Importing IDL into PICML

And IDL to PICML Mapping Reference

Jeff Parsons

Introduction

The Platform Independent Component Modeling Language (PICML) is the centerpiece of the Component Synthesis with Modeling Integrated Computing (CoSMIC) tool suite. A wealth of information is available at the CoSMIC website:

http://www.dre.vanderbilt.edu/cosmic/

PICML is a large modeling language with several facets - one such facet includes elements that represent components, interfaces and related data types that are the building blocks of component systems. These are precisely the kind of elements we find in CORBA IDL files. This document describes a tool that enables developers to jump-start a PICML project by mapping the interface definitions in a set of existing IDL files to corresponding elements of PICML. For a detailed description of each of these PICML elements, along with their semantics, motivation and intended use, see the documentation in

COSMIC_ROOT/PIM/PICML/docs

idl_to_picml is an executable program that imports IDL into PICML.
The idl_to_picml executable translates up to 1024 IDL files at a time into a single XML document that may be imported into the Generic Modeling Environment (GME) as a model in PICML. This model, which contains only the information present in the set of IDL files passed to idl_to_picml on the command line, may then be developed further in PICML to model a complete component system. The idl_to_picml executable iterates over the list of IDL files, using the parsing engine of the TAO IDL Compiler to create an abstract representation of each IDL file. This Abstract Syntax Tree (AST) is then traversed by a custom back end plugin which translates the AST and generates XML which conforms to the DTD describing XML files which can be imported/exported by all GME modeling languages. For more information about GME, see the GME Manual and User Guide, and The Generic Modeling Environment, both of which are PDF files included with the GME distribution. GME was developed at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University, and information about how to obtain GME may be found at the ISIS website:

http://www.isis.vanderbilt.edu/Projects/gme/

The TAO IDL Compiler is included with the ACE+TAO+CIAO distribution. Download and other information can be found at

http://www.cs.wustl.edu/~schmidt/

The back end of `idl_to_picml` uses the Xerces library to construct a DOM tree from the AST, then traverses the DOM tree to generate the XML document. For documentation and download information about Xerces, see

http://xml.apache.org/xerces-c/

Compiling

Although GME may be run only on Windows platforms, `idl_to_picml` is portable to any platform supported by both ACE+TAO+CIAO and Xerces, including but not limited to Windwos and most flavors of Unix/Linux. A makefile or Windows project can be generated using Makefile Project Creator (MPC) developed by Object Computing Incorporated (OCI). MPC is included in the ACE+TAO+CIAO distribution, and documentation can be found in the ACE_wrappers/MPC directory. More information about MPC can be found at the OCI website:

http://ociweb.com

An .mpc file for `idl_to_picml` is included in the same directory as this document. This file assumes that some environment variables have been set:

- ACE_ROOT - ACE_wrappers (root directory of the ACE+TAO+CIAO distribution
- TAO_ROOT - ACE_wrappers/TAO
- XERCESCROOT - path to the Xerces library on your machine

As mentioned above, `idl_to_picml` uses the TAO IDL Compiler parsing engine, and so depends on the that compiler's front end library. The .mpc file and all source files for the TAO IDL Compiler can be found in TAO_ROOT/TAO_IDL. This front end library in turn depends on the ACE library. The .mpc file and source files for the ACE library can be found in ACE_ROOT/ace. There are
many ways that all these things may be built and many ways to combine the builds with other projects, but a typical scenario in a fresh workspace might look like this:

```
cd $(ACE_ROOT)/ace
mwc.pl
make -f GNUmakefile.ACE
cd ../TAO/TAO_IDL
mwc.pl
make -f GNUmakefile.TAO_IDL_FE
cd $(COSMIC_ROOT)/PIM/PICML/interpreters/IDLImporter
mwc.pl
make
```

The `idl_to_picml` executable is created in COSMIC_ROOT/bin, so you may want to put that directory in your path for convenience.

Command Line Options

Since it uses the TAO IDL Compiler front end, `idl_to_picml` supports all command line options pertaining to that front end. Typing

```
idl_to_picml –u
```

will output a usage message with all command line options listed along with a brief description of each. For more information about the IDL compiler options, see `TAO_ROOT/docs/compiler.html`.

There are two command line options specific to `idl_to_picml`:

- `-x <filename>` Sets the name of the output XML file which, for GME import, has the extension .xme. Do not include the extension in the file name – it is added automatically. If this option is not used, the default file name will be "PICML_default_xme_file.xme".
- `-i <filename>` Sets the name of the (optional) input XML file exported from a PICML model. Unlike the `–x` option above, the extension must be included in `filename` for this option. If this option is used, IDL files imported to create the original PICML model can be reprocessed and the resulting XML file, containing the corresponding additions, removals, and changes, can be imported into GME. See **Re-importing Modified IDL Files** below for more information about this feature.
- `-m <directory path>` Specifies the path to the file `mga.dtd`, used to parse the input XML file if the `–i` option is used. The default value of this path is set to GME_ROOT/bin. The option needs to be used only if `mga.dtd` is in some other directory.
- `-o <directory>` Sets the (relative or absolute) path to the directory where the XML file will be generated. The default directory is the one from which `idl_to_picml` is executed.
- `-r <directory>` Can be passed instead of a list of IDL files. If this option is used, `idl_to_picml` will process all IDL files found in `directory` and all of its subdirectories.

Imported Model Features

Each imported PICML model will contain a folder called PredefinedTypes, containing all the PICML model elements representing basic types. These types cannot be created in a model, since in a sense they already exist, but they may be referred to by other model elements, so they are placed in this folder in every PICML model. PICML model elements corresponding to IDL declarations that refer to basic types will refer to elements in the PredefinedTypes folder according to the IDL to PICML mapping for basic types, described in the mapping section below.

All PICML model elements corresponding to IDL declarations appear in a folder called InterfaceDefinitions. The user may rename this folder if desired, and any subsequent new model elements added directly in the graphical modeling environment may go into the same folder, or into any number of other (possibly also renamed) InterfaceDefinitions folders that may be inserted in the model.

If the `-i` option is not used with `idl_to_picml` (in other words, if a new PICML model is being created from IDL files), the imported PICML model will also contain the following folders:

- ComponentBuild
- ComponentImplementations
- ComponentPackages
- ComponentTypes
- DeploymentPlans
- ImplementationArtifacts
- PackageConfigurations
- Targets
- TopLevelPackages

which must be filled in with the appropriate package, assembly, implementation and deployment elements in order to have a complete and usable PICML model. Most of these folder will be empty, but ComponentTypes, ComponentImplementations, and ImplementationArtifacts will have default elements added for each IDL component in the processed IDL files. These default elements can be renamed and/or modified by the modeler.

Re-importing Modified IDL Files

As mentioned above, the `-i <filename>` option will input the XML file `filename`, representing a PICML model, into the IDL importer and compare it with a set of modified IDL files, passed in either individually on the command line, or by use of the `-r <directory>` option. If the XML file was exported from a PICML model, all changes made to the model will be after the original import from IDL files will be preserved (except when relevant IDL declarations have been removed from the modified files, as described below). The IDL importer will output (to `stdout`) an informative diagnostic for each addition, removal, or substitution it makes. The following subsections provide details on the behavior of this feature, broken down by the three broad categories of possible modifications.

Additions to IDL Files

This is the simplest case, in which the corresponding additions will be made to the InterfaceDefinitions folder, along with default additions to the ComponentTypes, ComponentImplementations, and ImplementationArtifacts folders, as described in the previous section, for each added IDL component declaration (if any).

Removals from IDL Files

- Within the InterfaceDefinitions folder, model elements corresponding to the removed IDL declarations will be removed.

- Throughout the entire PICML model, any elements that reference, either directly or indirectly, elements removed as described in the bullet above, will also be removed.
- Throughout the entire PICML model, any GME Connections that have an endpoint at a PICML model element removed as described in the previous bullets will also be removed.

It's possible that some of the actions described above may leave the PICML model in an inconsistent state, that is, a check of all constraints will reveal violations. This behavior exists intentionally, motivated by the fact that it is often impossible to determine the modeler's intention when modifications to IDL file are re-imported. The output of the IDL importer in these cases is determined by what will most likely require the least amount of work for the modeler. Here are a couple of examples:

- A GME Connection is removed because one of its endpoints was removed, and the other endpoint requires this type of Connection, which is now missing. On one hand, suppose the modeler intends to replace the Connection, between the remaining endpoint and a new endpoint (which may have been added automatically as a result of an IDL addition) – a matter of 3 mouse clicks in GME. If the IDL importer had instead removed both endpoints of the Connection, the modeler could be faced with replacing many more PICML elements, since the removal of the second endpoint may itself have caused the removal of other PICML elements via a "domino effect". On the other hand, if the removed Connection is not to be replaced, it is a much simpler matter to remove the remaining endpoint, and any other PICML elements that reference or connect to it, than it would be to replace all these elements by hand if they were removed automatically.

- A Component instance is removed because the corresponding Component type was removed from InterfaceDefinitions. In a correct PICML model, there must also be a MonolithicImplementation whose name matches either the Component instance name or type name. Again, suppose the modeler's intention is to replace the removed Component with another type added to the modified IDL files. One or more instances of the new type will have to be created, which would have to be done in any case. If these new instance names are the same as the removed ones, nothing further needs to be done. In the worst case, the modeler will have to rename some MonolithicImplementations to match either the new instance or type names. On the other hand, if the IDL importer had automatically removed the MonolithicImplementation, all its elements would have to be recreated by hand.

In general, with a given set of modifications to IDL files, given a choice between requiring the modeler to remove or rename "dangling" elements to satisfy constraint checking in one scenario, or recreating these same elements by hand in another scenario, we have chosen the former alternative since it requires fewer steps for the modeler. Using the GME constraint checker is a convenient way to make sure no loose ends are overlooked.

Substitutions in IDL Files

- **To Identifiers** An identifier, in IDL, can be given to a type declaration, a member, or an operation parameter. In all of these cases, it is the identifier that determines uniqueness,

since in a given scope there may be any number of repeated IDL types as long as the identifiers are unique. For this reason, a change in an IDL identifier is treated by the IDL importer as the removal of the construct named by the original identifier and the addition of a new one of the same type. While it may be possible for the IDL importer to "drill down" through arbitrarily many levels of scope to determine that the type associated with some new identifier is identical to one associated with an identifier in the original IDL file, doing so might require a huge amount of time and resources, and in any case may not reflect the intention of the modeler.

- **To Type References**

    - **With Identifier** This case includes IDL members, attributes, parameters, constants, typedefs, ports and union discriminators, where the "type" associated with the identifier is actually a reference to a previously declared type or a basic predefined type. These use cases are all represented in a PICML model by a GME Reference, and the "referred" parts of these References (all in the InterfaceDefinitions folder) will be changed automatically by the IDL importer. In some cases, constraint violations may be introduced, for example if there is an existing connection between Component ports which become type mismatched. Again, the modeler may conveniently use the constraint checker to quickly locate places in the model that need to be updated. While it is possible for the IDL importer to detect port connections that have become type mismatched, we feel it's better to leave the connection than to remove it, since the latter course will lead to changed behavior in a Component assembly and, without constraint violation messages, may be unnoticed by the modeler.

    - **Without Identifier** This case includes operation exceptions, attribute exceptions, inheritance lists and supported interface lists. Since there is no identifier to serve as a point of commonality, the IDL importer treats these use cases as removal/addition pairs of changes.

- **To Directives** This case includes the `#pragma` directives plus the IDL keywords introduced to replace some of these directives:
    - `#pragma prefix`
    - `#pragma ID`
    - `#pragma version`
    - `typeprefix`
    - `typeid`

  The values of these things are represented in a PICML model by GME Attributes, and the IDL importer will automatically change these values. These GME Attributes are always present in any PICML model element that may have them, the default value for each being the empty string. Note that these default values differ from those in C++ or Java code generated from IDL, where the default version is "1.0" and the default type id is the repository id. Changes to these values made by the IDL importer will not affect anything else in a PICML model.

Relocations in IDL Files

IDL declarations that are moved within the same scope produce no changes in a PICML model when re-imported, since GME does not capture any ordering information within a scope. If an IDL declaration is moved to a different scope, however, it will be handled as a removal/addition pair of changes. Relocation to a different opening of the same IDL module, while considered movement within the same scope by IDL, will result in a removal/addition pair of changes in the PICML model, since each opening of an IDL module is mapped to its own PICML Package model element.

Special Cases

- **Valuetype/Eventtype Member Visibility** Public visibility is the default in PICML, so public members of ValueObject and Event in PICML have nothing special about them. Private members, however, are represented by a PrivateFlag model element, which is a GME Atom, and a GME Connection called MakeMemberPrivate to the member in question. Changing a member from public to private (or vice versa) will result in the addition (or removal) of the corresponding Atom and Connection

- **Union Case Label Value** This construct is treated the same as an identifier by the IDL importer, and thus result in a removal/addition pair of changes when the case label value is changed. Addition or removal of a case label in IDL, even without corresponding addition or removal of the associated union member (IDL allows union members to have multiple case labels), will result in the addition or removal of a PICML Label element (GME Atom) and an associated LabelConnection. The Label's name in the PICML model will be the same as the case label value in IDL.

Usage Notes

- Command line arguments that require a subsequent argument will parse correctly whether or not a space is left between the command line option and the argument.

- Command line arguments and file names may appear in any order on the command line. Any string not preceded by a dash and not following a dashed option that requires an argument will be treated as a file name.

- No XML is generated for included IDL files. This means that the closure set of all included IDL files must be passed on the command line. There are 3 exceptions to these inclusions:

    - orb.idl - a special case, even in the CORBA world, containing declarations already implemented in the ORB, hence even IDL compilers don't generate the contents of this file. The only thing in this file that can be legally referenced in application IDL files is CORBA::TypeCode, which is modeled as a predefined type called TypeEncoding in PICML.

- Components.idl - the CCM counterpart of orb.idl and the same explanation applies. There is nothing in this file that can be legally referenced in application IDL.

- files included by the above - Components.idl is a spec-defined IDL file that in CIAO is just a shell for several other IDL files, none of which contain anything that may be legally referenced in application IDL. Both files above also contain .pidl (pseudo IDL) files. These are peculiar to TAO. The unique extension is used to designate IDL files found in TAO_ROOT/tao that are used to generate parts of the ORB implementation. With the exception of the spec-defined sequences of basic types, there is nothing in these files that may be legally referenced in application IDL. For a detailed discussion of how sequences of basic types are handled by `idl_to_picml`, see the next section in this document.

- Of course, it is always possible that none of the contents of a given included IDL file will be referenced by any other IDL file in the project, and in such a case, it can be left out of the list of files passed to `idl_to_picml` and the resulting .xme file can be imported without error, but this use case begs the question of why that particular IDL file was included in the existing project in the first place.

- A set of IDL files may be placed on the `idl_to_picml` command line in any order. The order of the IDL files will dictate the order of elements appearing in the generated XML file, but not the order in which the corresponding elements appear in the Model Browser of GME.

- It is the responsibility of the user to make sure that all include paths other than the current directory, even for included IDL files not required on the command line, be passed to t `idl_to_picml` as a -I option.

- If the directory passed with a -o option does not exist, it will not be created, but an error message output instead. This behavior may change in future versions of the TAO IDL compiler and `idl_to_picml`.

IDL Issues

- Some IDL syntax deprecated by the OMG is not supported by `idl_to_picml`:
  - Anonymous sequences and arrays. It is preferable to typedef arrays and sequences before using them as member types, attribute types or operation parameter types.

  - Members that are also declarations, for example:

```
struct foo
{
   string str_member;


   enum bar
```

```
    {
       ONE,
       TWO,
       THREE
    } e_member;
};
```

- Some IDL features are not supported by the TAO IDL Compiler, and so cannot be supported by `idl_to_picml`:

  - Boxed valuetypes
  - Custom valuetypes
  - Truncatable valuetype inheritance
  - Fixed point types
  - Union discriminators or type char and wchar

  - The `import` keyword
  - The `native` keyword
  - The `context` keyword

- The CORBA specification defines several sequences of basic types - `octet, boolean, string, any`, etc. - that must therefore be included in ORB source files. In TAO, these types are generated from IDL contained in files with the extension `.pidl`, contained in the same directories as ORB source code files. These basic type sequences are also exceptional in that they are the only types declared in spec-defined IDL files that may be legally referenced by application IDL. This special case creates a problem for PICML and for `idl_to_picml`. They are too specific to CORBA to be included in the platform-independent PICML modeling language, but they cannot just be ignored, so a suitable mapping must be found from CORBA IDL to PICML for these types, as well as a suitable place in a PICML model to declare them.

- IDL sequences and arrays are mapped to the PICML model element called Collection. A Collection is a GME Reference, and it refers to the sequence element type. For each basic type sequence actually referenced in application IDL (which may be a subset of the ones seen through inclusion) `idl_to_picml` generates a Collection at global scope in the first IDL file processed. Each Collection so generated refers to the appropriate PICML model element in the PredefinedTypes folder. References to these sequences in IDL are mapped to some PICML model element type that specializes a GME Reference. This mapping process is reversed when generating IDL from PICML models (see PICML documentation for IDL generation).

IDL to PICML Mapping Reference

Each IDL file passed on the command line to `idl_to_picml` maps to a PICML model element called File. A File is the only PICML model element that may be inserted into an InterfaceDefinitions folder.

This section contains the specific mappings from IDL types and kinds to PICML model elements. The IDL pragma directive, include directive, basic types and other odds and ends are listed first, followed by the remaining IDL keywords in alphabetical order.

Pragma Directive

These are mapped in PICML to GME Attributes appearing in the PICML model elements corresponding to the IDL types that may legally have the particular pragma directive in question. Note that `#pragma prefix` and `#pragma ID` now overlap with the IDL keywords `typeprefix` and `typeid`, respectively. Either way they appear in an IDL file, they are mapped to the same thing in PICML.

- `prefix` – PrefixTag
- `ID` – SpecifyIdTag
- `version` – VersionTag

Include Directive

If an IDL file includes another IDL file, it is mapped to PICML as a model element called FileRef, which is a reference to the File model element corresponding to the included file. This included IDL file must be passed on the command line to idl_to_picml (subject to the exceptions described in Usage Notes) in the same set of IFL files as the including file.

Basic Types

PICML has model elements representing basic types which are found only in a GME folder of type PredefinedTypes and cannot be added from the Part Browser. References to basic types in IDL files are mapped to the appropriate GME Reference type in PICML, with the appropriate element in the PredefinedTypes folder as the model element referred to. Since it is platform-independent, PICML does not have an analogue for each and every IDL type, including the basic types. The mapping for IDL basic types is:

- `any` – GenericValue
- `boolean` – Boolean
- `char` – Byte
- `double` - RealNumber
- `float` - RealNumber
- `long` – LongInteger
- `long double` - RealNumber
- `long long` - LongInteger

- `Object` – GenericObject
- `octet` – Byte
- `short` – ShortInteger
- `string` – String
- `TCKind` – TypeKind
- `TypeCode` – TypeEncoding
- `unsigned long` – LongInteger
- `unsigned long long` – LongInteger
- `unsigned short` - ShortInteger
- `ValueBase` - GenericValueObject
- `wchar` – Byte
- `wstring` – String

Odds and Ends

Bounded strings – String and wstring bound information is not captured in the IDL to PICML mapping

Arrays – IDL arrays are mapped to the PICML model element called Collection, which is a reference to the model element corresponding to the IDL declaration of the array element type. The dimension of the array is not preserved in PICML models. As noted in the section IDL Issues, anonymous arrays (used as members without a wrapping typedef declaration0 are deprecated by the OMG, and are not supported in the IDL to PICML mapping.

Return types – Non-void IDL operation return types are represented in PICML by a model element called ReturnType, which is a reference to the model element corresponding to the IDL declaration of the type, or to a basic type. An IDL operation with a void return type is represented in PICML by a OnewayOperation or TwowayOperation that contains no ReturnType instance.

abstract

In IDL, an interface, valuetype or eventtype may be abstract. In PICML, the corresponding model elements Object, ValueObject, and Event each have a GME attribute called “abstract”, which is a Boolean GME attribute and is false by default.

attribute

Maps to a PICML model element called Attribute or, if the attribute is `readonly`, to a PICML model element called ReadonlyAttribute. The type of the IDL attribute is contained inside the Attribute or ReadonlyAttribute, in a PICML model element called AttributeMember, which is a reference to the model element representing the declaration of the type, or the basic type. If the IDL attribute has any associated `getraises` or `setraises` declarations, they appear in PICML (also contained by the Attribute or ReadonlyAttribute) as GetException or SetException, which are references to PICML Exception model elements (see `exception`).

case

Begins a union label in IDL. In PICML, IDL unions (see `union`) are represented by SwitchedAggregate, which contains a Discriminator (see `switch`) and one or more elements called Member. In a SwitchedAggregate, each Member must be connected to one or more PICML

model elements called Labels. The name of each Label is the stringified value of the IDL case label value. A default case in IDL will map to a Label with the name “default”.

component

Maps to a PICML model element called Component. For mappings of component-related types, see the relevant IDL keyword entry in this list. In IDL, a component must be managed by at least one home declaration, whereas in PICML a Component may have no connection to any ComponentFactory (see `home`). As with IDL a Component may optionally inherit from a single parent Component, by containing a PICML model element called Inherits, which is a reference to the model element corresponding to the IDL declaration of the base `component`. Also as with IDL, a Component may optionally support any number of Objects, by containing model elements called Supports, which are references to model elements corresponding to the IDL declarations of the supported interfaces.

const

Maps to a PICML model element called Constant, which has an attribute called “value” which contains the stringified representation of the IDL constant’s value. The PICML model element Constant is a GME Reference, and it refers to some other element in the PICML model which represents the type of the constant, which may be either a basic type or an enumerated type.

consumes

In IDL this represents an event sink in a component. It maps in PICML to a model element called InEventPort, which is a reference, referring to the model element corresponding to the IDL eventtype which is consumed.

context

No analogue in PICML.

custom

No analogue in PICML.

default

Maps to a PICML model element called Label that has the name “default” (see `case`).

exception

IDL exceptions map to PICML model elements called Exception. Like IDL exceptions, PICML Exceptions can have zero or more members. In PICML, these model elements are called Members, and they are references to the model elements corresponding to the IDL declarations of the member types.

emits

In IDL this represents an event source in a component, an event source that sends to a single subscriber (as opposed to `publishes`, which represents an event source sending to any number of subscribers). An event source is represented in PICML by a model element called OutEventPort.,

which has a Boolean attribute called “single_destination”. The IDL to PICML mapping sets this attribute to TRUE when mapping an `emits` declaration. The attribute is FALSE by default.

enum

The mapping of an IDL enum to PICML is straightforward. The corresponding PICML model element is called Enum. An Enum contains one or more PICML model elements called EnumValue. The name of an EnumValue corresponds to the name of an IDL enum memberf.

eventtype

Maps to a PICML model element called Event. For mappings of eventtype-related types, see the relevant IDL keyword entry in this list.

factory

In IDL, this declaration can be found either in a `valuetype` or a `home`. This is correspondingly true in PICML as well, and in PICML is called FactoryOperation. Also as with an IDL `factory`, a FactoryOperation is considered to be a specialized form of an operation., where the return type is implicit and only IN parameters are allowed (see `parameter`).

finder

In IDL, this is another form of specialized operation, found only in a `home` declaration. The corresponding PICML model element is called LookupOperation, and can be found only contained in a ComponentFactory (see `home`). Like `factory` and FactoryOperation, `finder` and LookupOperation have an implicit return type and may take only IN parameters, if any.

fixed

No analogue in PICML.

getraises

Mapped to a PICML model element called GetException, which is a reference to the Exception model element corresponding to the IDL `exception` declaration.

home

A component home in IDL maps to a PICML model element called ComponentFactory. ComponentFactory has the same semantics as an IDL `home`, including the constraint that it must manage exactly one Component, although in PICML, a Component need not always be managed by a ComponentFactory (see `component`). As with IDL, a ComponentFactory may optionally inherit from a single ComponentFactory, by containing a PICML model element called Inherits, which is a reference to the model element corresponding to the IDL declaration of the base `home`. Also as with IDL, a ComponentFactory may support any number of Objects (see `interface`), by containing PICML model elements called Supports, which are references to the model element corresponding to the IDL declaration of the supported interfaces. Finally, again as with IDL, a ComponentFactory may be associated with a key that assists in looking up the managed component in a database or repository. In IDL the keyword is `primarykey` and refers to a `valuetype`. A PICML ComponentFactory may optionally contain a single model element called LookupKey,

which ia a reference to the ValueObject model element corresponding to the IDL declaration of the `valuetype`.

import

No analogue in PICML.

in

In IDL this keyword qualifies an operation parameter. PICML has 3 types of operation parameters, called InParameter, InoutParameter, and OutParameter, to correspond to the 3 IDL operation parameter qualifiers. OnewayOperation, TwowayOperation, FactoryOperation and LookupOperation model elements may contain zero or more of these as circumstances and constraints dictate.

inout

See `in`.

interface

Maps to a PICML model element called Object. This model element has two attributes, called “abstract” and “local” corresponding to the similar IDL keywords that may precede `interface`. As with IDL, an Object may inherit from any number of other Objects. In PICML this is represented by the containment, in Object, of model elements called Inherits, which are references to the model elements corresponding to the IDL declarations of the parent interfaces.

local

In IDL this keyword may qualify an `interface` declaration. In PICML, an Object (see `interface`) has a Boolean GME attribute called “abstract” to indicate locality constraint. The default value for this attribute is FALSE.

manages

In IDL, this keyword represents the relationship between a `home` type and a `component` type. In PICML, this relationship is represented by a connection between a ComponentFactory (see `home`) and a Component (see `component`). However, if the Component does not appear in the model in the same scope as the ComponentFactory, an additional model element called a ComopnentRef (which is a reference to a Component) will be added to the ComponentFactory’s scope, and the connection will be shown between the ComponentFactory and the ComponentRef,

module

Maps to a PICML model element called Package. As with IDL modules, Packages can be nested, and the same Package name may appear more than once in its scope (corresponding to a reopened IDL `module`).

multiple

This IDL keyword qualifies a `uses` declaration, indicating that multiple connections may be made to this component port. In PICML, the model element representing a `uses` port is called

RequiredRequestPort (see `uses`), and this model element has a Boolean GME attribute called "multiple_connections" to store this information. The default value of this attribute is FALSE.

native

No analogue in PICML.

oneway

This IDL keyword qualifies an `operation`, indicating that no reply is expected from this type of request, and implying that the return type is `void`, that there are only IN parameters, if any, and that the operation may raise no user-defined `exception`. PICML represents these semantics with two model elements for operations, TwowayOperation and OnewayOperation.

out

See `in`.

primarykey

In IDL, this keyword indicates a `valuetype` that may be used by a `home` to look up a `component` in a database or some other kind of persistent storage. A PICML ComponentFactory (see `home`) may likewise contain a model element called LookupKey, which is a reference to the ValueObject (see `valuetype`) corresponding to the IDL declaration of the `valuetype` used as a `primarykey`.

private

This IDL keyword qualifies the member of a `valuetype` or `eventtype`. A PICML ValueObject (see `valuetype`) or Event (see `eventtype`) contains Member model elements that may each optionally be connected to a single model element called PrivateFlag. ValueObject and Event Members that are not connected to a PrivateFlag correspond to public `valuetype` and `eventtype` members.

provides

This type of IDL component port is represented in PICML by a model element called ProvidedRequestPort. It is a reference to the Object (see `interface`) corresponding to the IDL `interface` that implements the port. It is legal in IDL for this kind of port to be of type `Object,` in which case the PICML model element's reference would be to GenericObject in the PredefinedTypes folder (see the section Basic Types above).

public

This IDL keyword qualified the members of `valuetype` and `eventtype`. The corresponding PICML Members (see `valuetype`, `eventtype`) are public by default (see `private`).

publishes

This IDL keyword denotes a `component` port that is an event source. In PICML, a Component (see `component`) can contain zero or more model elements called OutEventPort. These model elements may also represent an IDL `emits` declaration, if the OutEventPort's GME Boolean attribute "single_destination" (FALSE by default) is set to TRUE. An OutEventPort is a reference to the Event model element corresponding to the IDL `eventtype` declaration that implements the port.

raises

In IDL, this keyword associates exceptions with an `operation`, `factory` or `readonly attribute`. In PICML, this keyword maps to a model element called ExceptionRef for TwowayOperations, OnewayOperations, (see `oneway`) FactoryOperations (see `factory`) and LookupOperations (see `finder`), and to a model element called GetExceptions (see `attribute`) for ReadonlyAttributes. Both model elements are references to the Exception corresponding to the IDL declaration of the `exception` the operation may raise.

readonly

This IDL keyword may qualify an `attribute` declaration. In PICML, these two keywords together map to a model element called ReadonlyAttribute (see `attribute`). Like an Attribute, a ReadonlyAttribute contains a single AttributeMember that is a reference to the model element corresponding to the IDL attribute's type, and zero or more GetException references (see `raises`, `attribute`).

setraises

This is an IDL keyword that refers to one or more exceptions that may be thrown when executing the mutator version of the operation implied by the `attribute` declaration. In PICML, a model element called SetExceptions fulfills this role, and it refers to the Exception model elements corresponding to the IDL `exception` declaration (see getraises).

sequence

The OMG deprecates anonymous sequences (used without a typedef) and anonymous sequences are not supported in the IDL to PICML mapping. An IDL `sequence` typedef is represented in PICML by a model element called a Collection (implying that it contains things of one type), which is a reference to the model element corresponding to the declaration of the IDL sequences's base type, or to a basic type, which has no declaration in IDL. Bounded IDL sequences are mapped to PICML the same as unbounded ones – the bound information is not preserved.

struct

An IDL struct is represented in PICML by a model element called Aggregate (implying that it contains things of dissimilar type). An Aggregate may contain one or more model elements called a Member, which is a reference to the actual member type.

supports

In IDL, a `valuetype`, `eventtype`, `component` or home can support the operations of other interfaces. PICML has a model element called Supports, that may be contained in its corresponding model elements – ValueObject, Event, Component and ComponentFactory to represent the same semantics. The Supports model element is a reference to the Object corresponding to the declaration of the IDL `interface` that is supported (see `component`, `home`, `valuetype`, `eventtype`).

switch

In IDL, this keyword is followed by the type of the `union` discriminator. In PICML, an IDL `union` is represented by a model element called a SwitchedAggregate (see `union`), implying that the model element is similar to an Aggregate (representing an IDL `struct`) but with only a single member "active" at any given time. A SwitchedAggregate must contain exactly one model element called Discriminator, which is a reference to the model element corresponding to the IDL type used by the discrimiator.

truncatable

No analogue in PICML.

typedef

This IDL keyword is represented in PICML by a model element called Alias, which is a reference to the model element corresponding to the aliased IDL type.

typeid

An IDL keyword that has been recently added to give official status to the replacement of the repository id with something other than what would normally be created by the IDL compiler, an activity also described by the older `#pragma ID.` The appearance of either one of these in an IDL file will set the SpecifyIdTag GME string attribute in the appropriate model element (see the Pragma Directives section above). The default value of this attribute is the empty string.

typeprefix

An IDL keyword that has been recently added to give official status to the repository id prefixing activity that is also indicated by the older `#pragma prefix`. The appearance of either one of these in an IDL file will set the PrefixTag GEM string attribute (see the Pragma Directives section above). The default value of thies attribute is the empty string.

union

Represented in PICML by a model element called SwitchedAggregate, implying similarity to the model element Aggregate (see struct) but with a mechanism to store (and send over the wire) a single member, designated "active" at any given time. The active member is indicated by the value of a Discriminator (see `switch`) model element contained in the SwitchedAggregate. As with IDL, PICML supports multiple case labels and a default case (see `label`, `default`).

uses

Represented in PICML by a model element called RequiredRequestPort, which is contained in a Component (see `component`), and which is a reference to the Object model element corresponding to the IDL `interface` implementing the port. If the `uses` keyword in IDL is followed by the `multiple` keyword (see `multiple`), the RequiredRequestPort's GME Boolean attribute "multiple_connections" will be set to TRUE – the default value is FALSE. It is legal in IDL for this kind of port to be of type `Object,` in which case the PICML model element's reference would be to GenericObject in the PredefinedTypes folder (see the section Basic Types above).

valuetype

Represented in PICML by a model element called ValueObject, implying that it has some of the semantics of a PICML Object (as an IDL `valuetype` has some of the semantics of an `interface`), but with additional semantics concerning the state. A ValueObject may contain a single Inherits model element, which is a reference to a model element corresponding to the base class valuetype in IDL. A ValueObject model element may also contain Supports model elements (see `supports`).

void

If an IDL operation has a void return type, the corresponding PICML OnewayOperation or TwowayOperation will simply contain no ReturnType model element (see the Odds and Ends section above).