GARTL: A Real-time Programming Language Based on Multi-version Computation

Chris Marlin Wei Zhao Graeme Doherty Andrew Bohonis

Department of Computer Science, The University of Adelaide,
G.P.O. Box 498, Adelaide, South Australia 5001, Australia.

Abstract

The increasing importance of real-time computing systems is widely known and such systems are presently the subject of much research. A particularly attractive approach to the programming of hard real-time systems is the identification of multiple versions of the task to be carried out. If this is done, then the system scheduler will be able to select the version which gives the most precise results in the time available - the more time available, the more precise the results. This approach to programming hard real-time systems is called multi-version computation. This paper explores the question of suitable language support for multi-version computation through the description of GARTL, a real-time programming language. Some aspects of the implementation of GARTL are also discussed.

1. Introduction

In the last few years, there has been increased interest in the design and analysis of real-time computer systems [11,14,16]. New, complex real-time applications are being envisioned, such as the space station, teams of mobile robots performing undersea exploration, and the integration of expert systems into embedded real-time applications such as avionics and process control. In all these systems, real-time software plays a critical role.

The principal problem in writing software to execute in a hard real-time environment is well-known: the software must be written so that the various hard real-time deadlines are met without fail. Since the real-time system will, in general, consist of various components or tasks, the above problem becomes in practice the more difficult problem of setting and achieving a number of subsidiary deadlines, one for each task which has to be carried out. Some of the tasks to be carried out will be periodic, such as those concerned with sensors on the external world. Other tasks will be aperiodic; examples of these include tasks which are activated by other tasks.

For all tasks, it may be possible to identify multiple versions of the task, which may have quite different resource requirements. In particular, the different versions may require different lengths of time for their execution. Typically, the version which takes the longest time to execute will also produce the most accurate result, yield the most information, or be superior to the others in some significant manner. The other versions will take smaller and smaller amounts of time to execute, but will produce results of lower and lower quality.

For example, consider a missile detection and identification system on, say, a ship. One would expect that, provided some minimum amount of time is available, this system would be capable of at least detecting the presence of a missile near the ship and indicating that it appears to be getting closer to the ship. If more time is available, the system may be able to provide more detailed information on the missile, such as its speed and altitude. If still more time is available, it might be possible for the system to also identify the missile type. These levels of information correspond to different versions of the missile detection and identification task.

The question of how long a particular version will take to execute is clearly a complex one and this execution time will frequently depend critically on parameter values and other conditions prevailing in its environment at the time of invocation. It will also be necessary to have some estimate of the time available for the task to execute; we will assume in the remainder of this paper that this latter estimate is available when required.

The focus of this paper is the design of a programming language called GARTL to support the above approach to programming hard real-time systems, and is part of a larger project concerned with multi-version computation as an approach to meeting hard real-time deadlines [3,6,7,8,9,10,12,15]. In terms of language support, it is necessary to provide the ability to declare the multiple versions of a task; among other things, this declarative mechanism will need to indicate the attributes of task upon which the execution is thought to depend. The other major aspect of language support is, of course, the invocation of the tasks; this language mechanism must have the ability to specify the task to be activated, the lowest acceptable version of the task and the deadline for completion of the task, among other things. An important innovation in the GARTL language is the way it permits interaction between the tasks and the system scheduler; this kind of interaction is extremely useful in programming hard real-time systems.

In the remainder of this paper, the GARTL language features to support the multi-version computation approach to real-time programming are first described. Then, some aspects of the implementation of the language are discussed. Finally, some conclusions are presented and planned future work is outlined.

2. The GARTL Language

The GARTL language has been designed as an extension of the C language[5]. In this section, the more important of the extensions are described and some of the design issues discussed.

2.1 Task specification

As shown in Figure 1, a GARTL task is specified in the form of a section of code similar to a C function of type void. This code executes in parallel with other tasks. A task may not access any variables which are not either declared locally, or declared as parameters to the task - the reason for this restriction is to make the dependencies of the task as clear as possible, and to simplify
the process of predicting the execution time in the
As in Figure
and they may be
I/O
cerned. Thus,
on a
write)
sibly
2
rametcr. in such
decland
ports, or other system

Figure 1. The form of a task specification in GARTL.

Resources
The resources specified in the "(resource use list)" may be files, I/O ports, or other system resources. The specific operations allowed on a resource will depend on the nature of the system resource concerned. Thus, for example

- a file resource might be readable and writable via the relevant system functions,
- an input port resource might be read-only, again via system functions, and
- a global RAM resource might be readable and writable via the relevant system functions.

The resources that a task requires must be specified in the task declaration as one of "readable, writeable or readable and writable; to achieve this, the syntax of each element in "(resource use list)" above has the following syntax (in its simplest form):

[read | write | readwrite] on (name list);

The scheduler will allow a number of tasks to be scheduled for possibly simultaneous execution if they only require read access on a particular resource; if a task is scheduled that requests write (or readwrite) access on a resource, then no other task can be scheduled for possibly simultaneous execution which also requests that resource either for reading or for writing. For example, the task "T1" in Figure 2 requires read access on resources "r1", "r2", "m1" and "r3", and write access on "m1" and "fl".

Figure 2. A GARTL task using various resources.

it is possible for the required resource to remain unspecified until the time that the task is scheduled for execution.

Global resources
Each resource which is not a parameter of the task can only be a resource which is globally declared. This allows the specification of an association between a name to be used within the program and an operating system resource (such as a file or an I/O port). Such a declaration has the form:

resource (name) is "(system name)" [raises (exception name)];

where "(name)" is the name by which the resource will be known throughout the program, "(system name)" is some identification of the system resource, and "(exception name)" is the name of an exception (see Section 2.4 below). For example, the declaration

resource DB is "/database" raises missing_DB;

declares a file resource with the local name "DB", and which is actually the file called "database" in the current directory; if it is not found, the exception "missing_DB" is raised, which might perhaps result in the creation of the file.

When a program begins execution, an exception is raised for each declared resource that is not present. The order in which the exceptions occur is not specified, but all are guaranteed to occur before the body of the main task begins execution. If the user specifies an exception name in the resource declaration, then this exception is raised for the main task. If the user does not specify an exception, however, then the exception "resource_missing" is raised. The default action for dealing with any exception which has no handler specified for it (including the "resource_missing" exception) is to terminate program execution. The exception handling facility described in Section 2.4 can be used to create the resource and then continue execution.

Preeemptable tasks
As shown in Figure 1, the task specification may be prefixed by the keyword "preeemptable" to indicate that it can be interrupted. When a task is scheduled, it has guaranteed access to all of its requested resources, and no other task should be scheduled that will violate this access. In designing a task, the programmer can assume that any changes that occur to a resource during the task's execution will not be the result of the action of some other task. That is, any changes that occur will only be caused either by the task itself or by some external stimuli, such as the arrival of data on an I/O port.

However, this can lead to the inefficient application of available resources to meet deadlines. For example, imagine that a task "T1" which will take 20 seconds has a deadline in 25 seconds and has just been scheduled on a system with one processor. Now imagine that another task "T2" requests scheduling; this task has a deadline in 5 seconds and will take 5 seconds. If "T1" could be interrupted, or preeempted, then both deadlines can be met; on the other hand, if "T1" is not interrupted, "T2" will certainly not meet its deadline. Rather than have the GARTL language implementation attempt to discern whether it is possible to interrupt a task's execution without unfortunate effects, the present version of the language requires the programmer to indicate whether a task can be safely interrupted by prefixing the task specification with the keyword "preeemptable".
Version numbers

Another aspect of the task specification is the provision of version numbers. As indicated in Section 1, the GARTL language supports an approach to the programming of hard real-time systems which involves the provision of multiple versions of a task. The scheduling of tasks is always based on the assumption a higher numbered version of a task is more desirable (say, in terms of the accuracy of the result) than a lower numbered version. It is up to the programmer to use version numbers consistent with this assumption. Consequently, the task specification includes the ability to specify a range of version numbers and to provide access to the version number of the current instance of the task through a version variable, in the following manner:

```c
#define <version variable>/</maximum version>
```

The version variable is implicitly an unsigned integer. Within the declarations and body of the task, the version variable can only have its value read. To facilitate the construction of the database of performance measurements and resource requirements used by the implementation (see Section 3), an upper limit on the version number must be specified by the user. For example, the task specification for "Refine" in Figure 3 indicates that the task has 30 versions, numbered from 1 to 30. In this case, the version number is held in the version variable "Ver" and is used to control the execution of a refinement loop in a simple manner. Declaring a task without any specification of multiple versions is equivalent to declaring the same task with a maximum version number of 1.

```c
task Refine#Ver:30(x)
{
  int x;
  int i;
  for (i=0; i<Ver; i++) { /* refine "x" */ }
}
```

Figure 3. A simple GARTL task which has multiple versions.

When a task has multiple versions, the different versions may not all have the same resource requirements. For example, a low numbered version may use a global RAM resource to hold a previously calculated approximate result, while a higher numbered version might look up a file and calculate a more accurate result. This is illustrated in Figure 4, where the task "GetName" has two versions: the first version requires only access to "last", whereas the second version also requires access to "the_file". The example in Figure 4 also shows how the memory requirements of a task can depend on the version number, in that version 1 has an array with 100 elements and version 2 has an array of size 200.

```c
task GetName#V2(id, name)
{
  int id;
  char *name;
  using
  readwrite on last : 1, 2;
  /* Both versions use "last" */
  read on the_file : 2;
  /* Only the second version uses "the_file" */
  { int my_array[V1=100];
    ...
  }
}
```

Figure 4. A GARTL task where the multiple versions have different resource requirements.

Monitored variables

As will be explained in more detail later, when a request for a task to be scheduled is submitted to the scheduler, the scheduler looks at the values of a subset of the parameters passed to the task, and uses its knowledge of past executions to determine if it is possible to successfully schedule a suitable version of the task. The parameters to be regarded as significant in predicting execution time from past executions of the task are specified by the programmer.

These are known as monitored variables. Using knowledge of the computation carried out by the task, the programmer must choose the parameters which are important in deciding how long a task will execute. Assignments can be made to monitored variables within a task, but it is only their values at the time of the scheduling request that is important in deciding how long it will take. If a task has no monitored variables, it is assumed that the time of execution of the task will be independent of the inputs, and always the same for a particular version of the task.

For example, Figure 5 depicts a task with monitored variables. The declaration of a monitored variable, such as that for "len", contains a list of collections of possible values for the monitored variable. Each collection is enclosed in square brackets and defines an equivalence class of values of the corresponding variable from the point of view of predicting the execution time of the task. Thus, the declaration in Figure 5 defines three equivalence classes of values and indicates to the language implementation that maintaining three sets of execution time records for this task will be sufficient to predict future execution times. The three equivalence classes for monitoring "len" defined by the declaration in Figure 5 are:

1. the values between 0 and 25 inclusive, along with the values between 27 and 49 inclusive,
2. the value 26, and
3. the values from 50 up to the maximum representable integer.

It is the programmer's responsibility to ensure that actual parameter values fall within one of the equivalence classes declared; if this assumption is violated, a "monitor.range" exception will be raised for the calling task. Only parameters with scalar types (such as "int" and "char"), or with floating point number types may be used as monitored variables. Because of the difficulty of specifying ranges of values for floating point number types, whenever a value occurs in a number of equivalence classes, the value is regarded as belonging to the last of the classes specified.

```c
task T1#V2(len, struct list.array list)
{
  /* "len" is the number of elements of "list"
    actually used */
  monitor
    len: [0..25][26..49][50..maxint];
    { /* body */ }
}
```

Figure 5. A task with monitored variables.

2.2 Mailboxes and messages

The message facility of GARTL provides a means for tasks to communicate among themselves, and for communication between tasks and the system scheduler. In the remainder of this section on mailboxes and messages, the term "tasks" may be assumed to include
the scheduler.) As far as the programmer is concerned, messages may be regarded as blocks of memory copied between tasks, via the system scheduler, using library routines of system calls. On the other hand, mailboxes are areas of memory local to task instantiations, and messages are sent to these areas.

Mailboxes may only be declared as variables local to tasks; in particular, they may not be declared within functions. When a message is sent, it must be sent to a particular mailbox. If the task that owns the mailbox is active, then the message will be copied via the system scheduler into the mailbox, where it can be read by the receiving task at some later point. Messages sent to mailboxes whose owning task is not active are simply lost. The last place in a mailbox is special in that if a mailbox is full, any new arriving messages will overwrite the last message in the mailbox.

Figure 6 depicts a task which declares two mailboxes: "MBint" and "MBpers". The mailbox "MBint" is intended to receive up to four messages containing integers, and "MBpers" is a mailbox which can hold only three messages containing integers, and "MBpers" is a mailbox which can hold only one message at a time, each such message containing a structure of the type shown.

```c
task R () {
    mailbox int MBint[4];
    mailbox
        struct {
            char name[8];
            int age;
        } MBpers[1];
    /* body */
}

Figure 6. A task containing two mailboxes.
```

In order to describe the location of a mailbox, there is a predefined type called "mailaddress". The mail address of a mailbox can be found using the "mailaddr" operator on a mailbox. For example, the code in Figure 7 sets the variable "MA" to the mail address of the mailbox "MB", where the contents of the latter are structures of the type shown in the figure.

```c
/* declarations */
mailbox struct {
    char name[8];
    int ID;
    MB[20];
    mailaddress MA;
} /* code */
MA = mailaddr MB;

Figure 7. Declaring and using mail addresses.
```

A message can be sent to a mailbox using the "send" library function. This takes as arguments the address of a message, and a mail address. It sends a copy of the message to the mailbox addressed. The number of bytes sent is just enough to fill one of the mailbox's messages and is not related to the type of the local copy of the message. Thus, it is the programmer's responsibility to make sure the messages are of the correct size. In a similar manner, messages can be received from a mailbox using the "recv" library function, but only if the mailbox has been declared local to the current task. When a message is received, a copy is placed into the memory starting at the address given by the second parameter to "recv". When this is done, the mailbox is allowed to reuse the space for another message; thus, if the mailbox is full, it will be able to receive one more message before it has to start overwriting messages again. The "recv" function returns the number of messages that the mailbox contained prior to the message being read out. If an attempt is made to read a message from a mailbox not declared locally, or if an attempt is made to read a message from a mailbox that does not currently hold a message, the returned message is undefined. The number of messages that have arrived in a locally declared mailbox can be tested with the "arrived" library function.

2.3 Task invocation

At any one time, there are likely to be a number of tasks executing, and a number that are ready to be executed. Except for the main task, the execution of every task is initiated by another task. In order to start a new (or child) task, the existing (or parent) task makes a scheduling request to the system scheduler. The parent task informs the scheduler of:

- the parameters to pass to the child task,
- which versions of a multiple-versioned task are acceptable, and
- how the child is to be executed.

There are essentially two ways in which a child can be executed: synchronously and asynchronously. In addition, there are two ways that the scheduler can be called; again, these are known as calling the scheduler synchronously and asynchronously. All four combinations of these possibilities are allowed and are discussed below.

When the scheduler receives a scheduling request, it estimates what the resource requirements of the new task will be, including processor time, and decides on the highest numbered version of the task that can be executed. If the highest version number the scheduler expects to be able to complete is nevertheless smaller than the minimum specified by the programmer, then the child task is not scheduled and is hence not executed.

A scheduling request will specify the task to be scheduled, the minimum allowable version number and the parameters to be transmitted to the task if it is successfully invoked. For example, the scheduling request

```c
call T1(x,y,z)
```

will schedule task "T1" with parameters "x", "y" and "z" only if version three or higher can execute successfully, and the request

```c
call T2(x+y)*c (d,e)
```

schedules "T2" with parameters "d" and "e" only if the best version which can be scheduled is greater than or equal to the result of the expression "(x+y)*c". If no minimum version is specified, then any version is regarded as acceptable. Note, however, that it is still possible that no child will be scheduled (in the case that even the lowest version cannot be scheduled in the time allowed).

Scheduler results

It is often desirable to know if a task has been successfully scheduled or not. In order to obtain this information back from the scheduler, it is possible to pass to the scheduler the address of a mailbox where the scheduler can write an integer representing the results of its efforts; this is done by giving its name before the task name in the invocation command. As soon as the scheduler has decided whether the child task can be scheduled or not, it sends an
An integer message to the given mailbox. This integer is the maximum version of the child task which can be executed. As mentioned above, if the integer is less than the minimum specified by the programmer for the task, then the child task is not actually scheduled for execution.

For scheduling requests that request multiple instances of a task (see “Periodically scheduled tasks” below), the scheduler may be provided with a mailbox that can hold multiple integer messages. Each element in the mailbox will hold the scheduler's result for that instance of the child. That is, the first element will hold the maximum version for the first instance, the second element will relate to the the second instance, and so on. If the mailbox fills up, the results will keep overwriting the last message in the mailbox.

As an example, consider the code in Figure 8. In this fragment of code, the integer variable "SR" is used to hold the result from the scheduler via the message in the mailbox "SMB". The parent task then uses the contents of this variable to decide what action to take.

```c
{  
  mailbox int SMB[1];  
  mail address MA = mailaddr SMB;  
  int SR;  
  int give-up=0;  
  call (MA) recognise#4();  
  /* the first attempt at "recognise" */  
  recv(MA, &SR);  
  /* now know scheduler has sent message */  
  upto 100 while ((SR<4) && (!give-up)) {  
    if (SR=0)  
      give-up = 1;  
    else  
      slow-down();  
    /* take recovery action */  
    call (MA) recognise#4();  
    /* and try again */  
    recv(MA, &SR);  
  }  
}  
```

Figure 8. Using results returned by the scheduler.

Synchronous task scheduling

When a child task is scheduled synchronously, the parent task suspends execution until the child finishes. As mentioned earlier, the scheduler may be invoked either synchronously or asynchronously in the case that the task is to be scheduled synchronously. If the scheduler is to be invoked synchronously, which is illustrated in Figure 9(a), the call syntax seen earlier is used and the parent suspends itself as soon as the scheduling request is made. In Figure 10, for example, the parent requests that the task "T1" be executed synchronously. At the point immediately after the scheduling request on the second line of Figure 10, the programmer can assume that the scheduler has decided whether or not "T1" will be scheduled. To find out the nature of this decision, the "SR" mailbox will need to be checked.

If, on the other hand, the scheduler is to be invoked asynchronously, one uses the keyword async call in place of call; in this case, shown in Figure 9(b), the parent continues execution until the scheduler is finished the process of deciding if the child can execute. If the child can execute, the parent will then suspend and wait until the child has terminated; this would normally be used where the child needs access to some resource being used by the parent. The parent will not terminate until all synchronously scheduled children have terminated. Note that the child does not necessarily start executing immediately (see "Timing constraints" below) and so the parent may have a long wait.

For scheduling requests which potentially invoke multiple instances of a task (see "Periodically scheduled tasks" below), the parent does not continue execution until the scheduler has decided on the success or failure of the scheduling of all the child tasks, and until the last child task which was successfully scheduled has terminated. If the scheduler was called asynchronously, the parent continues execution until the scheduler has decided if the first child can be scheduled successfully before suspending its own execution.

Asynchronous task scheduling

When a parent task makes a successful request for a child to be executed asynchronously, the child may be executed at any time within its timing constraints. This means that the parent task may terminate before the child task. Again, the scheduler may be called synchronously or asynchronously:

- if the keyword sched is used, then the scheduler is called synchronously, and
- if async_sched is used, the scheduler is called asynchronously.

The synchronous scheduler call is illustrated in Figure 11(a) and would be used if the parent was interested in knowing what the scheduler results were; Figure 11(b) shows the asynchronous version, which would be used if the parent was not interested in these results.

In a similar fashion to synchronous task scheduling, a synchronous call to the scheduler will cause the parent task to wait until the
scheduler has decided on the success of the scheduling of all instances of the child task, but it will not wait for the actual execution of the children. An asynchronous call to the scheduler will allow the parent to continue regardless of whether the scheduler or the child tasks have finished.

Consider the code in Figure 12, in which “T1” is scheduled asynchronously and the parent will be forced to wait until the scheduler has decided on the success of the request before continuing. This means that the parent can be guaranteed access to a non-empty mailbox at the point shown, since the scheduler will have finished its work by that point. If the version chosen is then found to be less than the minimum, some recovery action is taken.

```
sched (mailaddr MBX) T1();
recv(mailaddr MBX, &SchedRes);
/* MBX holds result messages */
if (SchedRes<7) recovery_action();
```

Figure 12. An example of asynchronous task scheduling.

Timing constraints

Various timing constraints can be specified in GARTL on a scheduling request:

- the *starting time* of the task,
- the *deadline*, which is the time by which the task must finish, and
- the *periodicity* of the task, which is the way in which an instance of a task may be requested to be started at given time intervals.

Child tasks are not necessarily started as soon as the request for them is accepted by the scheduler, in that the programmer may wish to start a task at some time in the future. This is achieved by specifying an earliest start time in the scheduling request, as in

```
sched Radar.Scanner() at 10*ticks.per.sec;
```

where “ticks.per.sec” is intended to be a number which is the number of system ticks that occur in one second. In fact, the task may not be started at exactly the time specified, as the scheduler may decide that the task should start later so that other tasks may also execute that require the same resources or are more urgent and require the processing time. Hence, it is only guaranteed that the child task will not execute before the start time specified. The expression that the user gives following the keyword “at” is and used as the time when the child task will start. If a number of instances are requested (via periodicity), then the start time given is the start time requested for the first instance. The time given is an interval in system ticks from the current time. If no start time is specified, then it is assumed that the task start time should be the current time.

If it is vital that a task be completed by a certain time, then the programmer should specify a deadline. The scheduler then uses its past knowledge of the execution time and resource requirements of this task in a manner which is outlined in Section 3 to try to decide which is the highest numbered version that can be executed between the start time and the deadline. A deadline is specified in the manner illustrated by the following example:

```
call (mailaddr CanDoMBX) ReadSign() in 3000;
```

which specifies that the “ReadSign” task must execute before 3000 system ticks are up. The expression following the keyword “in” represents the time allowed for the execution of that task. Note that it is, like the start time, an interval in system ticks, but it is measured from the start time of the task (or from the current time if no start time is specified). In the case of a periodically scheduled task, the deadline is taken to be relative to the start time for each instance, rather than from the start time specified for the first instance.

As another example, consider

```
call (mailaddr TurnedMBX) TurnMissile#2(HORIZ,15) at 3000 in 1000;
```

which indicates that the “TurnMissile” task is to be executed at least 3000 ticks from now, and before 4000 ticks from now.

**Periodically scheduled tasks**

Some tasks are of a nature that they perform some processing repeatedly, such as monitoring a gauge. This is most effectively achieved by starting some task to do this processing once every so often. Periodically scheduling a task is indicated as shown in the example

```
asyched GaugeReader(THERMOMETER) every 1000;
```

in which the parent task schedules a “GaugeReader” task every 1000 system ticks. The parent does not suspend waiting to find out if the scheduling was successful. The expression following “every” specifies the interval in system ticks in between the start times of consecutive instantiations of the task.

It is also possible to include a stopping constraint, to indicate that only a fixed number of periodic instantiations are to be invoked; this is achieved with an “until” expression. For example, the scheduling request

```
sched T1() at 100 every 10 until 133;
```

tries to schedule instances of “T1” at times 100, 110, 120 and 130 ticks. The expression given represents the an interval measured from the time of the scheduling call in system ticks. During this interval from the start time to this stopping time, the scheduler will continue to try to schedule new instances of the task. Note that an until expression is only syntactically correct if it is preceded by an every expression.

The task version for a periodically scheduled task can change at each instantiation. Hence if the system is heavily loaded, the the task instance may be a lower version if that is all that can be achieved, but a higher version may be used if the system is less heavily loaded.

**Missed deadlines**

If a task has not finished execution by the time its deadline arrives, then it is said to have missed its deadline. This is something of a disaster, because it means that the scheduler has not accurately estimated the execution time for the scheduled version of the task. Reflecting its serious nature, this situation is considered a run-time error and a “missed deadline” exception is raised for that task. This exception can be handled using the facilities described in Section 2.4 below.

**Non-terminating tasks**

The notion of being able to predict task execution times from past behaviour is based on the assumption that tasks will always have finite execution times. Thus, non-terminating tasks are to be avoided.
In order to assist the programmer to avoid non-terminating tasks, there is no goto statement in GARTL and all loop statements must have a specified upper limit on the number of iterations. The GARTL loop statements, of which there are three kinds, are thus all preceded by "upto (limit)". A "limit exceeded" exception will be raised if the specified limit, which must be a statically-evaluable constant, is exceeded at run-time.

Function definitions in GARTL require the specification of the maximum number of possible instantiations of that function by a single task. Once again, this maximum number must be statically-evaluable, and the "limit exceeded" exception is raised if its value is exceeded during execution.

2.4 Exceptions and exception handlers

Exceptions occur at unpredictable times. They may occur because deadlines are missed, resources are discovered to be missing, or due to the occurrence of other user-specified situations. Exceptions are always raised on behalf of some task, meaning that the task is somehow responsible for the exception. Every task can choose the exceptions that it will catch; however, if an exception is raised for a task that is not handled in the exception handler section of the task (see below), the default handler for that exception is invoked. If there is no user-specified default handler, then the default action is to terminate execution of the entire program.

User-specified exceptions can be declared in the simple manner illustrated by the following example:

```c
exception my_exception;
```

In this case, no default exception handler is given (and so the exception must either be handled in the task in which it is raised or program execution will be terminated). Specifying a default exception handler for a user-defined or predefined exception is a matter of giving a statement after the declaration of the exception name, as illustrated in Figure 13. Figure 13(a) defines a default exception handler for the user-defined exception "fatal_exception" which is being introduced, and Figure 13(b) defines a default exception handler for the predefined exception "missed.deadline";

```c
eception fatal_exception
    exit(FATAL_EXCEPTION_ERROR);
    /* Cause termination of the program. */

(a)

exception missed.deadline
    cleanup(); /* Perform some cleanup actions. */
    return; /* Cause the task responsible to exit. */

(b)
```

Figure 13. Specifying a default exception handler.

The way in which a programmer can specify that a task is to handle a particular exception is to include an exception handler section in the main body of the task. The form of this section is illustrated by the example in Figure 14. In this case, if a "missed.deadline" exception is raised for an instance of "T1", then "T1" will stop whatever it was doing, execute the "cleanup" routine, then exit the entire program. If a "my_exception" exception is raised for "T1", it stops what it was doing, schedules the "my_handler.task" task with parameters 1, 2, and 3, and then exits. If a "fatal_exception" exception is raised for "T1", then "T1" exits, but the execution of the rest of the program continues.

```c

3. Implementation

There are two major components in the implementation of the GARTL language described in the previous section:
- a compiler for translating GARTL to C, and
- an operating system, called GARTOS, to support real-time applications.

These are both components of GARTEN, an environment for the development of real-time software, which will also include facilities for testing and debugging hard real-time systems. At this stage, GARTEN is reasonably primitive, but we plan to improve the integration of the various components along the lines of multiple window, multiple view programming environments such as MultiView[1], also being developed at the University of Adelaide.

As mentioned above, the current GARTL implementation compiles GARTL source programs to C, which is then compiled in the standard way. The GARTL-to-C compiler has been generated using the GNU BISON[4] compiler generation system. The generated C code assumes a high level of operating system support for mailboxes, message-passing, scheduler interaction, and so on.

These facilities are provided by the GARTOS operating system in the form of a number of C-callable routines. Since the GARTOS operating system is not the focus of this paper, it will not be described fully here; nevertheless, some of its more important aspects for the implementation of GARTL will be highlighted.

A particularly important aspect of the GARTOS operating system is the ability of the scheduler to predict the computation time for a task. For each task, GARTOS sets up a database of m dimensions, where m is equal to the number of monitored variables of the task plus one. The structure of this database is illustrated in Figure 15, which shows that there are two monitored variables for the task "task4". Thus, the database for this task is three-dimensional, being addressed by various equivalence classes for the two variables and by version number.

The database for a task is used to store information about computation times for the task. A cell in the database corresponds to a given version of the task and a given set of input value ranges classified by the monitored variables. In each cell, statistical data on computation time for the task, such as the mean, variance, the maximum, and minimum, are recorded. Such data is collected whenever the task is executed. In the case that the variance and/or the difference between the average and maximum/minimum is too large, the GARTOS operating system gives a suitable warning message to the
programmer, indicating the possible misidentification of the monitored variables and/or the misclassification of their ranks.

At run time, when a scheduling request for a task is received, its database is searched and the maximum likely computation time for the indicated values of the monitored variables is computed. The database has been designed in such a way that this run-time search and computation can be carried out efficiently. The details of the structure and operations of the database will be reported elsewhere.

It is clear that the scheduler plays a critical role in any real-time system. It is the scheduler which makes the decision as to which task to run at what time, on what processor, and so on. The scheduler must be very fast and reliable. In implementing GARTEN, we have adopted the idea used in the Spring project[13] that the scheduler and many other important operating systems functions should be implemented in a separate, dedicated system processor. In our present implementation, we employ only two processors: one for system functions and one for the real-time application. However, we hope to be able to further improve the performance of real-time systems by using more extensive multiprocessors, such as the Leopard multiprocessor workstation[2], which is also under development at the University of Adelaide.

The main advantage of removing the scheduler’s load from the processor used by the application is to improve the predictability of the execution time of application tasks. We believe that this approach is justified in view of the current low cost of microprocessors.

4. Conclusions and Future Work

The GARTL language described in this paper has been designed to support the multi-version computation approach to programming hard real-time systems described in Section 1. Support for this paradigm has many advantages in terms of language features, both with regard to the specification of the tasks to be executed, for interaction with those tasks and the scheduler, and for the form of scheduling requests. These consequences have been outlined through the description of the appropriate GARTL features. The GARTL language has been specifically designed to provide direct support for the multi-version computation approach to programming hard real-time systems.

The FLEX programming language[6,7,10] supports a similar approach to programming real-time software, which is known as imprecise computation. In FLEX, there is a mechanism which keeps the partial (imprecise) computation results of a task. In the case a task which has to be preempted (due to the expiration of its deadline), these partial results are available to the caller of the task. In GARTL, a different approach is taken so that the partial result mechanism is not necessary. In the case of GARTL, the system scheduler is called when a task is to be invoked. The scheduler decides if this task can be finished by its deadline and if so which of its versions will be executed. Once such a decision has been made, the task is guaranteed to complete by its deadline. No preemption is needed due to the expiration of its deadline. As we mention below, this allows a missed deadline to be detected earlier; hence, the necessary recovery action can be taken earlier by the task’s invoker.

An important aspect of GARTL is the way it permits interaction between the application program and the system scheduler. Many other programming languages do allow an application task to invoke another application task at run-time. However, in those languages, there is no feedback from the system to the invoking task about the timing of invoked task. That is, the invoker only knows that the execution of the invoked task will be logically correct, but has no information on when the execution will be made and if the deadline of that invoked task can be met. As a result, a missed deadline cannot be detected by the invoking block, making it difficult to take necessary recovery actions. This is important for many real-time applications and amounts to a special kind of fault tolerance: a missed deadline fault is being detected at the earliest possible time - the time at which the task is being submitted. Part of our planned future work is to extend GARTL to allow the scheduling of multiple instances of a task on different processors; in this way, some hardware faults may be tolerated.

The operating system support required for a language supporting the multi-version computation approach to real-time systems has been indicated by outlining some aspects of GARTOS, the operating system which supports the GARTL implementation. One aspect of GARTOS which is of particular importance is the way in which it makes use of information about the past execution of the multiple versions of tasks to select the appropriate one to execute within the specified deadline (if this is possible). This approach is necessary because the execution of the multiple versions does lend itself to an a priori analysis which enables us to choose the appropriate version. The information about past execution of a task is stored in a database which tabulates each version’s execution time and other resource needs against attributes of the task on which the resource requirements are thought (by the programmer) to depend.

The present means of predicting the execution time of a task from past execution times under similar circumstances is presently reasonably crude and we are currently investigating other, more sophisticated techniques. These include techniques based on various heuristics.

As mentioned in the previous section, we expect to incorporate the GARTL language into an integrated environment for the development of real-time software. This environment will include various debugging, testing, and editing tools for supporting development of real-time systems. A salient feature of these tools is that they must necessarily be oriented towards the timing characteristics of real-time software. The environment will also contain facilities for browsing the database of information about task execution histories.
References


