Hans Reiser, Martin Steckermeier, Franz J. Hauck

IDLflex: A flexible and generic compiler for CORBA IDL

Technical Report TR-I4-01-06
2001-06-15

Friedrich-Alexander-University Erlangen-Nürnberg, Germany

Informatik 4 (Distributed Systems and Operating Systems)
Prof. Dr. Fridolin Hofmann

This paper has also been published as:

Hans Reiser, Martin Steckermeier, Franz J. Hauck: IDLflex: A flexible and generic compiler for CORBA IDL. *Proc. of the Net.ObjectDays* (Erfurt, Germany, Sep. 10-13, 2001), 2001. pp. 151–160.

IDL*flex:*
A flexible and generic compiler for CORBA IDL

Hans Reiser, Martin Steckermeier, Franz J. Hauck

Informatik 4, University of Erlangen-Nürnberg

{hsreiser, mstecker, fzhauck}@immd4.informatik.uni-erlangen.de

Abstract: *For the development of CORBA applications, an IDL compiler is needed that generates code for communication stubs, helper classes and implementation skeletons. For each IDL language mapping, for every version of a particular language mapping, and for every CORBA implementation, the generated code has to be different. Typically many different compilers have to be programmed because the code generation is usually hard-wired into a compiler. IDLflex is a generic IDL compiler which is able to generate arbitrary code for arbitrary languages. Only a mapping program written in an XML-based mapping language and a language-specific utility class have to be provided. Thus, IDLflex can be adapted to another language, another mapping or to another ORB implementation in a very fast way. Furthermore, IDLflex allows to easily integrate additional functionality into a CORBA-based system, as it was done within the AspectIX middleware project.*

1 Introduction

For the development of distributed applications, middleware platforms are used which support communication between the distributed components of an application. One such middleware is the Common Object Request Broker Architecture (CORBA) [5]. CORBA is an architecture that is realized by a particular CORBA implementation. Traditionally there are many vendors distributing different CORBA implementations.

CORBA allows the development of applications in terms of distributed objects, which can be written in various programming languages. To hide an object's implementation language from its clients, CORBA provides a so-called Interface Definition Language (IDL) which is used to define the interfaces of distributed objects. For each programming language, a language-mapping standard defines how IDL types are mapped to language types, how parameters are passed, how the interface of an object looks like, how the client can use object references, etc.

Communication in CORBA is defined in terms of remote object invocations. Clients can hold object references and invoke methods at the referred object. The invocation of a method will finally invoke a method at the possibly remote server object. For the implementation of remote invocations, CORBA uses a client-side stub. The stub code depends on the corresponding IDL-based interface definition, the language mapping and on the particular communication mechanisms used by a CORBA implementation. The stub code is automatically generated by a code-generator tool, usually called an IDL compiler.

First, an application developer describes the interface of a distributed object in IDL. Second, he will run the IDL compiler for the chosen ORB implementation and target programming language. The IDL compiler will generate a so-called skeleton which is a code frame for the object implementation. The application developer has to insert the actual object code into the skeleton in order to create the object implementation. For the client side of a remote object, the IDL compiler will generate code for the necessary stub objects. However, the chosen programming language for stub objects may be different from the programming language for the object implementation, as CORBA supports heterogeneous programming environments; different IDL compilers may be used to generate either code. Beside skeleton and stub code, an IDL compiler generates code for auxiliary programming elements that are prescribed by the corresponding language mapping (e.g., the Java language mapping defines holder and helper classes for certain types [4]).

Thus, an IDL compiler becomes an important part of a particular CORBA implementation. Usually, such compilers are not very flexible, because the code generation is hard-wired into the compiler. For each supported programming language there needs to be another compiler. With every new version of a language mapping the compiler code has to be adapted. Different vendors may have to use different compilers because the communication code inside of stub objects is usually vendor-dependent.

IDL*flex* is a novel approach to IDL compilers. The IDL-specific part is hard-wired into the compiler code whereas the mapping-specific code is generic and can be externally programmed and influenced in a very broad way. Thus, the compiler can run different mapping programs without any need to change the compiler itself. As the configuration of IDL*flex* is very easily changed, IDL*flex* is also very appropriate if ORB developers have to experiment with different generated code. The driving force behind IDL*flex* has been the *AspectIX* middleware project. *AspectIX* extends CORBA by generic quality-of-service support and the implementation of *AspectIX* needs completely different stub objects than standard CORBA. So, IDL*flex* helped to design the stub objects for *AspectIX* and became the *AspectIX* IDL compiler.

This paper is organized as follows: Section 2 introduces the implementation of IDL*flex*. In Section 3, we outline the advantages of IDL*flex* demonstrated by examples of the IDL-to-Java language mapping. Section 4 compares IDL*flex* to related work. Section 5 will give our conclusions. Finally, in an appendix we will briefly define IDL*flex*'s mapping and configuration language.

2 Implementation

The implementation of IDL*flex* is divided into three distinct parts as outlined in Fig. 1. The two parts on the left side are responsible for reading in the IDL description of one or more interfaces that shall be processed by IDL*flex*. The third part, on the right side, will process the interfaces and generate code. IDL*flex* is entirely written in Java and thus portable to every platform that provides a Java Virtual Machine.

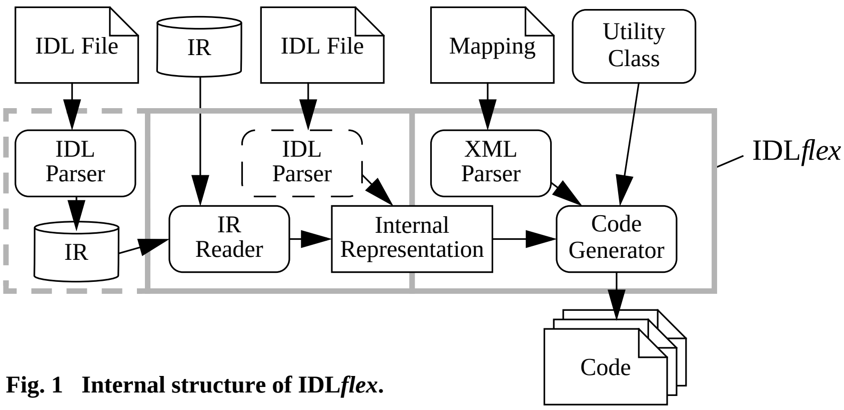

Fig. 1 Internal structure of IDL*flex*.

2.1 Reading IDL

As a first step, the IDL descriptions that shall be processed by IDL*flex* have to be read into the compiler. An internal data representation of an IDL interface has to be generated. This internal representation will finally serve as the basis for code generation.

In our first version of IDL*flex*, we retrieve IDL descriptions from an interface repository. The interface repository (IR) stores interface descriptions in form of CORBA objects. The types and the behavior of those objects are defined by the CORBA standard. Furthermore, an interface repository is necessary for other CORBA mechanisms like type checking and the Dynamic Invocation Interface (DII). Thus, an interface repository is usually part of a CORBA implementation. A reader component inside of IDL*flex* can read an existing interface repository, retrieve the IDL information, and convert it into the internal representation needed for code generation. So, the IDL description of an interface has first to be loaded into an interface repository and then be processed by IDL*flex*.

As this is cumbersome in certain cases, we incorporated an existing interface repository including a corresponding IDL parser into IDL*flex* (see the left hand and dashed part in Fig. 1). The parser can read files containing IDL descriptions and feed them into the interface repository. This is finally read by the IR-reader component and then processed. In our implementation we used the interface repository and IDL parser of the open source CORBA implementation *JavaORB* 2.2[1].

Reading from an interface repository has the disadvantage that it can only process IDL descriptions that are usually stored there. For example, the interfaces of CORBA pseudo objects are not stored in an interface repository. Pseudo objects are used within the CORBA standard to make internal mechanisms and services appear as CORBA objects. The indirect reading of IDL descriptions via an internal repository also wastes resources. Thus, the next version of IDL*flex* will have an additional internal IDL parser that can directly read IDL files and convert them into the internal representation (see the dashed component in the middle part of Fig. 1).

1. This ORB is no longer supported by its developers. However the *JavaORB* technology migrated to the *OpenORB*, also an open source project.

The internal data representation is derived from the representation inside of an interface repository. However, certain things have been simplified for the later processing. As an alternative we could have decided on directly generating code from the objects inside of an interface repository. We ruled this out due to the complex interface to the repository objects and for efficiency reasons.

In the internal IDL representation, every IDL item is represented by a Java object. So, IDL modules, interfaces, operations, attributes, exceptions, etc. are represented each by a particular object. This object contains all the necessary information like kind of IDL item, name, type, etc. A simple IDL interface of a `sumserver` with one method named `add()` is shown in Fig. 2.

```
interface sumserver
{
   void add( in long s1, in long s2, out long res );
}
```

Fig. 2 A simple IDL interface.

The interface named `sumserver` is internally represented by a Java object of type `InterfaceObj`. Such an object contains references to other objects representing either members of the interface and inherited interfaces. In this example, just one reference is contained, a reference to an object of type `OperationObj` representing method `add()`. The operation object in turn contains references to objects representing parameters. Parameter objects contain references to objects representing their types. The complete representation of the example interface is shown in Fig. 3. The internal objects also contain attributes to distinguish abstract and local interfaces as well as `in`, `inout` and `out` parameters, etc.

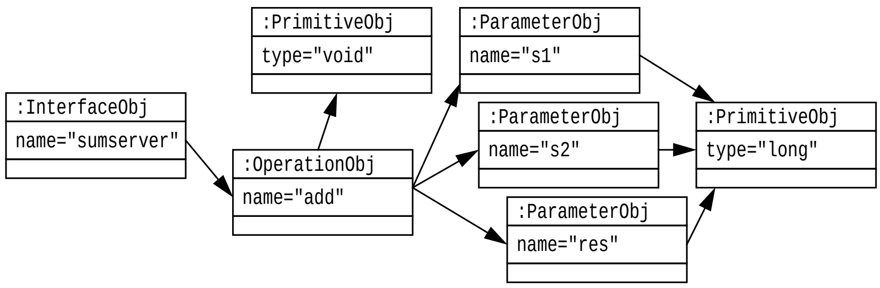

Fig. 3 The internal representation of interface `sumserver`**.**

2.2 Code Generation

As the code generation shall be generic there is no built-in code for code generation. Instead IDL*flex* provides an interpreter for a simple programming language. In this language the specific code generation depending on the IDL description fed into IDL*flex* can be programmed. This program is called a mapping.

The programming language to describe mappings is XML-based. The language defines statements as XML tags whereas ordinary text is usually directly written to the output code. There are three types of statements that are supported: output control, control flow and access to the internal data representation.

The output control allows to specify the name of an output file that will get the generated code. Usually a CORBA language mapping specification defines what files have to be generated. With the output control, the mapping program can generate code for and switch between multiple files. For code that has to be inserted multiple times internal string variables can be used. The output is then kept in these variables and can be inserted at arbitrary places in the mapping program.

Control flow statements allow to group mapping code into so-called components that can be invoked like sub-routines. Thus, common mapping code can be extracted and put into an own component. This serves for modularity and code reuse. Parameters can be passed to components by filling the above mentioned string variables. Furthermore, conditions can be tested at run-time and according to such a condition multiple control flows can be chosen in the mapping program. Often the same mapping has to be done for a set of objects of the internal representation (e.g., for all members of an interface). A special statement allows to iterate over such sets.

Another statement allows to retrieve information from an object of the internal representation. At each time the flow of control inside of the mapping program is implicitly associated with one of the objects of the data representation. This is similar to the implicit *this* pointer in object-oriented languages. The mapping program starts with the root object of the representation, usually representing an interface or an IDL module containing other modules and interfaces. With the access statement (XML tag `GET`), it is possible to retrieve the name, type and other information from the representation object. With different values for an XML attribute of tag `GET` the actual information is named. Access to other objects of the internal representation is possible by explicit de-referencing which is allowed for certain cases. Implicit de-referencing takes place when the iteration operator is used. Inside of the loop, the implicit representation object is always switched to another member of the iteration set, e.g., to the next member of an interface or to the next base interface.

As some mapping mechanisms need complex algorithms it is not reasonable to program these algorithms using the statements of our mapping language. Therefore, those algorithms can be provided in form of a utility class. Some defined methods of the utility class can intercept attribute values of the `GET` tag and thus replace this XML tag by arbitrary strings that are put instead into the output code. As an example, the utility class for the Java language mapping will provide the correct file name for an IDL interface as prescribed by the Java language mapping. Another example is the conversion of IDL names to language names. This procedure has to especially handle language keywords and other reserved names.

The utility class is usually used to encapsulate language-mapping–specific algorithms. As those algorithms are defined in the mapping standard there is hardly any need to change the utility class if it is already available for a certain language. The utility class is dynamically bound to IDL*flex*. The class name is defined in the mapping program.

The mapping program that is to be executed by IDL*flex* can be configured as a command-line parameter. The mapping program is parsed by an XML parser and read into memory as a DOM element tree. IDL*flex* uses the *Apache Xerxes* XML library. The mapping-program interpreter operates on the DOM elements. With the execution of the mapping statements in the mapping program, output code is generated. After completion of the mapping program IDL*flex* terminates.

2.3 Example

As a brief example of the code generation, Fig. 4 shows three components of the mapping program for the IDL-to-Java language mapping. The first component, named `OperationCompiler` is called whenever an operation has to be mapped to a Java method. This is necessary either for stub objects and for skeletons. We assume that the output file is already selected by the calling component. Ordinary text as the word `public` is then directly written to the selected output file. The next `GET` tag de-references the return type from the implicit `OperationObj`-Object and retrieves the Java type declaration. The following `GET` tag retrieves the name of the implicit operation object. Then a loop is executed iterating over all parameter objects stored in the implicit operation object. Within the loop each parameter object once becomes the implicit representation object for the loop body. Inside of the loop another component for the parameter type definition is executed followed by the name of the parameter. Finally, all defined exceptions are processed in another loop by calling yet another component.

```
<!-- Processing an operation -->
<COMPONENT NAME="OperationCompiler">
   public <GET OBJ="RETURN" T="TYPE:decl"/> <GET T="TYPE:name"/>(
      <ITERATE SPEC="PARAM">
         <CALL COMP="ParametersDefCompiler"/> <GET T="TYPE:name"/>
      </ITERATE> )
      <ITERATE SPEC="EXCEPT"><CALL COMP="ThrowsDefCompiler"/> </ITERATE>
</COMPONENT>

<!-- Processing the parameters -->
<COMPONENT NAME="ParametersDefCompiler">
   <IF ATTR="!ISFIRST">, </IF>
   <IF ATTR="OBJ:inarg"><GET OBJ="BASE" T="TYPE:decl"/>
   <ELSE/><GET OBJ="BASE" T="TYPE:holder"/>
   </IF>
</COMPONENT>

<!-- Processing thrown exceptions -->
<COMPONENT NAME="ThrowsDefCompiler">
   <IF ATTR="ISFIRST">throws <ELSE/>, </IF><GET T="TYPE:decl"/>
</COMPONENT>
```

Fig. 4 Mapping components for operations in the IDL-to-Java mapping.

The definition of the `ParametersDefCompiler` component shows two conditional statements of the mapping language. With `ISFIRST` a conditional statement can retrieve whether it belongs to the first run of a loop or not. The second statement checks whether the implicit parameter object is an `in` parameter or not.

Executing the component `OperationCompiler` on the internal operation object named `add` of our `sumserver` interface (see Fig. 2 and 3) leads to the output shown in Fig. 5. For the `out` parameter `res` a holder type has to be inserted. The IDL type `long` is mapped to Java's type `int`. This mapping happens in the utility class which processes the request `TYPE:decl`.

The output text is post-processed by a simple formatter that prunes unnecessary white space characters which are introduced as delimiters for mapping statements. This makes the output code more readable.

```
public void add( int s1, int s2, org.omg.CORBA.IntHolder res )
```

Fig. 5 Result of the `OperationCompiler` **for operation** `add`**.**

3 Advantages

The advantage of IDL*flex* lies in its flexibility. Existing mapping programs can be adapted to the needs of a certain CORBA implementation. One example is the generation of client-side stub objects. Stub objects have to look like the actual object but forward invocation request to the remote object on another host. Therefore, stub objects have to use communication mechanisms to contact the remote host, or more precisely, the remote object adaptor responsible for the remote object. These communication mechanisms are often vendor-specific; every CORBA implementation may use different, sometimes even incompatible, communication mechanisms. In this case, for each CORBA implementation just another, slightly different mapping program has to be provided.

In the Java mapping, communication mechanisms are entirely encapsulated outside of the stub object, so that stub objects of all vendors could look similar. However, the Java language mapping allows two different implementations of stub objects: one based on the dynamic invocation interface (DII) and another one based on output streams. With IDL*flex* for each implementation there could be just another mapping program. Even better, a mapping program for IDL*flex* can contain both variants and the variant is chosen by a command line option at execution time of IDL*flex*. Fig. 6 shows the mapping code for the selection. A command-line parameter named `-DDIIStub` selects the mapping via DII-based communication. Otherwise the stream-based mapping is selected. The two mapping variants are each defined in an own mapping component.

```
<IF ATTR="DEF:DIIStub">
   <CALL COMP="DIIStubCompiler"/>
<ELSE/>
   <CALL COMP="StreamStubCompiler"/>
</IF>
```

Fig. 6 Selection of two mapping variants.

IDL*flex* has also been used for creating stubs, and object references respectively, of the *AspectIX* middleware platform [3, 2]. *AspectIX* is a CORBA-based system, but integrates general quality-of-service requirements into distributed objects. To that end, client-side stub objects can carry object-specific code in *AspectIX*, because quality-of-service implementation often need not only code at the server- or object-side but also at the client side, e.g., for maintaining a communication link based on special protocols.

The client-side part of an *AspectIX* object is called a fragment. It consists of a generic part that can easily be generated by IDL*flex* by adapting the stub mapping of the standard Java mapping. The generic part forwards invocations to an object-specific part, called the fragment implementation. For that implementation, IDL*flex* can generate skeletons, similar to servant skeletons in plain CORBA. For the *AspectIX* project, IDL*flex* proved to be an efficient platform for experimenting with different code generation. Changes in the output code were very easy and the turn-around time for testing the newly designed output code, the new mapping program respectively, was very small.

4 Related Work

The only comparable project is *Flick* (Flexible IDL Compiler Kit) [1]. Flick is very flexible as it can deal with CORBA IDL, Sun RPC, Mach Messages and other interface description languages. Multiple back-end modules of the compiler kit allow to generate code for different languages and mappings. The back-end modules are explicitly programmed for a specific mapping. However, there is one back-end that allows to include a SGML-based mapping description into the code generation.

While Flick is a very complex system which urges the programmer to write C++ code for complex mapping changes, IDL*flex* has a lean design and a powerful but still simple mapping language allowing the description of CORBA IDL language mappings.

5 Conclusion

IDL*flex* is a flexible IDL compiler as it allows to specify the code generation for a specific language mapping including vendor-specific code by programming the output in an XML-based mapping language. Language-specific mapping constructs can be encapsulated in a utility class. However, we anticipate that the utility class is hardly ever changed for a specific language mapping, e.g., for Java. So the same IDL compiler can be used with different mapping programs for producing code for different languages and CORBA implementations.

IDL*flex* has been used to generate special object references of the *AspectIX* ORB architecture. Especially for experimenting with different *AspectIX* implementations, IDL*flex* was very helpful, because just the mapping file had to be changed and there was no need to adapt the IDL compiler.

6 Availability

The IDL*flex* software is available to the public and can be downloaded under the URL `http://www.aspectix.org/IDLflex/`. IDL*flex* has been put under the terms of the GNU Public License (GPL). The current version, as of May 2001, is beta software. It lacks some documentation and needs an update to the newest XML processing interfaces and libraries. Currently we have a mapping program for the CORBA 2.3 Java language mapping. This mapping does not yet consider value boxes.

References

[1] E. Eide, J. L. Simister, T. Stack, J. Lepreau: Flexible IDL compilation for complex communication patterns. *Scientific Progr.*, **7**(3, 4), 1999. pp. 275-287.

[2] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, M. Steckermeier: AspectIX: a quality-aware, object-based middleware architecture. *Proc. of the DAIS 2001 Conf.*, to appear 2001.

[3] F. J. Hauck, E. Meier, U. Becker, M. Geier, U. Rastofer, M. Steckermeier: A middleware architecture for scalable, QoS-aware and self-organizing global services. *Proc. of the USM '00 Conf.* (Munich, Sept. 12–14, 2000), LNCS 1890, Springer, 2000.

[4] Object Mgmt. Group, OMG: *Java language to IDL mapping specification*. OMG Doc. formal/99-07-59, June 1999.

[5] Object Mgmt. Group, OMG: *The Common Object Request Broker*, architecture and specification. Ver. 2.4.2, OMG Doc. formal/01-02-33. Framingham, Mass., Feb. 2001.

Appendix: The IDL*flex* mapping language

This section describes the XML elements of the mapping language, their attributes, and their semantics.

`<IDLflex>` Root element of the XML-based language. Attributes: `ROOT` describes the component that processes the root of the internal representation objects. `UTILITY` specifies the class name for the mapping-specific utility class. `WRITER` specifies the class name for a class post-processing the output.

`<COMPONENT>` Element that specifies a mapping component. Attributes: `NAME` specifies the name of the component.

`<CALL>` Element executes the mapping code of a referenced component. Attributes: `OBJ` references an internal representation object that is used within the component code; if omitted the internal object is not re-assigned. For allowed values for the `OBJ` tag see the `GET` tag. `NAME` is the name of the executed component.

`<ITERATE>` This elements iterates over a list of internal objects. Attributes: `SPEC` denotes the kind of objects used within the loop. The object set depends on the current internal object. Examples are: `MEMBER` for the members of modules, interfaces and enums. `ALLMEMBERS` for the members of interface including inherited members. `ELEMENTS` for the members of structs, unions and exceptions. `UNIQUE` for a list of all unique union members. `BASE` for the immediate base interfaces of interfaces; `ALLBASE` for all inherited interfaces. `EXCEPT` for the list of exceptions, `PARAM` for the list of parameters of an operation.

`<IF>` Starts a conditional control flow. Attributes: `TYPE` checks the type of the current representation object. `OBJ` selects another representation object for consideration. For allowed values see the `GET` tag. `ATTR` checks a condition defined in the utility class. Allowed conditions are:

`DEF:`name	Checks for a command line argument `-D` name
`ISFIRST`	True if the first representation object of an `ITERATE` element
`HAVE:RETURN`	True if an operation object has a non-void return type
`HAVE:`name	True if the named element list (allowed name see attribute `SPEC` of `ITERATE`) is nonempty.
`readonly`	True if attribute object is read-only
`abstractif`	True if interface is abstract
`inarg`, `outarg`, `inoutarg`	Exactly one is true for parameter objects
`wide`, `bound`	True if string object is wide or bound

`unique`	True if union member is unique
`isdefault`	True if union member is default
`needsdef`	True if implicit default label necessary for unions

Conditions may be inverted with the `!`-operator and concatenated by the or-operator `|`.

`<ELSE>` Element denoting the ending of the if and the start of an else part.

`<COND>` Instead of an attribute of the `IF`-tag, an `IF`-tag can be followed by multiple `COND`-tags describing alternative conditions. Attributes are the same as of the `IF`-tag.

`<ERROR>` Stops processing and prints the error message encapsulated by this tag.

`<FILE>` Redirects the following output text to a specific file. Attributes: `SPEC` selects a file. Allowed names are defined by the mapping-specific utility class. For the Java mapping the same names as for `FILE`:name of attribute `T` of tag `GET` are allowed.

`<SBOX>` Redirects the following output test to an internal variable. Attributes: `NAME` of the `SBOX` variable.

`<GET>` This is the generic tag to access external information, e.g., from the internal representation objects. Attributes: `T` denotes what is to be retrieved. `OBJ` denotes the internal representation object to be used for retrieval; if omitted the current object is used. `OBJ` may have the following content:

`BASE`	The type object of an array, attribute, constant, parameter, sequence, union or struct member
`CONTAINER`	The object the current object is contained in as a member
`DISCR`	The type object of the union discriminator
`RDISCR`	Same as `DISCR` but typedefs are resolved
`RESOLVE`	Referred internal object for typedefs
`RETURN`	The return type of an operation object
`SUPER`	The internal object for the enum of an enum member; the union or struct object of a union or struct member

Attribute `T` may have the following values (partially defined by the Java utility class):

`CONSTVAL`	Java code for the value of a constant object	
`COUNT:`name	Number of list elements of object lists (see `ITERATE` tag for allowed sets); numbers count from 0	
`DEF:`name	Retrieves content of command line option `-D`	
`DISCR:num/sym`	Numeric or symbolic reference to union discriminator of a union member	
`FILE:`name	Retrieves the file name for `TYPE`:name class of the Java mapping.	
`IDL:`name	Retrieves information from an internal object. `name` could be:	
	`bound`	Size of an array, sequence or string
	`defdiscr`	Smallest discriminator value of union
	`digits/scale`	Digits and scale of a fixed width number
	`discr`	Label number of union discriminator value
	`length`	Size of an array or sequence
	`name/fullname`	Name and qualified name of an internal object
	`value`	Value of an enum member
`INDEX`	Current number of `ITERATE` loop runs (counted from 0)	
`PACKAGEDEF`	Java package declaration for current object	
`PACKAGENAME`	Name of surrounding Java package	
`SBOX:`name	Retrieves the content of an `SBOX` variable	
`TYPE:`name	Conversion for the Java language mapping. Allowed names:	
	`decl`	Java-mapped type declaration of internal object
	`helper/holder`	Name of mapped helper/holder class
	`name`	Java-mapped name of internal object
	`operationif`	Interface name of operation interface
	`stub/skeleton/tie`	Stub/skeleton/tie Java class name
	`signatureif`	Interface name of signature interface