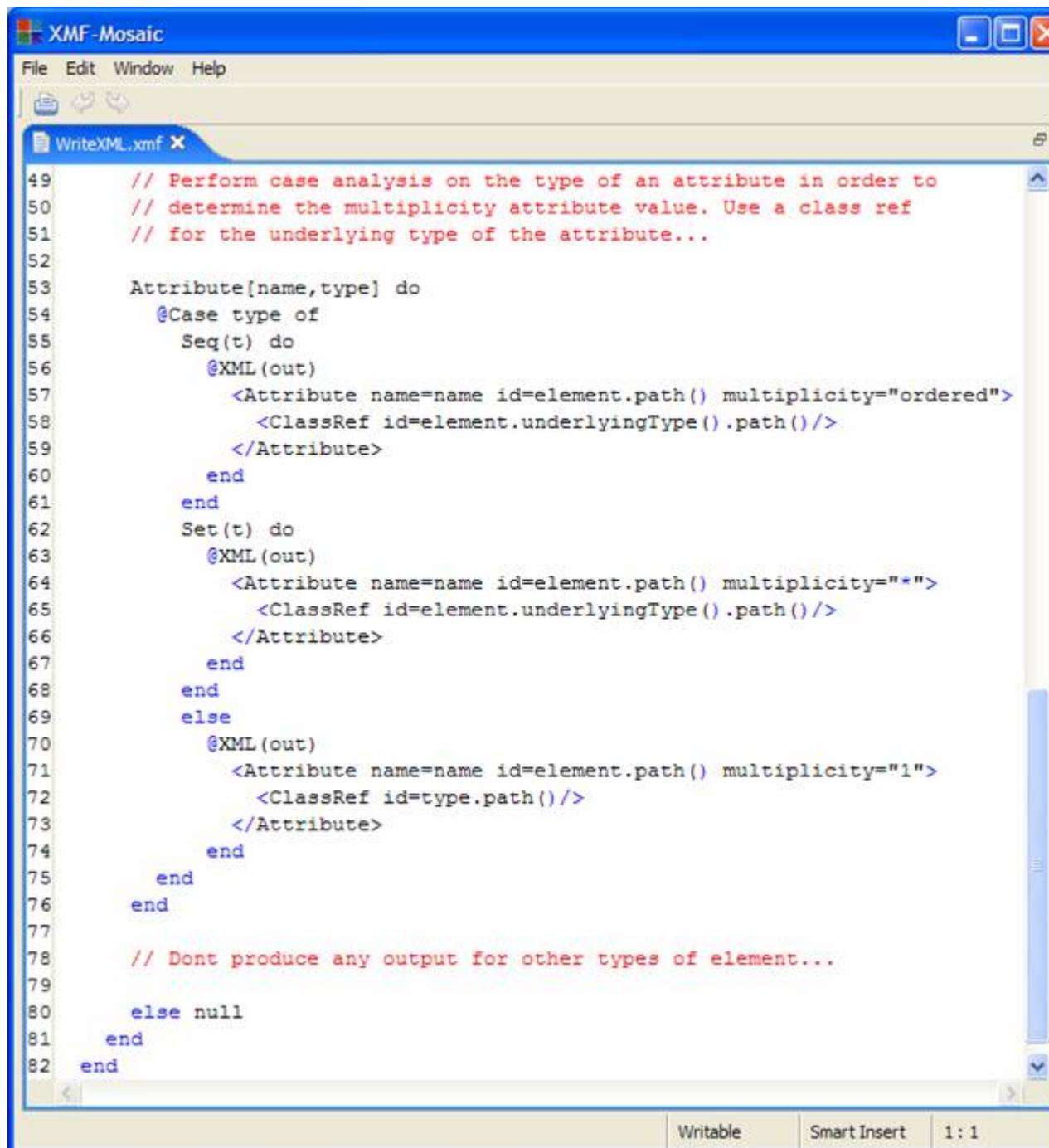
26     // Use references to the class parents (the full XML representation)
27     // should be produced via the containing package...
28
29     Class[name,attributes,parents,isAbstract,operations] do
30         @XML(out)
31             <Class name=name id=element.path() isAbstract=isAbstract>
32                 @For p in parents do
33                     @XML(out)
34                         <ClassRef id=p.path() />
35                     end
36                 end;
37                 @For a in attributes do
38                     writeXML(a,out)
39                 end;
40                 @For o in operations do
41                     @XML(out)
42                         <Operation name=o.name() id=o.path() />
43                     end
44                 end
45             </Class>
46         end
47     end

```

The code uses Ecore XML patterns to generate XML output. Line 33 uses a pattern to produce a classifier reference for the parents of a class, which is nested inside the pattern started in line 30.

The example above shows that XML pattern directed output can be nested. Line 33 uses a pattern to produce a classifier reference for the parents of a class; this is nested inside the pattern started in line 30.

Attributes are represented in XML as an element with a nested class reference for the type. The multiplicity of the attribute type is encoded in the attribute:

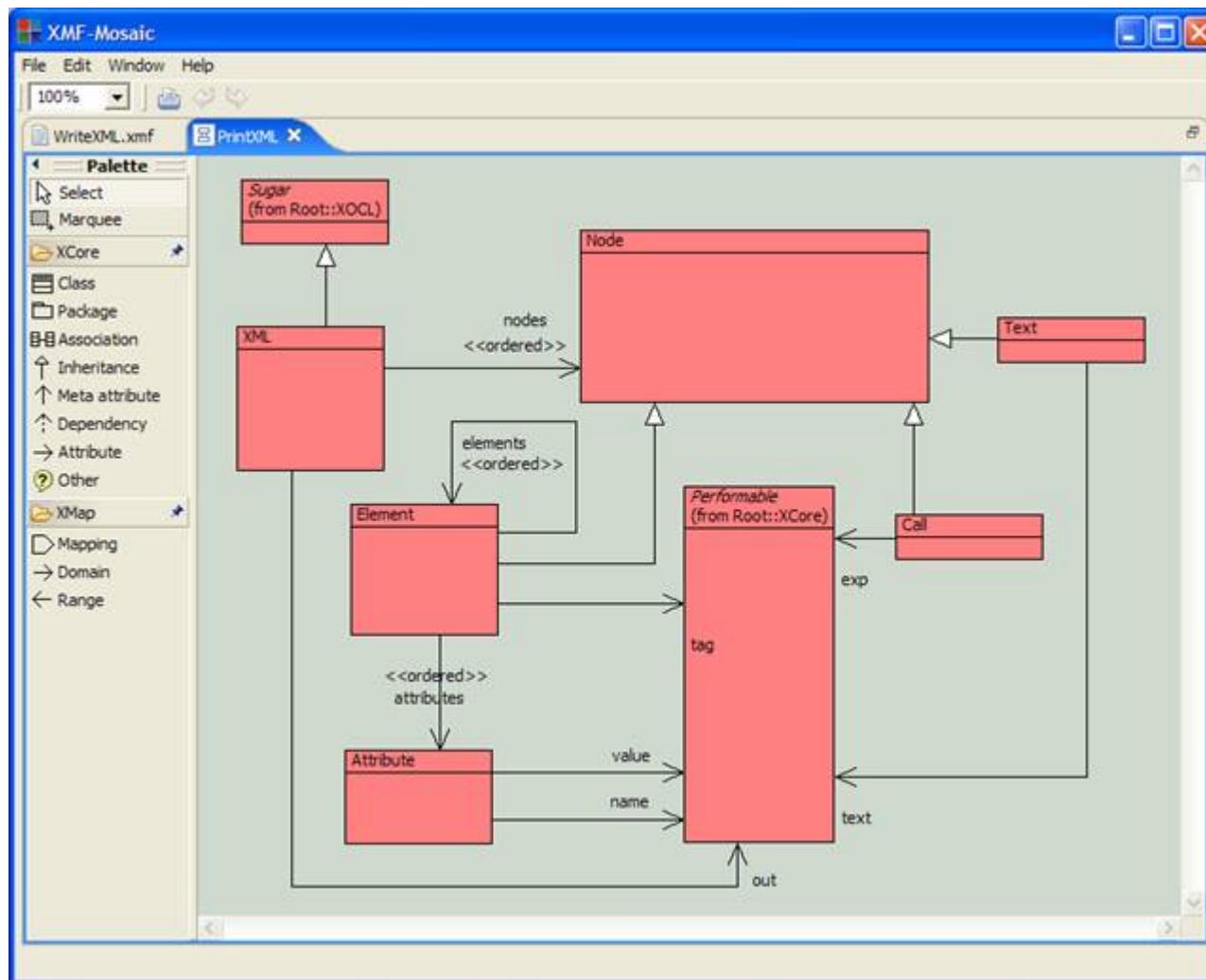


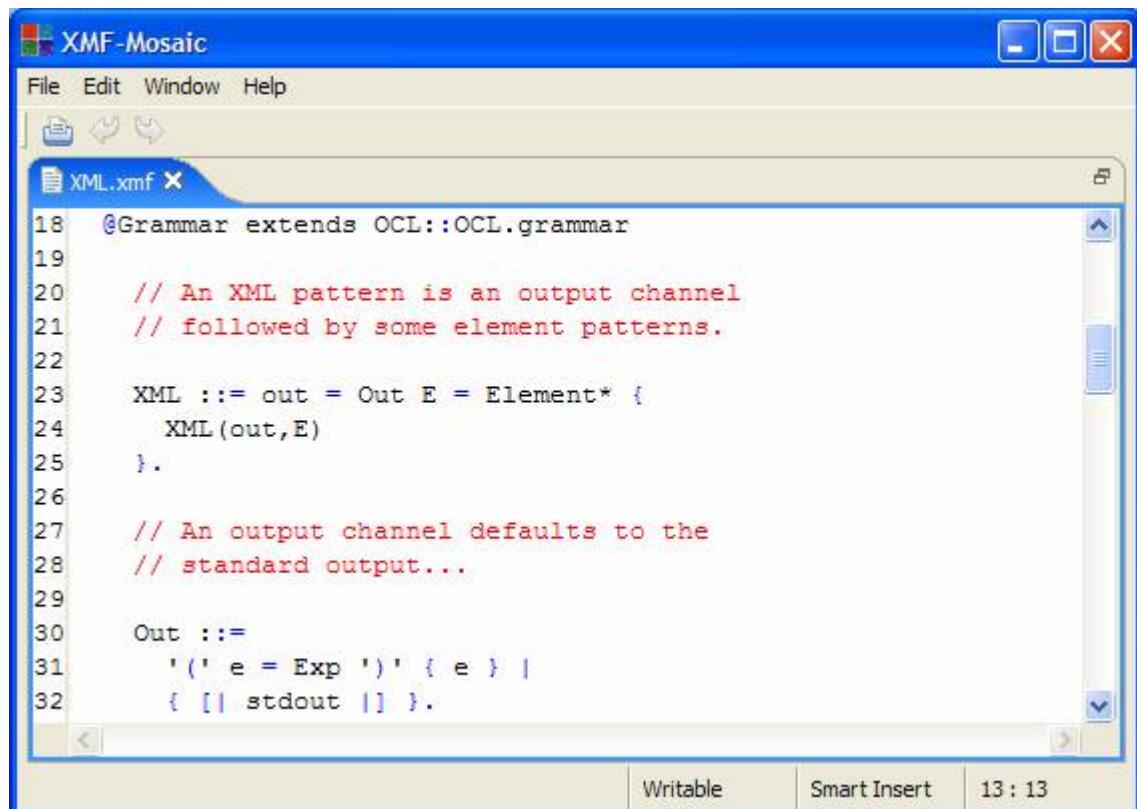
The screenshot shows the XMF-Mosaic editor interface. The title bar reads "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main window displays a code editor titled "WriteXML.xmf". The code is written in a domain-specific language (DSL) for generating XML. It uses numbered lines (49 to 82) and various control structures like "do", "end", and "else". The code performs case analysis on attribute types to generate XML elements like <Attribute>, <ClassRef>, and <Element>. The code is color-coded for syntax highlighting. At the bottom of the editor, there are status indicators: "Writable", "Smart Insert", and a file path "1:1".

```

49      // Perform case analysis on the type of an attribute in order to
50      // determine the multiplicity attribute value. Use a class ref
51      // for the underlying type of the attribute...
52
53      Attribute[name,type] do
54          @Case type of
55              Seq(t) do
56                  @XML(out)
57                      <Attribute name=name id=element.path() multiplicity="ordered">
58                          <ClassRef id=element.underlyingType().path() />
59                      </Attribute>
60                  end
61              end
62              Set(t) do
63                  @XML(out)
64                      <Attribute name=name id=element.path() multiplicity="*" />
65                          <ClassRef id=element.underlyingType().path() />
66                      </Attribute>
67                  end
68              end
69          else
70              @XML(out)
71                  <Attribute name=name id=element.path() multiplicity="1">
72                      <ClassRef id=type.path() />
73                  </Attribute>
74              end
75          end
76      end
77
78      // Dont produce any output for other types of element...
79
80      else null
81  end
82 end

```

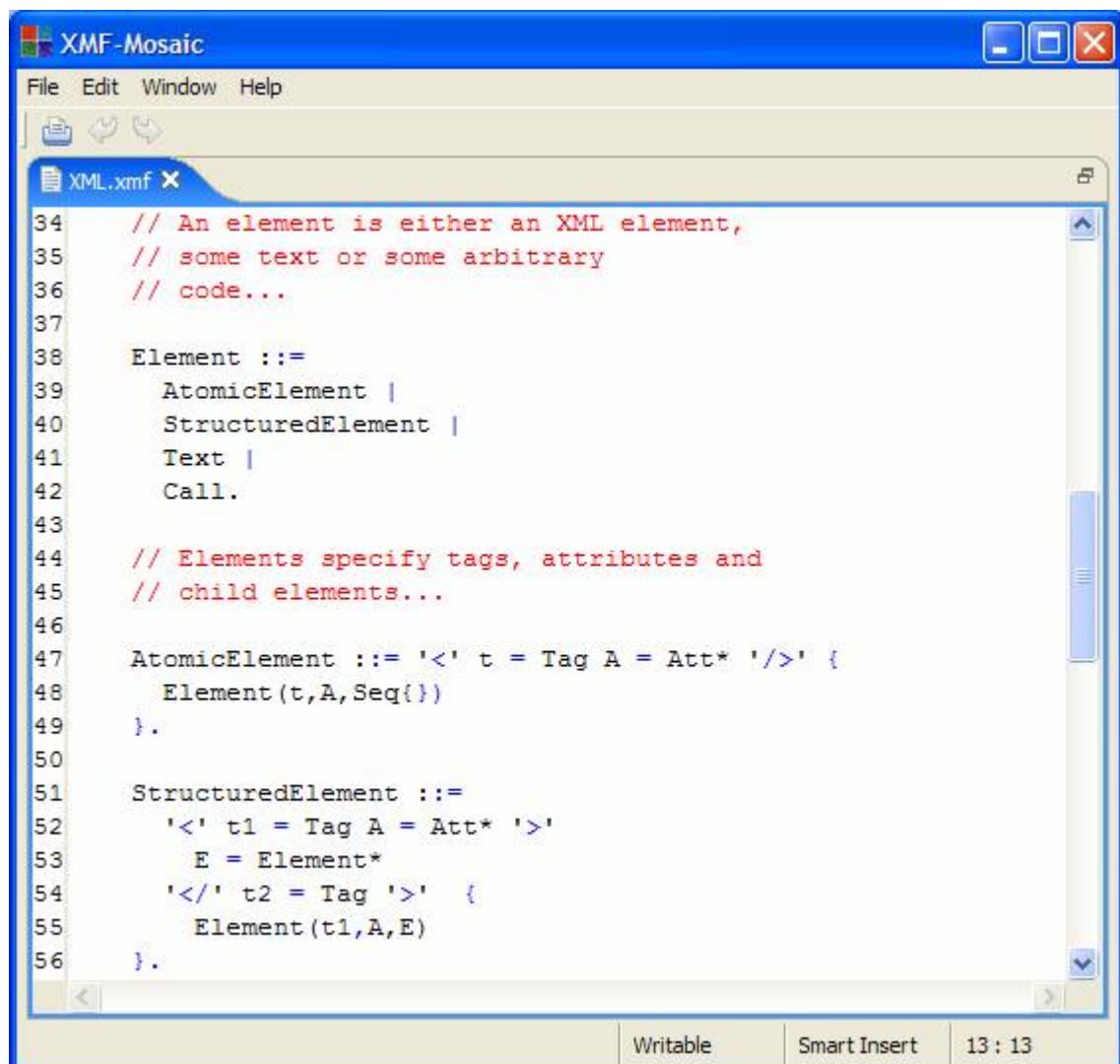




The screenshot shows the XMF-Mosaic editor interface. The title bar reads "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for new file, open file, save file, and close file. The main window displays a file named "XML.xmf" with the following content:

```
18 @Grammar extends OCL::OCL.grammar
19
20 // An XML pattern is an output channel
21 // followed by some element patterns.
22
23 XML ::= out = Out E = Element* {
24     XML(out,E)
25 }.
26
27 // An output channel defaults to the
28 // standard output...
29
30 Out ::=
31     '(` e = Exp ')` { e } |
32     { [] stdout [] }.
```

The status bar at the bottom right shows "Writable", "Smart Insert", and the time "13 : 13".



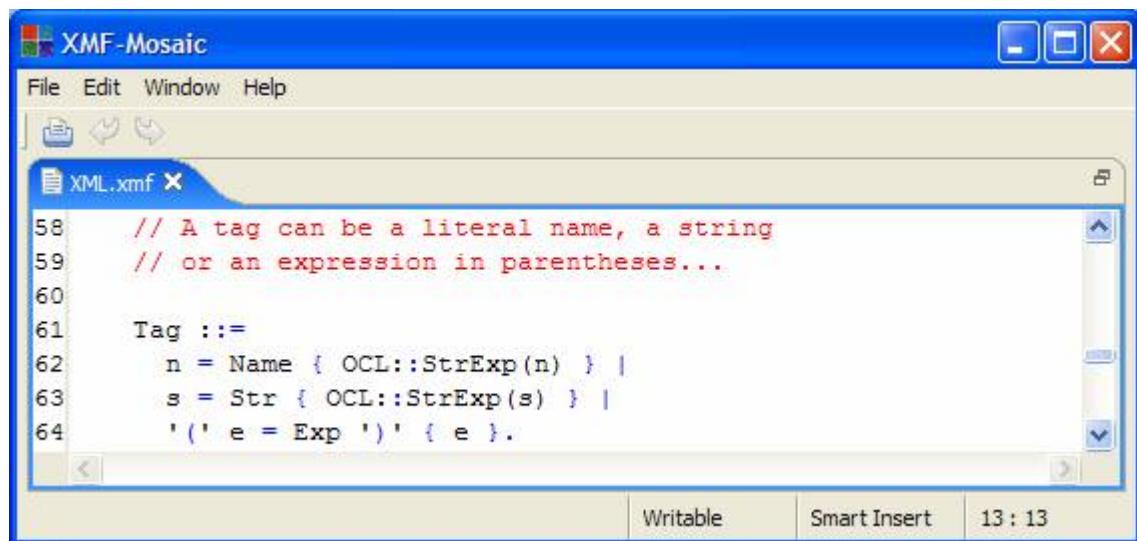
The screenshot shows the XMF-Mosaic interface with a single open editor window titled "XML.xmf". The code in the editor defines an OCL script for XML elements. The script uses the following grammar rules:

```

34  // An element is either an XML element,
35  // some text or some arbitrary
36  // code...
37
38  Element ::= 
39      AtomicElement | 
40      StructuredElement | 
41      Text | 
42      Call.
43
44  // Elements specify tags, attributes and
45  // child elements...
46
47  AtomicElement ::= '<' t = Tag A = Att* '/>' {
48      Element(t,A,Seq{})
49  }.
50
51  StructuredElement ::= 
52      '<' t1 = Tag A = Att* '>' 
53      E = Element*
54      '</' t2 = Tag '>' {
55          Element(t1,A,E)
56      }.

```

The status bar at the bottom right indicates the text is "Writable", has "Smart Insert" enabled, and the time is "13:13".



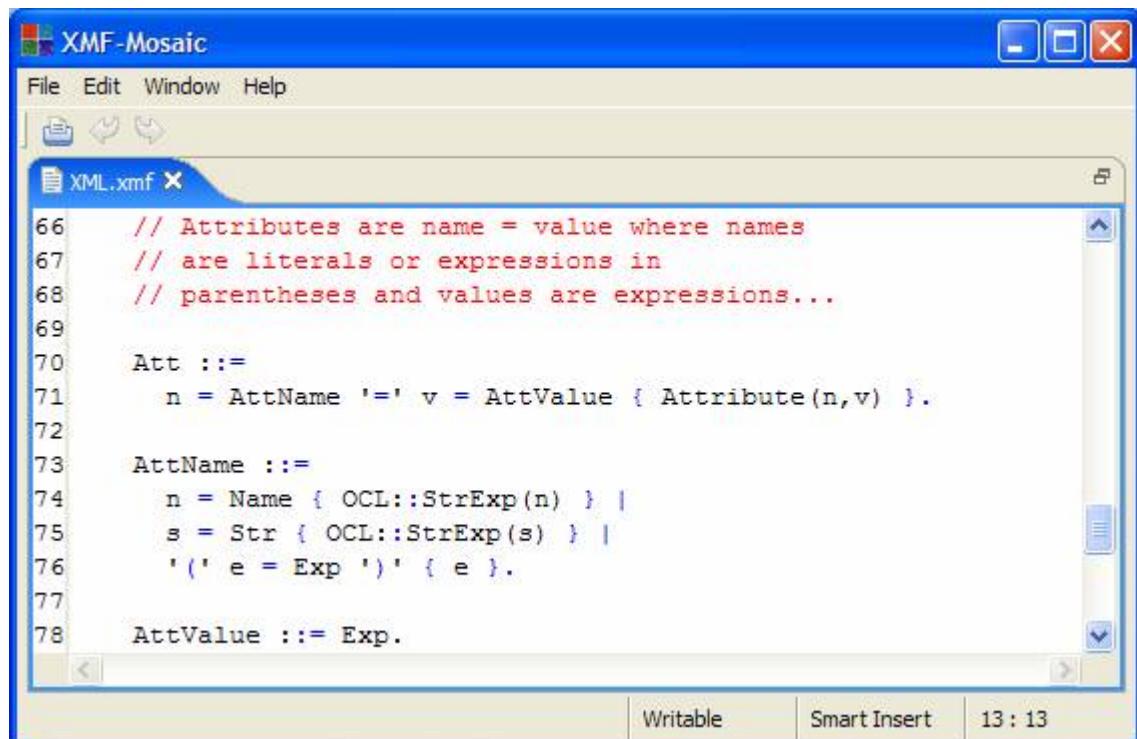
The screenshot shows the XMF-Mosaic interface with a single open editor window titled "XML.xmf". The code in the editor defines an OCL script for XML tags. The script uses the following grammar rule:

```

58  // A tag can be a literal name, a string
59  // or an expression in parentheses...
60
61  Tag ::=
62      n = Name { OCL::StrExp(n) } |
63      s = Str { OCL::StrExp(s) } |
64      '(' e = Exp ')' { e }.

```

The status bar at the bottom right indicates the text is "Writable", has "Smart Insert" enabled, and the time is "13:13".



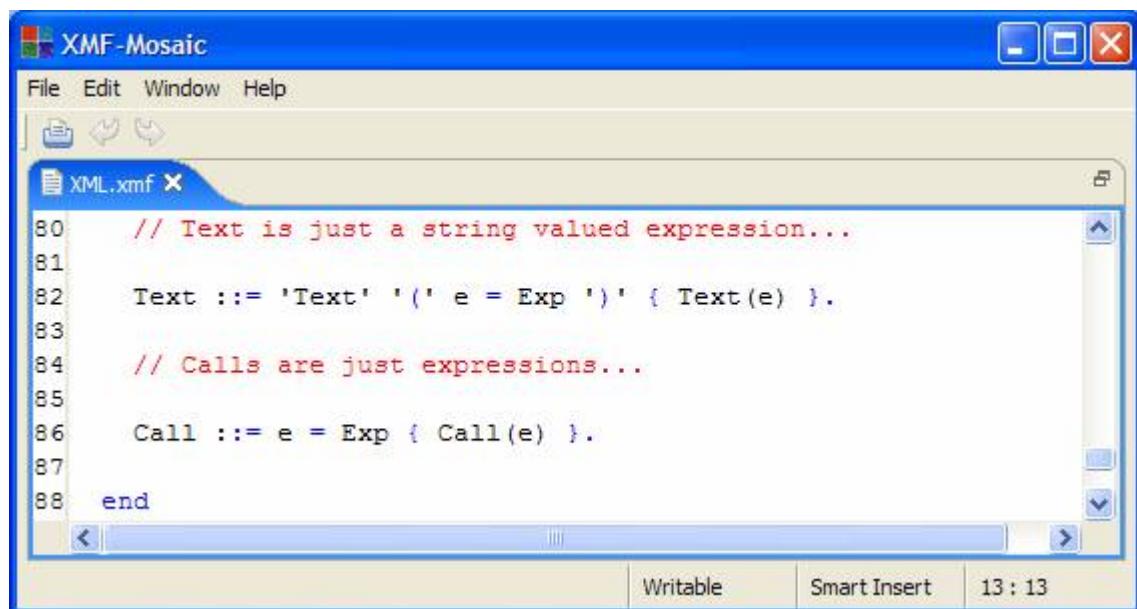
The screenshot shows the XMF-Mosaic editor interface with a blue title bar and window frame. The menu bar includes File, Edit, Window, and Help. A toolbar with icons for file operations is visible above the main area. The main window contains a text editor titled "XML.xmlf" with the following code:

```

66 // Attributes are name = value where names
67 // are literals or expressions in
68 // parentheses and values are expressions...
69
70 Att ::==
71     n = AttName '=' v = AttValue { Attribute(n,v) }.
72
73 AttName ::==
74     n = Name { OCL::StrExp(n) } |
75     s = Str { OCL::StrExp(s) } |
76     '(' e = Exp ')' { e }.
77
78 AttValue ::= Exp.

```

The status bar at the bottom right shows "Writable", "Smart Insert", and the time "13:13".



The screenshot shows the XMF-Mosaic editor interface with a blue title bar and window frame. The menu bar includes File, Edit, Window, and Help. A toolbar with icons for file operations is visible above the main area. The main window contains a text editor titled "XML.xmlf" with the following code:

```

80 // Text is just a string valued expression...
81
82 Text ::= 'Text' '(' e = Exp ')' { Text(e) }.
83
84 // Calls are just expressions...
85
86 Call ::= e = Exp { Call(e) }.
87
88 end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and the time "13:13".

XML Output Channels

Introduction

Basic XML Output

Object Formatters

Name Space Formatters

Saving Models as XML

Raising Events

Deploying Java

Introduction

Deploying Models

Deploying Parsers

Deploying Factories

Chapter 21. XOCL

Introduction

The basic technology that underpins XMF-Mosaic is a programming language called XOCL. The programming language is very similar to standard object-oriented languages such as Java or C#. In addition, XOCL is a meta-programming language meaning that it provides facilities for inspecting and controlling its own behaviour. This is the key to supporting flexible tool development. Tools defined in XOCL can inspect and interpret other tools defined in XOCL. Tools defined in XOCL can generate other tools defined in XOCL, and can generate tools defined in other languages such as Java.

This document is a technical description of XOCL. It defines all aspects of the language and provides a complete definition of the programming constructs and what they do. This document is not a description of how to use XOCL. For that you should read the example Walkthroughs in the Bluebook.

The document is structured as follows: the basic architecture of XOCL is described in terms of what it does and how programs are represented at the lowest level; the basic data types provided by XOCL are defined; XOCL program structure is defined including conditional constructs, binding constructs, errors and looping. Finally, the interface of each basic data type is defined.

Purpose

XOCL is a programming language that is intended to support powerful programming over meta-data. In achieving this aim, the following objectives have been addressed:

- *Efficiency.* XOCL provides a programming language that is used on primarily data intensive applications (as opposed to real-time applications, safety critical applications or applications that require intensive numeric processing). XOCL will process large data sets (several tens of thousands of objects) using average memory and average processor resources.
- *Extensibility.* XOCL is designed to be highly extensible. New language features can be added to XOCL by defining an XBNF grammar for the new feature and the rule to process the new syntax structures. In many cases, new syntax structures can be translated to basic XOCL; this provides a powerful macro-facility for defining declarative language constructs.
- *Dynamic.* XOCL is designed to be highly configurable and easy to modify at run-time. Although XOCL is a compiled language, new definitions can be introduced at any time during execution and existing definitions can be modified at run-time.
- *Meta-circularity.* XOCL is designed to allow existing language features to be extended and modified. XOCL understands its own rules of execution. New tools can introduce modification to these rules. XOCL can process its own syntax; the XOCL compiler and interpreter is written in XOCL.
- *Standards.* XMF-Mosaic aims to make standards available wherever these are appropriate. The environment provides a MOF-like meta-modelling language and a UML-like modeling language. The XML facilities of XMF can be used to import and export XMI encoded data. The language XOCL is based on the UML Object Constraint Language.
- *Conventional.* XOCL aims to provide a language that is as familiar to users as possible whilst achieving the aim of being a powerful basis for tool generation. The basic language features of XOCL will be familiar to users of standard object-oriented programming languages such as Java.
- *Complete.* XOCL provides a complete solution to tool construction. XOCL is not a scripting language (i.e. a lightweight language used as glue for programs written in other languages). XOCL provides features that support the implementation of industrial strength tools, including sophisticated data structures, error handling and a variety of input/output mechanisms.

Language Basics

XOCL programs consist of a collection of units (text files). Each unit can be compiled and loaded separately; however, there are restrictions on the order of compilation and loading multiple units that must be followed; these are explained in this section. A unit is processed by loading and then evaluating it. Loading involves syntax analysis.

A unit has entries in the following order:

- Parser declarations. A parser declaration is processed by the syntax analysis phase. Typically a declaration imports a collection of grammars that define new language constructs. Each language construct in the rest of the unit must be declared before it is used. Basic XOCL is imported by default.
- NameSpace imports. Global variables must be imported before they can be used. Each global variable is defined in a name space. Importing the name space makes all the names it defines available in the rest of the unit. The names in Root and XCore are imported by default.
- Commands and definitions. The rest of the unit is a sequence of any valid XOCL syntax as defined in terms of the parser declarations and imports that precede it. Definitions take the form of the keyword context followed by a name and then an expression whose value is a named element. The name must reference a name space. The effect of a definition is to add the named element to the named name space. Commands take the form of XOCL syntax followed by a semi-colon (;).
- A typical unit has the form shown on the right. All components are optional but, if present, must occur in the order shown.

```
// Comments at the head of the file...

parserImport <PATH>;
// More parser imports...

// Comments here...

import <PATH>;
// More imports...

// Comments here...

context <PATH>
<NAMEELEMENT>

// More defs...

<COMMAND>
// More commands...
```

When a unit is loaded, its syntax is analysed with respect to the parser declarations. If the syntax is legal then the unit is evaluated. Evaluation processes each name space import in turn. Each import is performed in the context of all preceding imports. If the imports are successful then the rest of the unit is evaluated in the context of the imported names. Definitions are evaluated by constructing the named element, referencing the name space and adding the named element to the name space. Commands are evaluated; any result produced by the commands is discarded.

The effect of evaluating a unit is the result of adding all definitions to their name spaces and any side-effects produced by the commands.

Note that although the basic unit of execution is the text file, XOCL is a meta-language and can be used to process definitions and expressions at run-time. This makes XOCL ideal for handling expressions typed at a command interpreter and definitions typed in forms as part of a user interface.

Overview of Syntax

XOCL is a textual language. When a unit is loaded its syntax is processed prior to evaluation. The first stage of syntax analysis is to recognize lexical tokens. A lexical token is a sequence of characters read from the input starting with a non-whitespace character. This section provides an overview of the token types:

- Integer tokens start with a numeric character and continue up to, but not including, the first non-numeric character.
- Parentheses (and).
- Braces { and }.
- Dot .
- At @.
- Comma ,.
- Special tokens are those that do not consist of an alpha-numeric character and are not listed elsewhere. Examples of special tokens are infix operators such as * and +, arrow -> and quasi-quotes [| and |].
- Name tokens start with an alpha-character and continue up to, but not including, the first non-alpha-numeric character.
- Symbol tokens start with a ‘ and end with the following ‘.
- String tokens start with a “ and end with the following “. Within strings the following escape characters are useful:
 - Newline \n
 - Tab \t
 - Return \r
 - String quote \”

Basic Data Types

XOCL is an object-oriented language that runs on the Virtual Machine (XVM) and uses a library of class definitions (written in XOCL) called XCore. Most data types in XMF are represented as instances of XCore classes; however, there is a sub-set of XCore, that is fundamental to XOCL and its execution on XVM. These classes are the basic data types necessary to run XOCL. They are listed here. Their interface definitions are given in subsequent sections of this document.

- Booleans. A boolean value is either true or false. Booleans are used to control branches in execution.
- Clients. XMF-Mosaic can act as a client that connects to and communicates with an external server. The connection is constructed and maintained through a client value.
- Channels. XOCL performs input and output through channels. StandardInputChannel and StandardOutputChannel define interfaces that are implemented by a wide variety of concrete classes that support different forms of input and output; for example: file i/o.
- Daemons. A daemon is a value that can be attached to an object in XOCL that monitors the state of the object. A daemon is activated when the object changes state.

- Integers. Positive and negative integers in the range -24^2 to +24^2.
- Floats. 64 bit floating point numbers.
- Null. The undefined value.
- Objects. An object is a data value with slots. Each slot has a name and a value. All data in XMF is represented in on the formats in this list. Most data in XMF is represented as objects. All types (classes) in XMF are objects and all types have their own types.
- Operations. An operation has a name, arguments and a body. The body of the operation is XOCL program code. When the operation is invoked, it is supplied with values for the arguments and it performs its body. An operation invocation returns the value produced by the body. Operations are the basis for all execution in XOCL. Operations are proper values in the seinse that they can be passed as argument values and stored in slots.
- Sequences. XOCL provides lisp-like lists which it calls sequences. A sequence is either a cons pair with a head and a tail or is the empty sequence. Cons pairs have state such that the head and the tail may be updated. Sequences provide a very flexible and efficient way of organising collections of data that change over time.
- Sets. XOCL provides a data type for representing sets. A set is like a sequence except that it has no state, its elements are unordered and it cannot contain duplicates. Although sets can be defined in terms of objects and sequences, they are provided as a basic data type for efficiency reasons (on the rare occasions that very large sets are necessary).
- Strings. A string is a sequence of character codes. Two strings are equal when they have the same sequence of character codes. A string has no state.
- Symbols. A symbol is a string with state. Two symbols are equal when they are the same data value in computer memory. Symbols are very similar to strings except they facilitate efficient name lookup in dictionaries. Named elements in XCore use symbols.
- Tables. A table associates keys with values. Keys have hash codes that make lookup and update efficient in tables.
- Threads. XOCL is multi-threaded. Each thread is a separate unit of computation that can share global data with other threads. XOCL multi-threading is provided to allow XMF to monitor interactions with multiple clients (although could be used for a variety of tasks). It is not expected that there would be more than 10 – 20 threads active at any given time.
- Vectors. Vectors are fixed size, indexable sequences of values.

Program Constructs

As described above, XOCL programs consist of units. Each unit contains parser declarations followed by name space imports and then a sequence of commands and definitions. Both commands and definitions contain general XOCL expressions; an expression is the basic component of an XOCL language construct. All XOCL language features are evaluated to produce a value; in addition some features produce a side effect as a result of evaluation. Expressions fall into a small number of different categories depending on how they are evaluated. This section provides an overview of XOCL program features and their evaluation mechanisms.

Self Evaluating Expressions

The simplest type of XOCL expression is self evaluating, these are often referred to as constants and include integer, string, Boolean and float literals. In addition, the empty sequence Seq{} and the empty set Set{} are self evaluating.

Variables and Update

A name is an XOCL expression that evaluates to a value depending on the association that the name has in the current context. In general, names refer to variable locations that contain values and are often referred to as variables. XOCL supports three types of variable: local, global and slot. Examples of local variables are operation parameters and let-introduced names. Examples of global variables are names that have been imported from name spaces. Slot variables refer to the state of the currently executing object.

Slot variables may be qualified or unqualified. A qualified reference includes the object that contains the slot followed by dot (.) and the name of the slot. An unqualified reference is just the name of the slot: such a reference assumes the object containing the slot to be self.

Global variable references occur may be qualified or unqualified. A qualified reference references the global variable via some or all of its containing name spaces; this is similar to a path in a file system. For example P::Q::V refers to the variable V in the name space Q which itself is contained in the name space P. The name space P is assumed to be available at the point of reference because it is imported. An unqualified reference to a global variable does not include any containing name spaces: they must be imported at the point of reference.

Variables may be updated using the := operator. The left hand operand must be a variable and the right hand operator must be an expression. The expression is evaluated and the resulting value is placed in the variable location. Slot updates must use qualified slot variables. Global updates must use qualified global variables.

The following class definition contains examples of all types of variable reference and update:

```
context P
  @Class C extends D, Q::E
    @Attribute v : Element end
    @Operation getV():Element
      v
    end
    @Operation setV(newV:Element)
      P::lastGoodV := self.v;
      self.v := newV
    end
    @Operation lastGoodV():Element
      import P
      in lastGoodV
    end
  end
end
```

Line 1 is an unqualified global reference to the name space P. Line 2 contains an unqualified global reference to D and a qualified global reference to E. Line 5 is an unqualified slot reference to v. Line 8 contains a qualified global update for lastGoodV and a qualified slot reference to v. Line 9 contains a qualified slot update for v and a local reference to newV. Line 13 contains an unqualified global reference to P and line 14 contains an unqualified global reference (since it has been imported) to lastGoodV.

Local variables are typically created when values are supplied as arguments to an operation or when local definitions are executed. The association between the Local variable name and the value persist for the duration of the operation definition or the execution of the body of the local block. In both cases, as the name suggests, variable values can change by side effect.

Local variables are established when arguments are passed to an operation or using a let expression. In both cases the variable can be referenced in the body of the expression, but not outside the body. In both cases the variables can be updated using v := e. Suppose we require an operation that takes two integers and returns a pair where the head is the smallest integer and the tail is the other integer:

```
context Root
@Operation orderedPair(x,y)
let min = 0;
    max = 0
in if x < y then min := x else min := y end;
    if x > y then max := x else max := y end;
Seq{min | max}
end
end
```

The definition of orderedPair shows how a let expression can introduce a number of variables (in this case min and max). If the let - bindings are separated using ; then the bindings are established in-parallel meaning that the variables cannot affect each other (i.e. the value for max cannot refer to min and vice versa). If the bindings are separated using then they are established in-series meaning that values in subsequent bindings can refer to variables in earlier bindings, for example:

```
context Root
@Operation orderedPair(x,y)
let min = if x < y then x else y end then
    max = if min = x then y else x end
in Seq{min | max}
end
end
```

Calling Operations

An operation is invoked by directly invoking it on some argument values or by sending an object a message for which the operation has been defined as the handler. The two types of invocation use the same underlying evaluation machinery, but are syntactically very different. In both cases there is only one parameter passing mechanism: values are passed into the operation and values with state can be modified by the operation. Variables cannot be modified in the sense of Pascal or Ada out parameters. The parameter passing mechanism is directly equivalent to that of Java.

Operations are applied to arguments using the conventional procedure call notation:

```
p(arg1,arg2,...,argn)
```

where p is an XOCL expression that evaluates to an operator and each argi is an XOCL expression that evaluates to produce an argument value.

Messages are sent to a value using the conventional method invocation notation:

```
o.m(arg1,arg2,...,argn)
```

where o is an XOCL expression that evaluates to produce a value (the receiver), m is a name and argi are the operation parameter expressions. To calculate the operation that is invoked, XOCL finds the type of o, and calculates its operator precedence list (OPL). The OPL contains all the operators defined by the type of o with the name m in order of most recently defined (with respect to inheritance) first. The first operation in the OPL is invoked.

The receiver of a message defines the value of self in the body of the operation invocation. In the case of direct operator application, the value of self is that which was in scope when the operator was defined. The value of self can be changed by sending the operation an invoke message:

```
p.invoke(o,Seq{arg1,arg2,...,argn})
```

Infix Operators

XOCL supports infix notation for the usual arithmetic and boolean operators. In most cases the evaluation of an infix expression will evaluate both sub-expressions in left to right order and then perform the appropriate function on the results. The exceptions are:

- p andthen q evaluates p, if p is false then q is not evaluated, otherwise this behaves as p and q.
- p orelse q evaluates p, if p is true then q is not evaluated otherwise this behaves as p or q.

Prefix Operators

XOCL supports one prefix operator – not. Note that prefix – for negative numbers is not supported (use infix – as in: 0-n).

Sequencing

XOCL expressions may produce side effects, either by changing the state of values, producing output or consuming input. Expression evaluation is controlled using the semi-colon (;) operator that sequences evaluation: e1; e2 is evaluated by evaluating e1 and then evaluating e2. The result of the sequenced expression is the value of e2. This operator associates to the right.

Special Forms

An XOCL special form is an expression that has its own evaluation rules that do not necessarily follow the usual rules of sub-expression evaluation followed by operator call. This section lists the special forms and defines their evaluation rules.

- Conditional expressions are defined using standard if...then...else...end notation. The test following the if is evaluated, if it is true then the consequent expression is evaluated otherwise the alternative expression is evaluated.
- A definition occurs in a program unit as the keyword context followed by a name space and a n expression whose value is a named element. The definition causes the named element to be added to the name space when it is performed. The only place this type of definition can occur is at the top level of an evaluation unit.
- Local variables are introduced using a let expression: let bindings in body end. The bindings introduce local variable names and their initial values. The body is an XOCL expression whose evaluation may reference the local variables. The variables are discarded when the evaluation is complete and the value of the let expression is the value of its body.
- A set or sequence is constructed using the form Set{exp1,exp2,...,expn} and Seq{exp1,exp2,...,expn}. The sub-expressions are evaluated and the result is a set or sequence containing the values.
- XOCL is based on OCL which provides a number of convenient types of iteration expression. These have the form: s->iterOp(v | body) where s is an expression whose value is a set or sequence, iterOp is one of forAll, exists, select, collect or reject, v is a variable name and body is an expression. Depending on the iteration operation the body expression is evaluated in a context where v is bound to successive elements selected from s. A special form of iteration expression is also provided: s->iterate(v w = e | body) where w is initialized to e and body is evaluated with v bound to successive elements of s; at each evaluation w is rebound to the value produced by body.
- OCL provides a notation for invoking sequence and set operations: s->collOp and s->collOp(arg1,arg2,...,argn). These are retained in XOCL for compatibility with OCL, but are unnecessary since method calling notation works just as well.
- A name space is imported for the scope of an expression e by import n in e end. All of the names defined in n are available in the expression e.
- XOCL provides an exception mechanism for handling errors and other exceptional circumstances during execution. An exception is created and thrown from the point of error using a throw e command (an example of an XOCL construct that does not behave like a standard expression). The expression e is evaluated and can produce any value. XCore provides a collection of exception classes that can be used and extended. The exception is caught by the most recently established try

... catch(x) ... end expression. The exception can be handled in by the catch or may be re-thrown to the next try expression.

Quasi-Quotes

XOCL is a language that provides features for meta-programming. A key feature of meta-programming is language processing: the ability to construct, transform and manipulate programs. XOCL provides quasi-quotes [| and |] for this purpose. Quasi-quotes can be placed around any XOCL expression e (including XOCL language extensions), the value of the resulting expression [| e |] is the syntax structure for e (as opposed to the value of e). For example: the value of [| x + 1 |] is not an integer, but a syntax structure representing an addition expression whose operator is + and whose left operand is a variable with name x and whose right operand is the constant 1.

Within quasi-quotes any expression surrounded by drop braces < and > is expected to produce syntax that is inserted into the syntax structure constructed by the quotes. For example:

```
let e = [| x + 1 |]
in [| y * <x> |]
end
```

line 1 creates a syntax structure called e that is then inserted into the syntax structure as the right operand of the multiplication expression in line 2. The resulting structure is equivalent to:

```
[| y * (x + 1) |]
```

Quasi quotes and drop braces are very important when meta-programming with languages. They make constructing code templates very easy. Code templates are used to construct XOCL language extensions and to facilitate mappings that translate from one language to another.

The Meta Character @

XOCL is an extensible language. New language features are easy to add to XOCL to provide new expression types, new command and new declarative definitions. A big problem of current programming languages is that they cannot be extended very easily. Thus, it is not possible to construct abstraction mechanisms that capture the key features of the application domain. This results in large amounts of program code that is difficult to maintain. Examples of such patterns are: new types of looping construct; a state machine definition; the observer pattern; new types of interface definition; coding standards; specific types of classes such as containers.

XOCL provides a novel feature that allows new language constructs to be conveniently added to the language. Once they are added, the new features are seamlessly integrated with all other language constructs. The language feature that supports this is the meta-character @.

When an XOCL unit is processed it is syntactically analysed and then evaluated. During the syntax analysis phase, if an expression starts with the character @ then the analyzer is informed that the characters up to and including the corresponding end are to be processed by a language extension. A language extension is defined by providing a grammar that parses the characters from the @ to the end; the grammar is expected to synthesize and return a syntax construct. Grammars can be attached to classes and the name following the @ should be a class that defines a grammar. A syntax construct is required to implement the Performable interface that allows the second phase of XOCL unit processing. These two features: grammars and Performable are all that is necessary to allow XOCL to be arbitrarily extended with new constructs.

Typically, syntax classes implement the Performable interface by translating to existing XOCL syntax classes (that already implement the Performable interface). This form of language extension is termed sugar and the processif translation is called desugaring.

XOCL makes extensive use of the @ feature. Looping constructs such as While, Find and For are all implemented as extensions of the basic XOCL language. Definitions for classes, packages, constraints and operations are defined using @.

Documentation

Comments can be inserted into XOCL in two ways. The first way is using the standard // <line of text> line comment and /* <para> */ paragraph comments. These can only be used in text files and are ignored by the parser.

The second way is using the @Doc <some text> end syntax. The text within an @Doc is ignored for the purposes of executing the statement. However, the comment is parsed and is stored as part of the XOCL statement. This means that documentation attached to an XOCL expression can be processed as part of a model. Note a semicolon is not required if the documentation is placed at the head of the code body (otherwise it is). Here is an example of its use:

```
context Float
    @Operation max(other:Integer):Integer
        @Doc
            Compares a float with other and returns
            the maximum value.
        end
        if self > other
        then self
        else other
        end
    end
```

Note, there are currently some limitations to the characters that can be used in an @Doc expression. In particular, the string end must not occur in a @Doc body, even if it is embedded in a word. To resolve this, a separator can be used, e.g. send becomes sen-d.

Error Handling

When an error occurs in XOCL, the source of the error throws an exception. The exception is a value that, in general, contains a description of the problem and any data that might help explain the reason why the problem occurred. An exception is thrown to the most recently established handler; intermediate code is discarded. If no handler exists then the XMF VM will terminate. In most cases, the exception is caught by a user-defined handler or, for example in the case of the XMF console, a handler established by the command interpreter.

When an exception is caught, the handler can inspect the contents of the exception and decide what to do. For example it may be necessary to re-throw the exception to the next-most recently established handler, since it cannot be dealt with. On the other hand, it is usual to catch the exception, print a message, patch up the problem, or just give up on the requested action.

Exception handling is performed by a try catch expression with the following form:

```
try
    normal
catch(x)
    abnormal
end
```

The try expression evaluates the normal sub-expression that may perform an arbitrary amount of computation. If the normal expression completes without throwing an exception then its value is returned as the value of the try expression. Alternatively, an exception is thrown at some point during the execution of normal:

```
throw e
```

where e is an expression that produces the exception. An exception can be any value, but is typically an instance of the class Exception or one of its sub-classes. An exception is handled by the most

recently established try expression. The evaluation of normal is terminated and execution passes to line 3. The variable x is bound to the thrown exception and abnormal is evaluated. If abnormal evaluates without throwing an exception then the value of abnormal is the value returned by the try expression. Otherwise, abnormal throws an exception (perhaps x) that is handled by the next most recently established try expression.

Control Statements

Flow of control in XOCL is controlled by the constructs defined in this section. All the basic control statements are supported, including If, Case, While and For. In addition, XOCL supports convenient notations for iterating over collections.

If

An if expression is used to choose between alternative expressions based on the outcome of a boolean expression. The different forms of if expression are outlined as follows:

```
1  if test
2  then consequent
3  end
4
5  if test
6  then consequent
7  else alternative
8  end
9
10 if test1
11 then consequent1
12 elseif test2
13 then consequent2
14 elseif test3
15 then ...
16 else alternative
17 end
```

Lines 1-3 show that an if can be used to construct a guarded expression. If the test in line 1 produces true then the consequent in line 2 is evaluated and produces the value of the if expression otherwise nothing is evaluated and the value of the if expression is undefined.

Lines 5 – 8 show how an if expression is used to choose between two different expressions. If the expression at line 5 produces true then the expression at line 6 is evaluated and produces the value of the if expression. Otherwise the expression at line 7 is evaluated and produces the value of the if expression.

Lines 10 – 17 show how the elseif keyword can be used to avoid deeply nested if expressions.

Case

A case expression is used to dispatch on a sequence of values. The simplest form of a case expression is shown as follows:

```
@Case e of
  p1 do
    e1
  end
  p2 do
    e2
  end
```

```
...
pn do
    en
end
else x
end
```

The case expression is evaluated as follows. The expression e at line 1 is evaluated to produce a value v. If v matches pattern p1 then e1 is evaluated and produces the value of the case expression. Otherwise if value v matches p2 then e2 is evaluated and produces the value of the case expression. Matching continues sequentially until a pattern pi matches and the corresponding expression ei produces the value of the case or the else-clause is reached. If the else-clause is reached then x is evaluated and produces the value of the case expression. Note that the else-clause is optional.

Note that the semantics of pattern matching is covered elsewhere in this document. The simplest form of pattern matching is against basic values:

```
@Case x of
    1 do
        // The value of x is 1...
    end
    2 do
        // The value of x is 2...
    end
    ...
    100 do
        // The value of x is 100...
    end
    else self.error("Illegal value for x: " + x.toString())
end
```

In general a case statement can match over a sequence of values:

```
@Case e1,e2,...,en of
    p11,p12,...,p1n do
        b1
    end
    p21,p22,...,p2n do
        b2
    end
    ...
    pm1,pm2,...,pmn do
        bm
    end
    else x
end
```

Caselnt

An integer case expression dispatches on the value of an integer. An integer case expression is much more efficient than a case expression because it compiles to a simple indexed dispatch in XVM. The following is an example that returns true when the value of char is an alpha-numeric character code:

```
@CaseInt[ 256 ] char of
    " "->at(0) do false end
    "\n"->at(0) do false end
    "\t"->at(0) do false end
    "\r"->at(0) do false end
    "("->at(0) do false end
```

```
" ")->at(0) do false end
"\\"->at(0) do false end
"0"->at(0) to "9"->at(0) do true end
"A"->at(0) to "z"->at(0) do true end
else false
end
```

Line 1 states that the range of values for char is 0 to 255. Each case arm uses s->at(0) to include a literal character code. Note that the values in case arms may be any XOCL expression but that the expression must have a value at compile time. A case arm has values to be matched before the keyword do. The values can take the following forms:

- A constant integer valued expression.
- A range x to y where x and y are constant integer valued expressions.
- A sequence of integer case values separated with comma (,).

TypeCase

In a pure OO program it should not be necessary to test the type of a value, all computation should be performed using message passing. However, few if any applications are pure and testing the type of a value at run-time is occasionally necessary. All values in XOCL can respond to an isKindOf message whose single argument is an XCore classifier. The return is a Boolean value determining whether or not the value is considered, with respect to inheritance, to be an instance of the classifier. Where there are many such tests on a value it is more declarative and potentially more efficient to use the TypeCase special form:

```
@TypeCase(x)
  type1 do
    exp1
  end
  type2 do
    exp2
  end
  ...
  typen do
    expn
  end
  else d
end
```

The expression x is evaluated in line 1 to produce a value. Each type (lines 2,5 etc) is a name or path referencing a classifier. If the value of x is of type type1 then exp1 is evaluated and produces the value of the case expression. Otherwise, checking proceeds case by case until the optional default case us evaluated. A value v is of type t when it is a direct instance of t or when the direct type of v inherits from t.

While

A While loop performs an action until a condition is satisfied (note a named element may use a symbol for its name so we ensure the name is a string using the toString operation):

```
context Root
  @Operation findElement(N:Set(NamedElement),name:String)
    let found = null
    in @While not N->isEmpty do
      let n = N->sel
      in if n.name().toString() = name
```

```
        then found := n
        else N := N->excluding(n)
    end
end
end;
found
end
end
```

For

It is often the case that While loops are used to iterate through a collection. This pattern is captured by a For loop:

```
context Root
@Operation findElement(N:Set(NamedElement),name:String)
let found = null
in @For n in N do
    if n.name().toString() = name
    then found := n
    end
end;
found
end
end
```

In general a For loop @For x in S do e end is equivalent to the following While loop:

```
let forColl = S;
isFirst = true
in @While not forColl->isEmpty do
    let x = forColl->sel
    in forColl := forColl->excluding(x);
    let isLast = forColl->isEmpty
    in e;
    isFirst := false
end
end
end
```

Where the variables forColl, isFirst and isLast are scoped over the body of the loop e. These can be useful if we want the body action to depend on whether this is the first or last iteration, for example turning a sequence into a string:

```
context Seq(Operation)
@Operation toString()
let s = "Seq{"
in @For e in self do
    s := s + e.toString();
    if not isLast then s := s + "," end
end;
s + "}"
end
end
```

A For loop may return a result. The keyword do states that the body of the For loop is an action and that the result of performing the entire loop will be ignored when the loop exits. Alternatively, the keyword produce states that the loop will return a sequence of values. The values are the results returned by

the loop body each time it is performed. For example, suppose we want to calculate the sequence of names from a sequence of people:

```
context Root
  @Operation getNames(people:Seq(Person)):Seq(String)
    @For person in people produce
      person.name
    end
  end
```

The keyword `in` is a For-loop directive. After `in` the loop expects one or more collections. The `in` directive supports multiple variables. This feature is useful when stepping through multiple collections in sync, as in:

```
context Root
  @Operation createTable(names:Seq(String),addresses:Seq(String),telNos:Seq(String))
    @For name,address,telNo in names,addresses,telNos produce
      Seq{name,address,telNo}
    end
  end
```

A For-loop also has a `where` option. This allows the elements to be filtered. The following only iterates over names of persons that are not excluded.

```
context Root
  @Operation getNames(people:Seq(Person),excluded:Seq(Person)):Seq(String)
    @For person in people where not excluded->includes(person) do
      person.name
    end
  end
```

Find

The Find construct is used to perform an action in terms of an element of a collection. Typically we want to find the first element in a collection that satisfies a given predicate and to perform an action. If no value exists that satisfies the predicate then we optionally want to perform some other action. This construct captures that pattern.

As an example, imagine we want find a transition in a statemachine whose `targetName` is equal to the name of a state, and that when we do, we want to return the `State` object with that name, or if one does not exist, return the initial state. The following operation `getTarget()` implements this using Find.

```
@Operation getTarget(name: String):XCore::Element
  let next = null
  in @Find(t,transition)
    when t.targetName = name
    do next := states->select(s |
      s.name = name)->sel
    else next := states->select(s |
      s.name = startName)->sel
  end;
  next
end
```

Iterators

There are number of important collection operations that can be used to iterate over elements in a collection to produce a result. Each iterator operation has an iterator variable, which is matched with

each element of the collection in turn. An iterator expression is then evaluated against the variable to produce a result.

There are four main iterator operations: select, collect, reject and iterate.

Select

```
C->select(c | <expression with c>)
```

Select filters the elements in the collection. The element c is added to the result if and only if the <Expr> is true. The following example will return only those values of the set that are greater than 5:

```
Set{1,2,3,4,5,6,7,8,9,10}->select(i | i > 5)
```

In the context of a class model, select statements are very useful for filtering collections of objects subject to some property. For instance, this operation returns all states of a StateMachine that match the name x and which are not an initial state of the StateMachine (see Part 2 of this Reference Manual):

```
context StateMachine
@Operation statesForName(x):Set(Element)
    states->select(s | s.name = x and not startName = x)
end
```

Reject

```
C->reject(c | <expression with c>)
```

Reject is the converse of select. It rejects any elements where <Expr> returns true.

As an example, the following rejects all the values of the sequence of characters that are alphabetically greater than the character "a":

```
Seq{"a","b"}->reject(x | x > "a")
```

Collect

```
C->collect(c | <expression with c>)
```

Collect iterates over the elements in the collection and collects together the result of evaluating the expression. The following collect expression add 1 to each element in the set:

```
Set{1,2,3,4,5,6,7,8,9}->collect(x | x+1)
```

Here is an example that collects together the names of the states in a state machine: context StateMachine

```
@Operation allStateNames()
    states->collect(s | s.name)
end
```

Iterate

```
C->iterate(c acc = <expression> | <expression with c and acc>)
```

The iterate operation is the most fundamental and general of the iterator operations. All other loop operations can be described as a special case of this operation. Like select and collect, an iterate has a iterator variable. In addition, it has an accumulator variable, which is given an initial value. The result of the iterate operation is the result of iterating over all the elements in the collection. For each successive element, the body expression is evaluated using the previous result of the accumulator. Here is an example of using iterate to add together a collection of numbers:

```
Seq{1,2,3,4,5,6,7,8,9,10}->iterate(e s=0 | s + e)
```

This sum() operation makes use of an iterate to sum a collection of integers:

```
context Seq(Element)
@Operation sum()
    self->iterate(e sum=0 | sum + e)
end
```

Assignment

In XOCL, values may be assigned to variables. The assignment keyword is “`:=`”. This should not be confused with “`=`” which returns a Boolean value.

The following code shows an assignment being used to set the name of a State by setting name to be x:

```
Context Java
@Operation createState (n:String):State
    let s = State() in
        s.name := n;
        s
    end
end
```

An assignment can be used to set global variables or local variables. When setting the value of a local attribute of a class, `self.attname` must be used as the target of the assignment. For instance, the following operation will set the name of the State to be x.

```
context State
@Operation setName(n)
    self.name := n
end
```

However, the following code will not be legal:

```
context State
@Operation setName(n)
    name := n
end
```

Pattern Matching

XOCL provides a powerful pattern matching language. This greatly simplifies the writing of pattern matching operations that would otherwise require a very imperative style of programming.

Patterns and Pattern Matching

A pattern is matched against a value. The pattern match may succeed or fail in a given matching context. A matching context keeps track of any variable bindings generated by the match and maintains choice points for backtracking if the current match fails.

Pattern matching can be viewed as being performed by a pattern matching engine that maintains the current pattern matching context as its state. The engine state consists of a stack of patterns to be matched against a stack of values, a collection of variable bindings and a stack of choice points. A choice point is a machine state. At any given time there is a pattern at the head of the pattern stack and a value at the head of the value stack. The machine executes by performing state transitions driven by the head of the pattern stack: if the outer structure of the pattern matches that of the value at the head of the value stack then:

- 0 or more values are bound.
- 0 or more choice points are added to the choice point stack.
- 0 or more component patterns are pushed onto the pattern stack.
- 0 or more component values are pushed onto the value stack.
- If the machine fails to match the pattern and value at the head of the respective stacks then the most recently created choice point is popped and becomes the new machine state. Execution continues until either the pattern stack is exhausted or the machine fails when the choice stack is empty.

Pattern Categories

This section describes the different categories of pattern. The semantics of matching are defined informally in terms of a general description and example definitions involving the pattern.

Variables

A variable pattern consists of a name, optionally another pattern and optionally a type. The simplest form of variable pattern is just a name, for example, the formal parameter `x` is a variable pattern:

```
let add1 = @Operation(x) x + 1 end in ...
```

Matching a simple variable pattern such as that shown above always succeeds and causes the name to be bound to the corresponding value. A variable may be qualified with a type declaration:

```
let add1 = @Operation(x:Integer) x + 1 end in ...
```

which has no effect on pattern matching. A variable may be qualified with a pattern as in `x = <Pattern>` where the pattern must occur before any type declaration. Such a qualified variable matches a value when the pattern also matches the value. Any variables in the pattern and `x` are bound in the process.

Constants

A constant pattern is either a string, an integer, a boolean or an expression (in the case of an expression the pattern consists of [followed by an expression followed by]). A constant pattern matches a value when the values is equal to the constant (in the case of an expression the matching process evaluates the expression each time the match occurs). For example:

```
let fourArgs = @Operation(1,true,"three",x = [ 2 + 2 ]) x end in ...
```

is an operation that succeeds in the case:

```
fourArgs(1,true,"three",4)
```

and returns 4.

Sequences

A sequence pattern consists of either a pair of patterns or a sequence of patterns. In the case of a pair:

```
let head = @Operation(Seq{head | tail}) head end in ...
```

the pattern matches a non-empty sequence whose head must match the head pattern and whose tail must match the tail pattern. In the case of a sequence of patterns:

```
let add3 = @Operation(Seq{x,y,z}) x + y + z end in ...
```

the pattern matches a sequence of exactly the same size where each element matches the corresponding pattern.

Constructors

A constructor pattern matches an object. A constructor pattern may be either a by-order-of-arguments constructor pattern (or BOA-constructor pattern) or a keyword constructor pattern. A BOA-constructor pattern is linked with the constructors of a class. It has the form:

```
let flatten = @Operation(C(x,y,z)) Seq{x,y,z} end in ...
```

where the class `\tt C` must define a 3-argument constructor. A BOA-constructor pattern matches an object when the object is an instance of the class (here `\tt C` but in general defined using a path) and when the object's slot values identified by the constructor of the class with the appropriate arity match the corresponding sub-patterns (here `xy` and `z`). A keyword constructor pattern has the form:

```
let flatten = @Operation(C[name=y,age=x,address=y]) Seq{x,y,z} end in ...
```

where the names of the slots are explicitly defined in any order (and may be repeated). Such a pattern matches an object when it is an instance of the given class and when the values of the named slots match the appropriate sub-patterns.

Conditions

A conditional pattern consists of a pattern and a predicate expression. It matches a value when the value matches the sub-pattern and when the expression evaluates to true in the resulting variable context. For example:

```
let repeat = @Operation(Seq{x,y}) when x = y Seq{x} end in ...
```

Note that the above example will fail (and probably throw an error depending on the context) if it is supplied with a pair whose values are different.

Sets

Set patterns consist of an element pattern and a residual pattern. A set matches a pattern when an element can be chosen that matches the element pattern and where the rest of the set matches the residual pattern. For example:

```
let choose = @Operation(S->including(x)) x end in ...
```

which matches any non-empty set and selects a value from it at random. Set patterns introduce choice into the current context because often there is more than one way to choose a value from the set that matches the element pattern. For example:

```
let chooseBigger = @Operation(S->including(x),y where x > y) x end in ...
```

Pattern matching in `chooseBigger`, for example:

```
chooseBigger(Set{1,2,3},2)
```

starts by selecting an element and binding it to `x` and binding `S` to the rest. In this case suppose that `x = 1` and `S = Set{2,3}`. The pattern `y` matches and binds 2 and then the condition is applied. At this point, in general, there may be choices left in the context due to there being more than one element in the set supplied as the first parameter. If the condition `x > y` fails then the matching process jumps to the most recent choice point (which in this cases causes the next element in the set to be chosen and bound to `x`). Suppose that 3 is chosen this time; the condition is satisfied and the call returns 3. The following is an example that sorts a set of integers into descending order:

```
context Root
@Operation sort(S)
@Case S of
  Set{} do Seq{} end
  S->including(x)
```

```
    when S->forAll(y | y <= x)
        do Seq{x | Q} where Q = sort(S)
    end
end
end
```

Sequences

Sequence patterns use the infix + operator to combine two patterns that match against two subsequences. For example the following operation removes a sequence of 0's occurring in a sequence:

```
context Root
@Operation remove0s(x)
@Case x of
    (S1 when S1->forAll(x | x <> 0)) +
    (S2 when S2->forAll(x | x = 0)) +
    (S3 when S3->forAll(x | x <> 0))
    do S1 + S3
end
end
end
```

Syntax

Syntax patterns consist of expressions within quasi-quotes [] and []. The quotes are a short-hand for writing out the equivalent constructor patterns. Syntax patterns provide a powerful way of constructing syntax mappings where the pattern is defined in terms of concrete syntax rather than the equivalent abstract syntax structures. Consider an operation that extracts the body of a letexpression:

```
context Root
@Operation getBody([| let x = value in body end |])
    body
end
```

Unfortunately, this will not work as you may expect since the syntax pattern states that the operation expects to be supplied with a let expression that consists of exactly one binding and where the body of the expression is the variable whose name is body. We wish to place patterns within the syntax construct that match against specific elements of the abstract syntax structure. To do this we use pattern-unquotes:

```
context Root
@Operation getBody([| let <| bindings |> in <| body |> end |])
    body
end
```

Within a syntax pattern the unquotes <| and |> are used to surround patterns that are to be matched against the abstract syntax structures occurring at that point in the supplied expression. In the example above, bindings is bound to the sequence of bindings in the let and body is bound to the body. The following example shows an operation that calculates the free variables occurring in an expression. The expression is limited to a small number of XOCL expressions:

```
context Root
@Operation FV(e)
@Case e of
    [| let <| Seq{} |> in <| e |> end |] do
        FV(e)
    end
    [| let <| Seq{ ValueBinding(v,e1) | bs } |>
        in <| e2 |>
```

```
    end
| ] do
  FV([ | let <bs> in <e2> end| ]) ->excluding(v)
end
[ | if <| e1 |>
  then <| e2 |>
  else <| e3 |>
  end
| ] do
  FV(e1) + FV(e2) + FV(e3)
end
[ | <| e1 |> = <| e2 |> | ] do
  FV(e1) + FV(e2)
end
Var[name=n] do
  Set{n}
end
end
end
```

The following call:

```
FV[ | let x = 10; y = 20 in if x = y then z else a end end | ])
```

produces the set

```
Set{a,z}
```

Pattern Contexts

Patterns may be used in the following contexts:

Operation Parameters. Each parameter in an operation definition is a pattern. Parameter patterns are useful when defining an operation that must deconstruct one or more values passed as arguments. Note that if the pattern match fails then the operation invocation will raise an error. Operations defined in the same class and with the same name are merged into a single operation in which each operation is tried in turn when the operation is called via an instance of the class. Therefore in the following example:

```
@Class P
  @Operation f(Seq{}) 0 end
  @Operation f(Seq{x | t}) x + self.f(t) end
end
```

an instance of P has a single operation f that adds up all the elements of a sequence.

Case Arms. A case expression consists of a number of arms each of which has a sequence of patterns and an expression. A case expression dispatches on a sequence of values and attempts to match them against the corresponding patterns in each arm in turn. For example, suppose we want to calculate the set of duplicated elements in a pair of sets:

```
context Root
  @Operation dups(s1,s2)
  @Case s1,s2 of
    s1->including(x),s2->including(y) when x = y do
      Set{x} + dups(s1,s2)
    end
    s1->including(x),s2 do
      dups(s1,s2)
    end
    s1,s2->including(y) do
```

```
        dups(s1,s2)
    end
Set{},Set{} do
    Set{}
end
end
end
```

In XMap transformations, as described in the XMap manual.

Data Type Operations

Boolean

A value of the Boolean type can either be true or false.

Operators

All the usual Boolean operators are provided.

Table 21.1.

<i>Operators</i>	<i>Syntax</i>	<i>Result Type</i>
Or	a or b	Boolean
And	a and b	Boolean
negation	not b	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean

Examples

```
not true
balance > 0 or balance < 100
x <> y implies x.name <> y.name
```

The following example shows the definition of the logical operator and. It uses an if statement to test whether other is a Boolean value and if it is returns the conjunct of self and other. An error is reported if the type is incorrect.

```
context Boolean
@Operation booland(other)
    if other.isKindOf(Boolean)
    then
        self and other
    else
        self.error("Boolean::booland expects a boolean " + other.toString())
    end
end
```

Channels

XOCL provides channels to perform input and output. The channel classes are defined in the IO package and are defined in a separate document. This section provides a very brief overview of input and output using channels.

Standard Input and Output

XOCL provides two channels as global variables defined in the name space Root. The output channel stdout is used to send characters to the standard output. The input channel stdin reads characters from the standard input. In both cases characters are represented as integer character codes in the range 0 to 255. Characters are written on output channels by outch.write(c) and read from input channels by inch.read().

Formatting Output

XOCL provides a convenient means for formatting output to channels. The global variable format is bound to a character formatter that is used as an operation by applying it to arguments that control the output of characters. The general form is:

```
format(outch,formatString,args)
```

where the format string is a string of characters controlling how the sequence of argument values is formatted as output to the output channel. A format string is a sequence of characters and format directives. Each character occurring in a format string is written to the output channel in sequence. A format directive starts with the tilde (~) character and controls how the output is written and how the format arguments are consumed and written to the output channel. A typical use of format will print a message followed by a newline. A newline is produced using the format directive ~% as in:

```
format(outch,"hello world~%",Seq{})
```

Note that there are no format arguments in the example above. Where there are no format args, they can be omitted:

```
format(outch,"hello world~")
```

The directive ~S consumes the next format argument, translates it to a string and prints it to the output channel. For example, if we want to print out the source and target of an edge with an arrow between them:

```
format(outch,"~S -> ~S~%",Seq{edge.source,edge.target})
```

Format defines many other directives that are defined in the XMF guide to IO.

File Based Input and Output

The IO package of XMF provides file channels for accessing and updating files. These channels are defined in the XMF guide to IO. A useful special form is provided that hides the details of file IO:

```
@WithOpenFile(inch <- filename)
  // Use inch.read() to read characters from the file.
  // inch.read() returns -1 when EOF is reached.
end

@WithOpenFile(outch -> filename)
  // Use outch in format expressions to write characters to the file.
end
```

In both of the examples above the filename is specified as an expression whose value is a string. The channels used to perform the input and output are automatically closed when the WithOpenFile expression completes. If an IO error occurs during the evaluation of the WithOpenFile then XOCL guarantees that the channels are closed before the error is reported.

Clients

XOCL provides clients that can be used to connect to external servers on ports and then communicate via input output channels with the server.

Daemons

An XOCL daemon is an operation that is attached to an object's slot such that whenever the slot changes state the operation is invoked. The invocation is referred to as firing the daemon. There are several different types of daemon depending on the required firing mode: whether the operation is invoked when any change occurs, when a new value is added to the slot or when a value is removed from the slot.

A new daemon is created using the Daemon constructor:

```
Daemon(id,type,slot,action,persistent,traced,target)
```

Where id is any XOCL value and is used to identify the daemon in a given context; type is an integer determining the firing mode of the daemon; slot is a symbol that names the slot that is being monitored by the daemon (or null if this is not appropriate); action is an operation to be invoked when the daemon fires; persistent is a boolean value determining whether or not the daemon will be saved when an object containing the daemon is saved to a XAR file; traced is a boolean value that determines whether or not tracing information is printed when the daemon fires; target is any XOCL value and provides a means of associating state with a daemon.

The type of the daemon is supplied as an integer value. The types are defined by an enumerated type defined in the class Daemon; the type determines the number of arguments that the action should have. The following table lists the types and provides skeleton operations of the appropriate form for the daemon type:

Table 21.2.

Type	Description	Action Skeleton
XCore::Daemon::ANY	Any change to the object will fire the daemon. The action is supplied with the object, the name of the slot that changes, the new value and the old value of the slot. The value of slot in the construction of the daemon is null.	@Operation(object,slot,newValue,oldValue) ... end
XCore::Daemon::VALUE	Any change to the named slot will fire the daemon.	@Operation(object,slot,newValue,oldValue) ... end
XCore::Daemon::ADD	The slot must contain a set or sequence. When a new value is added to the set or sequence the daemon will fire. The action is supplied with the added value.	@Operation(object,slot,value) ... end
XCore::Daemon::REMOVE	The slot must contain a set or sequence. When a value is removed from the set or sequence the daemon will fire. The action is supplied with the removed value.	@Operation(object,slot,value) ... end

Elements

All XOCL values are instances of the XCore class named Element. The operations defined by this class are available for all values in XMF-Mosaic. The following table shows the essential operations defined by this class. Note that XOCL is a dynamic language and new operations can be added to a class at any time. Adding operations to Element will cause these to be available for all values.

Table 21.3.

copy():Element	Returns a copy of the receiver. By default this returns the receiver. Sub-classes of Element may redefine this appropriately.
equals(other:Element):Boolean	Returns true when the receiver is equal to the argument. By default this is defined to be true when the identify (memory location) of elements with state are the same. Strings are compared character by character. Sets are equal when all elements are equal.
error(reason:String)	A convenient way of raising an error exception.
hashCode():Integer	Returns the code used to index into hash tables.
init(args:Seq(Element)):Element	All elements can be initialised with respect to arguments. See Object for more specific information.
isKindOf(c:Classifier):Boolean	Returns true when the receiver is an instance (direct or otherwise) of the argument. Note that null is considered an instance of everything.
isReallyKindOf(classifier:Classifier):Boolean	True when isKindOf is true and the receiver is not null.
of():Classifier	Returns the direct classifier of the receiver.
send(message:String,args:Seq(Element)):Element	Sends the supplied message with the given arguments to the receiver. Returns the result.
toString():String	Produces a string representation of the receiver. The system uses toString to display XOCL values in all circumstances. It is usual to provide an appropriate definition of toString in all class definitions. This aids debugging systems.
yield()	Halt the current thread and reschedule it.

Integers

This is the data type for integer values. The default value is 0.

Operators

All the usual operators on Integer types are provided.

Table 21.4.

Operator	Syntax	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
less than	a < b	Boolean
more than	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a + b	Integer or Float
minus	a - b	Integer or Float
multiplication	a * b	Integer or Float

divison	a / b	Integer or Float
operation	a.op()	Element

Operations

A variety of integer operations are also provided.

Table 21.5.

abs():Integer	Returns the absolute value of an integer.
add(other:Element):Element	Adds an integer to another integer or a float.
asSeq():Seq(Integer)	Turns an integer into a 24 bit sequence of binary values.
bit(index:Element):Element	Returns the ith bit after converting an integer into a 24 bit sequence of binary values.
byte(index:Integer):Integer	Returns the byte of an indexed by index. Bytes are indexed from 1 (low) to 4 (high).
div(other:Integer):Integer	Integer division returns the number of times an integer can be divided by other a whole number of times.
floor():Integer	Rounds a float down to an integer.
greater(other:Integer):Element	Returns true if an integer is greater than other.
isAlphaChar():Boolean	Returns true if an integer is a valid alphanumeric value.
isLowerCaseChar():Boolean	Returns true if an integer is a valid lower case alphanumeric value.
isNewLineChar():Boolean	Returns true if an integer is the new line alphanumeric value.
isNumericChar():Boolean	Returns true if an integer is a valid numeric alphanumeric value.
isUpperCaseChar():Boolean	Returns true if an integer is a valid upper case alphanumeric value.
isWhiteSpaceChar():Boolean	Returns true if an integer is the white space alphanumeric value.
less(other:Element):Element	Returns true if the integer is lower than other.
lsh(n:Integer):Integer	Left shift bit operation.
max(other:Integer):Integer	Compares an integer with other and returns the maximum value.
min(other:Integer):Integer	Compares an integer with other and returns the minimum value.
mod(other:Integer):Integer	Returns the remainder when an integer is divided by other.
mul(other:Element):Element	Multiples an integer by other.
round():Integer	Rounds a float to the nearest whole integer.
rsh(n:Integer):Integer	Right shift bit operation.
slash(other:Element):Element	Divides an integer by other.
sqrt():Element	Returns the square root of an integer.
sub(other:Element):Element	Subtracts other from an integer.

to(n:Integer):Seq(Integer)	Generates a sequence of integers from self to n.
toString():String	Converts an integer to a string.

Examples

```
13 * 42 = 546
12 > 10 = false
12.max(11) = 12
12.min(11) = 11
12.mod(5) = 2
1.to(3) = Seq{1,2,3}
```

The following example shows an operator definition for factorial. The operator is named fact, takes a single argument n and is defined in the global context Root which means that the name fact is available everywhere:

```
context Root
@Operation fact(n)
  if n = 0
    then 1
    else n * fact(n - 1)
  end
end
```

Another example of a global operation definition is gcd below that computes the greatest common divisor for a pair of positive integers. The example shows that operations can optionally have argument and return types:

```
context Root
@Operation gcd(m:Integer,n:Integer):Integer
  if m = n
    then n
    else
      if m > n
        then gcd(m-n,n)
        else gcd(m,n-m)
      end
    end
  end
```

Floats

The data type for real values. The default value is 0.0.

Operators

Table 21.6.

Operator	Syntax	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
less than	a < b	Boolean
more than	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >=	Boolean

plus	$a + b$	Integer or Float
minus	$a - b$	Integer or Float
multiplication	$a * b$	Integer or Float
division	a/b	Integer or Float
operation	$a.op()$	Element

Operations

Table 21.7.

abs():Float	Returns the absolute value of a float.
add(other:Element):Element	Adds a float to other.
cos():Element	Returns the cosine of a float.
div(other:Element):Element	Divides a rounded float by the result of rounding other.
floor():Element	Rounds a float down to an integer.
greater(other:Element):Element	Returns true if a float is greater than other.
init(args:Element):Element	No Documentation Specified
less(other:Element):Element	Returns true if a float is less than other.
max(other:Integer):Integer	Compares a float with other and returns the maximum value.
min(other:Integer):Integer	Compares a float with other and returns the minimum value.
mod(other:Element):Element	Returns a float modulo other after rounding down to integers.
mul(other:Element):Element	Multiply a float by other.
round():Element	Returns the result of rounding a float down.
sin():Element	Returns the sin() of a float.
slash(other:Element):Element	Divided a float by other.
sqrt():Element	Returns the square root of a float.
sub(other:Element):Element	Subtracts an integer from a float.
toString():String	Converts and integer to a string.

Examples

```
1.1 > 1.01 = true
3.5.round() = 4
3.2.floor() = 3
13.sqrt() = 3.6055512
```

The following operation defines the operation abs(). If the float is a negative number it is subtracted from 0, otherwise the value of the float is returned.

```
@Operation abs():Float
    if self < 0
        then
            0 - self
        else
            self
    end
```

end

Objects

XOCL objects are XCore elements with slots. A slot is an association between a name (symbol) and a value. A slot has state and can be updated. All objects in XMF-Mosaic are instances of classes that inherit from the XCore class Object. The essential operations supported by Object are defined below:

Table 21.8.

addDaemon(d:Daemon)	All objects have a collection of daemons. A daemon is an operation that is invoked whenever a slot of the object is updated
daemons():Seq(Daemon)	Returns the currently defined daemons for the receiver. Daemons are fired when the object changes state and when the objects daemons are active.
daemonsActive():Boolean	Returns whether or not the daemons of this object will be fired when an update takes place.
get(name:String)	Returns the value of the named slot of the receiver. The name may be a string or a symbol. An exception is raised if the receiver has no slot with the given name.
getStructuralFeatureNames():Set(String)	Returns the slot names of the object.
hasDaemonWithId(id:Element):Boolean	Returns true when the receiver has a daemon with the supplied id.
hasSlot(name:Element):Boolean	Returns true when the receiver has a slot with the given name. The name may be a string or a symbol.
init(args:Element)	When an object is initialised, by default we look for a constructor that has the same arity as the supplied arguments. If we find one then it is invoked. This operation causes constructors to be invoked when they are defined for the class of the receiver. If this operation is redefined then you should use super(args) in the sub-class to invoke this operation.
removeDaemon(d:Daemon)	Removes the supplied daemon.
set(name:String,value:Element)	Sets the named slot to the supplied value in the receiver. Raises an exception if the receiver has no slot with the supplied name. The name may be a symbol or a string.
setDaemons(daemons:Seq(Daemon))	Updates the daemons of the receiver to be the supplied sequence.
setDaemonsActive(active:Boolean)	Sets whether or not the daemons of this receiver will be fired when an update takes place

Null

The value null is an instance of the XCore class Null and is considered special in the sense that it can be viewed as the undefined value. It is an instance of all classes in XMF and is the default value of all slots whose type is not a basic type (such as String) or a set or sequence. There is only one null value and it is only equal to itself.

Operations

XOCL operations are used to implement both procedures and functions (queries). An operation has an optional name, some parameters, a return type and a body. Operations are objects with internal state; part of the internal state is the name, parameter information, type and body. Operations also have property lists that can be used to attach information to the operation for use by XOCL programs.

Operations can be created and stored in XMF data items. In particular, operations can be added to name spaces and then referenced via the name space (either where the name space is imported or directly by giving the path to the operation). We have seen many examples of adding operations to the name space called Root in earlier parts of this primer. The syntax:

```
context Root
    @Operation add(x,y) x + y end
```

can occur at the top-level of an XMF source file, compiled and loaded. It is equivalent to the following expression:

```
Root.add(@Operation add(x,y) x + y end);
```

Unlike the context expression, the call to add may occur anywhere in XMF code.

Operations are performed by sending them a message invoke with two arguments: the value of self (or target) to be used in the body of the operation and a sequence of argument values. The target of the invocation is important because it provides the value of self in the body of the operation and supplies the values of the slot-bound variables. The add operation can be invoked by:

```
add.invoke(null,Seq{1,2})
```

It produces the value 3. Note that in this case there is no reference to self or slot-bound variables in the body and therefore the target of the invocation is null. A shorthand for invocation is provided: add(1,2)

However, note that no target can be supplied with the shorthand. In this case the target will default to the value of self that was in scope when the operation was created.

Locally bound variables that are scoped over an operation are available within the body of the operation even though the operation is returned from the lexical context. This is often referred to as closing the local variable into the operation (or closure). This feature is very useful when generating behaviour that differs only in terms of context. Suppose that transition machine states have an action that is implemented as an operation and that the action is to be performed when the state is entered:

```
context StateMachines
    @Class State
        @Attribute name : String end
        @Attribute action : Operation end
        @Constructor(name,action) end
        @Operation enter()
            action()
        end
    end
```

Compiled operations have the following slots:

Table 21.9.

arity	Integer	The number of arguments required by the operation.
codeBox	Element	The XVM instructions.
dynamics	Seq(element)	The global variables available via imports.

globals	Seq(Element)	The global variables available via imports.
isVarArgs	Boolean	Whether this operation can take a variable number of arguments.
properties	Seq(Element)	A sequence of name value pairs.
sig	Seq(Element)	A type signature for the operation.
supers	Seq(Operation)	A sequence of operations headed by the owner of the slot. Used when invoking super.
target	Element	Value used as self within the operation.
traced	Operation	An operation used as a proxy when the operation is traced. (null is not traced).

Operations

Compiled operation defines the following operations:

Table 21.10.

addDaemon(daemon:Element)	Operations have daemons that monitor their slots.
addNameChangedDaemon(d:Element,actionSource:Element)	Creates a daemon to add a daemon that monitors the name of a compiled operation for changes. The args for the daemon are the new name and the old name.
arity():Integer	The number of arguments expected by the operation. Use this rather than reference the slot.
daemons():Seq(Operation)	Returns the daemons currently monitoring the operation.
disassemble():String	Displays the XVM instructions to stdout.
disassemble(out:Element)	Displays the XVM instructions to out.
doc():Element	Any documentation defined for the receiver.
get(name:String):Element	Reference a named slot of the receiver.
getStructuralFeatureNames():Set(String)	Get the slots defined for the receiver.
hasProperty(property:Element):Boolean	Returns true when the receiver has the supplied property.
hasSlot(name:String):Boolean	Returns true when the receiver has the supplied name.
importNameSpace(n:NameSpace)	Imports the supplied name space and its contents to the receiver. If the name space is already imported then no change is made. Otherwise the name space is added as the most specific imported name space.
importNameSpaces(N:Seq(NameSpace))	Imports the sequence of name spaces in the order that they are supplied.
imports(n:NameSpace):Boolean	Returns true when the receiver imports the supplied name space.
imports():Seq(NameSpace)	Returns the sequence of imported name spaces.

isVarArgs():Boolean	returns true when the receiver can be supplied with a variable number of arguments. Use this in preference to referencing the slot.
name():Symbol	Returns the name of the receiver. Use this in preference to referencing the slot.
owner():Element	Returns the owner of the receiver. Use this in preference to referencing the slot.
paramNames():Seq(String)	Returns the sequence of parameter names.
paramTypes():Seq(Classifier)	Returns the sequence of parameter types.
properties():Element	Returns the receivers property list.
property(property:Element):Element	Returns the value of the supplied property.
removeDaemon(daemon:Element)	Removes the supplied daemon.
set(name:String,value:Element)	Sets the supplied slot.
setArity(arity:Element)	Changes the arity (don't use).
setDaemons(daemons:Element)	Sets the daemons.
setDoc(doc:Element)	Sets the documentation.
setName(name:Element)	Sets the name.
setOwner(owner:Element)	Sets the owner.
setProperties(properties:Element)	Sets the properties.
setProperty(property:Element,value:Element)	Sets the property.
setSig(sig:Element)	Sets the type signature.
setSource(source:String)	Sets the source code string.
setSupers(supers:Element)	Sets the supers list.
setTarget(target:Element)	Sets the target.
sig():Seq(Element)	Returns the type signature.
source():String	Returns the source code string.
supers():Seq(Operation)	Returns the supers list.
target():Element	Returns the target.
trace():Operation	Returns the trace operation.
traced():Boolean	True when the receiver is traced.
type():Classifier	Returns the return type.
untrace()	Stops printing trace information when the receiver is invoked.
update(newOp:Element)	Replaces the receiver with the argument. The update is performed in place so that all references to the receiver are also updated.

Strings

A string is a sequence of characters. Literal strings are written in enclosing double string quotes, such as “fred” and “fido”.

Operators

Strings can be compared using equals (=) and can be concatenated.

Table 21.11.

Operator	Syntax	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
concatenate	a + b	String
operation	a.op()	Element

Strings can also be compared using <, <=, > and >= in which case the usual lexicographic ordering applies.

Operations

A large number of string operations are provided for manipulating strings. These are given below.

Table 21.12.

asBool():Boolean	Converts a string into a Boolean provided it has the string value true or false. String can be lower or upper case. An exception is raised if the string is invalid.
asFloat():Float	Converts a string into a Float. It splits the string on its decimal point, converts the two strings into integers and passes them to the Float constructor. An exception is raised if the result is not a Float.
asHTML():String	Transforms a string literal to HTML replacing any illegal HTML characters so that the string is faithfully printed.
asInt():Integer	Converts a string into an Integer. Raises an exception if it cannot be converted.
asSeq():Seq(Integer)	Converts a string into a sequence of character codes.
asSet():Set(Integer)	Converts a string into a set of character codes
asSymbol():Symbol	Converts a string into a Symbol. In general, symbols can be processed more efficiently than strings, e.g. as indexes in table lookups.
at(i:Integer):Integer	Returns the ith character in a string starting from position 0.
default():Element	Returns the default value for a string: the empty string
deleteFile():Boolean	Deletes the file given by a string path. Raises an exception if the file does not exist or cannot be deleted.
drop(n:Integer):Element	Removes the first i elements from a string.
edit():Element	Launches an editor for a string.
escapeCharsToNewLines():Element	Substitutes escape characters in a string for new lines.
exec(args:Seq(String)):Element	Currently not supported.
fileExists():Boolean	Returns true if the file given by a string path exists, otherwise false.
fileSize():Integer	Returns the size of a file given by a string path.

greater(other:String):Boolean	Returns true if a string is greater than the supplied string. The strings are compared alphabetically.
hasPrefix(prefix:String):Boolean	Returns true if a string is prefixed by the string, prefix.
hasSuffix(suffix:String):Boolean	Returns true if a string has a suffix, suffix.
indexOf(char:String):Integer	Returns the index of a character, char in a string.
isOlder(file:String):Boolean	Compares the last modified date of the file referenced by a string path with file. Returns true if file is older.
less(other:String):Boolean	Returns true if a string is less than the supplied string. The strings are compared alphabetically.
loadBin(verbose:Boolean):Element	Load the binary for the file referenced by a string path. Displays loading information if verbose is true. Raises an exception if it does not exist.
lookup():Element	Returns the value of the dynamic variable with the name defined by self or raises an error otherwise.
lowerCaseInitialLetter():String	Makes the first letter of a string lower case.
mkDir():Boolean	Creates a directory. Returns true when the directory already exists or is successfully created. Returns false when the directory cannot be created.
newLinesToEscapeChars():Element	Substitutes new lines in a string for escape characters.
padFrom(width:Integer,char:Integer):String	Pads up to a string with additional character codes up to width
padTo(width:Integer,char:Integer):String	Pads after a string with additional character codes up to width.
parentDir():String	Returns the parent directory of the file referenced by a string path. Returns an exception if it cannot be found.
readFile():String	Reads the file referenced by a string path, provided that it exists.
renameFile(newName:String):Boolean	Renames a file referenced by a string path. A rename is only performed if the file exists and there doesn't exist a file with the new name.
repeat(n:Integer):Element	Duplicates a string the given number of times.
reverse():String	Reverses the characters in a string.
size():Integer	Returns the size of a string.
splitBy(chars:String,start:Integer,last:Integer):Seq<String>	Splits a string into a sequence of strings around some characters. The variables start and last can be used to filter the returned string by returning the characters from start to last. Setting start and last to 0 will return the whole string.
stripNonAlphaChars():String	Strips all non-alphanumeric characters from a string.
stripWhiteSpace():String	Strips any whitespaces from a string.
subString(firstChar:String,pastLastChar:String):String	Uses indices to chop up a string. The first index is the starting character and the second index is 1+ the final character.

subst(new:String,old:String,all:Boolean):Element	Substitutes the string old in a string with new. If all is set to false, just replaces the first occurrence. If true replaces them all.
toLower():String	Converts all characters in a string to lower case.
toString():String	Prints a string.
toUpper():String	Converts all characters in a string to upper case.
truncate(width:Integer):String	Truncates a string by width characters.
upperCaseInitialLetter():String	Makes the first letter of a string upper case.

Examples

```
"to" + "day" = "today"
"tomorrow".size() = 8
"UPPER".toLower() = "upper"
"lower".toUpper() = "LOWER"
"Ceteva".subString(0,4) = "Ceta"
"lower" <> "LOWER" = true
"123.456".splitBy(".",0,0) = Seq{"123", "456"}
```

Characters are represented as integer ASCII codes. The following operation checks whether a string starts with an upper case character:

```
context Root
@Operation startsUpperCase(s:String):Boolean
  if s->size > 0
    then
      let c = s->at(0)
      in "A"->at(0) <= c and c <= "Z"->at(0)
    end
  else false
  end
end
```

Sequences

A sequence is a list of values that may have duplicates. A sequence literal is denoted by enumerating the elements in a Seq{ }. For example, a sequence of numbers: Seq{1,2,3}. Sequences can be nested, for example: Seq{Seq{"a","b"},1}.

The default value of a sequence is the empty sequence, Seq{ }. A Seq(Element) is an instance of the Sequence type. It contains the elements in the sequence.

Operators

Two sequences can be tested for equality and inequality. Two sequences are equal provides that they contain the same elements in the same order. Sequences can be concatenated.

Table 21.13.

Operator	Syntax	Result Type
equals	a = b	Boolean
not equals	a <> b	Boolean
concatenation	a + b	Sequence(Element)

Operations

XOCL provides a large number of sequence operations, including all those support by standard OCL.

Table 21.14.

append(s:Seq(Element)):Seq(Element)	Append two sequences.
asProperSeq():Seq(Element)	No Documentation Specified
asSeq():Seq(Element)	Turns a sequence into a sequence.
asSet():Set(Element)	Turn a sequence into a set.
asString():String	Turns a sequence of integers into a string.
asVector():Vector	Turns a sequence into a vector.
at(n:Integer):Element	Returns the nth element of a sequence starting from 0.
bind(key:Element,value:Element):Seq(Element)	Binds a key with a value and adds it to the head of the sequence.
binds(key:Element):Boolean	Returns true if a sequence contains a binding that matches the key/
butLast():Seq(Element)	Returns all elements but the last element.
contains(element:Element):Boolean	Returns true if the sequence contains the element.
default():Seq(Element)	Returns the default sequence: an empty sequence.
dot(name:String):Seq(Element)	Returns the result of iterating over a sequence and performing dot on each element.
drop(n:Integer):Seq(Element)	Drops the first n elements from a sequence.
equals(other:Element):Boolean	Returns true if a sequence equals another sequence. To be equal they must both be sequences and their elements should be equal and in the same order.
excluding(element:Element):Element	Returns all elements in the sequence excluding element.
exists(pred:Element):Boolean	Returns true when one element of the sequence satisfies the predicate otherwise it returns false.
flatten():Seq(Element)	Turns a sequence of sequences of X into a sequence of X.
forAll(pred:Element):Boolean	Returns true if all elements of the sequence satisfy the predicate otherwise returns false.
hasPrefix(prefix:Seq(Element)):Boolean	Returns true if a sequence is prefixed by the sequence prefix.
hasSuffix(suffix:Seq(Element)):Boolean	Returns true if a sequence is suffixed by the sequence suffix
head():Element	Returns the head of a sequence.
includes(element:Element):Boolean	Returns true if a sequence contains element.
includesAll(c:Collection(Element)):Boolean	Returns true if a sequence includes all the elements in the collection c.
including(e:Element):Seq(Element)	Returns the result of including the element e in a sequence. The element is added to the head of the sequence.

<code>indexOf(element:Element):Integer</code>	Returns the first index of the element in a sequence. If it is not found, returns -1.
<code>insertAt(e:Element,i:Integer):Seq(Element)</code>	Inserts an element e at position i in a sequence.
<code>inspectDialog(level:Element):Element</code>	No Documentation Specified
<code>isEmpty():Boolean</code>	Returns true if a sequence is empty.
<code>isKindOf(type:Classifier):Boolean</code>	Returns true if all elements in a sequence are instances of type.
<code>isProperSequence():Boolean</code>	Returns true if the last tail is a valid sequence.
<code>iter(iterator:Element,value:Element):Seq(Element)</code>	Iterates through a sequence, returning a sequence.
<code>last():Element</code>	Returns the last element of a non-empty sequence.
<code>linkAt(element:Element,index:Integer):Seq(Element)</code>	Returns the last element of a non-empty sequence.
<code>lookup(key:Element):Element</code>	Looks up a pair in a sequence using the key. Returns an error if the key cannot be found.
<code>max():Integer</code>	Returns the maximum valued element in the sequence.
<code>mul(s:Seq(Element)):Seq(Element)</code>	Generates a sequence containing all combinations of elements in the two sequences.
<code>prefixes():Seq(Element)</code>	Returns all possible prefixes of a sequence including the empty sequence.
<code>prepend(e:Element):Seq(Element)</code>	Prepend adds an element to the head of a sequence and returns a new sequence.
<code>qsort(pred:Operation):Seq(Element)</code>	Quicksorts the elements in the sequence. Is supplied with an operation of the form @Operation(x,y) predicate en-d where x and y will be elements in the sequence. An example predicate might be $x < y$.
<code>ref(nameSpaces:Element):Element</code>	Looks up a namespace path represented as a sequence of strings to the element found at the path. The operation takes a sequence of namespaces as an argument; the namespace arguments are used as the basis for the lookup.
<code>reverse():Seq(Element)</code>	Reverses a sequence.
<code>sel():Element</code>	Returns one element from a sequence.
<code>select(predicate:Element):Seq(Element)</code>	Applies a filter to a sequence of elements.
<code>separateWith(sep:String):String</code>	Constructs a string by concatenating the elements of a sequence together, separated by sep.
<code>set(key:Element,value:Element):Seq(Element)</code>	Sets the value of a binding in a sequence indexed by key. Creates a binding if one does not exist.
<code>size():Integer</code>	Returns the size of a sequence.
<code>sort(pred:Element):Seq(Element)</code>	Sorts a sequence using a comparison predicate of the form @Operation(x,y) predicate end. The predicate must be a comparison expression, e.g. $x < y$.
<code>sortByString():Seq(Element)</code>	Sorts by string value.
<code>sortNamedElements():Seq(NamedElement)</code>	Sorts a sequence of named elements. This operation is implemented in the kernel as is therefore very fast.
<code>sortNamedElements_CaseIndependent():Seq(Element)</code>	Sorts named elements by names ignoring case.

sortNames():Seq(String)	Sorts a sequence of names.
subSequence(starting:Element,terminating:Element)	Creates a subsequence given two indices. The first index is inclusive and is the starting index. The second index is exclusive and is the terminating index.
subst(new:Element,old:Element,all:Boolean):Seq(Element)	Replaces old for new in a sequence. If all is true, it will replace all elements, otherwise it will replace the first element.
tail():Seq(Element)	Returns the tail of a sequence.
take(n:Integer):Element	Takes n elements from the tail of a sequence.
toString():String	Produces a printed representation of a sequence.
zip(s:Seq(Element)):Seq(Element)	Produces a sequences of pairs by matching the first element of a sequence with the first element of s, and so on...

Examples

```

Seq{"red"}+Seq{"green"} = Seq{"red", "green"}
Seq{"red"}->at(0) = "red"
Seq{"red", "green"}->last() = "green"
Seq{"red"}->insertAt(0,"green") = Seq{"red", "green"}
Seq{"red", "green"}->indexOf("green") = 1
Seq{"red", "green", "amber"}->subSequence(1,2) = Seq{"green"}
Seq{"red", "green", "amber"}->tail() = Seq{"green", "amber"}

```

The operation butLast returns all elements in a sequence but the last element. It could have been defined as follows, note the use of Seq{head | tail} to construct a sequence with the given head and tail:

```

context Seq(Element)
@Operation butLast():Seq(Element)
if self->size = 0
then self.error("Seq(Element)::butLast: empty sequence.")
else if self->size = 1
then Seq{}
else Seq{self->head | self->tail->butLast}
end
end
end

```

Sets

A set is a list of values that may not have duplicates. A set literal is denoted by enumerating the elements in a Set{ }. For example, a set of numbers: Set{1,2,3}. Sets can be nested, for example: Set{Set{"a","b"},1}. The default value of a set is the empty set, Set{ }. A Set(Element) is an instance of the Set type. It contains the elements in the set.

Operators

Two sets can be tested for equality and inequality. Two sets are equal provided they contain the same elements irrespective of order.

Table 21.15.

Operator	Syntax	Result Type
----------	--------	-------------

equals	$a = b$	Boolean
not equals	$a \neq b$	Boolean

Operations

Table 21.16.

asSeq():Seq(Element)	Turns a set into a sequence.
asSet():Set(Element)	Turns a set into a set.
collect(filter:Element):Set(Element)	Returns the set of elements that result from evaluating filter over a set.
contains(element:Element):Boolean	Returns true if a set contains an element.
default():Set(Element)	Returns the default value of a set: the empty set.
dot(name:String):Set(Element)	Returns the result of iterating over a set and applying dot to the slot named name.
edit():Element	Launches a browser for a set.
excluding(element:Element):Set(Element)	The set excluding the element.
exists(pred:Element):Element	Returns true if an element satisfying the predicate exists in a set.
flatten():Set(Element)	Turns a set of sets into a set.
includes(element:Element):Boolean	Returns true if a set includes element.
includesAll(c:Collection(Element)):Boolean	Returns true if a set includes all elements from another collection.
including(element:Element):Set(Element)	The result of the set including element.
intersection(set:Set(Element)):Set(Element)	Returns the intersection of two sets.
isEmpty():Boolean	Returns true if the set is empty.
isKindOf(type:Classifier):Boolean	Returns true if all elements in a set are of the type.
iter(iterator:Element,value:Element):Element	Iterates over the elements in the set
max():Integer	Find the element with the maximum value in the set.
power():Set(Element)	Returns the powerset of elements in a set, i.e. all possible subsets of a set including the empty set.
reject(pred:Element):Set(Element)	Rejects any elements in the set that satisfy the predicate.
sel():Element	Selects a single element from a set.
select(predicate:Element):Set(Element)	Selects any elements in the set that satisfy the predicate.
size():Integer	Returns the size of the set.
toString():String	Prints the set as a string/
union(set:Set(Element)):Set(Element)	Returns the union of the two sets.

Examples

```

Set{1,2,3}->includes(1) = true
Set{1,2,3}->includesAll(1,2) = true
Set{1,2,3}->including(4) = Set{1,2,3,4}
Set{1,2,3}->excludes(1) = false
Set{1,2,3}->excludesAll(Set{4,5}) = true

```

```
Set{1,2,3}->excluding(1) = Set{2,3}
Set{1,2}->union(Set{2,3}) = Set{1,2,3}
Set{1,2,3} - Set{1,2,3} = Set{}
Set{1,2}->intersection(Set{2,3}) = Set{2}
Set{}->isEmpty() = true
Set{1,2,3}->size() = 3
Set{1}->sel() = 1
Set{Set{1},Set{2}}->flatten() = Set{1,2}
```

Suppose that the set operation includes was not provided as part of XOCL. It could be defined by:

```
context Set(Element)
@Operation includes(e:Element):Boolean
  if self->isEmpty
    then false
  else
    let x = self->sel
    in if x = e
      then true
      else self->excluding(x)->includes(e)
    end
  end
end
```

Symbols

Symbol is a sub-class of String. Whereas there may be two different strings with the same sequence of characters, there can only be one symbol with the same sequence of characters. This is useful when using names as the basis for lookup (in tables). For example XMF ensures that classes, packages, operations, slots are named using symbols so that the lookup of these features by name is as efficient as possible. If strings were used the lookup would necessarily involve a character-by-character comparison. Using symbols the lookup can use the symbols identity as the comparison operator. You can reference a symbol by constructing an instance: Symbol(name).

Operations

Table 21.17.

Operator	Syntax	ResultType
equals	a = b	Boolean
not equals	a <> b	Boolean
greater	a > b	Boolean
less than	a < b	Boolean

Symbols can be compared for equality, non-equality, greater than and less than, in which case the usual lexicographic ordering applies.

Tables

A table associates keys with values. Any element can be used as a key. A table has an initial size and can support any number of values. Use 'hasKey/1' to determine whether a table contains a key. Use 'get/1' to access a table via a key and 'put/2' to update a table given a key and a value. Use 'keys/0' to access the set of keys for a table.

Operations

Table 21.18.

get(key:Element):Element	Return the value of the supplied key in the table. If the key does not exist then an exception is raised. Use 'hasKey/1' to check if the key exists.
hasKey(key:Element):Boolean	Tests whether the table has a key or not.
keys():Seq(Element)	Returns all the keys in the table.
pprint():String	This operation prints out all the entries in a table.
put(key:Element,value:Element):Element	Add an association between the supplied key and value. Any existing association for the key is removed.
ref(index:Integer):Element	No Documentation Specified
remove(key:Element):Element	Remove the supplied key from the table. This succeeds whether the key exists in the tabel or not. Any daemons defined for the table are performed. The table is returned.
set(index:Integer,value:Element):Element	No Documentation Specified
size():Integer	Returns the size of a table.
toString():String	Returns a string representation of a table.
values():Set(Element)	Returns all elements in the tables as a set.

Examples

The following example adds and retrieves elements from a table:

```
context Root
@Class Fill
  @Attribute table : Table(1000) end
  @Operation add(key,element)
    self.table.put(key,element)
  end
  @Operation retrieve(key)
    self.table.get(key)
  end
end
```

Threads

XOCL provides threads for concurrent processing. A new thread of control is created using the special form:

```
@Fork(n)
  body
end
```

where n is a name used to identify the thread and body is an XOCL expression that is run when the thread starts. At any time the XVM is executing a single thread of control. The thread continues on the XVM until either it performs a read operation that blocks on input or when it explicitly calls yield. All XOCL values implement the yield operation. In both cases the thread is said to yield control. When a thread yields control, the XOS schedules another thread that is waiting. The scheduling algorithm aims to ensure that all waiting threads get scheduled providing that they yield.

Note that XMF is not intended to be a heavy-weight concurrent programming environment. Threads are provided for light-weight use, primarily for handling multiple connections from processes that communicate with XMF via input and output channels. There is nothing to stop threads being used to implement a variety of concurrent process architectures, however there are no facilities in XMF for controlling concurrent access to resource (such as locks, monitors etc).

Vectors

Vectors provide an efficient way of maintaining and accessing an array of values. Vectors are created using the constructor `Vector(<vector length>)`, where size is the length of the vector. A vector is indexed starting at position 0.

Operations

A vector is a fixed length array of elements. They are created using the constructor `Vector()`. Vectors provide very efficient insert (`put/2`) and lookup operations (`ref/1`).

Table 21.19.

<code>asSeq():Seq(Element)</code>	Converts a vector into a sequence.
<code>asString():String</code>	Converts a vector to a string.
<code>copyInto(vector:Element):Element</code>	Copies the elements of vector into self starting at position 0.
<code>put(index:Element,value:Element):Element</code>	Put the element value into a vector at position index.
<code>ref(index:Element):Element</code>	Returns the value at position index in a vector.
<code>size():Integer</code>	Returns the size of a vector.
<code>toString():String</code>	Returns a string representation of a vector.

Examples

The following operation creates a vector and populates it with element.

```
context Vector
@Operation fill(element:Element)
@For e in 0.to(self.size()) do
    self.put(e,element)
end
end
```

Debugging

XOCL operation invocation can be traced to check whether the operator is being called correctly. To trace an operation send it a trace message with no arguments. To stop the trace, send it an untrace message with no arguments. A traced operation will print out its arguments when it is called and its return value when it returns. All the operations defined in a name space N can be traced and untraced by `N.traceAll()` and `N.untraceAll()`.

Relationship to OCL and ASL

XOCL is intended to be a superset of OCL, providing many additional operations and data types over standard OCL. There are a few places where XOCL differs from standard OCL however. These include the following features:

Table 21.20.

XOCL	Standard OCL
OCL tags, e.g. inv: and pre: are not supported	These tags are defined by the grammar declaration, e.g. @OCL or by the context of the form it is being typed into.
isKindOf	oclIsKindOf
collect	Standard OCL collect always flattens the resulting collection. In XOCL flattening is not automatic, so the flatten operation must be applied where required.
iterate	Does not currently support types in accumulator declaration, e.g. ->iterate(a x : Integer = 0 ...)
XOCL currently does not support Bags	Standard OCL supports Bags

ASL (the Action Semantics Language) is the OMG's standard language for expressing dynamic behaviour. It is particularly oriented towards capturing concurrent and real-time behaviour. Although XOCL incorporates some of the features of ASL, XOCL is intended to fill a different gap: that of a general-purpose metaprogramming language. It therefore does not incorporate many of the complex features provided by ASL.

XOCL Grammar

```

AName ::= Name | Drop.

Apply ::= PathExp Args | KeyArgs .

ArithExp ::= UpdateExp [ ArithOp ArithExp ].

ArithOp ::= '+' | '-' | '*' | '/'.

Args ::= '(' ')' | Exp (',', Exp)* ')'.

AtExp ::= '@' AtPath @ 'end'.

AtPath ::= Name ('::' Name)*.

Atom ::= VarExp | 'self' | Str | Int | IfExp | Bool | LetExp |
       CollExp | AtExp | Drop | Lift | '(' Exp ')' | Throw | Try |
       ImportIn | Float.

AtomicPattern ::= Varp | Constp | Objectp | ConsP | Keywordp.

Binding ::= AName '=' LogicalExp.

Bindings ::= Binding (';' Binding)*.

Bool ::= 'true' | 'false'.

CollExp ::= SetExp | SeqExp.

CompareExp ::= ArithExp [ CompareOp CompareExp ].

CompareOp ::= '=' | '<' | '>' | '<>' | '>=' | '<='.

CompilationUnit ::= ParserImport* Import* (Def | TopLevelExp)* EOF.

```

```
Consp ::= Pairp | Seqp | Emptyp.

Constp ::= Int | Str | Bool | Expp.

Def ::= 'context' PathExp Exp.

Drop ::= '<' Exp '>'.

EmptyColl ::= Name '{' '}'.

Emptyp ::= Name '{' '}'.

Exp ::= OrderedExp.

Expp ::= '[' Exp ']'.

Float ::= Int '..' Int.

Import ::= 'import' TopLevelExp.

ImportIn ::= 'import' Exp 'in' Exp 'end'.

ParserImport ::= 'parserImport' Name ('::' Name)* ';' ImportAt.

IfExp ::= 'if' Exp 'then' Exp IfTail.

IfTail ::= 'else' Exp 'end' | 'elseif' Exp 'then' Exp IfTail | 'end'.

KeyArgs ::= '[' '(' ')' | KeyArg (',' KeyArg)* ')'.

KeyArg ::= Name '=' Exp.

Keywordp ::= Name ('::' Name)* '[' Keyps ']'.

Keyps ::= Keyp (',' Keyp)* | .

Keyp ::= Name '=' Pattern.

Lift ::= '[' '|' Exp '|']'.

LetBody ::= 'in' Exp | 'then' Bindings LetBody.

LetExp ::= 'let' Bindings LetBody 'end'.

LogicalExp ::= NotExp [ LogicalOp LogicalExp ].

LogicalOp ::= 'and' | 'or' | 'implies'.

NonEmptySeq ::= Name '{' Exp ((',' Exp)* '}') | '{' Exp '}')'.

NonEmptyColl ::= Name '{' Exp (',' Exp)* '}').

NotExp ::= CompareExp | 'not' CompareExp.

Objectp ::= Name ('::' Name)* '()' Patterns ')'.

OrderedExp ::= LogicalExp [ ';' OrderedExp ].
```

```
OptionallyArgs ::= Args | .

Pairp ::= Name '{' Pattern '|' Pattern '}'.

PathExp ::= Atom [ '::' AName (::: AName)* ].

Pattern ::= AtomicPattern ('->' Name ('(' Pattern ')')*)* ('when' Exp | ).

Patterns ::= Pattern (',' Pattern)* | .

RefExp ::= Apply
(
  '->'
  (
    'iterate' '(' AName AName '=' Exp '|' Exp ')'
  |
    AName
    (
      OptionallyArgs
    |
      '(' AName '|' Exp ')'
    |
      )
  )
|
  '..' AName
(
  Args
|
  )
) *.

Seqp ::= Name '{' Pattern SeqpTail.

SeqpTail ::= ',' Pattern SeqpTail | '}'.

SeqExp ::= EmptyColl | NonEmptySeq.

SetExp ::= EmptyColl | NonEmptyColl.

Throw ::= 'throw' LogicalExp.

TopLevelExp ::= LogicalExp ';'.

Try ::= 'try' Exp 'catch' '(' Name ')' Exp 'end'.

UpdateExp ::= RefExp ('=' ! LogicalExp | ).

VarExp ::= Name Token.

Varp ::= AName ('=' Pattern | ) (':' Exp | ).
```

Pattern Grammar

```
Pattern ::= Add ('->including(' Pattern '))* [ 'when' Exp ].  
  
Add ::= Atom [ '+' Add ].  
  
Atom ::= Varp | Constp | Cnstrp | Seqp | Setp | Syntaxp | '(' Pattern ')'.  
  
Varp ::= Name [ '=' Pattern ] [ '::' Type ]  
  
Constp ::= Integer | String | Boolean | '[' Exp ']'  
  
Cnstrp ::= BOAp | Keyp  
  
BOAp ::= Path '(' [ Pattern (',' Pattern)* ] ')'  
  
Keyp ::= Path '[' [ Name '=' Pattern (',' Name '=' Pattern) ] ']'  
  
Seqp ::= 'Seq{' [ Pattern (',' Pattern)* ] '}' | 'Seq{' Pattern '|' Pattern '}'  
  
Setp ::= 'Set{}'  
  
Syntaxp ::= '[' '| Exp '| ']'.  
  
Path ::= Name (':::' Name)*.
```

Chapter 22. XTools

Introduction

The XTools package is used to construct and deploy tools. An Xtool is some data together with interfaces for both user and inter-tool interaction. Since all aspects of tools are modeled, Xtools can be analysed, transformed and deployed. This document describes the XTools approach to tool definition and is a reference manual for Xtool definition mechanisms.

The XTools Architecture

Introduction

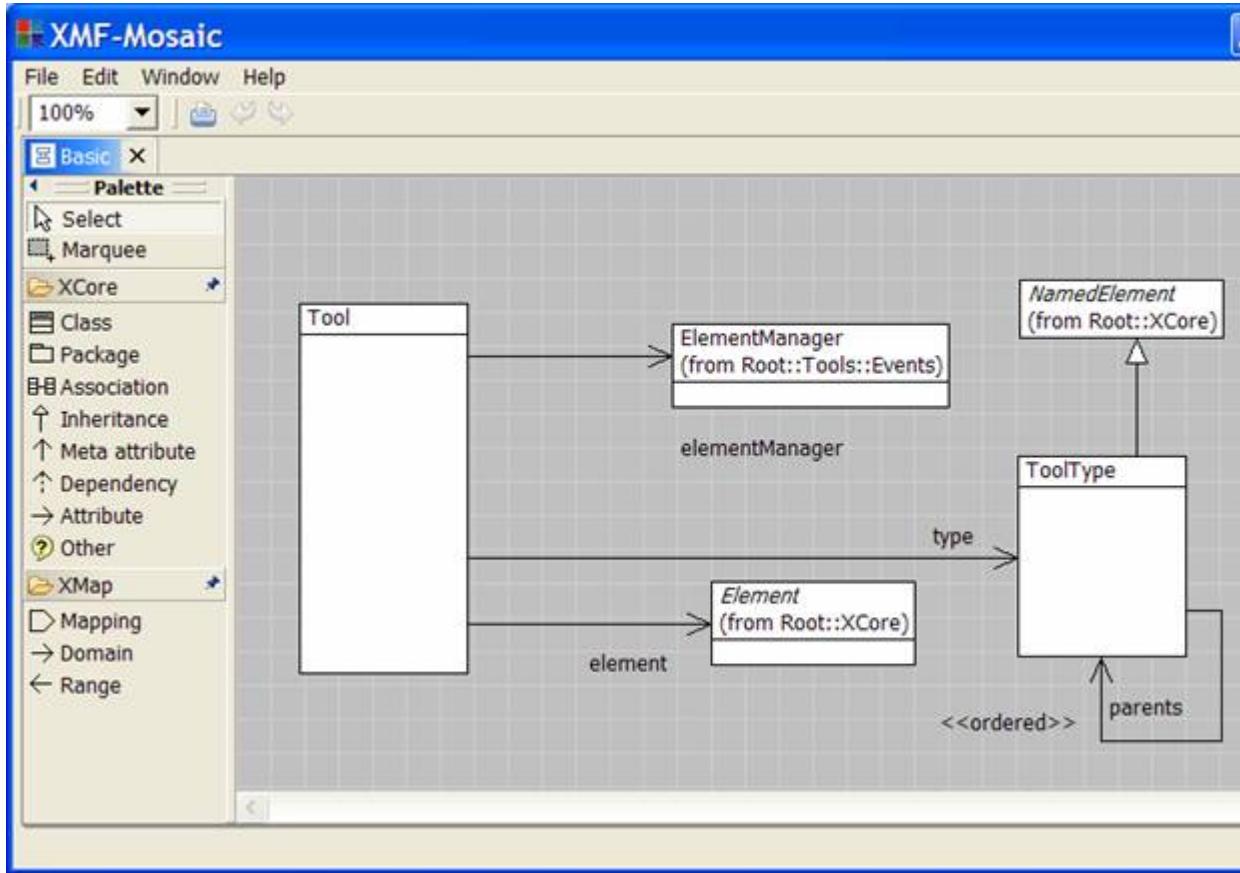
An Xtool is an instance of a domain model together with an exposed interface for constructing and manipulating the instance. An Xtool is modelled and is expressed in an XTools modelling language. The language hides away much of the implementation detail that is necessary to construct tools and ensures that multiple tools over a variety of domain models have a consistent look and feel. In addition, since an Xtool is modelled, it can be mechanically processed. In particular an Xtool can be transformed into different formats, for example exporting the definition as XML or as programming language source code.

Xtools manage changes in domain model instances by raising and handling events. Each Xtool contains a monitor for all the events that can be raised on the user interface and by external agents that change the domain model instance. By raising and handling events, multiple Xtools can be seamlessly composed without having to know anything about each other.

All Xtools conform to a basic architecture that ensures consistent look and feel across all generated tools and allows Xtools to work together to form tool federations. This section describes the components of the architecture.

Tool Component

The basic features of an Xtool are defined in the package Clients::Tools::Basic:



A tool consists of:

A tool type that defines how the tool's user interface should be constructed and managed. A tool type is like a class in that a single tool type can be the type of many tools. A tool type defines named attributes of the tool; the value of each attribute depends on the type of tool we are constructing: for example, a diagram tool can have text attributes and image attributes whereas a form tool can have tree attributes and text box attributes. The tool type also defines the events that are generated when the user interacts with the interface. For example, if the tool type defines a text field called name then an event name_Changed will be raised whenever the user types in the text field.

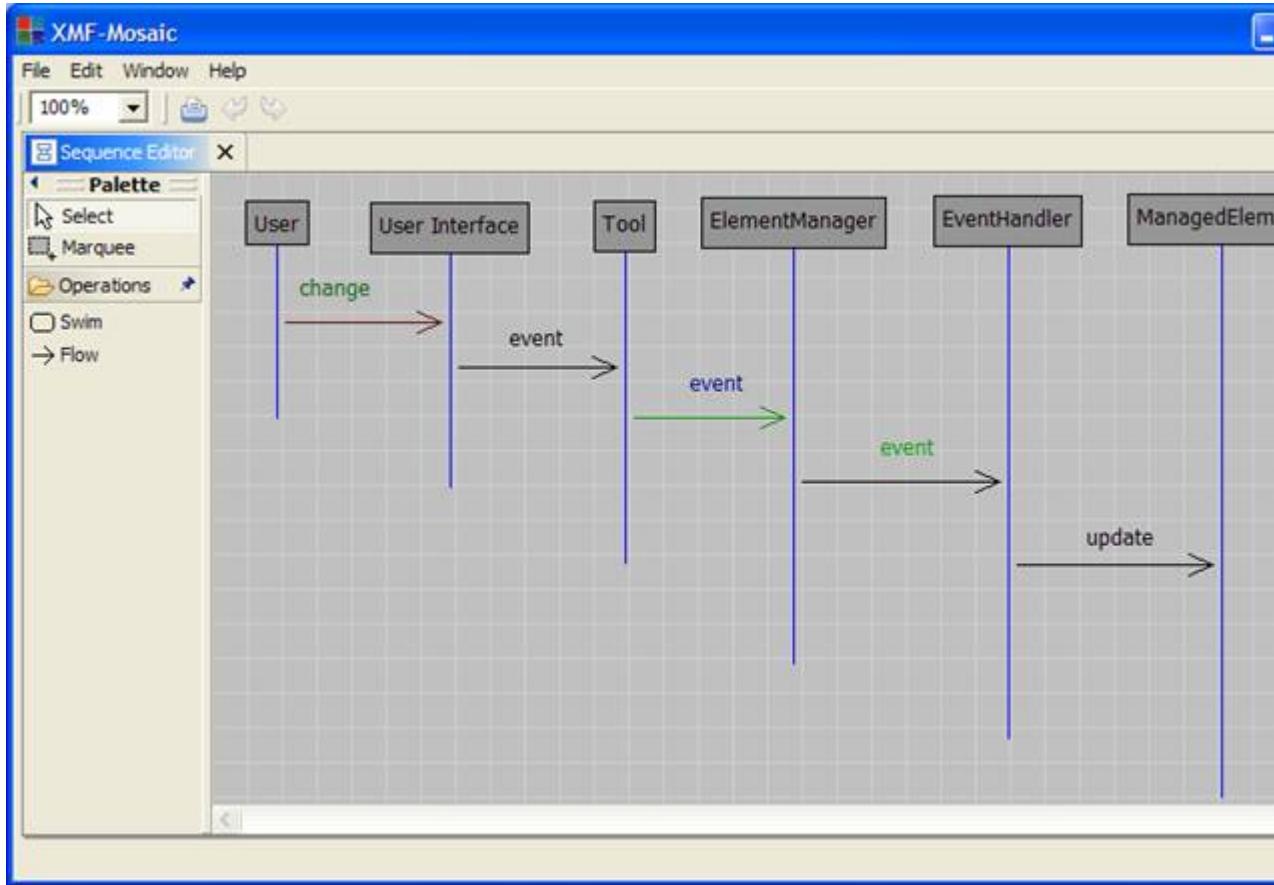
An element manager that defines how the tool responds to events. Events are generated either by user interaction with the interface or by modifications to the underlying element managed by the tool. An element manager contains a collection of event handlers.

A domain element that is managed by the tool. Change events from the user interface cause the element manager to modify the managed element. Change events from the managed element cause the element manager to modify the user interface associated with the tool.

Note that the user interface is not a prescribed part of a tool definition since the details of the user interface will depend on the type of tool.

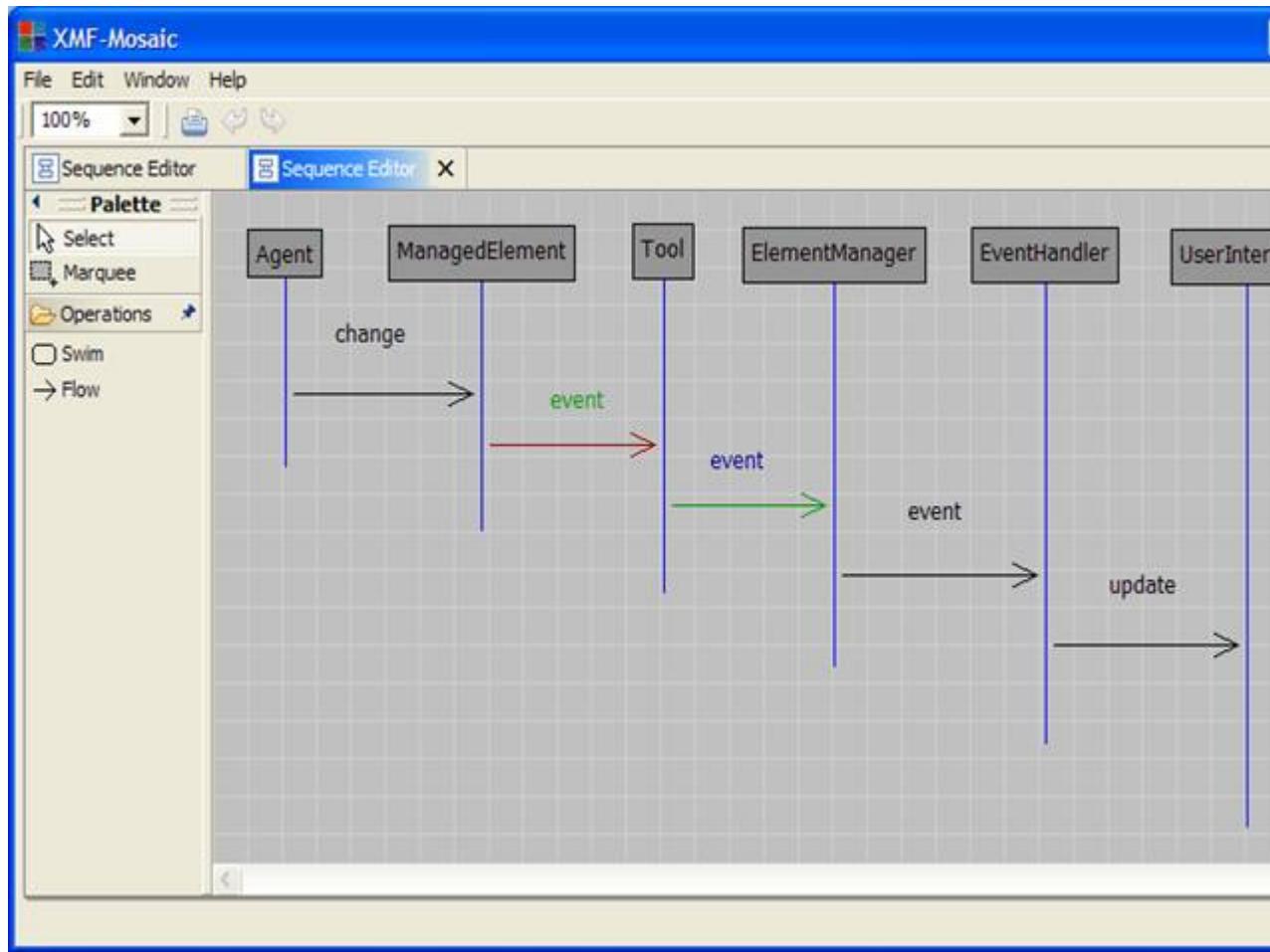
Tool Event

Tool execution occurs in the form of events. An event occurs due to a change arising in the user interface (such as a diagram edit); or due to a change in the element managed by the tool. Events are handled by a tool's element manager. The element manager contains a collection of event handlers; each handler listens for events of a particular type. The event handler runs some code that handles the event when it occurs. Typically the code propagates a change from the user interface to the managed element and vice versa. The following diagram shows how events are handled when a change occurs to a user interface:

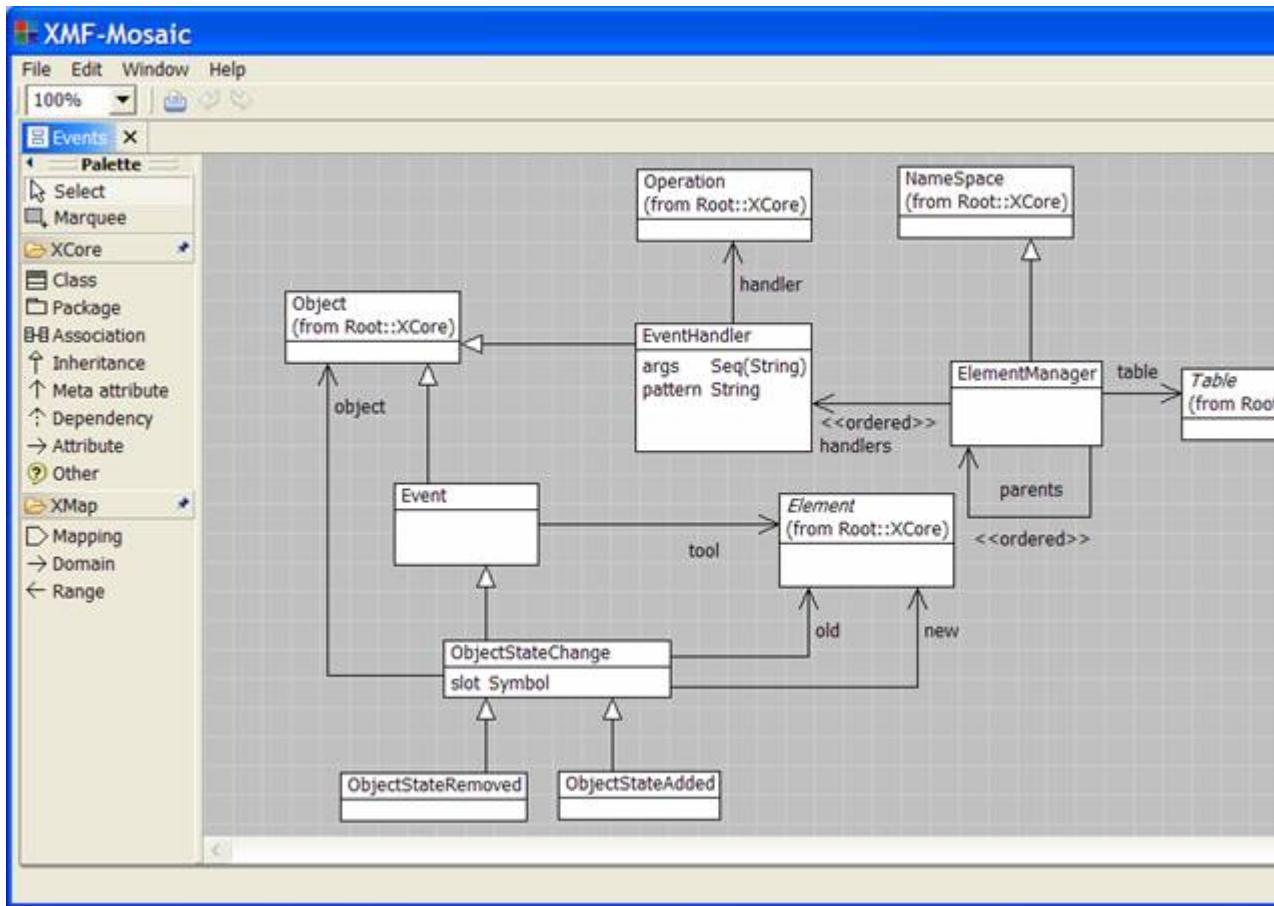


The user performs a modification to the interface via the mouse. This causes a state change in the interface object which is detected by a monitor and translated into an event of the appropriate type. The event is sent from the interface to the tool and then to the element manager. The element manager selects an event handler by matching against the type of the event. When a suitable event handler has been selected, it is executed. The body of the event handler performs appropriate updates to the managed element.

Conversely, the following diagram shows how changes to the managed element update the user interface. An external agent changes the managed element. If changes to the managed element are being monitored by the tool (sometimes this is not required) then the changes give rise to an event in the tool which, like changes to the user interface, give rise to an event in the element manager. An appropriate handler is selected which uses lookup mechanisms in the element manager to associate the changed element with a user interface component; the component is then updated.



The following model shows the basic architecture of the XTools event mechanism:



An event manager is a name space that defines event handlers. Each event handler has an operation that implements the code for handling the event. An event handler has a pattern that matches against the type of the event that the handler monitors. An element manager inherits handlers from its parents.

Creation events are typically propagated from user interface to managed element and vice versa. Once created, elements must be managed so that subsequent modification events can be propagated. An element manager has a table that is used to record associations between user interface elements and managed elements when they are created. Subsequent modification events can then look up the modified element in the table and propagate the change.

XTools provides a range of tool types. Each tool type provides a different collection of events. All XTool events inherit from the basic **Event** class that requires the event to record the tool that raised the event. The tool is particularly useful when implementing event handlers since it provides access to the managed element and the element manager with its table of associations.

Tool Definition

XTools provides a number of technologies for tool definition. The essential representation is using the XTools textual language to construct the tool components. The XTools classes define a number of grammars that must be imported using `parserImport` in order to use the textual format. The essential components of a tool definition are the tool type and the element manager. If the tool is to be defined in a single file then the following XMF code skeleton shows the key definitions:

```

parserImport Tools::DiagramTools::Types;
parserImport Tools::Events;

context MyPackage

@Package NewTool
  
```

```
@ToolType NewToolType  
    // Type component definitions...  
end  
  
@ElementManager NewToolManager  
    @EventHandler EventName()  
        // Handler body...  
    end  
    // Event handler definitions...  
end  
end
```

The tool type, element manager and event handlers need not be defined inside their container's textual definition. You may use context definitions to define these elements in isolation, for example:

```
context MyPackage  
    @ToolType NewTool  
        // Type component definitions...  
    end  
  
context MyPackage  
    @ElementManager NewToolManager  
    end  
  
context MyPackage::NewToolManager  
    @EventHandler EventName()  
        // Handler body...  
    end
```

Different context definitions may exist in different files (probably composed using a manifest). Of course, definitions must exist before they can be referenced, in the build order.

Once the tool type and the element manager have been defined a new tool is created by instantiating the appropriate XTools tool class. For example, if we are defining an diagram tool then we create an instance of the tool class Tools::DiagramTools::Structure::Tool:

```
import Tools::DiagramTools::Structure;  
import MyPackage::NewTool;  
  
...  
  
let tool = Tool(toolName, NewToolType, NewToolManager, element)
```

```
in tool.showDiagram()
end
```

...

A tool constructor takes four arguments: the name of the tool, the tool type, the tool manager and the managed element. The managed element can be any value: it is up to the event handlers to interpret this value appropriately. It is likely that new types of tool (based on diagram or form tools) will be required. It is easy to define your own extensions to these types of tool by sub-classing the appropriate Tool class.

Once a tool has been created, it is displayed and ready for use after a call of showDiagram. This call will create the user interface for the tool and display it. Subsequent calls to showDiagram will switch the tool focus back to this tool.

Tool Deployment

Tools can only be deployed into XMF-Mosaic in version 1.0 of XTools.

Diagram Tools

XTools supports a number of tool types. Diagram tools manage elements based on diagram editors. A diagram consists of nodes and edges with various display elements attached to the diagram components. A user interacts with the diagram by creating nodes and edges and editing the display elements. Each interaction gives rise to an event that is handled by the diagram tool's element manager. This section defines the features of diagram tools and provides a number of examples to show you how to go about creating and deploying a rich collection of domain specific tools.

Introduction

A diagram tool extends the basic notion of an Xtool with a diagram. XMF-Mosaic provides a model of diagrams in Clients::Diagrams; this is the basis for diagram tools.

Typically you will define a domain model and then decide how you want to construct elements of the model and interact with them. This design is the basis for a domain specific user interface (you may have more than one for any given domain model). The user interface is defined as an Xtool in terms of a diagram type and an element manager.

The following model shows a simple extension of basic XTools Tool to support diagrams:

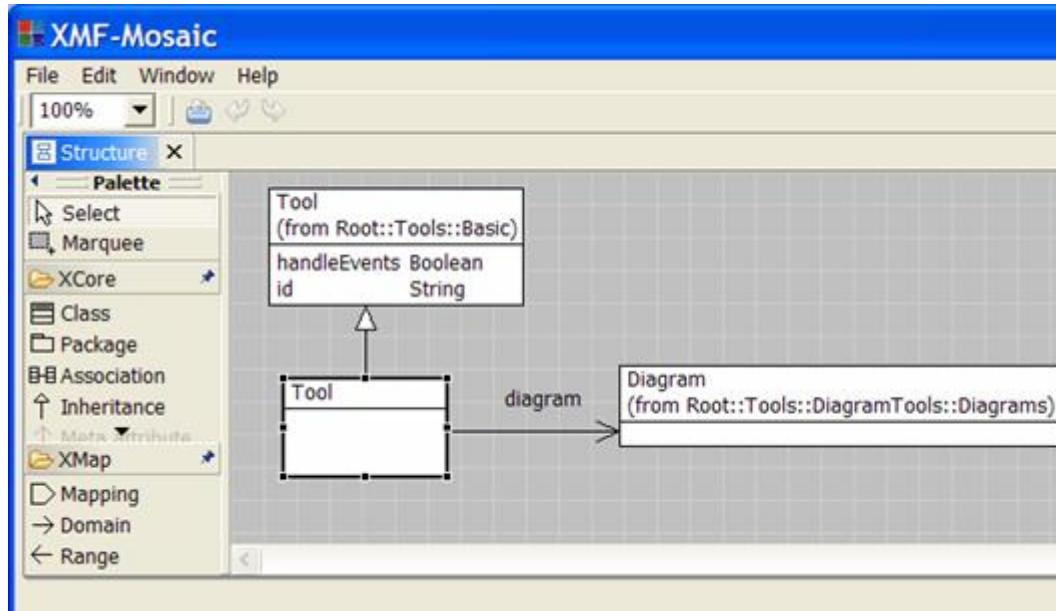


Diagram Tool Components

A diagram tool manages a diagram. The diagram is modelled and consists of a graph of nodes and edges. Each node contains a collection of display elements that describe how the node is to be rendered. A node has a number of ports that are used as places on the node to connect edges. A node without any ports cannot be connected via edges (but may be useful as a diagram element nonetheless).

An edge has a source and target port, has a line style, end arrows and a number of labels. The line style defines how the edge will be drawn (for example a dashed line). The end arrows defined the decoration, if any, that occurs on either end of the edge (for example an arrow head or a diamond). The labels of an edge are text fields that are attached to one of the start, end or middle of the edge. Labels may be readonly in which case they cannot be modified.

Node display elements may be one of the following:

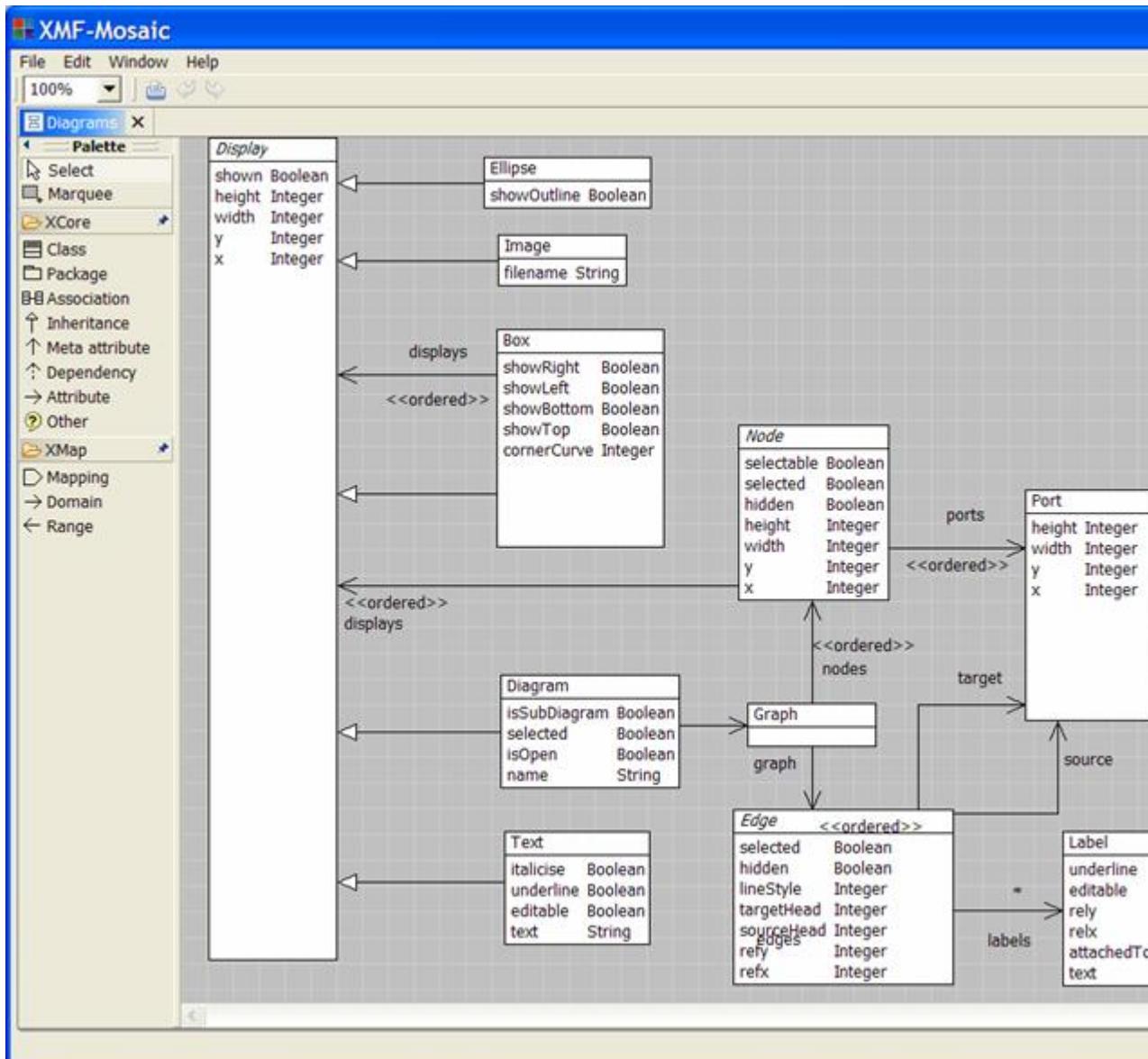
- Boxes. A box acts as a container of display elements. The edges of a box may be hidden or drawn. Boxes are useful when managing a collection of sub-elements on a node.
- Ellipse. An ellipse has a outline that may be hidden.
- Image. An image is a bitmap that is defined in an external file.
- Text. A text field may be readonly.

All display elements have a position relative to (0,0) which is the top left hand corner of its container (y increases down the screen). You do not have to worry about positions because XTools provides layout managers that deal with all the hard work of placing display elements.

All display elements have a width and height. In most cases these values are handled by the layout manager so you don't have to worry about positioning or size, you can focus on what the information is rather than how it is presented.

Both nodes and node display elements may be associated with ports. A port is a place on a node that acts as a receptor for edge connection. Whilst ports have position and size, associating a port with a diagram element in XTools automatically arranges for the port to be at the same location and size as the associated element. Typically you will associate a port with each node type you define. The node can therefore act as the source or target of an edge and XTools manages the resizing of the ports automatically when the node changes shape. Occasionally you may have a strange shaped node where various parts of the node can be connected differently; in this case you will associate ports with node display components.

The following model provides an overview of diagrams used in diagram tools. As noted above, you probably will not need to know about the detail of this model, just acquaint yourself with the logical relationships between the classes:



Nodes and Edges

A diagram consists of nodes and edges. For example, the diagram shown in the previous section consists of nodes of type `ClassNode` and edges of types `InheritanceEdge` and `AttributeEdge`. When defining a domain specific language editor, you need to list the node and edge types that you wish to display on the diagram. These may correspond directly to the classes in your domain model and the associations between them (node and edge types respectively). Of course the relationship between model classes and diagram types need not be so direct. This section describes how you define node and edge types in the XTools textual language.

A node type has the following format:

```
@NodeType name(properties) extends parents
  display elements
  menu
end
```

A node type has a name followed by a collection of comma separated properties in parentheses. Optionally you may define that a node type is an extension of a comma separated list of node type

names. This allows node type hierarchies to be constructed and can be important when defining edge types as described below.

Display elements are defined next followed by the menu for the node type. A node type may have the following properties:

HORIZONTAL, VERTICAL or OVERLAY layout directives for the display elements. This directive defines how the contained display elements will be arranged when a node of this type is created. These are defined in more detail in the section on layout later in this document.

hasport. This property defines that each instance of this node type has a port associated with the entire node. The port will automatically be resized when the node is resized.

An edge type has the following format:

```
@EdgeType name source -> target optSource optTarget optStyle  
    labels  
    menu  
end
```

An edge type has a name followed by the name of the source and target node types. An edge of the defined type may only be added to a diagram between nodes of the declared types. Note that node types are arranged in an inheritance hierarchy so that edge instances may be drawn between any sub-types of the declared source and target.

The source and target arrows are optional. If they are specified then they must both be specified and must be an integer that specifies the display element at the appropriate end. The integer values are as follows: noArrow, arrow, blackDiamond, whiteDiamond, blackArrow, whiteArrow. The optional style defines the way in which the edge will be drawn. The values are as follows: solidLine, dashLine, dottedLine, dashDottedLine, dashDotDotLine.

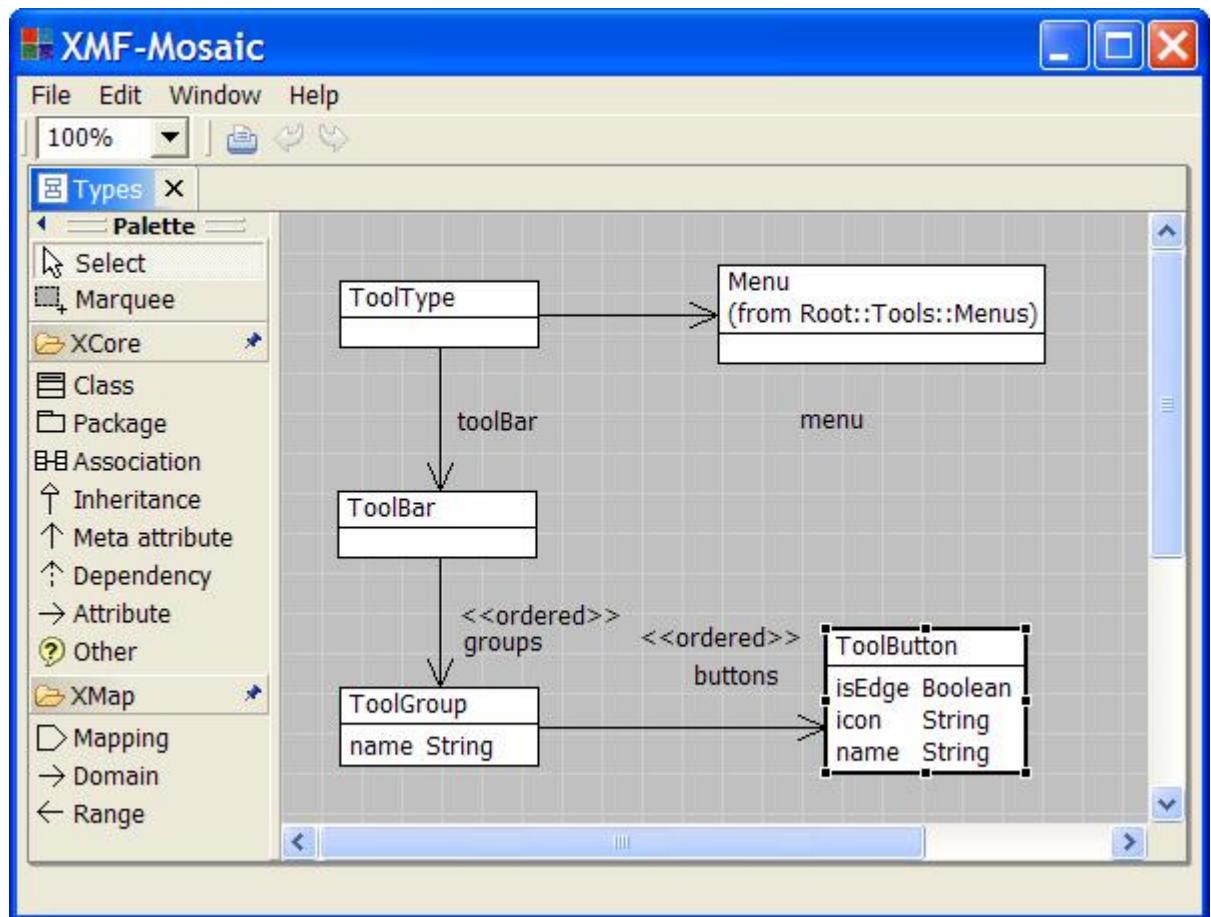
A label has the following format:

```
@Label name(position,dx,dy) string end
```

A label has a name. The name is used to navigate to the label in an instance of the enclosing edge type. The position of the label defines where it will be drawn relative to the edge. The position may be start, end or middle. The dx and dy values are integers that specify co-ordinates relative to the position of the label. The initial text of the label is given as a string in the body of the label type definition.

Toolbar

A toolbar is drawn at the left hand side of a diagram tool and contains tool groups and buttons. Groups are named collections of buttons. A toolbar is used to create instances of node and edge types. The following model shows the structure of toolbars:



The textual language for defining tool groups is as follows:

```

@ToolBar
  groups
end
  
```

Each group is defined as follows:

```

@ToolGroup name
  buttons
end
  
```

The name of the group appears in the tool bar. Each button is defined as follows:

```

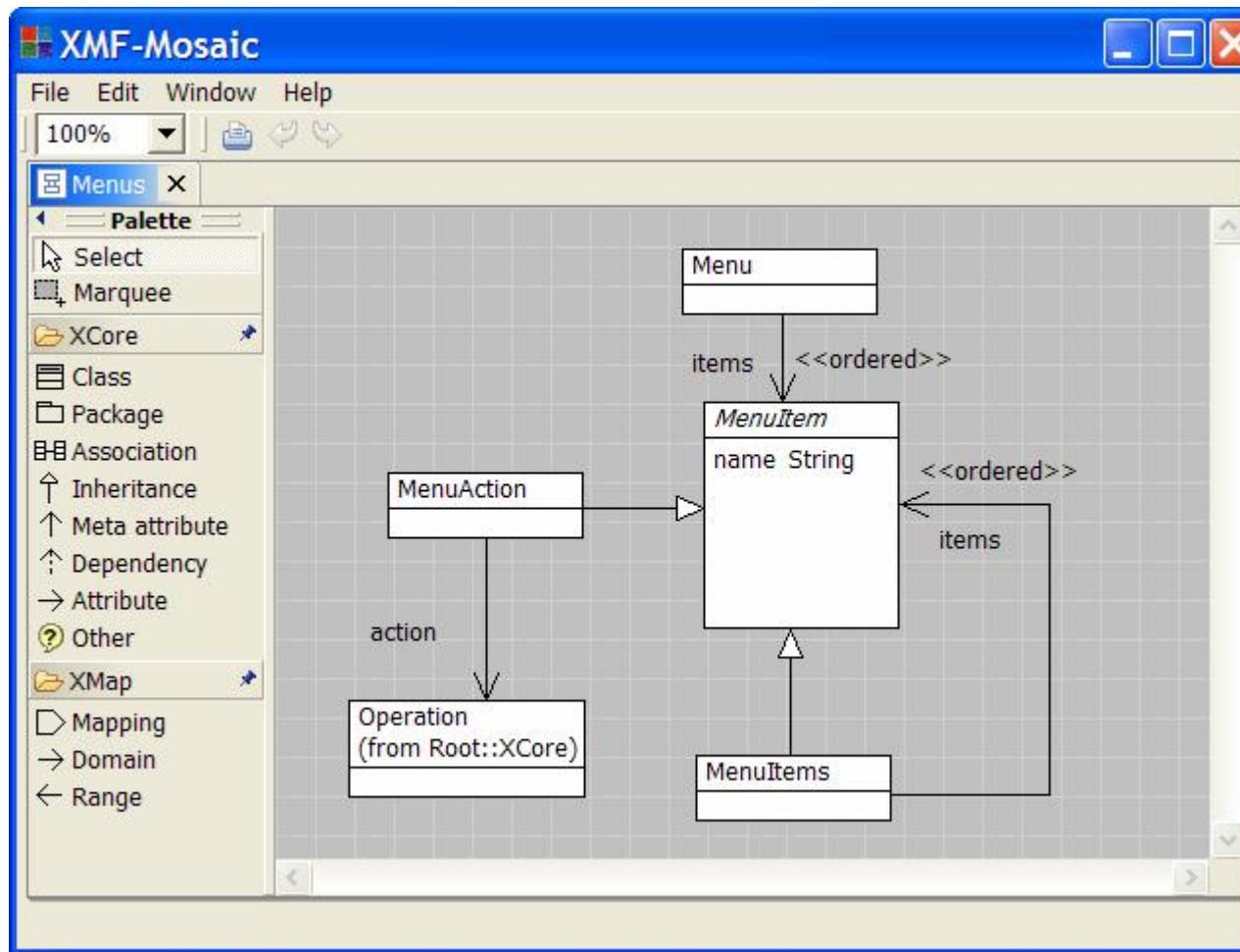
@ToolButton name optIsEdge
  icon = filename
end
  
```

The name of the button defines the text that appears on the diagram. The name of the button should be the name of a node type or an edge type. If it is the name of a node type then it defines a node creation button and will not contain isEdge. If it is an edge creation button then it must contain isEdge. The icon for the button is specified by giving the file containing the bitmap as a string.

Menus

Menus may be added to a large number of diagram tool elements. A menu definition specifies the functionality that is associated with the diagram element. When the element is selected on the diagram using a mouse right click, the menu appears offering menu actions. Menus can be defined for diagram

tool types, node types, display types and edge types. The following model shows the structure of a menu:



A menu is specified as follows:

```
@Menu
  items
end
```

A menu item is either a named sequence of menu items or a single menu action. A sequence of menu items produces a sub-menu and is used to group related menu actions. A menu action has an operation that is used to implement the action when it is selected.

```
@MenuItem name
  items
end
```

The body of a menu action is just XOCL code:

```
@MenuAction name
  body
end
```

A menu is displayed when you right click on an instance of the containing element type. You may then select one of the actions (possibly by navigating down sub-menus). The body of the selected action is performed. Within the body you may refer to the containing instance as self, and refer to the containing tool as tool. The action that is being performed can be referred to as action (this is useful in the rare circumstance that you want to remove the action from the menu).

Diagram Events

You perform operations on a diagram tool using the mouse. You may add a new node or re-attach an edge. When these changes take place, an event of an appropriate type is raised in the containing Xtool. The event contains references to the tool and to any elements that are involved in the diagram operation.

Events are raised and then handled by the event handlers defined by the tool's element manager. A handler is selected by calculating the event's raised name. Generally this name is composed of a path identifying the diagram component that changes and the type of the event. The path is a sequence of attribute names starting with the name of the type of the root diagram element (for example a node or edge) . For example, if we have the following node type:

When the text is modified on the diagram, the path component of the raised name is N_B_T and the event type (as we see below) is _Changed, therefore the raised name is N_B_T_Changed. The raised name is the name used in the event handler that will be used to handle the event.

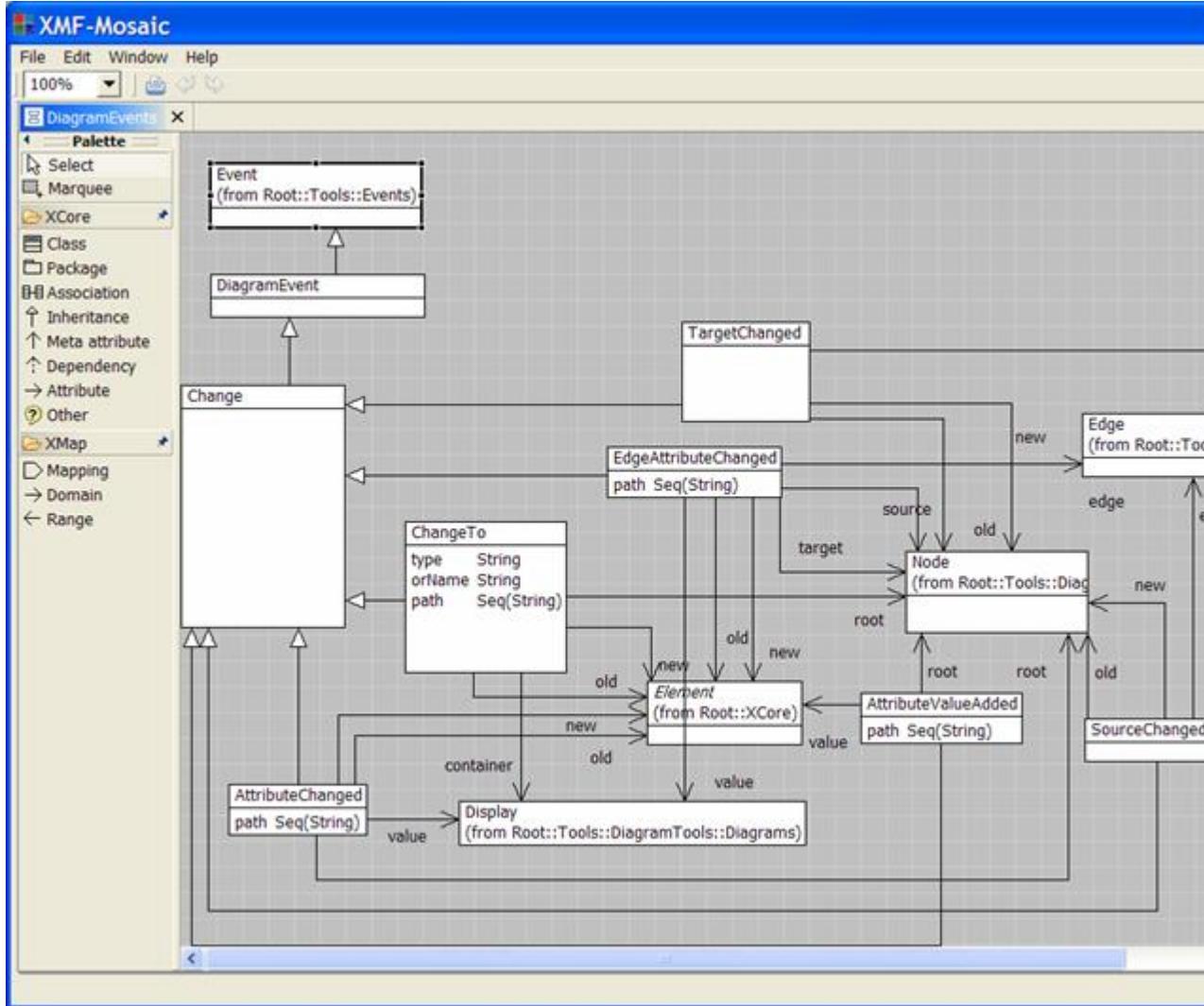
Diagram events fall into one of three general categories:

Change events. These occur when an existing diagram element is modified, for example by editing some text or by re-attaching an edge end. These also occur when an existing container is modified by adding or removing a new element.

Creation events. These occur when a new node or edge is created.

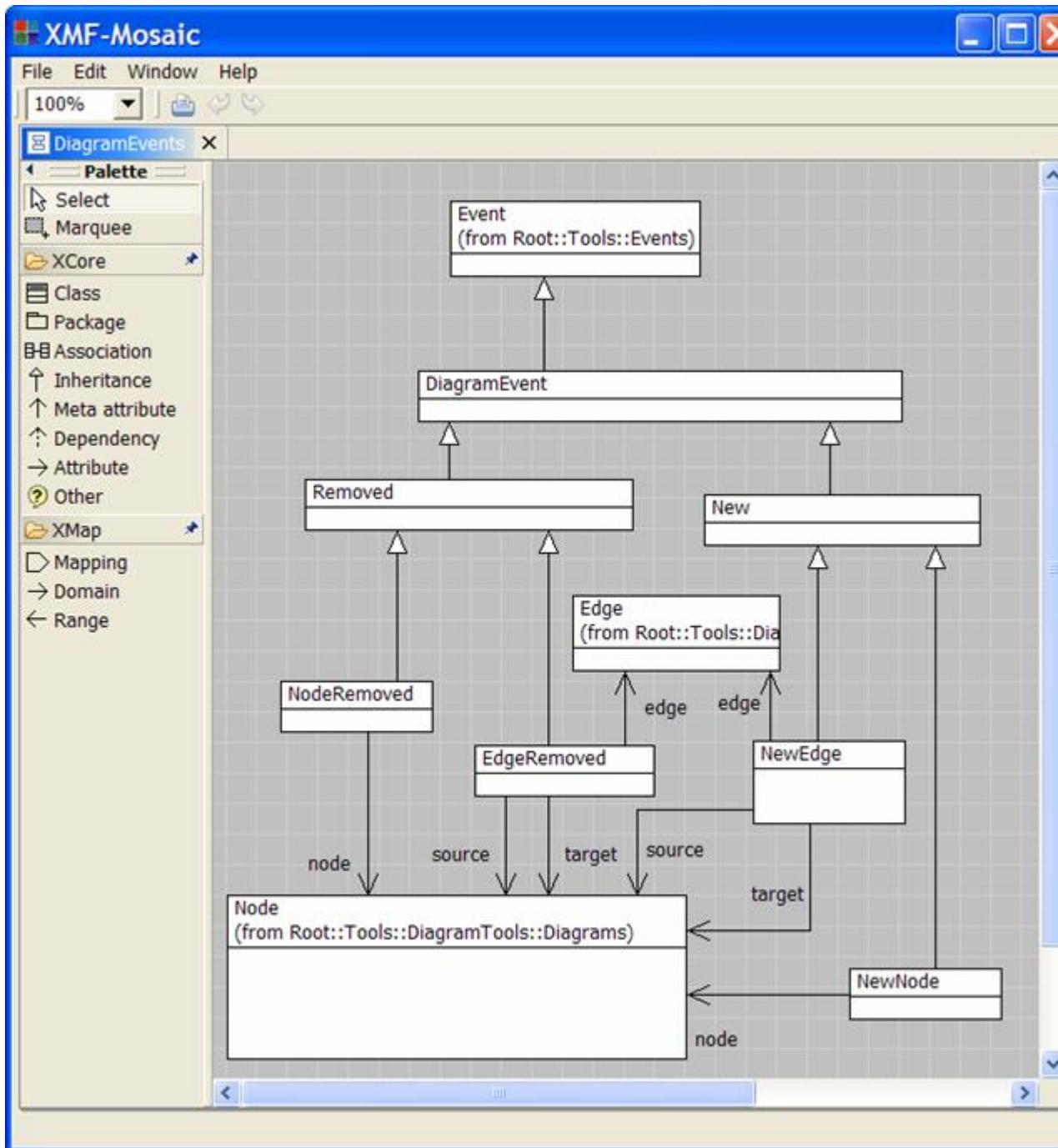
Deletion events. These occur when a node or edge is removed.

The following model shows the various types of change event:



The rest of this section describes each of the diagram event types and how their raised name is calculated.

- **AttributeChanged**. This event is raised when text is modified. The changed text element is the value of the event, the node containing the display element is the root of the event. The attribute value before and after the change is supplied via old and new in the event. The raised name is the path followed by _Changed.
- **EdgeAttributeChanged**. The same as AttributeChanged except that the text on an edge label is modified.
- **AttributeValueAdded**. When a container has a *-type (see later in the document), new instances of the *-ed type can be added. When a new instance is added this event is raised. The path identifies the attribute that has been added and the value of the event is the new display element. The raised name is the path followed by _Added.
- **SourceChanged**. When the source of an edge is moved from one node to another this event is raised. The source before and after the change is recorded in the old and new of the event. The raised name is the type of the modified edge followed by _Source_Changed.
- **TargetChanged**. As for source changed but the target of the edge has changed.
- **ChangeTo**



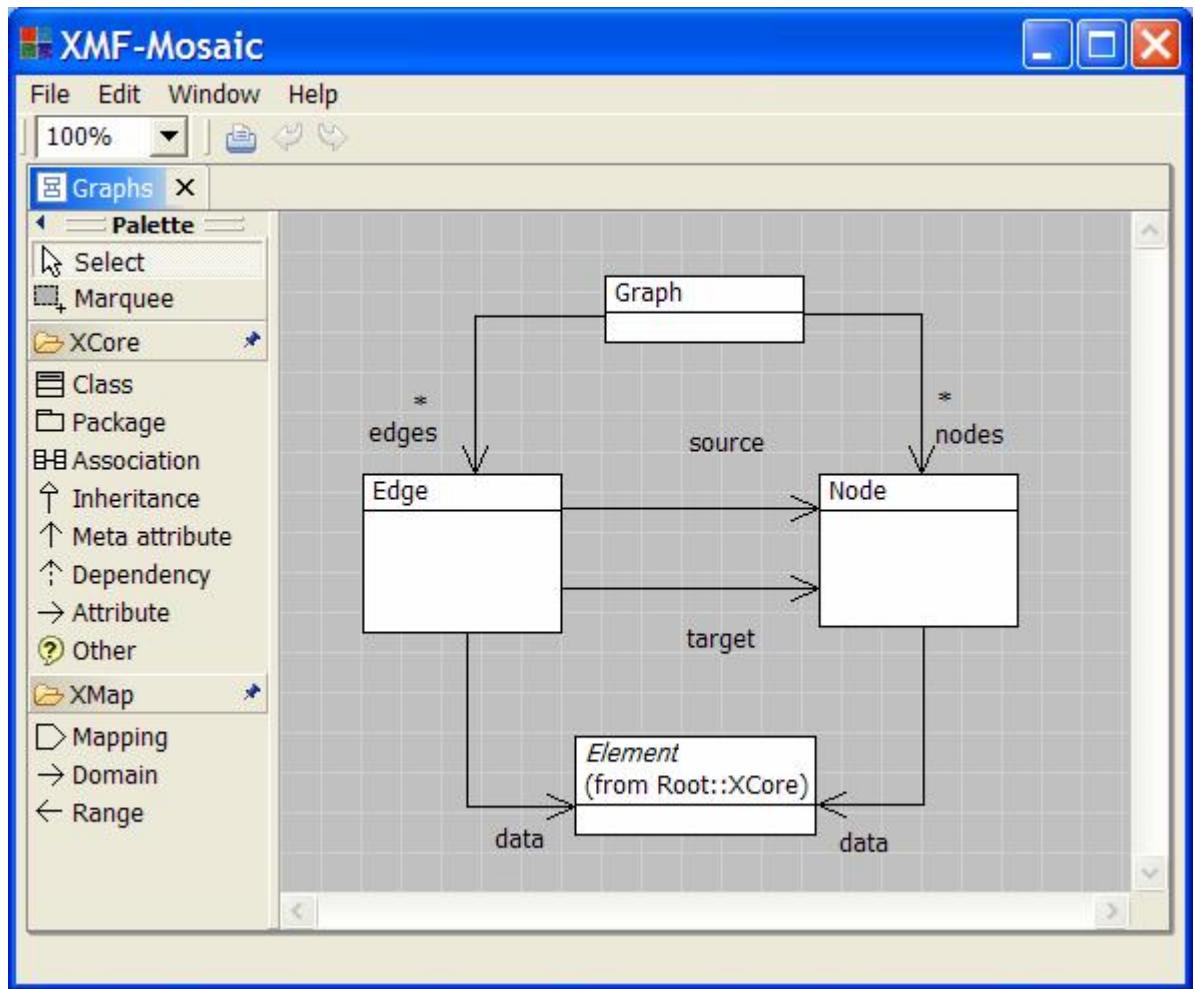
An Example Domain Specific XTool

This section provides a complete example of an Xtool. The tool manages a graph where the nodes and edges are labelled with text. The graph is updated by events occurring on a diagram interface and the diagram is updated by events occurring to the graph. The example is deliberately simple, but contains examples of most Xtool definition features.

A Domain Model

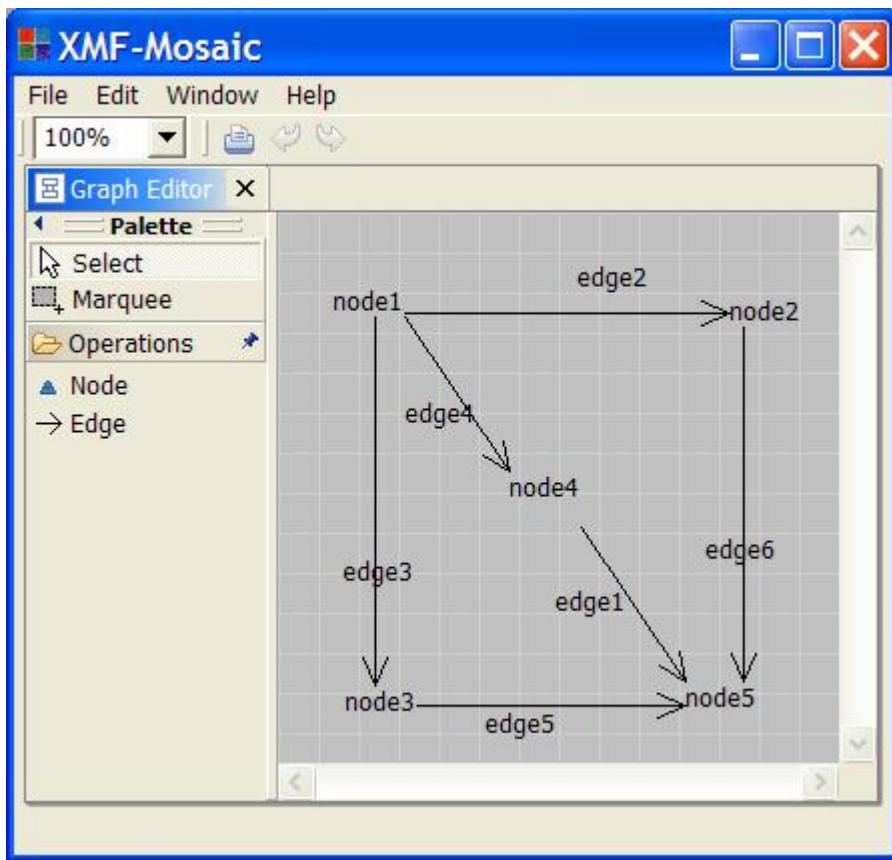
The first step in defining a DSL-tool is to define the domain model and its semantics. An Xtool exposes an interface over the domain model to the user and other tools. Our example domain model is one that it provided as standard with XMF-Mosaic: Graphs. The interface that we expose to the user is the

creation and deletion of graph nodes and edges. In addition we label the nodes and edges with strings and expose the setData operations via the diagram tool. The following model shows the domain model:



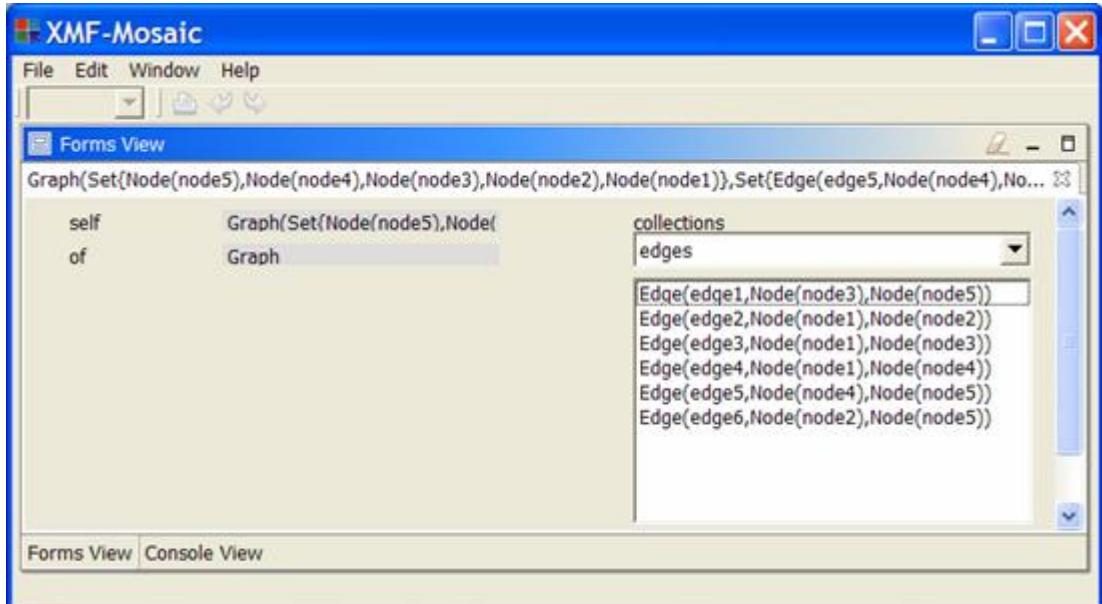
Design of the User Interface

Once we have defined the domain model and chosen the interface we want to expose, the next step is to informally sketch out the look and feel of the Xtool. In this case we will define a diagram tool to expose the chosen interface. The creation operations will be exposed via toolbar operations, node and edge deletion will be exposed via menu operations on the respective diagram elements. Data modification will be exposed via text modification on the diagram labels and edge source and target changes will be exposed by retargeting the edges on the diagram. The following screenshot shows an example usage of the tool we want to build:

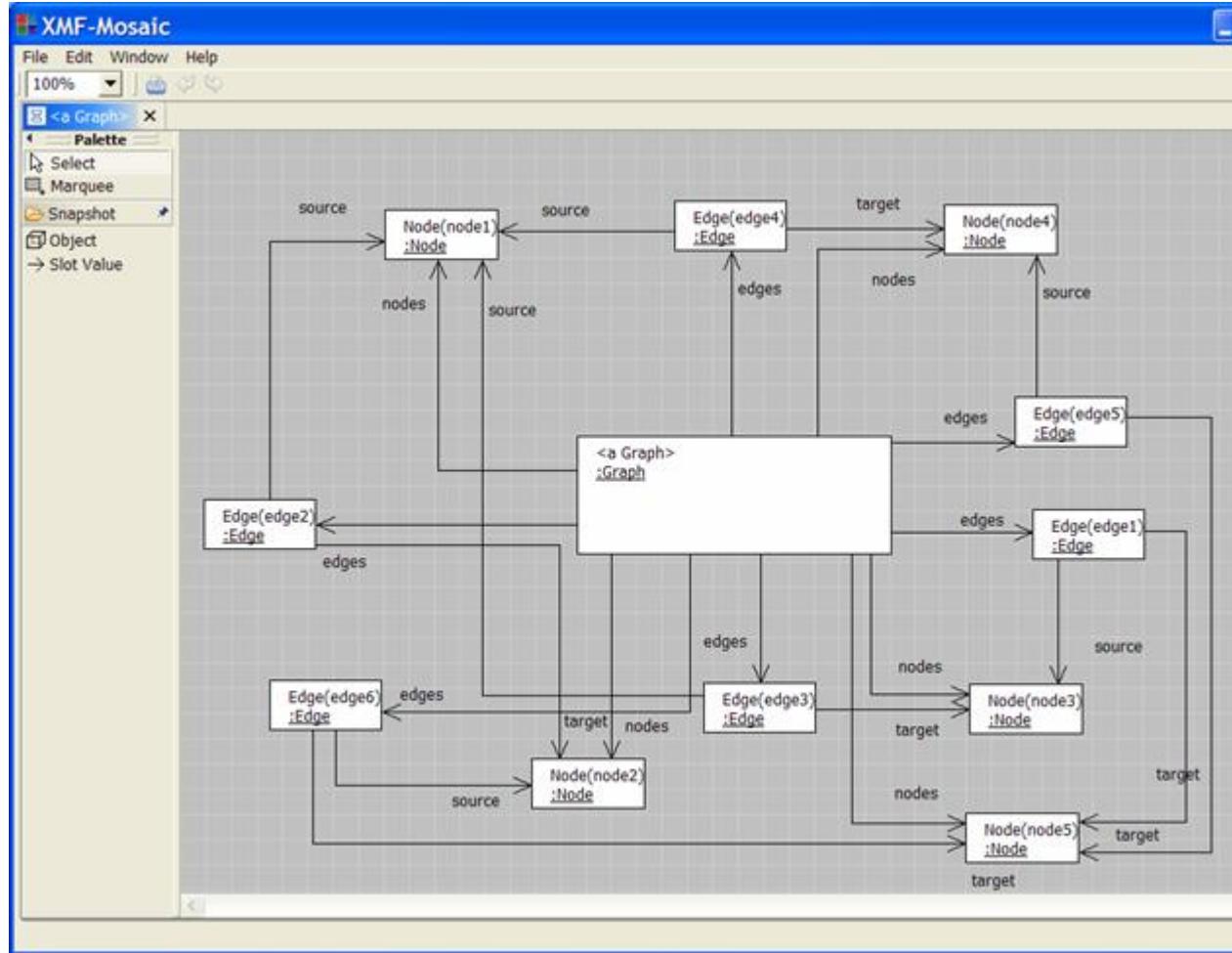


The left hand part of the tool shows a tool bar with a single tool group called Operations with a node creation button Node and an edge creation button Edge. The diagram canvas shows a graph diagram consisting of nodes (created by selecting the Node button and then clicking on the canvas) and edges (created by selecting the Edge button and linking two nodes). Each node and edge has a label that can be edited by selecting and modifying the text.

When we create and edit a graph diagram an underlying element is created and modified. In this case we will create and modify a graph from the package Graphs. This makes the example easy because the element and the diagram are in one-to-one correspondence. The graph consists of nodes and edges whose data components are the labels on the diagram. The following property editor shows the underlying graph element after constructing the diagram shown above:



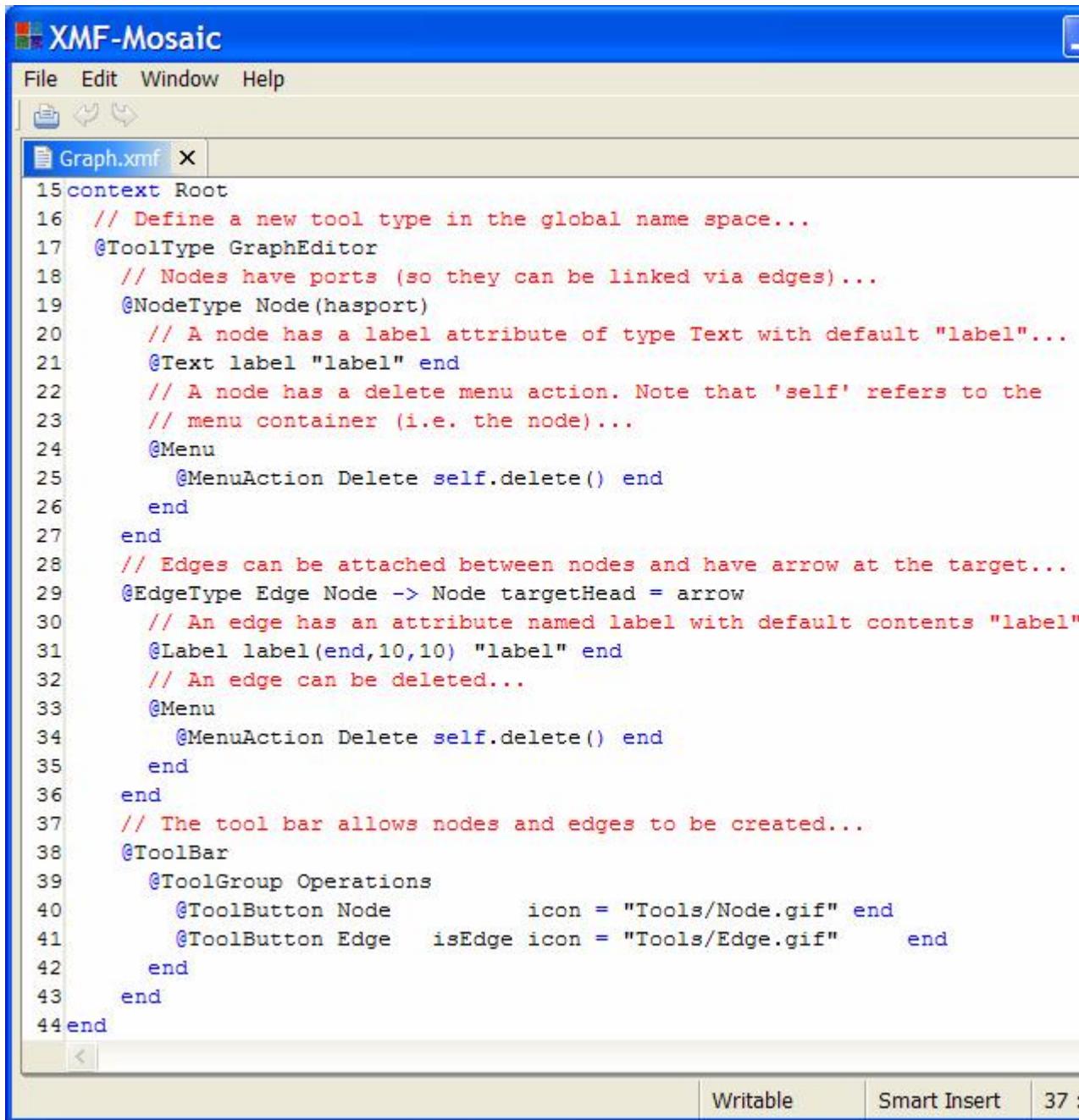
The following snapshot shows the graph as a snapshot diagram:



The rest of this section provides a step-by-step analysis of the graph editing tool definition.

Defining the Tool Type

The first step is to define the tool type. The tool type defines the nodes types, the edge types and the tool bar type. Each node and edge type defines named attributes. Node types have named attributes that are display elements; each display element contributes to how a node is drawn and to the events raised when a node is created and modified. Edge types have named attributes for the labels on the edge. Node, edge and display types can have menu types. The following tool snapshot shows the source code for the graph editor tool; the code includes comments that describe each of the type components:



```

15 context Root
16   // Define a new tool type in the global name space...
17   @ToolType GraphEditor
18     // Nodes have ports (so they can be linked via edges)...
19     @NodeType Node(hasport)
20       // A node has a label attribute of type Text with default "label"...
21       @Text label "label" end
22       // A node has a delete menu action. Note that 'self' refers to the
23       // menu container (i.e. the node)...
24       @Menu
25         @MenuAction Delete self.delete() end
26       end
27     end
28     // Edges can be attached between nodes and have arrow at the target...
29     @EdgeType Edge Node -> Node targetHead = arrow
30       // An edge has an attribute named label with default contents "label"
31       @Label label(end,10,10) "label" end
32       // An edge can be deleted...
33       @Menu
34         @MenuAction Delete self.delete() end
35       end
36     end
37     // The tool bar allows nodes and edges to be created...
38     @ToolBar
39       @ToolGroup Operations
40         @ToolButton Node           icon = "Tools/Node.gif" end
41         @ToolButton Edge      isEdge icon = "Tools/Edge.gif"      end
42       end
43     end
44 end

```

Writable

Smart Insert

37

Defining the Element Manager

Once the tool type has been defined it is possible to define the element manager for the tool. The element manager defines an event handler for each of the events that the tool must handle. Events are raised in two ways: either by modifications to the diagram (such as creating a new node or editing a label); or, by modifying the underlying data element that the tool is managing (such as adding a node to the graph managed by the graph editor). In the case of events raised by the data element, event handlers are only necessary if the tool is observing the data element. In some cases a tool is used to construct the underlying data element that is then exported or exclusively manipulated by the tool. In this case the tool is in full control of the underlying element and does not need to observe it for externally generated changes. Our graph editor will observe the underlying graph in order for the example to cover as many tool definition issues as possible.

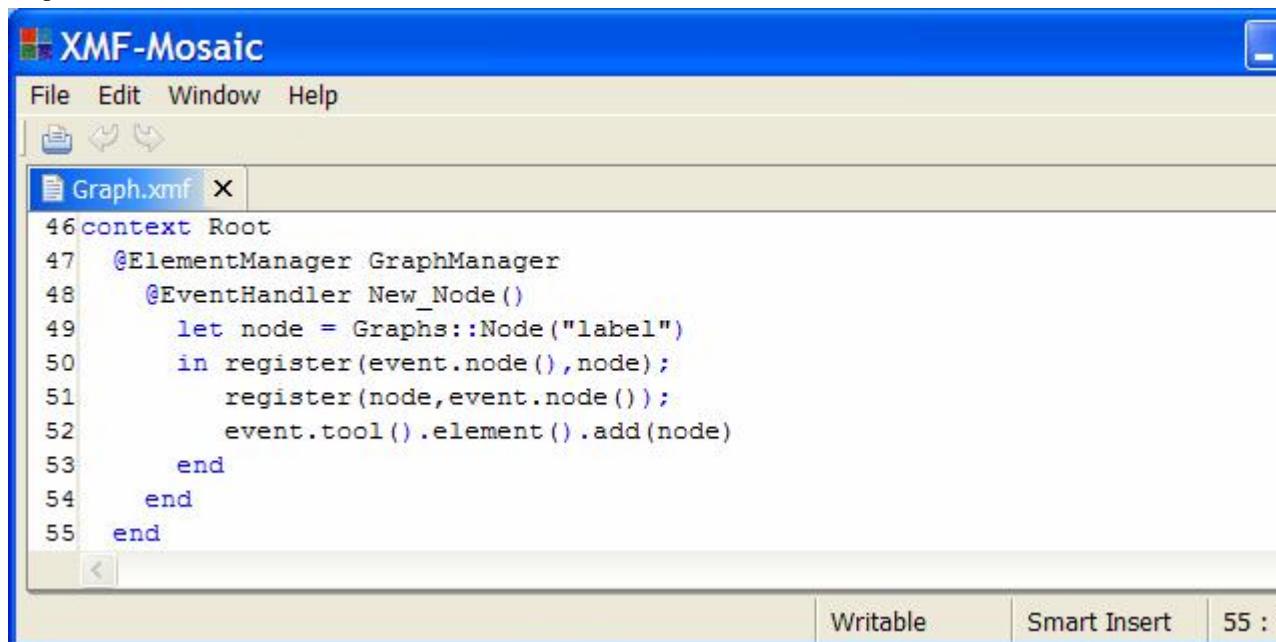
An element manager must define a creation event handler for each diagram node and edge type. A creation event is raised whenever a new node or edge is created on the diagram. An event handler has

a name that corresponds to the event that it is designed to handle. The event handler has an argument list (which will be empty for the purposes of this example) and a body. The body is supplied with the raised event as the value of the variable event. The state of the event is accessed using a collection of accessor operations; the names of the operations correspond to the attributes of the particular event type. All events have an operation tool() that returns the tool that raised the event.

The event handler body may also reference two special operations: register and find. Register is used to associate a key with a value in the element handler table. Typically this is used to associate tool elements with domain elements so that when events are raised, the handlers can map from a tool element to the appropriate domain element and vice versa. The graph editor example contains many examples of the use of these operations.

Handling Diagram Events

In the case of a node creation event, the new tool node is the value of node(). The following tool snapshot shows the source code for the node creation handler:



The screenshot shows a window titled "XMF-Mosaic" with a menu bar: File, Edit, Window, Help. Below the menu is a toolbar with icons for file operations. The main area is a code editor with a tab labeled "Graph.xmf". The code is as follows:

```

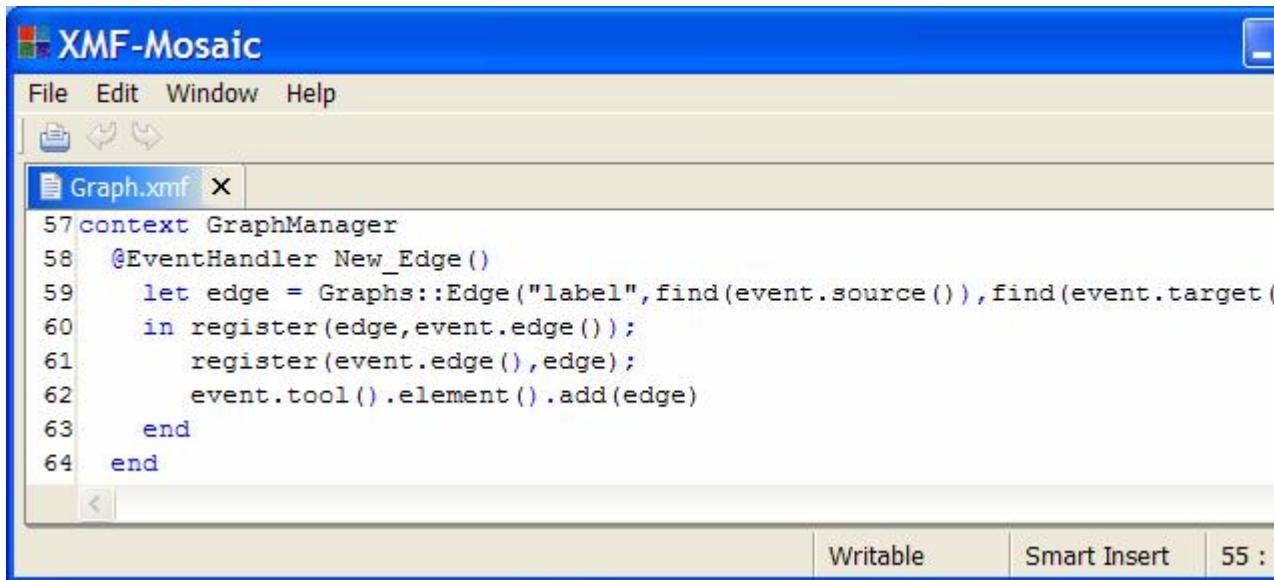
46 context Root
47   @ElementManager GraphManager
48   @EventHandler New_Node()
49     let node = Graphs::Node("label")
50     in register(event.node(), node);
51     register(node, event.node());
52     event.tool().element().add(node)
53   end
54 end
55 end

```

At the bottom of the code editor, there are three buttons: "Writable", "Smart Insert", and "55 :".

Line 49 creates a new graph node corresponding to the tool node whose creation raised the event. Once created, the two nodes are associated using the register operation in lines 50 and 51. Registration is necessary so that modifications to either node can map to, and therefore modify, the other node. Finally, the domain element is modified by adding the new node to the graph.

A new edge raises an event that is handled by the following definition:



The screenshot shows the XMF-Mosaic interface with a code editor window titled "Graph.xmf". The code is as follows:

```

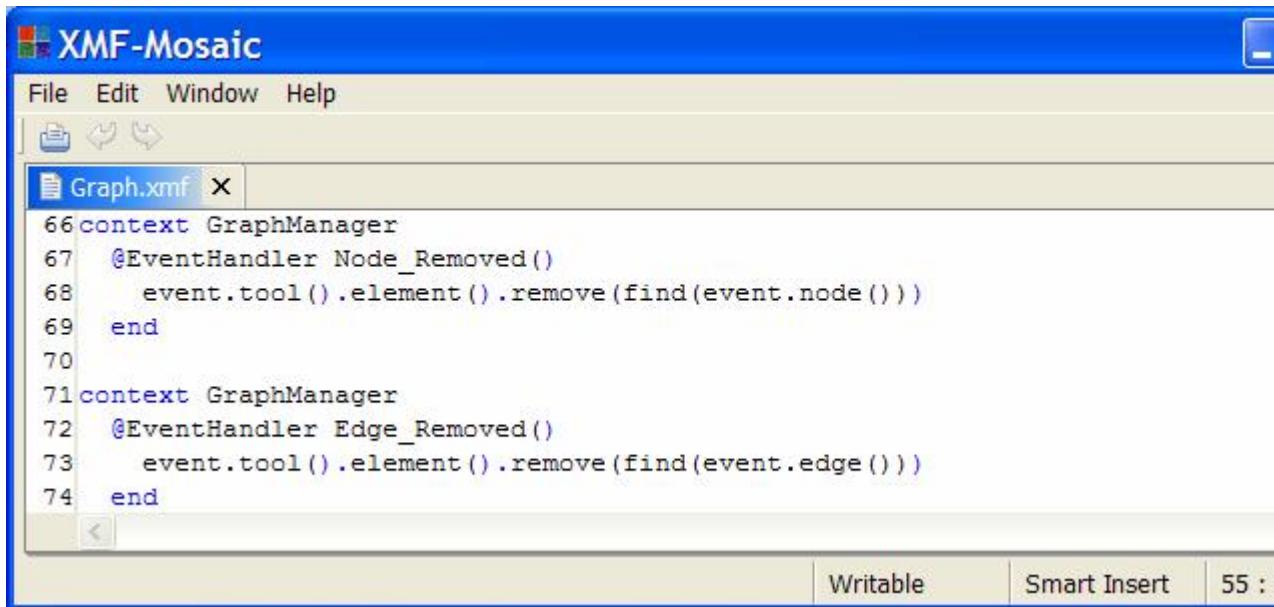
57 context GraphManager
58 @EventHandler New_Edge()
59   let edge = Graphs::Edge("label", find(event.source()), find(event.target()))
60   in register(edge, event.edge());
61   register(event.edge(), edge);
62   event.tool().element().add(edge)
63 end
64 end

```

At the bottom right of the code editor are buttons for "Writable", "Smart Insert", and the line number "55".

An edge creation event contains the new edge and its source and target nodes. Since the source and target nodes must have been created previously, they must have been registered by a node creation event handler. Therefore, the find operation can be used in line 59 to map from tool nodes to graph nodes when creating a graph edge. Once created, the edges are registered against one another in lines 60 and 61. The domain element is modified in line 62.

The tool type for a graph editor allows the nodes and edges to be removed via a menu action. When a tool element is deleted, an appropriate event is raised. The following handlers deal with node and edge deletion:



The screenshot shows the XMF-Mosaic interface with a code editor window titled "Graph.xmf". The code is as follows:

```

66 context GraphManager
67 @EventHandler Node_Removed()
68   event.tool().element().remove(find(event.node()))
69 end
70
71 context GraphManager
72 @EventHandler Edge_Removed()
73   event.tool().element().remove(find(event.edge()))
74 end

```

At the bottom right of the code editor are buttons for "Writable", "Smart Insert", and the line number "55".

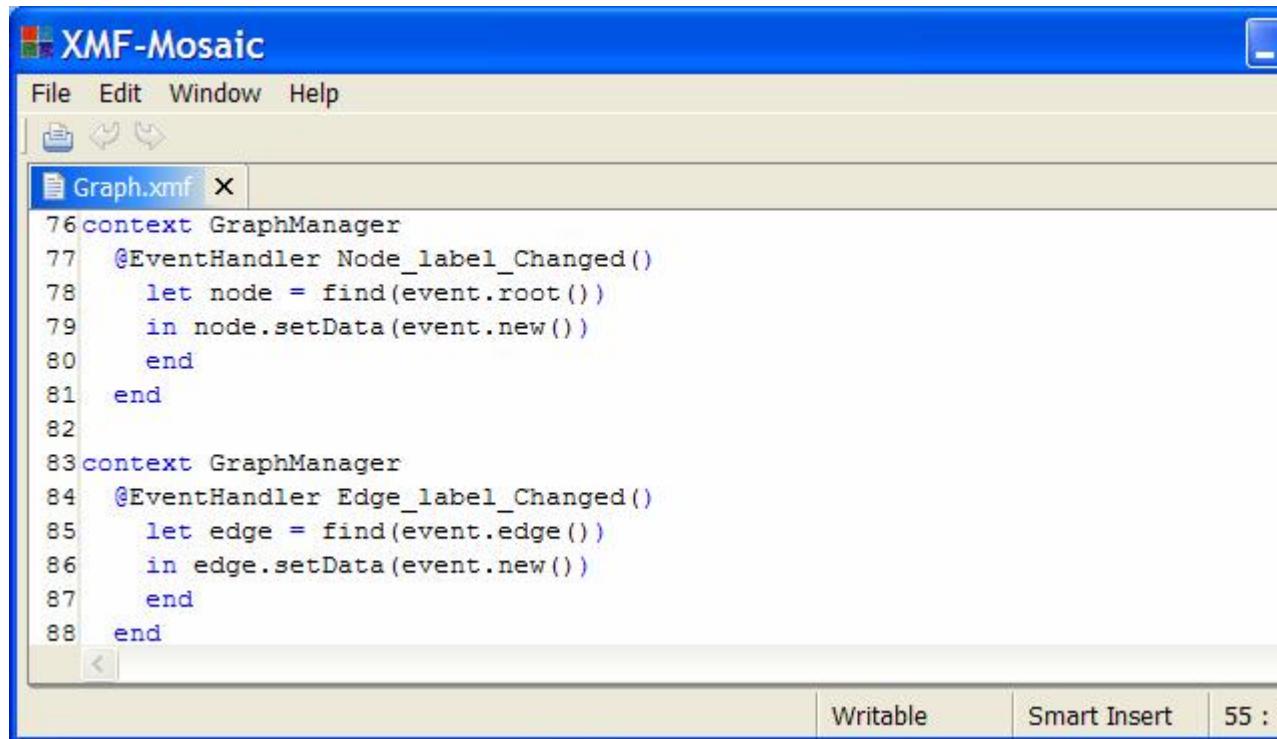
The find operation is used to map from the tool node to the graph node in lines 68 and 73. The graph elements are then removed from the graph.

The tool type for a graph editor specifies that a node and an edge both have labels. In the case of a node, the label is defined as a text attribute named `label`. Since we did not specify that these are read only, they can be modified on the diagram. When modifications take place, an appropriate event is raised.

In both cases the diagram components are tool attributes named `label`. Events that are raised in response to modifications to contained tool elements (such as a text component of a node or an edge label) have names that are based on the path from the root container of the element. For example, a modification to

a node label gives rise to an event named Node_label_Changed where the path from the root container to the modified element is Node_label. This rule applies no matter how deeply nested the modified element is.

The handlers for label modifications are given below.



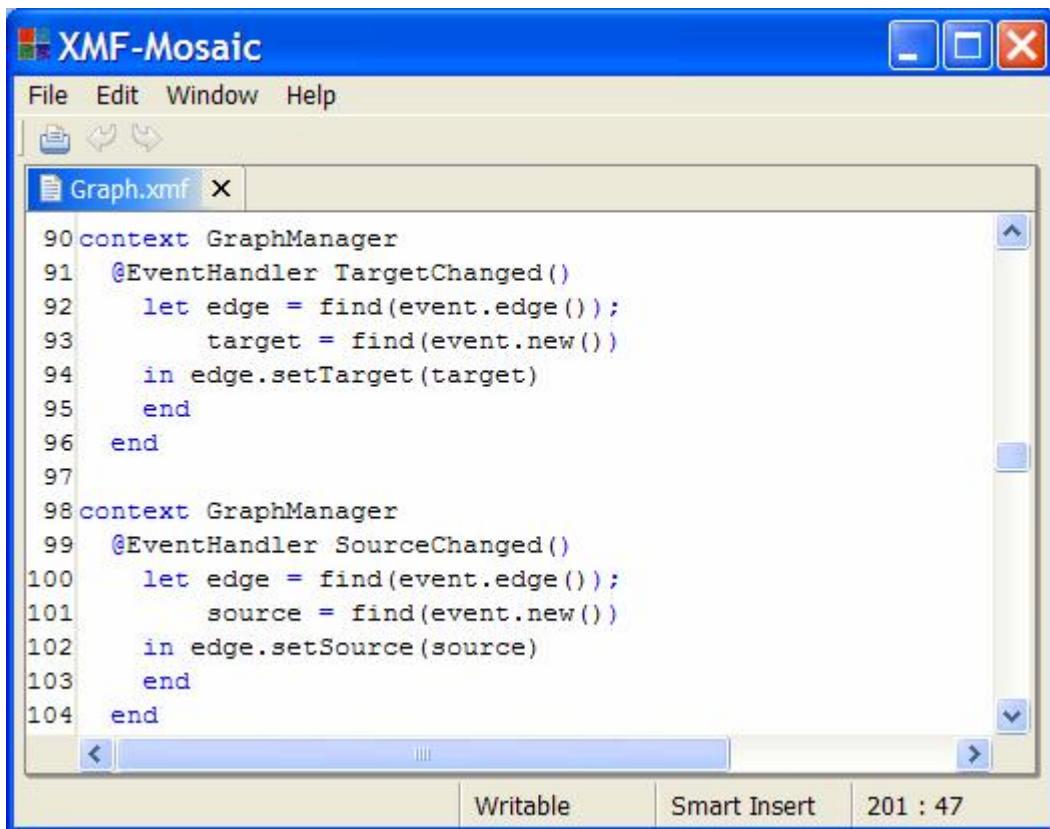
```

File Edit Window Help
Graph.xmf x
76 context GraphManager
77 @EventHandler Node_label_Changed()
78   let node = find(event.root())
79   in node.setData(event.new())
80 end
81 end
82
83 context GraphManager
84 @EventHandler Edge_label_Changed()
85   let edge = find(event.edge())
86   in edge.setData(event.new())
87 end
88 end

```

Writable Smart Insert 55 :

A node change event has an operation root() that returns the root container of the modified element. This is used in line 78 to find the graph node to modify. A change event also contains a value() operation that returns the modified tool element; this operation is not required for this example. A change event returns the value before the change as old() and the value after the change as new(). Line 79 changes the data on the graph node to be the modified label text.



The screenshot shows the XMF-Mosaic editor window. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main area is a code editor titled "Graph.xmf" containing the following code:

```

90 context GraphManager
91   @EventHandler TargetChanged()
92     let edge = find(event.edge());
93       target = find(event.new())
94       in edge.setTarget(target)
95     end
96   end
97
98 context GraphManager
99   @EventHandler SourceChanged()
100    let edge = find(event.edge());
101      source = find(event.new())
102      in edge.setSource(source)
103    end
104  end

```

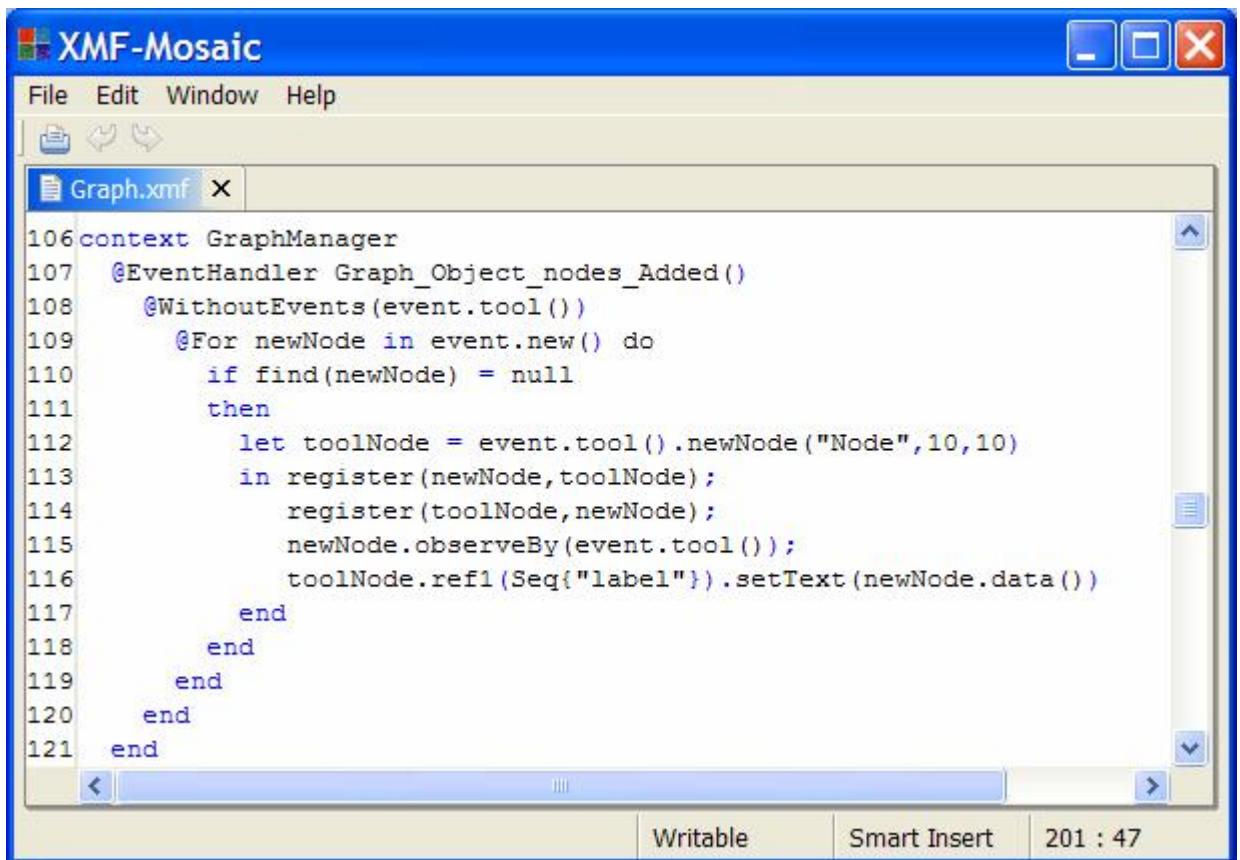
The status bar at the bottom shows "Writable", "Smart Insert", and "201 : 47".

This concludes the event handlers for events arising as a result of changes to the diagram.

Handling Managed Element Events

When a node is added to the graph we must modify the diagram on the tool to keep it in sync. Each event handler for managed element modifications has a name consisting of the type of the changed element, the word Object the name of the slot that is modified and the word Changed. The event has operations to access to the new and old values of the slot.

The following handler deals with changes to the nodes of the graph:



The screenshot shows the XMF-Mosaic editor window with the title bar "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main area displays the content of a file named "Graph.xmf". The code is as follows:

```

106 context GraphManager
107   @EventHandler Graph_Object_nodes_Added()
108     @WithoutEvents(event.tool())
109       @For newNode in event.new() do
110         if find(newNode) = null
111           then
112             let toolNode = event.tool().newNode("Node", 10, 10)
113               in register(newNode, toolNode);
114                 register(toolNode, newNode);
115                   newNode.observeBy(event.tool());
116                     toolNode.ref1(Seq{"label"}).setText(newNode.data())
117                     end
118                     end
119                     end
120                     end
121   end

```

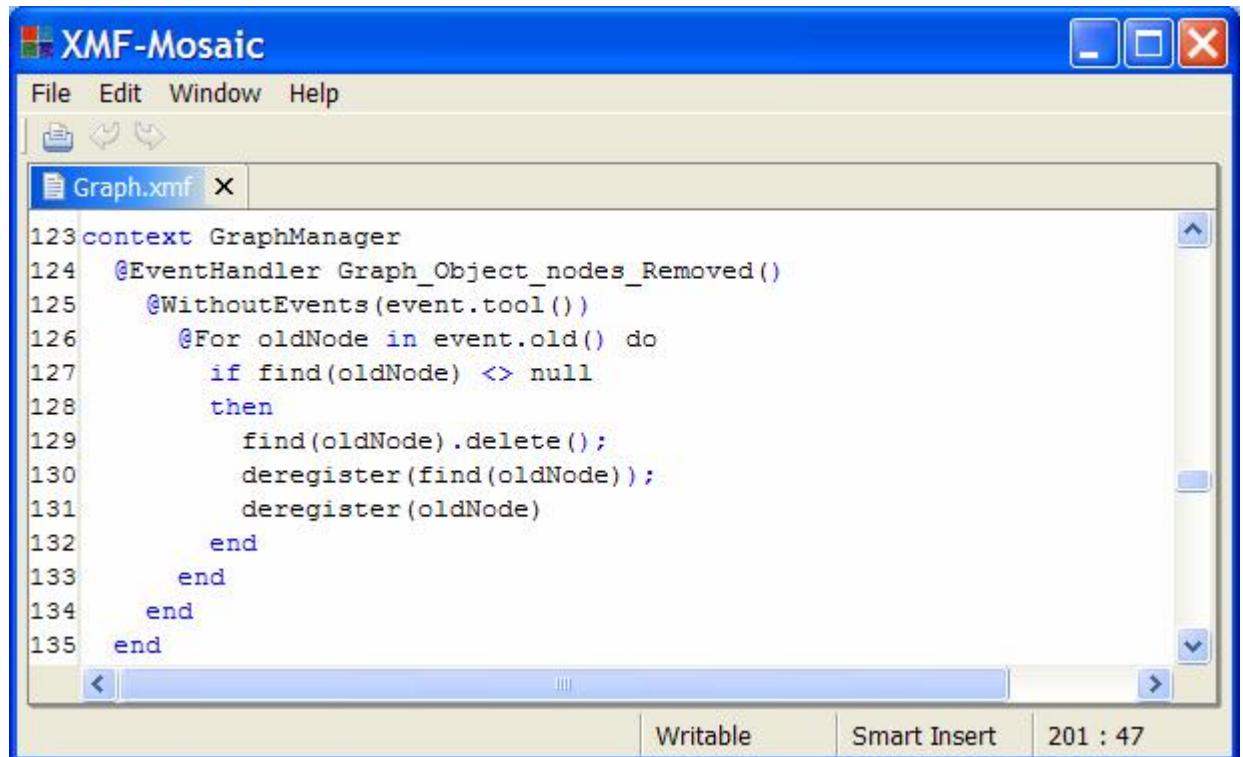
The status bar at the bottom right shows "Writable", "Smart Insert", and "201 : 47".

When designing event handlers for Xtools that process events from both the user interface and the managed element it is necessary to decide on a policy for preventing the same change being propagated back and forth. Consider the case of adding a new node on the diagram. This raises a New_Node event causing the managed graph to be updated with a new node. When this occurs, a Graph_Object_nodes_Added event is raised; since this event may have been caused by some external agent adding a new node to the graph, a reasonable implementation for this handler is to modify the diagram interface to add a new node. If we do this, we cause an infinite ping-pong between the user interface and the managed element.

To address this issue it is sensible to take each pair of dependent handlers and to ensure that events bottom out. This can be achieved by preventing events occurring in one of the handlers and by checking that actions have not already occurred as shown above in lines 108 and 110. When a new node is added to the managed graph in the handler for New_Node, a Graph_Object_nodes_Added event is raised; when this event is handled no New_Node event is raised (although the diagram will be updated as we will see). This ensures that diagram changes update the graph and graph changes update the diagram but that this does not lead to infinite regress. Note that it is possible to suppress events in both handlers, but this is not necessary; if you do suppress events in both then it is not necessary to include a registration check as in line 110 since neither handler will cause the other to be invoked.

Line 112 shows the creation of a new instance of the diagram node type Node. Xtools allow new node types to be instantiated at a given position on the diagram using the newNode operation. Once the new node is created, it is registered in lines 113 and 114. The label on the new node is updated to contain the graph node label in line 116.

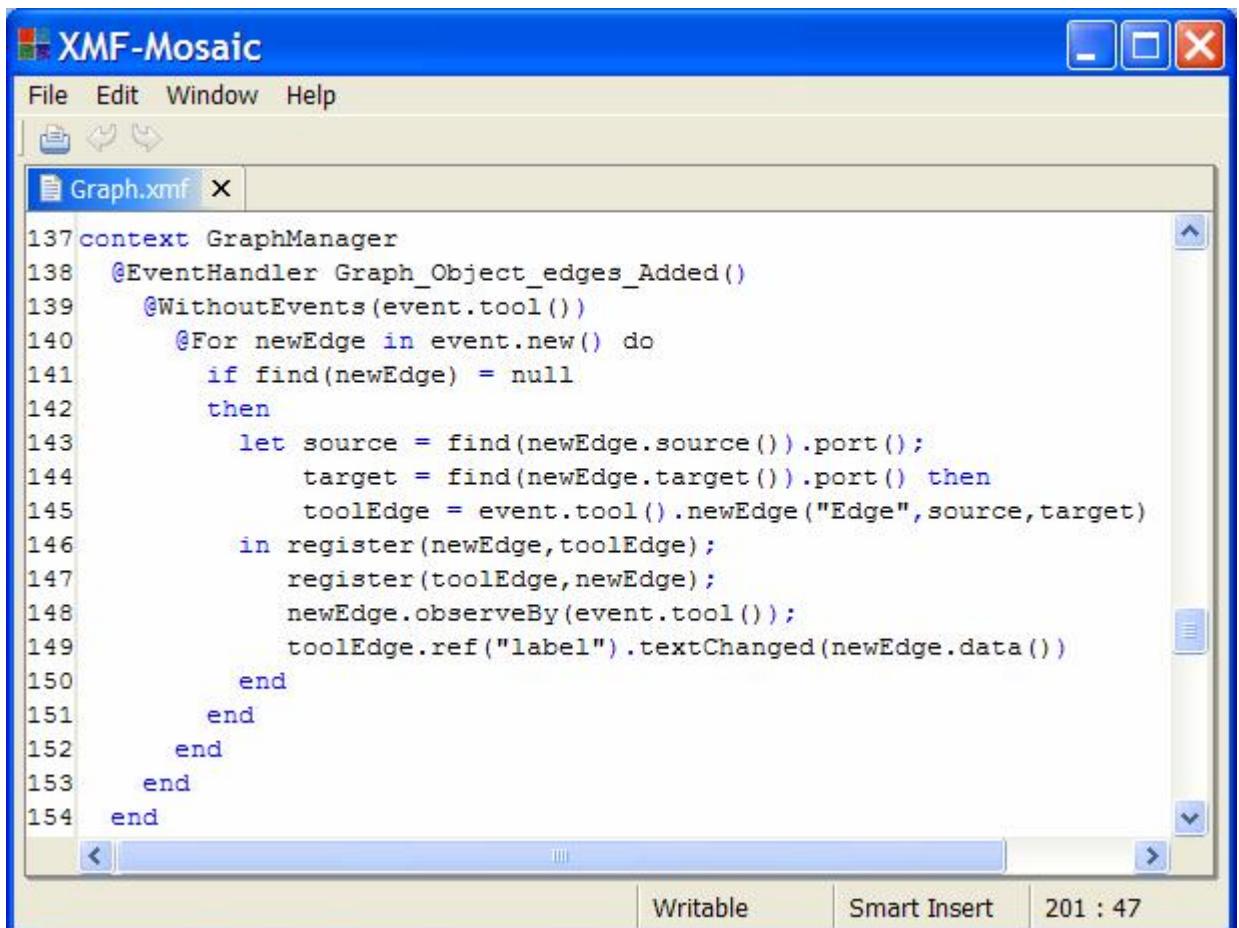
The following shows how the removal of nodes from the managed graph is handled:



The screenshot shows the XMF-Mosaic editor window with the title bar "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main area displays a code editor titled "Graph.xmf". The code is a snippet of Xtext grammar, specifically for a "GraphManager" context. It defines an event handler for "Graph_Object_nodes_Removed" events. The handler iterates over the removed nodes using "@For oldNode in event.old() do". For each node, it checks if it was previously found using "find(oldNode) <> null", then deletes it with "find(oldNode).delete()", deregisters it from the tool with "deregister(find(oldNode))", and finally deregisters it from the tool with "deregister(oldNode)". The code is numbered from 123 to 135. At the bottom of the editor are buttons for "Writable", "Smart Insert", and status information "201 : 47".

```
123 context GraphManager
124   @EventHandler Graph_Object_nodes_Removed()
125     @WithoutEvents(event.tool())
126       @For oldNode in event.old() do
127         if find(oldNode) <> null
128           then
129             find(oldNode).delete();
130             deregister(find(oldNode));
131             deregister(oldNode)
132           end
133         end
134       end
135     end
```

The nodes that have been removed are supplied as the old part of the event. We should only remove nodes if they are currently registered (line 127). All display elements implement a delete operation (line 129) that removes the element from the diagram (and fires events if they are enabled). Finally the elements are deregistered from the tool (lines 130 and 131).



```

137 context GraphManager
138   @EventHandler Graph_Object_edges_Added()
139     @WithoutEvents(event.tool())
140       @For newEdge in event.new() do
141         if find(newEdge) = null
142           then
143             let source = find(newEdge.source()).port();
144               target = find(newEdge.target()).port() then
145                 toolEdge = event.tool().newEdge("Edge",source,target)
146                   in register(newEdge,toolEdge);
147                     register(toolEdge,newEdge);
148                       newEdge.observeBy(event.tool());
149                         toolEdge.ref("label").textChanged(newEdge.data())
150                           end
151                             end
152                               end
153                                 end
154                                   end

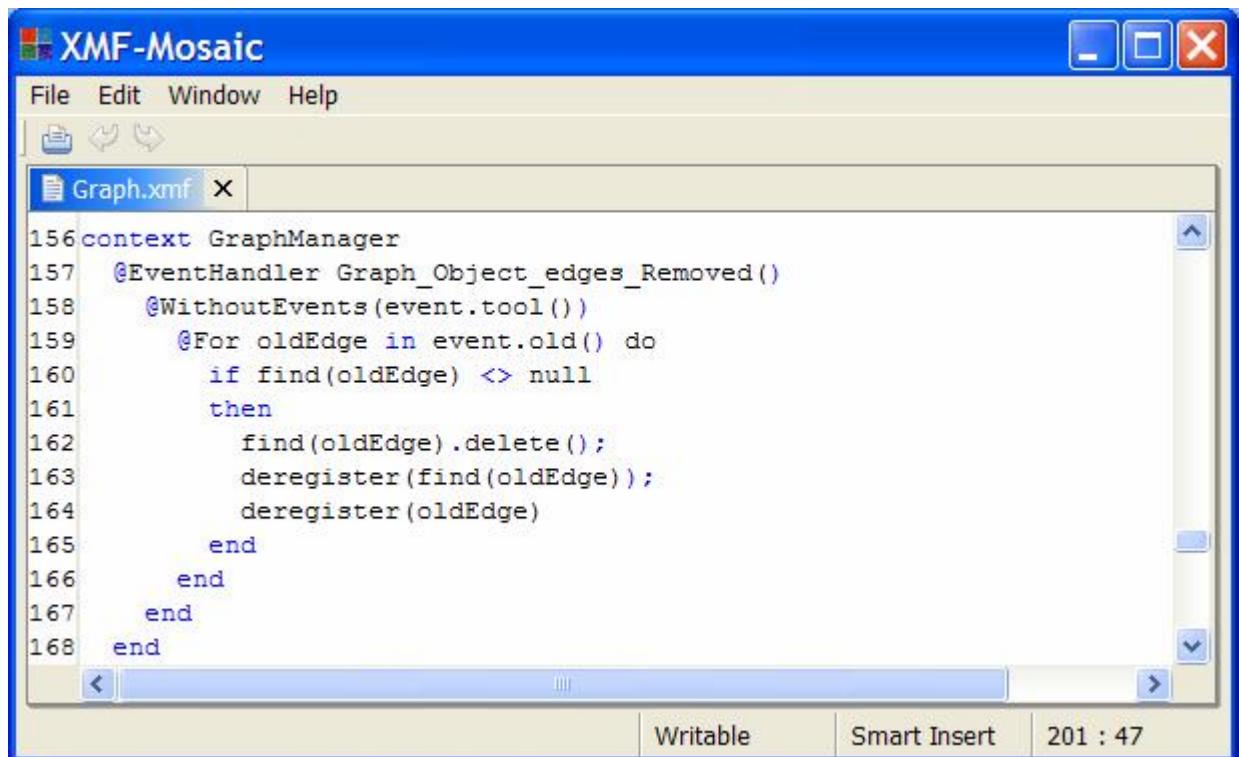
```

Writable Smart Insert 201 : 47

When an edge is added to the graph we must create an edge on the diagram. Diagram edges are attached to ports that are contained in nodes. A node may have more than one port (some may be associated with display components of the node), but providing that the node has at least one port, a port is produced by calling the port operation of the diagram node (lines 143 and 144).

A diagram tool provides a newEdge operation that draws a new edge between ports on the diagram (line 145). In order to register when the ends of an edge change we must observe the edge (line 148). The text in labels on edges are changed using the textChanged operation (line 149).

When an edge is removed from a graph that is monitored by a tool, the following event is raised. The handler, deletes the diagram element and deregisters the appropriate elements:



The screenshot shows the XMF-Mosaic editor window with the title bar "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". The toolbar has icons for file operations. The main area displays a code editor titled "Graph.xmf" containing the following Groovy script:

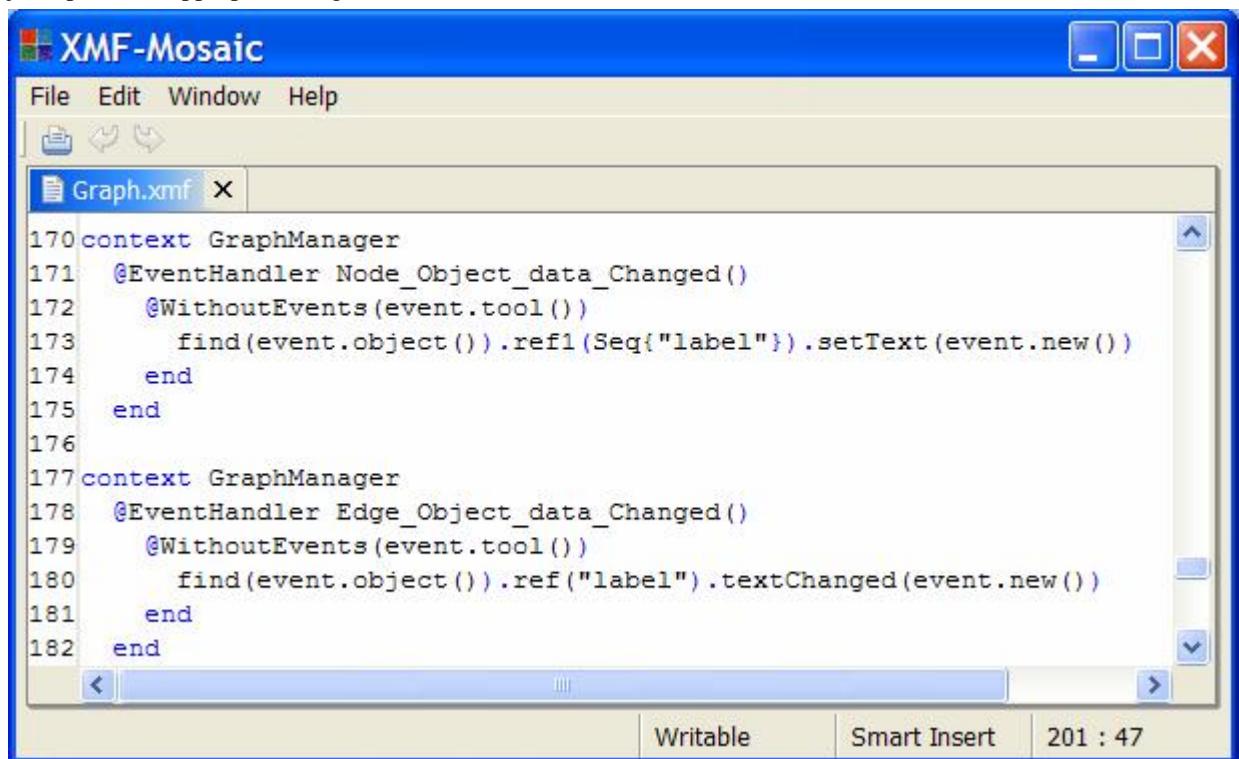
```

156 context GraphManager
157     @EventHandler Graph_Object_edges_Removed()
158         @WithoutEvents(event.tool())
159             @For oldEdge in event.old() do
160                 if find(oldEdge) <> null
161                     then
162                         find(oldEdge).delete();
163                         deregister(find(oldEdge));
164                         deregister(oldEdge)
165                     end
166                 end
167             end
168         end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and the line number "201 : 47".

When the data on a monitored graph node or edge changes the following events are raised. The handlers just update the appropriate diagram elements:



The screenshot shows the XMF-Mosaic editor window with the title bar "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". The toolbar has icons for file operations. The main area displays a code editor titled "Graph.xmf" containing the following Groovy script:

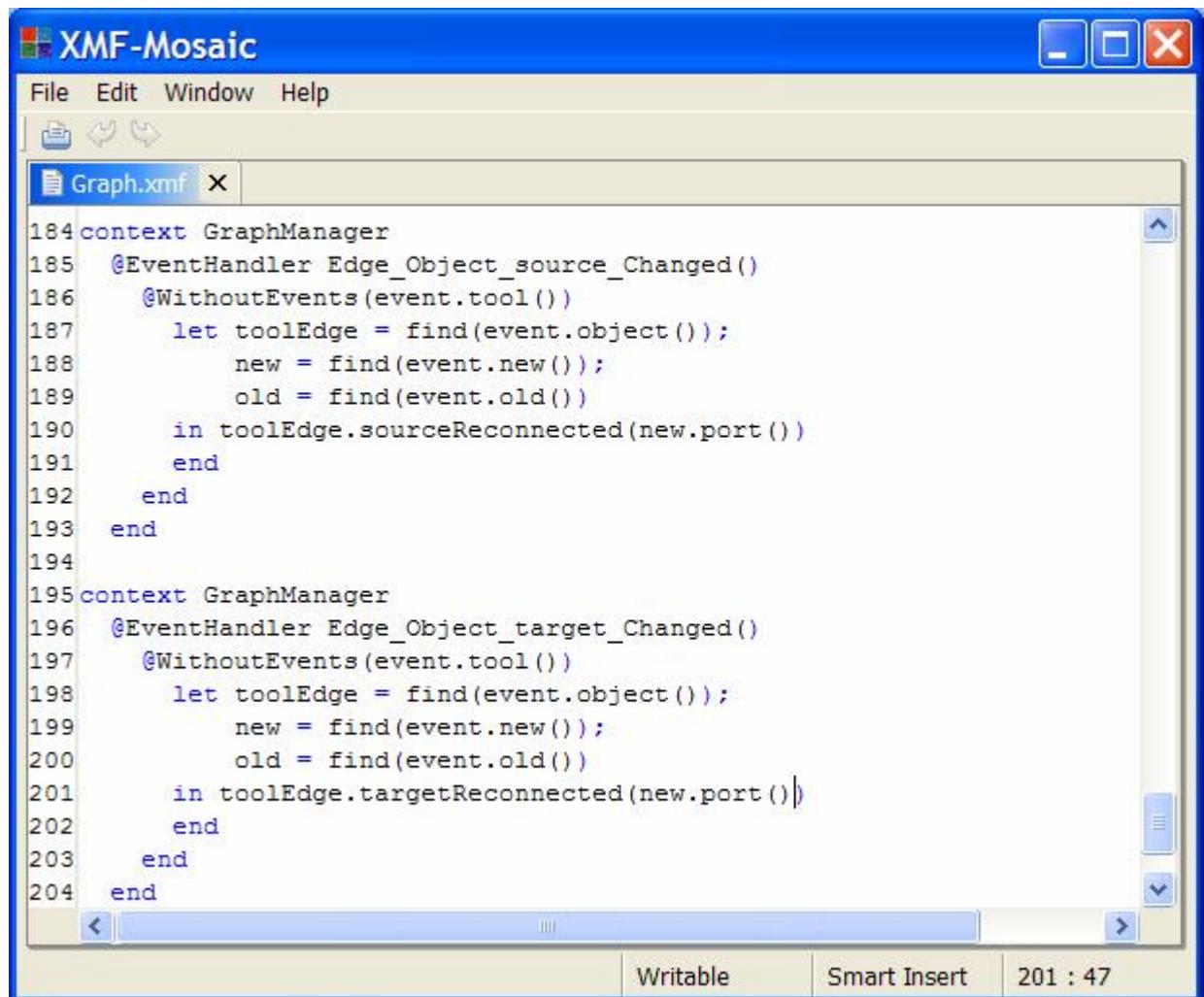
```

170 context GraphManager
171     @EventHandler Node_Object_data_Changed()
172         @WithoutEvents(event.tool())
173             find(event.object()).ref1(Seq{"label"}).setText(event.new())
174         end
175     end
176
177 context GraphManager
178     @EventHandler Edge_Object_data_Changed()
179         @WithoutEvents(event.tool())
180             find(event.object()).ref("label").textChanged(event.new())
181         end
182     end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and the line number "201 : 47".

Finally, the source and target of a monitored graph edge may change. A diagram edge provides operations `sourceReconnected` and `targetReconnected` that are used to change the corresponding port that the edge connects to.



```

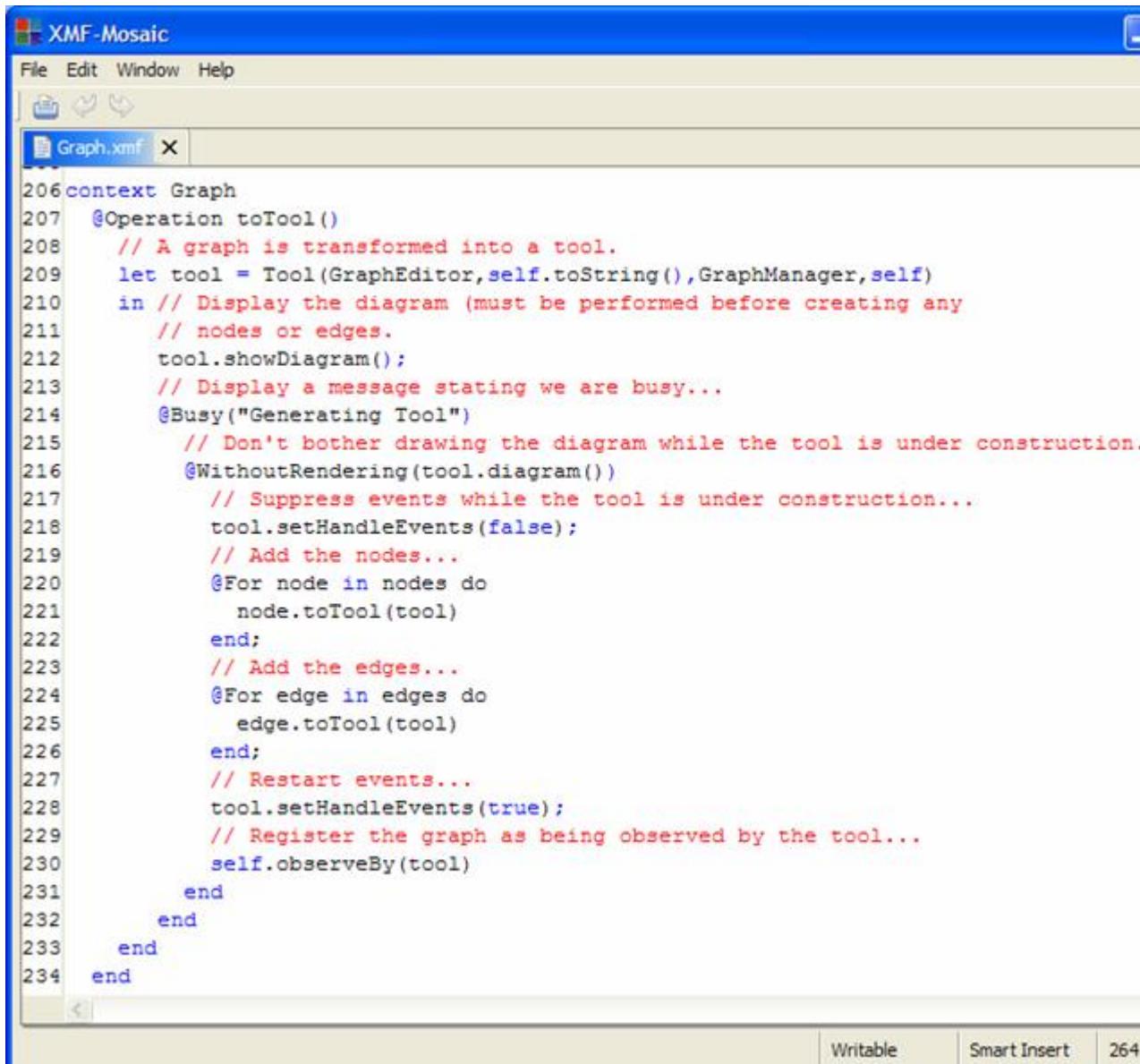
184 context GraphManager
185 @EventHandler Edge_Object_source_Changed()
186 @WithoutEvents(event.tool())
187 let toolEdge = find(event.object());
188 new = find(event.new());
189 old = find(event.old())
190 in toolEdge.sourceReconnected(new.port())
191 end
192 end
193 end
194
195 context GraphManager
196 @EventHandler Edge_Object_target_Changed()
197 @WithoutEvents(event.tool())
198 let toolEdge = find(event.object());
199 new = find(event.new());
200 old = find(event.old())
201 in toolEdge.targetReconnected(new.port())
202 end
203 end
204 end

```

Creating a Tool from a Domain Model Instance

It is often the case that we have an instance of the domain model and wish to create a tool that displays the current state of the instance and is then used to extend and modify the instance. Using XTools this is quite straightforward. We map from the domain instance to a tool and create instances of the appropriate tool types. This section shows how this is achieved for our graph editor tool.

To generate a tool from a graph we must create the new tool with the appropriate tool type and element manager and then map the nodes and edges of the graph to instances of the appropriate tool types. The following operation definition adds a new operation to Graph that implements the mapping. The source code contains comments explaining each of the steps in the mapping:



The screenshot shows the XMF-Mosaic interface with the 'Graph.xmf' file open in the main editor window. The code is written in Xtext, defining an operation 'toTool' for a 'Graph' context. The code handles the transformation of a graph into a tool, including displaying the diagram, showing a busy message, and registering the graph as an observer. The code is color-coded for syntax highlighting.

```

206 context Graph
207     @Operation toTool()
208         // A graph is transformed into a tool.
209         let tool = Tool(GraphEditor, self.toString(), GraphManager, self)
210         in // Display the diagram (must be performed before creating any
211             // nodes or edges.
212             tool.showDiagram();
213             // Display a message stating we are busy...
214             @Busy("Generating Tool")
215                 // Don't bother drawing the diagram while the tool is under construction
216                 @WithoutRendering(tool.diagram())
217                     // Suppress events while the tool is under construction...
218                     tool.setHandleEvents(false);
219                     // Add the nodes...
220                     @For node in nodes do
221                         node.toTool(tool)
222                     end;
223                     // Add the edges...
224                     @For edge in edges do
225                         edge.toTool(tool)
226                     end;
227                     // Restart events...
228                     tool.setHandleEvents(true);
229                     // Register the graph as being observed by the tool...
230                     self.observeBy(tool)
231                 end
232             end
233         end
234     end

```

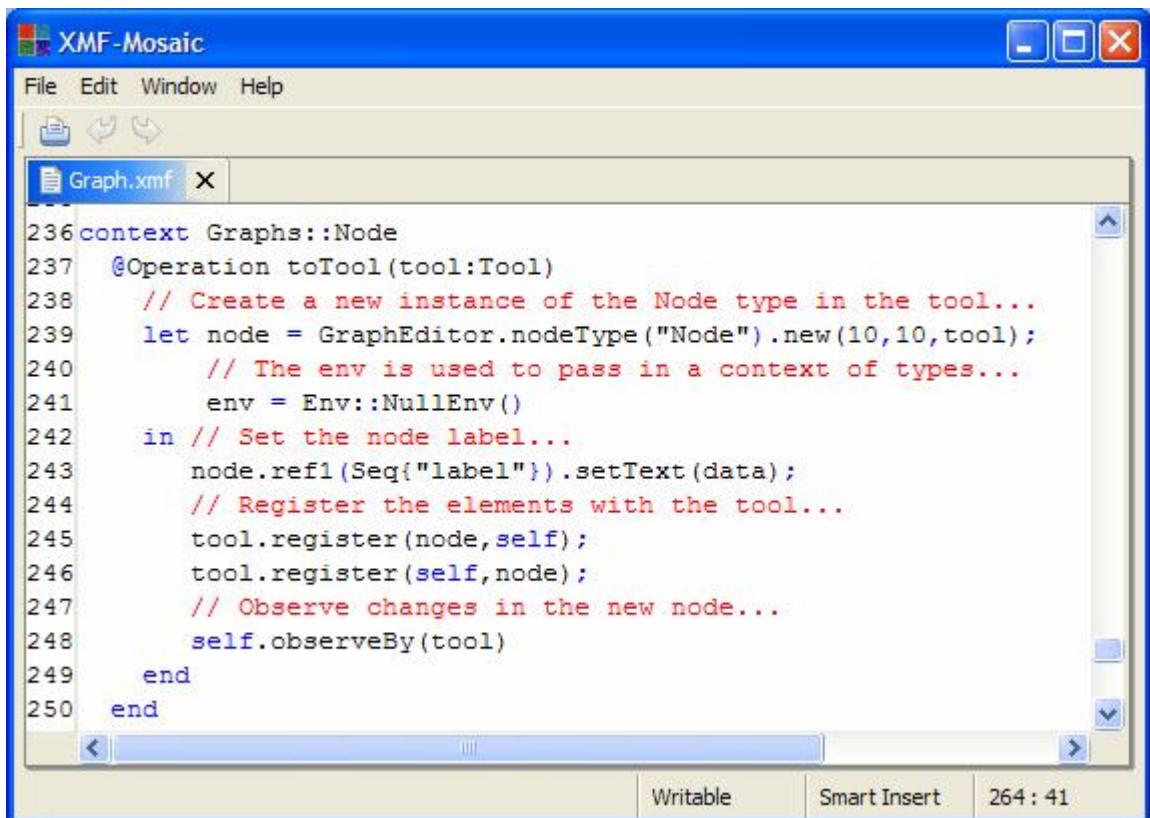
Writable

Smart Insert

264

The definition of `Graph::toTool` is a typical example of how to define a mapping from a domain instance to an Xtool. We create a tool in line 209. The constructor arguments for a tool are the type, an id string, the manager and the domain instance. Since a graph is a container of nodes and edges, we use auxiliary definitions of `toTool` to map these elements passing the new tool as an argument. Finally, in line 230 if we want changes in the domain instance to be reflected in the tool we must register the tool as an observer.

The following operation shows how a graph node is mapped to a diagram node:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. A central editor window titled "Graph.xmf" contains the following code:

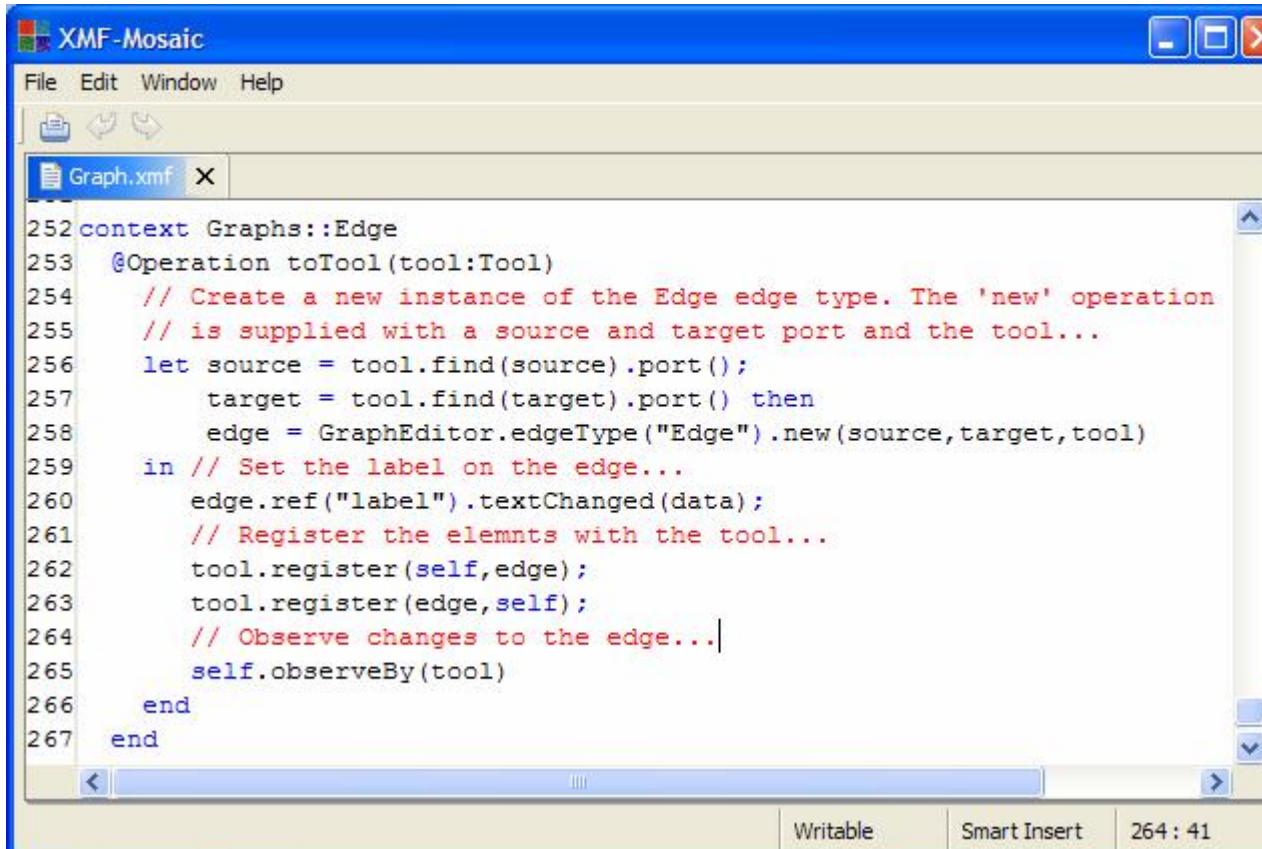
```

236 context Graphs::Node
237 @Operation toTool(tool:Tool)
238     // Create a new instance of the Node type in the tool...
239     let node = GraphEditor.nodeType("Node").new(10,10,tool);
240         // The env is used to pass in a context of types...
241         env = Env::NullEnv()
242     in // Set the node label...
243         node.ref1(Seq{"label"}).setText(data);
244         // Register the elements with the tool...
245         tool.register(node,self);
246         tool.register(self,node);
247         // Observe changes in the new node...
248         self.observeBy(tool)
249     end
250 end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and the time "264:41".

Node and edges types can be referenced in a tool type:



The screenshot shows the XMF-Mosaic IDE interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. A central editor window titled "Graph.xmf" contains the following code:

```

252 context Graphs::Edge
253 @Operation toTool(tool:Tool)
254     // Create a new instance of the Edge edge type. The 'new' operation
255     // is supplied with a source and target port and the tool...
256     let source = tool.find(source).port();
257         target = tool.find(target).port() then
258             edge = GraphEditor.edgeType("Edge").new(source,target,tool)
259         in // Set the label on the edge...
260             edge.ref("label").textChanged(data);
261             // Register the elements with the tool...
262             tool.register(self,edge);
263             tool.register(edge,self);
264             // Observe changes to the edge...
265             self.observeBy(tool)
266         end
267     end

```

The status bar at the bottom right shows "Writable", "Smart Insert", and the time "264:41".

Display Elements

Introduction

XTool node types consist of a collection of display element types. Each display element type is a named attribute of the node. When a node is created, the display type is instantiated and its instance can be referred to by giving the path from the node (the root) to the named display element.

Each display element type has a textual syntax that is used to define it. This section shows you how to define display element types by showing each one in the context of the type Node in our graph editor example. The general form for the textual definition of a display element is:

```
@ElementType name(properties)
    attributes
    display components
    menu
end
```

The properties of the element type are names and values enclosed in parentheses. If no properties are defined then the parentheses should be omitted. The attributes of a display element are name/value pairs of the form name = value. Typically, the attributes specify the maximum and minimum dimensions of the element and features such as color. The display components of an element are the contained elements; only boxes may contain sub-components. Each element may have a menu.

All elements can be associated with a port in the element's root node. By specifying the property hasport, you associate the instances of the element type with a port such that the port will move and resize with the instance relative to the containing node.

Most elements can have a colors. The value of a color attribute should be one of the predefined colors: red, green, blue, or should be a string "r,g,b" where the three components are integers in the range 0 to 255 specifying the red, green and blue color contributions respectively.

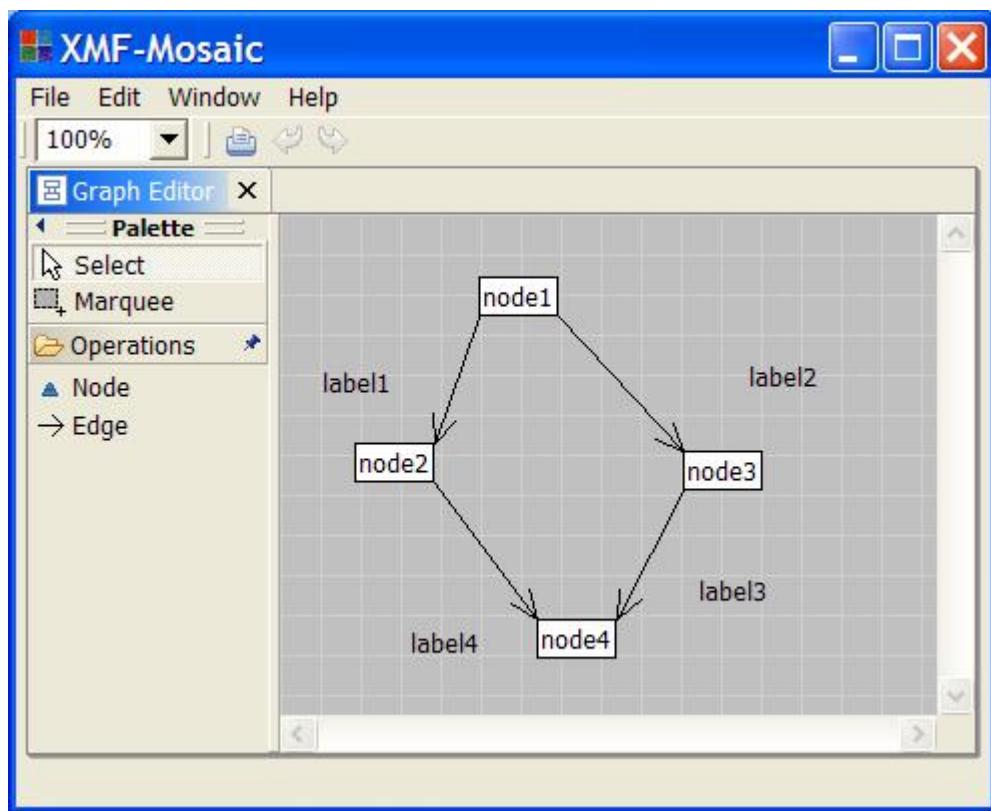
Many types can be specified with minimum and maximum dimensions. By stating these attributes you are placing bounds on the amount that instances of the type can be resized. Specifying the same value for the minimum and maximum dimensions requires that instances of the type are of a fixed size.

Many types can be padded. A padding attribute specifies the amount of space that should be left surrounding instances of the type when placing the instance in a display container (such as a box or a node). Padding can be specified to the left, right, on top and below elements. If the padding attributes are omitted then it is assumed that no space need be left between adjacent elements and between elements and the borders of their containers.

All types may specify a layout property. The layout defines how instances of the type will be placed within their container. Elements may be aligned with their container in which case they stretch to fit the size of their container; elements may be left and right justified, and centered within a container. Interpretation of the layout property depends on the content layout property of the container.

Box

A box is a container of display elements. A box may have a border and may be filled or empty. A box specifies the layout format of its contained elements (layout is the subject of a later section). Suppose we want to have the label of a node contained within a box:



The corresponding node type definition is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE GraphBox SYSTEM "GraphBox.dtd">
<@NodeType Node(hasport)>
<20> // Place a box round the label...
<21> @Box nodeBox
<22> // Pad out the text in the box...
<23> @Text label "label" padLeft = 2 padRight = 2 padTop = 2 padBottom = 2 end
<24> end
<25> // A node has a delete menu action. Note that 'self' refers to the
<26> // menu container (i.e. the node)...
<27> @Menu
<28> @MenuAction Delete self.delete() end
<29> end
<30> end

```

The syntax for boxes is as follows:

```

@Box name(hasport,hide,contentLayout,layout,noFill)
minWidth = integer
minHeight = integer
maxWidth = integer
maxHeight = integer
cornerCurve = integer
padLeft = integer
padRight = integer

```

```

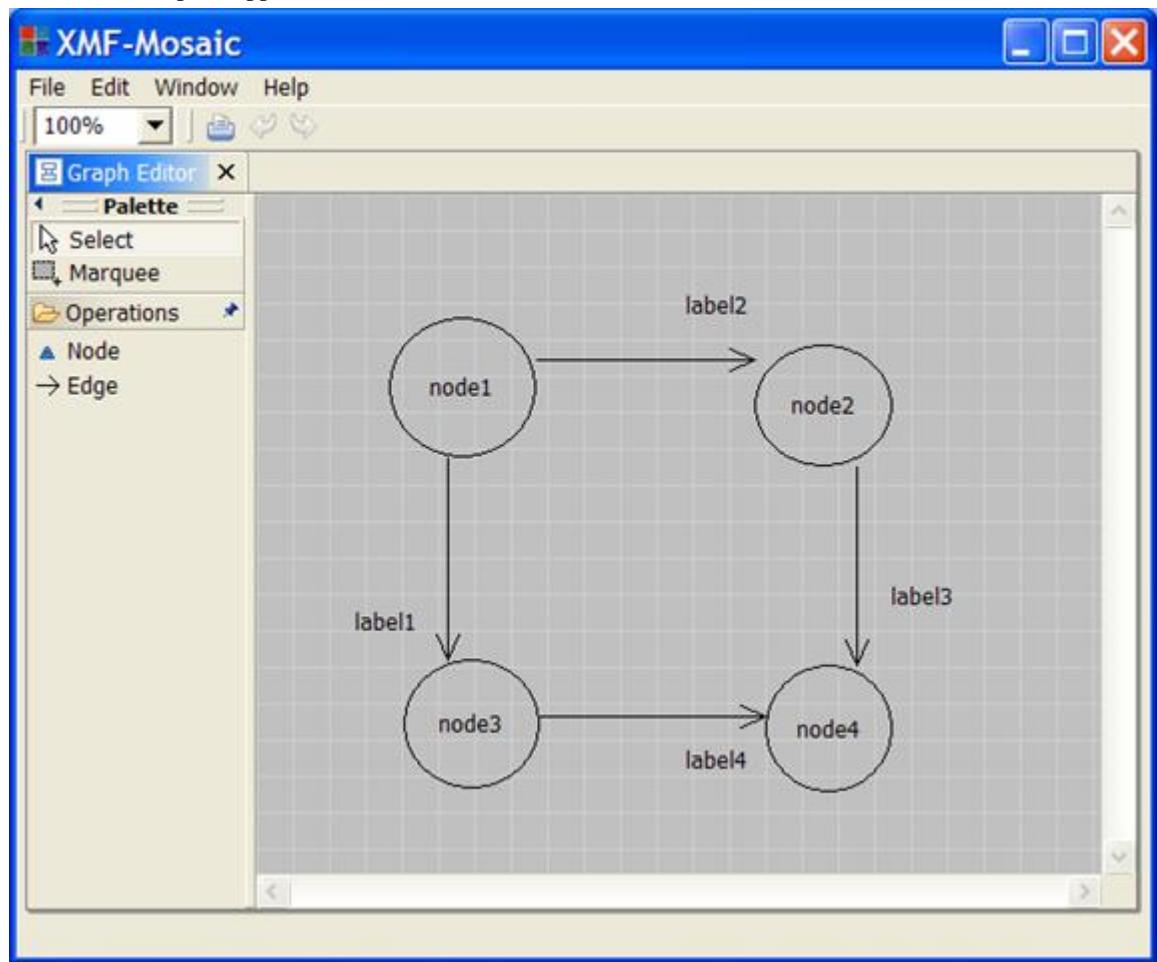
    paddingTop = integer
    paddingBottom = integer
    fillColor = color
    lineColor = color
end

```

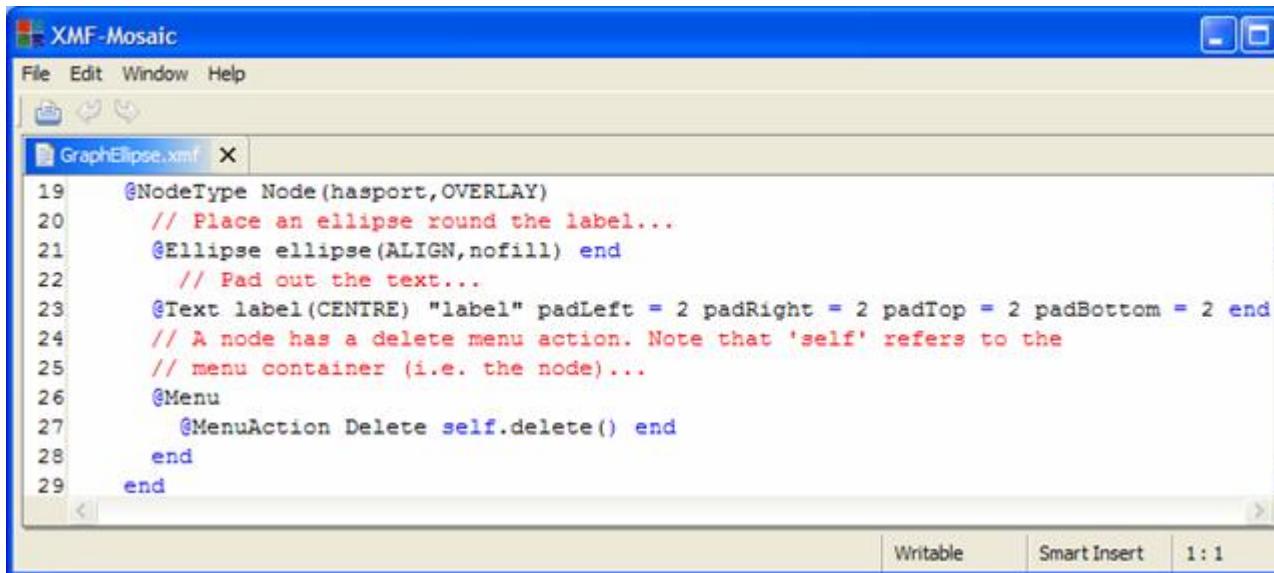
All components of a box definition are optional except for the name. If hasport is present then the box defines the location of a port in the root node. A hide directive is one of the following hideLeft, hideRight, hideTop or hideBottom and may be repeated. A content layout directive is one of the following HORIZONTAL, VERTICAL or OVERLAY. A layout directive is one of the following: LEFT, RIGHT, ALIGN. If nofill is present then the box is not filled. The minimum and maximum dimension attributes specify the sized below and above which the box cannot be resized. The padding attributes specify the white space that is left between the box and its container and adjacent elements.

Ellipse

Ellipses are used to draw ovals and circles. Ellipses are not containers of elements, but can be placed in boxes with overlay layout to give the impression that they contain element that are displayed over them. For example, suppose that we want to have nodes with circles around the labels:



The node type is defined as follows:



The screenshot shows the XMF-Mosaic editor interface. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main window displays an XML file named "GraphEllipse.xml". The code is as follows:

```

19     @NodeType Node(hasport,OVERLAY)
20         // Place an ellipse round the label...
21         @Ellipse ellipse(ALIGN,nofill) end
22             // Pad out the text...
23             @Text label(CENTRE) "label" padLeft = 2 padRight = 2 padTop = 2 padBottom = 2 end
24             // A node has a delete menu action. Note that 'self' refers to the
25             // menu container (i.e. the node)...
26             @Menu
27                 @MenuAction Delete self.delete() end
28             end
29         end

```

At the bottom right of the editor window, there are buttons for "Writable", "Smart Insert", and a ratio "1:1".

Note that the node type has content layout OVERLAY meaning that all the contents will be drawn over each other in the order that they appear in the type definition. The contained ellipse type has ALIGN layout so that instances grow and shrink with their container. Each ellipse is left unfilled.

The syntax for ellipses is as follows:

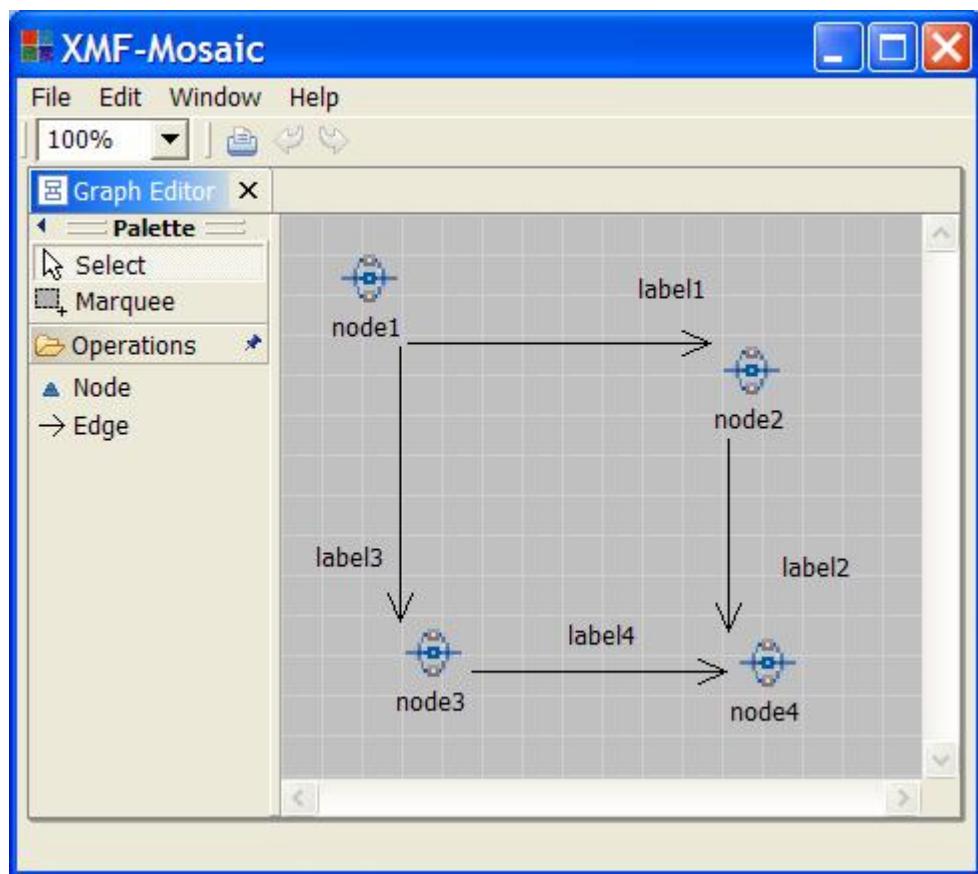
```

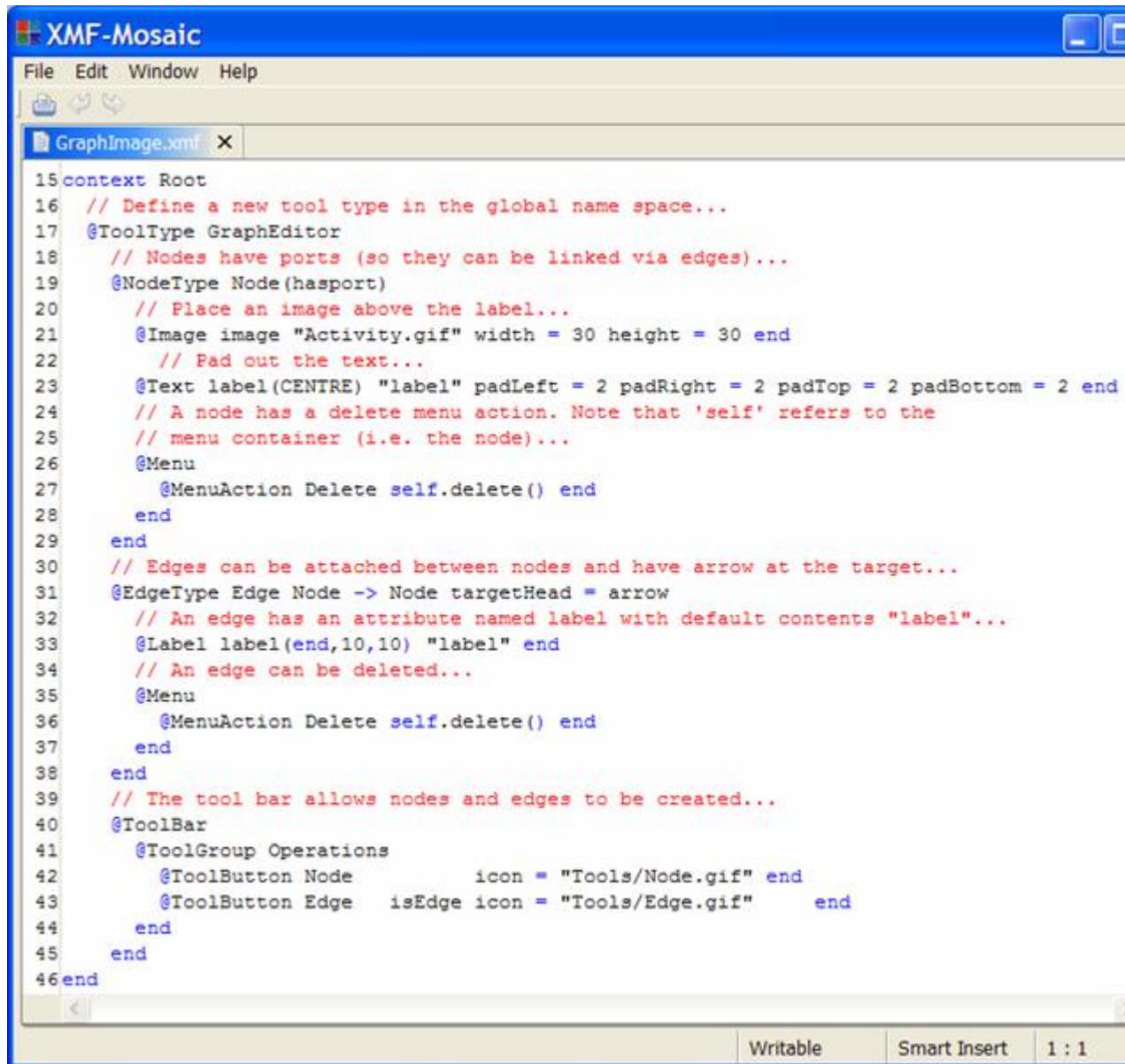
@Ellipse name(hasport,noOutline,layout,nofill)
    minWidth = integer
    minHeight = integer
    maxWidth = integer
    maxHeight = integer
    padLeft = integer
    padRight = integer
    padTop = integer
    padBottom = integer
    fillColor = color
    lineColor = color
end

```

All properties and attributes are optional.

Image





The screenshot shows the XMF-Mosaic application window. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main area is a code editor titled "GraphImage.xml". The code is an XML document with line numbers from 15 to 46. It defines a "Root" context with various configurations for "GraphEditor", "Node", "Image", "Text", "Menu", and "Edge" elements. It also describes a "ToolBar" with "Operations" and specific "ToolButton"s for "Node" and "Edge". The status bar at the bottom right shows "Writable", "Smart Insert", and "1 : 1".

```

15 context Root
16   // Define a new tool type in the global name space...
17   @ToolType GraphEditor
18     // Nodes have ports (so they can be linked via edges)...
19     @NodeType Node(hasport)
20       // Place an image above the label...
21       @Image image "Activity.gif" width = 30 height = 30 end
22         // Pad out the text...
23       @Text label(CENTRE) "label" padLeft = 2 padRight = 2 padTop = 2 padBottom = 2 end
24     // A node has a delete menu action. Note that 'self' refers to the
25     // menu container (i.e. the node)...
26     @Menu
27       @MenuAction Delete self.delete() end
28     end
29   end
30   // Edges can be attached between nodes and have arrow at the target...
31   @EdgeType Edge Node -> Node targetHead = arrow
32     // An edge has an attribute named label with default contents "label"...
33     @Label label(end,10,10) "label" end
34     // An edge can be deleted...
35     @Menu
36       @MenuAction Delete self.delete() end
37     end
38   end
39   // The tool bar allows nodes and edges to be created...
40   @ToolBar
41     @ToolGroup Operations
42       @ToolButton Node icon = "Tools/Node.gif" end
43       @ToolButton Edge isEdge icon = "Tools/Edge.gif" end
44     end
45   end
46 end

```

Text

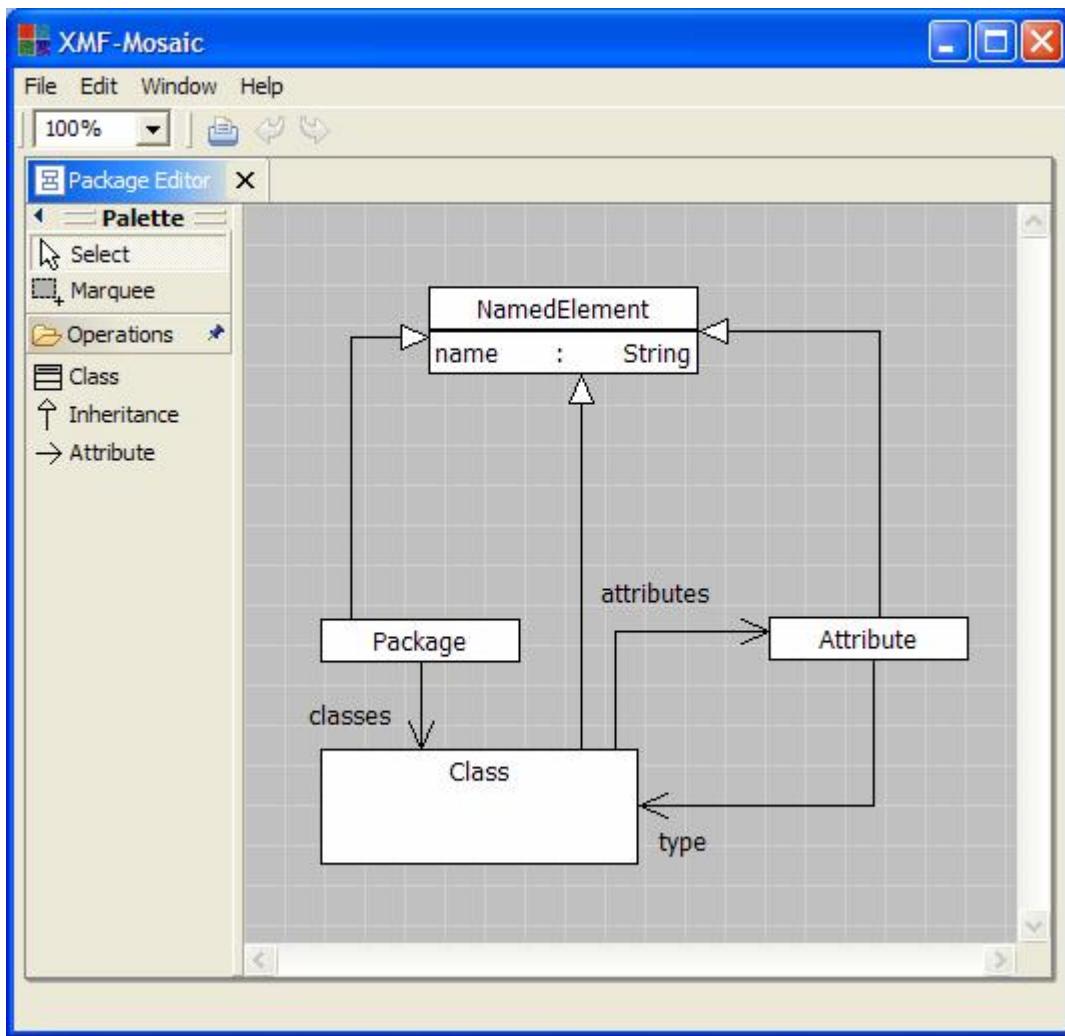
Diagram Layout

Element Layout

Container Layout

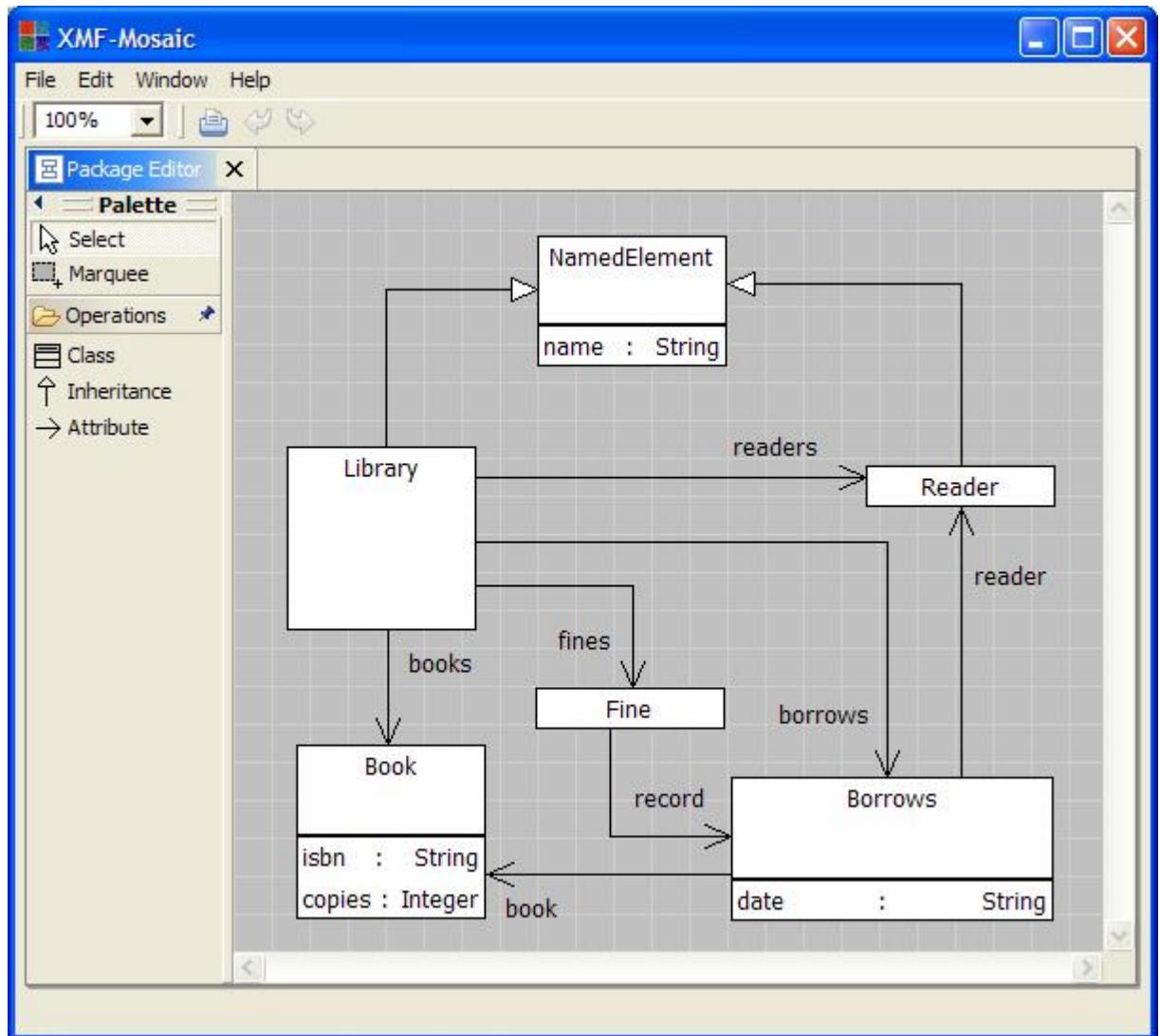
Example Tool: Class Diagrams

A typical example of a modelling tool is a class diagram editor. The domain model is shown below:

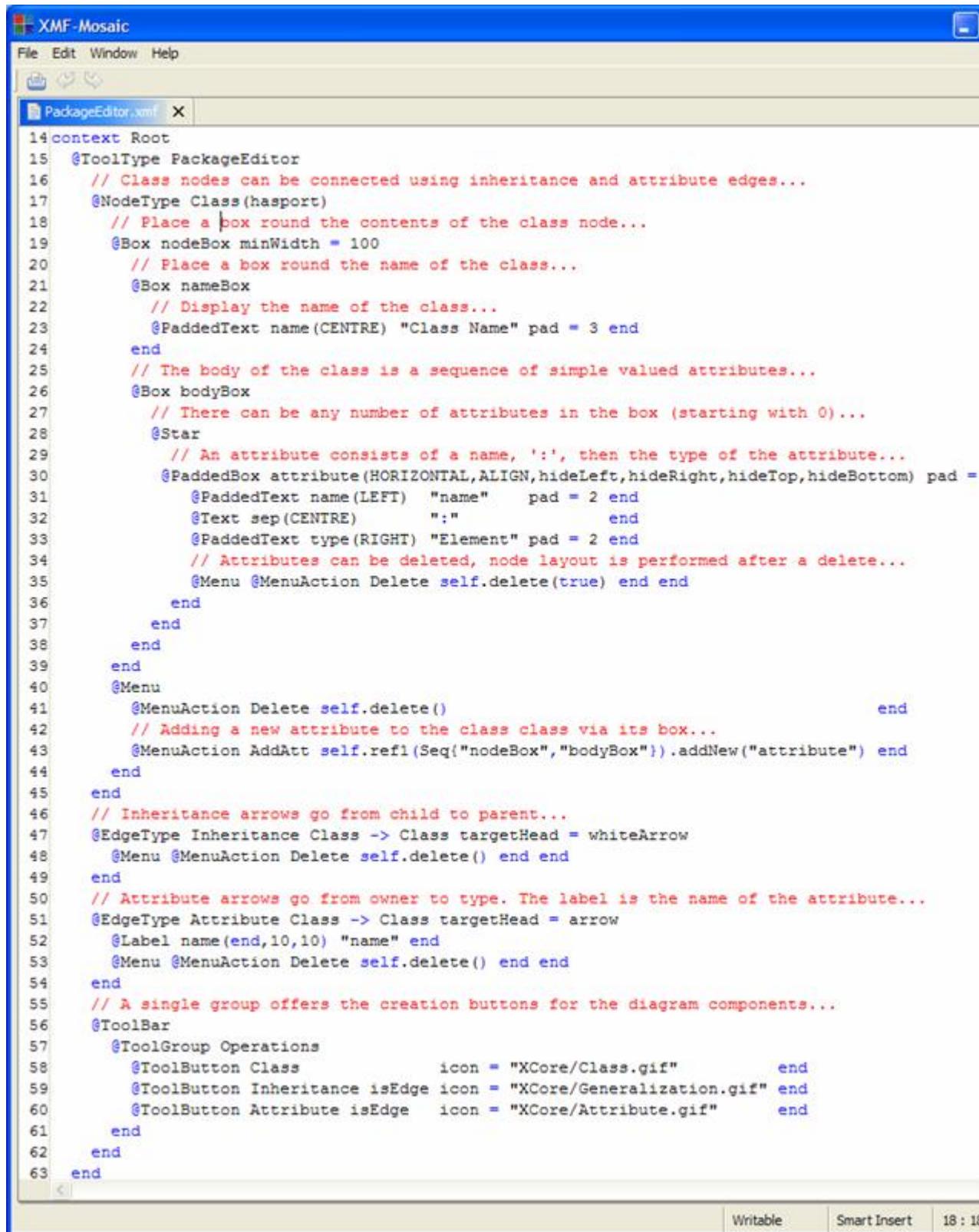


The interface that we wish to expose over the domain model is the ability to create classes and attributes in a single package. The names of the classes and attributes can be changed and the types of the attributes can be modified. We will make a distinction between builtin classes used as attribute types (for example `NamedElement::name : String`) and other classes. Attributes of classes with builtin types will be shown inside the class box whereas all other attribute will be attached to classes via an edge (such as `Attribute::type`).

The domain model is an example of a meta-model in that it describes itself and the model has been constructed using the Xtool for package editing. Another example, (showing a non-meta model) is a standard information model for a Library as follows:



The tool type for a package editor is shown below. It contains an example of the use of Star to define that part of a class node that contains a sequence of attribute definitions. Initially bodyBox will be emptyInstances of the display type attribute may be added to, and deleted from, bodyBox.



The screenshot shows the XMF-Mosaic application window. The title bar says "XMF-Mosaic". The menu bar includes "File", "Edit", "Window", and "Help". Below the menu is a toolbar with icons for file operations. The main area is a code editor with tabs. The active tab is "PackageEditor.xmf". The code is an XML script defining a "Root" context. It includes sections for "ToolType" (with "PackageEditor"), "NodeType" (with "Class" and its attributes), "EdgeType" (with "Inheritance" and "Attribute" edges), and a "ToolBar" section with "Operations" containing "ToolButton"s for Class, Inheritance, and Attribute.

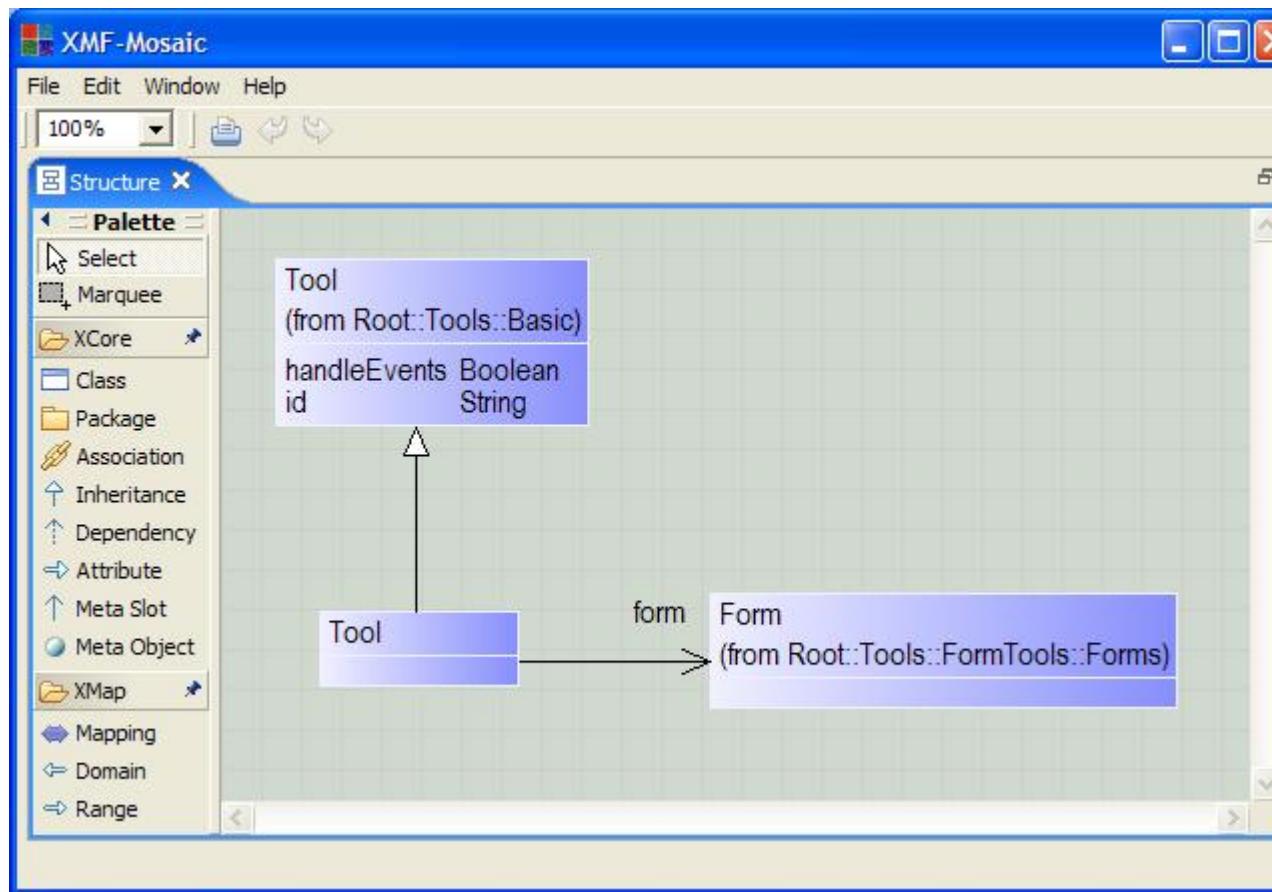
```

14 context Root
15   @ToolType PackageEditor
16     // Class nodes can be connected using inheritance and attribute edges...
17     @NodeType Class(hasport)
18       // Place a box round the contents of the class node...
19       @Box nodeBox minWidth = 100
20         // Place a box round the name of the class...
21         @Box nameBox
22           // Display the name of the class...
23           @PaddedText name(CENTRE) "Class Name" pad = 3 end
24       end
25       // The body of the class is a sequence of simple valued attributes...
26       @Box bodyBox
27         // There can be any number of attributes in the box (starting with 0)...
28         @Star
29           // An attribute consists of a name, ':', then the type of the attribute...
30           @PaddedBox attribute(HORIZONTAL,ALIGN,hideLeft,hideRight,hideTop,hideBottom) pad =
31             @PaddedText name(LEFT) "name" pad = 2 end
32             @Text sep(CENTRE) ":" end
33             @PaddedText type(RIGHT) "Element" pad = 2 end
34             // Attributes can be deleted, node layout is performed after a delete...
35             @Menu @MenuAction Delete self.delete(true) end end
36           end
37         end
38       end
39     end
40     @Menu
41       @MenuAction Delete self.delete() end
42       // Adding a new attribute to the class class via its box...
43       @MenuAction AddAtt self.refl(Seq("nodeBox","bodyBox")).addNew("attribute") end
44     end
45   end
46   // Inheritance arrows go from child to parent...
47   @EdgeType Inheritance Class -> Class targetHead = whiteArrow
48     @Menu @MenuAction Delete self.delete() end end
49   end
50   // Attribute arrows go from owner to type. The label is the name of the attribute...
51   @EdgeType Attribute Class -> Class targetHead = arrow
52     @Label name(end,10,10) "name" end
53     @Menu @MenuAction Delete self.delete() end end
54   end
55   // A single group offers the creation buttons for the diagram components...
56   @ToolBar
57     @ToolGroup Operations
58       @ToolButton Class icon = "XCore/Class.gif" end
59       @ToolButton Inheritance isEdge icon = "XCore/Generalization.gif" end
60       @ToolButton Attribute isEdge icon = "XCore/Attribute.gif" end
61     end
62   end
63 end

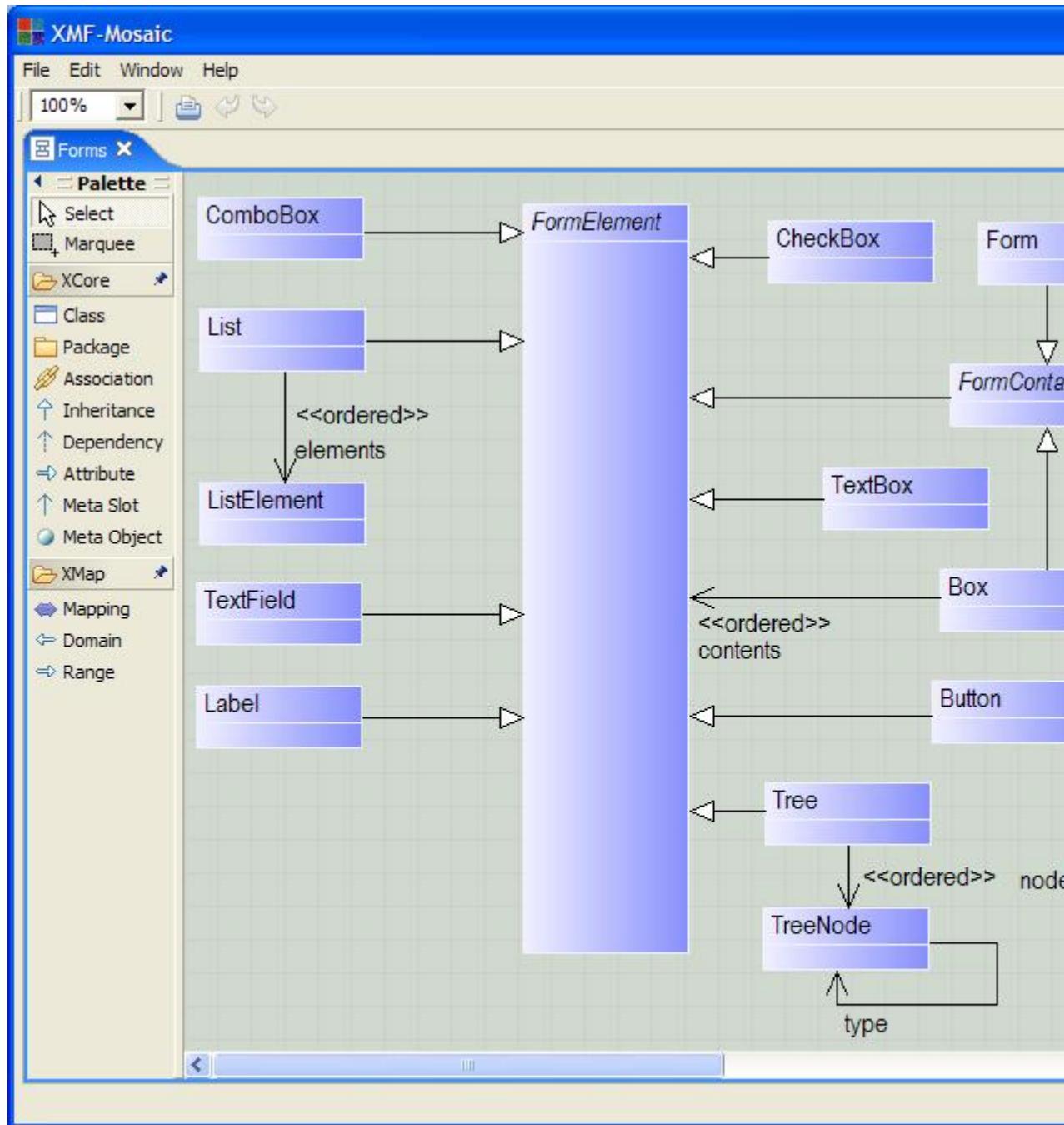
```

Form Tools

Introduction

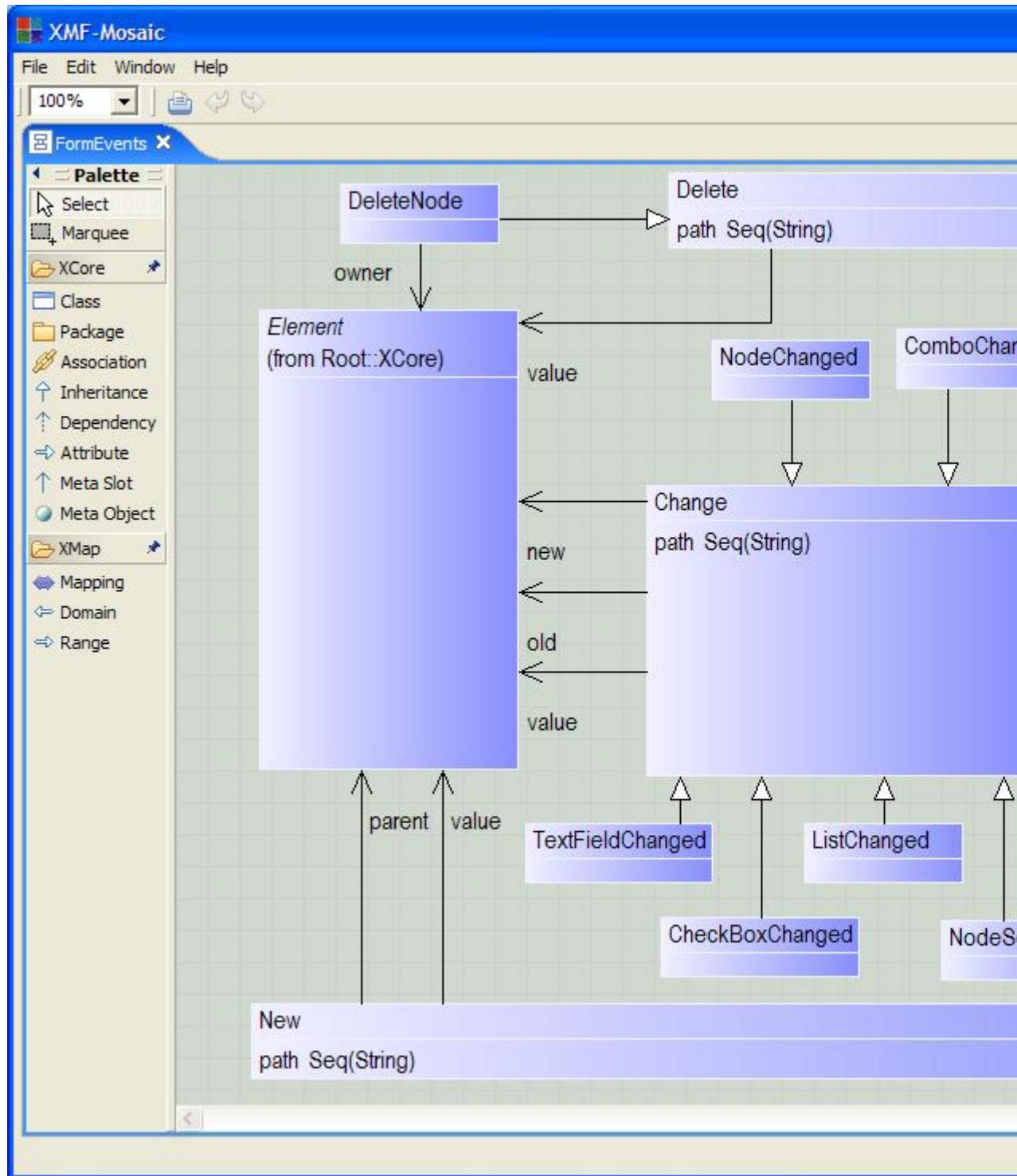


Form Components



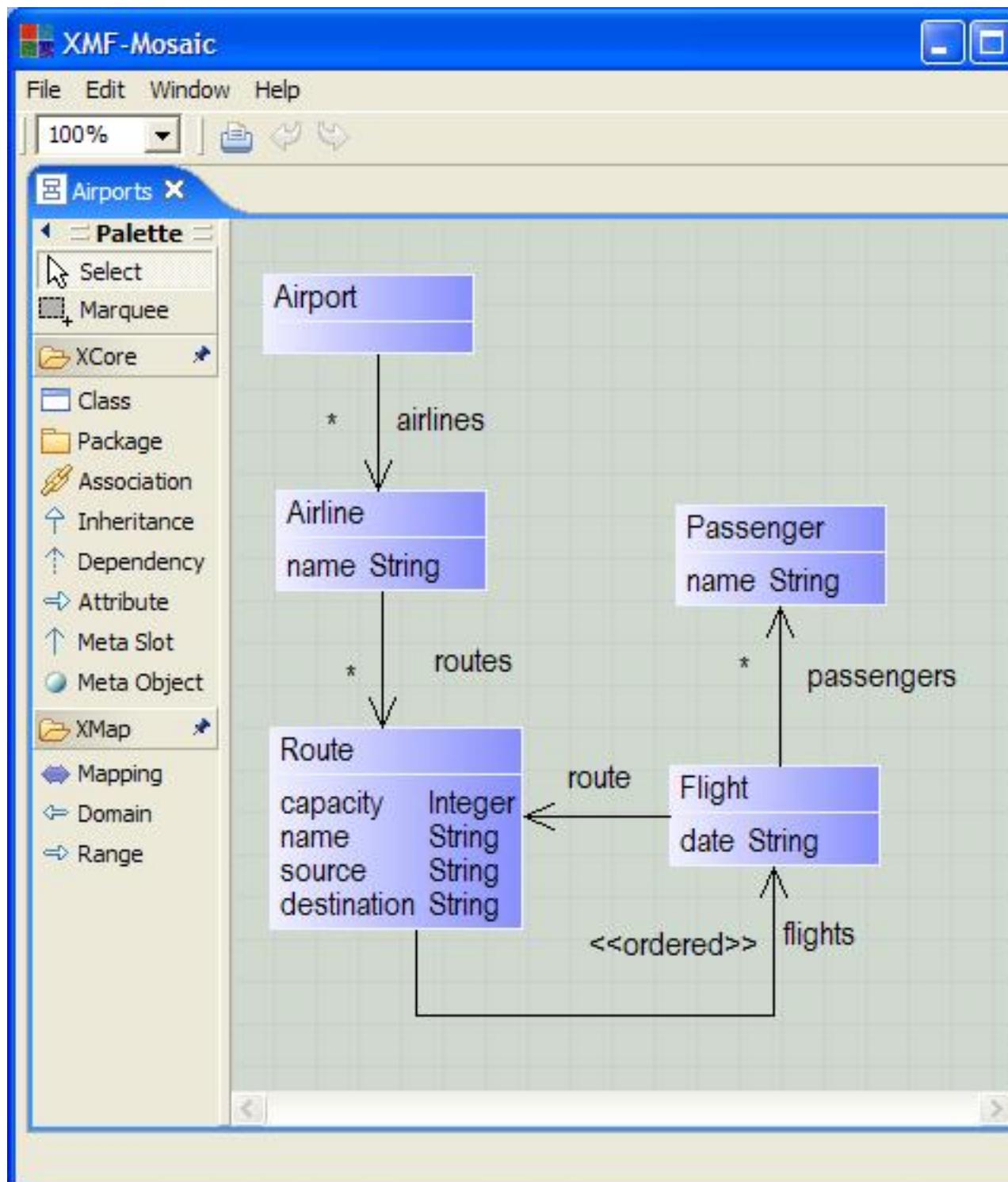
Menus on Forms

Form Events

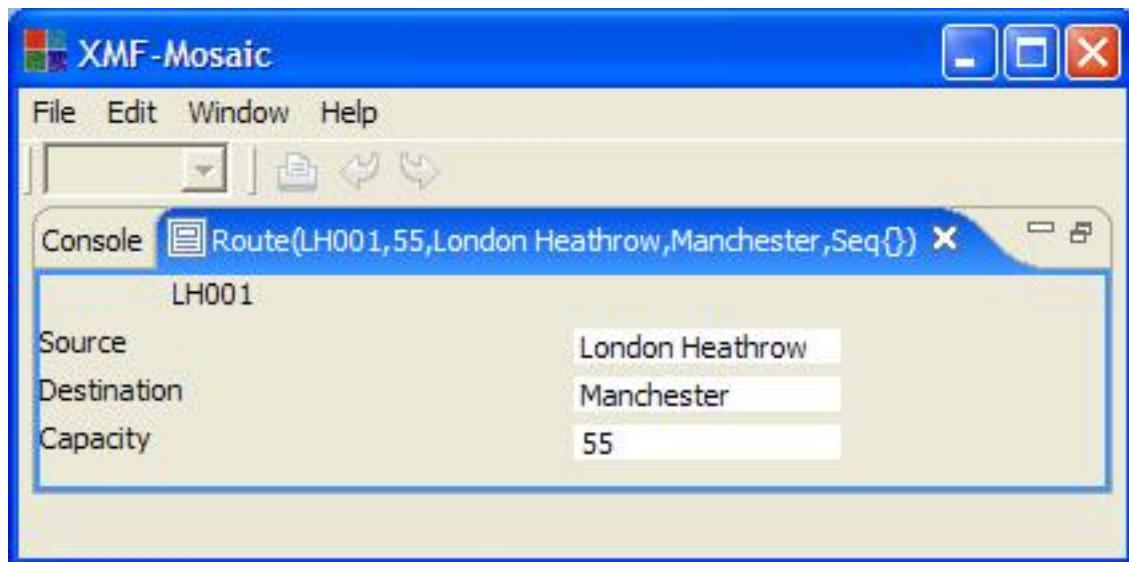


Example: Airports

Domain Model



Property Editor



Browser

Chapter 23. Clients

Introduction

XMF is a standalone kernel that in its vanilla form has no means of interacting with the environment outside of its own world. There are numerous ways of providing this interaction including file based IO and XML based IO. A further way of providing this interaction which is particularly well suited, but not limited, to user interfaces is to write a client which sits between the user interface and the XMF kernel. This is the approach taken to implementing the Mosaic components of XMF-Mosaic and XMF (specifically its underlying operating system XOS) provides a number of mechanisms to support this activity. These mechanisms are discussed and illustrated in this document.

Introduction

XMF supports three different types of clients.

Message : A message based client communicates by passing structured messages to and from the kernel. Currently message based clients must be written in Java as Eclipse plugins so that they are able to utilise external Java libraries provided by XMF-Mosaic.

Internal : An Internal client is written in Java and communicates over stream based data. Unlike Message based clients, internal clients have no inherent messaging architecture although one could be easily built on this foundation. Message based clients must be written in Java and are run in the same process as the XMF kernel.

External : An external client are like internal clients and communicate over stream based data. However external clients connect to the XMF kernel (via XOS) using socket based communication. This means that external clients can be written in any language that supports socket communication and such clients can also be run using a distributed architecture.

Message Based Client

This section describes how to construct both the Java (Eclipse) and XMF implementation for a message based client. This is done by way of a traffic light example. Such a style of tool might be used as a tool to simulate a model using an external GUI. The implementation of this client assumes a certain level of Eclipse knowledge, further information about Eclipse can be found in Eclipse's help system or at <http://www.eclipse.org>.

Eclipse Implementation

Dependencies

A message based client is constructed as an Eclipse plugin. Since it is necessary to use libraries provided by XMF-Mosaic, the new client needs to declare this dependencies. The simplest way of doing this is to import the XMF plugins com.ceteva.client and com.ceteva.xmf into an Eclipse workspace, and then in the newly created plugin declare the dependencies as illustrated below.



```

<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
    id="org.myproject.trafficlights"
    name="Trafficlights Plug-in"
    version="1.0.0"
    provider-name=""
    class="org.myproject.trafficlights.TrafficlightsPlug...

    <runtime>
        <library name="trafficlights.jar">
            <export name="*"/>
        </library>
    </runtime>

    <requires>
        <import plugin="org.eclipse.ui"/>
        <import plugin="org.eclipse.core.runtime"/>
        <import plugin="com.xactium.client"/>
        <import plugin="com.xactium.xmf"/>
    </requires>

    <extension
        point="org.eclipse.ui.startup">
    </extension>

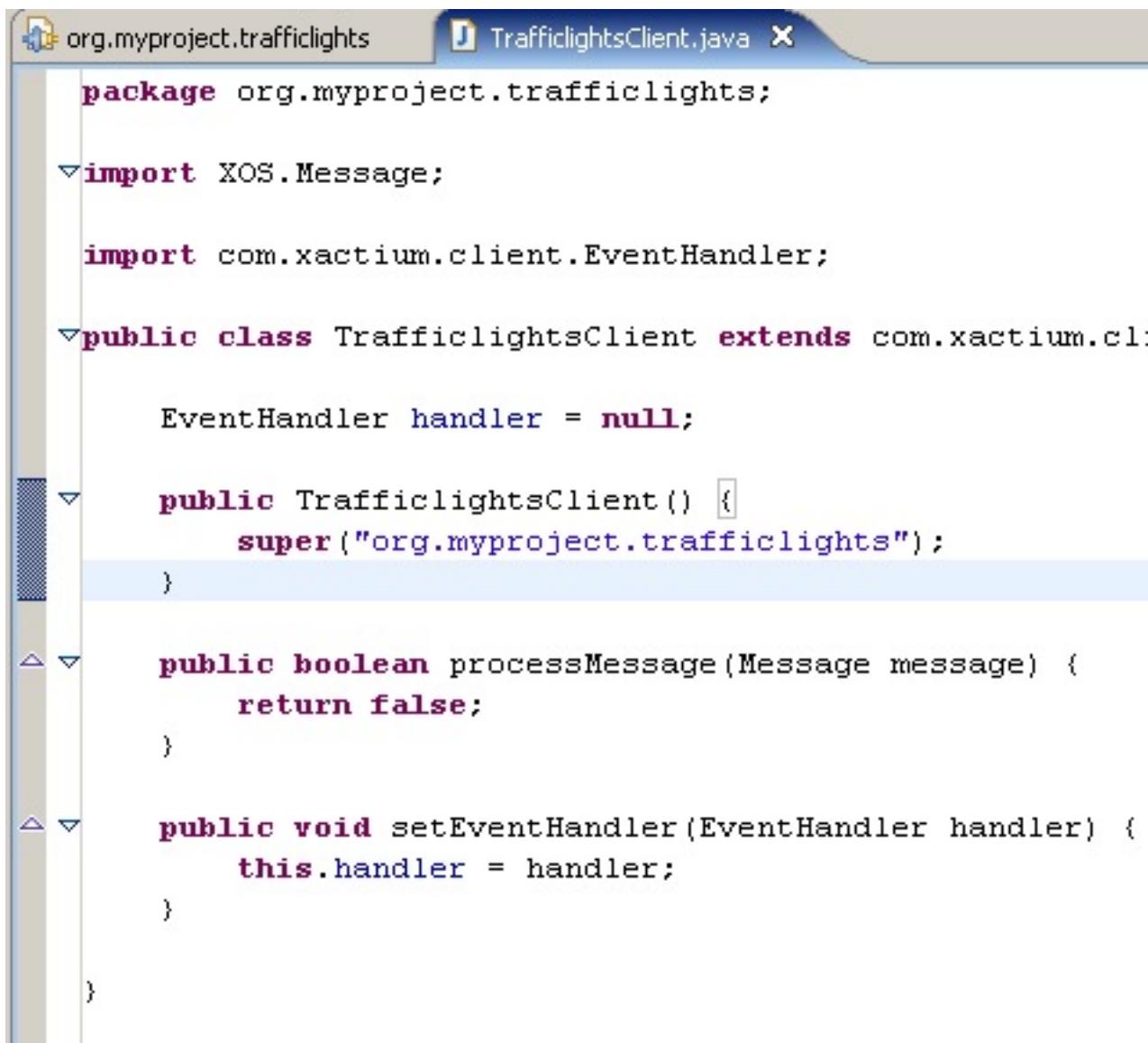
</plugin>

```

Generally speaking, message based clients should connect to XOS prior to the machine being started. Therefore the plugin must connect as soon as it is started. This is achieved by way of the Eclipse early startup interface which must be declared in the as illustrated below. We will show later in this document how this facilitates the early connection of the client.

Basic structure

The figure below illustrates a basic structure of message client is shown below. A message client must extend com.ceteva.client.Client and implement a number of abstract methods. The constructor for the client must pass as an argument the name that this client is going to identify itself as, in this case org.myproject.trafficlights. A method must be implemented to process messages passed to the client, a boolean return value indicates whether or not the passed message was successfully processed. Finally the setEventHandler method is called by XOS to set the handler for events, the general pattern is to record the handler so that it can be usefully referenced in the future. In this case the handler is scoped over the class.



```

package org.myproject.trafficlights;

import XOS.Message;
import com.xactium.client.EventHandler;

public class TrafficlightsClient extends com.xactium.cli...

    EventHandler handler = null;

    public TrafficlightsClient() {
        super("org.myproject.trafficlights");
    }

    public boolean processMessage(Message message) {
        return false;
    }

    public void setEventHandler(EventHandler handler) {
        this.handler = handler;
    }
}

```

Handling Messages

A message consists of a name and a number of arguments. The client we are building can potentially receive three messages: setRed, setAmber and setGreen; each message has a single boolean argument. In the figure below the incoming message is tested against the three possible messages the client can handle. In addition to checking the messages name, the messages arity is also checked. The ability to check a message's arity enables the same message name to be handled in different ways depending on the number of parameters passed.

In the case of this example the state of the message is simply printed to standard output and true is returned to indicate to XOS that the message has been successfully processed. If none of the message handlers match then the false value is returned.

The screenshot shows a Java code editor with the file `TrafficlightsClient.java` open. The code implements a client for traffic lights using the XOS framework.

```
package org.myproject.trafficlights;

import XOS.Message;
import com.xactium.client.EventHandler;

public class TrafficlightsClient extends com.xactium.cl...
```

EventHandler handler = null;

```
public TrafficlightsClient() {
    super("org.myproject.trafficlights");
}

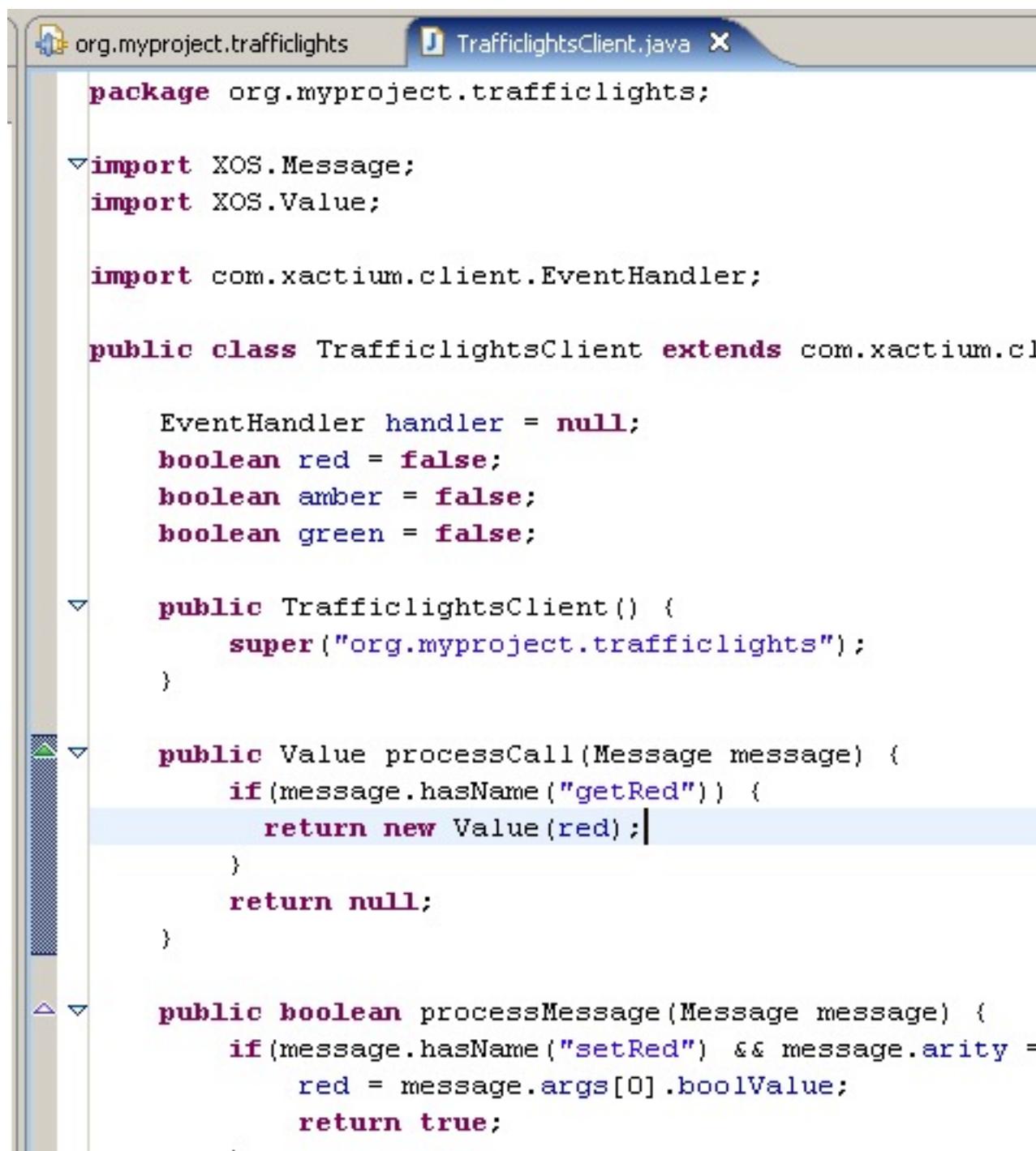
public boolean processMessage(Message message) {
    if(message.hasName("setRed") && message.arity == 1) {
        boolean state = message.args[0].boolValue;
        System.out.println("Red light on: " + state);
        return true;
    }
    else if(message.hasName("setAmber") && message.arity == 1) {
        boolean state = message.args[0].boolValue;
        System.out.println("Amber light on: " + state);
        return true;
    }
    else if(message.hasName("setGreen") && message.arity == 1) {
        boolean state = message.args[0].boolValue;
        System.out.println("Green light on: " + state);
        return true;
    }
    return false;
}

public void setEventHandler(EventHandler handler) {
    this.handler = handler;
}
```

Handling Calls

Sometimes it is necessary for a client to be asked a question to which it returns an answer. For example XMF may ask its traffic light client the current value of the red light. This scenario is provided by the call mechanism which is like a message but rather than the return value indicating successful processing, it is an actual value that XMF requires. When an XMF call is made then it switches to synchronous communication rather than the asynchronous communication of regular message processing.

The figure below illustrates the pattern for implementing call handlers. The method processCall is overridden to provide handlers for the call. Return values must be of type Value and are formed by passing primitive types into the constructor of Value. In this case a null value is returned to indicate failure to handle the call.



The screenshot shows an IDE interface with a Java file named 'TrafficlightsClient.java' open. The code implements a client for a traffic lights service, handling calls for getting the red light status and setting the red light status.

```

package org.myproject.trafficlights;

import XOS.Message;
import XOS.Value;

import com.xactium.client.EventHandler;

public class TrafficlightsClient extends com.xactium.cl

    EventHandler handler = null;
    boolean red = false;
    boolean amber = false;
    boolean green = false;

    public TrafficlightsClient() {
        super("org.myproject.trafficlights");
    }

    public Value processCall(Message message) {
        if (message.hasName("getRed")) {
            return new Value(red);
        }
        return null;
    }

    public boolean processMessage(Message message) {
        if (message.hasName("setRed") && message.arity == 1)
            red = message.args[0].boolValue;
        return true;
    }
}

```

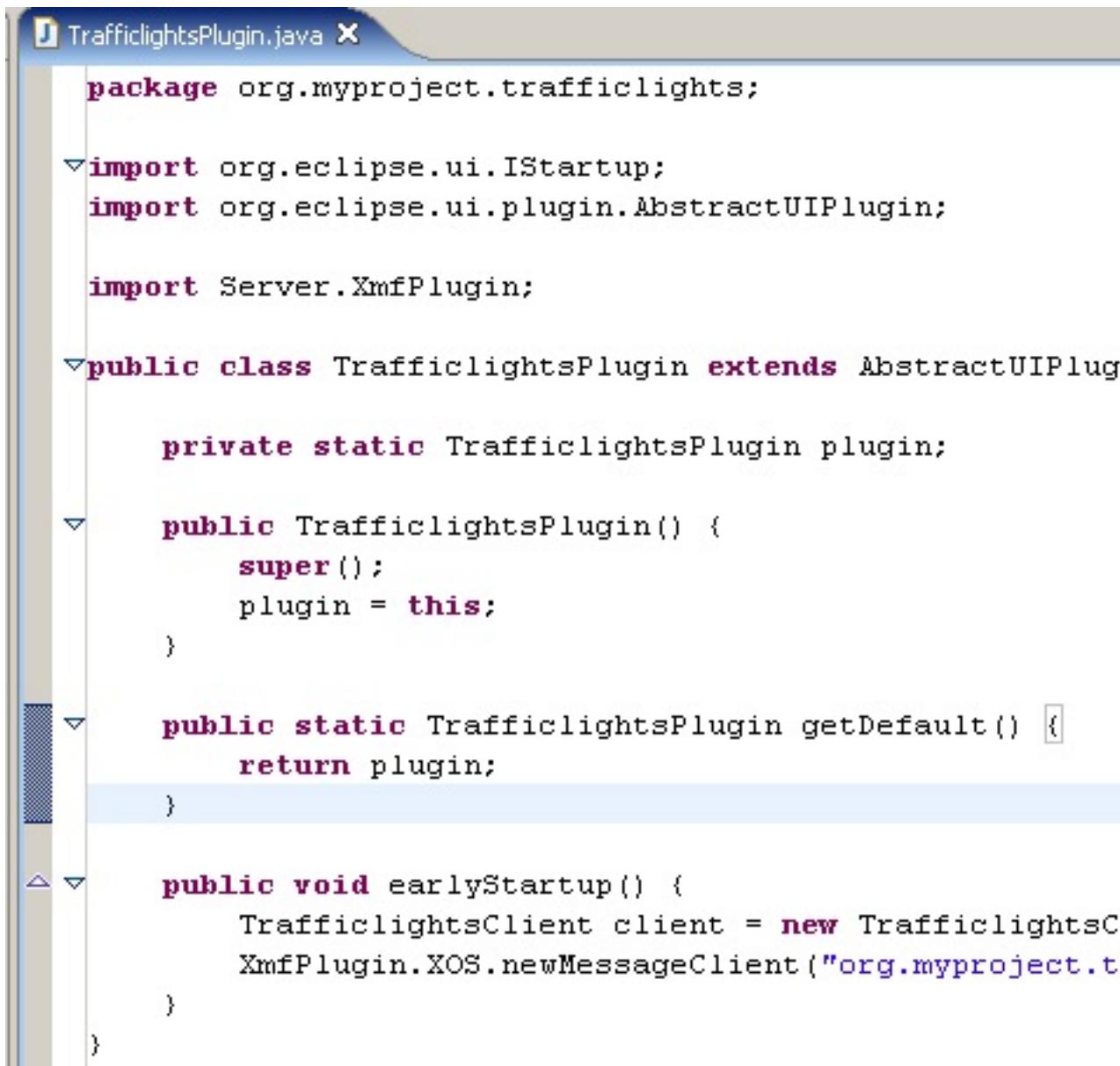
Raising Events

A client can send information to XMF by sending messages on the event handler. In the example below a java method is implemented which sends an event reporting the number of cars in a queue. An event is constructed by asking the handler for a new event when supplied with the event's and arity. The parameter values are then assigned to the newly constructed event, in this case this is a single parameter denoting the length of the queue. Finally the event is raised on the handler.

```
public void reportCarQueue(int length) {  
    Message event = handler.newMessage("reportCarQueue", 1);  
    event.args[0] = new Value(length);  
    handler.raiseEvent(event);  
}
```

Registering the Client

As discussed at the beginning of this section, a message client should generally connect to XMF prior to the machine being started. The pattern for doing this is illustrated below in the context of the traffic light example, the early startup method is called as a result of the plugin's class implementing the IStartup interface (and the declarative extension shown earlier in the plugin.xml file). This method creates a new instance of the client and informs XOS of this client by passing both the client's name and the client itself.



The screenshot shows the Eclipse IDE interface with a Java file named "TrafficlightsPlugin.java" open. The code implements an XMF plugin for the Eclipse UI. It includes imports for IStartup, AbstractUIPlugin, and Server.XmfPlugin, and extends AbstractUIPlugin. It contains a static plugin variable and methods for early startup and getting the default plugin.

```

package org.myproject.trafficlights;

import org.eclipse.ui.IStartup;
import org.eclipse.ui.plugin.AbstractUIPlugin;

import Server.XmfPlugin;

public class TrafficlightsPlugin extends AbstractUIPlug

    private static TrafficlightsPlugin plugin;

    public TrafficlightsPlugin() {
        super();
        plugin = this;
    }

    public static TrafficlightsPlugin getDefault() {
        return plugin;
    }

    public void earlyStartup() {
        TrafficlightsClient client = new TrafficlightsC
            XmfPlugin.XOS.newMessageClient("org.myproject.t
    }
}

```

XMF Implementation

Architecture

In this section we will show how a client can be modelled in XMF that interfaces to the Eclipse client described in the previous section.

The first step is to create a package to put the xmf client.

```

Welcome  TrafficLights.xmf X

parserImport XOCL;

context Root;

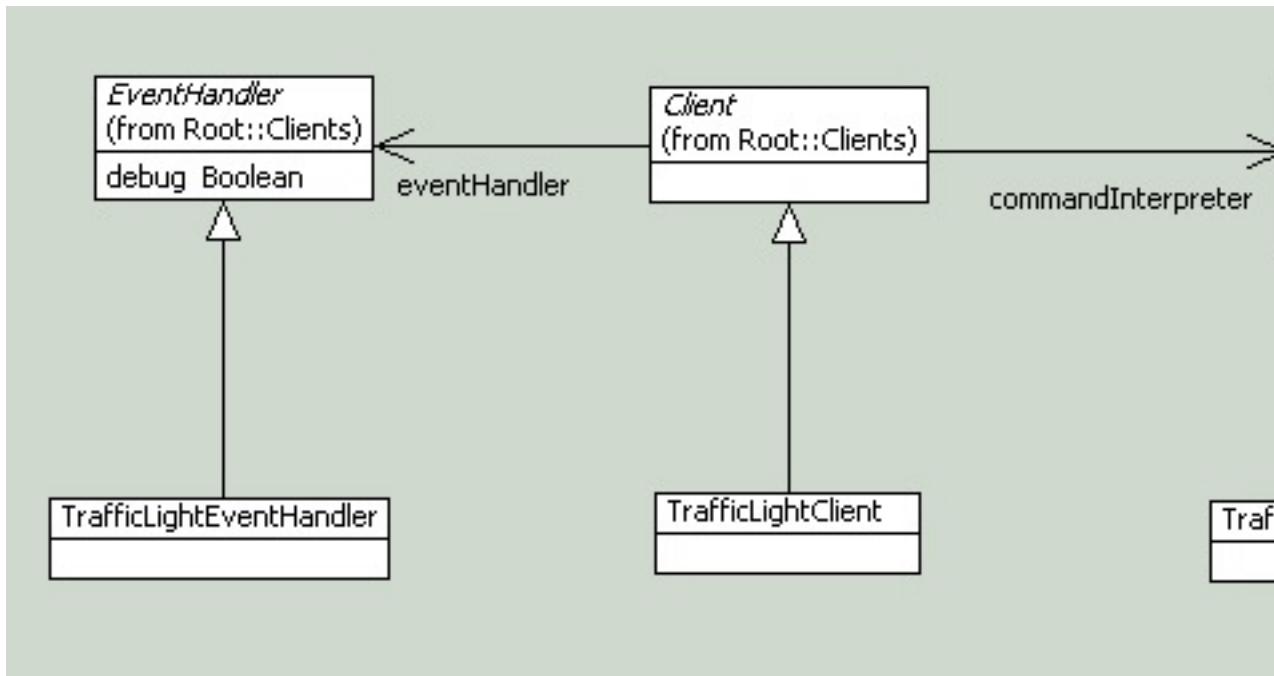
@Package TrafficLights

// This package defines an interface to an external
// client which simulates a traffic light

end

```

The general architecture for an XMF message based client is shown below. An event handler is defined which specialises the abstract class `Root::Clients::EventHandler`, this describes how events are received by the XMF client. Similarly a command interpreter is defined which specialises the abstract class `Root::Clients::CommandInterpreter`, this describes how both messages (commands) and calls are sent to the Java client and how. A client is then defined which specialises `Root::Clients::Clients`, this acts as a wrapper to both the event handler and the command interpreter. The next sections will examine each of these components in detail in the context of the traffic light example.



Sending Messages and Making Calls

Messages are sent, and calls are made, in the command interpreter. Generally speaking each message is wrapped in an operation and the body of the operation contains a send command of the form illustrated in the following example:

```

@Operation setRed(state:Boolean)
    @SendCommand(self)
        setRed(state)
    end

```

```
end
```

The body of the send command contains the command named and the parameter values to bind to the command enclosed in brackets (and separated by commas where there are more than one). In the above example, when the setRed/1 operation is called then the setRed(..) message is sent to the client. The figure below shows the command interpreter implementing each of the different message types supported by the Java client.



A screenshot of a code editor window titled "TrafficLightCommandInterpreter.xmf". The code is written in XOCL (eXtensible Object Command Language). It defines a class "TrafficLightCommandInterpreter" that extends "CommandInterpreter". The class contains four operations: "setRed", "setAmber", "setGreen", and "getRed". Each operation is annotated with "@SendCommand(self)" and has a corresponding implementation method ("setRed(state)", "setAmber(state)", "setGreen(state)", "getRed()"). The code also imports "Clients" and defines a context "TrafficLights". The "parserImport" statement at the top imports "XOCL".

```
parserImport XOCL;

import Clients;

context TrafficLights

@Class TrafficLightCommandInterpreter extends CommandInterpreter

    @Operation setRed(state:Boolean)
        @SendCommand(self)
            setRed(state)
        end
    end

    @Operation setAmber(state:Boolean)
        @SendCommand(self)
            setAmber(state)
        end
    end

    @Operation setGreen(state:Boolean)
        @SendCommand(self)
            setGreen(state)
        end
    end

    @Operation getRed()
        @CallCommand(self)
            getRed()
        end
    end

end
```

The above figure also gives an example of a call command which is exactly the same for as a send command but is wrapped as a call command. It is important to note that when a send command is made, the message is sent and code execution continues regardless of whether the message was successfully processed. By contrast when a call command is made, code execution halts and waits for the return value of the call command prior to continuing execution. The getRed() operation will return the value of the result of the getRed() call command (a boolean value).

Handling Events

Events are processed in an event handler, the standard pattern for writing these is shown below. When the event handler receives an event, it passes the event and its parameters to the operation `dispatchEvent`. This operation should be overridden to perform client specific event handling. In the figure below the incoming event is tested to see whether it has the name `reportCarQueue`, if so it is passed to the `readCarQueue` method and simply printed to standard output.



```

parserImport XOCL;

import Clients;

context TrafficLights

@Class TrafficLightEventHandler extends EventHandler

@Operation dispatchEvent(message,parameters)
    @Case message.name of
        "reportCarQueue" do self.readCarQueue(parameters)
    end
end

@Operation readCarQueue(parameters)
    format(stdout,"Number of cars queuing:~S~%",parameters)
end

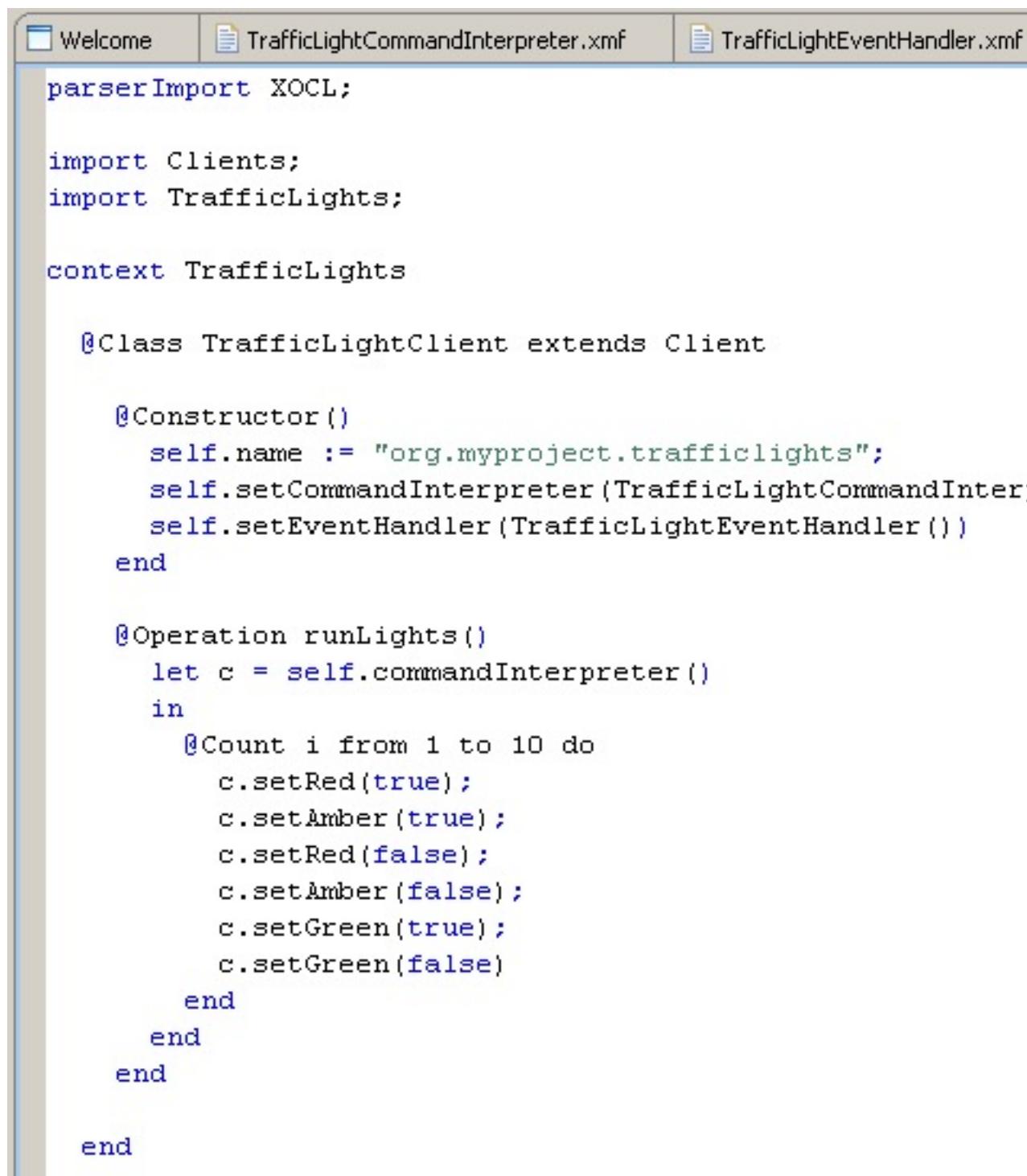
end

```

As we we will demonstrate in the next section, the event handler is run in its own thread. This allows the event handler to process events regardless of what is happening in the main thread of execution.

Putting it Together

The figure below shows the standard approach to defining an XMF client using the command interpreter and event handler. The constructor for the client sets the clients name, this must match the name defined in the Java client which in this case is `org.myproject.trafficLights`. As shown below, the constructor then creates an instance of both the command interpreter and the event handler (note that if communication is uni-directional either the command interpreter or event handler may be omitted as appropriate).



The screenshot shows a code editor window with three tabs at the top: 'Welcome', 'TrafficLightCommandInterpreter.xmf', and 'TrafficLightEventHandler.xmf'. The 'TrafficLightCommandInterpreter.xmf' tab is selected. The code in the editor is XCL (XOS Configuration Language) and defines a client named 'TrafficLightClient' that interacts with 'TrafficLights'.

```

parserImport XOCL;

import Clients;
import TrafficLights;

context TrafficLights

@Class TrafficLightClient extends Client

@Constructor()
    self.name := "org.myproject.trafficlights";
    self.setCommandInterpreter(TrafficLightCommandInter-
    self.setEventHandler(TrafficLightEventHandler())
end

@Operation runLights()
    let c = self.commandInterpreter()
    in
        @Count i from 1 to 10 do
            c.setRed(true);
            c.setAmber(true);
            c.setRed(false);
            c.setAmber(false);
            c.setGreen(true);
            c.setGreen(false)
        end
    end
end

end

```

In addition to constructing the client, the above example defines a further operation `runLights/0` which simply tests that the command interpreter works by sending a number of messages.

Starting the Communication

In order to start the communication between XMF and the Java client it is necessary to take two steps. The first step involves informing XOS about the new Java client by adding details to its `startup.txt` file which can be found in the directory `XMF_INSTALL/xmfMosaic/plugins/com.ceteva.xmf_x.x.x/Server` (replacing the `x.x.x` with the current version number of XMF, if there are multiple versions then consider the largest number to be the most recent). In this file you will see details of the existing Mosaic clients, simply append the new client to this list as illustrated below:

```
-message :com.ceteva.mosaic:wait  
...  
-message :org.myproject.trafficlights:wait  
...
```

The final step is to create an instance of the XMF client and add it to XMF's client manager as shown below. If the client has an event handler as in the traffic light example, then it is necessary to start the event handler in its own independent thread so that it can handle events independently of the main thread of execution. This can be done using the pattern demonstrated below where a call to Client::start/0 is wrapped in a fork. The code below also runs the example operation defined in the previous section.



A screenshot of a code editor window titled "RunLights.xmf". The code in the editor is:

```
parserImport XOCL;  
  
import Comms;  
import IO;  
  
let  
    client = TrafficLights::TrafficLightClient()  
in  
    xmfc.clientManager().add(client);  
    @Fork(TrafficLightEvents)  
        client.start()  
    end;  
    client.runLights()  
end;
```

Internal Clients

External (Socket Based) Clients