

Generating Human-Usable Textual Notations for Information Models

Jim Steel, Kerry Raymond
CRC for Enterprise Distributed Systems (DSTC)
{steel, kerry}@dstc.edu.au

Abstract

Existing techniques for the transfer of information in and out of model-based repositories, and in particular the XMI format, are designed for expedient machine processing, and have significant drawbacks for human users. This report describes a system that automatically generates a producer and consumer for a human-usable textual notation corresponding to a given information model. This HUTN system is based on the Meta-Object Facility, an OMG standard for the definition of information models and the subsequent mapping of these models to CORBA interfaces.

The primary design goal of the system is human usability, and this is achieved through consideration of the successes and failures of common programming languages. The system uses an abstract base syntax that is applied to all models, and allows for user alteration of the language through the provision of several language customisations.

1. Introduction

The concept of model-driven development, whereby the production of enterprise software can be partially or fully automated, is gaining considerable momentum in groups such as the Object Management Group (OMG). Information models developed using languages such as UML and MOF can now be used as bases for the generation of repositories and other component tools for the storage, transfer, and manipulation of conformant information. However, the technologies used in the generation of these components are still in their infancy, and a number of essential components are still to be investigated and developed. One of these components addresses the problem of providing a medium for the

inspection, editing, and transport of information from generated repositories. Some standards [XMI98] have been and are being developed, but these are generally focused on machine-based transfer, and have often resulted in syntaxes that are difficult for humans to use. Another, more common, scenario for the transfer of this data incorporates the use of a custom-made medium or language, sometimes textual but more often graphical.

It would be useful to be able to quickly produce a textual language (as implemented in a producer and a consumer) for the transfer of information between a user and the repository. The need for a textual format stems from a variety of sources. While a graphical notation is a powerful and often intuitive mechanism for the display of the information, it is sometimes not available to all users. Also, a graphical representation can be unwieldy and confusing for large sets of data, where graph-style notations particularly can become very cluttered. Other users desire the ability to use text-based tools such as 'grep' and 'sed' to search for or replace information within the document. Obviously, careful consideration must also be placed into the design of the language's usability.

This paper describes a mechanism by which a language can be generated to fully describe the data held by a repository, while still being intuitive to a human user. The system is designed with a user-centric design philosophy, weighing usability issues as the primary consideration. This usability is enhanced through the system's configuration feature that allows the user to make small adjustments to the syntax of the language. The system uses an XMI data stream in combination with an XSL style sheet to produce readable information representing the contents of a repository. The user can then peruse or modify this data, before returning it to a repository using a generated parser. The Distributed Systems Technology Centre (DSTC)'s dMOF product is used as the repository

system, and is explained in Chapter 2.

This project was partially inspired by the Enterprise Distributed Object Computing (EDOC) series of Requests For Proposal, produced by the Object Management Group (OMG). The requests address the need for a common modelling technique for business object models, and the subsequent need for popular tools supporting it, including compliance with CORBA and UML. The third request in this suite [Hutn99] calls for a “Human-Usable Textual Notation” for expressing these business models. This request suggests that the resultant notation might be “based on a generic language approach that might be readily applied to other profiles”. For the purposes of this project, this suggestion has been extended to constitute the idea of a more generic language generator for MOF-based information models.

The work reported here is intended to form the basis for a DSTC submission in response to this RFP. The prototype described in chapter 5 has been used to generate a parser and producer for a draft of the EDOC meta-model, and the resultant tools are being used internally to DSTC for prototyping purposes.

The content of this paper is structured as follows. Section 2 gives a background to MOF and XMI. Section 3 describes the usability and design decisions considered for the generated languages. Section 4 provides the details of the generated languages. Section 5 describes the prototype that has been developed, and section 6 summarises and draws conclusions from the work.

2. Background

2.1 The Meta-Object Facility (MOF)

The repository system used for this project is the Meta-Object Facility [MOF00], a standard of the Object Management Group (OMG). The MOF specifies a small but complete set of modelling concepts that can be used to express information models. The MOF standard also provides a mapping from these modelling concepts to CORBA IDL (Interface Definition Language, which is then extended to allow for the generation of a repository for the data modelled using the MOF.

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, 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 [MOF00].

The MOF is an attractive candidate for a repository upon which to base a textual notation generator for a number of reasons. Firstly, it is currently without a convenient method for transporting data in and out of repositories. An XMI system (as described below) is available, but this system also has shortcomings with respect to human usability. Also, the MOF standard provides connectivity through CORBA interfaces, making communication with external programs simpler and cleaner than systems without such interfaces. Thirdly, the MOF modelling schema is simple yet powerful, with fewer concepts than many other systems, which makes a generic generation facility significantly more simple to implement than for a system with a complex set of modelling elements.

2.2 XML Metadata Interchange (XMI) Format

While the MOF provides a mapping from models to CORBA-based repositories, it does not address the issue of transporting the metadata between tools implementing the IDL interfaces. To meet the obvious need for such a mechanism, the OMG has adopted the XML Metadata Interchange (XMI) Format standard [XMI00]. 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) [XML98].

The XMI specification has two main components: a set of rules for producing an XML DTD from a model, and a set of rules for the transfer of data between XMI and a MOF-compliant repository or tool. A brief outline of the mapping will be provided here. Each instance of a MOF Package, Class, or Association is represented by an XML element. In addition, every instance of a MOF Class contains an XMI identifier in the form of an attribute labelled “xmi.id” on the instance’s XML element. When a class instance appears by reference (rather in the form of a full declaration), it is referenced by an “xmi.idref” attribute in the XML element. MOF attributes can be represented in two ways. In the first, the attribute is represented in a separate XML element, with the attribute value expressed between start and end tags (with the exception of Boolean and enumerated attributes, which are expressed in an XML attribute of the child element).

In the second method, the attribute is represented as an XML attribute on the XML element of the class instance. This method cannot be used for contained class instance attributes.

One benefit gained from basing the XMI format on XML is that it makes available the use of the XML Stylesheet Language, or XSL. The W3C's XSL standard consists of two main documents: the XSL language itself and XSL transformations [XSL00, XSLT99], which allow for the parsing and processing of an XML document, and the subsequent production of a resultant document, in XML or some other form. Like other stylesheet languages such as CSS and DSSSL [CSS99, DSSSL96], XSL works by allowing the specification of a number of stylesheets, each of which are recursively matched and evaluated against the source document.

With these and other tools available, the XMI/XML format provides documents that can be easily consumed and produced by machines, but that are not easily readable or writable by humans.

3. Design of the HUTN Languages

3.1 Usability Considerations

The primary design goal for the generated languages was usability, and in particular the ability to learn the language quickly. However, before considering the usability aspects of the design, two conditions were established. Firstly, that the language use the ASCII character set and, secondly that the language be able to be fully generated without human intervention.

The first step in considering the usability issues was to identify the target user group of the languages. The decision was made to assume some basic programming language familiarity, but not to demand programming proficiency. Several previous works on programming language design for usability [McIver96, RL97] were consulted, and from these were assembled a set of principles on aspects such as the appropriate use of symbols and punctuation, the use of reserved words, and the satisfaction of user expectations. Although it is generally not part of a language's formal definition, indentation plays a large role in the readability and navigability of a language. As such, an indentation policy was also included in the design of the HUTN producer.

Despite its usability drawbacks, XMI does provide a semantic structure that is also be appropriate for human-

readable languages. As such, many aspects of the HUTN language are expressed as a translation of XMI.

The most significant decision made with respect to the usability of the languages was to include the ability to automatically generate a language. It was likely that users of the HUTN languages would be exposed to a number of generated languages, either through the evolution of a single domain model, or through the use of a number of models. As such, it was also necessary for the languages to exhibit a degree of uniformity across between generated languages. This was largely achieved through the use of a common basic structure, which is discussed further in the next section. One of the most powerful ways to increase a language's usability is to allow the designer to customise the language to better reflect and fit the domain model. For this purpose, the HUTN system allows a number of language customisations, which are described in Section 3.3. Since the language can still be generated without the customisations, the resultant design still satisfies the full-automation requirement set out above.

3.2 The Base Language

Reusing the structural and syntactic features of existing languages was identified as a good way to enhance the learnability of the HUTN languages, and to ensure that the user's expectations are satisfied. To this end, some consideration was given to the essential style of programming languages, so that it might be applied to the base language.

Languages, on the whole, represent information in a fairly similar way. A document invariably consists of a set of concepts, each of which consists of a number of other concepts, and so on until the concepts are nothing but simple pieces of atomic data. This can be seen in both procedural and object-oriented programming languages, as well as in natural English. For example, an English essay could be said to consist of a series of paragraphs, each of which contains a series of sentences, which in turn each contain a series of words. A piece of source code for the Java programming language could consist of a series of import statements, package statements, and class definitions, which contain variables and methods, which contain sets of parameters and statements.

At different levels of depth on this 'concept tree', the representation of the containing concept changes. One common change is for concepts higher on this tree to be introduced in some way. For example the essay with its paragraphs might first have a title, or chapters within a thesis might have chapter numbers and titles. A method

declaration in a Java class definition has a visibility value, a method name, and a return type. By contrast, where an element is the only possible element in its position, it may go without an introduction, such as sentences within a paragraph, or statements within a Java method definition. However, to be effective this requires some language familiarity on the part, something that cannot be assumed for the HUTN system.

In structured notations, such as programming languages, it is often necessary to separate the contained concepts using some form of punctuation. Java, for example, uses braces to delimit method bodies, commas to separate method parameters, and semicolons to terminate statements. Written English uses full stops to terminate sentences, and commas or parentheses to delimit phrases. The choice of symbols for separating punctuation can also be dependent on the depth of the concept on the tree. For example, braces are often associated in programming languages with high-level or major concepts such as procedure declarations, while commas are often associated with low-level or minor ones, such as a list of method parameters.

The MOF modelling concepts underlying the HUTN languages also conform to this 'concept tree' paradigm. Package instances contain Class instances, Class instances contain Attribute values, Association instances contain Class instances, and so on. Accordingly, the HUTN language core has been based around these ideas of concept containment, introduction and delimitation.

The MOF Class, Package, and Association concepts have been classified as 'major' concepts, warranting an introduction for their instances. The introduction is a simple one, consisting of the name of the Class, Package, or Association and some identifying string. (When converting from XMI, the XMI Id provides a logical and automatically unique identifier). The appearance of this introduction is very similar to the introductions of procedures or functions in Pascal or C. Curly braces, as used in many languages deriving syntactic features from C, are used to delimit the bodies of these major concepts. Class instances can also be referenced by other parts of the document. This is done by simply displaying the introduction of the instance without the body. Because it is always possible to know whether an instance or an instance reference will appear, this does not cause the parser problems that might otherwise occur.

By contrast, MOF Attributes are denoted as minor concepts, and as such are represented differently. In their case, the attribute name is followed by a colon, followed in turn by the value of the attribute. The attributes' representations are separated by white space, with no

other separator or terminator. Should one have been included, the logical choice for a terminator would have been the semicolon, as is commonly used in programming languages. However, white space is sufficient in this case, since it is always possible to know how many white-space separated 'words' will appear in an attribute's value. Simple attribute values cannot contain white space, with the exception of string-typed attributes, whose values are delimited by double-quote characters. Attributes whose values are class instances are represented either as instance references or as full instance declarations, depending on the nature of the attribute. These representations do have more than one 'word' in their value, but do not cause problems because the number and nature of the words are always known to the parser.

References are displayed with the reference name followed by a colon and the representation of the class instance that is referred to. This is almost identical to the representation of attributes, which could be seen as violating the principle of 'different forms for different features'. However, the role of references in the MOF is in many ways to provide a class instance with attribute-like access to other class-instances that are related through associations, but still provide the benefits of associations, such as visibility from both participants. For this reason, the underlying 'feature' of references and class-instance valued attributes is essentially the same, and thus their representations should in fact be similar.

3.3 User Customisations

The HUTN language generator provides six user customisations. Attributes can be declared as an identifying attribute, keyword, or adjective, or given a default value. References can be declared as either typeless or nameless, depending on their containment semantics. Each of these customisations is made on the attribute within the scope of a given class. Thus an attribute can have differing representations in different classes. For classes that inherit from other classes, conflicts between customisations are resolved by considering the subclass's customisation first. However, while a subclass can in this way override the customisation of its parent, it may not negate it. It was considered unlikely that such a negation would be necessary, and the resultant configuration syntax would be an unnecessary confusion.

3.3.1 Attribute Customisations

The first attribute customisation available is the selection of an attribute as the identifying attribute of a class. Class instances are objects that can be referred to by other objects, such as references and associations. By default, the attribute that appears in the introduction to a class instance, and by which the instance is identified, is the XMI Id, which bears no relation to the model or its domain. This customisation provides for the selection of a singleton string attribute to be used as the identifying attribute, and to appear in the introduction to the class instance. An attribute selected as the identifying attribute does not appear in the body of the class instance. Since MOF does not support the denotation of an attribute as being unique at any level, it is left up to the language designer to ensure that the attribute's values will be unique.

The second attribute customisation is the denotation of a singleton boolean attribute as a keyword. In essence, this means that the instead of representing the attribute with its name and value, that an attribute value of 'true' is represented by the attribute name, and an attribute value of 'false' is represented by the absence of the attribute name.

The third attribute customisation, the adjective, is almost identical to that of a keyword. Like a keyword, an adjective is a singleton boolean attribute whose state is determined by either the presence or absence of the attribute name. The difference is that the adjective is placed before the class name in the introduction of a class instance, rather than in the body. This is similar to the visibility and return type modifiers seen in languages such as C, Java and Pascal, although the programming modifiers are more often of other types than boolean, such as enumerations. While it would be a great stylistic advantage to allow other data types to be represented as adjectives, it raises difficulties with overlapping values, and makes the language more difficult to learn.

The fourth and final customisation on attributes within a class is to allocate the attribute a default value. For the producer, this means that the attribute will only be represented in the class instance's body if its value is something other than the default value. For the consumer, this means that the absence of the attribute in the body signifies that the attributes value is the same as the default value. This is very useful for attributes that are not interesting unless they are different. At present, only string attributes can be assigned default values, although there is no reason that this could not be expanded to include all

data types.

3.3.2 Reference Customisations

There are two customisations available for references within the context of a class. Only one of these is available for use, and which one depends on the containment properties of the association referred to by the reference. An association represents a *containment relationship* if one of the participating classes is wholly contained by the other. That is, the *contained instance* does not exist outside the scope of the other instance. This is equivalent to a black-diamond relationship in UML. A reference can be said to be a *containing reference* if the class to which the reference belongs is the containing class in the referred association.

If the reference is a containing reference, it may be customised as a nameless reference. This means that the name of the reference, which normally precludes the contained element in the reference representation, may be omitted. This is not possible if instances of a class can be contained via two different associations. However in most models, this is not the case.

If the reference is not a containing reference, it may be customised as a typeless reference. This means that the class name normally present before the referred object's identifier can be omitted. This customisation assumes that the identifiers of the referred classes will be unique.

4. HUTN Language Mappings

This chapter describes the syntax of the generated languages, in terms of the MOF modelling concepts as outlined in Section 2.1. The later sections of the chapter discuss the more generic syntactic features of the language: indentation, name-scope optimisation, and string delimitation.

The examples presented throughout this section (with the exception of the name-scope reduction examples) are derived from the FamilyPackage model, a simple model for describing families and the people, pets, and other objects associated with them.

4.1 Package Representations

A *Package* in the MOF type structure is a concept used for containing a collection of related classes and associations. Package instances are represented as simple block objects. Identifying attributes are not permitted on

packages and, as such, packages are prefaced and identified by the name of the package, followed by the string-delimited XMI ID. Between a set of braces appear the class and association instances of the package, in accordance with the mappings described below in Sections 4.2 and 4.5. An example of the representation of a package instance is given in Figure 1.

```
FamilyPackage "xmi-id-001" {
  Class instances here
  Association instances here
}
```

Figure 1: Package instance representation

4.2 Class Representations

As classes are the objects that convey the most information, they are the basis for all of the language configurations. There are six possible configurations, whose descriptions are provided in section 3.3.

The syntax of class instance definitions is as follows. Those attributes defined by the user as adjectives, and whose values are true, appear first, in keyword form, followed by the name of the class. This is followed by the class' identifier: the identifying attribute if one has been selected, or the XMI Id if not. The representations of the remaining attributes then appear between a set of braces, in the forms described in section 4.3. Figure 2 shows a representation of a class Family with an adjective 'nuclear' and a string-typed attribute 'familyName'. Though it is not possible for class instances to have different customisations, the first instance is shown with no identifying attribute, and the second with 'familyName' as the identifying attribute.

```
FamilyPackage "xmi-id-001" {
  Family "xmi-id-002" {
    familyName: "The McDonalds"
    Attribute representations
    Reference representations
  }
  nuclear Family "The Smiths" {
    Attribute representations
    Reference representations
  }
}
```

Figure 2: Class instance representation

The parser does not place restrictions on the order of the attribute and reference representations within a class instance. Similarly, no restrictions are placed on the order that a producer will output these representations, although it is expected that implementations of the producer will use a consistent ordering.

4.3 Attribute Representations

The representation of attributes is dependent upon whether or not the attributes have been customised in any way, on the type of the attribute, and on its multiplicity (whether it is optional, single-valued, or multi-valued). All attributes except those specified as keywords, are prefaced by the name of the attribute followed by a colon. This is followed by a representation of the attribute value.

String values comprise the only primitive type that is allowed to contain white space, and they therefore require a delimiter. A discussion of string delimiters and escape characters is included in section 4.8. All other primitive type values (whose attributes are not customised) are represented without delimiters. Enumerated attribute values are expressed with the value, and not the index, of the enumerated element.

Optional attributes are expressed in the same way as other attributes when their value is present, and are simply not represented when their value is absent. Multi-valued attributes are enclosed within parentheses, and are separated by white space. When multi-valued attributes are produced from XMI, primitive-typed lists are separated by a single space, and class-instance valued attributes are separated by a new-line character (the consumer does not enforce this spacing).

Figure 3 presents an example of a number of attributes' representations. The 'migrants' attribute has been defined as a keyword on the Family class, the 'nuclear' attribute as an adjective of Family, 'familyName' as the identifying attribute of Family, and 'name' has been selected as the identifier of Person.

```
FamilyPackage "xmi-id-001" {
  Family "The McDonalds" {
    migrants
    Address: "7 Main Street"
    Reference representations
  }
  nuclear Family "The Smiths" {
    Address: "5 Main Street"
    Reference representations
  }
  Person "Namdou Ndiaye" {
    age: 7
    sex: male
    Reference representations
  }
}
```

Figure 3: Simple attribute representation

Attributes whose values are instances of a class can be represented in two separate ways. If the attribute class instance is contained by the enclosing class instance (that is, it does not exist outside of the containing instance's

scope), then the attribute instance is represented as a class in the same manner as described in Section 4.2. Alternatively, if the attribute class instance is not contained, then it is represented simply by its class name followed by its identifier. (Where the identifier is the value of the identifying attribute of the class if one exists, or alternatively the instance's XMI Id). An example in which 'child' is a contained attribute and 'friend' is a non-contained attribute, both of Family, is presented in Figure 4.

```
FamilyPackage "xmi-id-001" {
  Family "The McDonalds" {
    friend: Person "Bruce Thompson"
    pet: Dog "Sparky" {
      Attribute and reference representations
    }
  }
  Friend "Bruce Thompson" {
    Attribute and reference representations
  }
}
```

Figure 4: Class-instance valued attribute representation

4.4 Reference Representations

References are a means for classes to be aware of class instances that play a part in an association, by providing a view into the association as it pertains to the observing instance. For this reason, the representation within a class instance of a reference depends in part on whether the reference is a containing reference (as described in section 3.3.2).

Like that of an attribute, the representation of a reference begins with the name of the reference followed by a colon. If the reference is a containing reference, then it is followed by a representation of the instance in accordance with the protocol for displaying class instances (Section 4.2). If the association that is referred to is not a containment relationship, then the subsequent depiction consists of the class name followed by the instance's identifier (being the value of the class's identifying attribute or the instance's XMI Id). Figure 5 shows the Family class with references, 'naturalChild' and 'adoptedChild', to a containing reference and a non-containing reference respectively.

If 'naturalChild' were customised as a nameless reference, the 'naturalChild:' prefix would be omitted. If 'adoptedChild' were declared as a typeless reference, the 'Person' token could be omitted.

```
FamilyPackage "xmi-id-001" {
  Family "The Smiths" {
    Attribute representations
```

```
Reference representations
naturalChild: Person "Harry Smith" {
  Attribute and reference representations
}
naturalChild: Person "Joan Smith" {
  Attribute and reference representations
}
adoptedChild: Person "Dylan Smith"
}
Person "Dylan Smith" {}
Attribute and reference representations
}
```

Figure 5: Contained reference representation

4.5 Association Representations

As described previously, an association constitutes a relationship between two classes, and can appear in two forms: either containment relationships or non-containment relationships. Further to this, classes can contain references into associations (see Section 4.4). This results in three separate methods for representing the links between class instances involved in an association.

If one or more of the classes participating in the association contains a reference into the association, then the link is displayed within the representation of the referencing class, as described above in Section 4.4.

If the association represents a containment relationship, and there are no classes containing references to the association, then the association contents are displayed within the representations of class instances. That is, the contained instance is shown within the representation of the containing instance. The representation for this contained instance is exactly the same as if it were referenced, except that the name of the association is substituted for the name of the reference. The display of the association name is necessary to avoid conflicts where two associations have contained instances of the same class. An example of the representation of unreferenced containment associations is presented in Figure 6, where CarOwnership is an association between Family and Car with Car instances being contained by Family instances.

The third method deals with the case when the association is not a containment relationship, and neither of the participating classes contains a reference into the association. In this case, a list appears containing references to the class instances participating in the association. In more detail, this list consists of the name of the association followed by a block (denoted by opening and closing braces) containing the pairs of instances participating in the relationship. Each instance in the pair will be preceded by the name of the role it plays in the

association and a colon. Each class instance is then represented simply by the name of the instance's class, followed by its identifier.

Figure 6 shows the latter two types of association representation, with CarOwnership being an association in which the Family class instances contain Car instances, and sponsorship being a non-containment association between Family and Person.

```

FamilyPackage "xmi-id-001" {
  Family "The McDonalds" {
    CarOwnership: Car "755-BDL" {
      Attribute and reference representations
    }
  }
  Person "Bob Briggs" {
    Attribute and reference representations
  }
  sponsorship {
    sponsor: Family "The McDonalds"
    sponsored: Person "Bob Briggs"
    Other pairs in the sponsorship association
  }
}

```

Figure 6: Containment association representation

4.6 Indentation

The stream generated from the repository will be significantly more acceptable to the reader if it is appropriately indented. This is a feature that need only be enforced in the producer, since it was decided that a prescribed indentation style in the parser was not appropriate.

In many programming languages, the start of a new major concept (such as a procedure or class) is denoted by a hanging indent. This is also the case in these languages, with a hanging indent introduced by new package instances, new class instances, and new association instances. Lines that run over 80 characters in length are likely to cause inconveniences on some systems, so these lines will be broken up (placing the break between words rather than within words, wherever possible). The overflow will be placed on the next line with two additional temporary indentations. The language extracts provided as examples in the previous sections provide demonstrations of the indentation styles adopted for each concept.

4.7 Name Scope Optimisation

Names of packages, associations, and classes in the MOF include all of the information about the concept's scope.

This fully qualified name consists of a number of scope-level components, separated by dots. For example, an attribute contains information about which class it is in, and what package that class is contained by. However, while this scope information is necessary in a broad context, these names provide more information than is necessary to uniquely identify a model concept within the model.

The names of packages, associations, and classes are therefore optimised to make them as short as possible while still being unique within the domain model. (Since attribute names are unique within their class, they are simply represented by their local name). This is done as follows. First, a set of all names is assembled, and each is broken down into a sequence of words (one for each scope level). A possible scoped name is then created for each name, constituting the last word of the word sequence for that name. If this possible name is unique within the set of possible names, then it is accepted as the scope-optimised name. If not, then the process is repeated with the last two words of the name sequence. This continues until all names have been optimised. The table shown in Table 1 presents an example of a set of names and their reductions.

Table 1: Name optimisations

Fully Scoped Name	Scope-Optimised Name
Genealogy.Family.Child	Family.Child
Genealogy.Family.Father	Father
Genealogy.Tree.Child	Tree.Child
Genealogy.Tree.Branch	Genealogy.Tree.Branch
Flora.Tree.Branch	Flora.Tree.Branch
Flora.Flower	Flower

4.8 String Delimitation

The HUTN system provides all string attributes (including identifiers) with double quote symbols for string delimiters. The backslash character is defined as an escape character, so that the “\” sequence symbolises a literal quote character, and a “\\” sequence denotes a literal backslash character. In addition to these, other escape characters such as \n and \t are also available.

For the extraction of string values from an XMI stream, the assumption is made that any of the leading and trailing white space is superfluous, and it is thus stripped from the

string's representation in the HUTN language.

5. A Prototype HUTN generator.

This section outlines the development of a prototype HUTN system. Discussion is first made on the position of the system with respect to existing components. Following this, the components implemented for the purposes of the prototype will be described.

The design of the system is illustrated below in Figure 7, with the components to be implemented as part of the HUTN system shaded. These new components depend heavily on a number of existing programs. The MOF Model Repository is a repository for information models, which are created in a custom model definition language (called the Meta-Object Definition Language or MODL). DSTC's dMOF product [Dmof01] is used for this purpose, since it has the advantage of being able to generate fully functional instance repositories from the model in the Model Repository. These instance repositories expose CORBA interfaces compliant with the MOF Specification [Mof00].

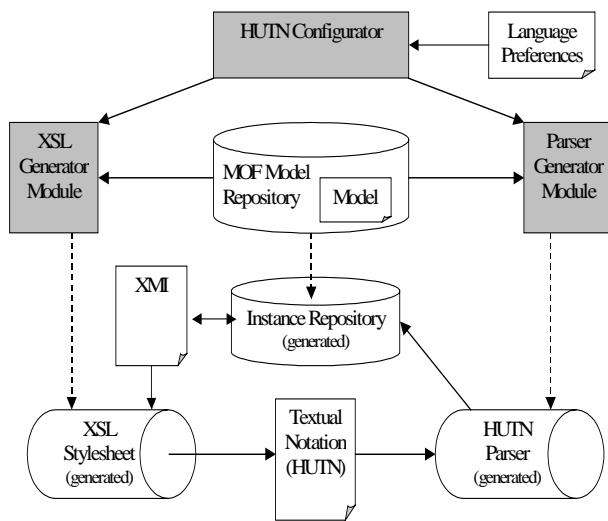


Figure 7: Structure of the HUTN System

The HUTN system is divided into three basic components. The XSL generator component is responsible for the creation of an XSL style sheet for converting a stream of XMI into the target human-usable language. The Grammar Generator component generates a grammar and associated backend code for the parsing of the language back into a MOF-compliant repository. Finally, the Configurator component is responsible for parsing a file

containing language configurations, and for communicating these preferences to the two generator components.

The Grammar Generator and the XSL Generator components are designed around a common generator architecture, that provides a simple mechanism for communicating with the MOF. The architecture is enacted through the use of an existing Java package included as part of the MOF system. This package and the architecture it supports are described in Section 5.1, followed by descriptions of the three major components of the HUTN system.

It was decided to implement the HUTN modules in the Java programming language. Java provides a number of features that make it suitable for this project, such as its mature object orientation and use of interfaces, and its connectivity with CORBA. The CORBA product used for this implementation was Inprise's "Visibroker 4.0 for Java" product [Visibroker]. This was chosen because it is the system used in the dMOF product, and was thus less likely to induce compatibility problems.

5.1 The Generator Architecture

The generator architecture used for the XSL and Parser generator modules is based on a package used for retrieving and acting on model information in a MOF repository. The package is designed to serve as a utility framework for programs that generate code or conduct similar model-driven activities. The classes were originally designed for the repository generation facility included in a predecessor of the dMOF product, and similar classes are used in the present dMOF generators.

The package provides one class for each MOF modelling concept (for example, Class, Association, Data Type), and implements methods for retrieving most of the common features of that element that might be useful for a system accessing a model. For more obscure features, a system can use the methods provided by the MOF's CORBA interfaces. Typically, a system using the package will extend a number of these classes to include generation or activity functions as required.

One of the important features provided by the package is that of object caching. This avoids the creation of duplicate generator objects, when model components are accessed by more than one route. For example, if a class object is created by requesting all classes within a package, it need not be instantiated again if it is requested as a role in an association. Also, since modules using this

package often extend the generator classes, the object caching mechanism can be extended to handle the instantiation of these extended generator classes in place of the base classes.

In the cases of the XSL Generator module and the Parser Generator module, classes are created extending the generator classes, with methods for generating the respective XSL and compiler source code, as well as some additional utility methods. The process of generating begins with the acquisition of a reference to some top-level object, such as a package definition. This object is an instance of an extended generator class, and is activated using some generator method. This method in turn calls the generator methods on the classes and associations contained within the package. This passing of control further down the hierarchy continues until the generation can be performed by a single object without delegation to another object. At this point the classes below on the hierarchy need not be extended from the base generator classes.

5.2 The XSL Generator

The XSL generator module creates an XSL style sheet that provides a mapping from the XMI format of the modelled data to the syntax required as per Section 4. This is achieved using the architecture described above, with a series of cascading generate methods beginning at the package level. The style sheet comprises a series of template rules, one for each package, class and association that appears in the data model. When the style sheet is activated, it searches the top level of the XMI tree for any of the appropriate objects. When it finds an element matching a template, it transfers control to the template.

Package and association templates behave in almost exactly the same manner. Once a package or association instance element is encountered, the appropriate template simply displays the introductory information according to the rules listed in Section 4. It then reverts to applying templates to process the class instances, association instances, and class instance references that appear within the package or association instance.

The generator produces three templates for each class in the information model. The first is activated for a class instance that appears directly within a package instance, and not as a contained instance within another class instance. This template does the representation for the class structure and for the simple attributes. Attributes whose values are embedded class instances or references to class instances are dealt with by delegating control to

their relevant templates. The second template processes a class instance that appears embedded within another class instance. Since there is no XMI Id in this case, if the class has no identifying attribute defined, then the template creates a random identifier. Other than this difference, the template behaves in the same manner as the previous one. The third template is for a class instance reference, and simply produces the short representation as defined in Section 4.4. Since in XMI an identifying attribute's value only appears in the declaration of a class instance, and not in that of a class instance reference, the template must find the value of this attribute somewhere else on the document tree. It does this by defining the "xmi.id" attribute as an XML identifying attribute in the XMI DTD. In conjunction with the "xmi.idref" attribute supplied in the instance reference element, the template can locate the node on the tree where the referred class instance is declared, and thus find the value of the identifying attribute.

5.3 The Grammar Generator

The role performed by the Parser Generator is divided into two parts: grammar generation and backend generation. The grammar generation is responsible for the production of a language grammar file that can recognise a HUTN stream, reconstruct the data types, and provide the information to a backend that processes it as required. The backend is responsible for the addition of the information back into the instance repository. The reason for this partitioning is for flexibility, as at some point it may be useful for the system to be able to revert the HUTN stream into XMI, or to act on it in some different way. Alterations such as these might then be realised simply by providing the appropriate method implementations and attaching a new backend generator to the existing generator.

The grammar file constructed by this module is for the javaCC [JavaCC] program, produced by Meta-mata. JavaCC input consists of a driving program followed by a number of lexical and parsing rules, and this is then used to produce an LL(k) parser. The product was chosen over other Java-based compiler-compiler packages for its integrated lexical analysis features. While other packages often require a separate program for lexical analysis, javaCC allows for lexical rules intermingled with parse rules, which is useful for the purposes of the HUTN system.

The first section of the generated grammar file comprises a simple program that initialises the data stream and

parses the command line arguments. The program also includes a simple error detection and display mechanism. This is followed by a number of lexical declarations such as numbers and string constants, which are followed in turn by the parsing rules. One parsing rule is constructed for each package, class, and association.

The package and association rules are simple. Once the data has been parsed, the rule simply delegates to a method on the backend and delegates control to the next level of the concrete syntax tree.

Class rules also have the task of marshalling the data type values that are parsed. Since all values are read in as strings, the first step arranges the values into variables in line with those that are passed to the constructors in the repository. Simple types are presented in the form designated by the repository's CORBA interfaces. Enumerated types are passed as integers, and class instances are passed as a string concatenation of the instance's class name and its identifier. Because there are restrictions placed on the number of occurrences that an attribute can make, these constraints must be checked next. Once this is done, the attributes and references are conveyed to the appropriate method on the backend.

An additional parsing rule is constructed for each enumerated type, as well as a single rule for booleans. In this way, 'true', 'false' and the values of the enumerated types form the set of reserved words in the language. The final element of the generated grammars is a simple lexical rule for identifiers. Declaring this rule after the parse rules allows for the declaration of string literals within rules without causing reduce-reduce conflicts in the parser.

The Java code generated for the parser backend is responsible for the addition of objects into the instance repository. However, since class instances are passed only as string concatenations of class names and identifiers, a correlation must be kept of identifying strings and actual repository object references. This is done by the backend code. Also, to avoid duplicate entries and forward referencing problems in associations and references, all entries in associations (regardless of how they appear) are stored and added to the repository after parsing has been performed. This requires a storage mechanism for each association in the model. Beyond these tasks, the role of the backend is simple.

The first method generated is an initialise method, that is provided with a reference to an instance repository and initialises the association stores, name mappings, and the backend's connection to the repository. A method is then

generated for each class, package and association in the model. For classes and packages these methods check for name inconsistencies and create the appropriate objects in the instance repository. For associations, the methods simply add entries into the post-processing store the relevant association. The backend lastly contains a finalise method that transfers these stored association entries into the repository and closes its connection with the repository.

5.4 The HUTN Configurator

The HUTN configurator is a simple parser designed for the acquisition of the user's language preferences, and for the transfer of these to the generation modules. The parser is itself generated using the above parser generation tool, from a MOF model of the available configurations. Once parsed in, language configurations are stored in an instance repository and accessed by the XSL and parser generators using CORBA interfaces.

6. Conclusions

There are three properties that make the HUTN system useful. The first is that it is generic, in that it can provide a language for any model that can be specified using the MOF techniques. Secondly, the system provides full automation, which is useful for systems whose information models are undergoing change. Thirdly, the customisation mechanisms available provide the user with a degree of control over the appearance and consequently the usability of the language.

Generic

The language mappings described in Section 5 provide a set of syntactic rules providing complete coverage for all of the significant MOF modelling concepts. This means that a language can be rapidly created for any model specified using these concepts. In addition, since the MOF modelling concepts have been designed as a basic set of common concepts, there will almost always be a simple mapping from these concepts to alternative modelling techniques. Therefore it should be possible to use the syntax described to develop a similar system for other modelling and repository tools, such as Rational Rose.

The proposal that resulted in the XMI standard was supported and created by a number of companies that play a significant role in the modelling and repositories field, such as Rational, Unisys and IBM, and the format is

gaining in popularity and use in applications such as UML design tools. Since the HUTN system is based on transformations to (and potentially from) XMI, it could of course be used with XMI from sources other than the MOF. A pre-requisite would be the availability of a MOF model from which to generate the producer and consumer.

Fully Automated

The second benefit of the system is that it is fully automated. The task involved in the manual implementation of a parser allows for greater flexibility in language design, but requires a significant amount of time and effort. Furthermore, a manually constructed parser is subject to problems with information models that are likely to change. Automated generation means that changes made to a language, be they as a result of a change in the underlying model or a change in the syntax, can be implemented uniformly and quickly across the entire system. In this way, automation avoids problems of consistency in changing languages, and greatly reduces the time involved in the evolution of an information model/repository suite.

User-customisable

The third benefit provided by the HUTN system is that of flexibility of syntax. Having established a base language that provides generic usability and functionality, the ability to alter the representation of some attributes within a class instance means that some of the usability decisions of the language can be passed back to the creator (and presumably the user) of the language. This can only serve to enhance the usability of the language, by allowing domain knowledge to be used to optimise the language for its application.

7. 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).

8. References

[CSS99] Cascading Style Sheets level 1, W3C Recommendation 17 December 1996, revised 11 Jan 1999. <http://www.w3.org/TR/1999/REC-CSS1-19990111>.

[Dmof01] DSTC. dMOF – DSTC's Meta-Object Facility (MOF) Product. <http://www.dstc.edu.au/Products/CORBA/MOF>.

[DSSSL96] The Document Style and Semantics Specification Language (DSSSL) Standard, International Standard ISO/IEC 10179:1996(E).

[Hutn99] A Human-Usable Textual Notation for the UML Profile for EDOC. Request for Proposal, OMG Document ad/99-03-12.

[Java] James Gosling, Bill Joy and Guy Steele. "The Java™ Language Specification", First Edition. Sun Microsystems, 1996

[JavaCC] Sun Microsystems & Metamata. The Java™ Parser Generator. <http://www.metamata.com/javacc/index.html>.

[McIver96] Linda McIver and Damian Conway. Seven Deadly Sins of Introductory Programming Language Design. In Proceedings, 1996 Conference on Software Engineering: Education and Practice. IEEE Computing Society Press, Los Alamitos, CA, USA 1996. Pp 309-316.

[Mof00] Meta-Object Facility (MOF) Version 1.3 Specification. OMG TC document formal/2000-04-03, 2000.

[RL77] Frederic Richard and Henry F. Ledgard. A Reminder for Language Designers. ACM SIGPLAN Notices, Vol. 12 No. 12 (December 1977). Pp 73-82.

[Visibroker] Inprise Corporation. "Visibroker 3.4 for Java". <http://www.visigenic.com/visibroker/>

[XMI00] XML-Based Model Interchange (XMI) Version 1.1 Specification, OMG TC document formal/2000-11-02, 2000.

[XML98] eXtensible Markup Language (XML) 1.0, World Wide Web Consortium Recommendation 10-February-1998. <http://www.w3.org/TR/1998/REC-xml-19980210>.

[XSL00] Extensible Stylesheet Language (XSL) Specification, W3C Candidate Recommendation 21 November 2000. <http://www.w3.org/TR/2000/CR-xsl-20001121>.

[XSLT99] XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999. <http://www.w3.org/TR/2000/CR-xslt-19991116>.