The Distributed Ada Run-Time System, DARTS

by

M. Carlsson Göthe, D. Wengelin and L. Asplund.
Institute of Physics, Box 530
S-751 21 Uppsala, Sweden.

Abstract

A Distributed Ada Run-Time System, DARTS, is presented. The system can be used in conjunction with a pre-partitioning as well as a post-partitioning paradigm. A single program can be partitioned to run on a loosely coupled multiprocessor system. The distributed units are tasks, task objects, packages, variables, procedures, and functions. Task objects can be dynamically distributed. High fault tolerance is assured by unit redistribution. Design decisions, implementation details and ideas are presented.

INTRODUCTION

Distribution of hardware and software for scientific, industrial, and military computer systems has significantly increased system performance. Several methods, practices, and standards exist for designing distributed hardware, but only a few such improvements have been made in the software area. The effect of the software crisis, the cost of software overwhelming the cost of hardware, is more accentuated for distributed systems in embedded applications than for other types of applications.

Hardware for distributed systems is either tightly coupled or loosely coupled. A system where processors share a common memory is defined as a tightly coupled system. Interprocess communication is performed by shared data structures in the common memory. In a loosely coupled system the memory is localized to each node, and communication is performed over some interconnecting network.

The goal of this project is to develop a distributed Ada run-time system for loosely coupled distributed systems. High fault tolerance is required. The system will be used in a testbed for embedded scientific and military applications.

Partitioning Ada programs.

Partitioning of programs is performed by pre-partitioning or post-partitioning. In pre-partitioning, the software structure is based on the hardware topology [1], [2], [3]. Hence, the software may be designed to improve the fault tolerance without considerable run-time overhead.

In post-partitioning, the design process is divided into two phases: functional design (application) and functional distribution (hardware mapping) [4], [5], [6], [7]. The partitioning information and source code are usually kept separate. Strong requirements are put on the run-time system for efficiency, especially when high fault tolerance is required.

Considerable work has been made to evaluate the proper units of distribution in the Ada language [8]. Five methods may be found; partitioning on Ada programs [9], on tasks [2], on packages [1], [10], on any part of an Ada program [4], [6], or by adding new partitioning rules and mechanisms into the Ada language [11].

Breaking a system into separate programs is the prevailing approach in the design of distributed systems [9], [12]. The method is used for Ada as well as other programming languages. The communication between the various programs is performed by calls to some added IO-packages. As the partitioning decisions are taken early in the system life-cycle, the partitioning tends to become static during the following phases and changes may require redesign of the entire system.

Another shortcoming is that Ada only performs type checking within a program, and no compiler support is given for the compatibility of the data types between programs. Furthermore, Ada defines a rich variety of mechanisms for data-flow and process control within a program. With the use of IO-packages for inter-program communication, we are restricted to subprogram calls.

The partitioning on tasks is widely spread and accepted in the scientific community [2], [13], [14]. The task is the basic structure for concurrency in Ada and therefore suitable for partitioning. The Ada Language Reference Manual [15] states that tasks may execute in a multicomputer environment, but there are only limited language facilities for distributing these tasks on different processors. Furthermore, the task is not a compilation unit and must therefore be encapsulated in a package. The task also lacks the declarative part and must therefore use the encapsulating unit for its declaration purposes. Finally, some severe problems, such as task termination dependencies to non local nodes, will arise.

The partitioning on packages is also widely accepted [1], [10], [16]. Several arguments may be put in favor for this method. First, Ada packages are library units. Also, the package is the main unit of logical program decomposition. However, constraints are often put on the declarations in package interfaces. When distributing on library packages, only minor changes are required to allow for distribution on library subprograms as well [17].

The method of partitioning on any part of an Ada program is developed in the APPL project [18]. The aim is to supply an application independent system that supports the execution in a distributed environment. The functional mapping of the application is described in a separate language, APPL (Ada Program Partitioning Language). The development of a system may start on a uniprocessor and may later, in the final integration phase, be transferred to the distributed target. The APPL approach gives the application no knowledge of the distribution. Hence, all fault tolerance has to be implemented within the underlying run-time system.

Another method for partitioning is to add new mechanisms for partitioning and distribution to the Ada language. Several authors note the absence of abstraction of a virtual node in the language. The required compilation unit should combine the declarative ability of a package and the parallel and stand-alone ability of a task. A suggestion have been made which involves the combination of a main procedure and a package into the compilation unit *partition* [11].

Fault tolerance.

One of the main purposes of distributed systems, beside increased performance, is fault tolerance. Fault tolerance involves error detection, error signaling, and error recovery by means of controlled system degradation and redistribution. The error recovery may be transparent [4], [5] or non-transparent [14], [6]. In transparent recovery, the run-time system handles the recovery and reconfiguration. The application is not aware of an error state in the system. In non-transparent recovery, the error is signaled to the application and it may take any appropriate steps to degrade the service. A full description of the actions required after a processor failure is given by Knight *et al* [14].

Ada does not fully support error detection and error signaling [6], neither does the LRM [15] define the state of a distributed program after the loss of a portion of the computing environment. The exception mechanism does not allow error states to be transferred between parallel activities asynchronously since exceptions may only be transferred during a rendezvous.

The loss of hardware will result in the loss or malfunction of software components, such as variables, subprograms or tasks. However, the only predefined exception for the detection of lost software resources is TASKING_ERROR, which is raised by the run time system in the caller of an abnormal task. Exceptions for the detection of other lost software resources can be added to the package SYSTEM. These exceptions should be raised by the run time system at the access of the lost resource, analogously with TASKING_ERROR.

Kamrad *et al* [5] state that too many software designs include unnecessary details of the hardware configuration in its reconfiguration strategy. From the software point of view it is of no interest that a processor is lost, but rather the loss of the software operations which that processor supported. Furthermore, introducing hardware details prevents the software from being reusable.

Various authors illustrate the statement above. Kamrad *et al* [5] specifies a mechanism, in a separate partitioning language, that makes it possible to raise a user defined asynchronous exception into a list of named task when a defined state is set. Arévalo *et al* [2] defines a death-notice mechanism; if a task wants to be informed of the death of another, it gives directions to the run-time system to get calls through an entry point at these events. These two examples show error signaling that does not involve hardware information. Another example is given by Knight *et al* [6] where two exceptions, NODE_FAIL and COMM_FAIL, are raised in every process that survives a processor failure or network failure respectively.

THE DISTRIBUTED ADA RUN-TIME SYSTEM, DARTS.

The Distributed Ada Run-Time System, DARTS, is developed by the Measurement and Data Acquisition group at the Department of Physics, Uppsala University, in collaboration with the Swedish Defence Research Establishment. The system is one example of solving some of the problems described in the introduction above. The intention is to implement the entire run-time system software in Ada, to test the behavior of Ada in real time applications, and to examine portability and reuse of software components.

Partitioning and distribution in DARTS.

DARTS can be used for pre-partitioning as well as post-partitioning. Using the pre-partitioning approach, the partitioning information is added to the application source code as pragmas. In case of a post-partitioning approach, the partitioning could be performed by some CASE tool generating the transformed Ada code.

DARTS aims to support the distribution of :

- tasks
- task objects
- packages
- variables
- subprograms

A software component that is selected for distribution is called a *distributed unit* (DU).

A program is partitioned into distributed units. A virtual node (VN) consists of a set of DUs that can execute on the node. The subset that actually executes is defined at run-time. Performance and fault tolerance implies that a DU may be a member of several virtual nodes, and subprograms may even execute on several VNs simultaneously. Virtual nodes are assigned to physical nodes (PN). Currently, DARTS can only map a single virtual node onto each physical node.

The state of a DU on a given VN depends on the preparations for execution. A *local* DU is currently executing on the node. The node must be prepared to receive calls to the DU from other nodes. A *remote and idle* DU is idle on the local node. All calls to the DU are forwarded to a remote node where the DU is currently executing. In case of a node failure, the state of the DU may be changed from *remote and idle* to *local*. The *remote* state implies that the DU is executing on a remote node. In case of a node failure the unit may be redistributed to another node, but not to this node. The states of the DUs on a node are set at startup using a configuration file.

The distribution is performed by source code transformation. This involves the insertion of additional code into the application source code. The inserted code interfaces the application to the distributed run-time system.

A number of global exceptions are declared to handle failure states. Recovery is transparent in the case of stateless units. Only when a lost DU is referenced, exceptions signal the permanent or temporary inaccessibility of the DU, as in the case of TASKING_ERROR in a non-distributed environment. This mechanism has been chosen since a process needs no information that a DU is lost if no communication or synchronization is required.

The DARTS is designed to put callers into a hibernating state during remote calls [19]. No busy-wait in communication routines, as described by Eisenhauer *et al* [20], is needed.

Syntax for partitioning.

Other work [17] in the distributed Ada field indicate that pragmas are a proper base for distribution and reconfiguration information. This is partly due to the fact that Ada does not prohibit the adding of pragmas. One drawback is that the partitioning information is spread over the source code.

Two pragmas are used for program partitioning; pragma Distribute and pragma Redistribute. Pragma Distribute is used to identify a distributed unit and associate the unit to its executing virtual node(s). Pragma Redistribute is used to enumerate the possible target nodes for redistribution. In some cases, such as calls to a subprogram DU executing on several nodes, the allocation of a task object DU, or during a redistribution, a choice of node has to be made. The choice is ruled by a distribution criterion for the DU. The distribution criterion is specified as one of the following:

- CURRENT_LOAD
- AVERAGE_LOAD
- MAILING_LOAD
- SPACE.

System overview.

The DARTS consists mainly of three parts; the communication layer, the distribution layer, and the application layer.

The communication layer is the lowest level of the distributed Ada run-time system. The layer handles the abstraction of the network and supports the upper layers with functionality for node event handling and low level byte transfers.

The distribution layer contains logic for handling the current configuration of the distribution, high level internode handshaking, node load monitoring, and redistribution. It supplies the application layer with primitives for message transfers to distributed units, and provides a means to determine if a given DU is executing locally or remotely.

The application layer consists of adapted application code, generated by a source code transformer. The transformation is made by adding passive server units (SU), i e alternate bodies, to the distributed units. The server unit forwards any call to the actual DU, which may execute on the same node or on a remote node. Server units are sometimes named 'local agents' [17], or 'client stubs' [21].

One SU exists on a node for each DU that executes on the node, may execute on the node, or is used on the node. Some modifications may be necessary in the application code to adapt calls to the SU. All such modifications are performed by the source code transformer.

The communication between SU and DU is made by passing command messages, constituting remote calls and rendezvous, as well as unit creation, abortion, and elaboration.

SOURCE CODE TRANSFORMATION.

Identification of Distributed Units.

A unique unit number is generated for each distributed unit. The number is given at compile time by the source code transformer for static units. For dynamically created units the number is generated at run-time. Information about each unit is held in a unit identification record (UI). Figure 1 shows the UI declaration.

It is not allowed to separate a DU from the scope of the used non-local entities, unless these are made distributed. To obtain the necessary visibility from the distribution layer, all nested SUs identifiers are given a unique extension and put in the scope of the distribution layer.

Transformation of Procedures and Functions.

The distribution of a subprogram is simply performed by replacing the body of the subprogram with a server body forwarding the calls to the DU on the executing node. The subprogram UI is included in the server unit body as a constant. The actual subprogram code is included in the SU, if the DU is selected for local execution. The implementation of the SU is described in Figure 2. It is not allowed to separate a subprogram DU from the scope of the used global variables, unless these variables are DUs.

Transformation of Distributed Variables.

The DARTS concept comprises two different paradigms for distributed variables. The first is based on a totally distributed ownership of the variable, the second defines a single owner with all others using that one instance. Both paradigms use the same support from the distribution layer.

In the first case, the pragma DISTRIBUTE is used to identify all owners of the variable. A local copy is maintained in the distribution layer on each node and each variable update will be transformed to an update of the local copy in conjunction with a broadcast to update all other instances in the network. A variable reference is transformed to a reference to the local copy of the variable.

In the second case, the pragma DISTRIBUTE is used to identify the owner of the variable. Any update or reference to the variable is transformed to a call to update or obtain the value held by the owner. A pragma REDISTRIBUTE indicates an alternate owner of the variable.

Transformation of Packages.

The transformation of a package involves the creation of a procedure to contain the package executable part. This initiation procedure is called when DARTS elaborates the package during system startup or reconfiguration. Also, all entities declared in the package specification is automatically regarded as distributed units. Figure 3 shows the transformation of a sample package.

Transformation of Tasks and Task types.

Tasks and task type objects are replaced by server units implemented as packages, with all entries declared as procedures using the entry identifiers as procedure names. Two additional parameters are added to each entry procedure. The first parameter is used to identify the called distributed task and the second parameter is used for sending the time value in timed entry calls. Additional subprograms are used for initiation and abortion, and for obtaining task attributes. All task objects derived from a task type are handled by the same SU package. Figure 4 shows a task declaration and the corresponding server unit package.

All tasks that are selected for distribution are transformed to task types, as suggested by Bishop *et al* [22]. The transformed code handles the creation of the task object. A *new* statement is transformed to a call to the NEW_UNIT function. Figure 5 and 6 show the transformation of a declaration of a static task object and a declaration of a dynamic task object with a *new* statement. Note that the activation of the task, in Figure 5, is delayed until the beginning of the parent block, while the activation of the task object, in Figure 6, is performed in the *new* statement. An *abort* statement is transformed to a call to the ABORT_UNIT procedure in the SU package. The task attributes T'CALLABLE and T'TERMINATED are obtained by calls to corresponding functions.

The parent-child synchronization is made possible by adding an additional entry, AWAIT_-TERMINATION, to the task type declaration. This entry is accepted at the completion of the task executable part. Figure 7 shows the transformation of a task type into a DU.

The entry calls in the application source code are transformed to fit the SU package. Figure 8 shows the transformation of a basic entry call. Note that the task object is provided as a parameter, and that the package name is used for clarity only, since the use clause makes the SU directly visible.

A timed entry call is rewritten into a block containing a call to the server package and an exception handler containing the time-out executable code. Figure 9 shows the transformed call. The conditional entry call is transformed into a timed entry call with a delay time of 0.0 seconds, in accordance to the functionality specified in the LRM [15]. The basic entry call is implemented as a timed entry call with infinite time.

When the call is made, the time-out parameter is used in a timed entry call on the remote node. If a time-out occurs, a time-out error message is returned to the calling node, resulting in the raising of a TIME_OUT_ERROR exception. This exception will be caught in the exception handler shown in Figure 9 and 10.

The termination mechanism in Ada is based on the block structure [15]. If a block, task, or subprogram has dependent tasks, it terminates when it has completed and all dependent tasks have terminated or are ready to terminate. An algorithm for termination in a multiprocessor environment is described by Flynn *et al* [23].

However, any efficient implementation of task termination requires access to the run-time system. The aim of the DARTS project was to keep the system portable and, hence, DARTS only supports immediate termination after completion. Synchronization is performed as remote or local rendezvous, between the parent and, in sequence, each child. The transformation of the terminate alternative is not addressed.

KEY MECHANISMS.

Remote calls.

The DARTS implements a remote call mechanism that avoids busy waits [19]. Unlike Volz *et al* [17], who uses a distribution package with a pool of call agents tasks for each distributed unit, DARTS uses only one pool of general and reusable call agent tasks and a single distribution package. This minimizes the storage needed for task agents in the distributed run-time system. To decouple the application from the lower layers, a pool of agent tasks is used on the calling node. This facilitates an orderly recover of a communication failure.

The distribution package consists mainly of the FORWARD_TO procedure. This procedure, called by the call agents on an executing node, interprets the DU identification number, unpacks the parameters and performs a call to the identified DU. As the call is completed, the return parameters are packed and sent to the calling node.

Exception handling.

The exception handling in DARTS may be divided into two parts;

- System Exceptions,
- User Exceptions.

The system exceptions are TIME_OUT_ERROR, used for the distributed timed rendezvous, DU_LOST_ERROR, used for signaling the loss of a distributed unit, and DU_INACCESSIBLE_ERROR, used for signaling the temporary inaccessibility of a DU during redistribution. For user exceptions, a simplified exception handling is used. All user exceptions are mapped into a single USER_ERROR exception. The same method is used by Atkinson *et al* in the DIADEM project [1].

Initiation and redistribution.

Distributed units are initiated in a uniform manner at system startup and during reconfiguration. A unit is idle until initiated. The initiation is handled by the distribution layer.

The initiation is performed by sending a command message to the DU to be initiated. At the remote node a call agent calls the initiate-entry, or -procedure, of the DU.

The redistribution logic is contained in the distribution layer and is implemented as a task. The application continues its execution during redistribution, although all references to DUs currently redistributed are signaled by the predefined DU_INACCESSIBLE_ERROR exception. If a DU with several simultaneously executing copies is redistributed, calls are simply redirected to the remaining DU copies.

The redistribution task is activated by the detection of a node failure. At redistribution, one of the the remaining nodes is selected to be master of the redistribution. The master node accepts *redistribution requests* from the other nodes, and, using a *redistribution acknowledgement*, signals the acceptance of the redistribution mastership. The master then evaluates all DUs that were executing on the failed node. For each such DU, the master selects a target node and sends an initiation message to the DU on the selected node. When all DUs have been processed, the master sends a *redistribution completed* message to all nodes in the network. This message releases the redistribution state in the system and puts all redistribution handlers to sleep.

Project status and performance tests.

DARTS was originally implemented on a VAX-cluster using the DEC Ada compiler. The communication layer was first based on mailboxes where the nodes were simulated as processes on a single machine. In a second version of the communication layer, Ethernet communication was used between workstations.

Currently DARTS is revised for increased performance and adapted to bare MC68030 boards (Force CPU37ZBE) with Ethernet communication. The compiler used is the TeleGen2 cross compiler, version 3.23 [24].

The performance data, on the MC68030 boards, available at this stage are not complete, nor fully analyzed. However, it has been found that a complete remote procedure call takes 12.2 ms, using an unoptimized version of DARTS. One observation is that rendezvous times are not very expensive using modern Ada compilers. A 'seize' operation on a semaphore implemented as a task takes less than 80 μs, and the Ethernet interrupt task handles an interrupt and buffers the incoming packet in another rendezvous in approximately 400 μs. However, packing and unpacking parameters, and transferring parameter blocks to and from Ethernet buffers, is time consuming.

CONCLUSIONS.

The Distributed Ada Run-Time System represent one approach to distribute Ada programs. We have found that some restrictions must be put on the language for use in distributed applications. High portability requirements on DARTS impose restrictions on the use of the underlying run-time system, preventing an efficient solution to the exception transferring problem and the distributed task termination problem.

We have also found that error recovery may, in the case of stateless DUs, be invisible to the application. The application can be informed of a failure at a reference to a lost program part. Hence, there is no need for an asynchronous exception mechanism to transfer failure states to the application.

Most Ada mechanisms are maintained in DARTS. This includes:

- remote and local, timed entry calls,
- remote and local subprogram calls,
- shared variables,
- exceptions at remote and local calls,
- dynamic creation and abortion of distributed tasks,
- limited task termination.

The ability to execute several instances of procedures, and the possibility to allocate tasks of the same task type on any number of nodes, make DARTS a vehicle to achieve high performance.

REFERENCES.

[1] C. Atkinson, T. Moreton and A. Natali,
Ada for Distributed Systems, Cambridge University Press, 1988.

[2] S. Arévalo and A. Alvarez,
Fault tolerant distributed Ada,
2nd International Workshop on Real Time Ada Issues, 1988.

[3] A.D. Hutcheon and A.J. Wellings,
Supporting Ada in a Distributed Environment,
2nd International Workshop on Real Time Ada Issues, 1988.

[4] D. Cornhill,
A Survivable Distributed Computer System For Embedded Applications Written In Ada, Ada Letters, Vol 3, Number 3, 1983.

[5] M. Kamrad, R. Jha and G. Eisenhauer,
Reducing the Complexity of Reconfigurable Systems in Ada,
2nd International Workshop on Real Time Ada Issues, 1988.

[6] J. Knight and M. Rouleau,
A New Approach to Fault Tolerance in Distributed Ada Programs,
2nd International Workshop on Real Time Ada Issues, 1988.

[7] M. Kamrad, R. Jha and D. Cornhill, *Distributed Ada.* ACM, 1987.

[8] D. Cornhill,
Four approaches to partitioning of Ada programs for execution of distributed targets, IEEE Computer Society Conference on Ada Applications and Environments, 1984.

[9] R. Fors, U. Olsson and G. Larsson,
The use of Ada in large shipborne weapon control system,
Ada in Industry, Cambridge University Press, 1988.

[10] R.M. Clapp and T. Mudge,
Ada on a Hypercube, Ada Letters, Vol 9, Number 2, 1989.

[11] A.B. Gargaro, S.J. Goldsack, R.A. Volz and A.J. Wellings,
Supporting Reliable Distributed Systems in Ada9X,
Proc. of the symposium Distributed Ada 1989.

[12] R. V. Scoy, J. Bamberger, and R. Firth,
An overview of DARK, Ada Letters, Vol 9, Number 7, 1989.

[13] J. Armitage and J. Chelini,
Ada Software on Distributed Targets: A Survey of Approaches,
Ada Letters, vol 4, Number 4.

[14] J. Knight and I. A. Urquhart,
On the implementation and use of Ada on fault-tolerant distributed systems, IEEE
Transactions on Software Engineering, Vol SE-13, No 5, May 1987.

[15] *Reference Manual for the Ada Programming Language,*
(ANSI/MIL-STD-1815A),Ada Joint Program Office,
Department of Defence, Washington, D.C. 20301, 1983.

[16] T. Mudge, *Units of Distribution for Distributed Ada.*ACM, 1987.

[17] R.A. Volz, P. Krishnan and R.J. Theriault,
Distributed Ada- A Case Study.
Proc. of the symposium Distributed Ada 1989.

[18] D. Cornhill,
Distributed Ada Project. Honeywell,1982.

[19] D. Wengelin, M. Carlsson-Göthe and L. Asplund,
A System Structure to Avoid Busy Wait, Ada Letters, Vol 10, Number 1, 1990.

[20] G. Eisenhauer, R. Jha and J.M. Kamrad II,
Targeting a Traditional Compiler to a Distributed Environment,
Ada Letters, Vol 9, Number 2, 1989.

[21] A.D. Hutcheon and A.J. Wellings,
The York Distributed Ada Project,
Proc. of the symposium Distributed Ada 1989.

[22] J.M. Bishop, S.R. Adams and D.J. Pritchard,
Distributing Concurrent Ada Programs by Source Translation,
Software-Practice and Experience, Vol 17, December 1987.

[23] S. Flynn, E. Schonberg and E. Schonberg,
The Efficient Termination of Ada Tasks in a Multiprocessor Environment,
Ada Letters, Vol 7, Number 7,1987.

[24] L. Asplund, M. Carlsson Göthe, D. Wengelin and G. Bray.
Real time compilers for the 68020.
Ada Letters, Vol 9, Number 7,1989.

```
-- Declare the distributed unit kinds.
type UNIT_KIND is (     TASK_TYPE_KIND,    DERIVED_TASK_OBJECT_KIND,
                        VARIABLE_KIND,     SUBPROGRAM_KIND,
                        PACKAGE_KIND);

-- The id of a distributed unit.
type UNIT ( KIND : UNIT_KIND := SUBPROGRAM_KIND) is
  record
    UNIT_NO : NATURAL;
    case KIND is
      when DERIVED_TASK_OBJECT_KIND =>
        DERIVED_FROM : NATURAL := 0;
        DERIVED_TASK_OBJECT_SUBUNIT : NATURAL := 0;
        THE_TASK_OBJECT : ACCESS_TYPE := ( others => 0 );
      when TASK_TYPE_KIND => TASK_SUBUNIT : NATURAL := 0;
      when  PACKAGE_KIND =>  PACKAGE_SUBUNIT : NATURAL := 0;
      when others =>         null;
    end case;
  end record;
```

Figure 1. The Unit Identification record, UI.

```
-- The original source code.
procedure INC(X: in out INTEGER) is
begin
  X:= X + 1;
end INC;
pragma DISTRIBUTE( INC, TO => MACHINE_2);
pragma REDISTRIBUTE( INC, TO => MACHINE_1);

-- The transformed SU on machine 1.
procedure INC(X: in out INTEGER) is
  use DISTRIBUTED_MAIL;
  MY_D_U: constant DISTRIBUTED_UNIT(SUBPROGRAM_KIND) :=
    (KIND => SUBPROGRAM_KIND, UNIT_NO => 1);

  -- Local DU.
  procedure INC_LOCAL(X: in out INTEGER) is
  begin
    X:= X + 1;
  end INC_LOCAL;

begin
  if UNIT_MODE(MY_D_U) = LOCAL then INC_LOCAL(X);
  else -- REMOTE or REMOTE_AND_IDLE
     declare
      OUT_C, IN_C: COMMAND_TYPE;
    begin
      OUT_C.KIND:= MESSAGE_REQUEST;
      INTEGER_HANDLER.PACK(X, OUT_C.PARAMETERS); -- Pack parameters.
      SEND(MY_D_U, OUT_C, IN_C); -- Send and block caller.
      INTEGER_HANDLER.UNPACK(X, IN_C.PARAMETERS); -- Unpack parameters.
    end;
  end if;
end INC;
```

Figure 2. An implementation of a procedure server unit. The generated code allows both local and remote operation depending on the unit state. Note the unit identification declared as a constant and used when performing a remote call.

```
-- The original source code.
package MY_PACKAGE is
 procedure SOME_PROCEDURE;
 procedure SOME_OTHER_PROCEDURE;
end MY_PACKAGE;

package body MY_PACKAGE is
 -- Package declarative and
 -- implementation part.
begin
 -- Package executable part.
end MY_PACKAGE;


-- The transformed source code.
package MY_PACKAGE is
 procedure SOME_PROCEDURE;
 procedure SOME_OTHER_PROCEDURE;
 -- Added subprograms.
 procedure INITIATE_UNIT;
end MY_PACKAGE;

package body MY_PACKAGE is
 -- Package declarative and implementation part.
 procedure INITIATE is
 begin
    -- Package original executable part.
 end INITIATE;
-- Empty executable part.
begin
 null;
end MY_PACKAGE;
```

Figure 3. *The transformation of a package into a DU.*

```
-- The original source code.
declare
 task THE_SERVER is
    pragma DISTRIBUTE( TO => MACHINE_1);
    entry FIRST_ENTRY;
    entry SECOND_ENTRY;
 end THE_SERVER;
 task body THE_SERVER is separate;
begin
 -- Executable code.
end;


-- The transformed source code.
-- Extracted to the uttermost application
-- level due to visibility reasons.
package THE_SERVER_001 is
 subtype THE_SERVER_T001 is UNIT;
 procedure FIRST_ENTRY(
    TASK_OBJECT : THE_SERVER_T001;
    TIMEOUT : DURATION := DURATION'LAST );
 procedure SECOND_ENTRY(
    TASK_OBJECT : THE_SERVER_T001;
    TIMEOUT : DURATION := DURATION'LAST );
 -- New unit.
 function NEW_UNIT(
    THE_BLOCK_ID : BLOCK_ID )
    return THE_SERVER_T001;
 -- Abort procedure.
 procedure ABORT_UNIT(
    TASK_OBJECT : in out THE_SERVER_T001);
 -- Task attributes.
 function UNIT_CALLABLE(
    TASK_OBJECT : THE_SERVER_T001)
    return BOOLEAN;
 function UNIT_TERMINATED(
    TASK_OBJECT : THE_SERVER_T001)
    return BOOLEAN;
 procedure AWAIT_TERMINATION(
    TASK_OBJECT : THE_SERVER_T001);
end THE_SERVER_001;

-- The transformed code.
declare
 THE_SERVER :THE_SERVER_001.THE_SERVER_T001;
begin
 THE_SERVER := THE_SERVER_001.NEW_UNIT;
 begin
    -- Executable code.
 end;
 -- Added to the end of the block declaring
 -- the task or the body of the package to
 -- synchronize task termination.
 THE_SERVER_001.AWAIT_TERMINATION( THE_SERVER);
end;
```

Figure 4. *The transformation of a task declaration into a SU declaration and the transformation of the declaring block.*

```
-- The original source code.
declare
 A_SERVER : THE_SERVER;
begin
 A_SERVER.FIRST_ENTRY;
end;

-- The transformed source code.
declare
 use THE_SERVER_002;
 A_SERVER : THE_SERVER;
begin
 A_SERVER := THE_SERVER_002.NEW_UNIT;
 FIRST_ENTRY( A_SERVER);
 THE_SERVER_002.AWAIT_TERMINATION( A_SERVER);
end;
```

Figure 5. *A declaration of a static task object.*

```
-- The original source code.
declare
 A_SERVER : THE_SERVER_POINTER
    := new THE_SERVER;
begin
 A_SERVER.FIRST_ENTRY;
end;

-- The transformed source code.
declare
 use THE_SERVER_002;
 A_SERVER : THE_SERVER_POINTER
    := new THE_SERVER'(THE_SERVER_002.NEW_UNIT);
begin
 FIRST_ENTRY( THE_SERVER.all);
end;
```

Figure 6. *A declaration of a dynamic task object.*

```
task type THE_SERVER is
 entry FIRST_ENTRY;
 entry SECOND_ENTRY;
end THE_SERVER;

task body THE_SERVER is
 -- Task body declarative part.
begin
 -- Task body executable part.
end THE_SERVER;


-- The transformed source code.
task type THE_SERVER is
 entry FIRST_ENTRY;
 entry SECOND_ENTRY;
 -- Entry added by the transformer.
 entry AWAIT_TERMINATION;
end THE_SERVER;

task body THE_SERVER is
begin
 declare
    -- Task body declarative part.
 begin
    -- Task body executable part.
 end;
 accept AWAIT_TERMINATION;
end THE_SERVER;
```

Figure 7. *The transformation of a task DU.*

```
-- The original source code.
THE_SERVER.FIRST_ENTRY;

-- The transformed source code.
THE_SERVER_002.FIRST_ENTRY( THE_SERVER);
```

Figure 8. *The transformation of a basic entry call.*

```
-- The original source code.
select
 THE_SERVER.FIRST_ENTRY;
or
 delay 10.0;
 -- Time-out executable part;
end  select;

-- The transformed source code.
begin
 THE_SERVER_002.FIRST_ENTRY( THE_SERVER, TIMEOUT => 10.0);
exception
 when DISTRIBUTED_EXCEPTIONS.TIME_OUT_ERROR =>
    -- Time-out executable part;
end;
```

Figure 9. *The transformation of a timed entry call.*

```
-- The original source code.
select
 THE_SERVER.FIRST_ENTRY;
else
 -- Time-out executable part;
end  select;

-- The transformed source code.
begin
 THE_SERVER_001.FIRST_ENTRY( THE_SERVER, TIMEOUT => 0.0);
exception
 when DISTRIBUTED_EXCEPTIONS.TIME_OUT_ERROR =>
    -- Time-out executable part;
end;
```

Figure 10. *The transformation of a conditional entry call.*