Technical Report

CMU/SEI-88-TR-019
ESD-TR-88-20

Generalized Image Library: A Durra Application Example

Mario R. Barbacci
Dennis L. Doubleday

July 1988

Technical Report

CMU/SEI-88-TR-019
ESD-TR-88-020
July 1988

Generalized Image Library: A Durra Application Example

Mario R. Barbacci
Dennis L. Doubleday

Software for Heterogeneous Machines Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl Shingler SIGNATURE ON FILE
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Generalized Image Library: A Durra Application Example

Abstract: Durra is a language designed to support the construction of distributed applications using concurrent, coarse-grain tasks running on networks of heterogeneous processors. An application written in Durra describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

This report describes an experiment in writing task descriptions and type declarations for a subset of the Generalized Image Library, a collection of utilities developed at the Department of Computer Science at Carnegie Mellon University. The experiment illustrates the development of a ''typical'' Durra application. This is a three step process: first, a collection of tasks (programs) is designed and implemented (these are the GIL programs); second, a collection of task descriptions corresponding to the task implementations is written in Durra, compiled, and stored in a library; and finally, an application description is written in Durra and compiled, resulting in a set of resource allocation and scheduling commands to be interpreted at runtime. A few sample application descriptions were developed as part of the experiment and are also reported in this document.

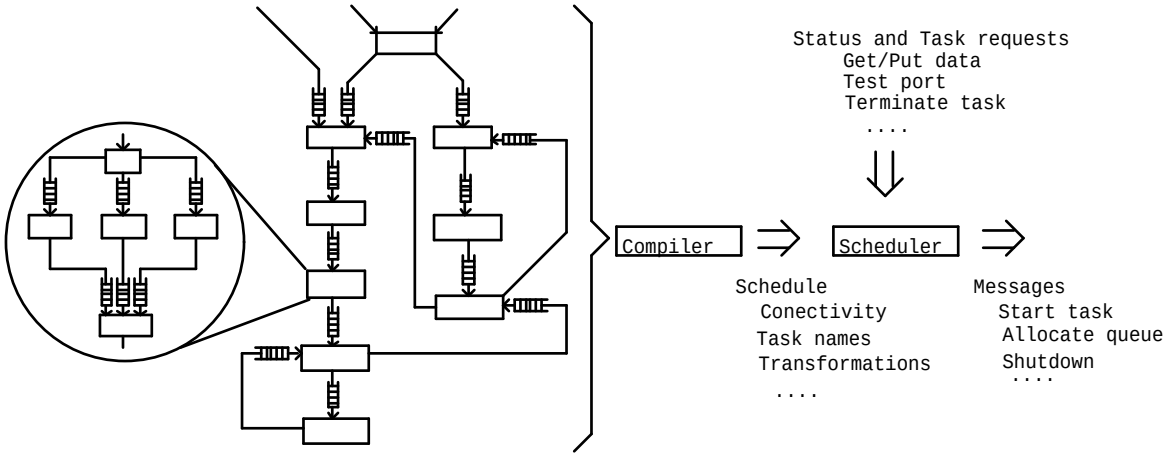
1. Introduction to Durra

Durra [1, 2] is a language designed to support the construction of distributed applications using concurrent, coarse-grain tasks running on networks of heterogeneous processors. An application written in Durra selects and reuses *task descriptions* and *type declarations* stored in a library. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

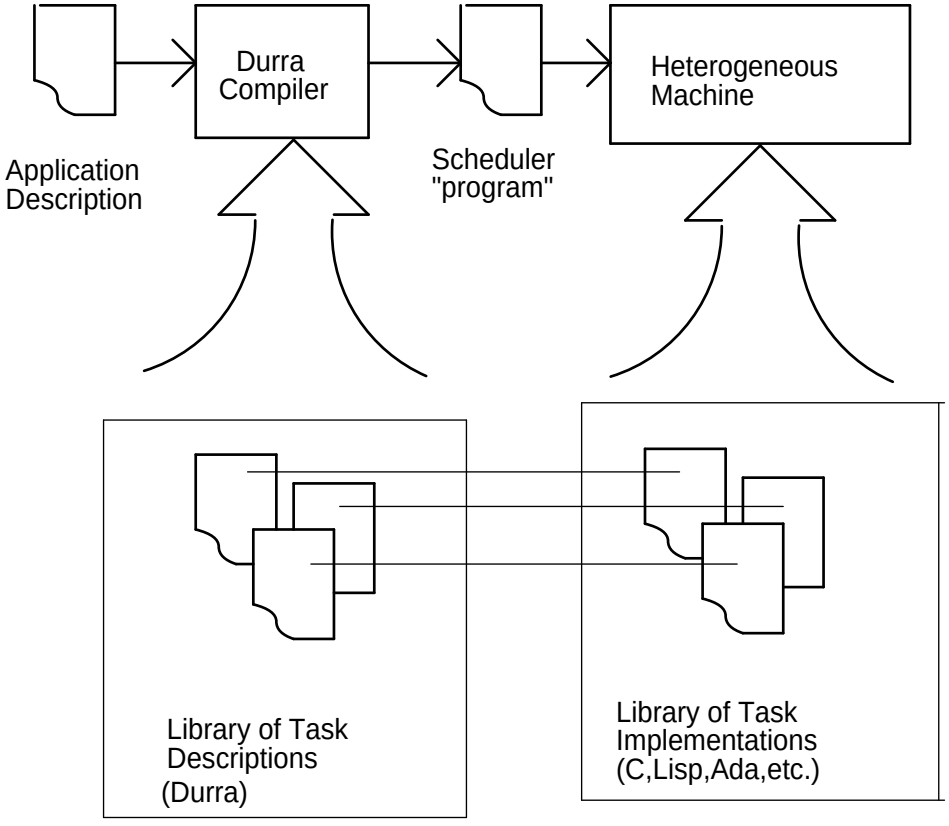
Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term ''description language'' rather than ''programming language'' to emphasize that a Durra application is not translated into object code in some kind of executable (conventional) ''machine language.'' Instead, a Durra application is a description of the structure and behavior of a logical machine to be synthesized into resource allocation and scheduling directives that are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a heterogeneous machine. This is the translation process depicted in Figure 1-1.a.

We see three distinct phases in the process of developing an application using Durra: the creation of a library of tasks, the creation of an application using library tasks, and the execution of the application. These three phases are illustrated in Figure 1-1.b.

During the first phase, the developer writes descriptions of the components tasks and declarations of the data types. The type declarations specify the kinds of data that will be produced and consumed by the tasks in the application. The task descriptions specify the properties of the task implementations (programs). For a given task, there may be many



a -- Compilation of an Application Description



b -- Developing an Application

Figure 1-1: Scenario for Developing an Application in Durra

implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes. For each implementation of a task, a description must be written in Durra, compiled, and entered in the library.

During the second phase, the user writes an *application description*. Syntactically, an application description is a single task description and could be stored in the library as a new task. This allows writing of hierarchical application descriptions. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands or instructions to be interpreted by the scheduler.

During the last phase, the scheduler loads the task implementations (i.e., programs corresponding to the component tasks) into the processors and issues the appropriate commands to execute the programs.

The data types transmitted between the tasks are declared independently of the tasks. In Durra, these data type declarations specify scalars (of possible variable length), arrays, simple record types, or unions of other types, as shown in Figure 1-2.

```
type packet is size 128 to 1024;
                                        -- Packets are of variable length.
type tails is array (5 10) of packet;
                                    -- Tails are 5 by 10 arrays of packets.
type rec is record (rows: integer, columns: integer, data: packet);
                        -- Rec data consists of two integers and a packet.
type mix is union (heads, tails);
                                      -- Mix data could be heads or tails.
```

Figure 1-2: Durra Type Declarations

Task descriptions are the building blocks for applications. Task descriptions include the following information (see Figure 1-3): (1) its interface to other tasks (**ports**) and to the scheduler (**signals**); (2) its **attributes**; (3) its functional and timing **behavior**; and (4) its internal **structure**, thereby allowing for hierarchical task descriptions. For the purposes of this report, the relevant components of a task description are the behavioral specifications.

```
task task_name
  ports                    -- Communication between a process and a queue
    port_declarations
  signals                -- Communication between a process and the scheduler
    signal_declarations
  attributes                        -- Miscellaneous properties of the task
    attribute_value_pairs
  behavior                  -- Functional and timing behavior of the task
    requires predicate                          -- Precondition on input data
    ensures predicate                         -- Postcondition on output data
    timing timing expression                   -- Ordering of port operations
  structure          -- Used to describe the internal structure of the task
    process_declarations                -- Declaration of internal processes
    bind_declarations       -- Mapping of internal ports to this task's ports
    queue_declarations                        -- Links between internal ports
    reconfiguration_statements       -- Dynamic modifications to the structure
end task_name
```

Figure 1-3: A Template for Task Descriptions

2. Introduction to the Generalized Image Library

The Generalized Image Library [4, 5] is a collection of programs that can be used to manipulate a variety of image representations produced by different image devices and stored in different disk formats. The Generalized Image Library (GIL) programs can be invoked directly by a user (via UNIX Shell commands) or by programs (via C procedure calls). For the purposes of this report, we describe GIL from a user's point of view, rather than from a programmer's point of view. The former is closer to the Durra development paradigm in which individual tasks are treated as black boxes, executing concurrently, and communicating via typed messages.

The user manipulates images by invoking the appropriate programs (i.e., image operations) together with a list of image names (i.e., input and output images). An image name encodes information about the format, in addition to a file name, or a physical device name. The most general format for images in the GIL is called GIF (Generalized Image Format). A GIF image is stored as a matrix of pixels, where a pixel can be an integer (signed or unsigned) or a floating-point number of arbitrary precision.

For example, a user could invoke the GIL **add** command:

```
add elms.gif oaks.gif forest.gif
```

to add the corresponding pixels of the ‘‘elms’’ and ‘‘oaks’’ images and produce the ‘‘forest’’ image.

Constant images, denoted by the keyword **constant** are virtual input images of unspecified dimensions, filled with a constant value for all pixels (i.e., all fetch operations return the same constant value):

```
add elms.gif constant:100 bright_elms.gif
```

Each pixel of ‘‘elms’’ will be increased by 100, storing the results in the new image ‘‘bright_elms.’’

The keyword **display** denotes the display device appropriate for the machine in which the GIL is used (this is obviously site dependent). For instance, the following command copies an image to the display device where it can be viewed on a monitor:

```
imgcp tree.gif display
```

The simplest kind of image name is just a file name of the form *name.ext* where *ext* is an identifier that indicates the image format (e.g., ‘‘gif’’ for images in Generalized Image Format). This default can be overruled by specifying the format as a keyword preceding the file name: *format_keyword*:*file_name*, as in

```
gif:some_file_name
```

GIL programs take advantage of the UNIX operating system, and in particular, of the UNIX pipeline mechanism, to enhance program composition by combining sequences of simple programs in a single shell command line. Users of GIL can pipeline images between GIL

programs via the standard UNIX streams (input, output, and error). For example ([4], page 3), the following shell command uses the **smooth** program to low-pass filter an image. It then sends the resulting low-pass image into **subimg**, where it is subtracted from the original image to produce a high-pass image:

```
smooth gauss -5 original.img - | subimg original - highpass.img
```

The keywords **unsigned:**, **signed:**, and **float:** used as a prefix to an image name may be used to select the pixel characteristics of a new image being created by a program. If the pixel characteristics are not specified in an image name, each program assumes some default pixel type. The keyword is followed by an integer number that indicates the number of bits allocated for each pixel:

```
unsigned:12:some_image.img
```

A number of simple image operations can be specified directly as prefixes to the image names used as parameters to the GIL programs. There are a large number of these ''prefix'' operations, but the following list should convey the general idea:

shift:*rows*,*cols*:*image_name*
: This operation is used to shift the coordinates of an image.

crop:*init_row*,*end_row*,*init_col*,*end_col*:*image_name*
: This operation is used to select a portion of an image.

divide:*H_divisions*,*V_Divisions*,*portion_number*:**display**
: This operation is used to access a piece of a display image. The parameters specify the number of horizontal and vertical divisions and the index of the piece to be accessed (pieces are numbered starting at the upper-left corner, ending at the bottom-right corner).

extend:*fill_value*:*init_row*,*end_row*,*init_col*,*end_colimage_name*
: This operation is the opposite of **crop:**. Instead of reducing the size of an image, **extend** increases the image size by filling around the image with a constant value or with replicated pixels.

tee:*image_name_1*,*image_name_2*
: This operation is useful for putting an output image in two places at once.

magnify:*H_factor*,*V_factor*:*image_name*
: This operation is used to magnify or reduce an image.

In the remainder of this report, we will illustrate the use of Durra to build ''applications'' consisting of collections of GIL tasks (programs) connected through queues, and transmitting images as messages.

Before proceeding, however, it is important to clarify the intent of the examples. If all the tasks used in an application were restricted to having exactly one input and one output port, the advantages of using Durra might not seem obvious. That is, besides having some flexibility in the selection of tasks from the library, the structure declarations would look rather verbose compared with the UNIX shell syntax. However, as soon as we need to use tasks with multiple streams, the advantages of using Durra are clear. We are no longer

limited to single pipelines connecting input and output streams. Rather, we can build arbitrary graph-like configurations of tasks, each sending and receiving images through multiple ports. We took the Generalized Image Library as a starting point because their tasks and types are reasonably easy to describe and then we ‘‘extended’’ the library by assuming the existence of complex, multi-ported tasks and applications. In all fairness, there is nothing to prevent the development of multi-ported tasks in GIL. It is just that they would not fit very well within the UNIX pipelining paradigm and would be restricted to run separately, using image files as input and output parameters.

3. Type Declaration and Task Description Examples

In this section, we illustrate a number of Durra type declarations and task descriptions representative of the those that would be found in a GIL application.

3.1. Scalar Types

```
type integer is size 32;

type float is size 32;

type unsigned_8 is size 8;

type unsigned_16 is size 16;

type unsigned_32 is size 32;
```

Figure 3-1: Scalar Types

The declarations in Figure 3-1 illustrate scalar types in Durra. Types ‘‘integer’’ and ‘‘float’’ are declared to be 32 bits long. The unsigned types come in various sizes. Durra type sizes reflect the characteristics of the data produced by the various tasks in an application. The type size information is used by the Durra runtime environment [[3]] to allocate storage for the queues. The format of the data (e.g., sign, mantissa, and exponent fields), on the other hand, is of no concern to the Durra compiler or its runtime system.

3.2. Array Types

```
type integer_image is array of integer;

type float_image is array of float;

type unsigned_8_image is array of unsigned_8;

type unsigned_16_image is array of unsigned_16;

type unsigned_32_image is array of unsigned_32;
```

Figure 3-2: Array Types

The declarations in Figure 3-2 illustrate array types in Durra. The language allows for the specification of arrays of fixed dimensions as well as arrays of unspecified dimensions, like those in the examples above. For example, type ‘‘unsigned_08_image’’ is an array of unspecified size whose elements are of type ‘‘unsigned_08.’’ Of course, this is not adequate

for transmitting images between programs. The programs need to know the actual dimensions of the images, and these can be provided using Durra record types, as shown below.

3.3. Record Types

```
type rec_float_image is record
                  (rows: integer, columns: integer, data: float_image);

type rec_integer_image is record
                (rows: integer, columns: integer, data: integer_image);

type rec_unsigned_8_image is record
             (rows: integer, columns: integer, data: unsigned_8_image);

type rec_unsigned_16_image is record
            (rows: integer, columns: integer, data: unsigned_16_image);

type rec_unsigned_32_image is record
            (rows: integer, columns: integer, data: unsigned_32_image);

type rectangle is record
                (first_row: integer, last_row: integer,
                 first_col: integer, last_col: integer);
```

Figure 3-3: Record Types

The declarations in Figure 3-3 illustrate record types in Durra. The language allows for the specification of simple records, without Ada- or Pascal-style variants. Each field of a record is denoted by a field name and a field type. These examples declare records containing images of the types described before. These record types carry the right information so that programs can now pass images around.

The first five declarations in Figure 3-3 illustrate basic image types. For the sake of future examples, we have also declared a record type (the last declaration in the figure) that can be used to specify the position of a rectangle within an image array. Its use will become clear momentarily.

3.4. A Simple Task Description

To illustrate the use of the image type declarations introduced so far, let's assume that a program exists that can crop or select a portion of an image. The task description for this program is shown in Figure 3-4.

The example describes a task that crops an input image by selecting a portion of the image contained within some rectangle. The task has one input port, ''in_image,'' that receives images of type ''rec_floating_image'' (i.e., images whose pixels are 32-bit floating-point

```
task crop_image
  ports
    in_image  : in rec_float_image;
    parameters: in rectangle;
    out_image : out rec_float_image;
  behavior
    timing
      loop
          ((in_image || parameters)
                      -- get input image and rectangle description
            delay[0.1, 0.3]                -- compute cropped image
            out_image);                        -- put cropped image
  attributes
    implementation = "crop";
    processor = vax;
    author = "brahms";
end crop_image;
```

Figure 3-4: Task Description for a Cropping Task

numbers) and one output port, ‘‘out_image,’’ that sends images of the same type. In addition, it has a second input port, ‘‘parameters,’’ used to receive various parameters that control the task operation. The behavior of the task is described by the timing expression. It indicates that the task first reads an input image and a rectangle definition, then generates a cropped image (taking between 0.1 and 0.3 seconds for the operation), and finally writes the output image. The keyword **loop** indicates that this behavior is repeated continuously.

The attributes of the task description provide additional information about the task. These include the name of the task implementation (i.e., the actual executable program), the name of the processor on which this implementation can execute, and the name of the author of this implementation of the task.

3.5. A Simple Application Description

To illustrate the use of the type declarations and task descriptions introduced so far, let’s assume that we want to build a small application consisting of three tasks. The application would take an image from some input device (task 1), crop it (task 2), and send it to some output device (task 3).

The task introduced in Section 3.4 will serve as our ‘‘cropping’’ program. We also need task descriptions for the input and output devices, and these are illustrated in Figure 3-5.

The behavior of the input device task indicates that it loops continuously, generating and sending images of type rec_float_image through its single output port. The behavior of the output device task indicates that it loops continuously, consuming images of type rec_float_image from its single input port. This task also has one output port, which the task

```
task input_device
  ports
    out_image: out rec_float_image;
  behavior
    timing loop (delay[0.2, 0.2] out_image);
  attributes
    implementation = "scanner";
    processor = vax;
end input_device;
```

a -- Task Description for the Input Device

```
task output_device
  ports
    in_image  : in rec_float_image;
    parameters: out rectangle;
  behavior
    timing loop (parameters in_image);
  attributes
    implementation = "display";
    processor = vax;
end output_device;
```

b -- Task Description for the Output Device

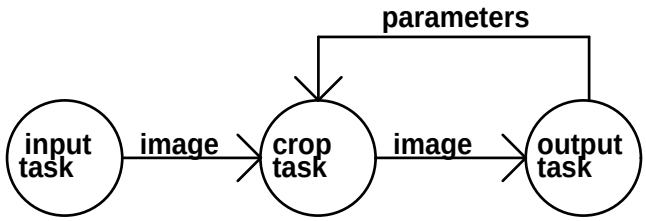
Figure 3-5: Input and Output Device Tasks

uses to send parameters specifying the cropping to be done to the original images before they can be displayed by the output device.

The application we want to build is a three-stage pipeline where images are passed from stage 1 (the input device) to stage 2 (the cropping task) to stage 3 (the output device). In addition, stage 3 sends the appropriate cropping parameters to stage 2. A picture of the pipeline and the Durra application description are illustrated in Figure 3-6.

The application description instantiates three processes (‘‘p1,’’ ‘‘p2,’’ and ‘‘p3’’) and three queues (‘‘q1,’’ ‘‘q2,’’ and ‘‘q3’’) that connect the processes’ ports in the obvious manner. The process declarations specify not only the name of the task we want to instantiate, but also the type and direction of its ports. This is usually required if an application library contains ‘‘families’’ of tasks, i.e., tasks that have the same name but that differ in other properties. In this particular example, we need to specify the port types because we could have device and cropping tasks that operate on images with integer pixels or with unsigned pixels of different lengths.

Although this application description is complete and correct, it suggests a deficiency in the task implementations, namely, that they only operate on images of exactly one type of pixel. This is not necessarily wrong. If we are dealing with some complicated operation, each implementation could use some carefully coded algorithm, optimized for each image type.




```
task crop_pipeline
  structure
    process
      p1: task input_device
              port
                out1: out rec_float_image;
            end input_device;
      p2: task crop_image
              port
                in1:  in  rec_float_image;
                in2:  in  rectangle;
                out1: out rec_float_image;
            end crop_image;
      p3: task output_device
              port
                in1:  in  rec_float_image;
                out1: out rectangle;
            end output_device;
    queue
      q1: p1.out1 >> p2.in1;
      q2: p2.out1 >> p3.in1;
      q3: p3.out1 >> p2.in2;
end crop_pipeline;
```

Figure 3-6: Application Description

For our simple cropping operation, on the other hand, it makes more sense to implement a general-purpose cropping task that operates on arbitrary image types. This can be achieved by using union types, as described below.

3.6. Union Types

To describe tasks that can transmit more than one type of data through the same port, Durra allows the specification of union types, whereby an object of a union type could be of any of a number of types specified in a union type declaration. For example, we could introduce the most general kind of image type via the declaration in Figure 3-7.a.

It is up to the sender and receiver tasks to agree on an interpretation of the image data. This can be achieved by, for instance, augmenting the image-carrying records with a field identifying the pixel type, as shown in Figure 3-7.b, and by using the convention that the value of the ‘‘pixel_type’’ field ranges between 0 (‘‘float_image’’) and 4 (‘‘unsigned_32_image’’).

```
type general_image is union
            (float_image, integer_image,
             unsigned_8_image, unsigned_16_image, unsigned_32_image);
```

a -- Union Type Description

```
type rec_general_image is record
                (rows: integer, columns: integer,
                 pixel_type: integer, data: general_image);
```

b -- Augmented Record Type Description

Figure 3-7: Images of Arbitrary Pixel Types

The tasks described in Section 3.5 can now be reimplemented to handle images of this general type. The corresponding task and application descriptions are shown in Figure 3-8.

Observe that in the new version of the application description the process declarations are simpler than in the previous version. There is no need to specify the port types since, by replacing all the type-specific tasks with one task that operates on arbitrary image types, the task name is now sufficient to uniquely identify the task in the library.

```
task input_device
  ports
    out_image: out rec_general_image;
  behavior
    timing loop (delay[0.2, 0.2] out_image);
  attributes
    implementation = "scanner";
    processor = vax;
end input_device;
```

a -- Task Description for the Input Device

```
task output_device
  ports
    in_image  : in rec_general_image;
    parameters: out rectangle;
  behavior
    timing loop (parameters in_image);
  attributes
    implementation = "display";
    processor = vax;
end output_device;
```

b -- Task Description for the Output Device

```
task crop_image
  ports
    in_image  : in rec_general_image;
    parameters: in rectangle;
    out_image : out rec_general_image;
  attributes
    implementation = "crop";
    processor = vax;
    version = 12;
end crop_image;
```

c -- Task Description for the Cropping Task

```
task crop_pipeline
structure
    process
        p1: task input_device;
        p2: task crop_image;
        p3: task output_device;
    queue
        q1: p1.out_image >> p2.in_image;
        q2: p2.out_image >> p3.in_image;
        q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

d -- Application Description

Figure 3-8: Operating on Arbitrary Image Types

4. Task Selection Examples

The previous examples illustrated two ways of identifying a library task in a process declaration. The task selection could be based just on a task name (Figure 3-8.d), or based on the number, direction, and type of its ports (Figure 3-6). The language supports a rather elaborate set of task selection features as illustrated by the examples below.

4.1. Rules for Matching Task Selections with Task Descriptions

If a task selection contains port declarations, the port names provided in the task selection override the port names provided in the task declaration (that is, queue declarations would have to refer to the port names used in the task selection, and not to the port names used in the task description). The port declaration lists must otherwise be identical, i.e., the number, the order, the directions, and the types must be identical.

If a task selection provides a signal declaration clause, the signal declaration list must be identical to that provided in the task description, i.e., the names, number, and directions must be identical.

If a task selection provides a functional behavior predicate or a timing expression, the corresponding predicate or timing expression in the task description must imply that of the task selection. Currently there are no facilities to check these implications and timing expressions, so for the time being the behavioral information part of a task description is treated as commentary information.

If a task selection specifies an attribute not present in a task description, no match occurs, i.e., the compiler skips this description and continues searching for a candidate. If a task description provides an attribute not specified in a task selection, the attribute is ignored.

If a task selection provides a predicate (a disjunction) for an attribute, a matching task description must provide values that satisfy the predicate, i.e., the disjunction yields **true** when evaluated in the context of the values declared for the attribute.

4.2. Multiple Implementations of the Same Task

To continue with the cropping task introduced before, let's assume the we have several implementations of a cropping task. For each of these task implementations, we need to write a task description. A collection of these task descriptions appears in Figure 4-1. These task descriptions have the same interface and exhibit similar behavior. In addition, all of them have ''implementation'' and ''processor'' attributes (the ''implementation'' attribute values are, of course, different). They exhibit more variability in their other attributes or attribute values.

```
task crop_image
  ports
    in_image  : in rec_general_image;
    parameters: in rectangle;
    out_image : out rec_general_image;
  attributes
    implementation = "crop";
    processor = vax;
    version = 12;
end crop_image;
```

a -- Implementation 1

```
task crop_image
  ports
    in_image  : in rec_general_image;
    parameters: in rectangle;
    out_image : out rec_general_image;
  attributes
    processor = vax;
    implementation = "crop_1";
    author = "brahms";
end crop_image;
```

b -- Implementation 2

```
task crop_image
  ports
    in_image  : in rec_general_image;
    parameters: in rectangle;
    out_image : out rec_general_image;
  attributes
    processor = vax;
    implementation = "crop_2";
    author = "rachmaninoff";
    date = "2/22/88";
end crop_image;
```

c -- Implementation 3

```
task crop_image
  ports
    in_image  : in rec_general_image;
    parameters: in rectangle;
    out_image : out rec_general_image;
  attributes
    processor = vax;
    implementation = "crop_3";
    author = "mahler";
    date = "2/25/88";
end crop_image;
```

d -- Implementation 4

Figure 4-1: A Collection of Cropping Task Descriptions

The first task description in Figure 4-1 has a ‘‘version’’ attribute (with value ‘‘12’’). The second task description has no ‘‘version’’ attribute, instead it has an ‘‘author’’ attribute (with value ‘‘brahms’’). The third task description has an ‘‘author’’ attribute (with value ‘‘rachmaninoff’’) and a new attribute, ‘‘date’’ (with value ‘‘3/22/88’’). Finally, the fourth task description is similar to the previous one in that it has both ‘‘author’’ and ‘‘date’’ attributes but with different values (‘‘mahler’’ and ‘‘2/25/88,’’ respectively).
We can now use this collection of task descriptions to build different versions of the same application.

4.3. Constrained Task Selections

If we have multiple task descriptions for the image cropping task, the simple pipeline application description of Section 3.6 and Figure 3-8 will not compile properly. The Durra compiler will complain that there is an ambiguity: Multiple library entries match the task selection used to declare process p2. This is because there are a number of candidate task descriptions with the same name (‘‘crop_image’’). A task selection must now include additional information to resolve the ambiguity.

Figure 4-2 shows several alternative descriptions for the pipeline application. Each of these application descriptions selects a different version of the cropping task by specifying a different attribute expression in the task selections.

The language allows for more elaborate attribute matching patterns than just value equality, as the previous examples illustrate. A task selection can specify an arbitrary attribute expression. In this case, the expression is evaluated in terms of the attribute values given in the task description. A match occurs if the value of the expression used as a task selection attribute value is satisfied (TRUE). The example in Figure 4-3 illustrates this feature. The task selection specifies a library task whose value for the ‘‘author’’ attribute is either ‘‘bach’’ or ‘‘mahler.’’

The name of an attribute can appear in any context in which its value can appear. For instance, if the application developer defines an attribute ‘‘Queue_Size’’ with an integer value, then ‘‘Queue_Size’’ can appear anywhere an integer value is expected. This permits the developer to name, say, a queue size and use the name to declare queues with identical size in a number of task descriptions. Another use is to instantiate ‘‘families’’ of tasks, i.e., tasks that share the same value for some attribute. For instance, the example in Figure 4-4 specifies that the task to be selected from the library must be a task description whose ‘‘author’’ attribute value must be the same as that of the ‘‘author’’ attribute of the task selected for the input device task, whatever that is. This is achieved by not giving a specific value as the ‘‘author’’ attribute value for the task selected for process p2 but giving the global name for the ‘‘author’’ attribute of the input device task, ‘‘p1.author.’’

```
task crop_pipeline
  structure
    process
      p1: task input_device;
      p2: task crop_image attributes version = 12; end crop_image;
      p3: task output_device;
    queue
      q1: p1.out_image >> p2.in_image;
      q2: p2.out_image >> p3.in_image;
      q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

a -- Implementation 1

```
task crop_pipeline
  structure
    process
      p1: task input_device;
      p2: task crop_image attributes author = "brahms"; end crop_image;
      p3: task output_device;
    queue
      q1: p1.out_image >> p2.in_image;
      q2: p2.out_image >> p3.in_image;
      q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

b -- Implementation 2

```
task crop_pipeline
  structure
    process
      p1: task input_device;
      p2: task crop_image attributes date = "2/22/88"; end crop_image;
      p3: task output_device;
    queue
      q1: p1.out_image >> p2.in_image;
      q2: p2.out_image >> p3.in_image;
      q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

c -- Implementation 1

```
task crop_pipeline
  structure
    process
      p1: task input_device;
      p2: task crop_image attributes author = "mahler"; end crop_image;
      p3: task output_device;
    queue
      q1: p1.out_image >> p2.in_image;
      q2: p2.out_image >> p3.in_image;
      q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

d -- Implementation 1

Figure 4-2: Simple Attribute Matching

```
task crop_pipeline
  structure
    process
      p1: task input_device;
      p2: task crop_image
            attributes
              author = "bach" or author = "mahler";
          end crop_image;
      p3: task output_device;
    queue
      q1: p1.out_image >> p2.in_image;
      q2: p2.out_image >> p3.in_image;
      q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

Figure 4-3: Expression Attribute Matching

```
task crop_pipeline
  structure
    process
      p1: task input_device;
      p2: task crop_image
            attributes
              author = p1.author;
          end crop_image;
      p3: task output_device;
    queue
      q1: p1.out_image >> p2.in_image;
      q2: p2.out_image >> p3.in_image;
      q3: p3.parameters >> p2.parameters;
end crop_pipeline;
```

Figure 4-4: Global Attribute Matching

4.4. Summary of Task Selection Features

Task selections are templates used to identify and retrieve task descriptions from the library.

A given task, e.g., edge-detection in a vision application, might have a number of implementations that differ along dimensions such as algorithm used, code version, performance, or processor type. To select among a number of alternative implementations, the application developer provides a task selection as part of a process declaration. This task selection lists the desirable features of a suitable implementation.

Syntactically, a task selection looks somewhat like a task description without the **structure** part, and in which all other components are optional. For example, while a task declaration

requires the declarations of the ports, in a task selection the declaration of the ports is optional.

The name of a task is the minimal part of a task selection. Local, port names can be introduced by providing a port declaration, provided that the number, direction, and data types of the ports specified in the task selection are identical to those specified in the task description. If the port declarations are left out of the task selection, the original names used in the task description are used instead.

A task can be identified and selected from the library just by its name (if the name is unique in the library), by its interface properties (e.g., port types), by its attributes (e.g., version number), by its functional or timing behavior (e.g., a pre-condition), or by any combination of all of these. This degree of flexibility in the specification of the properties of a task is a step in the direction of development of software reuse as a methodology for the development of complex, distributed software systems.