

Anti-Yacc: MOF-to-text

David Hearnden, Kerry Raymond, Jim Steel
{hearnden, kerry, steel}@dstc.edu.au
CRC for Enterprise Distributed Systems (DSTC)
University of Queensland, Australia

Abstract

The Object Management Group (OMG) is pursuing its new Model-Driven Architecture (MDA) strategy. The Meta-Object Facility (MOF) is an important technology to support the MDA, both as a general modelling technique but also specifically to support the definition of other modelling systems. The OMG has standardised the generation of repositories based on MOF models. Anti-Yacc is a tool that can be used to extract the contents of MOF-based repository in textual form. The Anti-Yacc tool takes as input the specification of grammar rules, lexical rules, and MOF-extraction rules, from which a Java program is generated to extract the contents of a MOF-based repository on demand. Anti-Yacc can be used for code generation, interfacing with legacy syntaxes, and general report writing.

1. Introduction

In September 2001, the Object Management Group (OMG) formally adopted a new approach to distributed enterprise architecture: the Model-Driven Architecture (MDA) [1]. The main thrust of MDA is that systems development should start with high-level specifications written independent of platform technologies and then be transformed or refined progressively into deployable technologies. The MDA proposes a framework of models, both Platform-Independent Models (PIMs) and Platform-Specific Models (PSMs), with tools to automate (wherever possible) the translation between models. The benefit to enterprise distributed systems of the MDA approach are that the use of high-level specifications and automated transformation will:

- allow for the more succinct expression of systems and hence the more precise capture of business requirements and processes
- enable an application to be rapidly ported to different platforms with guaranteed interworking (both at the protocol and conceptual levels)
- facilitate change being rapidly and consistently propagated throughout deployed applications, including changes in the choice of implementation technology

The success of the MDA will ultimately depend on the availability of a range of appropriate models (both PIMs and PSMs) as well as the availability of development tools to support the modelling and transformation needs.

1.1. Standards for the MDA

OMG already has a number of mature modelling standards: the Meta-Object Facility (MOF) [2] and the Unified Modelling Language (UML) [3], to be used as bases for modelling. In addition, for MOF, the OMG has standardised the generation of repositories and other component tools for the storage, transfer, and manipulation of MOF-based information. MOF and UML are being used extensively within the OMG for the definition of models, both for domain-specific models as well as generic PIMs and PSMs.

However, there has been less activity to date within the OMG and its member companies on the standards and tools for transforming models and integrating them into the development environment. Existing standards within the OMG include:

- the generation of CORBA-based repositories for instances of a given MOF model (informally known as the MOF-to-CORBA mapping)[2]
- the generation of Java-based repositories for instances of a given MOF model, known as Java Metadata Interface (JMI) [4] — (standardised by the Java Community Process in collaboration with OMG)
- the XML-based interchange of the contents of MOF-based repositories (XMI) [5] [6]
- the automatic generation of a language and parser to populate a MOF-based repository (Human Usable Textual Notation RFP) [7][8] (expected to be adopted in June 2002)

A number of other MDA-relevant standards activities [9] are expected to commence in 2002 in the OMG, including:

- MOF 2.0 Versioning and Life Cycle Management RFP;
- MOF 2.0 Query/Views/Transformations RFP;
- MOF 2.0 Federation/Facility/Directory RFP.

However, one area that is not being currently addressed by OMG is the integration from MDA technology into existing text-based tools and development environments.

1.2. Motivation for Anti-Yacc

The motivation for the Anti-Yacc tool is to provide a means to render the content of a MOF-based repository (known as a MOFlet) in a textual form conforming to some specified syntax. This will often be required in the MDA to convert a specification expressed in terms of a model into text-based forms used by the tools that support that model. For example, a platform-specific model for the Java Messaging System must be rendered as a Java program that conforms to the syntax expected by the Java compiler. Another example might be the rendering of a WWW model into a set of HTML pages, which again must conform to the syntax understood by browsers.

As the principles of MDA become more widely employed, we expect to see a great increase in automated generation of code, configuration and documentation. Many existing tools in these areas use text input, and we see the existence of a flexible MOF-to-text tool like Anti-Yacc as an essential ingredient in the MDA toolkit.

Although not a primary goal, another benefit of a MOF-to-text tool like Anti-Yacc is its availability to be a general report writer for a MOF-based repository.

A number of tools that generate text from specific MOF-based repositories already exist; we have first-hand experience in developing such tools by hand. As a consequence, it is our firm belief that the task can be made considerably easier (quicker and less error-prone) by using a MOF-to-text tool like Anti-Yacc.

1.3. Structure of this paper

Section 2 describes background technologies and some related work, while Section 3 gives an overview of Anti-Yacc. Section 4 introduces the MOF model for Anti-Yacc, with more details of the Anti-Yacc “backend” (which extracts information from the MOFlet) given in Section 5. Section 6 has examples of using Anti-Yacc, and introduces the concrete syntax through these examples. Section 7 discusses the rationale behind some of the design decisions of Anti-Yacc and explores possible future developments, while the overall conclusions are presented in Section 8.

2. Background

2.1. The Meta-Object Facility (MOF)

The MOF specifies a small but complete set of modelling concepts that can be used to describe information models. The MOF standard also provides a mapping from these modelling concepts to CORBA IDL (Interface Definition Language), that is then extended to allow for the generation of a repository for the data modelled using the MOF, as shown in Figure 1. Such generated repositories are known as MOFlets; specifically, the repository generated from the X-model is known as the X-MOFlet.

The main MOF modelling concepts that will be discussed in this paper are: *Package*, for containment of classes and associations; *Class*, which contains attributes and participates in associations; *Association*, which represents a set of links between instances of two specified classes, and which can have composition properties; *Attribute*, either in the form of one of a range of data types or an instance of a class; and *Reference*,

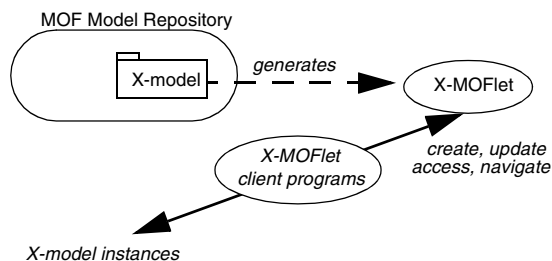


Figure 1. MOF models and MOFlets

which is a class's view on an association in which it participates. For more detail on these and other MOF modelling concepts, consult the specification [2].

2.2. XML Metadata Interchange (XMI)

The OMG has developed the XML Metadata Interchange (XMI) Format standard [5][6]. The XMI standard defines a set of mappings from the MOF modelling concepts to a representation in XML (eXtensible Markup Language), a standard of the World Wide Web Consortium (W3C) [10].

The XMI specification has two main components: a set of rules for producing an XML DTD or XML Schema from a model, and a set of rules for the transfer of data between XMI and a MOF-compliant repository or tool. These rules are embodied in producer and consumer programs generated to support the model-specific XMI format, as shown in Figure 2.

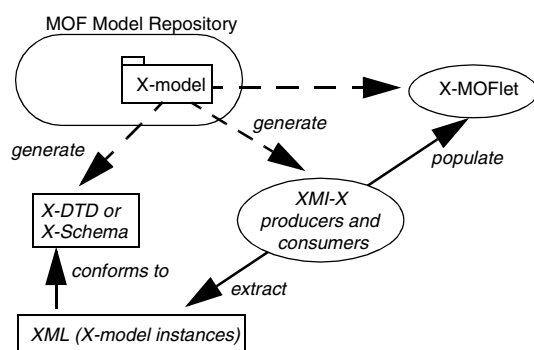


Figure 2. The operation of XMI tools

In XMI, each object (class instance) in the MOFlet package is rendered as a unique identifier (needed for internal or external cross-reference) together with the values of its attributes and references (to other objects with which it is associated). If the object contains other objects (via a MOF containment association), then the

contained object is rendered as part of the contents of the container object in XMI. Where possible, association links are rendered as reference values in the source or target object, but may be rendered as a separate table of links using object cross-references.

2.3. Human-Usable Textual Notation (HUTN)

In 1999 the OMG initiated a standardisation effort for a Human Usable Textual Notation for the Enterprise Distributed Object Computing (EDOC) standard [11] in order to provide a human-friendly textual input language [7]; XML was explicitly excluded as being insufficiently human-friendly on a large scale. However, the proposal currently being considered for adoption [8] has taken a more generic approach and instead defines a means of creating a human-friendly language for a nominated MOF model. The HUTN tool also generates producers and consumers (parsers) for this language, enabling a MOFlet of that model to be rendered textually or populated, as shown in Figure 3. The generated parser backend populates a MOFlet for that model.

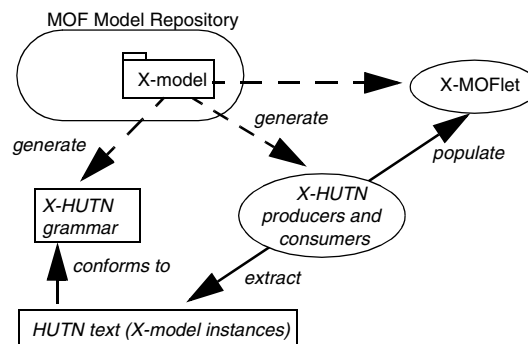


Figure 3. The operation of HUTN tools

The primary design goal of HUTN is human usability, and this is achieved through consideration of the successes and failures of common programming languages. The languages generated for each model are all different, but are very similar in structure as they are generated from patterns parameterised with the specifics of each model. In some areas, there is scope for customisation based on user preference (e.g. the selection of default values to reduce the volume of text).

As HUTN languages and their model-specific tools can be fully automated, HUTN is extremely useful for both rapid systems development and prototyping, as any change to the model can automatically regenerate the model-specific HUTN language and tools.

2.4. How is Anti-Yacc different?

Like XMI and HUTN, Anti-Yacc renders the content of a MOFlet in textual form, conforming to some syntactic rules (grammar). However, XMI and HUTN are designed to work with any MOF model, as they generate their target grammar based on predefined patterns. Although HUTN supports some customisation of the generated language, it is still restricted to producing languages that conform to predefined patterns. Similarly XMI cannot produce XML conforming to arbitrary DTDs or Schemas, only to those conforming to predefined patterns.

In contrast, Anti-Yacc is capable of producing text that conforms to an arbitrary user-supplied EBNF target. This can include XML conforming to non-XMI DTDs/Schemas. The price of this flexibility is that Anti-Yacc's applicability is limited to a user's nominated MOF model. Obviously not every MOF model can support a given target grammar, as the values required by that grammar may not reflect information held in that model.

Also, XMI and HUTN are designed to output all of the information in a package (or containment tree), whereas Anti-Yacc is capable of selecting, combining, or converting information. Finally, Anti-Yacc is only concerned with output from a MOFlet; XMI and HUTN support input and output.

3. Overview of Anti-Yacc

In order to understand Anti-Yacc, it is useful to compare it with Yacc. As the name suggests, Anti-Yacc was inspired and strongly influenced by Yacc.

Yacc [12] is a well-known parser generator used in UNIX systems. A parser is a tool for converting text into an instance of an abstract model. Parser generators typically work with a grammar that specifies the syntactic structure of the input text, and actions to perform upon recognition of those syntactic elements. Thus a parser generator (such as Yacc) takes as input a

mapping from the concrete syntax to the abstract syntax of a model (the mapping typically provided as code fragments).

Anti-Yacc (as the name suggests) reverses this process, by taking as input a mapping from the semantics of the model to the syntactic structure of some textual language. Typically, the syntactic structure of a grammar used to parse and construct a model can be the basis for an Anti-Yacc grammar, which is then decorated with various code fragments to drive the production of text.

Just as Yacc requires lexical rules to reduce the input to a stream of tokens, conversely Anti-Yacc requires lexical rules to convert the output stream of tokens into presentable text. Lexical rules in Anti-Yacc are typically concerned with presentation issues such as horizontal or vertical white space.

Although Anti-Yacc is a reversal of processes of Yacc, Anti-Yacc is not intended to be capable of reversing specific Yacc processing. It may be possible in some cases to do so, but generally the original parsing process will have resulted in some information loss (e.g. comments in a programming language) that cannot be magically recreated. Also Yacc can be unconstrained as to the nature of any backend database or system, whereas Anti-Yacc was specifically designed to render the contents of a MOFlet.

3.1. Operation of Anti-Yacc

Anti-Yacc operates in the following manner. Anti-Yacc is given a grammar (as a text file) and a meta-model (in the MOF repository), and uses the two to generate a "walker" program (written in Java). The walker program can then be compiled and executed against a target MOFlet, generating text output, as shown in Figure 4.

In the sense that Anti-Yacc is analogous to Yacc, a generated X-walker is analogous to a generated parser.

4. The Anti-Yacc Model

Anti-Yacc defines a MOF model for the Anti-Yacc generation rules, enabling the generation rules to be stored in a MOFlet.

The generation rules are composed of three distinct parts:

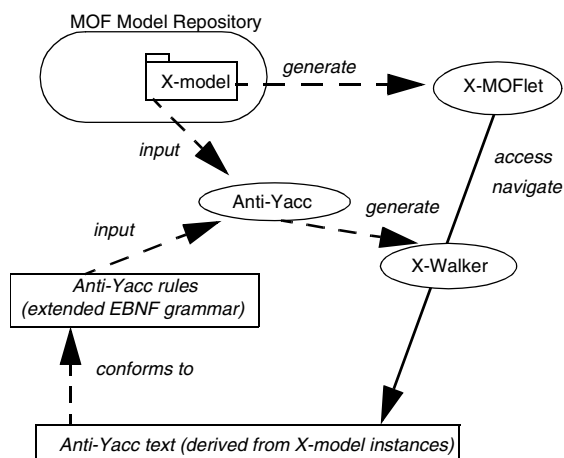


Figure 4. The operation of Anti-Yacc tools

- syntax rules, that define the grammar of the text to be generated;
- lexical rules, to control output presentation and style;
- MOF-extraction rules, to navigate and extract information from the source MOFlet to provide both the values to appear in the generated text and the decision-making to choose the appropriate syntax rules.

4.1. Common model

Each component of the generation rules is based on a common model of EBNF [13] shown in Figure 5 (with details of attributes elided for space reasons).

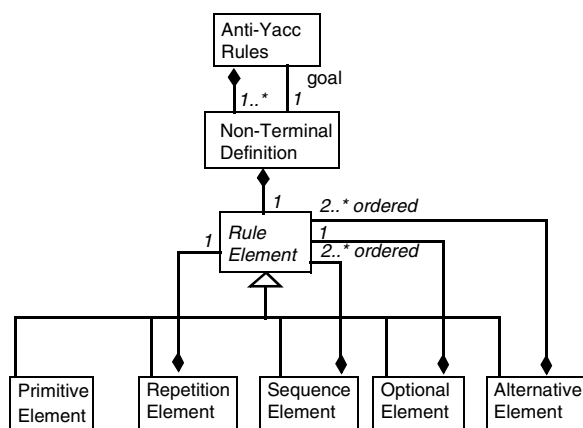


Figure 5. Common EBNF model

The Anti-Yacc rules consist of the definition of rules for each of the non-terminal symbols in the grammar. One of the non-terminals has the special status of being the goal of the grammar. Each of the rules for a non-terminal symbol can take one of 5 forms, and these forms can be recursively used. The five forms are:

- the primitive element, which represents atomic information, usually string/numeric and other literal values, e.g.:

“begin”

Identifier

- the sequence element represents a list of rule elements, e.g.

X Y Z

- the repetition element represents the repeating of a single specified rule element, e.g.:

{X}*

{Y}+

- the optional element represents the presence or absence of a single specified rule element, e.g.:

X?

- the alternative element represents a choice between a list of rule elements, e.g.:

X | Y | Z

Each of the component models further specialises each of the classes corresponding to the syntactic, lexical or value-determining needs of that component.

4.2. The syntactic model

As the common model captures most aspects of an EBNF grammar, the syntactic model is a small extension to refine the definition of Primitive Element to distinguish between the kinds of primitive element, as shown in Figure 6. The primitive element may be any of.:

- constant values, e.g. reserved words;
- generated values (to be supplied by the MOF-extraction model), typically these are the counterpart of the Identifier, Number, or String tokens in a Yacc grammar;
- a use of another non-terminal element, invoking a nested execution of the Anti-Yacc rules.

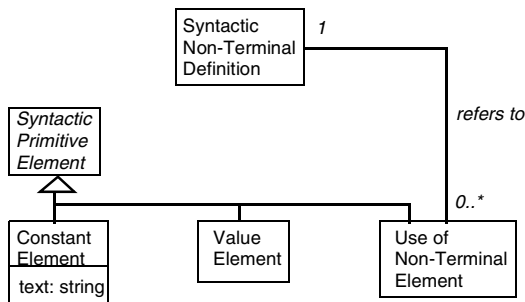


Figure 6. Part of the Syntax Model

The syntactic model is analogous to the grammar in Yacc.

4.3. The lexical model

The lexical model introduces lexical elements which control the presentation of the generated text, as shown in Figure 7.

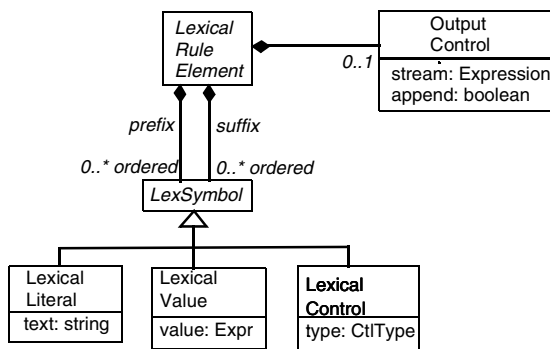


Figure 7. Part of the Lexical model

Each lexical rule element can be preceded or followed with a list of lexical elements. These elements may be strings (lexical literals), computed values (lexical values), or lexical control (such as spaces, tabs, and newlines). Indentation is controlled with two lexical controls, one to increase indentation and another to decrease indentation. Output controls may be associated with a lexical rule element to redirect the output stream, usually to another file. Output controls apply for the duration of that rule element. If a sub-rule has its own output control, then the previous one is stacked and resumed later. An example use of output controls is to create separate files for each Java class.

The lexical model can be considered to be analogous to the use of Lex [14], the lexical analysis most commonly used in conjunction with Yacc.

4.4. The MOF-extraction model

The MOF-extraction model introduces the expression model to navigate and query the source MOFlet, the parameterisation needed to provide a context for the evaluation of expressions, and the decision constructs needed to select the appropriate syntactic form. The MOF-extraction model can be considered as analogous to the backend code in Yacc.

Figure 8 shows the model which associates primary elements with expressions to be evaluated on the source MOFlet. Typically most expressions associated with Extraction Value require some starting point for their evaluation (object reference into the source MOFlet), and this is usually relative to the starting point of the containing Rule Element. For example, if rendering a Java class, one will wish to render its name and other attributes. To evaluate the name, there needs to be some reference to the modelling construct that represents the Java class.

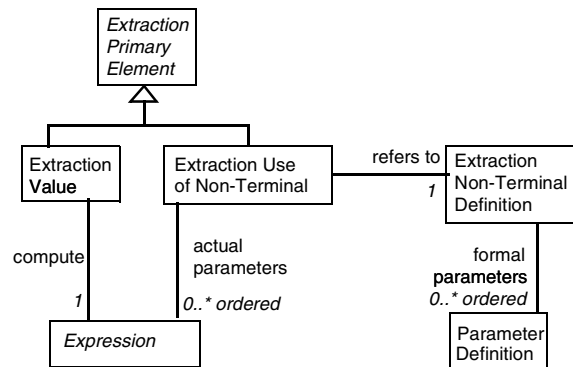


Figure 8. Expressions in the Extraction Model

Therefore, there needs to be a system of parameter-passing from the container rule element to the contained rule element. This is achieved by associating formal parameters of the Non-Terminal Definition with the actual parameters (expressions) with the use of those non-terminals. Each non-terminal can be thought of as a procedure in a conventional programming language. These formal parameters can then be used in the expressions relating to primary elements, substituted at

run-time with the actual parameter values. This is equivalent to the naming of values \$\$, \$1, \$2 etc in Yacc.

The Expression model will not be presented in detail but consists of the usual unary and binary operators applicable to the common data types. An expression can use any of the following terms:

- parameters (as defined for that non-terminal);
- values within the source MOFlet (including object references);
- constants and enumeration values defined in the source MOF model;
- operations defined in the source MOF model whose implementation is provided by the source MOFlet;
- built-in variables and operations provided by the Anti-Yacc implementation.

The type system for both parameters and expressions is that of the source MOF model as well as certain built-in types of Anti-Yacc. The Anti-Yacc tool can do type checking to ensure that types and terms exist within the source MOF model, that their type (including their cardinalities) are consistent with their usage, and that any operations called are defined as queries (without side-effects).

One specific sub-type of expression is Native Expression, which enables the user to provide native code (Java code in the case of our tool) for situations requiring greater expressive power than can be captured directly using the Expression model, e.g. calling externally-defined methods. Correspondingly, there is support for native types. In our current tool, the use of native expressions and native types reduces the type safety of the overall expression.

The most obvious use of Expressions is to render their value in the output text. The less obvious use is to evaluate boolean expressions as guards to optional and alternative rules as shown in Figure 9.

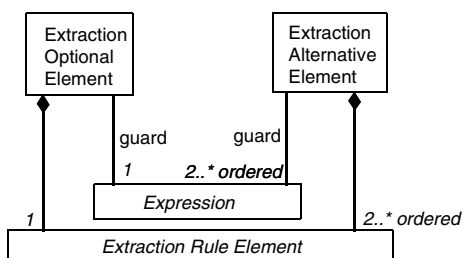


Figure 9. Guards in the Extraction Model

Each optional element has a guard that must evaluate to true if its contained extraction rule is to be processed. Similarly, each set of alternatives has a corresponding set of guards which determine which of the alternate extraction rules is to be executed. In Anti-Yacc, the extraction rules and their guards are ordered, and the guards are evaluated in that order, effectively creating a cascading sequence of if-then-else tests. It is a run-time error if all of the guards for alternate extraction rules evaluate to false.

The final aspect of the extraction model to be discussed is repetition as shown in Figure 10.

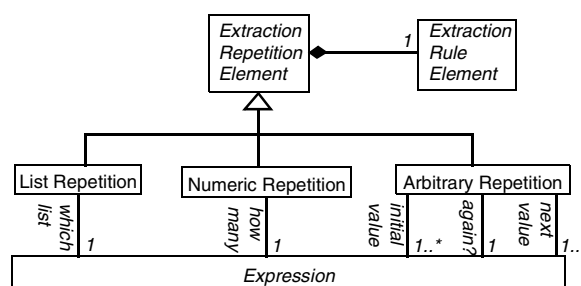


Figure 10. Repetition in the Extraction

There are three kinds of repetition in Anti-Yacc:

- List repetition
- Numeric repetition
- Arbitrary repetition

List repetition (perhaps more correctly described as collection repetition) is for iterating over some collection of values. It is the most common kind of repetition used in practice, as the MOF supports a number of collection types including sets, bags, and lists, and these can be used in representing attribute values or references (links to other associated objects). Examples of list repetition are to iterate over all of the attributes of a class or all of the children of a parent node. There is one execution of the extraction rule element for each element of the collection resulting from the evaluation of the “which list” expression. List repetition can also specify that the list is to be processed in sorted order, by supplying an expression to be evaluated on each list element to determine its position.

Numeric repetition executes the extraction rule element a fixed number of times based on the integer resulting from the evaluation of the “how many” expression.

Arbitrary repetition is the most general kind of repetition, and the least used in practice. It resembles the “for” statement in Java or C:

```
for (X=initial_value; again?;X=next_value)
    extraction_rule(X)
```

It commences by executing the extraction rule over the value returned by the “initial value” expression, and then over the values returned by each successive execution of the “next value” expression. The “again?” expression is evaluated prior to the execution of the extraction rule and the iteration is terminated when it evaluates to false. To support the most general framework of repetition, there can be multiple initial values and their corresponding next values. There is an iterator for each initial/next value pair. Although it may appear to be a rather complex construct, it is often needed in practice. An example is an extraction rule that has to combine information from successive elements of two linked lists; the initial values would point to the two list heads and the next values would each navigate to the successive element in their respective lists.

4.5. Integrating the models

The syntactic, lexical and extraction models are linked together as shown in Figure 11.

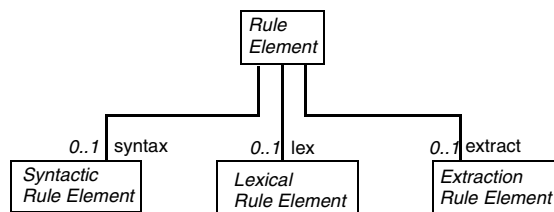


Figure 11. Integrating the models

5. Anti-Yacc Details

5.1. In-built variables and functions

To support repetition effectively, there are some built-in variables available for use in expressions used in connection with repetition:

- **\$@** the current iteration element;
- **\$#** the current iteration count (starts from 0);
- **\$\$** the current iteration list (for list repetition only).

Anti-Yacc defines three inbuilt functions for working with collections:

- **size()** which returns the number of elements in the collection, a long;
- **isEmpty()** which is equivalent to **size() == 0**;
- **elementAt(long)** which returns the element at the given index (starts from 0).

5.2. Type checking

Type checking is not as straightforward as it might appear as there are three closely related type systems in use:

- the types introduced by a MOF model
- the equivalent CORBA IDL types plus some introduced ones (created during the generation of the MOFlet)
- the equivalent programming language types plus some introduced ones (generated by the applicable CORBA language mapping) for native types.

For example, in the MOF model, an attribute may have type Bag of the data type string. The MOFlet will have the corresponding CORBA IDL type of sequence <string>, while the MOFlet client will treat the attribute as an array of the Java object String.

Where pure expressions (i.e. not native expressions) are used, then types can be mostly restricted to those of the MOF model. However, a MOF model has no type that corresponds to the object reference that is the entry point into a MOFlet, and this object reference is the actual parameter that must be passed at the commencement of execution of the rules for the goal non-terminal.

So, for complete type checking, there are places when correspondence between the type systems must be taken into account. In our current tool, the use of native expressions are not type safe due to the additional complexity required to derive the type of the result of these expressions and then relate it back into the type systems of CORBA and the MOF model.

6. Examples

For these examples, consider the (rather simple) model of a university in Figure 12.

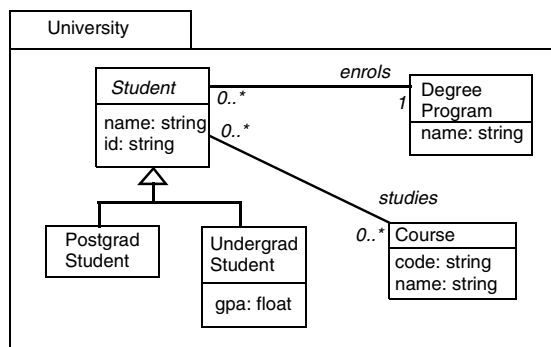


Figure 12. Model of a university

Using the concrete syntax developed for Anti-Yacc, the rules for rendering a student's enrolment details would be a single sequence of primitive elements:

```
student (Student s):
  <s.id> 'enrolled' <s.enrols.name> ;
@student: # sp # sp # ;
```

from which the following text could be produced:

123456 enrolled B.A.

The first rule combines the syntactic and MOF-extraction rules in a single syntax based on the traditional EBNF syntax with MOF-extraction expressions enclosed in < and >. Note that the non-terminal **student** has the formal parameter **s** of type **Student** (from the model). The second rule prefixed with @ is concerned with purely lexical considerations of the first rule, specifically putting a space (denoted by **sp**) between each element (denoted by #). In both rule types, a semicolon terminates the rule.

A more interesting example is to produce a list of the courses in which a student is enrolled:

```
undergrad(UndergraduateStudent s):
  <s.name> 'studying' ':'
  $(s.studies) <$@.code>*
@undergrad: # sp # # nl ind (# nl)* und nl ;
```

This illustrates list repetition, where the list of courses **s.studies** is identified by the **\$(...)** syntax and the repeated processing (denoted by *****) to be performed on each list element is to print the course code of that element (denoted by **\$@**). The corresponding lexical rule must also reflect the repetition **(..)*** and instructs that a newline (denoted by **nl**) is to be added (as a suffix)

to each course code; the **ind** and **und** denotes increasing and decreasing the level of indentation (**und** = unindent). An example of the output would be:

Mary Smith studying:
CS100
EE123
MA115

If the rendering of course information was more complex, then it might be better to have two pairs of rules, one for the undergraduates and one for the courses:

```
undergrad(UndergraduateStudent s):
  <s.name> 'studying' ':'
  $(s.studies) <course($@)>*
@undergrad: # sp # # nl ind (# nl)* und nl ;
course(Course c): <c.code> ':' <c.name> ;
@course: # # sp #
```

which would produce the following output:

Mary Smith studying:
CS100: Introduction to Programming
EE123: Electrical Engineering Principles
MA115: Advanced Calculus

Note that the more complex rendering of each course is still regarded as a single element # in the lexical rule for **undergrad**.

An example of using alternative elements would be to rate a student based on their grade point average (**gpa** in the model):

```
undergrad(UndergraduateStudent s):
  <s.name> 'awarded'
  ([s.gpa >= 6.0] 'honours'
  |[s.gpa >= 4.0] 'pass'
  |[s.gpa < 4.0] 'fail'
  );
@undergrad: # sp # sp (# | # | # '!') ;
```

The guards for each of the alternative rule elements are enclosed in [...] and the alternatives separated by |. As in EBNF, parentheses (..) are used to group elements. Again, notice that the lexical rule has the same structure, and that it has been decided to emphasise a failing student by adding (lexically) an exclamation mark ! (see Section 7.4 for a discussion on the wisdom of doing so). Example output would be:

John Smith awarded fail!

Any element of the syntax can be made optional by placing a guard before it:

```
undergrad(UndergraduateStudent s):  
<s.name> [s.gpa >= 6.0] '(first class)';
```

```
@undergrad: # [ sp # '!' ];
```

which produces either of (depending on the grade point average):

```
Betty Blue
```

```
Betty Blue (first class)!
```

7. Discussion and Future Work

Anti-Yacc has been prototyped using DSTC's dMOF product [15] and has been successfully used in code generation experiments, including the rendering of MODL (the MOF model language used by dMOF), Java, and Anti-Yacc's own concrete syntax. However, as developers and users, there are some debatable design decisions and a number of areas for enhancement and future work.

7.1. Concrete syntax

In our current Anti-Yacc tool, the syntactic rules and the MOF-extraction rules are interspersed (in the style of Yacc) while the lexical rules are presented separately (although usually immediately following). The drawback to this approach is that the lexical rule has to be carefully written to match the structure of the syntactic/extraction rules; the Anti-Yacc tool naturally checks that they are structurally similar. Also, it is easy to get confused by the use of the undistinguished # mark to denote the various elements defined in the syntactic/extraction rules. Or to put it simply, it is easy to get the lexical rule wrong!

However, our previous experience of interspersing the lexical elements with the syntactic/extraction elements created very long rules, which users found it difficult to read and write due to the clutter of white space controls. It may be necessary to support both styles, so that full interspersing can be used for the simpler rules and separate lexical rules can be used for the more complex structures. Lexical rules could be made easier if the # token could carry some optional description, e.g.:

```
@undergrad: #name sp #enrolled #colon nl  
ind (#coursecode nl)* und nl ;
```

Although more verbose, it would be definitely more readable.

The current concrete syntax also uses symbols such as @, \$, # extensively reflecting the strong Yacc influence and a current user community fluent in C, Perl and other Unix technologies, where compact representation is preferred to the use of reserved words. However, a wider user community will probably require the introduction of additional alternative syntaxes that are less compact but more self-explanatory. Also, it would be valuable to migrate the concrete syntax to be more similar to other expression languages used in the MDA community, e.g. the Object Constraint Language (OCL) which is currently the subject of an OMG standardisation effort [16].

7.2. Built-in functions

There is a need for more built-in variables and functions to support specific elements in the model, including:

- optional attributes
- narrowing from a supertype to a subtype
- iterating over all elements of a type

As MOF supports optional attributes (cardinality 0..1), it is very common to render these using an optional EBNF element. The guard for such optional element is almost invariably a test to see if the attribute is present. Unfortunately CORBA IDL does not have support for optional attributes and so testing to see if an optional attribute is present in a MOFlet involves attempting to get its value, which throws a NotSet exception if the attribute is not present. The expression model in Anti-Yacc does not currently support exception-catching and so native expressions must be used. Even if the expression model did support exceptions, it would still result in some fairly cumbersome rules. A better solution would be for Anti-Yacc to provide built-in functions of the form **isSetFoo()** for each of the optional attributes in the model to greatly simplify these guards on optional attributes.

Similarly, the CORBA-to-Java language mapping for testing if a supertype is an instance of a subtype and "narrowing" the object reference must also be expressed using native code. In the university model in Figure 12, the rules for processing a student (an abstract supertype) must appear as:

```

student(Student s):
  [{s._is_a(UndergradStudentHelper.id())}
  undergrad({UndergraduateStudentHelper.nar
  row(s)})]
  [{s._is_a(PostgradStudentHelper.id())}
  postgrad({UndergraduateStudentHelper.narr
  ow(s)})]

```

where {...} denotes native expressions. Clearly having some built-in functions (generated from the model) to test subtypes and some automated narrowing would enable the rules to be far more readable, e.g.:

```

student(Student s):
  [isUndergraduateStudent(s)]
  undergrad(s)
  [isPostgraduateStudent(s)]
  postgrad(s)

```

Or as this is quite a common pattern, there could be more direct syntactic support, e.g.:

```

student(Student s) ->
  undergrad(Undergraduate s)
  | postgrad(Postgraduate s)

```

since Anti-Yacc has access to the model and knows about supertype-subtype relationships.

Another common pattern is to perform some rule over all instances of a particular class. While the generated MOFlet has attributes that provide these lists of instances, there is no way to express this in terms of the source model (one of the incompatibilities between the concurrent type systems discussed in Section 5.2), requiring the use of native expressions:

```

university ({_UniversityPackage up}):
  $({up.student_ref().all_of_type_student()})
  <student($@)>

```

The generation of built-in variables to refer to the MOFlet implementation of the overall university model and the collection of all instances of a class would simplify this pattern, e.g.:

```

university(University u):
  $(u.AllStudent) <student($@)>

```

In summary, the generation of built-in variables and functions based on the source model would eliminate many common needs for native expressions, which would improve both the readability and type-safety of Anti-Yacc rules.

7.3. Alternative rule elements

Currently the guards of alternative rule elements are regarded as a cascading sequence of tests. An alternative would be to evaluate all guards and randomly select one that evaluates true, a more non-deterministic approach. The advantage of the non-deterministic approach is that it is more declarative, each guard precisely identifies the pre-condition for selecting that rule element. In the cascading deterministic approach, the precondition for a rule element is composed of its guard and the negation of the preceding guards, which is far less obvious. However, many programmers are more comfortable with determinism, especially when the decision criteria is complex, and also the implied negation of preceding guards can make the expression of the guards more compact than in the non-deterministic approach.

7.4. The Anti-Yacc model

As can be seen in the models in Section 4 and the examples of Section 6, there is some scope for the Anti-Yacc user to produce the same output by rendering the same information either lexically, syntactically or by extraction. This is analogous with Lex and Yacc, where the boundary between lexical analysis, syntactic analysis and backend processing is also somewhat fluid and a matter of style for the user. If Anti-Yacc had been based on a single monolithic model instead of separate models, then these distinctions would have been blurred, and this may make it simpler for the user.

However, our use of separate models helps to clarify the intended expressive power, whereas blurring the boundaries hides issues that need to be exposed. In particular, having a single model often suffers the risk of having the model (the abstract syntax) being driven by the concrete syntax, as it is tempting to be subverted by what is easy to denote rather than what needs to be expressed.

Also, our use of separate models is intended to better support future developments involving the plug-n-play of different sets of lexical, syntactic and extraction rules.

Our current implementation creates an Anti-Yacc MOFlet for storing Anti-Yacc rules. Given that the purpose of the rules are to create a single walker application, it may seem unnecessary to maintain a repository of these Anti-Yacc rules.

However, it is in the spirit of the Model-Driven Architecture to use models and to capture the development lifecycle in repositories for traceability and re-use. MOF models (packages) can be imported, inherited, nested, clustered and cross-linked, and we intend to experiment with re-use of textual renderings when working with multiple related models, further leveraging our plug-n-play experiments.

8. Conclusions

For the Model-Driven Architecture to become a reality, there is a urgent need for tools to support the development of systems. MDA is based on a series of transformations of platform-independent models (PIMs) into platform-specific models (PSMs). While platform-specific technologies of the future might be revised to extract their input from PSM repositories, many current tools (e.g. language compilers) still require their input in textual form, hence the need to provide a quick-and-easy way to rapidly render text from a model-based repository

We have presented the motivation, model and concrete syntax of Anti-Yacc, and discussed a number of decision decisions and future directions for the work.

So, in summary, Anti-Yacc:

- meets an urgent development need;
- occupies a precise niche in an MDA tool suite;
- is built itself on MDA principles.

Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Industry, Science and Resources).

References

- [1] OMG, "Model Driven Architecture - A Technical Perspective", OMG Document ormsrc/2001-07-01, 2001.
- [2] OMG, "Meta-Object Facility (MOF) Version 1.3", OMG document formal/2000-04-03, 2000.
- [3] OMG, "Unified Modelling Language (UML) Version 1.4", OMG document formal/2001-09-67, 2001.
- [4] Java Community Process, "The Java Metadata Interface (JMI) Specification", JSR 40, <http://jcp.org/jsr/detail/40.jsp>
- [5] OMG, "XML Metadata Interchange (XMI) Version 1.2", OMG TC document formal/2002-01-02, 2002
- [6] OMG, "XMI Production of XML Schema", OMG document ptc/2001-12-03, 2001.
- [7] OMG, "A Human-Usable Textual Notation for the UML Profile for EDOC: Request for Proposal", OMG Document ad/99-03-12, 1999.
- [8] DSTC et al., A Human-Usable Textual Notation for the UML Profile for EDOC.Revised submission, OMG Document ad/2002-03-02.
- [9] OMG, "MOF 2.0 Core Request for Proposal", OMG document, ad/2001-11-14, 2001.
- [10] World Wide Web Consortium, "eXtensible Markup Language (XML) 1.0", W3C Recommendation 10-February-1998, <http://www.w3.org/TR/1998/REC-xml-19980210>.
- [11] OMG, "UML Profile for Enterprise Distributed Object Computing", OMG document ptc/2002-02-05,2002.
- [12] Stephen C. Johnson, "YACC — yet another compiler compiler", CSTR 32, Bell Laboratories, 1974.
- [13] Niklaus Wirth, "What can we do about the Unnecessary Diversity of Notation for Syntactic Definitions?", Comm. ACM, 20:11, pp 822-823, November 1977.
- [14] M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator", CSTR 39, Bell Laboratories, 1975.
- [15] DSTC, "dMOF - DSTC's Meta-Object Facility (MOF) Product", <http://www.dstc.edu.au/Products/CORBA/MOF>.
- [16] OMG, "UML 2.0 OCL Request for Proposal", OMG document ad/2000-09-03, 2000.