A Description of Cluster Code
Generated by the Durra Compiler

Dennis L. Doubleday
Michael J. Gardner
Charles B. Weinstock

December 1991
A Description of Cluster Code Generated by the Durra Compiler

Dennis L. Doubleday
Michael J. Gardner
Charles B. Weinstock

Distributed Systems Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
Table of Contents

1 Introduction 1

2 Standard Template 3

3 Cluster-Specific Code Fragments 9
   3.1 With Clauses for User Procedures 9
   3.2 With Clauses for Channel Packages 9
   3.3 Enumeration of Channels Used 9
   3.4 Link_Task_Info Case Alternatives 10
   3.5 Instantiations of Process_Shell 10
   3.6 Task Object Declarations for Each Imported Procedure 10
   3.7 Start_Link_Process Case Alternatives 11
   3.8 Reconfiguration Trigger Functions 11
   3.9 Level Configuration Procedures 12
   3.10 Get_Port Case Alternatives 12
   3.11 Get_Port_Return Case Alternatives 13
   3.12 Send_Port Case Alternatives 13
   3.13 Send_Port_Return Case Alternatives 13
   3.14 Test_Input_Port Case Alternatives 13
   3.15 Test_Output_Port Case Alternatives 14
   3.16 Case Alternative for Each Configuration Level 14
   3.17 Type_Table Entries 16
   3.18 Process_Table Entries 17
   3.19 Process Attribute Value Assignments 18
   3.20 Link_Table Entries 18
   3.21 Link Attribute Value Assignments 19
   3.22 Level_Table Entries 19
   3.23 Cluster_Table Entries 21

References 23
A Description of Cluster Code Generated by the Durra Compiler

Abstract: Durra is a language and support environment for the specification and execution of distributed Ada applications. The Durra programmer specifies the distribution of application components by assigning them to virtual nodes called clusters. For each cluster named in an application description, the Durra compiler generates an Ada package body with a standardized format. Within the confines of the format, the content of the package body varies according to the requirements placed upon the cluster by the Durra application description. The cluster-specific package body is compiled and linked with a fixed set of Ada compilation units, common to all clusters, to form a multitasking Ada program. The intended audience for this document is Durra application developers, who will need an understanding of the concepts presented here in order to be effective Durra application debuggers.

1 Introduction

The Durra language [1] is a task-level application description language. The basic building blocks of the language are the task description, which specifies the properties of an associated Ada subprogram or subsystem, and the channel description, which specifies the properties of an Ada package implementing a communication facility. Task descriptions may be either primitive or compound. A primitive task description represents a single thread of control. A compound task description is a composition of other task and channel descriptions. Channel descriptions are syntactically similar to primitive task descriptions although the implementations exhibit different behaviors. Task implementations are active components; they initiate requests to send or receive messages by calling procedures provided by the runtime environment. Channel implementations are passive components; they wait for and respond to requests from the runtime environment.

A Durra programmer describes an application as a collection of processes (instances of Durra task descriptions) connected to each other in a graph structure by links (instances of channel descriptions). Lower level components are used as building blocks for higher-level task descriptions. Application descriptions are simply compound task descriptions that describe a complete application.

A component's input/output interface is specified by the ports section of its description. Ports are named, unidirectional, locally-defined conduits through which processes may transmit/receive data. Ports have a Durra data type associated with them to allow semantic checking of intercomponent port connections.

1. Throughout this document, the term task refers to a generalized "thread of control" concept rather than to the analogous Ada construct, except where noted.
2. The actual Ada code that implements a Durra task may, in fact, be a multitasking program. However, from the Durra perspective the program is a single thread of control.
A compound task description must include additional information about its structure. Its component processes and links are defined in its *components* section and the manner in which they are logically connected (which may vary dynamically) is specified in its *structures* section. If the structure of the compound task is allowed to vary, then there must be a *reconfigurations* section that describes a set of structural changes and the conditions under which the changes will occur. The *clusters* section specifies the physical grouping of components into executable images, which may well be orthogonal to the logical connections described in the *structures* section.

If the compound description is a complete application description, then the Durra compiler generates an Ada package body with a standardized format for each cluster defined in the application. Within the confines of the format, the content of the package body (called *Tables*) varies according to the requirements placed upon the cluster by the Durra application description. The cluster-specific *Tables* package body imports the implementations of the components assigned to the cluster, creates Ada tasks to serve as threads for the Durra process implementations, and creates instances of the Ada task types that implement the Durra links assigned to the cluster. The package body contains a set of subprograms that route inter-task communications; a set of subprograms that evaluate reconfiguration conditions, if any; and a set of Ada tasks that together control the runtime configuration of Durra processes assigned to the cluster. The package body also defines a set of tables, common to all clusters in the application, that describe the complete application structure. The *Tables* package body is compiled and linked with a fixed set of Ada compilation units, common to all clusters, to form a multitasking Ada program. If only one cluster is specified in the application description, then this program is the complete application implementation. Otherwise, the application is distributed and the cluster program will communicate at runtime with other cluster programs.

Each generated *Tables* package body will consist of two parts:

1. A standard template that is constant across all applications.
2. A cluster-specific part that is distributed throughout the standard template.

In the specifications in the following sections, program text comprising the standard template appears in *boldface italic*, while program text that is included for sample purposes but will vary with the application appears in *italic*. Text surrounded by the "<>" character pair is a placeholder for actual program text. If expansion and explanation of a placeholder is required, the placeholder refers to a subsequent section of this document. Program text lines beginning with the string "- -" are simply commentary.
2 Standard Template

The following code comprises the standard template for the Tables package body. Placeholders in the template substitute for cluster-specific code fragments which will be described in later sections.

```ada
with Configuration_Manager;
with Durra_Interface;
with Process_Shell;
with Storage_Manager;
with String_Pkg;
with Text_IO;
-- packages Calendar, System, Durra_Interface.Types, and Table.Types
-- are "withed" by the package specification

package body Tables is

-- <comment indicating version of code generator used to create this package body>

package CM renames Configuration_Manager;
package SM renames Storage_Manager;
package SP renames String_Pkg;
package DI renames Durra_Interface;

-- package TT renames Table.Types in package specification
-- package DT renames Durra_Interface.Types in package specification

---------------
-- TYPES
---------------
type Channel_Types is (<enumeration of Channels used (see section 3.3 on page 9)>);
type Link_Task_Info (Channel_Type : Channel_Types) is
  record
    case Channel_Type is
    <case alternative for each value of Channel_Type (see section 3.4 on page 10)>
    end case;
  end record;
type Link_Task_Ptr is access Link_Task_Info;
type Link_Task_Index is array (TT.LnkID Range range <>) of Link_Task_Ptr;

---------------
-- OBJECTS
---------------
```
Link_Task_Table : Link_Task_Index(1..<n>);
-- where n = number of links defined in the application

<Instantiation of Process_Shell for each user procedure in the cluster
(see section 3.5 on page 10)>

<Task object declaration for each Durra process assigned to the cluster
(see section 3.6 on page 10)>

_****************************************
-- LOCAL SUBPROGRAMS
_****************************************

procedure Start_Link_Process (Channel_Type : in Channel_Types;
                            The_Link : in TT.Link_Table_Ptr) is

begin
  case Channel_Type is
    <case alternative for each value of Channel_Type (see section 3.7 on page 11)>
  end case;
  The_Link.Initialized := TRUE;
end Start_Link_Process;

-- Assume level numbers run from 1..n
<A set of functions with names of the form Lx, where x is in the range 0..n.
These functions are used to determine which level to go to next, and when to do it.
The 0th level is equivalent to ENTER.
(see section 3.8 on page 11)>

<A set of procedures with names of the form Configure_Level_x where x is in the range 1..n+1.
These functions are used to configure to a particular level and to start and stop
relevant processes and links. The n+1 level is EXIT.
(see section 3.9 on page 12)>

_****************************************
-- VISIBLE SUBPROGRAMS
_****************************************

procedure Get Port(
                       The_Port : in TT.Port_Table_Ptr;
                       Data_Location : in System.Address;
                       Data_Size : out NATURAL;
                       Data_Type_ID : out DT.Type_ID_Range_Plus_Null;
                       Completed : out BOOLEAN) is

begin
  case Link_Task_Table(The_Port.Associated_Link.ID).Channel_Type Is

CMU/SEI-91-TR-19
<case alternative for each value of Channel_Type (see section 3.10 on page 12)>
end case;
end Get_Port;

procedure Get_Port_Return(
    The_Port : In TT.Port_Table_Ptr;
    Size_of_Data : out NATURAL;
    Type_ID : out DT.Type_ID_Range_Plus_Null)
begin
    case Link_Task_Table(The_Port.Associated_Lnk.ID).Channel_Type Is
        <case alternative for each value of Channel_Type (see section 3.11 on page 13)>
    end case;
end Get_Port_Return;

procedure Send_Port(
    The_Port : In TT.Port_Table_Ptr;
    Data_Location : In System.Address;
    Data_Size : In POSITIVE;
    Data_Type_ID : In DT.Type_ID_Range_Plus_Null;
    Completed : out BOOLEAN;
    Priority : In DT.Message_Priority_Range :=
        DT.NULL_MESSAGE_PRIORITY)
begin
    case Link_Task_Table(The_Port.Associated_Lnk.ID).Channel_Type Is
        <case alternative for each value of Channel_Type (see section 3.12 on page 13)>
    end case;
end Send_Port;

procedure Send_Port_Return(The_Port : In TT.Port_Table_Ptr) Is
    Link_Task : Link_Task_Ptr;
begin
    case Link_Task_Table(The_Port.Associated_Lnk.ID).Channel_Type Is
        <case alternative for each value of Channel_Type (see section 3.13 on page 13)>
    end case;
end Send_Port_Return;
procedure Test_Input_Port
    ThePort : in TT.Port_Table_Ptr;
    Type_of_Next_Input : out DT.Type_ID_Range_Plus_Null;
    Size_of_Next_Input : out NATURAL;
    Inputs_Available : out NATURAL) is
begin
    Link_Task_Table(ThePort.Associated_Link.ID).Channel_Type is
        <case alternative for each value of Channel_Type (see section 3.14 on page 13)>
    end case;
end Test_Input_Port;

procedure Test_Output_Port
    ThePort : in TT.Port_Table_Ptr;
    Slots_Available : out NATURAL) is
begin
    case Link_Task_Table(ThePort.Associated_Link.ID).Channel_Type is
        <case alternative for each value of Channel_Type (see section 3.15 on page 14)>
    end case;
end Test_Output_Port;

-- VISIBLE TASKS

task body State_Changer is
    Done : BOOLEAN := FALSE;
    Next_Level_ID : INTEGER := 0;
    Prev_Level_ID : INTEGER := 0;
begin
    accept start;
    while not Done loop
        if Cluster_Table(This_Cluster_ID).Master then
            -- The master is in charge of reconfiguration decisions
            case Next_Level_ID is
                when 0 => Next_Level_ID := L0(0); -- This is the entry condition
                when <1..n+1> =>
                    <One case alternative for each configuration level in the application and one additional alternative for termination (see section 3.16 on page 14)>
                when others =>
                    Text_IO.Put_Line("Illegal reconfiguration to Level" ")
Level_ID_Range'IMAGE(Next_Level_ID) & " requested.");

end case;
else
  -- Non-masters respond to requests from the master
  accept Reconfigure(To_Level : in NATURAL) do
    Prev_Level_ID := Next_Level_ID;
    Next_Level_ID := To_Level;
  end Reconfigure;
  case Next_Level_ID is
    when 0 => null;
    when <1..n+1> =>
      <One case alternative for each configuration level in the applications and one additional alternative for termination (see section 3.16 on page 14)>
    when others =>
      Text_IO.Put_Line("Illegal reconfiguration to Level" &
                       TT.Level_ID_Range'IMAGE(Next_Level_ID) & " requested.");
  end case;
end if;
end loop;
accept Finish_Up;
end State_Changer;

_-----------------------------------------------------------------
-- TABLE DEFINITIONS _-----------------------------------------------------------------
begin

This_Cluster_ID := <index of this cluster in the Cluster_Table>;
The_Master := <index of the master in the Cluster_Table>;

Type_Table :=
  new TT.Type_Index'(   
    <one Type_Table_Entry for each data type defined in the application
    (see section 3.17 on page 16)>
  );

_-----------------------------------------------------------------

Process_Table :=
  new TT.Process_Index'(   
    <one Process_Table_Entry for each process in the application
    (see section 3.18 on page 17)>
  );

<series of assignments to the Attributes field of any Process containing attribute values
(see section 3.19 on page 18)>
Link_Table :=
    new TT.Link_Index'(
        <one Link_Table_Entry for each link in the application
            (see section 3.20 on page 18)>
    );

<series of assignments to the Attributes field of any Link containing attribute values
    (see section 3.21 on page 19)>

-- ******************************************************

Level_Table :=
    new TT.Level_Index'(
        <one Level_Table_Entry for each configuration level in the application
            (see section 3.22 on page 19)>
    );

-- ******************************************************

Cluster_Table :=
    new TT.Cluster_Index'(
        <one Cluster_Table_Entry for each cluster in the application
            (see section 3.23 on page 21)>
    );

end Tables;
3 Cluster-Specific Code Fragments

This section describes the cluster-specific code fragments referred to by the placeholders in the standard template description above.

3.1 With Clauses for User Procedures

There must be a "with" clause for every Ada procedure named in the Durra application description as the implementation of a Durra process assigned to the cluster. An Ada procedure is the implementation of a Durra process if it is named in the procedure_name attribute for that process.

Example:

with Producer;
with Consumer;
with Console;
etc.

3.2 With Clauses for Channel Packages

There must be a "with" clause for every Ada package named in the Durra application description as the implementation of a Durra link assigned to the cluster. An Ada package is the implementation of a Durra link if it is named in the package_name attribute for that link.

Example:

with FIFO_Channel;
with Broadcast_Channel;
etc.

3.3 Enumeration of Channels Used

Each channel package named in a "with" clause must have a corresponding enumeration literal in type Channel_Types. The enumeration literal name is the name of the package with the suffix "_Type" appended.

Example:

type Channel_Types is (FIFO_Channel_Type, Broadcast_Channel_Type);
3.4 **Link_Task_Info Case Alternatives**

In the definition of the Link_Task_Info record, there must be a case alternative for each literal in the enumerated type Channel_Types. The form of the case alternative must be:

```ada
when <Channel_Type value=>
    <channel package name>_Link : <channel package name>.Channel_Task;
```

Example:

```ada
when FIFO_Channel_Type =>
    FIFO_Channel_Link : FIFO_Channel.Channel_Task;
```

3.5 **Instantiations of Process_Shell**

*Process_Shell* is the name of a generic package supplied with the Durra runtime library. This package exports an Ada task type that serves as a "wrapper" around the Ada subprogram named as its actual parameter. There must be an instantiation of *Process_Shell* for each user procedure named in a "with" clause. The form of the instantiation must be:

```ada
package <Ada procedure name>_Shell is new Process_Shell(<Ada procedure name>);
```

Example:

```ada
package Producer_Shell is new Process_Shell(Producer);
package Consumer_Shell is new Process_Shell(Consumer);
package Console_Shell is new Process_Shell(Console);
```

3.6 **Task Object Declarations for Each Imported Procedure**

There must be an Ada task object declaration for each Durra process assigned to the cluster. The object declaration for each process must refer to the instantiation of *Process_Shell* associated with the Ada procedure that implements the Durra process. The process ID of a process is its index in the *Process_Table* (see section 3.18 on page 17).

```ada
Process_<process ID> : <Ada procedure name>_Shell.Caller;
```

Example:

```ada
Process_1 : Producer_Shell.Caller;
Process_2 : Consumer_Shell.Caller;
Process_3 : Console_Shell.Caller;
Process_4 : Producer_Shell.Caller;
```

etc.
3.7 **Start_Link_Process Case Alternatives**

In the body of the `Start_Link_Process` subprogram, there must be a case alternative for each literal in the enumerated type `Channel_Types`. The form of the case alternative must be:

```plaintext
when <Channel_Type value> =>
    Link_Task_Table(The_Link.ID) := new Link_Task_Info(<Channel_Type value>);
    Link_Task_Table(The_Link.ID).<channel package name>_Link.Initialize(The_Link);
```

Example:

```plaintext
when FIFO_Channel_Type =>
    Link_Task_Table(The_Link.ID) := new Link_Task_Info(FIFO_Channel_Type);
    Link_Task_Table(The_Link.ID).FIFO_Channel_Link.Initialize(The_Link);
```

3.8 **Reconfiguration Trigger Functions**

For each configuration level specified in the application description there is a function which determines which level to go to next, and when to go to it. An additional function for the initial level (level 0), a hidden level not specified in the application description, determines when and at what level to start the application.

```plaintext
function L<Level ID>(Prev: In NATURAL) return NATURAL Is
    theDelta : DURATION := DURATION'LAST;
    aDelta : DURATION;
begin
    loop
    -- A series of statements that evaluate reconfiguration expressions at this level.
    -- When an expression evaluates true the function returns the new level to go to.
    -- The expressions take the following form:
    if <reconfiguration condition> then
        return <new Level ID>;
    end if;
    -- A series of statements that determine the next time the expressions should be
    -- evaluated in the absence of a signal. The expressions take the form:
    aDelta := <duration evaluation>;
    if aDelta < theDelta then
        theDelta := aDelta;
    end if;
    CM.Reconfiguration_Configuration_Task.Start(theDelta);
    CM.Reconfiguration_Configuration_Task.Check;
    end loop;
end L<Level ID>;
```
3.9 Level Configuration Procedures

For each configuration level there is a procedure that actually carries out the steps necessary to configure for that level. For all but the termination level (the level entered when an application is about terminate) the procedure is of the form:

```
procedure Configure_Level_<Level ID> is
begin
CM.Do_Level_Configuration(<Level ID>);
<A series of statements that start processes and links in the configuration. They take on the form>
If This_Cluster_ID = Link_Table(<x>).Cluster_ID and then
    not Link_Table(<x>).Initialized then
    Start_Link_Process(
        <link_type>,
        Link_Table(<x>));
end if;
If This_Cluster_ID = Process_Table(<y>).Cluster_ID and then
    not Process_Table(<y>).Initialized then
    Process_<z>.Start(<y>);
end if;
end Configure_Level_<Level ID>;
```

For the termination level (level n+1), the procedure has the form:

```
procedure Configure_Level_<Level ID> is
begin
CM.Do_Level_Configuration(<Level ID>);
-- A series of statements that stop processes in the configuration.
-- They take the form:
If This_Cluster_ID = Process_Table(<y>).Cluster_ID and then
    Process_Table(<y>).Initialized then
    Process_<z>.Stop;
end if;
end Configure_Level_<Level ID>;
```

3.10 Get_Port Case Alternatives

In the body of the Get_Port subprogram, there must be a case alternative for each literal in the enumerated type Channel_Types. The form of the case alternative must be:

```
when <Channel_Types value> =>
    Link_Task_Table(The_Port.Assoclated_Link.ID).
        <channel package name>_Lnk.Get_Port(
            The_Port,
            Data_Location,
            Data_Size,
```
3.11 **Get_Port_Return** Case Alternatives

In the body of the Get_Port_Return subprogram, there must be a case alternative for each literal in the enumerated type Channel_Types. The form of the case alternative must be:

```
when <Channel_Types value> =>
    Link_Task_Table(The_Port.Associated_Link.ID).<channel package name>_Link.
    Get_Port_Return(The_Port.Connection_Point)(Size_of_Data, Type_ID);
```

3.12 **Send_Port** Case Alternatives

In the body of the Send_Port subprogram, there must be a case alternative for each literal in the enumerated type Channel_Types. The form of the case alternative must be:

```
when <Channel_Types value> =>
    Link_Task_Table(The_Port.Associated_Link.ID).
    The_Port, Data_Location, Data_Size, Data_Type_ID, Completed, Priority;
    <channel package name>_Link.Send_Port(  
```

3.13 **Send_Port_Return** Case Alternatives

In the body of the Send_Port_Return subprogram, there must be a case alternative for each literal in the enumerated type Channel_Types. The form of the case alternative must be:

```
when <Channel_Types value> =>
    -- compiler bug requires this workaround
    Link_Task := Link_Task_Table(The_Port.Associated_Link.ID);
    Link_Task.<channel package name>_Link.
    Send_Port_Return(The_Port.Connection_Point);
```

3.14 **Test_Input_Port** Case Alternatives

In the body of the Test_Input_Port subprogram, there must be a case alternative for each literal in the enumerated type Channel_Types. The form of the case alternative must be:

```
when <Channel_Types value> =>
    Link_Task_Table(The_Port.Associated_Link.ID).
```
3.15 **Test Output Port Case Alternatives**

In the body of the Test Output Port subprogram, there must be a case alternative for each literal in the enumerated type `Channel Types`. The form of the case alternative must be:

```plaintext
when <Channel Types value> =>
    Link_Task_Table(The_Port.Associated_Link.ID)._Link._Link_Test_Output_Port(The_Port, Slots_Available);
```

3.16 **Case Alternative for Each Configuration Level**

There are two case statements in the body of the State Changer task. The first is executed by the master cluster and the second is executed by all other clusters. For each configuration level in the application there must be a case alternative in both case statements. The case alternative choice is the ID of the configuration level, which is its index in the `Level Table`. There are also alternatives for ENTER (alternative 0) and EXIT (alternative n+1). For the master, the form of the alternative is as follows:

```plaintext
when <level ID> =>
    CM.Reconfiguration_Task.Configure_to_Level(<level ID>);
    accept Reconfigure(To_Level : in NATURAL);
    Configure_Level_<level ID>;
    Next_Level_ID := L<level ID>(Prev_Level_ID);
    Prev_Level_ID := <level ID>:

The ENTER alternative is:

```plaintext
when 0 => Next_Level_ID := L0(0);
```

The EXIT alternative is:

```plaintext
when <n+1> =>
    CM.Reconfiguration_Task.Configure_to_Level(<n+1>);
    accept Reconfigure(To_Level : in NATURAL);
    Configure_Level_<n+1>;
    Done := TRUE;
```

For the non-master clusters, the form of all alternatives except the ENTER alternative is as follows:

```plaintext
when <level ID> => Configure_Level_<level ID>:
```
The *ENTER* alternative for non-master clusters is:

```plaintext
when 0 => null;
```

Example:

In the following example, there are two application levels, Level 1 and Level 2. Level 0 is the *ENTER* level and Level 3 is the *EXIT* level.

```plaintext
if Cluster_Table(This_Cluster_ID).Master then
  case Next_Level_ID is
    when 0 => Next_Level_ID := L0(0);

    when 1 =>
      CM.Reconfiguration_Task.Configure_to_Level(1);
      accept Reconfigure(To_Level : in NATURAL);
      Configure_Level_1;
      Next_Level_ID := L1(Prev_Level_ID);
      Prev_Level_ID := 1;

    when 2 =>
      CM.Reconfiguration_Task.Configure_to_Level(2);
      accept Reconfigure(To_Level : in NATURAL);
      Configure_Level_2;
      Next_Level_ID := L2(Prev_Level_ID);
      Prev_Level_ID := 2;

    when 3 =>
      CM.Reconfiguration_Task.Configure_to_Level(3);
      accept Reconfigure(To_Level : in NATURAL);
      Configure_Level_3;
      Done := TRUE;

    when others =>
      Text_IO.Put_Line("Illegal reconfiguration to Level" &
                       "TT.Level_ID_Range'IMAGE(Next_Level_ID) & "requested.");
  end case;
else
  accept Reconfigure(To_Level : in NATURAL) do
    Prev_Level_ID := Next_Level_ID;
    Next_Level_ID := To_Level;
  end Reconfigure;

  case Next_Level_ID is
    when 0 => null;
```
when 1 => Configure_Level_1;
when 2 => Configure_Level_2;
when 3 => Configure_Level_3;
when others =>
  Text_IO.Put_Line("Illegal reconfiguration to Level" &
          TT.Level_ID.Range'IMAGE(Next_Level_ID) & "requested.");
end case;
end if;

3.17 Type_Table Entries

There must be an entry in the Type_Table for each Durra type defined in the application description. The Type_Table_Entry assignment shall have the form:

<sequential index n, starting at 1> => new TT.Type_Table_Entry’(
   Kind => <TT.Size_Type or TT.Union_Type>,
   Name => SP.Make_Persistent("<Durra type name>"),
   ID => <n>,
   Free_List => <if the type has a fixed upper bound, then
                 SM.Create_Free_List(<upperbound/8>),
                 else null,>
   Bytes_Required => <maximum size of data, or 0 if unbounded>,
   -- if Kind is Size_Type then
   Upper_Bound => <upper bound from type definition>,
   Lower_Bound => <lower bound from type definition>
   -- elsif Kind = Union_Type then
   Component_Types => null
)

Example:

1 => new TT.Type_Table_Entry’(
   Kind => TT.Size_Type,
   Name => SP.Make_Persistent("GENERAL"),
   ID => 1,
   Free_List => SM.Create_Free_List(4),
   Bytes_Required => 4,
   Upper_Bound => 32,
   Lower_Bound => 32
)
3.18 Process_Table Entries

There must be an entry in the Process_Table for each process defined in the application description. The Process_Table_Entry assignment shall have the form:

<sequential index n, starting at 1> => new TT.Process_Table_Entry(')
    Name => SP.Make_Persistent("<expanded Durra process name>"),
    ID => <n>,
    Start_Time => DT.NULL_TIME,
    Initialized => FALSE,
    Ports => new TT.Port_Index(')
        -- for each port defined for this process, one Port_Table_Entry
        <sequential index m, starting at 1> => TT.new Port_Table_Entry(')
            Name => SP.Make_Persistent("<simple/expanded Durra port name>"),
            ID => <m>,
            Owner_Process => null,
            Data_Type => Type_Table(<ID of data type for this port>),
            Associated_Link => null,
            Connection_Point => DT.Port_ID_Range.LAST,
            Is_Input => <FALSE or TRUE, depending on port direction>
        )
    ),
    Attributes => TT.Attribute_Pairs.Create,
    Blocked => FALSE,
    Blocked_Data_Buffer => System.NO_ADDR,
    Blocked_Data_Size => 0,
    Blocked_Data_Type_ID => NULL_TYPE_ID,
    Remote_Data => null,
    Cluster_ID => 0
    )

Example:

1 => new Process_Table_Entry(')
    Name => SP.Make_Persistent("MAIN.P1"),
    ID => 1,
    Start_Time => DT.Null_Time,
    Initialized => FALSE,
    Ports => new TT.Port_Index(')
        1 => new TT.Port_Table_Entry(')
            -- Port MAIN.P1.OUT1
            Name => SP.Make_Persistent("OUT1"),
            ID => 1,
            Owner_Process => NULL,
            Data_Type => Type_Table(1),
            Associated_Link => NULL,
            Connection_Point => DT.Port_ID_Range.LAST,
            )
Is_Input => FALSE
}
),
Attributes => TT.Attribute_Pairs.Create,
Blocked => FALSE,
Blocked_Data_Buffer => System.NO_ADDR,
Blocked_Data_Size => 0,
Blocked_Data_Type_ID => DT.NULL_TYPE_ID,
Remote_Data => null,
Cluster_ID => TT.Cluster_ID_Range'LAST
)

3.19 Process Attribute Value Assignments

For each attribute of each process defined in the application description, there must be an assignment of the form:

```
TT.Attribute_Pairs.Append
(Element => (SP.Make_Persistent("<simple attribute name>"),
            SP.Make_Persistent("<attribute value>")),
    L => Process_Table(<process_table_index>).Attributes);
```

Example:

```
TT.Attribute_Pairs.Append
(Element => (SP.Make_Persistent("cluster"),
            SP.Make_Persistent("cl1")),
    L => Process_Table(1).Attributes);
```

3.20 Link_Table Entries

There must be an entry in the Link_Table for each link defined in the application description. The Link_Table_Entry assignment shall have the form:

```
<sequential index n, starting at 1> => new TT.Link_Table_Entry'
    Name => SP.Make_Persistent(<expanded Dura link name>),
    ID => <n>,
    In_Ports => new TT.Port_Index'
        -- one entry for each input port defined for this link
        <sequential index m, starting at 1> => null
    ),
    Out_Ports => new TT.Port_Index'
        -- one entry for each output port defined for this link
        <sequential index j, starting at 1> => null
    ),
    Attributes => TT.Attribute_Pairs.Create,
```

CMU/SEI-91-TR-19
**BufferSize** => <message buffer bound specified for link>,  
**Cluster_ID** => TT.Cluster_ID_Range'LAST

Example:

```lisp
1 => new TT.Link_Table_Entry'(
   Name => SP.Make_Persistent("MAIN.Q1"),
   ID => 1,
   In_Ports => new TT.Port_Index' (1 => null),
   Out_Ports => new TT.Port_Index' (1 => null),
   Buffer_Size => 10,
   Cluster_ID => TT.Cluster_ID_Range'LAST
)
```

### 3.21 Link Attribute Value Assignments

For each attribute of each link defined in the application description, there must be an assignment of the form:

```lisp
TT.Attribute_Pairs.Append
   (Element => (SP.Make_Persistent("<simple attribute name>"),
                SP.Make_Persistent("<attribute value>")),
   L => Link_Table(<link_table_index>).Attributes);
```

Example:

```lisp
TT.Attribute_Pairs.Append
   (Element => (SP.Make_Persistent("cluster"),
                SP.Make_Persistent("cl1")),
   L => Link_Table(1).Attributes);
```

### 3.22 Level_Table Entries

There must be an entry in the Level_Table for each configuration level defined in the application description. The Level_Table_Entry assignment shall have the form:

```lisp
<sequential index n, starting at 1> => new TT.Level_Table_Entry'(
   Number_of_Processes => <number of processes in Process_Table>,
   Number_of_Links => <number of links in Link_Table>,
   Number_of_Connections => <number of connection records for this level>,
   Name => SP.Make_Persistent(<expanded name of configuration level>),
   ID => <n>,
   Processes => (  
      -- for each process in the Process_Table
      <sequential index m, starting at 1> =>
   )
```
(The_Process => Process_Table(<m>),
Cluster_ID => <cluster ID or 0 if not active at this level>
)

),

Links => (  
  -- for each link in the Link_Table  
  <sequential index j, starting at 1> =>  
  (The_Li nk => Link_Table(<j>),
  Cluster_ID => <cluster ID or 0 if not active at this level>)  
)

),

Connections => (  
  -- for each port requiring connection at this level  
  <sequential index k, starting at 1> =>  
  (The_Port => Process_Table(<s>).Ports(<t>),
  The_Li nk => Link_Table(<r>),
  Connection_Point => <v>  
  )  
)

)

In the Connections assignment, the values s, t, r, and v vary according to the process ports to be connected, the links to which they are to be connected, and the link port index where the process port is attached to the link. All the variables are positive integer values. The meaning of the field Connection_Point varies with the type of process port being connected. If the port is of type In_Port, then Connection_Point is an index into the Out_Ports index of the Link_Table_Entry; if the port is of type Out_Port, then Connection_Point is an index into the In_Ports index of the Link_Table_Entry.

Example:

1 => new TT.Level_Table_Entry' (  
  Number_of_Processes => 3,  
  Number_of_Links => 1,  
  Number_of_Connections => 3,  
  Name => SP.Make_Persistent("MAIN"),  
  ID => 1,  
  Processes => (  
    1 => (The_Process => Process_Table(1), Cluster_ID => 1),  
    2 => (The_Process => Process_Table(2), Cluster_ID => 1),  
    3 => (The_Process => Process_Table(3), Cluster_ID => 0)  
  ),  
  Links => (  
    1 => (The_Link => Link_Table(1), Cluster_ID => 1)  
  ),  
  Connections => (  
    1 => (The_Port => Process_Table(1).Ports(1),
    The_Link => Link_Table(1),
3.23 Cluster Table Entries

There must be an entry in the Cluster Table for each cluster defined in the application description. The Cluster Table Entry record shall have the form:

<sequential index n, starting at 1> => new TT.Cluster_Table_Entry(' 
  ID => <n>,
  Host_Name => SP.Make_Persistent("<host processor >", or "localhost" if no compile-time configuration file),
  Connected => FALSE,
  Launched => FALSE,
  Command => SP.Make_Persistent('"'),
)

Example:

1 => new TT.Cluster_Table_Entry(' 
  ID => 1,
  Host_Name => SP.Make_Persistent("hx.sei.cmu.edu"),
  Connected => FALSE,
  Launched => FALSE,
  Command => SP.Make_Persistent('"'),
)
References

**A Description of Cluster Code Generated by the Durra Compiler**

Durra is a language and support environment for the specification and execution of distributed Ada applications. The Durra programmer specifies the distribution of application components by assigning them to virtual nodes called clusters. For each cluster named in an application description, the Durra compiler generates an Ada package body with a standardized format. Within the confines of the format, the content of the package body varies according to the requirements placed upon the cluster by the Durra application description. The cluster-specific package body is compiled and linked with a fixed set of Ada compilation units, common to all clusters, to form a multitasking Ada program. The intended audience for this document is Durra application developers, who will need an understanding of the concepts presented here in order to be effective Durra application debuggers.