

D: A Language Framework for Distributed Programming

Cristina Videira Lopes

PhD Thesis, College of Computer Science, Northeastern University. November 1997.

© Copyright 1997 Xerox Corporation. All rights reserved.

Appendix C. The Aspect Weaver

The translation of DJ programs into Java is done by a pre-processor, of which the Aspect Weaver is the most important module. The overall architecture is depicted in Figure 42. The Parser takes DJ programs and constructs representations of them (parse trees); the Semantic Analyzer checks the semantic validity of the tree, and, at the same stage, there may be some local transformations on the trees that facilitate the implementation of the Weaver (e.g. transforming COOL condition variable declarations into ordinary Java variable declarations, transforming RIDL traversal specifications into simpler structures, etc.); the Weaver takes those transformed parse trees and outputs new trees that represent woven Java programs; finally, the un-Parser takes those internal representations of Java programs and outputs Java.

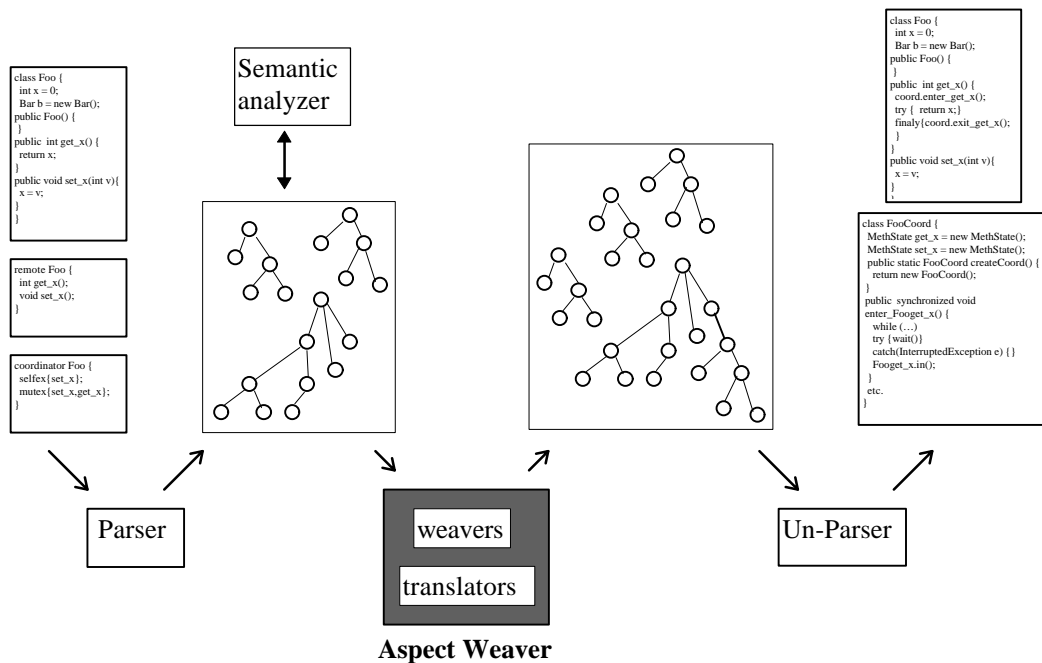


Figure 42. Architecture of the pre-processor that translates DJ programs into Java programs.

This appendix contains the algorithms implemented by the Aspect Weaver. They are presented as a mapping between the input and the output structures. The algorithms will be presented in pseudo-code, along with some comments.

I Notation

The pseudo-language that will be used is a mix of C and Lisp, and similar to any imperative language with type annotations. The only basic types are symbols, lists and integers. Symbols are shown in *italic*, and lists are denoted by `()`, with the elements separated by commas. New types are defined using the pseudo-instruction **record**. Records are denoted by `[<type_of_record>, ...]`. For simplicity sake, the generic **null** symbol will be used to denote the null value of any type (i.e. symbols, lists, integers and records). The other pseudo-instructions are: variable assignment (`:=`); environments (**global**, **let**); loops (**foreach**, **while**); branching (**if else**); return from functions (**return**); de-structuring (**given**); function definitions (**function**) and function calls (e.g., `foo(arg1, arg2)`).

About the pseudo-records

Pseudo-records are used to represent the input and output structures of the Weaver (see Figure 42). A typed record is a finite list whose first element is a token that defines the record's type and each following element (called "field") of the list is a pair keyword/value. For example, the class

```
class Foo extends Bar {
  private int x = 0;
  public Foo() {super();}
  public Foo(int value) { x = value; }
  public synchronized int get_x() { return x; }
}
```

is represented by the record

```
[class, name: Foo, super: Bar,
  variables: ([vardecl, qualifiers: private, type: int, name: x,
              init: [literal, 0]]),
  constructors: ([constructor, args: (), body: ([supercall, args: ()]),
                [constructor, args: (int),
                  body: ([assign, left: [var_ref, name: x],
                        right: [var_ref, name: value]])]),
  methods: ([method, qualifiers: public synchronized, type: int,
            name: get_x, params: (), body: [return, expr: x]])]
```

A record may be created empty, partially complete or complete. For example, `[class]` denotes an empty class record, with all fields defaulting to typed null values; `[class, name: Foo]` denotes a class named `Foo`, and with all other fields defaulting to typed null values.

For the sake of simplicity, the following notation will be used: `r.field` is the value of the element whose keyword is `field` in record `r`. For example, if `c` denotes the class record shown above, `c.super` denotes the value of the pair that has "super" as keyword, i.e. `Bar`.

II Definitions and Auxiliary Functions

1 *The Input and Output Records*

The input of the Weaver consists of representations of JCore classes, COOL coordinators and RIDL portals. The output consists of representations of Java classes and interfaces. All constructs of these languages are represented here by pseudo-records. The input records describe slightly transformed DJ parse trees, and the output records represent Java programs. Since JCore is Java without the synchronized statement, the input records are a super-set of the output records. The following list of record definitions describes the most important records used by the Weaver.

```

/** JCore and Java */
// Only 3 records are shown.
record class      = [name: symbol, super: symbol,
                    interfaces: list of symbol,
                    variables: list of vardecl,
                    constructors: list of constructor,
                    methods: list of method]
record interface = [name: symbol, supers: list of symbol,
                    methods: list of method] // these method bodies
                                           // are always null.
record constructor = [params: list of param, body: statement]
record method      = [qualifiers: combination of symbol,
                    type: symbol, name: symbol,
                    params: list of param,
                    throws: list of symbol, body: statement]

/** COOL */
record coordinator = [granularity: one of {per_object, per_class},
                    classes: list of class,
                    vars: list of vardecl,
                    selfex: list of qualified_name,
                    mutexes: list of mutex,
                    mmanagers: list of mmanager]
record vardecl     = [type: symbol, name: symbol, init: expression]
record mutex       = [mux: list of qualified_name]
record mmanager    = [mnames: list of qualified_name,
                    requires: expression,
                    on_entry: statement,
                    on_exit: statement]
record qualified_name = [cname: symbol, mname: symbol]
/** RIDL */
record portal      = [class: class, operations: list of operation]
record operation   = [type: ridl_type, name: symbol,
                    params: list of ridl_type]
record ridl_type   = [type: symbol, name: symbol,
                    mode: one of {gref, copy},
                    traversal: traversal]
record traversal   = [incompletes: list of incomplete_class]
record incomplete_class = [name: symbol, missing: list of symbol]

```

Note: From here on, there is, sometimes, a slight abuse of terminology by using, for example, “the class” instead of “the record representing a class”. Also, the handling of Java values is sometimes

shortened: the literal records are dropped out, the type of the value is left unspecified, and only the value is shown.

2 Constants

```
// All of these constants are for weaving RIDL

ZERO          = [literal, value: 0]
ZEROOBJECT    = [new, class: Integer, args: (ZERO)]

WRITEEXTERNAL = [method, qualifiers: public, type: void,
                 name: writeExternal,
                 params: ([param, type: OutputStream, name: out])]
READEXTERNAL  = [method, qualifiers: public, type: void, name: readExternal,
                 params: ([param, type: InputStream, name: in])]
D_WRITEEXTERNAL = [method, qualifiers: public, type: void,
                  name: _d_writeExternal,
                  params: ([param, type: OutputStream, name: out],
                          [param, type: Traversal, name: t])]
D_READEXTERNAL = [method, qualifiers: public, type: void,
                  name: _d_readExternal,
                  params: ([param, type: InputStream, name: in],
                          [param, type: Traversal, name: t])]

BYPASSWRITETEST = [if, expr: [not, expr: [invocation, obj: c, meth: bypassPart]]]
BYPASSREADTEST  = [if, expr: [invocation, obj: c, meth: bypassPart]]

READOBJECT      = [invocation, obj: in, meth: readObject]
READTOKEN       = [invocation, obj: [cast, type: String, expr: READOBJECT],
                  meth: equals, args: ([literal, DObject])]

NEWINSTANCE     = [invocation, obj: [invocation, obj: Class, meth: forName,
                                     args: (classname)],
                  meth: newInstance]

CATCHINGWRAPPER = ([catch, exception: [param, type: DInvalidRemoteException,
                                       name: e],
                   body: [invocation,
                          obj: [field_ref, obj: System, field: err],
                          meth: println,
                          params: ([literal,
                                   value: "Invalid remote operation"])]])
CATCH_INMARSHALING = [catch, exception: [param, type: Exception, name: e],
                      body: [invocation,
                             obj: [field_ref, obj: System, field: err],
                             meth: println,
                             params: ([invocation, obj: e,
                                       meth: toString])]]]
```

3 Auxiliary Functions

- `Append(<list>, <element1>, ..., <elementn>)`: appends the given elements to the given list and returns the list. The given list is extended with the new elements.
- `Clone(<record>)`: returns a record which is identical to the given record.

- `Concat(<lists of symbols> or <single symbols>)`: returns a symbol that is the concatenation of all the symbols given.
- `LookupClass(<symbol>)`: the input is a class name. This function returns the class record representing the class with the given name. This record may be incomplete; the lookup of a class may result in filling only the record's name, super, interfaces, and method signatures (i.e. no variables, no constructors and no method bodies). The return record itself may be the null record, if the class does not exist or if the given name is null.
- `LookupCoordinators()`: returns a list containing all coordinator records (e.g. from file extensions). These records are incomplete: they only contain the `classes` field, and each of those fields only contains the name of the class.
- `LookupClassWithAspect(<class record>, <symbol>)`: returns a record representing the closest class in the class hierarchy, starting at (and including) the given class, that is associated with an aspect module of the kind given by the symbol (i.e. *cool* or *ridl*). As in the function before, the resulting class record may be incomplete. If no class is associated with an aspect module of the given kind, the function returns ***null***.
- `Match(<symbol>, <list of symbols>)`: returns *true* if the given list contains at least one occurrence of the given symbol; *false* otherwise.
- `Methods(<list of typed record>)`: returns a list of symbols corresponding to the “method” fields of all the given records. (similar to `Names`, below)
- `Names(<list of typed record>)`: returns a list of symbols corresponding to the “name” fields of all the given records. E.g. `Names([class, name: Foo], [class, name: Bar])` is `(Foo, Bar)`.
- `Qname(<class record>, <method record>)`: returns a symbol consisting of the concatenation of the given class name and the given method name. E.g. `Qname([class, name: Foo], [method, name: bar])` is `Foo``bar`.
- `Types(<list typed record>)`: returns a list of symbols corresponding to the “type” fields of all the given records. (similar to `Names`, above)

III The Weaving Engine

The weaving engines of COOL and RIDL are similar, and are captured by two generic weaving functions:

- `direct_weave`, which weaves a class when it is directly associated with an aspect module,
- `inherit_weave`, which weaves a class when it is not directly associated with an aspect module, but inherits an aspect module from a superclass.

The aspect-specificity of this weaving engine shows up only in the calls to `wrapper_body`. Wrapper bodies are considerably different for each aspect and for when aspects are combined together. But the core of the weaving engine, given by those two functions, is aspect-independent.

The fundamental reason why there are two generic engines, instead of one, is the following. According to the implementation architecture described in Chapter 4, when a class is directly associated with an aspect module, the weaver must process the class's own methods as well as *all* non-private methods of *all* superclasses, whereas when the class inherits an aspect module, the weaver simply needs to process the class's own methods.

```

//direct_weave is called when the class is directly associated with an aspect
// module (coordinator or portal).
// Input: class: the class that is to be woven;
//         newvars: the variable declarations that must be woven;
//         init_code: initialization code to be appended to every constructor;
//         aspect: a token indicating which aspect is being woven;
// Output: the woven class.
function direct_weave (class oftype class,
                      newvars oftype list of vardecl,
                      init_code oftype statement,
                      aspect oftype symbol)
let wclass = Clone(class) in
  // weave the extra variable declarations
  foreach var ∈ newvars,
    Append(wclass.variables, var)
  // weave the initialization in every constructor
  if wclass.constructors == null // no constructors. Weave one.
    wclass.constructors := ([constructor, body: init_code])
  else
    foreach const ∈ wclass.constructors,
      Append(const.body, init_code)

  // Process the methods implemented in the class. Make them into
  // implementation/wrapper pairs.
  foreach meth ∈ class.methods,
    let implmeth = Clone(meth), wrappermeth = Clone(meth) in
      implmeth.name := Concat(_d_, implmeth.name)
      replace_calls_to_super(implmeth.body)
      wrappermeth.body := wrapper_body(wrappermeth, class.name, aspect)
      Append(wclass.methods, implmeth, wrappermeth)
  // Then, the methods that are inherited. Add only the wrapper method
  // in wclass for each method that is inherited.
  let super = LookupClass(class.super), methodNameList = () in
    while super ≠ null
      foreach meth ∈ super.methods,
        if Match(meth.name, methodNameList) == false // not seen before
          Match(meth.name, Names(class.methods)) == false //truly inherited
          and private ∉ meth.qualifiers
            Append(wclass.methods, [method, qualifiers: meth.qualifiers,
                                   type: meth.type,
                                   name: meth.name,
                                   params: meth.params,
                                   body:wrapper_body(meth, class.name,
                                                    aspect)])
            Append(methodNameList, meth.name)
      super := LookupClass(super.super)
  return wclass
end.

```

```

// inherit_weave is called when the class is not associated with an aspect,
// module but it has a superclass that is.
// New methods defined in this class are transformed into a pair of
// implementation/wrapper methods. Methods that are overridden from the
// super with aspect module, simply get transformed in implementation
// methods - the wrapper is inherited from the super.
// Input: class: the class record;
//          superc: the class record of the closest superclass that is directly
//                  associated with a coordinator;
// Output: the woven class.
function inherit_weave (class oftype class,
                       supers_with_aspect oftype list of class)
  let wclass = Clone(class) in
    foreach meth  $\in$  class.methods,
      let implmeth = Clone(meth) in
        implmeth.name := Concat(_d_, implmeth.name)
        replace_calls_to_super(implmeth.body)
        Append(wclass.methods, implmeth)

    if Match(meth.name, Names(Methods(supers_with_aspect))) == false
      // the "wrapper" method. Here, it just calls the implementation
      // method. (Careful about the return type)
      let invocationToImplmeth = [invocation, meth: implmeth.name,
                                args: Names(meth.params)] in

        if meth.type == void
          meth.body := invocationToImplmeth
        else
          meth.body := [return, expr: invocationToImplmeth]
      //else, the wrapper is the one in the super with aspect module.
    return wclass
end.

// wrapper_body dispatches the call according to the specific aspect
// that is being woven.
// Input: meth: the original method for which the wrapper is being generated;
//          cname: the classname of the class where the method is implemented;
//          aspect: a token indicating the aspect that is being woven.
// Output: the body of the aspect-specific wrapper method.
function wrapper_body (wrappermeth oftype method,
                       cname oftype symbol, aspect oftype symbol)
  if aspect == cool
    return wrapper_body_cool(meth, cname)
  else
    if aspect == ridl
      return wrapper_body_ridl(meth)
    else
      return wrapper_body_coolridl(meth, cname, aspect)
end.

```

```

// replace_calls_to_super takes the body of the given method meth and replaces
// the direct calls to super.<meth.name> with calls to super._d<meth.name>.
// (destructively). This is the most costly function to implement, since
// it implies that the parser must parse JCore method bodies.
// Implementers may simply chose to disallow direct calls to super(), to avoid
// parsing full Java.
// Input: meth: a record representing an "implementation" method
// Output: the same record, but with the said substitutions.
function replace_calls_to_super (meth oftype method)
  foreach inv oftype invocation ∈ meth.body such that
    obj == super and meth == meth.name,
    inv.meth := Concat(_d_, meth.name)
end.

// weave is an auxiliary entry point to the weaving engine for weaving exactly
// one aspect in isolation of the other. It is called by weave_cool and
// weave_ridl.
// Subsection §3 presents a new version of this function for weaving both
// aspects at the same time.
// Input: class: record representing the class that is to be woven
//          newvars: a list of new variable declarations to be woven
//          init_code: initialization code to be appended to every constructor
//          aspect: a token indicating which aspect is being woven
// Output: record representing the woven class.
function weave (class oftype class,
               newvars oftype list of vardecl,
               init_code oftype statement,
               aspect oftype symbol)
  let class_with_aspect = LookupClassWithAspect(class, aspect) in
    if class_with_aspect == null // no weaving.
      return Clone(class)
    else // test if it's direct or inherited association with aspect module
      if class_with_aspect.name == class.name // direct
        return direct_weave(class, newvars, init_code, aspect)
      else // inherited
        return inherit_weave(class, (class_with_aspect))
end.

```

1 COOL-specific Functions

```

global coordvarname oftype symbol

// weave_cool is the top function for a stand-alone COOL weaver.
// Input: class: the class record.
// Output: another class based on the input class but with woven code at
//          particular points.
function weave_cool (class oftype class)
  let allcoordinators = LookupCoordinators(),
    if  $\exists$  coord  $\in$  allcoordinators such that class.name  $\in$  Names(coord.classes)
    let coordclassname = Concat(Names(coord.classes), Coord) in
      coordvarname := Concat(_, coordclassname)
      let newvars = ([vardecl, type: coordclassname, name: coordvarname]),
        init_code = init_coordinator_code(coordclassname) in
        return weave (class, newvars, init_code, cool)
    else return class // no weaving.
end.

// wrapper_body_cool generates the body of coordination wrapper methods.
// Input: meth: the original method for which the wrapper is being generated;
//          cname: the classname of the class where the method is implemented;
// Output: the body of the wrapper method.
// Note: coordvarname is a global variable that holds the name that the
//          coordinator variable has in the class that is being woven.
function wrapper_body_cool(meth oftype method, cname oftype symbol)
  let s = [sequence] in
    s.statements := ([invocation, obj: [var_ref, name: coordvarname],
                      meth: Concat(enter_, cname, meth.name),
                      args: this],
                    [try, body: try_body_cool(meth),
                      finally: [invocation,
                                obj:[var_ref, name: coordvarname],
                                meth: Concat(exit_, cname, meth.name),
                                args: this]])

  return s
end.

// try_body_cool generates the body of the try statement inside the wrapper
// methods, which basically consists of a call to the implementation method.
// Input: meth: the method record.
// Output: a Java statement that is either the direct invocation (if there
//          is no return value) or a return statement returning the result
//          of the invocation.
function try_body_cool(meth oftype method)
  let invocationToImplmeth = [invocation, meth: Concat(_d_, meth.name)
                              args: Names(meth.params)] in

  if meth.type == void
    return invocationToImplmeth
  else
    return [return, expr: invocationToImplmeth]
end.

// init_coordinator_code returns the assignment record that represents
// the initialization of the coordinator variable.
function init_coordinator_code(coordclassname)
  return [assignment, left: coordvarname, // coordvarname is global
          right:[invocation, obj:coordclassname,
                 meth:createCoord]]
end.

```

2 RIDL-specific Functions

```

global pvarname oftype symbol
global ppvarname oftype symbol

// weave_ridl is the top function of a stand-alone RIDL weaver.
// Input: class: the class record.
// Output: another class based on the input class but with woven code at
//          particular points.
// Note:   the function marshaling_methods, called at the end, is part of
//          the RIDL weaver.
function weave_ridl (class oftype class)
  let pclassname = Concat(class.name, P),
      ppclassname = Concat(class.name, PP),
      traversalsclassname = Concat(class.name, Traversals) in
  pvarname := Concat(_p, class.name),
  ppvarname := Concat(_rself, class.name),
  let newvars = ([vardecl, qualifiers: public,
                 type: pclassname,
                 name: pvarname],
                [vardecl, qualifiers: public,
                 type: ppclassname,
                 name: ppvarname, init: null]),
      init_code = init_p_code(pclassname) in
  let wclass = weave(class, newvars, init_code, ridl),
      marshals = marshaling_methods(class) in
  Append(wclass.interfaces, DObject)
  foreach meth  $\in$  marshals,
    Append(wclass.methods, meth)
  return wclass
end.

// wrapper_body_ridl generates the body of portal wrapper methods.
// Input: meth: the original method for which the wrapper is being generated;
// Output: the body of the wrapper method.
// Note:   ppvarname is a global variable that holds the name that the
//          PP variable has in the class that is being woven.
function wrapper_body_ridl(meth oftype method)
  return ([if, expr: [not_equal, left: ppvarname, right: null],
          then: [try, body: try_body_ridl,
                catches: CATCHINGWRAPPER],
          else: [invocation, meth: Concat(_d_, meth.name),
                args: Names(meth.params)]]
end.

// try_body_ridl generates the body of the try statement inside the wrapper
// methods, which basically consists of a call to the PP object.
// Input: meth: the method record.
// Output: a Java statement that is either the direct invocation (if there
//          is no return value) or a return statement returning the result
//          of the invocation.
function try_body_ridl(meth oftype method)
  let invocationToPP = [invocation, obj: ppvarname, meth: meth.name,
                       args: Names(meth.params)] in
  if meth.type == void
    return invocationToPP
  else
    return [return, expr: invocationToPP]
end.

```

```

// init_p_code returns the assignment record that represents
// the initialization of the P variable.
function init_coordinator_code(pclassname)
  return [assignment, left: pvarname, // pvarname is global
         right:[new, class:pclassname, args: this]]
end.

// The following functions generate the marshaling methods.
// In the following code, the field named "statements" of sequence records
// is omitted for making the code more readable; also omitted are the
// field names "left" and "right" of binary boolean expressions.

function marshaling_methods (class oftype class)
  let writeExt = Clone(WRITEEXTERNAL), readExt = Clone(READEXTERNAL),
      dwriteExt = Clone(D_WRITEEXTERNAL), dreadExt = Clone(D_READEXTERNAL) in
  writeExt.body := simple_write_body(class.variables)
  readExt.body := simple_read_body(class.variables)
  dwriteExt.body := traversal_write_body(class.name, class.variables)
  dreadExt.body := traversal_read_body(class.name, class.variables)
  return (writeExt, readExt, dwriteExt, dreadExt)
end.

// Code for packing (write functions)
function simple_write_body(vars oftype list of vardecl)
  let s = [sequence] in
    foreach var ∈ vars,
      if static ≠ var.qualifiers
        Append(s.statements, [invocation, obj: s, meth: writeObject,
                             args: (var.name)])

    return s
end.

function traversal_write_body(cname oftype symbol,vars oftype list of vardecl)
  let s = [sequence, (traversal_method_vardecl(cname))] in
    foreach var ∈ vars,
      if static ≠ var.qualifiers
        let bypasstest = Clone(BYPASSWRITETEST) in
          bypasstest.expr.expr.args := ([stringliteral, value: var.name])
          if is_primitive(var.type)
            bypasstest.then := [invocation, obj: s, meth: WriteObject,
                               args: (var.name)]
          else
            if is_object(var)
              bypasstest.then := write_object(var.name)
            else // it's array
              bypasstest.then := write_array(var.name)
            Append(s.statements, bypasstest)
        return s
    end.

function write_object (v oftype expression)
  return
  [if, expr: [or,terms:([equal, v, null],
                      [and, terms:([notequal, v, null],
                                   [not, expr: [instanceof, v,DObject]]))]),
   then: [sequence, (WRITEOBJECT,
                    [invocation, obj: out, meth: writeObject, args:(v)])]
   else: [sequence, (WRITEDOBJECT,
                    [invocation, obj: out, meth: writeObject,

```

```

        args: ([invocation, obj: [invocation, obj: v,
                                meth: getClass],
              [invocation, obj: [cast, class: DObject, expr: v],
                                meth: _d_writeExternal, args: (out,t)]]])
end.

function write_array (varname oftype symbol)
return
[if, expr: [equal, varname, null],
 then: [invocation, obj: out, meth: writeObject, args: (ZEROOBJECT)]
 else: [sequence, statements:
       ([invocation, obj: out, meth: writeObject,
         args:([new, class: Integer,
                args:(array_length(varname))])],
        [for, base: [vardecl, type: int, name: _i, init: ZERO],
          test: [lessthan, _i, array_length(varname)],
          action: [incr, obj: _i],
          body: write_object(element_ref(varname))]]])]
end.

// Code for unpacking (read functions)
function simple_read_body(vars oftype list of vardecl)
let s = [sequence] in
  foreach var ∈ vars,
    if static ∉ var.qualifiers
      Append(s.statements, [invocation, obj: s, meth: readObject,
                           args: (var.name)])
return s
end.

function traversal_read_body(cname oftype symbol, vars oftype list of vardecl)
let s = [try, body: sequence, (traversal_method_vardecl(cname))
        catches: (CATCH_INMARSHALING)] in
  foreach var ∈ vars,
    if static ∉ var.qualifiers
      let bypasstest = Clone(BYPASSREADTEST) in
        bypasstest.expr.args := ([literal, value: var.name])
        bypasstest.then := ([assignment, left: var.name, right: null])
        if is_primitive(var.type)
          bypasstest.else := assign_after_read(var.name, var.type,
                                                [invocation, obj: in,
                                                 meth: ReadObject])
        else
          if is_object(var)
            bypasstest.else := read_object(var.name, var.type)
          else // it's array
            bypasstest.else := read_array(var.name, var.type)
        Append(s.body.statements, bypasstest)
return s
end.

function read_object(v oftype expression, type oftype symbol)
return
[if, expr: READTOKEN,
 then: [sequence, (READCLASS,
                  assign_after_read(v,type,[cast, type: type,

```

```

                                expr:NEWINSTANCE)),
    [invocation, obj: [cast, type: DObject, expr: v],
      meth: _d_readExternal,
      args: (in, t)]]]
  else: [assignment, left: v, right: [cast, type: type, READOBJECT]]]
end.

function read_array(varname oftype symbol, type oftype symbol)
  return
    [sequence, (READLENGTH,
      [if, expr: [equal, n, [literal, 0]],
        then: [assignment, varname, null],
        else: [sequence,
          ([assignment, varname, [new, class:type, size:n]],
            array_iteration(varname,
              read_object(element_ref(varname),type)))]])]
end.

function assign_after_read(var oftype expression, type oftype symbol,
  read_expr oftype expression)
  return [assignment, left: var, right: [cast, type: type, expr: read_expr]]
end.

function traversal_method_vardecl(cname oftype symbol)
  return [vardecl, type: DPartCutter, name: c,
    init: [invocation, obj: t, meth: isIncomplete,
      args: ([literal, value: cname])]]]
end.

function array_length (varname oftype symbol)
  return [field_ref, obj: varname, field: length]
end.

function element_ref (varname oftype symbol)
  return [array_ref, obj: varname, index: _i]
end.

function array_iteration (varname oftype symbol, body oftype expression)
  return [for, base: [vardecl, type: int, name: _i, init: ZERO],
    test: [lessthan, _i, array_length(varname)],
    action: [incr, obj: _i],
    body: body]
end.

```

3 Weaving COOL and RIDL Together

In order to weave COOL and RIDL together, a few extra functions are necessary, two at the top level and one at the bottom. However, the generic weaving engine consisting of `direct_weave` and `inherit_weave` is used and remains unchanged.

```

// all global variables from each of the separate weavers are set here
global coordvarname oftype symbol
global pvarname oftype symbol
global pppvarname oftype symbol
// a couple of extra ones that are used for generating the wrapper
global class_with_coord oftype class
global class_with_portal oftype class

```

```

// weave_both is the top function of for weaving COOL and RIDL together.
// It simply sets all global variables, and constructs the representations
// of the new cool and ridl variable declarations and initialization code.
// Input: class: the class record.
// Output: another class based on the input class but with woven code at
//           particular points.
function weave_both (class oftype class)
  let pclassname = Concat(class.name, P),
      ppclassname = Concat(class.name, PP) in
    pvarname := Concat(_p, class.name),
    ppvarname := Concat(_rself, class.name),
  let allcoordinators = LookupCoordinators(),
    if  $\exists$  coord  $\in$  allcoordinators such that class.name  $\in$  Names(coord.classes)
      let coordclassname = Concat(Names(coord.classes), Coord) in
        coordvarname := Concat(_, coordclassname)
  let newcoolvars = ([vardecl, type:coordclassname, name: coordvarname]),
    init_cool_code = init_coordinator_code(coordclassname),
    newridlvars = ([vardecl, qualifiers: public,
                   type: pclassname,
                   name: pvarname],
                  [vardecl, qualifiers: public,
                   type: ppclassname,
                   name: ppvarname, init: null]),
    init_ridl_code = init_p_code(pclassname) in
  let wclass = weave(class, newvars, init_code, ridl),
    marshals = marshaling_methods(class) in
  Append(wclass.interfaces, DObject)
  foreach meth  $\in$  marshals,
    Append(wclass.methods, meth)
  return wclass
end.

```

```

// weave is the entry point to the weaving engine for weaving both aspects
// at the same time. It is called by weave_both.
// Input:  class: record representing the class that is to be woven;
//          newcoolvars: a list of new variable declarations coming from cool;
//          newridlvars: a list of new variable declarations coming from ridl;
//          init_cool_code: initialization code coming from cool;
//          init_ridl_code: initialization code coming from ridl;
// Output: record representing the woven class.
// When weaving both aspects at the same time, there are exactly 9 different
// combinations for the association of the class with the two aspects:
// 1. no direct or inherited coordinator (coord, for short) or
//    portal(ptal, for short);
// 2. direct coord and no ptal;
// 3. inherited coord and no ptal;
// 4. direct ptal and no coord;
// 5. inherited ptal and no coord;
// 6. direct both coord and ptal;
// 7. direct coord and inherited ptal;
// 8. direct ptal and inherited coord;
// 9. inherited both coord and ptal (not necessarily the same super for both)
// Each of these combinations results in different weavings.
//
function weave (class oftype class,
               newcoolvars oftype list of vardecl,
               newridlvars oftype list of vardecl,
               init_cool_code oftype statement,
               init_ridl_code oftype statement)
class_with_coord := LookupClassWithAspect(class, cool)
class_with_portal := LookupClassWithAspect(class, ridl)
if class_with_coord ≠ null or class_with_portal ≠ null // 1
  /** The first 4 cases are single aspect weaving */
  if class_with_coord ≠ null and class_with_portal == null // 2
    if class_with_coord.name == class.name // 2
      return direct_weave(class, newcoolvars, init_cool_code, cool) // 2
    else // 3
      return inherit_weave(class, class_with_coord) // 3
  else
    if class_with_coord == null and class_with_portal ≠ null // 4
      if class_with_portal.name == class.name // 4
        return direct_weave(class, newridlvars, init_ridl_code, ridl) // 4
      else // 5
        return inherit_weave(class, class_with_portal) // 5
    /** The next 4 cases are the new ones */
    else
      if class_with_coord.name == class.name and // 6
        class_with_portal.name == class.name // 6
        return direct_weave(class, Merge(newcoolvars, newridlvars),
                           [sequence: (init_cool_code, init_ridl_code),
                            coolridl]) // 6
      else
        if class_with_coord.name == class.name and // 7
          class_with_portal.name ≠ class.name // 7
          return direct_weave(class, newcoolvars, init_cool_code, coolridl) // 7
        else
          if class_with_coord.name ≠ class.name and // 8
            class_with_portal.name == class.name // 8
            return direct_weave(class, newridlvars, init_ridl_code, coolridl) // 8
          else // 9
            return inherit_weave(class, (class_with_coord, class_with_portal)) // 9
    else return Clone(class)
end.

```

```

// wrapper_body_coolridl generates the body of coordination wrapper methods.
// Input: meth: the original method for which the wrapper is being generated;
//         cname: the classname of the class that is being woven;
// Output: the body of the wrapper method.
//
// This function needs to decide on the precise combination of wrappers, which
// depends on the superclasses with aspect modules. The logic is as follows:
// Case 1. class_with_coord.name = class_with_portal.name = cname
//         This means that the class that is being woven is directly
//         associated with both a coordinator and a portal.
//         Therefore, the wrapper must merge both aspect wrappers for this
//         class.
// Case 2. class_with_coord.name = cname and class_with_portal ≠ cname
//         This means that the class that is being woven is directly
//         associated with a coordinator and inherits a portal.
//         Therefore, the wrapper should contain only the COOL wrapper.
//         However, when the method overrides a method of class_with_portal,
//         the RIDL wrapper must be included too.
// Case 3. class_with_portal = cname and class_with_coord.name ≠ cname
//         This means that the class that is being woven is directly
//         associated with a portal and inherits a coordinator.
//         Therefore, the wrapper should contain only the RIDL wrapper.
//         However, when the method overrides a method of class_with_coord,
//         the COOL wrapper must be included too.
// Case 4. class_with_portal ≠ cname and class_with_portal ≠ cname
//         does not occur here; such case is handled by inherit_weave. This
//         function is called only if both aspect modules are associated
//         with this class, and at least one of them is directly associated.
//
// Note: global variables: coordvarname, ppvarname.
//
function wrapper_body_coolridl(meth oftype method, cname oftype symbol)
  let coolwrapper = [sequence,
                    statements: ([invocation, obj: coordvarname,
                                  meth: Concat(enter_, cname, meth.name),
                                  args: this],
                                  [try, body: try_body_cool(meth),
                                   finally: [invocation,
                                             obj: coordvarname,
                                             meth: Concat(exit_, cname, meth.name),
                                             args: this]])],
                    nocoolwrapper = [invocation, meth: Concat(_d_, meth.name),
                                      args: Names(meth.params)] in

  if (class_with_coord.name == cname and class_with_portal.name == cname) or
    (class_with_coord.name ≠ cname and // the exception of Case 3
     Match(meth.name, Names(class_with_coord.methods))) or
    (class_with_portal.name ≠ cname and // the exception of Case 2
     Match(meth.name, Names(class_with_portal.methods)))
  return ([if, expr: [not_equal, left: ppvarname, right: null],
            then: [try, body: try_body_ridl,
                  catches: CATCHINGWRAPPER],
            else: coolwrapper])
  else // not inherited from super with other aspect
  if class_with_coord.name == cname // only COOL wrapper
  return coolwrapper
  else // only RIDL wrapper
  return ([if, expr: [not_equal, left: ppvarname, right: null],
            then: [try, body: try_body_ridl,
                  catches: CATCHINGWRAPPER],
            else: nocoolwrapper])
end.

```

IV Translation Engine for COOL

```

// translate_coordinator is the top function of the translation for COOL.
// Input: coord: a coordinator.
// Output: a class resulting from the given coordinator (coordinator class).
function translate_coordinator (coord oftype coordinator)
  let c = [class, name: Concat(Names(coord.classes), Coord)] in
    /** The variables */
    if coord.granularity == per_class // add the "theCoord" variable
      c.variables := ([vardecl, qualifiers: static, type: boolean,
                     name: one, init: false],
                    [vardecl, qualifiers: static, type: c.name,
                     name: theCoord])

    // The method state variables. One per method of all the classes,
    // including methods inherited from superclasses
    foreach class ∈ coord.classes,
      let aclass = class, methodnames = () in
        while aclass != null
          foreach method ∈ aclass.methods,
            if Match(method.name, methodnames)==false // avoid repeating names
              Append(methodnames, method.name) // of overridden methods
              Append(c.variables, [vardecl, type: MethState,
                                   name: Qname(class,method),
                                   init: [new, class: MethState]])
          aclass := LookupClass(aclass.super) // up the class hierarchy

    // The condition and ordinary variables
    foreach var ∈ coord.vars,
      Append(c.variables, [vardecl, type: var.type, name: var.name,
                          init: var.init]))

    /** The methods */
    // The factory method
    c.methods := ([method, qualifiers: public static synchronized,
                  type: c.name, name: createCoord,
                  body: factory_body(coord, c.name)])
    // The before and after methods
    foreach class ∈ coord.classes,
      let aclass = class, methodnames = () in
        while aclass != null
          foreach method ∈ aclass.methods,
            if Match(method.name, methodnames)==false // avoid repeating names
              Append(methodnames, method.name) // of overridden methods
              Append(c.methods, [method, qualifiers: public synchronized,
                                  type: void,
                                  name: Concat(enter_, Qname(class,
                                                            method))
                                  body: before_body(coord, c.name)],
                            [method, qualifiers: public synchronized,
                              type: void,
                              name: Concat(exit_, Qname(class,
                                                            method))
                              body: after_body(coord, c.name,
                                                method)])
          aclass := LookupClass(aclass.super) // up the class hierarchy
    return wclass
end.

```

```

// The next three functions generate the bodies of the methods of the
// coordinator class that results from translating a COOL coordinator.
// factory_body returns a record representing the body of the factory method
//   Input: granularity: the granularity of the coordinator;
//           cname: the name of the class that corresponds to the coordinator
//           that is being translated.
//   Output: a statement record.
// before_body and after_body generate the body of a "before" method and a
// "after" method, respectively.
//   Input: coord:coordinator record; class: class record; meth: method record
//   Output: a statement record.

function factory_body (granularity oftype symbol, cname oftype symbol)
  let s oftype statement in
    if granularity == per_class
      s := [sequence, ([if, expr: [not, expr: [var_ref, name: one]],
                        then: [assignment, left: [var_ref, name: theCoord],
                                           right: [new, class: cname]]]),
           [return, expr: [var_ref, name: theCoord]]]
    else s := [return, expr: [new, class: cname]]
  return s
end.

function before_body(coord oftype coordinator, class oftype class,
                    meth oftype method)
  let qname = [qualified_name, class: class.name, method: meth.name] in
    if qname ∉ coord.selfex and
      ∃ mutex ∈ coord.mutexes, qname ∉ mutex.mux and
      ∃ mm ∈ coord.mmanagers, qname ∉ mm.mnames
      return [] // Nothing to be coordinated. Return empty body.
    else
      let s oftype statement in
        s := [sequence,
              statements:([while, expr: waiting_condition(coord,class,meth),
                          body: COOLWAITBODY],
                          [invocation, obj:[var_ref, name: qname], meth: in])]
        foreach mm ∈ coord.mmanagers,
          if mm contains on_entry statements for meth
            Append(s.statements, mm.on_entry)
        return s
      end.
    end.
end.

```

```

function after_body (coord oftype coordinator, class oftype class,
                    meth oftype method)
  let qname = [qualified_name, class: class.name, method: meth.name] in
  if qname  $\notin$  coord.selfex and
     $\forall$  mutex  $\in$  coord.mutexes, qname  $\notin$  mutex.mux and
     $\forall$  mm  $\in$  coord.mmanagers, qname  $\notin$  mm.mnames
    return [] // Nothing to be coordinated. Return empty body.
  else
    let s oftype statement in
      // First, the call to "out" on the method state object
      s := [sequence,
           statements:([invocation, obj:[var_ref,name: qname], meth: out])]
      // Then, the on_exit statements
      foreach mm  $\in$  coord.mmanagers,
        if mm contains on_exit statements for meth
          Append(s.statements, mm.on_exit)
      // Finally, the call to notifyAll
      Append(s.statements, [if, expr: [equals,
                                     left: [field_ref, obj: qname,
                                             field: [var_ref, name: depth]]
                                     right: [literal, 0]]
                           then: [invocation, meth: notifyAll]])

    return s
  end.

// The next function generates the waiting condition for method m of the JCore
// class c, c.m. The waiting condition depends on the exclusion constraints in
// the self-exclusion set, the exclusion constraints in the mutual exclusion
// sets, and on the requires clause, if any, declared in a method manager. A
// thread wanting to execute method c.m must wait if:
// - c.m is selfex and another thread is executing M; or
// - for any other method c'.m' , c.m is mutually exclusive with c'.m'
//   and another thread is executing c'.m'; or
// - the pre-condition, as declared in a method manager, is false.
// Input: coord: coordinator record; class: class record, meth, method record
// Output: a boolean expression
function waiting_condition (coord oftype coordinator, class oftype class,
                          meth oftype method)
  let qname = [qualified_name, class: class.name, method: meth.name],
      condSet = () in
    // Check if method is self-exclusive
    if qname  $\in$  coord.selfex
      Append(condSet, [invocation, obj: [var_ref, name: qname],
                    meth: isBusyByOther])
    // Check mutual exclusion with other methods
    foreach c'  $\in$  coord.classes
      foreach m'  $\in$  c'.methods,
        let qname' = [qualified_name, name: c'.name, method: m'.name] in
          if qname'  $\neq$  qname and  $\exists$  mutexk  $\in$  coord.mutexes such that
            qname  $\in$  mutexk and qname'  $\in$  mutexk
            Append(condSet, [invocation, obj: [var_ref, name: qname'],
                          meth: isBusyByOther])
    // Check if there exists a pre-condition
    if  $\exists$  mm  $\in$  coord.mmanagers such that mm contains requires clause for qname
      Append(condSet, [not, expr: mm.requires])

    if CondSet == () return false
    else return [or, terms: condSet]
  end.

```

V Translation Engine for RIDL

```

// translate_portal is the top translation function for RIDL.
// Input: ptal: portal record.
// Output: a list of four class records:
//         pri: the record corresponding to the Java interface;
//         p: the record corresponding to the P class;
//         pp: the record corresponding to the PP class;
//         ct: the record corresponding to the traversals class.
function translate_portal (ptal oftype portal)
    return (generate_ri(ptal), generate_p(ptal),
           generate_pp(ptal), generate_traversals(ptal))
end.

// generate_pri generates the Java interface corresponding to the given RIDL
// portal
// Input: ptal: a portal record.
// Output: the record corresponding to the Java interface.
function generate_pri (ptal oftype portal)
    let pri = [interface, name: Concat(ptal.class.name, PRI), supers: Remote] in
        foreach op ∈ ptal.operations, given op.params ≡ (rt1, ..., rtn),
            Append(pri.methods, [method, type: ridl2java_type(type), name: op.name,
                                params: ([param, type: ridl2java_type(rt1),
                                        name: rt1.name],
                                        ...
                                        [param, type: ridl2java_type(rtn),
                                        name: rtn.name]),
                                throws: RemoteException])

    return ptal
end.

// generate_p returns the record of the class used to instantiate P
// objects associated with D remote objects.
// Input: ptal: a portal record.
// Output: the record corresponding to the class used to instantiate Ps
function generate_p (ptal oftype portal)
    let p = [class, name: Concat(ptal.class.name, P),
            interfaces: Concat(ptal.class.name, PRI)] in
        p.variables := ([vardecl, type: ptal.class.name, name: myself],
                       [vardecl, type: RemoteStub, name: mystub])
        p.constructors := ([constructor, params: ([param, type: ptal.class.name,
                                                name: s])
                          body: PCONSTRUCTORBODY])

    // iterate over the remote operations, and generate one method for
    // each of them, attaching it to the methods of p
    foreach op ∈ ptal.operations, given op.params ≡ (rt1, ..., rtn),
        Append(p.methods,
              [method, qualifiers: public,
               type: ridl2java_type(op.type),
               name: op.name,
               params: ([param, type: ridl2java_type(rt1),
                       name: rt1.name],
                       ... ,
                       [param, type: ridl2java_type(rtn),
                       name: rtn.name]),
               throws: RemoteException],
               body: p_method_body(op,
                                   traversal_names(ptal.class.name,
                                                  ptal.operations,
                                                  op.name))]

    return p
end.

```

```

// p_method_body returns the record representing the body of a method of
// the P class, which corresponds to the given remote operation.
// Input: op: an operation record;
//          Tnames: the traversal names for the return and argument objects;
//          traversal names may be null.
// Output: the record representing the body of a P method.
function p_method_body(op oftype operation, Tnames oftype list of field_ref)
  given op.params ≡ (rt1, ..., rtn), Tnames ≡ (tret, ta1, ..., tan),
  let calltoreal = [invocation, obj: myself, meth: op.name,
                  args: (java2ridl_obj(rt1), ...,
                        java2ridl_obj(rtn))]

  // Process the return type
  if op.type.name == void, // no return object.
    // simply generate the call the real object.
    return calltoreal
  else if op.type.mode == gref, // return object is sent by gref.
    // return the p reference of the return of
    // the call to the real object
    return [return, expr: [field_ref, objname: calltoreal, field: _p]]
  else // return object is sent by copy.
    // return a new DArgument having as parameters the return of the
    // call to the p object and the traversal name
    return [return, expr: [new, class: DArgument, args: (calltoreal, tret)]]
end.

// generate_pp returns the record of the class used to instantiate PP
// objects associated with D remote objects.
// Input: ptal: a portal record.
// Output: the record corresponding to the class used to instantiate PPs
function generate_pp (ptal oftype portal)
  let pp = [class, name: Concat(ptal.class.name, PP)] in
    pp.variables := ([vardecl, type: ptal.class.name, name: rself])
    pp.constructors := ([constructor], // the null-ary constructor
                      [constructor, params: ([param,
type:Concat(ptal.class.name,PRI),
                      name: s]
                      body: PPCONSTRUCTORBODY)])

  // iterate over the methods of the class, not over the remote operations
  let aclass = class, methodnames = () in
    while aclass != null
      foreach meth ∈ ptal.class.methods, given meth.params ≡ (p1, ..., pn),
        if Match(meth.name, methodnames)==false // avoid repeating names
          Append(methnames, meth.name) // of overridden methods
          Append(pp.methods,
                [method,qualifiers: public, type: meth.type, name: meth.name,
                 params: ([param, type: p1.type, name: p1.name], ... ,
                         [param, type: pn.type, name: pn.name])
                 body: pp_method_body(meth, ptal.operations,
                                     traversal_names(ptal.class.name,
                                                     ptal.operations,
                                                     op.name))])

      aclass := LookupClass(aclass.super)
    return pp
end.

```

```

// pp_method_body returns the record representing the body of a method of
// the PP class.
// Input: meth: a method record representing a method from a JCore class;
//         remoteops: a list of remote operation records;
//         Tnames: the traversal names for the return and argument objects;
//         traversal names may be null.
// Output: the record representing the body of a PP method.
function pp_method_body (meth oftype method,
                        remoteops oftype list of operation,
                        Tnames oftype list of field_ref)
  if Match(meth.name, Names(remoteops)) == false
    return INVALIDREMOTEOPERATION
  else
    let op = get_operation_from_name(meth.name, remoteops) in
      return [try, body: remote_call(op, Tnames),
             catches: ([catch, exception: RemoteException,
                       body: catch_body(op.type.name)]]]
end.

// remote_call returns a record representing the call from the PP object to
// its counterpart P object.
// Input: op: an operation record;
//         Tnames: the traversal names for the return and argument objects;
//         traversal names may be null.
// Output: the call from the pp to the p.
function remote_call (op oftype operation, Tnames oftype list of field_ref)
  given op.params ≡ (rt1, ..., rtn), Tnames ≡ (tret, ta1, ..., tan),
  let calltop = [invocation, obj: rself, meth: op.name,
                args: (ridl2java_obj(rt1, ta1), ... ,
                      ridl2java_obj(rtn, tan))]

  // Process the return type.
  if op.type.name == void, // no return object.
    // simply generate the call the P object
    return calltop
  else if op.type.mode == gref, // return object is passed by gref.
    // return the proxy to remote object of
    // the return of the call
    return [return, expr: [new, class: op.type.name,
                          args:([new, class: Concat(op.type.name,PP),
                                args: calltop]]]
  else // return object was passed by copy.
    // This method gets a DArgument back from the remote call to P.
    // Extract the object.
    return [return, expr: [field_ref, obj: calltop, field: obj]]
end.

function catch_body (typename oftype symbol)
  let s = [sequence, statements: (ERRORMESSAGE)] in
    // if the return type of the method is not void, we need to return
    // something when a RemoteException is thrown.
    if typename ≠ void
      Append(s.statements, [return, expr: NullValueForType(type)]
    return s
end.

```

```

// generate_traversals returns the record representing the class that
// contains the repository of traversals associated with a RIDL
// portal.
// Input: ptal: a portal record representing a RIDL portal.
// Output: a record representing the traversals class, or null if there are
// no traversal specifications in ptal.
function generate_traversals (ptal oftype portal)
  let vars = (), counter = 0 in
    // name the traversals sequentially as they appear in the portal
    foreach op ∈ ptal.operations,
      foreach type ∈ {op.type} ∪ Types(op.params),
        if is_primitive(type.name) == false and
          type.mode == copy and type.traversal ≠ null,

          Append(vars, [vardecl, qualifiers: static, type: Traversal,
            name: Concat(t, ToString(counter))])

          counter := counter+1
    if counter = 0, return [] // there are no traversals in r
    else
      let traversalclass = [class, name: Concat(ptal.class.name, Traversals),
        variables: vars] in
        Append(traversalclass.variables, THEONCEBOOLEAN)
        traversalclass.methods := ([method,
          qualifiers: public static synchronized,
          type: void, name: init,
          body: traversals_init_body(ptal)])

      return traversalclass
end.

// traversals_init_body returns a record representing the body of the
// method for instantiating the traversals associated with a portal.
// The statements in the return record are a translation of the traversals
// in RIDL.
// Input: ptal: a portal record representing the RIDL portal
// Output: a record representing the body of the init method
function traversals_init_body (ptal oftype portal)
  let s = [sequence, statements:(THEINCOMPLETECLASSVARDECL, THEONCETEST)],
    counter = 0, ctname = Concat(ptal.class.name, Traversals) in
    // the traversals were named sequentially as they appear in the portal
    foreach op ∈ ptal.operations,
      foreach type ∈ {op.type} ∪ Types(op.params),
        if is_primitive(type.name) == false and
          type.mode == copy and type.traversal ≠ null,
          let tname = Concat(t, ToString(counter)) in
            Append(s.statements, [assignment, left: tname,
              right: [new, class: Traversal,
                args: (tname, ctname)])

            foreach iclass ∈ type.traversals.incompletes,
              Append(s.statements, [assignment, left: c,
                right: [new, class: IncompleteClass,
                  args: iclass.name]])

            foreach part ∈ iclass.missing,
              Append(s.statements, [invocation, obj: c, meth: bypass,
                args: part])
              Append(s.statements, [invocation, obj: tname,
                methname: incompleteClass,
                args: (iclass)])

            counter := counter+1
    Append(s.statements, THEONCEASSIGNMENT)
  return s
end.

```

```

// traversal_names returns a list containing the names of the traversals
// associated with the return and argument objects of a particular operation.
// Input:  classname: the name of the class associated with the portal
//          that is being translated.
//          operations: a list of operation records representing the remote
//                    operations of a portal;
//          opname: the operation name.
// Output: a list of traversal names. The size of this list is exactly the
//          number of arguments of the operation plus one (return value). The
//          first traversal corresponds to the return value, the second to the
//          first argument, etc.
function traversal_names (classname oftype symbol,
                        operations oftype list of operation,
                        opname oftype symbol)
  if Match(opname, Names(operations)) == false // no such remote operation.
    return null
  else
    let the_op = get_operation_from_name(opname, remoteops) in
      let counter = objs_passed_by_copy_until_op(operations, the_op),
          traversalnames = () in
        foreach rtype ∈ {op.type} ∪ Types(op.params),
          if is_primitive(rtype.type) == false and
             rtype.mode == copy and type.traversal ≠ null
            Append(traversalnames,
                  [field_ref, obj: Concat(classname, Traversals),
                   field: Concat(t, ToString(counter))])
            counter := counter+1
          else
            Tnames ← (null)
  end.

// The next 3 functions convert between Java types/objects and
// the types/objects used by the DJ library
function ridl2java_type (rtype oftype ridl_type)
  if is_primitive(rtype.type) == true return rtype.type
  else if rtype.mode == gref return Concat(rtype.type, RI)
  else return DArgument
end.

// Java2RIDL_arg converts arguments from the lower DJ library format
// to the source format
function java2ridl_obj (rtype oftype ridl_type)
  if is_primitive(rtype.type) == true // simply return the argument name
    return rtype.name
  else if rtype.mode == gref // it's a P. Instantiate a local proxy
    return [new, class: rtype.type,
            args: ([new, class: Concat(rtype.type, PP)
                  args: rtype.name])]
  else // it's a DArgument. Extract the internal object.
    return [field_ref, obj: rtype.name, field: obj]
end.

// RIDL2Java_arg converts arguments from the source format to the
// lower DJ library format
function ridl2java_obj (rtype oftype ridl_type, tname oftype symbol)
  if is_primitive(rtype.type) == true // simply return the argument name
    return rtype.name
  else if rtype.mode == gref // return the P object associated with argument
    return [field_ref, obj: rtype.name, field: _p]
  else // Build DArgument from argument and traversal.
    return [new, class: DArgument, args: (rtype.name, tname)]
end.

```

