

The IDL*flex* mapping specification language

(Version 1.1)

Hans Reiser
University of Erlangen-Nürnberg

September 5, 2001

1 Introduction

The purpose of this document is to provide a complete manual for the development of IDL*flex* mapping programs. IDL*flex* is a generic, flexible IDL compiler developed at the University of Erlangen-Nürnberg. It is able to generate arbitrary code from CORBA IDL. The mapping from IDL to a specific programming language is defined in an XML based mapping language. This language is design to be simple and easy to read, and is not adequate to express rather complex operations, which form only a minor part of the code generation process. Thus, the mapping developer may provide some extra functionality by a Java based *Utility* class.

The following section specifies the syntax and semantics of the XML tags (statements) used by the mapping programming language. Section 3 describes the interface and the functionality of the *Utility* class. Section 4 explains the internal representation of IDL definitions.

2 XML mapping program

The XML tags can be grouped into several categories, as shown by Table 2. A full listing of the corresponding DTD (document type definition) can be found in appendix B. The syntax and semantics of all XML tags is explained in detail in the next sections.

At any time there is an implicit reference to an IDL object associated with the execution of each XML statement. This implicit reference is called `currentIDL`. All

IDLflex COMPONENT CALL	Structuring the XML mapping program
FILE SBOX	Handling code output streams
GET	Dynamic creation of text
IF SWITCH CASE	Conditional expressions
ITERATE	Iterating over content of the IDL definitions
PLUGIN ERROR	Other

Figure 1: XML tags of the *IDLflex* mapping language

operations that need informations from IDL definitions obtain their data from the definition addressed by this reference. In most XML statements it is possible to explicitly change the `currentIDL` reference via an attribute “OBJ”.

2.1 Structuring the XML mapping program

```
<IDLflex ROOT="..." UTILITY="...">
...
</IDLflex>
```

This is the master element of any XML mapping program. Each program must contain exactly one `IDLflex` element. There are two attributes: `ROOT` specifies the root component. This component is the first component to be executed with `currentIDL` initialized to the root container of the IDL object tree. Thus it corresponds to the main method of a traditional C program. `UTILITY` specifies which *Utility* class the IDL compiler shall use. This class is written in Java.

```
<COMPONENT NAME="...">
...
</COMPONENT>
```

The whole XML mapping program can be structured into so-called components. Each component can be “executed” multiple times, i.e. they can be used like subroutines in

traditional programming languages. Thus they enable writing easy-to-read programs and also provide the possibility to reuse components multiple times.

```
<CALL NAME="..." [OBJ="..."]/>
```

The `CALL` tag is used to execute another component as subroutine. The `NAME` attribute specifies the component's name. The optional `OBJ` attribute changes `currentIDL` while executing the called component. After returning from the subroutine `currentIDL` is restored to its previous value.

2.2 Handling output

To create output it is necessary to open an output stream. There are two statements for opening such a stream: `FILE` and `SBOX`. Everything between the start tag and the end tag of `FILE` resp. `SBOX` is used to create the output. This content may consist of plain text, which is directly used as output, and of other XML statements, which also may produce output.

In the case of nested `FILE` or `SBOX` statements, the inner-most output stream is used as the active stream, and the outer streams are temporarily suspended.

```
<FILE SPEC="..." [OBJ="..."]> ...(content to be written)... </FILE>
```

The `FILE` statement is used to create an file output stream. The translation of the argument `SPEC` to a file name is done by the *Utility* class. See section 3 for more details.

```
<SBOX NAME="..." [OBJ="..."]> ...(content to be written)... </SBOX>
```

This statement creates a special case of an output stream: The data is written to an internal buffer (save-box), which can later be used to insert it in various places in other output streams.

Both variants allow changing `currentIDL`. In the case of `FILE`, the new reference is passed to the *Utility* class.

2.3 Dynamic creation of text

```
<GET T="..." [OBJ="..."]/>
```

The `GET` statement is used to create text dynamically. The attribute `T` controls the text generation. *IDLflex* defines some standard labels for this attribute as shown below. The set of these core labels is usually extended by the destination language specific *Utility* class.

```

module A {
    interface C {
        // ...
    }
}

```

Figure 2: IDL definition example

No restrictions are made on the name space for such extension labels. The *Utility* class may even overload some of the standard labels. Note that **GET** is an empty XML tag.

Currently predefined labels for the **T** attribute:

```
<GET T="DEF:xxx"/>
```

Returns the value *yyy*, which was defined by `-Dxxx=yyy` on the *IDLflex* command line.

```
<GET T="IDL:xxx"/>
```

Retrieves a value directly from the IDL information referenced by **currentIDL**. Most IDL elements allow retrieving **IDL:id** (the IDL-ID), **IDL:name** (the name of the IDL element) and **IDL:fullname** (the fully scoped name of the IDL element). More details can be found in chapter 4, which explains the internal representation of IDL definitions and lists the labels for all IDL informations.

Example:

Assume the IDL definition shown in figure 2 and let **currentIDL** point to the definition of interface C. Then **IDL:name** produces the interface name C, **IDL:fullname** produces the fully scoped name `:A:C` and **IDL:id** produces `IDL:A/C:1.0`.

```
<GET T="SBOX:xxx"/>
```

Inserts the content of the save-box with the specified name at the current position of the output stream. See **<SBOX>** for further details on save-boxes.

```
<GET T="LIST:Count:xxx"/>
```

This statement retrieves the number of elements in **currentIDL**'s member list with the name *xxx*.

```
<GET T="LOOP:Count"/>
```

This statement retrieves the number of elements in the member list of the inner-most iteration loop. This is equivalent to changing the IDL reference to the appropriate element and then getting `LIST:Count:xxx`.

```
<GET T="LOOP:Index"/>
```

This get label retrieves the current iteration counter of the inner-most iteration loop, written in decimal notation. Iterations are counted starting from 0.

The additional labels that are defined in the *Utility* class of our Java mapping are shown in appendix A.

2.4 Conditional expressions

IDL*flex* supports two types of conditional expression:

```
<IF [OBJ="..."] COND="..."> ...conditionally created text... </IF>
```

```
<SWITCH [OBJ="..."]>
  <CASE [OBJ="..." COND="..."> ... </COND>
  <CASE [OBJ="..." COND="..."> ... </COND>
  [<DEFAULT> ... </DEFAULT>]
</SWITCH>
```

The simple IF statement evaluates the condition and only if this evaluation yields true it executes its content. The SWITCH statement allows choosing one out of many branches. Only the first CASE with a true condition is executed. If no true condition is found, the (optional) DEFAULT branch is used. Again, the OBJ attribute may be used to change the **current** IDL reference. The new reference is used while evaluating the condition and while executing the body content. When using OBJ in a SWITCH statement, the new reference is used for all cases. In the case of using OBJ in a CASE statement, it refers only to this statement and its body.

Similar to GET there is a predefined set of condition labels, which can be extended via the *Utility* class. Currently the following predefined labels are supported:

```
<IF COND="DEF:xxx"> ... </IF>
```

This condition is true if `xxx` was defined on IDL*flex*'s command line (`-Dxxx`).

```
<IF COND="LOOP:First"> ... </IF>
<IF COND="LOOP:Last"> ... </IF>
```

These conditions may only be used inside the scope of an `ITERATE` statement. They yield true if the iteration is currently working on the first resp. last element of its iteration container.

```
<IF COND="HAVE:xxx"> ... </IF>
```

This condition yields true if the current IDL definition contains a non-empty container named `xxx`. See the section *Iterating* and the chapter on the internal representation of IDL definitions on more info about IDL containers.

```
<IF COND="IDL:xxx"> ... </IF>
```

This condition evaluates a property of an IDL definition. The supported labels `xxx` depend on the type of the current IDL definition referenced by `currentIDL`. See the chapter 4 for further details.

```
<IF COND="TYPE:xxx"> ... </IF>
<IF TYPE="xxx"> ... </IF>
```

These two statements are equivalent in their semantic. The second form is defined as a short cut for the first one for historical reasons. The statements verify the type of the IDL definition referenced by `currentIDL`. The set of existing types and their labels is also described in section 4

It is possible to derive more conditions from these core labels: Prepending a “!” to a label negates the condition. Concatenating several core or negated conditions with the “|” operator creates a new condition that is true as soon as one of its components is true.

For example, consider these two statements:

```
<IF COND="!LOOP:First"> ... </IF>
<IF TYPE="ArrayObj|SequenceObj"> ... </IF>
```

The first expression is true, if the current iteration is not the first iteration. The second statement is true, if the current IDL reference points to an array or sequence definition.

2.5 Iterating

```
<ITERATE [OBJ="..."] NAME="..."> ... </ITERATE>
```

```

class Utility {
    String getName(String label);
    boolean getAttribute(String label);
}

```

Figure 3: Public interface of the *Utility* class

The **ITERATE** statement allows the iteration over content lists of IDL definitions. Depending on the IDL definition, different content lists are possible, like the parameters of a method, the exceptions thrown by a method, the member elements of a struct, etc. Each content list can be referenced by a name. The names of content lists are shown in chapter 4, which explains the internal representation of IDL definitions.

ITERATE is the only statement which implicitly changes the **currentIDL** reference: For each member of the content list, the reference is set to this member and the body of the **ITERATE** statement is executed.

3 The *Utility* class

The *Utility* class may be used to provide additional, target language specific functionality. As described above the XML mapping language interacts with the *Utility* class when the **GET**, **IF/SWITCH**, **FILE** and **PLUGIN** statements are executed.

Figure 3 shows the public interface of the *Utility* class. Any language specific *Utility* class must be derived from this superclass.

The **GET** statement causes *IDLflex* to call the **getName** method of the *Utility* class. The user provided subclass should handle all of its own labels, and pass all other labels to the superclass.

Similarly, the **IF** and **CASE** statements use the method **getAttribute** to make a decision. The handling of negation and composition by the or operator are handled by the compiler, and **getAttribute** gets called for each individual condition fragment. Again, the user-provided utility class shall process all its own condition labels and pass all others to the super class.

The constructor of the *Utility* class may register plugin objects with the compiler in its constructor. These may then be instantiated and used as described above with the **PLUGIN** statement. All instances of an plugin module have an **ID** which can be used to reference it.

4 IDL Objects

... Internal representation of IDL constructs ... (TODO)

A The Java Utility class

Our Java Utility class defines following labels for the `GET` statement:

```
<GET T="JAVA:PkgName"/>
<GET T="JAVA:PkgDecl"/>
```

These two statements retrieve the Java package name where the file that is currently being created is located. The second variant produces a complete package declaration (`package ...;`). It suppresses its output, if the current file is not inside the scope of any package.

```
<GET T="JAVA:ConstVal"/>
<GET T="JAVA:DiscrVal"/>
```

The first statement produces the value of a constant declaration, formatted in correct syntax for a Java constant. It only may be used with `currentIDL` pointing to a constant declaration.

The second statement can be called with `currentIDL` referencing a `UnionObj` or a `UnionMemberObj`. For `UnionObj`, it retrieves the value used to represent the default branch. For `UnionMemberObj`, it retrieves the discriminator of this union member. Each value is represented numerically.

```
<GET T="JAVA:TYPE:name"/>
<GET T="JAVA:TYPE:decl"/>
<GET T="JAVA:TYPE:holder"/>
<GET T="JAVA:TYPE:helper"/>

<GET T="JAVA:TYPE:stub"/>
<GET T="JAVA:TYPE:skeleton"/>
<GET T="JAVA:TYPE:operationif"/>
<GET T="JAVA:TYPE:tie"/>

<GET T="JAVA:TYPE:basicidl"/>
<GET T="JAVA:TYPE:newarray"/>
```

This set of labels is used to create Java names. The first statement uses `IDL:name` to produce a valid Java name. In this process, it does all transformations for name collision avoidance with Java reserved words as defined in the CORBA standard. All other statements use additional transformations of this names. First, they conditionally produce fully scoped names, when it is necessary, i.e. if the file that currently is being created contains a different Java package than the referenced `currentIDL` object belongs to. Then, they perform further changes to create names for Holder and Helper class. For `currentIDL` pointing to an interface, you can produce the name of Stub, Skeleton, Operation Interface and Tie classes.

B XML DTD definition

```
<!ENTITY % all "#PCDATA|CALL|FILE|SBOX|GET|IF|SWITCH|ITERATE|PLUGIN|ERROR">
```

```
<!ENTITY lt      "&#38;#60;">
```

```
<!ENTITY gt      "&#62;">
```

```
<!ENTITY amp     "&#38;#38;">
```

```
<!ENTITY apos    "&#39;">
```

```
<!ENTITY quot    "&#34;">
```

```
<!ELEMENT IDLflex (COMPONENT|TABLE|MAPTABLE)*>
```

```
<!-- ATTLIST IDLflex ROOT CDATA #REQUIRED
      UTILITY CDATA #REQUIRED
      WRITER CDATA #IMPLIED-->
```

```
<!-- ELEMENT IMPORT EMPTY-->
```

```
<!-- ATTLIST IMPORT NAME CDATA #REQUIRED-->
```

```
<!-- ELEMENT TABLE (#PCDATA|ENTRY)*-->
```

```
<!-- ATTLIST TABLE NAME CDATA #REQUIRED SEPARATOR CDATA #IMPLIED-->
```

```
<!-- ELEMENT ENTRY EMPTY-->
```

```
<!-- ATTLIST ENTRY TAG CDATA #REQUIRED-->
```

```
<!-- ELEMENT MAPTABLE (#PCDATA|MAP)*-->
```

```
<!-- ATTLIST MAPTABLE NAME CDATA #REQUIRED-->
```

```
<!-- ELEMENT MAP EMPTY-->
```

```
<!-- ATTLIST MAP TAG CDATA #REQUIRED
      MAP CDATA #REQUIRED-->
```

```
<!-- ELEMENT COMPONENT (%all;)*-->
```

```
<!-- ATTLIST COMPONENT
      NAME CDATA #IMPLIED-->
```

```
<!-- ELEMENT CALL EMPTY-->
```

```
<!-- ATTLIST CALL
      NAME CDATA #REQUIRED
      OBJ CDATA #IMPLIED-->
```

```
<!-- ELEMENT FILE (%all;)*-->
```

```
<!-- ATTLIST FILE
      SPEC CDATA #REQUIRED
      OBJ CDATA #IMPLIED-->
```

```

<!ELEMENT SBOX (%all;)*>
<!ATTLIST SBOX
    NAME CDATA #IMPLIED
    DELETE CDATA #IMPLIED
    OBJ CDATA #IMPLIED>

<!ELEMENT GET EMPTY>
<!ATTLIST GET
    OBJ CDATA #IMPLIED
    T CDATA #REQUIRED>

<!ELEMENT IF (%all;)*>
<!ATTLIST IF
    OBJ CDATA #IMPLIED
    TYPE CDATA #IMPLIED
    COND CDATA #IMPLIED>

<!ELEMENT SWITCH (CASE|DEFAULT)*>
<!ATTLIST SWITCH
    OBJ CDATA #IMPLIED>

<!ELEMENT CASE (%all;)*>
<!ATTLIST CASE
    OBJ CDATA #IMPLIED
    TYPE CDATA #IMPLIED
    COND CDATA #IMPLIED>

<!ELEMENT DEFAULT (%all;)*>
<!ATTLIST DEFAULT
    OBJ CDATA #IMPLIED
    TYPE CDATA #IMPLIED
    COND CDATA #IMPLIED>

<!ELEMENT ITERATE (%all;)*>
<!ATTLIST ITERATE
    OBJ CDATA #IMPLIED
    NAME CDATA #REQUIRED>

<!ELEMENT PLUGIN EMPTY>
<!ATTLIST PLUGIN
    ID CDATA #REQUIRED
    OP CDATA #REQUIRED>

<!ELEMENT ERROR (#PCDATA)>

```