RATIONAL MACHINES INSTRUCTION SET

VERSION 1.Ø

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

This document describes the instruction set as defined by the **Rational Machines Architecture.** In particular, we provide detailed information regarding the composition and functionality of each instruction. Implementation-specific formats may be found in a corresponding processor reference manual. Additionally, the **Rational Machines Systems Concept** document includes a rationale for the organization of this instruction set. In each of these documents, we presume that the reader has an understanding of the semantics of the Ada* programming language.

This document is divided into three major sections, namely:

* GENERAL CONCEPTS -- Chapter 2
* DETAILED DISCUSSION -- Chapters 3 - 9
* SUMMARY INFORMATION -- Chapters 10 - 14

Chapter 2 introduces the primitive data types and exceptions that are recognized by the instruction set. Chapters 3 through 9 are organized by groups of functionally related instructions. These seven chapters provide detailed information on the form and effect of every instruction and their options. The last five chapters, Chapters 10 through 14, are provided as a convenience to the reader to aid in locating specific instruction set information.

For detailed information regarding the organization of each class of stacks as defined by the architecture, consult the **Rational Machines Run-time Structure.**

* Ada is a trademark of the Department of Defense, Ada Joint Program Office

CHAPTER 2
GENERAL CONCEPTS

The Rational Machines instruction set directly supports and encourages the use of modern software engineering methodologies. In particular, the instruction set is optimized for supporting the use of object-oriented programming in Ada-like languages. The design of the instruction set is heavily influenced by the premise that a well-structured program consists of many small modular components with controlled and well-specified interfaces.

Every program *segment* consists of one or more *words*, where each word contains one or more *instructions*. A program segment represents either a task or a package, and so the number of words per segment will vary. On the other hand, the number of instructions per word is generally a fixed number for each implementation. Each instruction is further divided into an *opcode* and one or more *fields* which provide operand information for the instruction.

2.1 DATA TYPES

The Rational Machines instruction set is *strongly typed*, which means that there exists a unique and well-defined set of operations associated with every primitive *data type* recognized by the architecture. No other operations are legal, and furthermore, *objects* of incompatible types may not implicitly operate with each other.

This elementary set of data types was designed to directly and efficiently support the semantics of high-order programming languages similar to Ada. Collectively, we call these primitive data types the **OPERAND_CLASS.** The operations associated with objects of each OPERAND_CLASS are found in Chapter 11, the OBJECT/OPERATION CROSS-REFERENCE.

The following types are recognized by the architecture:

```
type OPERAND_CLASS is
  (ACCESS_CLASS,         ANY_CLASS,
   ARRAY_CLASS,          DISCRETE_CLASS,
   ENTRY_CLASS,          EXCEPTION_CLASS,
   FAMILY_CLASS,         FLOAT_CLASS
   MATRIX_CLASS,         PACKAGE_CLASS,
   RECORD_CLASS,         SEGMENT_CLASS,
   SELECT_CLASS,         SUBARRAY_CLASS,
   SUBMATRIX_CLASS,      SUBVECTOR_CLASS,
   TASK_CLASS,           VARIANT_RECORD_CLASS,
   VECTOR_CLASS);
```

Consult the **Rational Machines Run-time Structure** for a complete explanation regarding the *characteristics* of each of these types, and their representation on the various machine stacks. In the following sections we provide a summary description of the primitive types.

2.1.1 **ACCESS_CLASS** Denotes a pointer to an object of a specific type.

Characteristics of ACCESS_CLASS objects include the type of the designated access objects, reference to a specific collection and its creator, and an indication of the designated objects being reclaimable and/or homogeneous.

2.1.2 **ANY_CLASS** Denotes an object of arbitrary type. ANY_CLASS objects are operands only of DECLARE_VARIABLE and EXECUTE, and so represent generic declarative and imperative instructions. Characteristics of ANY_CLASS objects are determined at execution time.

2.1.3 **ARRAY_CLASS** Denotes a composite object consisting of components of the same component type, indexed by n-dimensions. Characteristics of ARRAY_CLASS objects include reference to the element type, dimension data, item size, and subarray size.

2.1.4 **DISCRETE_CLASS** Denotes an enumeration or integer object. Characteristics of DISCRETE_CLASS objects include reference to its minimum and maximum values, and define if the object is unsigned.

2.1.5 **ENTRY_CLASS** Denotes an entry of a task. Characteristics of ENTRY_CLASS objects include the entry name and queue information.

2.1.6 **EXCEPTION_CLASS** Denotes an exception. Characteristics of EXCEPTION_CLASS objects include its identity, scope, and where it was raised.

2.1.7 **FAMILY_CLASS** Denotes a family of entries. Characteristics of FAMILY_CLASS objects include the entry name, queue information, and references to members of the family.

2.1.8 **FLOAT_CLASS** Denotes a floating point value. Characteristics of FLOAT_CLASS objects include references to its minimum and maximum values and its accuracy.

2.1.9 **MATRIX_CLASS** Denotes a two dimensional array. Characteristics of MATRIX_CLASS objects are identical to ARRAY_CLASS objects, except that MATRIX_CLASS objects are used entirely to support EXECUTE instruction optimizations.

2.1.10 **PACKAGE_CLASS** Denotes a package. Characteristics of PACKAGE_CLASS objects include the declarative level, privacy, import, and generic information, and references to its corresponding code segment.

2.1.11 **RECORD_CLASS** Denotes a composite object consisting of named components, which may be of different types. Characteristics of RECORD_CLASS objects include a descriptor for each named component, which in turn specify the field type, size, and placement within the composite object.

2.1.12 **SELECT_CLASS** Denotes an executable object that handles processing of a task select statement. Characteristics of SELECT_CLASS objects include reference to the select statement and additionally provide information regarding component clauses such as accept, delay, terminate, and else.

2.1.13 **SEGMENT_CLASS** Denotes a code segment. Objects of this type are created and used to transform a data segment of some form into executable code. Characteristis of SEGMENT_CLASS objects include its size and its root.

2.1.14 **SUBARRAY_CLASS** Denotes an n-1 dimensional array as a substructure of an n-dimensional parent. SUBARRAY_CLASS objects are used entirely to support EXECUTE optimizations. Characteristics of SUBARRAY_CLASS objects include reference to the element type, dimension data, and item size.

2.1.15 **SUBMATRIX_CLASS** Denotes a one dimensional array as a substructure of a two dimensional parent. SUBMATRIX_CLASS objects are used entirely to support EXECUTE optimizations. Characteristics of SUBMATRIX_CLASS objects include reference to the element type, dimension data, and item size.

2.1.16 **SUBVECTOR_CLASS** Denotes a slice of a one dimensional parent. SUBVECTOR_CLASS objects are used to entirely to support EXECUTE optimizations. Characteristics of SUBVECTOR_CLASS objects include reference to the element type, dimension data, and item data.

2.1.17 **TASK_CLASS** Denotes a task. Characteristics of TASK_CLASS objects include the declarative level, privacy, import, entry, and generic information, and references to its corresponding code segment.

2.1.18 **VARIANT_RECORD_CLASS** Denotes a discriminated union of objects consisting of named components, which may be of different types. There exists a fixed part of the record object common to all variant, and which contains a discriminant field indicating which one of the possible variants is contained in a particular instance. Characteristics of VARIANT_RECORD_CLASS objects include references to the fixed and variant parts, size and number of the discriminants, and references to the nature of the variant fields.

VECTOR_CLASS

2.2 EXCEPTIONS

The Rational Machines instruction set defines facilities for dealing with errors that arise during program execution. In particular, the instruction set recognizes several different *exceptions* that cause suspension of normal program execution. These exceptions include:

```
ALLOCATION_ERROR,        CAPABILITY_ERROR,
CONSTRAINT_ERROR,        ELABORATION_ERROR,
INSTRUCTION_ERROR,       MACHINE_RESTRICTION,
NUMERIC_ERROR,           OPERAND_CLASS_ERROR,
RESOURCE_ERROR,          SELECT_ERROR,
SOME_ERROR,              TASKING_ERROR,
TYPE_ERROR,              VISIBILITY_ERROR       : exception;
```

Chapter 12, EXCEPTION/INSTRUCTION CROSS-REFERENCE, lists each exception and the instructions that may raise that exception. In the following sections, we summarize the conditions that under which each exception may be raised.

2.2.1 **ALLOCATION_ERROR** Not yet implemented.

2.2.2 **CAPABILITY_ERROR** Raised when attempting to access an entity that is private or otherwise out of scope.

2.2.3 **CONSTRAINT_ERROR** Raised in any of the following situations: upon an attempt to violate a range constraint, an index constraint, or a discriminant constraint; upon attempt to use a record component that does not exist for the current discriminant values; and upon attempt to use a selected component, an indexed component, a slice, or an attribute, of an object designated by an access value if the object does not exist because of the access value is null.

2.2.4 **ELABORATION_ERROR** Raised when attempting to access an entity other than a program unit that is not yet completely elaborated.

2.2.5 **INSTRUCTION_ERROR** Raised when the machine attempts to execute an illegal instruction.

2.2.6 **MACHINE_RESTRICTION** Raised when attempting to create an object that is larger then the machine can allocate or index.

2.2.7 **NUMERIC_ERROR** Raised by the execution of a predefined numeric operation that cannot deliver the mathematical result, and for real types, within the declared accuracy.

2.2.8 **OPERAND_CLASS_ERROR** Raised when attempting to perform an operation that is illegal for an object of the given OPERAND_CLASS.

2.2.9 **RESOURCE_ERROR** Raised when unable to extend a CONTROL_STACK, DATA_STACK, or TYPE_STACK.

2.2.10 **SELECT_ERROR** Raised during the execution of a selective wait statement that has no else part, if this execution determines that all alternatives are closed.

2.2.11 **SOME_ERROR** Raised upon an attempt to call a subprogram, activate a task, or instantiate a generic unit before elaboration of the corresponding unit body. In addition, the exception is raised when dynamic storage allocation of a task or collection of designated access objects is exceeded.

2.2.1 **TASKING_ERROR** Raised when exceptions arise during intertask communication.

2.2.13 **TYPE_ERROR** Raised when attempting to perform an invalid type derivation.

2.2.1 **VISIBILITY_ERROR** Raised when attempting to access an entity that is not currently visible.

2.3 STATE

Each processor within a given implementation contains a *program counter* that points to the currently executing instruction. This program counter contains an *address* that refers to a specific segment, a word within that segment (the *offset*), and an instruction within the word (the *index*). Generally, the *state* of a given program segment includes the value of its corresponding program counter plus the condition of each of the stacks associated with that segment.

CHAPTER 3
IMPERATIVE INSTRUCTIONS

An *imperative instruction* invokes an operation upon an object of a given type. This class of instructions is perhaps the most important one, since its semantics forms the key to defining and enforcing data abstraction and information hiding at even the lowest levels of the architecture. Imperative instructions include the following opcodes:

* ACTION -- Perform a system level operation
* CALL -- Invoke a subprogram, block, accept, or select
* EXECUTE -- Perform an operation upon a typed object

In the following sections, we treat each opcode in detail.

3.1 ACTION

The ACTION instruction performs a system level operation. Formally, ACTION takes the form:

```
type ACTION_INSTRUCTION is
  record
    TO_PERFORM : UNCLASSED_ACTION;
  end record;
```

The operation TO_PERFORM is of the type **UNCLASSED_ACTION,** which we further define as:

```
type UNCLASSED_ACTION is
  (-- ACTIVATION_OPERATIONS
       ACCEPT_ACTIVATION,           ACTIVATE_TASKS,
       SIGNAL_ACTIVATED,            SIGNAL_COMPLETION,
   -- CREATION_OPERATIONS
       MAKE_NULL_UTILITY,           MAKE_SELF,
       NAME_MODULE,                 NAME_PARTNER,
   -- IMPORT_OPERATIONS
       INTRODUCE_IMPORT,            OVERWRITE_IMPORT,
       REMOVE_IMPORT,
   -- INTERFACE_OPERATIONS
       ALTER_BREAK_MASK,            BREAK_OPTIONAL,
       BREAK_UNCONDITIONAL,         ESTABLISH_FRAME,
       EXIT_BREAK,                  QUERY_BREAK_ADDRESS,
       QUERY_BREAK_CAUSE,           QUERY_BREAK_MASK,
       QUERY_FRAME,                 SET_BREAK_MASK,
       SET_INTERFACE_SCOPE,         SET_INTERFACE_SUBPROGRAM,
   -- NO_OPERATION
       IDLE,
   -- REFERENCE_OPERATIONS
       ACTIVATE_SUBPROGRAM,         CALL_REFERENCE,
       DELETE_ITEM,                 DELETE_SUBPROGRAM,
       SET_VISIBILITY,
   -- RESOURCE_OPERATIONS
```

```
              QUERY_RESOURCE_LIMITS,      QUERY_RESOURCE_STATE,
              RECOVER_RESOURCES,          RETURN_RESOURCES,
              SET_RESOURCE_LIMITS,
          -- SLEEP_OPERATIONS
              INITIATE_DELAY,             PROPOGATE_ABORT,
          -- STACK_OPERATIONS
              MARK_AUXILIARY,             MARK_DATA,
              MARK_TYPE,                  POP_AUXILIARY,
              POP_CONTROL,                POP_DATA,
              POP_TYPE,                   PUSH_CONTROL,
              SWAP_CONTROL);
```

In the following sections we provide a detailed description of each UNCLASSED_ACTION.

3.1.1 **ACTIVATION_OPERATIONS** These operations provide the protocol for the activation of a task or package. Since the Rational Machines architecture treats each subprogram as subordinate to another package or task, subprogram activation is achieved with a different set of instructions, namely the CALL instruction (section 3.2) for activating locally declared subprograms, and REFERENCE_OPERATION ACTION instructions (section 3.1.6) for activating remote, yet visible, subprograms.

ACTIVATION_OPERATIONS include:

* ACCEPT_ACTIVATION
* ACTIVATE_TASKS
* SIGNAL_ACTIVATION
* SIGNAL_COMPLETION

3.1.1.1 <u>ACCEPT_ACTIVATION</u>

* PURPOSE: Signal that elaboration of visible part of module is complete; module is now ready to accept activation from the parent.
* FUNCTION: Change module current mode to ACTIVATING, and send the message NOFIFY_DECLARED to the declaring module.
* STACKS: No change except due to message passage.
* EXCEPTIONS: None

3.1.1.2 <u>ACTIVATE_TASKS</u>

* PURPOSE: Signal all children tasks that they may begin execution.
* FUNCTION: Send the message ACTIVATE_MODULE to each child; execution of the current module may proceed once all children have been successfully activated.
* STACKS: No change except due to message passage.
* EXCEPTIONS: TASKING_ERROR may be raised if a child cannot be activated.

3.1.1.3 <u>SIGNAL_ACTIVATED</u>

* PURPOSE: Signal the creator of a module that

elaboration of the module body is complete. For a package module, this means that the package body has been executed; in the case of a task module, this means that the module is activated and is running concurrently with the parent.

* FUNCTION: Change module current mode to EXECUTING, and send the message NOTIFY_ACTIVATED to the declaring module.
* STACKS: No change except due to message passage.
* EXCEPTIONS: None.

3.1.1.4 SIGNAL_COMPLETION

* PURPOSE: Signal the creator of a module that processing of the module is complete.
* FUNCTION: If module is a task, mark the current mode as TERMINATING and wait for all dependent children to terminate. Additionally, purge any entry queues and send the message END_RENDEZVOUS to any waiting callers. Once all children are terminated or are ready to terminate, send the message NOTIFY_TERMINATION to the declaring module. When deallocation of the dependent children and the module itself begins, the module current mode is marked as COMPLETED.
  If module is a package, wait for all dependent children to terminate. Once all children are terminated, send the message NOTIFY_TERMINABLE to the declaring module. Start deallocation of the dependent children and the module itself, and mark the module current mode as TERMINATED.
* STACKS: Postcondition:
  QUEUE_STACK is purged.
  No other change except due to message passage.
* EXCEPTIONS: If module is a task, and callers are waiting in any entry queues, the message END_RENDEZVOUS has the side effect of raising TASKING_ERROR in any calling tasks.

3.1.2 **CREATION_OPERATIONS** These operations provide a facility for the creating new subprograms, packages, or tasks within the current context.

CREATION_OPERATIONS include:

* MAKE_NULL_UTILITY
* MAKE_SELF
* NAME_MODULE
* NAME_PARTNER

3.1.2.1 MAKE_NULL_UTILITY

* PURPOSE: Create a null subprogram variable.

* FUNCTION: Push a null SUBPROGRAM_VAR control word on the CONTROL_STACK of the current module.
* STACKS: Postcondition:
  SUBPROGRAM_VAR word pushed on top of CONTROL_STACK
* EXCEPTIONS: None.

3.1.2.2 MAKE_SELF

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.2.3 NAME_MODULE

* PURPOSE: Create a module variable.
* FUNCTION: Create a module variable identical to that of the current module (either a TASK_VAR or a PACKAGE_VAR), and push a corresponding control word on the CONTROL_STACK of the current module.
* STACKS: Postcondition:
  Push a TASK_VAR or a PACKAGE_VAR on top of CONTROL_STACK.
* EXCEPTIONS: None.

3.1.2.4 NAME_PARTNER

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.3 **IMPORT_OPERATIONS** These operations provide facilities for manipulating the IMPORT_STACK.
IMPORT_OPERATIONS include:

* INTRODUCE_IMPORT
* OVERWRITE_IMPORT
* REMOVE_IMPORT

3.1.3.1 INTRODUCE_IMPORT

* PURPOSE: Add an import item to a given module.
* FUNCTION: Pop the CONTROL_STACK to determine the target of import, and follow the type path to access its current import information on its corresponding TYPE_STACK. Pop the CONTROL_STACK again to determine the entity that is to be imported, and add a reference to that entity in the target module's IMPORT_STACK, which is extended if necessary. A path is added to the target module's TYPE_STACK leading from the target module's import information and refering to the imported entity.

* STACKS: Precondition:
  Top of CONTROL_STACK must contain a MODULE_VAR.
  Top - 1 of CONTROL_STACK must contain an IMPORT_VA
  Postcondition:
  Top of CONTROL_STACK is reduced by two.
  IMPORT_VAR is added to target module IMPORT_STACK.
  Path is added to target module's TYPE_STACK from its import information and refering to the imported entity.
* EXCEPTIONS: CAPABILITY_ERROR may be raised if the module that is the target of the import is not statically nested relative to the current module.

3.1.3.2 <u>OVERWRITE_IMPORT</u>

* PURPOSE: Write over an import item in a given module.
* FUNCTION: Pop the CONTROL_STACK to determine the target of import, and follow the type path to access its current import information on the corresponding TYPE_STACK. Pop the CONTROL_STACK again to determine the site of the existing import that is to be overwritten. Pop the IMPORT_STACK at that site to remove the import entity.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain a MODULE_VAR.
  Top - 1 of CONTROL_STACK must contain a DISCRETE_VAR indicating the site scope delta.
  Postcondition:
  Top of CONTROL_STACK is reduced by two.
  IMPORT_VAR is removed from target module IMPORT_STACK at the given site.
* EXCEPTIONS: CONSTRAINT_ERROR may be raised if an import does not already exist at the given site.

3.1.4 **INTERFACE_OPERATIONS** These operations provide an interface between modules and their external environment. INTERFACE_OPERATIONS primarily are used in support of the programming environment breakpoint and debugging facilities.

INTERFACE_OPERATIONS include:

* ALTER_BREAK_MASK
* BREAK_OPTIONAL
* BREAK_UNCONDITIONAL
* ESTABLISH_FRAME
* EXIT_BREAK
* QUERY_BREAK_ADDRESS
* QUERY_BREAK_CAUSE
* QUERY_BREAK_MASK
* QUERY_FRAME
* SET_BREAK_MASK
* SET_INTERFACE_SCOPE
* SET_INTERFACE_SUBPROGRAM

3.1.4.1 <u>ALTER_BREAK_MASK</u>

* PURPOSE: Modify the breakpoint mask for the current module.
* FUNCTION: Pop the CONTROL_STACK to get the site of the current module key, and then read the INTERFACE_KEY value at that site. Pop the CONTROL_STACK again to access the new breakpoint mask. Decode the mask, and write the new value back to the INTERFACE_KEY.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain a VARIABLE_REF the points to the INTERFACE_KEY site.
  Top - 1 of CONTROL_STACK must contain a DISCRETE_VAR that contains encoded breakpoint information.
  Postcondition:
  Top of CONTROL_STACK is reduced by two.
  Module INTERFACE_KEY is altered.
* EXCEPTIONS: CAPABILITY_ERROR may be raised if key site is not local to the current module.
  OPERAND_CLASS_ERROR is raised if INTERFACE_KEY is not found.

3.1.4.2 BREAK_OPTIONAL

* PURPOSE: Force a breakpoint action if only if breakpoints are enabled.
* FUNCTION: Examine the currently enabled breakpoint conditions, and if optional breakpoints are set, raise the BREAKPOINT_ACTION exception.
* STACKS: No change.
* EXCEPTIONS: BREAKPOINT_ACTION may be raised.

3.1.4.3 BREAK_UNCONDITIONAL

* PURPOSE: Force an unconditional breakpoint action.
* FUNCTION: Raise the BREAKPOINT_ACTION exception.
* STACKS: No change.
* EXCEPTIONS: BREAKPOINT_ACTION is raised.

3.1.4.4 ESTABLISH_FRAME

* PURPOSE: Establish a new frame on the current CONTROL_STACK
* FUNCTION: Pop the CONTROL_STACK to access the site of the current INTERFACE_KEY. Pop the CONTROL_STACK again to access the depth of the frame to be established. Pop the CONTROL_STACK a third time to get the name of the corresponding code segment. Pop the CONTROL_STACK a fourth time to get the displacement within the segment marking the start of the executable code. Next, trace down the current activation links to find the frame at the requested depth. Push the state of the frame (ACCESSIBLE, INACCESSIBLE, NON_EXISTANT) on

the CONTROL_STACK. If the frame is ACCESSIBLE, mark the CONTROL_STACK to indicate the creation of a new frame, with a subprogram using the given code segment and displacement.

* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR indicating the INTERFACE_KEY_SITE.
  Top - 1 of CONTROL_STACK contains a DISCRETE_VAR indicating the frame depth.
  Top - 2 of CONTROL_STACK contains a DISCRETE_VAR indicating the name of a code segment.
  Top - 3 of CONTROL_STACK contains a DISCRETE_VAR indicating the start of executable code within the code segment.
  Postcondition:
  Top of CONTROL_STACK reduced by four, and then FRAME_STATUS is pushed, followed by a new ACTIVATION_STATE and a new ACTIVATION_LINK.
* EXCEPTIONS: None.

3.1.4.5 EXIT_BREAK

* PURPOSE: Exit the current frame and establish breakpoints.
* FUNCTION: The current INTERFACE_KEY is accessed, and breakpoints are enabled according to the key. The current frame is then popped.
* STACKS: Postcondition:
  The CONTROL_STACK, DATA_STACK, and TYPE_STACK are popped to remove the outer frame.
* EXCEPTIONS: INSTRUCTION_ERROR may be raised if the INTERFACE_KEY is not found, or is found beyond the top of the CONTROL_STACK.

3.1.4.6 QUERY_BREAK_ADDRESS

* PURPOSE: Push the current breakpoint address on the CONTROL_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the site of the INTERFACE_KEY. Locate the key, and push its BREAK_ADDRESS value on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR indicating the key site.
  Postcondition:
  Top of CONTROL_STACK now contains a DISCRETE_VAR indicating the BREAK_ADDRESS.
* EXCEPTIONS: CAPABILITY_ERROR may be raised if key site is not local to the current module.
  OPERAND_CLASS_ERROR raised if INTERFACE_KEY is not found.

3.1.4.7 QUERY_BREAK_CAUSE

* PURPOSE: Push the current breakpoint cause on the CONTROL_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the site of the INTERFACE_KEY. Locate the key, and push its BREAK_CAUSE value on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR indicating the key site.

  Postcondition:
  Top of CONTROL_STACK now contains a DISCRETE_VAR indicating the BREAK_CAUSE.
* EXCEPTIONS: CAPABILITY_ERROR may be raised if key site is not local to the current module.

  OPERAND_CLASS_ERROR raised if INTERFACE_KEY is not found.

3.1.4.8 QUERY_BREAK_MASK

* PURPOSE: Push the current breakpoint mask on the CONTROL_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the site of the INTERFACE_KEY. Locate the key, and push its RESTORE_ENABLE value on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR indicating the key site.

  Postcondition:
  Top of CONTROL_STACK now contains a DISCRETE_VAR indicating the encoded RESTORE_ENABLE mask.
* EXCEPTIONS: CAPABILITY_ERROR may be raised if key site is not local to the current module.

  OPERAND_CLASS_ERROR raised if INTERFACE_KEY is not found.

3.1.4.9 QUERY_FRAME

* PURPOSE: Push the current state of the current frame on the CONTROL_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the site of the current INTERFACE_KEY. Pop the CONTROL_STACK again to access the depth of the frame to be queried. If the frame is ACCESSIBLE, push the scope of the outer frame on the CONTROL_STACK, and push the return address on the CONTROL_STACK. In all cases, next push the encoded state of the frame (ACCESSIBLE, INACCESSIBLE, NON_EXISTANT) on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR indicating the INTERFACE_KEY SITE.
  Top - 1 of CONTROL_STACK contains a DISCRETE_VAR indicating the frame depth.

  Postcondition:

Top of CONTROL_STACK reduced by four, and then, if the frame is ACCESSIBLE, push a DISCRETE_VAR indicating the frame outer scope, followed by a DISCRETE_VAR indicating the frame return address. Finally, push a DISCRETE_VAR indicating the encoded frame state.

* EXCEPTIONS: None.

3.1.4.10 SET_BREAK_MASK

* PURPOSE: Establish a new breakpoint mask and debugging information for the current frame.
* FUNCTION: Pop the CONTROL_STACK to access the name of the target module. Pop the CONTROL_STACK again to get the new value of the breakpoint mask. Write the value to that module's DEBUGGING_INFO on the CONTROL_STACK, and set BREAKPOINT_ON in the CONTROL_STATE, also on the CONTROL_STACK.
* STACKS: Precondition:
  - Top of CONTROL_STACK contains a DISCRETE_VAR indicating the module name.
  - Top - 1 of CONTROL_STACK contains a DISCRETE_VAR indicating the new breakpoint mask.

  Postcondition:
  - Top of CONTROL_STACK reuduce by two.
* EXCEPTIONS: None.

3.1.4.11 SET_INTERFACE_SCOPE

* PURPOSE: Establish a new debugging scope for the current frame.
* FUNCTION: Pop the CONTROL_STACK to access the name of the target module. Pop the CONTROL_STACK again to get the new value of the breakpoint scope. Write the value to that module's DEBUGGING_INFO on the CONTROL_STACK.
* STACKS: Precondition:
  - Top of CONTROL_STACK contains a DISCRETE_VAR indicating the module name.
  - Top - 1 of CONTROL_STACK contains a DISCRETE_VAR indicating the new debugging scope.

  Postcondition:
  - Top of CONTROL_STACK reuduce by two.
* EXCEPTIONS: None.

3.1.4.12 SET_INTERFACE_SUBPROGRAM

* PURPOSE: Set the a reference to an interface subprogram for a target module.
* FUNCTION: Pop the CONTROL_STACK to access the name of the target module. Pop the CONTROL_STACK again to access the interface subprogram. Write this value to the target module CONTROL_STACK at

the offset for interface subprograms.

* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR indicating the target module name.
  Top - 1 of CONTROL_STACK contains a SUBPROGRAM_VAR indicating the interface subprogram.
  Postcondition:
  Top of CONTROL_STACK is reduced by two.
* EXCEPTIONS: INSTRUCTION_ERROR is raised if the interface subprogram is not code for call or for interface.

3.1.5 **NO_OPERATION** This operation provides a null execution facility.
NO_OPERATION includes the single UNCLASSED_ACTION:

* IDLE

3.1.5.1 IDLE

* PURPOSE: Provide a null execution facility.
* FUNCTION: Do nothing.
* STACKS: No change.
* EXCEPTIONS: None.

3.1.6 **REFERENCE_OPERATIONS** These operations provide facilities for activating remote subprograms.
REFERENCE_OPERATIONS include:

* ACTIVATE_SUBPROGRAM
* CALL_REFERENCE
* DELETE_ITEM
* DELETE_SUBPROGRAM
* SET_VISIBILITY

3.1.6.1 ACTIVATE_SUBPROGRAM

* PURPOSE: Set an indirectly accessed subprogram as active.
* FUNCTION: Pop the CONTROL_STACK to get a subprogram reference. Access the site of the subprogram, and set the SUBPROG_ACTIVE at that site.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a SUBPROGRAM_REF.
  Postcondition:
  Top of CONTROL_STACK reduced by one.
* EXCEPTIONS: INSTRUCTION_ERROR will be raised if reference subprogram is not code for call, or if the site of the subprogram is not found.

3.1.6.2 CALL_REFERENCE

* PURPOSE: Call an indirectly accessed subprogram.
* FUNCTION: Pop the CONTROL_STACK to get a subprogram reference. Access the site of the subprogram, and establish a new frame for the referenced subprogram.

* STACKS: Precondition:
  Top of CONTROL_STACK contains a SUBPROGRAM_REF.
  Postcondition:
  Top of CONTROL_STACK reduced by one, and then a new ACTIVATION_STATE and a new ACTIVATION_LINK are pushed to mark the new frame.
* EXCEPTIONS: INSTRUCTION_ERROR will be raised if reference subprogram is not code for call.

3.1.6.3 DELETE_ITEM

* PURPOSE: Delete an entity.
* FUNCTION: Pop the CONTROL_STACK to get a variable reference. Pop the CONTROL_STACK again to access the deletion key. Access the referenced variable, and mark the location as deleted.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a VARIABLE_REF.
  Top - 1 of CONTROL_STACK contains a DELETION_KEY.
  Postcondition:
  Top of CONTROL_STACK is reduced by two.
  Referenced variable is marked as deleted on CONTROL_STACK.
* EXCEPTIONS: CAPABILITY_ERROR is raised if DELETION_KEY is not found or if it is locked.
  OPERAND_CLASS_ERROR is raised if referenced entity is not found.

3.1.6.4 DELETE_SUBPROGRAM

* PURPOSE: Delete a subprogram.
* FUNCTION: Pop the CONTROL_STACK to ge a subprogram reference. Pop the CONTROL_STACK again to get a deletion key. Access the referenced subprogram and mark the location as deleted.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a SUBPROGRAM_REF.
  Top - 1 of CONTROL_STACK contains a DELETION_KEY.
  Postcondition:
  Top of CONTROL_STACK reduced by two.
  Referenced subprogram is marked as deleted on the CONTROL_STACK.
* EXCEPTIONS: CAPABILITY_ERROR is raised if DELETION_KEY is not found or if it is locked.
  OPERAND_CLASS_ERROR is raised if referenced subprogram is not found.

3.1.6.5 SET_VISIBILITY

* PURPOSE: Set the visibility of an entity.
* FUNCTION: Pop the CONTROL_STACK to access a variable reference. Access the deletion key at that site and mark the key as locked and sets visibility.

* STACKS: Precondition:
  Top of CONTROL_STACK contains a VARIABLE_REF.
  Postcondition:
  Top of CONTROL_STACK is reduced by one.
* EXCEPTIONS: INSTRUCTION_ERROR is raised if DELETION_KEY is not found at the referenced site.

3.1.7 **RESOURCE_OPERATIONS** These operations provide facilities for allocating and recovering resources.
RESOURCE_OPERATIONS include:

* QUERY_RESOURCE_LIMITS
* QUERY_RESOURCE_STATE
* RECOVER_RESOURCES
* RETURN_RESOURCES
* SET_RESOURCE_LIMITS

3.1.7.1 QUERY_RESOURCE_LIMITS

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.7.2 QUERY_RESOURCE_STATE

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.7.3 RECOVER_RESOURCES

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.7.4 RETURN_RESOURCES

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.7.5 SET_RESOURCE_LIMITS

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.8 **SLEEP_OPERATIONS** These operations provide facilities for putting a module to sleep or to abort a task.
SLEEP_OPERATIONS include:

* INITIATE_DELAY
* PORPOGATE_ABORT

3.1.8.1 INITIATE_DELAY

* PURPOSE: Put a module to sleep for a specified delay.
* FUNCTION: Pop the CONTROL_STACK to access the delay period. Put the module on a clock queue, to be awaken after the specified delay.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a VALUE_VAR indicating the delay period.
  Postcondition:
  Top of CONTROL_STACK is reduced by one. Other changes result due to context switch from being placed on the clock queue.
* EXCEPTIONS: None.

3.1.8.2 PROPOGATE_ABORT

* PURPOSE: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.1.9 **STACK_OPERATIONS** These operations provide primitive facilities for manipulating various stacks as defined by the architecture.
STACK_OPERATIONS include:

* MARK_AUXILIARY
* MARK_DATA
* MARK_TYPE
* POP_AUXILIARY
* POP_CONTROL
* POP_DATA
* POP_TYPE
* PUSH_CONTROL
* SWAP_CONTROL

3.1.9.1 MARK_AUXILIARY

* PURPOSE: Mark both the DATA_STACK and TYPE_STACK.
* FUNCTION: Read the top of both the DATA_STACK and the TYPE_STACK, and save these values in an AUXILIARY_MARK on the CONTROL_STACK. In the ACTIVATION_LINK of the current frame, set AUXILIARY_MARKED to TRUE.
* STACKS: Postcondition:
  AUXILIARY_MARK pushed on top of CONTROL_STACK.
  Current ACTIVATION_LINK is updated.
* EXCEPTIONS: None.

3.1.9.2 MARK_DATA

* PURPOSE: Mark the DATA_STACK.
* FUNCTION: Read the top of the DATA_STACK and save the

value in an AUXILIARY_MARK on the CONTROL_STACK. In the ACTIVATION_LINK of the current frame, set AUXILIARY_MARKED to TRUE.

* STACKS: Postcondition:
  AUXILIARY_MARK pushed on top of CONTROL_STACK.
  Current ACTIVATION_LINK is updated.
* EXCEPTIONS: None.

3.1.9.3 MARK_TYPE

* PURPOSE: Mark the TYPE_STACK.
* FUNCTION: Read the top of the TYPE_STACK and save the value in an AUXILIARY_MARK on the CONTROL_STACK. In the ACTIVATION_LINK of the current frame, set AUXILIARY_MARKED to TRUE.
* STACKS: Postcondition:
  AUXILIARY_MARK pushed on top of CONTROL_STACK.
  Current ACTIVATION_LINK is updated.
* EXCEPTIONS: None.

3.1.9.4 POP_AUXILIARY

* PURPOSE: Pop to the last mark of the DATA_STACK and the TYPE_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the current AUXILIARY_MARK. Pop both the DATA_STACK and TYPE_STACK down to the point of the last mark. In the ACTIVATION_LINK of the current frame, reset AUXILIARY_MARKED.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an AUXILIARY_MARK.
  Postcondition:
  Top of CONTROL_STACK reduced by one.
  Current ACTIVATION_LINK is updated.
  DATA_STACK and TYPE_STACK both popped to position before the last mark.
* EXCEPTIONS: INSTRUCTION_ERROR raised if CONTROL_STACK does not have a valid AUXILIARY_MARK.

3.1.9.5 POP_CONTROL

* PURPOSE: Pop the CONTROL_STACK.
* FUNCTION: Read the top of the CONTROL_STACK to determine the nature of the entity on type. Pop the CONTROL_STACK to remove this entity.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain a VALUE_VAR or a STRUCTURE_VAR.
  Postcondition:
  Top of CONTROL_STACK reduced by one.
* EXCEPTIONS: OPERAND_CLASS_ERROR raised if top of CONTROL_STACK does not have a valid VALUE_VAR or a STRUCTURE_VAR.

3.1.9.6 POP_DATA

* PURPOSE: Pop to the last mark of the DATA_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the current AUXILIARY_MARK. Pop both DATA_STACK down to the point of the last mark. In the ACTIVATION_LINK of the current frame, reset AUXILIARY_MARKED.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an AUXILIARY_MARK.
  Postcondition:
  Top of CONTROL_STACK reduced by one.
  Current ACTIVATION_LINK is updated.
  DATA_STACK popped to position before the last mark.
* EXCEPTIONS: INSTRUCTION_ERROR raised if CONTROL_STACK does not have a valid AUXILIARY_MARK.

3.1.9.7 POP_TYPE

* PURPOSE: Pop to the last mark of the TYPE_STACK.
* FUNCTION: Pop the CONTROL_STACK to access the current AUXILIARY_MARK. Pop the TYPE_STACK down to the point of the last mark. In the ACTIVATION_LINK of the current frame, reset AUXILIARY_MARKED.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an AUXILIARY_MARK.
  Postcondition:
  Top of CONTROL_STACK reduced by one.
  Current ACTIVATION_LINK is updated.
  TYPE_STACK popped to position before the last mark.
* EXCEPTIONS: INSTRUCTION_ERROR raised if CONTROL_STACK does not have a valid AUXILIARY_MARK.

3.1.9.8 PUSH_CONTROL

* PURPOSE: Duplicate the top entry on the CONTROL_STACK.
* FUNCTION: Read the top of the CONTROL_STACK. Push the same value on top of the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain a VALUE_VAR or a STRUCTURE_VAR..
  Postcondition:
  Top of CONTROL_STACK increased by one.
  Top two entities are identical.
* EXCEPTIONS: OPERAND_CLASS_ERROR raised if top of CONTROL_STACK does not have a valid VALUE_VAR or a STRUCTURE_VAR.

3.1.9.9 SWAP_CONTROL

* PURPOSE: Reverse the top two elements of the CONTROL_STACK.
* FUNCTION: Read the top element of the CONTROL_STACK, then read the second element. Write each value in the opposite

offset.

* STACKS: Precondition:
Top of CONTROL_STACK and top - 1 of CONTROL_STACK must both contain either a VALUE_VAR or a STRUCTURE_VAR.
Postcondition:
Top two elements of the CONTROL_STACK are reversed

* EXCEPTIONS: OPERAND_CLASS_ERROR is raised if either the top or the top - 1 of the CONTROL_STACK are not a VALUE_VAR nor a STRUCTURE_VAR.

3.2 CALL

The CALL instruction invokes a subprogram, block, accept, or select. Formally, CALL takes the form:

```
type CALL_INSTRUCTION is
  record
    OBJECT : OBJECT_REFERENCE;
  end record;
```

The OBJECT to call is of the type **OBJECT_REFERENCE,** which we further define as:

```
type OBJECT_REFERENCE (LEVEL: LEXICAL_LEVEL := Ø) is
  record
    case LEVEL is
      when Ø .. 1 => SCOPE_OFFSET : SCOPE_DELTA;
      when others => FRAME_OFFSET : FRAME_DELTA;
    end case;
  end record;
```

To complete our definition of the OBJECT_REFERENCE, we define **FRAME_DELTA, LEXICAL_LEVEL,** and **SCOPE_DELTA** as:

```
MAX_FRAME : constant INTEGER := implementation_defined;
MAX_LEVEL : contstnt INTEGER := implementation_defined;
MAX_SCOPE : constant INTEGER := implementation_defined;

type FRAME_DELTA   is new INTEGER range -(MAX_FRAME + 1) .. MAX_FRAME;
type LEXICAL_LEVEL is new INTEGER range               Ø .. MAX_LEVEL;
type SCOPE_DELTA   is new INTEGER range               Ø .. MAX_SCOPE;
```

Note that if the LEXICAL_LEVEL of the CALL.OBJECT_REFERENCE has a value of Ø or 1, this indicates that the called entity is in a local scope; otherwise, the called entity will be found in an enclosing frame.

* PURPOSE: Invoke a subprogram, block, accept, or select.

* FUNCTION: Trace the OBJECT_REFERENCE to find the corresponding SUBPROGRAM_VAR. Mark the CONTROL_STACK to indicate the creation of a new frame. Control is transfered to the

first instruction of the SUBPROGRAM_VAR.

* STACKS: Postcondition:
A new ACTIVATION_STATE and ACTIVATION_LINK are pushed on the CONTROL_STACK.

* EXCEPTIONS: INSTRUCTION_ERROR is raised if the referenced SUBPROGRAM_VAR is not found.

3.3 EXECUTE

The EXECUTE instruction performs an operation upon a typed object. Formally, EXECUTE takes the form:

```
type EXECUTE_INSTRUCTION is
  record
    ON_CLASS      : OPERAND_CLASS;
    OPERATION     : OPERATOR;
    FIELD         : FIELD_INDEX;
    FIELD_ACCESS  : FIELD_ACCESS_MODE;
    FIELD_KIND    : FIELD_SORT;
  end record;
```

The operand ON_CLASS identifies the **OPERAND_CLASS** of the object that a particular EXECUTE will operate upon (section 2.1). Generally, the target operand will be on the top of the CONTROL_STACK. As the following sections will illustrate, not all ON_CLASS values are legal for a given OPERATION;in addition, the specific function performed depends upon both. If an attempt is made to EXECUTE an OPERATION that is not appropriate for the ON_CLASS entity, the exception INSTRUCTION_ERROR will be raised. If the OPERATION is appropriate, but the ON_CLASS entity is not found during execution (such as in the correct position on the CONTROL_STACK), then the exception OPERAND_CLASS_ERROR is raised. Finally, if the ON_CLASS entity is found but the object is private or otherwise out of scope, then the exception CAPBAILITY_ERROR is raised.

OPERATION is of the type **OPERATOR,** which we further define as:

```
type OPERATOR is
  (-- ACCESS_OPERATIONS
       ALL_READ_OP,                  ALL_REFERENCE_OP,
       ALL_WRITE_OP,                 ALLOCATE_OP,
       ALLOCATE_WITH_CONSTRAINT_OP,
       ALLOCATE_WITH_INITIAL_VALUE_OP,
       ALLOCATE_WITH_SUBTYPE_OP,
      -IS_NULL_OP,                  ~NOT_NULL_OP,
       NULL_OP,
   -- ALIGNMENT_OPERATION,
       MAKE_ALIGNED_OP,
   -- ARITHMETIC_OPERATIONS
       DIVIDE_OP,                    MINUS_OP,
       MODULO_OP,                    PLUS_OP,
       REMAINDER_OP,                 TIMES_OP,
   -- ARRAY_OPERATIONS
       APPEND_OP,                    CONCATENATE_OP,
```

```
          PREPEND_OP,                   SLICE_READ_OP,
          SLICE_WRITE_OP,               SUBARRAY_OP,
     -- ATTRIBUTE_OPERATIONS
          ADDRESS_OP,                   COUNT_OP,
          FIRST_OP,                     IS_COMPLETED_OP,
          IS_CONSTRAINED_OP,            IS_TERMINATED_OP,
          LAST_OP,                      LENGTH_OP,
          PREDECESSOR_OP,               POSITION_OP,
          SIZE_OP,                      SUCCESSOR_OP,
          VALUE_OP,
     -- BOUNDS_OPERATIONS
          BOUND_CHECK_OP,               BOUNDS_OP,
          REVERSE_BOUNDS_OP,            SET_BOUNDS_OP,
     -- COMPLETION_OPERATIONS
          COMPLETE_CONSTRAINED_OP,  COMPLETE_DEFINED_OP,
          COMPLETE_DERIVED_OP,          COMPLETE_TYPE_OP,
     -- CONVERSION_OPERATIONS
          CONVERT_ACTUAL_OP,            CONVERT_OP,
     -- ELEMENT_OPERATION
          ELEMENT_TYPE_OP,
     -- EQUALITY_OPERATIONS
          EQUAL_OP,                     NOT_EQUAL_OP,
     -- EXCEPTION_OPERATIONS
          RAISE_OP,                     RAISED_ADDRESS_OP,
          RAISED_NAME_OP,               RAISED_SCOPE_OP,
          RAISED_VARIETY_OP,
     -- FIELD_OPERATIONS
          FIELD_EXECUTE_OP,             FIELD_READ_OP,
          FIELD_REFERENCE_OP,           FIELD_TYPE_OP,
          FIELD_WRITE_OP,
     -- IMPORT_OPERATION,
          AUGMENT_IMPORTS_OP,
     -- LOGICAL_OPERATIONS
          AND_OP,                       OR_OP,
          XOR_OP,
     -- MEMBERSHIP_OPERATIONS
          CHECK_IN_TYPE_OP,             IN_TYPE_OP,
          NOT_IN_TYPE_OP,
     -- RANGE_OPERATIONS
          ABOVE_RANGE_OP,               BELOW_RANGE_OP,
          IN_RANGE_OP,                  NOT_IN_RANGE_OP,
     -- RELATIONAL_OPERATIONS
          GREATER_EQUAL_OP,             GREATER_OP,
          LESS_EQUAL_OP,                LESS_OP,
     -- SEGMENT_OPERATIONS
          SEGMENT_NAME_OP,              SEGMENT_NUMBER_OP,
          SEGMENT_STORE_OP,
     -- TASKING_OPERATIONS
          ABORT_OP,                     ACTIVATE_OP,
          COND_CALL_OP,                 CONTINUE_OP,
          ENTRY_CALL_OP,                FAMILY_CALL_OP,
          FAMILY_COND_OP,               FAMILY_TIMED_OP,
          GUARD_WRITE_OP,               INTERRUPT_OP,
          RENDEZVOUS_OP,                TIMED_CALL_OP,
     -- UNARY_OPERATIONS
          ABSOLUTE_VALUE_OP,            DECREMENT_OP,
```

```
                INCREMENT_OP,              NOT_OP
                UNARY_MINUS_OP,
            -- VARIABLE_OPERATIONS
                MAKE_CONSTANT_OP,          MAKE_VISIBLE_OP,
                RUN_UTILITY_OP,
            -- VARIANT_OPERATIONS
                MAKE_CONSTRAINED_OP,       SET_CONSTRAINED_OP,
                SET_VARIANT_OP);
```

The elements **FIELD, FIELD_ACCESS,** and **FIELD_KIND** serve to further qualify the ON_CLASS/OPERATION combination, by referencing a particular component of a composite structure. These three elements are applicable only for OPERATIONs that apply the ANY_CLASS, PACKAGE_CLASS, RECORD_CLASS, SELECT_CLASS, TASK_CLASS, and VARIANT_RECORD_CLASS objects. NO_VARIANTS is use as the value of FIELD whenever the ON_CLASS/OPERATION combination requires no further qualification, or if further qualification is meaningless. We can complete the form of the EXECUTE_INSTRUCTION with the following:

```
FIELD_SIZE             : constant INTEGER := implementation_defined;

type FIELD_ACCESS_MODE is (DIRECT, INDIRECT);
type FIELD_INDEX       is new INTEGER range Ø .. (2 ** FIELD_SIZE) - 1;
type FIELD_SORT        is (FIXED, VARIANT);

NO_VARIANTS            : constant FIELD_INDEX := FIELD_INDEX'LAST;
```

In the following sections we provide a detailed description of each OPERATION. Since we have already mentioned the conditions under which CAPABILITY_ERROR, INSTRUCTION_ERROR, and OPERAND_CLASS_ERROR will be raised, we will omit references to these exceptions in the following discussion.

3.3.1 **ACCESS_OPERATIONS** These operations provide facilities for constructing and testing designated access objects.

ACCESS_OPERATIONS include:

* ALL_READ_OP
* ALL_REFERENCE_OP
* ALL_WRITE_OP
* ALLOCATE_OP
* ALLOCATE_WITH_CONSTRAINT_OP
* ALLOCATE_WITH_INITIAL_VALUE_OP
* ALLOCATE_WITH_SUBTYPE_OP
* IS_NULL_OP
* NOT_NULL_OP
* NULL_OP

3.3.1.1 ALL_READ_OP

* PURPOSE: Get value of a designated access object.
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop an ACCESS_VAR off the CONTROL_STACK and trace its reference to the value of the designated object. Push the value on the CONTROL_STACK. If the value is not composite,

the value pushed will be a VALUE_VAR; for structures, the value pushed will be an INDIRECT_VAR.

* STACKS: Precondition:
  Top of CONTROL_STACK contains an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced by one, and then a VALUE_VAR or an INDIRECT_VAR is pushed on the stack.
* EXCEPTIONS: CONSTRAINT_ERROR is raised of the ACCESS_VAR is null.
  TYPE_ERROR is raised if the referenced object is not the type expected by the ACCESS_VAR.

3.3.1.2 ALL_REFERENCE_OP

* PURPOSE: Build a reference to the value of a designated access object.
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop an ACCESS_VAR off the CONTROL_STACK and trace its reference to the value of the designated object. Create a reference to the value, and push the reference on the CONTROL_STACK. If the value is not composite, the reference value pushed will be a VARIABLE_REF; for structures, the value pushed will be an INDIRECT_VAR.
* STACKS: Precondition:
  Top of CONTROL_STACK contains an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced by one, and then a VARIABLE_REF or an INDIRECT_VAR is pushed on the stack.
* EXCEPTIONS: CONSTRAINT_ERROR is raised of the ACCESS_VAR is null.
  TYPE_ERROR is raised if the referenced object is not the type expected by the ACCESS_VAR.

3.3.1.3 ALL_WRITE_OP

* PURPOSE: Put the value of a designated access object.
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop an ACCESS_VAR off the CONTROL_STACK and trace its reference to the value of the designated object. Pop the CONTROL_STACK again to access the new value. Copy this value to the designated access object.
* STACKS: Precondition:
  Top of CONTROL_STACK contains an ACCESS_VAR.
  Top - 1 of CONTROL_STACK contains a VALUE_VAR the indicates the new designated access object value.
  Postcondition:
  Top of CONTROL_STACK is reduced by two.
* EXCEPTIONS: CONSTRAINT_ERROR is raised of the ACCESS_VAR is null.

TYPE_ERROR is raised if the referenced object or the VALUE_VAR is not the type expected by the ACCESS_VAR.

3.3.1.4 ALLOCATE_OP

- PURPOSE: Create a designated access object.
- ON_CLASS: ACCESS_CLASS only
- FUNCTION: Pop the CONTROL_STACK to get an ACCESS_VAR object. Trace the type path to determine the type of the designated object. Allocate space in the collection on the DATA_STACK associated with the ACCESS_VAR, and update the value of the ACCESS_VAR to point to the newly allocated object. Push the ACCESS_VAR back on the CONTROL_STACK.
- STACKS: Precondition:
  - Top of CONTROL_STACK must contain an ACCESS_VAR.

  Postcondition:
  - Top of CONTROL_STACK is reduced by one, and the an ACCESS_VAR that points to the newly allocated object is pushed back on the CONTROL_STACK.
- EXCEPTIONS: SOME_ERROR is raised when there is no space remaining in a given collection.

  TYPE_ERROR is raised when the ACCESS_VAR points to an empty type, or if the ACCESS_VAR type information cannot be located.

3.3.1.5 ALLOCATE_WITH_CONSTRAINT_OP

- PURPOSE: Create a constrained designated access object.
- ON_CLASS: ACCESS_CLASS only
- FUNCTION: Pop the CONTROL_STACK to get an ACCESS_VAR object. Trace the type path to determine the type of the designated object. Pop the CONTROL_STACK to get the constraints upon the designated object (see STACKS below). Allocate space in the collection on the DATA_STACK associated with the ACCESS_VAR, and update the value of the ACCESS_VAR to point to the newly allocated object. Push the ACCESS_VAR back on the CONTROL_STACK.
- STACKS: Precondition:
  - Top of CONTROL_STACK must contain an ACCESS_VAR.
  - If type of designated access object is an array, the array bounds constraint pairs are next on the stack, in order of the indices; maximum bound constraints are below the minimum bound constraints.
  - If type of designated access object is a record with discriminants, the variant index information is next on the stack, followed by each discriminant constraint, in order.

  Postcondition:
  - Top of CONTROL_STACK is reduced to below the original ACCESS_VAR, and the an ACCESS_VAR that points to the newly allocated object is pushed back on the CONTROL_STACK.
  - The DATA_STACK is used for intermediate calculations, but is returned to its initial state.

g * EXCEPTIONS: SOME_ERROR is raised when there is no space remainin in a given collection.
TYPE_ERROR is raised when the ACCESS_VAR points to an empty type, if the ACCESS_VAR type information cannot be found, or if the designated access object cannot be further constrained.
CONSTRAINT_ERROR is raised if the constraint values are not compatible with the designated access object.

3.3.1.6 ALLOCATE_WITH_INITIAL_VALUE_OP

* PURPOSE: Create a designated access object with an initial value.
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop the CONTROL_STACK to get an ACCESS_VAR object. Trace the type path to determine the type of the designated object. Pop the CONTROL_STACK to get the initial value of the designated object (see STACKS below). Use this value to determine any constraints upon the designated access object. Allocate space in the collection on the DATA_STACK associated with the ACCESS_VAR, set the designated object to the initial value, and update the value of the ACCESS_VAR to point to the newly allocated object. Push the ACCESS_VAR back on the CONTROL_STACK.
* STACKS: Precondition:
Top of CONTROL_STACK must contain an ACCESS_VAR.
Postcondition:
Top of CONTROL_STACK is reduced to below the original ACCESS_VAR, and the ACCESS_VAR that points to the newly allocated object is pushed back on the CONTROL_STACK.
The DATA_STACK is used for intermediate calculations, but is returned to its initial state

g * EXCEPTIONS: SOME_ERROR is raised when there is no space remainin in a given collection.
TYPE_ERROR is raised when the ACCESS_VAR points to an empty type, if the ACCESS_VAR type information cannot be found, or if the initial value is not of the correct type.
CONSTRAINT_ERROR is raised if the initial value is not compatible with the type of the designated access object.

3.3.1.7 ALLOCATE_WITH_SUBTYPE_OP

* PURPOSE: Create a designated access object constrained by a subtype.
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop the CONTROL_STACK to get an ACCESS_VAR object. Trace the type path to determine the type of the designated object. Pop the CONTROL_STACK to get the subtype constraint for the designated

object (see STACKS below). Allocate space in the collection on the DATA_STACK associated with the ACCESS_VAR, and update the value of the ACCESS_VAR to point to the newly allocated object. Push the ACCESS_VAR back on the CONTROL_STACK.

* STACKS: Precondition:
  Top of CONTROL_STACK must contain an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced to below the original ACCESS_VAR, and the ACCESS_VAR that points to the newly allocated object is pushed back on the CONTROL_STACK.
  The DATA_STACK is used for intermediate calculations, but is returned to its initial state
* EXCEPTIONS: SOME_ERROR is raised when there is no space remaining in a given collection.
  TYPE_ERROR is raised when the ACCESS_VAR points to an empty type, if the ACCESS_VAR type information cannot be found, or if the initial value is not of the correct type.
  CONSTRAINT_ERROR is raised if the subtype is not compatible with the designated access object.

3.3.1.8 <u>IS_NULL_OP</u>

* PURPOSE: Determine if an access variable is null
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop the CONTROL_STACK to get the ACCESS_VAR. If it has a null value, push a TRUE value on the CONTROL_STACK, otherwise, push a FALSE value.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced by one, and then a boolean DISCRETE_VAR is pushed on the stack.
* EXCEPTIONS: None.

3.3.1.9 <u>NOT_NULL_OP</u>

* PURPOSE: Determine if an access variable is not null
* ON_CLASS: ACCESS_CLASS only
* FUNCTION: Pop the CONTROL_STACK to get the ACCESS_VAR. If it has a null value, push a FALSE value on the CONTROL_STACK, otherwise, push a TRUE value.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced by one, and then a boolean DISCRETE_VAR is pushed on the stack.
* EXCEPTIONS: None.

3.3.1.10 <u>NULL_OP</u>

* PURPOSE: Give a null value to an access variable.

* ON_CLASS: ACCESS_CLASS only.
* FUNCTION: Pop the CONTROL_STACK to get the ACCESS_VAR. Set it to a null value, and push the ACCESS_VAR back on the CONTROL_STACK. Note that this action does not directly deallocate the designated access object.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced by one, and then a null ACCESS_VAR is pushed on the stack.
* EXCEPTIONS: None.

3.3.2 **ALIGNMENT_OPERATION** This operation provides a facility for forcing a value into a given alignment.

ALIGNMENT_OPERATION includes the single OPERATOR:

* MAKE_ALIGNED_OP

3.3.2.1 MAKE_ALIGNED_OP

* PURPOSE: Currently unimplemented instruction.
* ON_CLASS: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

* ON_CLASS: ACCESS_CLASS only.
* FUNCTION: Pop the CONTROL_STACK to get the ACCESS_VAR. Set it to a null value, and push the ACCESS_VAR back on the CONTROL_STACK. Note that this action does not directly deallocate the designated access object.
* STACKS: Precondition:
  Top of CONTROL_STACK must contain an ACCESS_VAR.
  Postcondition:
  Top of CONTROL_STACK is reduced by one, and then a null ACCESS_VAR is pushed on the stack.
* EXCEPTIONS: None.

3.3.2 **ALIGNMENT_OPERATION** This operation provides a facility for forcing a value into a given alignment.

ALIGNMENT_OPERATION includes the single OPERATOR:

* MAKE_ALIGNED_OP

3.3.2.1 MAKE_ALIGNED_OP

* PURPOSE: Currently unimplemented instruction.
* ON_CLASS: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.3.4 **ARITHMETIC_OPERATIONS** These operations provide facilities for the usual arithmetic functions.

ARITHMETIC_OPERATIONS include:

* DIVIDE_OP
* MINUS_OP
* MODULO_OP
* PLUS_OP
* REMAINDER_OP
* TIMES_OP

3.3.4.1 DIVIDE_OP

* PURPOSE: Divide two values yielding a third
* ON_CLASS: DISCRETE_CLASS and FLOAT_CLASS
* FUNCTION: Pop value_1 off the CONTROL_STACK. Pop value_2 off the CONTROL_STACK. Divide value_2 by value_1. Push the result of the operation back on the CONTROL_STACK. The result type is the same as the types of the two operands. [TOS-1] / [TOS]
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR or a FLOAT_VAR.
  Top - 1 of CONTROL_STACK contains a value of the same type.
  Postcondition:
  Top of CONTROL_STACK is reduced by two, and then a value of type DISCRETE_VAR or FLOAT_VAR is pushed on the stack.

* EXCEPTIONS: NUMERIC_ERROR is raised if the divide operation results in a value that cannot be represented on the given implementation. AND DIVIDE BY ZERO

3.3.4.2 MINUS_OP

* PURPOSE: Subtract two values yielding a third
* ON_CLASS: DISCRETE_CLASS and FLOAT_CLASS
* FUNCTION: Pop value_1 off the CONTROL_STACK. Pop value_2 off the CONTROL_STACK. Take value_~~1~~ 2 and subtract value_~~2~~ 1. Push the result of the operation back on the CONTROL_STACK. The result type is the same as the types of the two operands. [TOS-1] - [TOS]
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR or ~~a FLOAT_VAR~~ OK.
  Top - 1 of CONTROL_STACK contains a value of the same type.
  Postcondition:
  Top of CONTROL_STACK is reduced by two, and then a value of type DISCRETE_VAR or FLOAT_VAR is pushed on the stack.
* EXCEPTIONS: NUMERIC_ERROR is raised if the minus operation results in a value that cannot be represented on the given implementation.

3.3.4.3 MODULO_OP

* PURPOSE: Find the modulus of two values yielding a third
* ON_CLASS: DISCRETE_CLASS ~~and FLOAT_CLASS.~~
* FUNCTION: Pop value_1 off the CONTROL_STACK. Pop value_2 off the CONTROL_STACK. Take value_~~1~~ 2 modulus value_~~2~~ 1. Push the result of the operation back on the CONTROL_STACK. The result type is the same as the types of the two operands. [TOS-1] MOD [TOS]
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR or ~~a FLOAT_VAR.~~
  Top - 1 of CONTROL_STACK contains a value of the same type.
  Postcondition:
  Top of CONTROL_STACK is reduced by two, and then a value of type DISCRETE_VAR or ~~FLOAT_VAR is pushed on the stack~~.
* EXCEPTIONS: NUMERIC_ERROR is raised if the modulus operation results in a value that cannot be represented on the given implementation. OR DIVIDE BY ZERO

3.3.4.4 PLUS_OP

* PURPOSE: Add two values yielding a third
* ON_CLASS: DISCRETE_CLASS and FLOAT_CLASS
* FUNCTION: Pop value_1 off the CONTROL_STACK. Pop value_2 off the CONTROL_STACK. Take value_1

and add value_2. Push the result of the operation back on the CONTROL_STACK. The result type is the same as the types of the two operands.

* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR or a FLOAT_VAR.
  Top - 1 of CONTROL_STACK contains a value of the same type.
  Postcondition:
  Top of CONTROL_STACK is reduced by two, and then a value of type DISCRETE_VAR or FLOAT_VAR is pushed on the stack.
* EXCEPTIONS: NUMERIC_ERROR is raised if the plus operation results in a value that cannot be represented on the given implementation.

3.3.4.5 REMAINDER_OP

* PURPOSE: Divide two values yielding a remainder value
* ON_CLASS: DISCRETE_CLASS ~~and FLOAT_CLASS~~
* FUNCTION: Pop value_1 off the CONTROL_STACK. Pop value_2 off the CONTROL_STACK. Take the remainder of value_2 divided by value_1. [TOS-1] REM [TOS]
  Push the result of the operation back on the CONTROL_STACK. The result type is the same as the types of the two operands.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR or ~~a FLOAT_VA~~R.
  Top - 1 of CONTROL_STACK contains a value of the same type.
  Postcondition:
  Top of CONTROL_STACK is reduced by two, and then a value of type DISCRETE_VAR or ~~FLOAT_VAR is pushed~~ on the stack.
* EXCEPTIONS: NUMERIC_ERROR is raised if the remainder operation results in a value that cannot be represented on the given implementation. DIVIDE BY ZERO

3.3.4.6 TIMES_OP

* PURPOSE: Multiply two values yielding a third
* ON_CLASS: DISCRETE_CLASS and FLOAT_CLASS
* FUNCTION: Pop value_1 off the CONTROL_STACK. Pop value_2 off the CONTROL_STACK. Take value_1 and multiply by value_2. Push the result of the operation back on the CONTROL_STACK. The result type is the same as the types of the two operands.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a DISCRETE_VAR or a FLOAT_VAR.
  Top - 1 of CONTROL_STACK contains a value of the same type.

Postcondition:
Top of CONTROL_STACK is reduced by two, and then a value of type DISCRETE_VAR or FLOAT_VAR is pushed on the stack.

* EXCEPTIONS: NUMERIC_ERROR is raised if the times operation results in a value that cannot be represented on the given implementation.

3.3.4 **ARRAY_OPERATIONS** These operations provide facilities for handling basic array manipulation.

ARRAY_OPERATIONS include:

* APPEND_OP
* CONCATENATE_OP
* PREPEND_OP
* SLICE_READ_OP
* SLICE_WRITE_OP
* SUBARRAY_OP

3.3.4.1 APPEND_OP

* PURPOSE: Append one array to another.
* ON_CLASS: VECTOR_CLASS, SUBVECTOR_CLASS
* FUNCTION: Pop the CONTROL_STACK to get the first array value, and construct an image of the value. Pop the CONTROL_STACK again, and copy the value beginning from the the start of the first image. Push the result back on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a VECTOR_VAR.
  Top - 1 of CONTROL_STACK contains a VECTOR_VAR.
  Postcondition:
  Top of CONTROL_STACK reduced by two, and then a new VECTOR_VAR is pushed.
* EXCEPTIONS: None.

3.3.4.2 CONCATENTATE_OP

* PURPOSE: Concatenate one array to another.
* ON_CLASS: VECTOR_CLASS, SUBVECTOR_CLASS
* FUNCTION: Pop the CONTROL_STACK to get the first array value, and construct an image of the value. Pop the CONTROL_STACK again, and copy the value beginning from the the end of the first image. Push the result back on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a VECTOR_VAR.
  Top - 1 of CONTROL_STACK contains a VECTOR_VAR.
  Postcondition:
  Top of CONTROL_STACK reduced by two, and then a new VECTOR_VAR is pushed.
* EXCEPTIONS: None.

3.3.4.3 <u>PREPEND_OP</u>

* PURPOSE: Currently unimplemented instruction.
* ON_CLASS: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.

3.3.4.4 <u>SLICE_READ_OP</u>

* PURPOSE: Construct an array slice value.
* ON_CLASS: VECTOR_CLASS, SUBVECTOR_CLASS.
* FUNCTION: Pop the CONTROL_STACK to access the target VECTOR_VAR. Pop the maximum ARRAY_INDEX_INFO, then pop the minimum ARRAY_INDEX_INFO. Using these constraints, extract the slice from the first array, and push the result on the CONTROL_STACK.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a VECTOR_VAR.
  Top - 1 of CONTROL_STACK contains a DISCRETE_VAR indicating the maximum index bounds.
  Top - 2 of CONTROL_STACK contains a DISCRETE_VAR indicating the minimum index bounds.
  Postcondition:
  Top of CONTROL_STACK reduced by three, and then a VECTOR_VAR is pushed on the stack.
* EXCEPTIONS: None.

3.3.4.5 <u>SLICE_WRITE_OP</u>

* PURPOSE: Write an array slice.
* ON_CLASS: VECTOR_CLASS, SUBVECTOR_CLASS.
* FUNCTION: Pop the CONTROL_STACK to access the source VECTOR_VAR. Pop the maximum ARRAY_INDEX_INFO, then pop the minimum ARRAY_INDEX_INFO. Using these constraints, extract the slice from the first array. Pop the CONTROL_STACK again to access the target VECTOR_VAR. Copy the slice into the target.
* STACKS: Precondition:
  Top of CONTROL_STACK contains a VECTOR_VAR.
  Top - 1 of CONTROL_STACK contains a DISCRETE_VAR indicating the maximum index bounds.
  Top - 2 of CONTROL_STACK contains a DISCRETE_VAR indicating the minimum index bounds.
  Top - 3 of CONTROL_STACK contains a VECTOR_VAR.
  Postcondition:
  Top of CONTROL_STACK reduced by three.
* EXCEPTIONS: None.

3.3.4.6 <u>SUBARRAY_OP</u>

* PURPOSE: Currently unimplemented instruction.
* ON_CLASS: Currently unimplemented instruction.
* FUNCTION: Currently unimplemented instruction.
* STACKS: Currently unimplemented instruction.
* EXCEPTIONS: Currently unimplemented instruction.