A Task-level Description Language: Preliminary Design

M.R. Barbacci
Software Engineering Institute and
Department of Computer Science

and

J.M. Wing
Department of Computer Science

Carnegie Mellon University
Pittsburgh, PA 15213-3890
3 July 1986

1. Context

Computation-intensive, real-time applications such as vision, robotics, and vehicular control require efficient concurrent execution of multiple *tasks*, e.g. sensor data collection, obstacle recognition, and global path planning, devoted to specific pieces of the application. Since the speed and resources required of each task may vary, these applications can best exploit a computing environment consisting of multiple special- and general-purpose processors that are logically, though not necessarily physically, loosely connected. We call this environment a *heterogeneous machine.*

During execution time, *processes*, which are instances of tasks, run on possibly separate processors, and communicate with each other by sending messages of different types. Since the patterns of communication can vary over time and the speed of the individual processors can vary over a wide range, additional hardware resources, in the form of switching networks and data buffers are also required in the heterogeneous machine.

The application developer is responsible for prescribing a way to manage all of these resources. We call this prescription a *task-level application description.* It describes the tasks to be executed and the intermediate queues required to store the data as it moves from producer to consumer processes. A *task-level description language* is a notation to write these application descriptions. The problem we are addressing is the design of a task-level description language.

We are using the term "description language" rather than "programming language" to emphasize that a task-level application description is not translated into object code in some kind of executable "machine language". Rather, it is to be understood as a description of the structure and behavior of a logical machine, to be synthesized into resource allocation and scheduling directives. These directives are to be interpreted by a combination of software, firmware, and hardware in a heterogeneous machine.

2. Current Status

2.1. Scenario

We assume that each of the processors in a heterogeneous machine has languages, compilers, libraries of programs, and other software development tools that cater to the special properties of a processor's architecture. For example, if an image processing application requires a task for computing matrix multiplication, we assume code for it exists on one or more processors in the heterogeneous machine, perhaps in assembly language on a systolic array processor or in C on a Sun workstation.

>From the user's (i.e. application developer's) viewpoint we imagine the following scenario of how he or she is to use our task-level language.

1. The user writes a task-level application description. This description contains (1) *task selections*, which are used to retrieve *task descriptions* and, ultimately, *task implementations* from the library, and (2) *a process-queue graph*[1], which is used to describe the connections between processes (instances of tasks) and queues (means of communication between processes).
2. The user compiles this description. The compiler generates a set of resource allocation and

[1] akin to a dataflow graph where processes are nodes and queues are arcs.

scheduling commands to be interpreted by the scheduler.

3. The user links the output of the compiler with the appropriate run time support facilities and loads the resulting scheduler "program" on the scheduler.
4. The scheduler loads the task implementations (i.e. the real code) on the processors and interprets the scheduling commands and initialization code for the machine.
5. The heterogeneous machine runs the processes on processors as dictated by the schedule.

Although this scenario is currently biased toward a static configuration of processes, we do not wish to preclude dynamic reconfiguration from our language. Since we are interested in language design we will focus on the first step of our scenario, that of creating a task description.

2.2. Task Descriptions

Task descriptions, written and stored in task libraries, are building blocks for task-level programs. A task-description (see, for example, Figure 1a) contains the essential information that an application programmer needs to build task-level applications descriptions. The two most interesting kinds are *performance* information and *structural* information. All other information, such as task, queue, and port names, input and output data types flowing in queues and through ports, exceptional conditions signalled or handled, can either be checked by the compiler or treated as documentation.

Performance information comes in the form of (1) *timing expressions* under **timing**, used to indicate what protocol the task uses to consume and produce data and (2) *attributes* such as **speed** and **resources**, under **attribute**.

Structural information, found under **structure**, defines a process-queue graph (see, for example, Figure 1b) and possible dynamic reconfiguration. For the sake of brevity, we discuss only two features of defining a process-queue graph: process declarations and queue-connect statements. First, a process declaration of the form

process-name: **task** *task-selection*

creates a process as an instance of the specified task. Since a given task (e.g. convolution) might have a number of different implementations that differ along different dimensions such as algorithm used, code version, performance, processor type, etc., the task selection in a process declaration specifies the desirable features of a suitable implementation[2], and is used to select among a number of alternative implementations.

Second, a queue-connect statement such as

navigator.out > road-selection > road-predictor.in

creates a queue through which data of type road-selection flows between the output port of the navigator process with the input port of the road-predictor process. Here, navigator and road-predictor must have been declared as processes.

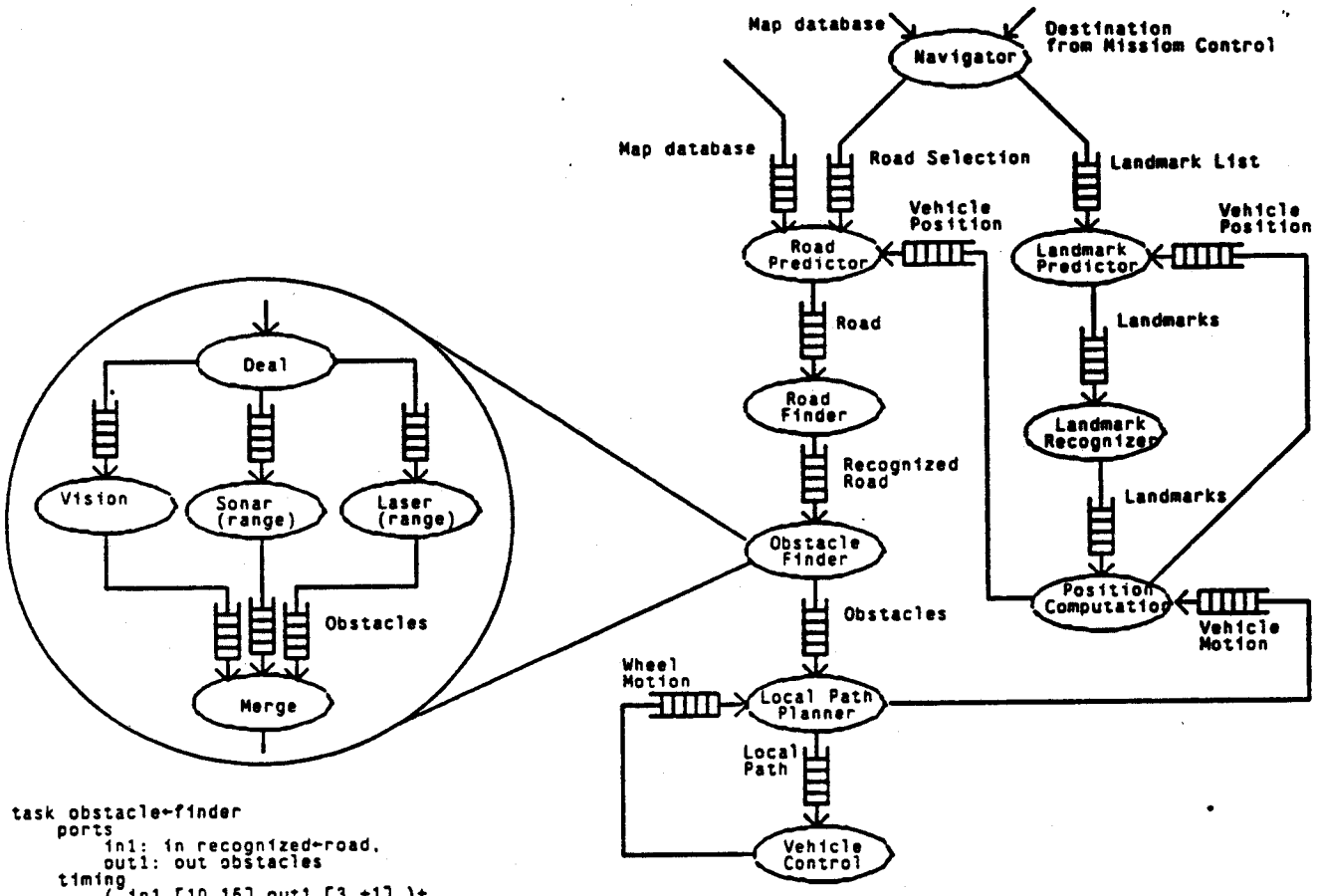
Note that since task selections appear within task descriptions, there is direct linguistic support for

[2]The task selection contains at least the name of a task and, optionally, attribute, interface, functional, performance requirements, i.e., anything, but structural information.

reflecting hierarchically structured tasks. This is illustrated by the encircled graph to the left of Figure 1b, which depicts the process-queue subgraph of the obstacle-finder task, which itself is a component of a larger description.

3. Plans for the Immediate Future

Although the goal is to design and implement a a task-level programming language that can be used for different architectures and for varying applications, our first intended use is bound by both a specific architecture and by specific applications. In particular, our target architecture is the HET0 machine, currently being designed in the Computer Science Department, CMU. The highlights of this machine are a cross-bar switch, intelligent buffers on the input sockets, and a scheduler processor that can communicate with all processors, buffers, and the switch. Our target application is an autonomous land vehicle, with an initial bias toward the vision-related tasks.



(b) Process-Queue Graph

```
task obstacle-finder
    ports
        in1: in recognized-road,
        out1: out obstacles
    timing
        ( in1 [10,15] out1 [3,+1] )+
    attribute
        author = "MRB",
        speed  = "50 msec"
    structure
        pdeal:  task deal attribute mode = BY-TYPE end,
        pmerge: task merge attribute mode = FIFO end,
        psonar: task sonar,
        plaser: task laser attribute processor = warp1,

        pdeal.in1   = obstacle-finder.in1
        pdeal.out1  > sonar-road > psonar.in1,
        psonar.out1 >            > pmerge.in1,
        pdeal.out2  > laser-road > plaser.in1,
        plaser.out1 >            > pmerge.in2,
        pmerge.out1 = obstacle-finder.out1

        if time-of-day = [ 6:00:00 , +12:00:00 ]
        then    begin                -- dynamic reconfiguration
                pvision: task vision attribute processor = warp2 end,

                pdeal.out3   > vision-road > pvision.in1,
                pvision.out1 > obstacles   > pmerge.in3
                end
end task
```

(a) Task Description

Figure 1 -- Task-level Description Language