On-Line Upgrade of Program Modules Using AdaPT

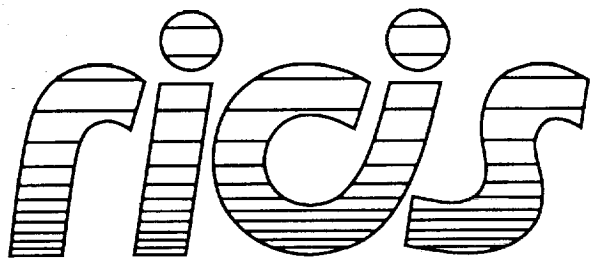
Raymond S. Waldrop
Richard A. Volz
Gary W. Smith
Texas A&M University

Stephen J. Goldsack
A. A. Holzbach-Valero
Imperial College, London, England

June 17, 1993

Cooperative Agreement NCC 9-16
Research Activity No. SE.35

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

On-Line Upgrade of Program Modules Using AdaPT

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Raymond S. Waldrop, Richard A. Volz and Gary W. Smith of Texas A&M University and A. A. Holzbacher-Valero and Stephen J. Goldsack of Imperial College, London, England. Dr. E.T. Dickerson served as RICIS research coordinator.

Funding was provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA research coordinator for this activity was Terry D. Humphrey of the Systems Software Section, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

On-Line Upgrade of Program Modules Using AdaPT

Task T7 Report

NASA Subcontract #074
Cooperative Agreement NCC-9-16
Research Activity # SE.35

Period of Performance: May 1, 1990 - March 31, 1993

Submitted to
RICIS

Submitted by
Ray Waldrop, Texas A&M University
Richard Volz, Texas A&M University
Gary W. Smith, Texas A&M University
A. A. Holzbacher-Valero, Imperial College, London, England
S. J. Goldsack, Imperial College, London, England

On-Line Upgrade of Program Modules Using AdaPT

Raymond S. Waldrop, Richard A. Volz,
Gary W. Smith
Texas A&M University
A. A. Holzbacher-Valero and S. J. Goldsack,
Imperial College, London

June 17, 1993

Contents

1 Introduction

One purpose of our research is the investigation of the effectiveness and expressiveness of AdaPT[1], a set of language extensions to Ada 83, for distributed systems. As a part of that effort, we are now investigating the subject of replacing, e.g. upgrading, software modules while the software system remains in operation. The AdaPT language extensions provide a good basis for this investigation for several reasons:

- they include the concept of specific, self-contained program modules which can be manipulated,
- support for program configuration is included in the language, and
- although the discussion will be in terms of the AdaPT language, the AdaPT to Ada 83 conversion methodology being developed as another part of this project will provide a basis for the application of our findings to Ada 83 and Ada 9X systems.

The purpose of this investigation is to explore the basic mechanisms of the replacement process. With this purpose in mind, we will avoid including issues whose presence would obscure these basic mechanisms by introducing additional, unrelated concerns. Thus, while replacement in the presence of real-time deadlines, heterogeneous systems, and unreliable networks is certainly a topic of interest, we will first gain an understanding of the basic processes in the absence of such concerns. The extension of the replacement process to more complex situations can be made later.

A previous report[3] established an overview of the module replacement problem, a taxonomy of the various aspects of the replacement process, and a solution to one case in the replacement taxonomy. This report provides solutions to additional cases in the replacement process taxonomy: replacement of partitions with state and replacement of nodes. The solutions presented here establish the basic principles for module replacement. Extension of these solutions to other more complicated cases in the replacement taxonomy is direct, though requiring substantial work beyond the available funding.

2 Overview of AdaPT

AdaPT has been described in detail in[1]. This section will provide a brief introduction to the major features of AdaPT. New features introduced in AdaPT are the **partition**, the **node**, and the **public**.

- **Partitions.** A partition may be considered to constitute a "class" in the sense used in object oriented systems and languages. However, it is closely modeled on the Ada package, presenting, in an interface specification, the items which are made available for

```
partition P is
   :
end P;
partition body P is
   :
begin
   :
end P;
```

Figure 1: Sample Partition Declaration

its interaction with other system components. Thus its interface may contain procedures and functions, task declarations, and constants and exception declarations. It may not contain any object or type declarations. An outline of a partition declaration is shown in Figure 1. To help in defining the initial configuration of a partition instance, a partition may have parameters (**in** parameters only), which are supplied by the program invoking the allocator when a new instance of the partition is created. The partition is the *unit of distribution* in AdaPT.

A partition is a library unit, and constitutes a type declaration. Other units may have **with** clauses to give them access to the definition in the library, and within the scope of the with clause they may declare variables of the type. However, the type is an implicit access type, and no instance of the partition is created by such a declaration. Creation of new instances of a partition are obtained by the use of **new** allocator statements, but these are permitted only in the definition of *nodes* which are described in the following paragraphs. Once a partition instance has been created, references to that instance may be circulated by using an assignment statement to copy the value of one (access) variable to another.

The use of library units "withed" by a partition leads to a special problem. Such packages may have "state", and consequently cannot be shared safely between different instances of a partition and between different partitions which may "with" the same unit. Thus, the semantics of **with** clauses for partitions are different from those for packages in a normal Ada program. All units in the transitive closure of the directed graph formed by the *with* clauses of a partition, up to but not including any public unit or any other partition, form part of the partition. These units are replicated as a whole with each replication of the partition.[1] Each instance of a package or other object included in such a dependency graph, belongs therefore to one and only one partition instance. In contrast therefore to the public units described below, we sometimes refer to such packages as *non-public*

[1] We note that Ada 9x has similar replication rules.

```
node N is
   :
end N;

with P;
node body N is
   MY_P : P := new P'PARTITION;
   :
begin
   :
end N;
```

Figure 2: Sample Node Declaration

units.[2]

- **Nodes.** Nodes differ very little from partitions. They too have features corresponding to those of packages; like partitions they have separate interfaces and bodies, and instance variables to reference them. However, nodes *can* create new instances of partitions and other nodes. Their role is to serve as units which will eventually be compiled and linked to form executable binary objects. They are thus the *units of configuration* in AdaPT. Figure 2 shows a simple node definition which includes the creation of a partition instance. Note again that MY_P is an access type which points to an instance of **partition** P. Thus, to create the object to which MY_P points, we must create an object of the anonymous type for **partition** P. This is accomplished using the attribute 'PARTITION.

  The issue of system construction, start up and elaboration is described in AdaPT as a normal Ada main program call for a first selected node, called the *distinguished* node; this then "creates" others and so recursively until the whole system is elaborated.

- **Conformant Partitions.** To support the provision of changed modes in a program, particularly as a technique for recovery following failure of part of the system, partitions can have "peers" which have *identical* interfaces but different bodies. In object oriented terminology they would be of the same "type", possibly one a subtype of the other, capable of providing the same set of actions for a client, albeit with different effect. In AdaPT, a conformant partition has the same interface as the partition to which it conforms, and access variables pointing to instances of conformant partitions may be used interchangeably with access variables pointing to instances of the original partition. An example of the creation of a conformant partition is shown in Figure 3.

  It is likely that conformant partitions may give rise to extra overhead. Some provisions may need to be made to explicitly note partitions which can not be determined to definitely

[2]The word *private* has, of course, other connotations in Ada, including AdaPT.

```
partition P is
   :
end P;
partition body P is
   :
end P;
partition Q is P; -- Q has the same interface as P
partition body Q is -- Q may have a different body as well as different context clauses
   :
begin
   :
end Q;
```

Figure 3: Conformant Partition Declaration

not have conformant peers.

- **Conformant Nodes.** In exact analogy with the idea of conformant partitions, it is proposed to support conformant nodes.
- **Public Units.** Partitions and nodes need to share certain information, including type information needed for the parameters. Neither unit is allowed, however, to declare types in their specifications. In order to share information, a new unit entitled a *public* is provided. To facilitate replication of a public across partitions and nodes, publics may not contain any variable state. They may include types (except for access types), task types, task access types, static constants, subprograms (including generic subprograms), packages (which inherit the restrictions placed on public units), privates, exceptions, renames, and pragmas. The context clause of a public unit may include only other public units. Types in public units may be private, and may be defined along with operations on them so that they are "abstract data types". An example of a public declaration is shown in Figure 4.

3 AdaPT and the Replacement Process

AdaPT was designed to provide language support in the areas of program distribution and configuration. The features of AdaPT which provide this support are the new program unit constructs (**publics**, **partitions**, and **nodes**) and the use of the access variable paradigm as a means of referring to specific instances of partitions and nodes. The first subsection below

```
public P is
   :
end P;
public body P is
   :
end P;
```

Figure 4: Public Declaration

will discuss the usefulness of these constructs in providing for replacement of program modules while the program remains on-line.

In order to achieve on-line upgrades, support for dynamic allocation and deallocation of program modules is necessary and will be discussed in separate subsections below.

3.1 AdaPT's Support For Program Configuration

The strength of AdaPT's support for program configuration lies in its explicit definition of typed modules for program distribution (partitions) and configuration (nodes), and in its use of access variables to refer to instances of those typed program modules. Because partition and node instances are instances of a type, they may be manipulated by the program itself at runtime. Instances of these types may be created in an orderly manner using the allocator, and a single access variable may be made to refer to different instances of a partition by changing the value of that access variable[3]. Program reconfiguration can thus be accomplished by changing the values of a set of access variables in an orderly manner.

3.2 Operation of the Allocator

The original definition of AdaPT merely stated that instances of partitions and nodes were created by the use of an allocator. This allocator was responsible for performing all the necessary steps for creating and initializing the unit being created. The allocator then returned a pointer to the unit thus created. No more detailed mention was made as to the means by which unit instantiation was accomplished.

To provide the capability of an on-line upgrade of a program module, it is necessary for an executing AdaPT program to be able to dynamically link with and load object code which

[3]It should be remembered that although an access variable may refer to an instance of a partition or node, that instance's existence is not dependent on that access variable. Thus, multiple access variables may refer to a single instance. However, if a situation occurs in which no access variable refers to an instance of a partition or node, there is no mechanism for rediscovering that instance, and that instance is lost to the program.

was not in existence when the program's execution was first initiated. To provide executing AdaPT programs with this ability, we interpret the definition of the allocator to be such that it causes the underlying AdaPT run-time system to search the program library for the most recent version of the object code corresponding to the program unit of which the allocator is creating an instance. This object code will then be loaded onto the physical processor, and elaboration of the program unit instance will proceed according to the rules set forth in [1].

3.3 Deallocation of Partition Storage

The original definition of AdaPT in [1] made the implicit assumption that, once in use, partition instances are never discarded. Thus, no method for deallocating partition instances was discussed. There are several outstanding issues associated with such deallocation which merit further study. In this report, we will not address those issues. However, to provide a flavor of the possible use of such deallocation, we make use of a variant of Ada 83's UNCHECKED_DEALLOCATION procedure.[4] The procedure we will use has this form:

```
generic
   type NAME is partition;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);
```

An instantiation of this generic procedure is made using the name of the partition type that will be deallocated. (Recall that a partition declaration defines an access type to an anonymous type.)

4 The Replacement Process

Having presented an overview of AdaPT, we now begin a discussion of the replacement process itself. First, we discuss five parameters which may be used to characterize the replacement process. We then use these parameters to form a taxonomy of the replacement process.

4.1 Characterization of the Replacement Process

The complexity of the general problem of program module replacement is due to the wide variety of situations under which the replacement process must occur. The study of this problem can be simplified by breaking it down into a number of cases. To allow the problem space to be broken down, we have determined five parameters which can be used to classify instances of the problem. These parameters are:

- the type of replacement,

[4] See [2] for additional discussion of this issue.

- the type of module to be replaced,
- the location of the replacement module(s),
- the need for a replacement module's state to match that of the module it is replacing, and
- the degree of change involved between the specification of the original module and the specification of its replacement.

These five parameters will be explained below.

Types of Replacement

We divide replacement processes into two types: *planned* and *unplanned*. A planned replacement is one where the system knows about the upcoming replacement before the module to be replaced is deactivated. An unplanned replacement is one where the system does not know of the need for the replacement until the module in question is found to be no longer in service. The main difference between these two cases is that the system designer typically has more options open to him in the planned case due to the fact that the original module is still available for use. An example of a planned replacement is that of an operator instructing the system to replace a program module with a new version of that module. (This new version would presumably incorporate bug fixes, expanded capabilities, etc.) An example of an unplanned replacement is that of a loss of power to a physical machine. The latter would result in the unexpected loss of all system functions resident on that processor. Our present work is focused on planned replacements, i.e. upgrades.

Kinds of Replacement Modules

The design of AdaPT provides two syntactic units that can be replaced, nodes and partitions. These are the only module types whose replacement will be considered in this discussion. The replacement of a node will usually require the replacement of its partitions.

Possible Locations

There are three possible situations regarding the location of the replacement module(s):

- *local*, meaning the replacement module is to reside on the same node as the module being replaced,
- *remote*, meaning the replacement module is to reside on a different node from the module being replaced, and

- *multiple remote*, meaning that various portions of a node are replaced by modules on different nodes.

State

There are significant differences in the replacement process depending upon the role of state in the module being replaced. There are two different cases to be considered:

- The module to be replaced has no state and creates no state via the allocator.
- The module to be replaced contains state whose consistency must be maintained throughout the replacement process.

Module Specifications

As a program evolves over time, changes will be made to various modules of that program. These changes fall into three categories:

- changes in which the module's specification remains unchanged, as in conformant partitions in AdaPT,
- changes in which the module's specification is extended, i.e. items are added to the module's specification, as in inheritance in object oriented languages, and
- changes in which the module's specification is reduced, i.e. items are removed from the module's specification, as is permitted in some object oriented languages.

A fourth category, where some items are added to the module's specification while other items are removed, is merely a composition of the second and third categories listed above. We will therefore not address this fourth category separately.

4.2 Taxonomy of the Replacement Process

In the previous section, we presented five parameters of the replacement process. In our discussions of these parameters, we listed the possible values these parameters may take on. Any instance where a module is to be replaced may be classified by listing the values of these parameters. Because there are a finite number of parameters, and a finite number of values for those parameters, there is a finite number of combinations of those parameters values. Additionally, some combinations of parameter values will not occur.

To aid in understanding what cases are possible, we have created the acyclic directed graph shown in Figure 5. In this graph, the vertices other than "Enter" and "Exit" represent the

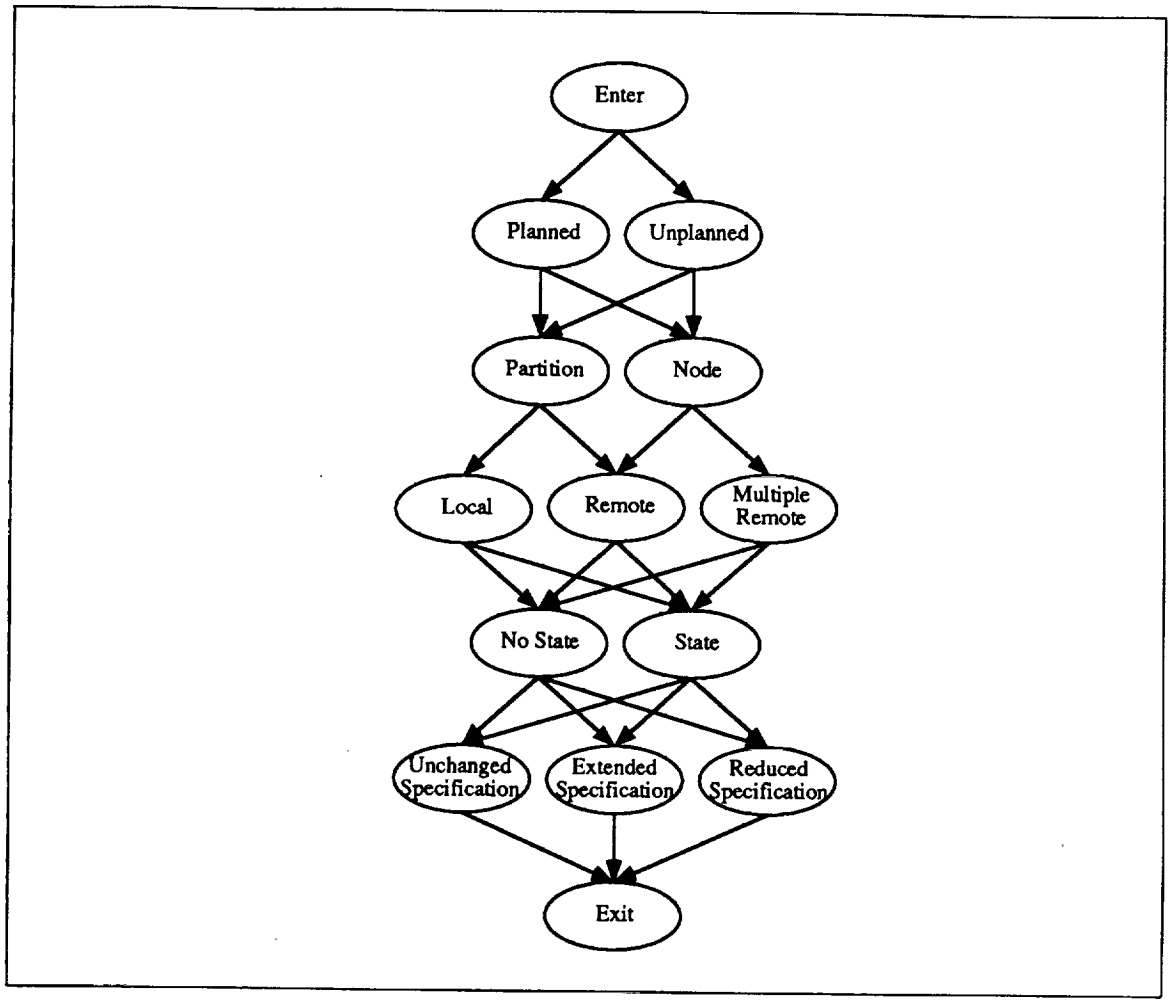


Figure 5: Reconfiguration Situation Classification Graph

possible values for the five parameters of the replacement process, with the vertices representing values corresponding to the same parameter being placed at the same level as measured from the "Enter" vertex. The arcs connecting the nodes represent possible combinations, i.e. the presence of an arc from "Partition" to "Remote" indicates that this combination of values is permissible, while the absence of an arc from "Partition" to "Multiple Remote" indicates that this combination is not permissible. A path describing a module replacement situation may be obtained by traversing the graph from vertex marked "Enter" to the vertex marked "Exit". At each vertex encountered during the traversal, the path should follow the arc to the vertex which represents the parameter value corresponding to the situation being classified, or the arc to "Exit" in the case of the last level of vertices.

Our approach to the replacement problem is to investigate the various possible cases to learn what techniques are needed to solve that particular case. These techniques can then be

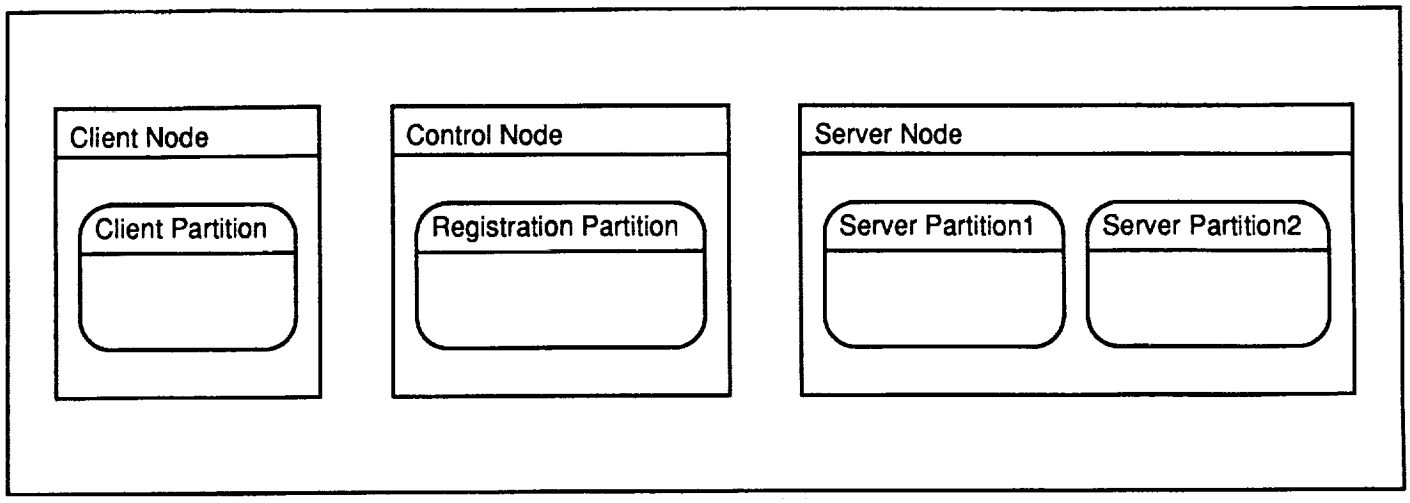


Figure 6: Physical placement of partitions and nodes1

applied to solve the general case. In the next section, we present a solution to one of the possible module replacement situations.

5 Partition Replacement

5.1 Description

In our previous work[3] we supplied a solution to stateless partition replacement. In this section we build upon that solution by introducing partition replacement with state. Specifically, we will show the replacement of local server and client partitions while maintaining state information across replacement instances.

Our system will consist of 3 Nodes and several partitions. The nodes include a client node, a server node, and a controller node, each with at least one partition as shown in Figure 6. The controller node is designated as the distinguished node. Upon allocation by the run-time system, the controller node creates the other two nodes. The control node also initiates the replacement process, presumably through interaction with the user.

In our solution, all client routines (users of the server routines) reside on partitions on the client node. Our protocol could have several client nodes, each with varying number of client partitions. In order to simplify our example, we include only one client node and one client partition. In the first example, the client partition is not replaceable. The second example provides for replaceable client partitions.

The server node contains two partitions and they are are independently replaceable. Replacement is complicated since clients must be notified of the change and given access to the

replacement. As there may be many client routines in the general case, explicit client notification of server changes was deemed too costly. Our solution implicitly notifies clients of a server change through an "out" parameter on normal service routine calls. This parameter consists of a reference to the latest server partition instance. If there is not a replacement in progress, the server partition passes back a reference to itself. If a replacement is in progress, the server partition redirects the service call to the replacement and returns a reference to the replacement.

During replacement, the replaced server partition instance can not be deallocated until all potential clients have been notified of the change. Our solution requires that clients "register" through a registration partition. The registration partition is created by the controller node and is not replaceable in our present solution. The server node informs the registration partition of the initial server partition instance and of any later server partition changes. Upon registration, the clients are given access to the latest server partition instance and the server partition is notified so that it can track the number of clients registered with it.

As clients are redirected during a replacement they are automatically registered with the replacement and deregistered from the replaced partition. The replaced partition can be deallocated as soon as all clients have been redirected (count of currently registered clients is zero). Since redirection occurs during service calls, the amount of time that both old and new partition instances are both active depends on the maximum amount of time between service calls for any registered clients. If this is an unacceptably long period of time or if a client no longer needs a service, a direct deregistration option is also provided to the client. If a client deregisters, it must register again before using the service routine.

This solution assumes that the clients will not send a second request until they have received a reply to their first request. If the clients did not wait for the reply to their request some other mechanism would have to be used to determine when a server partition could be safely deallocated. To avoid obscuring the objective of our example, i.e.. to study the underlying mechanisms, we chose not to include such considerations in this example.

5.2 Replaceable Server Partitions

In this section we give the solution for replaceable server partitions. The solution consists of the following AdaPT units:

- **package** LOCKER,
- **public** SERVER1_PUBLIC,
- **public** SERVER2_PUBLIC,
- **partition** SERVER1_PARTITION,
- **partition** SERVER2_PARTITION,

- **partition** CLIENT_PARTITION,
- **partition** REGISTRATION_PARTITION,
- **node** SERVER_NODE,
- **node** CLIENT_NODE,
- **node** CONTROL_NODE,

The package LOCKER is used to provide control over shared variables similar to the Readers-Writers problem. General solutions to the Readers-Writers problem can be found elsewhere. The implementation we are using is given below.

```
-- This package provides a simple solution to the readers-writers problem.
-- It is modelled after the solution in Barnes' "Programming in Ada", 3rd ed.
package LOCKER
   task type LOCK is
      entry READ;
      entry WRITE;
      entry DONE;
   end LOCK;
end LOCKER;

package body LOCKER is
   task body LOCK is
      NO_WRITE : BOOLEAN := FALSE;
      READERS : NATURAL := 0;
   begin
      accept WRITE;
      accept DONE;
      CONTROL:
      loop
         select
            accept READ;
            READERS := READERS + 1;
         or
            accept DONE;
            READERS := READERS - 1;
         or
            accept WRITE do
               CLEAR_READERS:
               while READERS > 0 loop
                  accept DONE;
                  READERS := READERS - 1;
               end loop CLEAR_READERS;
            end WRITE;
            accept DONE;
         end select;
      end loop CONTROL;
   end LOCK;
end LOCKER;
```

As stated previously, a registration partition is used to control access to the service routines. Clients are expected to register with a server through a call to the appropriate registration

routine. Registration returns access to the latest service partition instance and allows the server partition to track the number of clients.

The registration partition is given below. It is intended to be a partition within the control node, which is supplied later in this section. The SET_INSTANCE routine is called by the control or server node to inform the registration partition of the latest instance of a service partition. The REGISTER routine is called by clients to gain access to the latest instance.

In order to simplify our example, the registration partition is non-replaceable. It could be viewed as a service routine and be made replaceable using much the same techniques as we show for the replaceable server partitions.

Letting SERVER1_PARTITION and SERVER2_PARTITION be defined later, the registration partition has the following form.

```
with SERVER1_PARTITION;
with SERVER2_PARTITION;
partition REGISTRATION_PARTITION is

  package SERVER1 is
     function REGISTER return SERVER1_PARITION;
     procedure SET_INSTANCE (NEW_SERVER : in SERVER1_PARTITION);
  end SERVER1;

  package SERVER2 is
     function REGISTER return SERVER2_PARITION;
     procedure SET_INSTANCE (NEW_SERVER : in SERVER2_PARTITION);
  end SERVER2;

end REGISTRATION_PARTITION;


with SERVER1_PARTITION;
with SERVER2_PARTITION;
with LOCKER;
partition body REGISTRATION_PARTITION is

  package SERVER1 is
     CURRENT_PARTITION : SERVER1_PARTITION;     -- reference to current instance
     PARTITION_LOCK : LOCKER.LOCK;              -- r/w control task

     -- inform latest instance and return referenct to latest instance
     function REGISTER return SERVER1_PARITION
     begin
        PARTITION_LOCK.READ;
        CURRENT_PARTITION.REGISTER;
        PARTITION_LOCK.DONE;
        return CURRENT_PARTITION;
     end REGISTER;

     -- a new instance has been created.
     procedure SET_INSTANCE (NEW_SERVER : in SERVER1_PARTITION);
     begin
        PARTITION_LOCK.WRITE;
        CURRENT_PARTITION := NEW_SERVER;
        PARTITION_LOCK.DONE;
```

```
      end SET_INSTANCE;
  end SERVER1;

  package SERVER2 is
      -- same as the Server1 package except with Server2_Partition.
  end SERVER2;

end REGISTRATION_PARTITION;
```

Sample client partition and node routines are given below. At this point the only requirement for the client routines is that they register with the registration partition prior to using the server partition. In the next section, the client partitions are made replaceable.

```
with REGISTRATION_PARTITION;
partition CLIENT_PARTITION (REGISTRATION : in REGISTRATION_PARTITION);

with REGISTRATION_PARTITION;
with SERVER1_PARTITION;
with SERVER2_PARTITION;
partition body CLIENT_PARTITION (REGISTRATION : in REGISTRATION_PARTITION) is

    task COMPUTE;
    task body COMPUTE is
      DATA: ....
      -- Register and obtain latest instance of the server partition
      SERVER1 : SERVER1_PARTITION := REGISTRATION.SERVER1.REGISTER;
      SERVER2 : SERVER2_PARTITION := REGISTRATION.SERVER2.REGISTER;
    begin
       loop
          -- Perform normal execution, including service calls
          SERVER1.SERVICE_PROCEDURE(DATA,SERVER1);
          SERVER2.SERVICE_PROCEDURE(DATA,SERVER2);
       end loop;
    end COMPUTE;

end CLIENT_PARTITION;


with REGISTRATION_PARTITION;
node CLIENT_NODE (REGISTRATION : in REGISTRATION_PARTITION);

with CLIENT_PARTITION;
with REGISTRATION_PARTITION;
node body CLIENT_NODE (REGISTRATION : in REGISTRATION_PARITION) is

  CLIENT : CLIENT_PARTITION := new CLIENT_PARTITION'PARTITION(REGISTRATION);

end CLIENT_NODE;
```

As we noted earlier, there are two service partitions entitled SERVER1_PARTITION and SERVER2_PARTITION. We show only the code for the first. SERVER2_PARTITION would be very similar, differing only in implementation of actual service routines.

The service partitions may include state which needs to be maintained across replacements. Because partitions may not include types in their specifications, publics are used to declare the state data types. SERVER1_PUBLIC is given below.

```
public SERVER1_PUBLIC is
  type SERVER1_DATA is
     record
        -- state data for Server1
     end record;
end SERVER1_PUBLIC;
```

Below is the specification of SERVER1_PARTITION, followed by a brief description and its body.

```
with SERVER1_PUBLIC;
partition SERVER1_PARTITION is

   procedure SERVICE_PROCEDURE (PARAM_DATA : in out INTEGER;
                                CALL_NEXT : out SERVER1_PARTITION);

   procedure INITIALIZE(SELF_REFERENCE : in SERVER1_PARTITION;
                        INITIAL_DATA : in SERVER1_PUBLIC.SERVER1_DATA:=(...));

   procedure REPLACE(REPLACEMENT : in SERVER1_PARTITION);

   procedure REGISTER;
   procedure DEREGISTER;
   procedure BLOCK_FOR_ZERO_CLIENTS;

end SERVER1_PARTITION;
```

Note that the service routine contains an out parameter of SERVER1_PARTITION type. This is used to return to the client a reference to the latest service partition instance. If no replacement is in progress a reference to the same partition instance (self-reference) is returned. If a replacement is in progress a reference to the replacement instance is passed back to the client.

The INITIALIZE routine passes in a self-reference and state data to the partition instance. Initialization must be made before service routines are handled. If no second parameter is given, default data is used.

The REPLACE procedure is used to inform a partition instance that it is being replaced. A reference to the replacement instance is passed in. The replaced partition uses that reference to initialize the replacement with the current state data and redirects any further service calls to the replacement.

The REGISTER and DEREGISTER routines are used to internally track the number of clients currently registered with this particular instance of the server partition. The routine

BLOCK_FOR_ZERO_CLIENTS will not return until the number of clients currently registered is zero. At that point this partition instance can be safely deallocated.

Below is the body of SERVER1_PARTITION.

```
with LOCKER;
with SERVER1_PUBLIC;
partition body SERVER1_PARTITION is

   STATE_DATA : SERVER1_PUBLIC.SERVER1_DATA;

   -- reference to the latest active partition.  Self-reference until a
   -- replacement is in progress.  Redirect used to note that a replacement
   -- is in progress while ACTIVE_LOCK is a read/write control task.
   ACTIVE_PARTITION : SERVER1_PARTITION;
   REDIRECT : BOOLEAN := FALSE;
   ACTIVE_LOCK : LOCKER.LOCK;

   task COORDINATOR is
      entry REGISTER_ENTRY;
      entry DEREGISTER_ENTRY;
      entry ZERO_CLIENTS_ENTRY;
   end COORDINATOR_TYPE;
   task body COORDINATOR is
      NUM_CLIENTS : NATURAL := 0;
   begin
      loop
         select
            accept REGISTER_ENTRY;
            NUM_CLIENTS := NUM_CLIENTS + 1;
         or
            accept DEREGISTER_ENTRY;
            NUM_CLIENTS := NUM_CLIENTS - 1;
         or
            when NUM_CLIENTS := 0 =>
            accept ZERO_CLIENTS_ENTRY;
         end select
      end loop;
   end COORDINATOR;


   -- Normal service procedure.  Call_Next is used to note which
   -- partition instance the client should use on the next call.
   procedure SERVICE_PROCEDURE (PARAM_DATA : in out INTEGER;
                                CALL_NEXT : out SERVER1_PARTITION) is
   begin

      -- if replacement in progress, deregister client from this server,
      --      register with and redirect service call to replacement.
      ACTIVE_LOCK.READ;
      if REDIRECT then
         COORDINATOR.DEREGISTER_ENTRY;
         ACTIVE_PARTITION.REGISTER;
         ACTIVE_PARTITION.SERVICE_PROCEDURE(PARAM_DATA,ACTIVE_PARTITION);
      else
         --------  Perform Normal Service
      end if;
      CALL_NEXT := ACTIVE_PARTITION;
```

```
        ACTIVE_LOCK.DONE;
    end SERVICE_PROCEDURE;

    -- Send in a self partition reference and initialize state data.  Must be
    -- called before a normal service is performed.
    procedure INITIALIZE(SELF_REFERENCE : in SERVER1_PARTITION;
                         INITIAL_DATA : in SERVER1_PUBLIC.SERVER1_DATA:=(...)) is
    begin
        ACTIVE_LOCK.WRITE;
        ACTIVE_PARTITION := SELF_REFERENCE;
        STATE_DATA := INITIAL_DATA;
        ACTIVE_LOCK.DONE;
    end INITIALIZE;

    -- Start the shutdown process on this partition and initialize the
    -- replacement server with current state data.
    procedure REPLACE(REPLACEMENT : in SERVER1_PARTITION) is
    begin
        ACTIVE_LOCK.WRITE;
        REDIRECT := TRUE;
        ACTIVE_PARTITION := REPLACEMENT;
        ACTIVE_PARTITION.INITIALIZE_PARTITION(ACTIVE_PARTITION,STATE_DATA);
        ACTIVE_LOCK.DONE;
    end REPLACE;

    procedure REGISTER is
    begin
      COORDINATOR.REGISTER_ENTRY;
    end REGISTER;

    procedure DEREGISTER is
    begin
        COORDINATOR.DEREGISTER_ENTRY;
    end DEREGISTER;

    procedure BLOCK_FOR_ZERO_CLIENTS is
    begin
      COORDINATOR.ZERO_CLIENTS_ENTRY;
    end BLOCK_FOR_ZERO_CLIENTS;

end SERVER1_PARTITION;
```

In our solution, the server partitions are created by the server node. When a replacement is initiated, the server node creates the new partition and informs the registration partition of the change. It then directs the replaced partition instance to initialize its replacement with the current state data and to start redirecting service calls. When all clients have been directed to the replacement, the replaced partition can be deallocated.

The code for SERVER_NODE is given below. Note that it contains two replacement tasks, corresponding to the two server partitions. The only difference between the two tasks is in the types of partitions to be replaced.

```
with REGISTRATION_PARTITION;
```

```
node SERVER_NODE (REGISTRATION : in REGISTRATION_PARTITION) is
   procedure REPLACE_SERVER1;
   procedure REPLACE_SERVER2;
end SERVER_NODE;


with SERVER1_PARTITION;
with SERVER2_PARTITION;
with REGISTRATION_PARTITION;
node body SERVER_NODE (REGISTRATION : in REGISTRATION_PARTITION) is

   -- task to control the replacement of server1_partition
   task SERVER1_REPLACEMENT is
      entry FINISHED_SETUP;
      entry START_REPLACEMENT;
   end SERVER1_REPLACEMENT;
   task body SERVER1_REPLACEMENT is
      CURRENT_PARTITION : SERVER1_PARTITION;
      OLD_PARTITION : SERVER1_PARTITION;
   begin
      -- create initial instance, inform registration partition, and initialize
      CURRENT_PARTITION := new SERVER1_PARTITION'PARTITION;
      REGISTRATION.SERVER1.SET_INSTANCE(CURRENT_PARTITION);
      CURRENT_PARTITION.INITIALIZE(CURRENT_PARTITION);
      accept FINISHED_SETUP;
      loop
         accept START_REPLACEMENT;
         OLD_PARTITION := CURRENT_PARTITION;
         CURRENT_PARTITION := new SERVER1_PARTITION'PARTITION;
         REGISTRATION.SERVER1.SET_INSTANCE(CURRENT_PARTITION);

         -- Directs old partition to initialize its replacement and to start
         -- redirecting service calls.
         OLD_PARTITION.REPLACE(CURRENT_PARTITION);

         -- Blocks until all clients have been redirected to replacement.
         OLD_PARTITION.BLOCK_FOR_ZERO_CLIENTS;
         UNCHECKED_DEALLOCATION(OLD_PARTITION);
      end loop;
   end SERVER1_REPLACEMENT;

   -- Same as for Server1, substituting SERVER2_PARTITION for SERVER1_PARTITION
   task SERVER2_REPLACEMENT is
      ...
   end SERVER2_REPLACEMENT;

   procedure REPLACE_SERVER1 is
   begin
      SERVER1_REPLACEMENT.START_REPLACEMENT;
   end REPLACE_SERVER1;

   procedure REPLACE_SERVER2 is
   begin
      SERVER2_REPLACEMENT.START_REPLACEMENT;
   end REPLACE_SERVER2;

begin
   SERVER1_REPLACEMENT.FINISHED_SETUP;
   SERVER2_REPLACEMENT.FINISHED_SETUP;
end SERVER_NODE;
```

The overall system is controlled through the node CONTROL_NODE which is designated as the distinguished node. It's elaboration by the run-time system results in elaboration and initialization of the other nodes and partitions.

```
node CONTROLLER_NODE is
  pragma DISTINGUISHED;
end CONTROLLER_NODE;

with CLIENT_NODE;
with SERVER_NODE;
with REGISTRATION_PARTITION;
node body CONTROLLER_NODE is

    -- create registration partition and client node.
    REGISTRATION : REGISTRATION_PARTITION:= new REGISTRATION_PARTITION'PARTITION;
    CLIENTS: CLIENT_NODE := new CLIENT_NODE'NODE(REGISTRATION);

    -- handles interrupt to initiate server partition replacement
    task REPLACEMENT is
        entry SERVER1;
        for SERVER1 US at ...;

        entry SERVER2;
        for SERVER2 US at ...;
    end REPLACEMENT;

    task body REPLACEMENT is
        SERVERS : SERVER_NODE;
    begin
        SERVERS := new SERVER_NODE'NODE(REGISTRATION);
        loop
            select
                accept SERVER1 do
                  SERVERS.REPLACE_SERVER1;
                end SERVER1;
            or
                accept SERVER2 do
                  SERVERS.REPLACE_SERVER2;
                end SERVER2;
            end select;
        end loop;
    end REPLACEMENT;

end CONTROLLER_NODE;
```

The control node initiates the replacement process through the handling of an interrupt and calls the appropriate routine in the server node. The server node creates the replacement, informs the registration partition and starts its internal replacement task. The replacement task informs the replaced partition of the change and waits for all clients to be directed to the replacement, at which time the replaced partition can be deallocated. Note that initialization of the replacement partition instance with current state data is handled directly by the replaced instance.

In this section we have shown a solution to the replacement of server partitions. In the next section we show how client partitions can also be made replaceable.

5.3 Replaceable Client Partitions

Replacement of client partitions is potentially simpler than replacement of server partitions. If the only routine to call a client partition is the node which created it, then a replaced client partition instance can be deallocated as soon as the node has directed it to pass its state data to the replacement instance. If other routines had access to the client partition, then they must be informed of the change before the replacement instance is deallocated. If an unknown number of routines may have access, then a registration scheme would need to be implemented in much the same fashion as server partition replacement.

In our solution, the only changes necessary to make the client nodes replaceable are to the control node, which initiates replacements, and to the client partitions and node themselves. No change to the server partitions, server node, or registration partition is necessary. We have also generalized our solution to two client partitions.

```
public CLIENT1_PUBLIC is
  type CLIENT_DATA is
      record
          X : INTEGER;
          Y : INTEGER;
      end record;
end CLIENT1_PUBLIC;


with REGISTRATION_PARTITION;
with CLIENT1_PUBLIC;
partition CLIENT1_PARTITION (REGISTRATION : in REGISTRATION_PARTITION) is
    procedure INITIALIZE (DATA: in CLIENT1_PUBLIC.CLIENT_DATA := (...));
    procedure REPLACE (REPLACEMENT : CLIENT_PARTITION);
end CLIENT1_PARTITION;


with REGISTRATION_PARTITION;
with SERVER1_PARTITION;
with SERVER2_PARTITION;
with CLIENT1_PUBLIC;
partition body CLIENT1_PARTITION (REGISTRATION : in REGISTRATION_PARTITION) is

    STATE_DATA : CLIENT_PUBLIC.CLIENT_DATA;
    SERVER1 : SERVER1_PARTITION;
    SERVER2 : SERVER2_PARTITION;

    task COMPUTE is
        entry START;
        entry STOP;
    end COMPUTE;
    task body COMPUTE is
    begin
        accept START;
        SERVER1 := REGISTRATION.REGISTER_SERVER1;
```

```
        SERVER2 := REGISTRATION.REGISTER_SERVER2;
        loop
            select
                accept STOP;
                SERVER1.DEREGISTER;
                SERVER2.DEREGISTER;
                exit;
            else
                ----
                -- Perform normal execution, including service calls
                ----
                SERVER1.SERVICE_PROCEDURE(STATE_DATA.X,SERVER1);
                SERVER2.SERVICE_PROCEDURE(STATE_DATA.Y,SERVER2);
            end select;
        end loop;
    end COMPUTE;

    -- Initialize with data.  Use default data if necessary.
    procedure INITIALIZE (DATA: in CLIENT1_PUBLIC.CLIENT_DATA := (...)) is
    begin
        STATE_DATA := DATA;
        COMPUTATION.START;
    end INITIALIZE;

    -- Start the shutdown process on this partition and initialize the
    -- replacement server with current state data.
    procedure REPLACE (REPLACEMENT : CLIENT_PARTITION) is
    begin
        COMPUTE.STOP;
        REPLACEMENT.INITIALIZE(STATE_DATA);
    end REPLACE;

end CLIENT1_PARTITION;


with REGISTRATION_PARTITION;
node CLIENT_NODE (REGISTRATION : in REGISTRATION_PARTITION) is
     procedure REPLACE_CLIENT1;
     procedure REPLACE_CLIENT2;
end CLIENT_NODE;


with CLIENT1_PARTITION;
with CLIENT2_PARTITION;
with REGISTRATION_PARTITION;
node body CLIENT_NODE (REGISTRATION : in REGISTRATION_PARTITION) is


    task CLIENT1_REPLACEMENT is
        entry START_REPLACEMENT;
    end CLIENT1_REPLACEMENT;
    task body CLIENT1_REPLACEMENT is
        CURRENT_PARTITION : CLIENT1_PARTITION;
        OLD_PARTITION : CLIENT1_PARTITION;
    begin
        -- Create initial partition instance and Initialize with default data
        CURRENT_PARTITION := new CLIENT1_PARTITION'PARTITION(REGISTRATION);
        CURRENT_PARTITION.INITIALIZE;
        loop
            accept START_REPLACEMENT;
```

```
            OLD_PARTITION := CURRENT_PARTITION;
            CURRENT_PARTITION := new CLIENT1_PARTITION'PARTITION(REGISTRATION);

            -- directs the old partition to intitialize its replacement
            OLD_PARTITION.REPLACE(CURRENT_PARTITION);
            UNCHECKED_DEALLOCATION(OLD_PARTITION);
        end loop;
    end CLIENT1_REPLACEMENT;

    task CLIENT2_REPLACEMENT is
        -- Same as for Client1, using CLIENT2_PARTITION
    end CLIENT2_REPLACEMENT;

    procedure REPLACE_CLIENT1 is
    begin
      CLIENT1_RELACEMENT.START_REPLACEMENT;
    end REPLACE_CLIENT1;

    procedure REPLACE_CLIENT2 is
    begin
      CLIENT2_RELACEMENT.START_REPLACEMENT;
    end REPLACE_CLIENT2;

end CLIENT_NODE;

node CONTROLLER_NODE is
  pragma DISTINGUISHED;
end CONTROLLER_NODE;

with CLIENT_NODE;
with SERVER_NODE;
with REGISTRATION_PARTITION;
node body CONTROLLER_NODE is

    -- create the registration partition
    REGISTRATION: REGISTRATION_PARTITION := new REGISTRATION_PARTITION'PARTITION;

    task REPLACEMENT is
        entry SERVER1;
        for SERVER1 US at ...;

        entry SERVER2;
        for SERVER2 US at ...;

        entry CLIENT1;
        for CLIENT1 US at ...;

        entry CLIENT2;
        for CLIENT2 US at ...;
    end REPLACEMENT;

    task body REPLACEMENT is
        SERVERS : SERVER_NODE;
        CLIENTS: CLIENT_NODE;
    begin
        -- create the server and client nodes.  They in turn create the
        -- client and server partitions
        SERVERS := new SERVER_NODE'NODE(REGISTRATION);
        CLIENTS := new CLIENT_NODE'NODE(REGISTRATION);
        loop
```

```
            select
                accept SERVER1 do
                  SERVERS.REPLACE_SERVER1;
                end REPLACE_SERVER1;
            or
                accept SERVER2 do
                  SERVERS.REPLACE_SERVER2;
                end REPLACE_SERVER2;
            or
                accept CLIENT1 do
                  CLIENTS.REPLACE_CLIENT1;
                end REPLACE_SERVER2;
            or
                accept CLIENT2 do
                  CLIENTS.REPLACE_CLIENT2;
                end REPLACE_SERVER2;
            end select;
        end loop;
    end REPLACEMENT;

end CONTROLLER_NODE;
```

6 Node Replacement

In this section, we illustrate online replacement of the server node. Upon server node replacement, a new server node instance is created along with its corresponding new server partition instances. The replaced server node instance then directs a remote replacement of its partitions with the corresponding partitions on the new node. Once all clients (of both server partitions) have been redirected to the new server partitions, the replaced node deallocates its partitions and the control node deallocates the server node.

Only the server node and control node require changes from the server partition replacement introduced earlier. Revised versions of these nodes are presented below.

```
with REGISTRATION_PARTITION;
node SERVER_NODE (REGISTRATION : in REGISTRATION_PARTITION) is
    procedure REPLACE_SERVER1;
    procedure REPLACE_SERVER2;
    procedure REPLACE_NODE;
    procedure INITIALIZE_PARITITIONS;
end SERVER_NODE;

with SERVER1_PARTITION;
with SERVER2_PARTITION;
with REGISTRATION_PARTITION;
node body SERVER_NODE (REGISTRATION : in REGISTRATION_PARTITION) is

    task SERVER1_REPLACEMENT is
        entry FINISHED_SETUP;
        entry INIT_PARTITION;
        entry LOCAL;
        entry REMOTE (NEW_PARTITION : in SERVER1_PARTITION);
```

```
   entry FINISHED_REMOTE;
end SERVER1_REPLACEMENT;

task body SERVER1_REPLACEMENT is
   CURRENT_PARTITION : SERVER1_PARTITION;
   OLD_PARTITION : SERVER1_PARTITION;
begin
   CURRENT_PARTITION := new SERVER1_PARTITION'PARTITION;
   REGISTRATION.SERVER1.SET_INSTANCE(CURRENT_PARTITION);
   accept FINISHED_SETUP;
   loop
      select
         accept INIT_PARTITION;
         CURRENT_PARTITION.INITIALIZE(CURRENT_PARTITION);
      or
         accept LOCAL;
         OLD_PARTITION := CURRENT_PARTITON;
         CURRENT_PARTITION := new SERVER1_PARTITION'PARTITION;
         REGISTRATION.SERVER1.SET_INSTANCE(CURRENT_PARTITION);

         -- Directs old partition to initialize its replacement and to start
         -- redirecting service calls.
         OLD_PARTITION.REPLACE(CURRENT_PARTITION);

         -- Blocks until all clients have been redirected to replacement.
         OLD_PARTITION.BLOCK_FOR_ZERO_CLIENTS;
         UNCHECKED_DEALLOCATION(OLD_PARTITION);
      or
         accept REMOTE (NEW_PARTITION : in SERVER1_PARTITION) do
           OLD_PARTITION := CURRENT_PARTITION;
           CURRENT_PARTITION := NEW_PARTITION;
         end REMOTE;

         -- Directs old partition to initialize its replacement and to start
         -- redirecting service calls.
         OLD_PARTITION.REPLACE(CURRENT_PARTITION);

         -- Blocks until all clients have been redirected to replacement.
         OLD_PARTITION.BLOCK_FOR_ZERO_CLIENTS;
         UNCHECKED_DEALLOCATION(OLD_PARTITION);
         accept FINISHED_REMOTE;
      end select
   end loop;
end SERVER1_REPLACEMENT;

procedure REPLACE_SERVER1 is
begin
   SERVER1_REPLACEMENT.LOCAL;
end REPLACE_SERVER1;

-- Same as for Server1, substituting SERVER2_PARTITON for SERVER1_PARTITION
task SERVER2_REPLACEMENT is...
procedure REPLACE_SERVER2 is...

-- Only want the partitions intialized with default data (this routine)
-- if this was the very first server node created.  In other cases,
-- the replaced partitions will directly initialize the new paritions.
procedure INITIALIZE_PARITITIONS is
begin
   SERVER1_REPLACEMENT.INIT_PARTITION;
```

```
        SERVER2_REPLACEMENT.INIT_PARTITION;
    end INITIALIZE_PARITITIONS;

    -- Inform this node that it is being replace.  Obtain references to the
    -- new service partitions and perform a remote replacement to them.
    -- This routine will not return until all clients have been redirected
    -- to the new partitions and the local partitions have been deallocated.
    -- This node can therefore be safely deallocated upon return.
    procedure REPLACE_NODE is
        NEW_SERVER1 : SERVER1_PARTITION;
        NEW_SERVER2 : SERVER2_PARTITION;
    begin
        NEW_SERVER1 := REGISTRATION_PARTITION.SERVER1.REGISTER;
        NEW_SERVER2 := REGISTRATION_PARTITION.SERVER2.REGISTER;

        SERVER1_REPLACEMENT.REMOTE(NEW_SERVER1);
        SERVER2_REPLACEMENT.REMOTE(NEW_SERVER2);

        SERVER1_REPLACEMENT.FINISHED_REMOTE;
        SERVER2_REPLACEMENT.FINISHED_REMOTE;

        NEW_SERVER1.DEREGISTER;
        NEW_SERVER2.DEREGISTER;
    end REPLACE_NODE;

begin
    SERVER1_REPLACEMENT.FINISHED_SETUP;
    SERVER2_REPLACEMENT.FINISHED_SETUP;
end SERVER_NODE;

node CONTROLLER_NODE is
  pragma DISTINGUISHED;
end CONTROLLER_NODE;

with CLIENT_NODE;
with SERVER_NODE;
with REGISTRATION_PARTITION;
node body CONTROLLER_NODE is

    -- create registration partition and client node
    REGISTRATION: REGISTRATION_PARTITION := new REGISTRATION_PARTITION'PARTITION;
    CLIENTS: CLIENT_NODE := new CLIENT_NODE'NODE(REGISTRATION);

    task REPLACEMENT is
        entry SERVER1;
        for SERVER1 US at ...;

        entry SERVER2;
        for SERVER2 US at ...;

        entry NODE_REPLACE;
        for NODE_REPLACE US at ...;
    end REPLACEMENT;
    task body REPLACEMENT is
        CURRENT_SERVERS : SERVER_NODE;
        OLD_SERVERS : SERVER_NODE;
    begin
        -- create initial server node
        CURRENT_SERVERS := new SERVER_NODE'NODE(REGISTRATION);
        CURRENT_SERVERS.INITIALIZE_PARTITIONS;
```

```
        loop
            select
                accept SERVER1 do
                  SERVERS.REPLACE_SERVER1;
                end REPLACE_SERVER1;
            or
                accept SERVER2 do
                  SERVERS.SERVER2_SUPPORT.REPLACE_PARTITION;
                end REPLACE_SERVER2;
            or
                -- Create new server node and inform old server that it is
                -- being replaced.    The old server controls the remote
                -- replacement of the partitions from it to the new server node.
                -- Then, deallocate the replaced server node
                accept NODE_REPLACE;
                OLD_SERVERS := CURRENT_SERVERS;
                CURRENT_SERVERS := new SERVER_NODE'NODE(REGISTRATION);
                OLD_SERVERS.REPLACE_NODE;
                UNCHECKED_DEALLOCATION(OLD_SERVERS);

            end select;
        end loop;
    end REPLACEMENT;

end CONTROLLER_NODE;
```

7 Conclusions and Future Work

In this paper, we have discussed online replacement of program units using AdaPT. Specifically, we have summerized the major features of AdaPT, presented a replacement taxonomy and introduced solutions to specific cases in the taxonomy.

Features of the replacement taxonomy include the following classifications:

- type of replacement (planned, unplanned)
- unit type (partition, node)
- location (local, remote)
- presence of state
- specification change

We have included solutions to the problems of planned partition and node replacement using dynamic linking and loading of the latest compiled versions of the unit. In each of our solutions, partition state is maintained across the replacement process. The node replacement example includes remote replacement.

Further cases in the taxonomy which we propose to study include replacement of units with specification changes. The general case can be considered a sequence of changes with: 1) no specification changes, 2) only specification increases, and 3) only specification decreases.

We anticipate the use of a compilation tool which will allow specification changes without recompilation of the entire system, if it can be determined that units are unaffected by the change, and online inclusion of the new unit into the system in much the same way as we have shown in this paper. In particular, if the specification of a unit is increased, there will be no immediate impact on other units in the system and there is no need to recompile them. At a later point other units may themselves be upgraded, utilizing the new features of the increased specification.

We also envision the handling of reduced specifications. The compiler or a post-compilation tool such as ASIS[4] can be used to check references to the items in the specification of the unit being replaced. If the compiler or tool can determine that no other unit in the system references an item, that item can be safely removed from the new specification. Because there are no references to the removed item, other units are unaffected and need not be recompiled.

References

[1] A. B. Gargaro, S. J. Goldsack, R. A. Volz, and A. J. Wellings, "A Proposal to Support Reliable Distributed Systems in Ada 9X," Technical Report 90-10, Department of Computer Science, Texas A&M University, College Station, Texas, 1990.

[2] A. A. Holzbacher-Valero, S. J. Goldsack, R. Volz and R. Waldrop. "Transforming AdaPT to Ada83," Status Report, subcontract #074 cooperative agreement NCC-9-16.

[3] R. S. Waldrop, R. A. Volz, G. W. Smith,A. A. Holzbacher-Valero and S. J. Goldsack. "On-Line Replacement of Program Modules Using AdaPT," Status Report, subcontract #074 cooperative agreement NCC-9-16.

[4] J. B. Bladen, S. J. Blake, and D. Spenhoff. "Ada Semantic Interface Specification (ASIS)," TRI-Ada '91 Conference Proceedings, October 1991.