THE *STRESS* HARD REAL-TIME SYSTEM SIMULATOR

Mike Richardson

ABSTRACT

STRESS is a simulator and analysis tool for hard-real time systems. It comprises a programming language by which target systems can be specified, and a set of tools which can be applied to the specification. These comprise a schedulability analyser, an execution simulator, and a simulation result display tool. This document provides a user manual for *STRESS*.

1. GETTING STARTED

This section describes the analysis and simulation of a *STRESS* program and the subsequent display of the simulation results.

1.1. The *STRESS* Program

The program which is used in this section is shown below, and is based on example 4-1 in Sha's paper on the priority ceiling protocol [Sha90]. We assume that the program is stored in a file called *example1*. It is shown in figure 1. For clarity, keywords appear in bold typeface.

```
system
    node node_1
        processor proc_1
            order     dma

#include    "pripre.s"
#include    "ceiling.s"

            semaphore s0
            semaphore s1
            semaphore s2

            periodic j0
                period 15 deadline 15 offset 4
                [1,1]  p (s0)
                       [1,1]
                       v (s0)
                [1,1]
            endper
            periodic j1
                period 20 deadline 20 offset 2
                [1,1]  p (s1)
                       [1,1]  p (s2)
                              [1,1]
                              v (s2)
                       [1,1]
                       v (s1)
                [1,1]
            endper
            periodic j2
                period 25 deadline 25 offset 0
                [1,1]  p (s2)
                       [3,3]  p (s1)
                              [1,1]
                              v (s1)
                       [1,1]
                       v (s2)
                [1,1]
            endper
        endpro
    endnod
endsys
```

Figure 1: An Example *STRESS* Program

The **system** comprises a single processing **node** called *node_1*, which in turn contains a single **processor**, called *proc_1*. This contains three binary semaphores *S0* ... *S2* and three periodic tasks, *J0* ... *J2*. The two #include'd files contain, respectively, code for the task scheduler and to implement the priority ceiling protocol.

Each task has a specified period, deadline and offset. Hence, task *J0* will be release at

times 4, 19, 34, ... with respective deadlines 19, 34, 49, ... The line **order dma** specifies that the tasks are priority ordered using the deadline monotonic ordering, so that *J0* has the highest priority and *J2* the lowest (in fact, the scheduler task has an even higher priority as it has a still shorted deadline).

Each tasks executes code which variously consumes processor time, and locks and unlocks semaphores. The notation *[n,m]* indicates that the task should consume between *n* and *m* (inclusive) units (called *ticks*) of processor time. In all cases in this example, *n* and *m* are the same, to specify exact amounts of processor time; otherwise, a value is chosen randomly in the range. The instructions *p(name)* lock the semaphore *name* (perform the Dijkstra P- operation), and *v(name)* unlock the semaphore (the Dijkstra V- operation).

Note that the layout is unimportant, and newlines are ignored, except that lines such as the #include lines, which are processed by the C preprocessor, must stand on their own.

1.2. The *STRESS* Front-End

The easiest way to run *STRESS* is using the mouse-driven front end. This is invoked from a normal shell window using the command */usr/drtee/stress3/bin/Sun/driver* (assuming that you are using a Sun3 with Sun Windows or Mux: see the miscellany section for other machines). Note that the shell window should be left visible, since it will be used for text output from the various *STRESS* tools.

Starting the front end will produce a new window looking like that shown in figure 2.

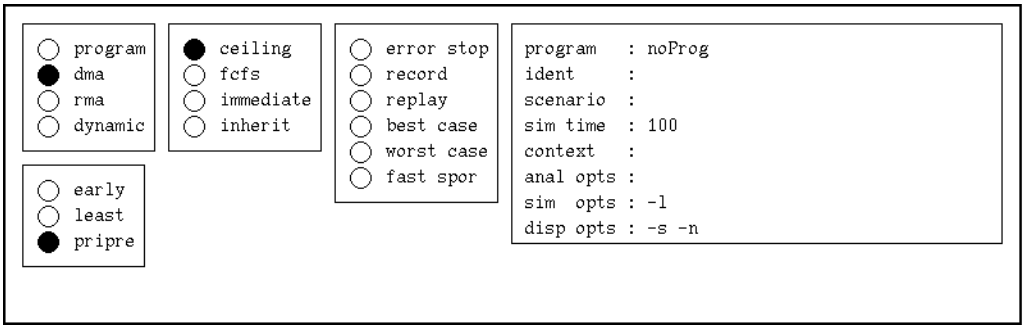

Figure 2: *STRESS* Front-End

The window contains a number of button operated options, some text options, and has an associated pop-up menu. The buttons are operated by pointing at the buttons and clicking the left mouse button. The text options are changed by pointing at the option and clicking the left mouse button; the new value can be entered into the small window which appears, and is terminated by the <return> key.

To set up the front end for the example program, select the **program** option and change it to *example1*. The buttons in the regions starting with **program**, **early** and **ceiling**, which select the default priority ordering, scheduling algorithm and resource management schemes assumed by the schedulability analyser, can be left unchanged, since these are specified within the program. The options in the remaining set of buttons can also be left unchanged.

1.3. The *STRESS* Schedulability Analyser

We will first run the schedulability analyser tool, to establish whether then task set specified in *example1* can be scheduled.

The left hand mouse button should be depressed with the pointer within the front end

window, but away from the options. This causes a menu to appear; the first entry, **analysis**, should be selected by pointing to that entry, and then releasing the mouse button. This starts the analyser.

Output from the analyser appears in the window from which the front end was started. The example program will result in output which includes the output shown in figure 3. Both the simple and the exact deadline monotonic analysis algorithms report that the task set can be feasibly scheduled.

```
Simple Deadline Monotonic  Analysis : proc_1@node_1 : Passes.
Exact Deadline Monotonic  Analysis : proc_1@node_1 : Passes.
analysis returning
```

Figure 3: Part of the Analysis Output

The analyser also generate some information which will be used during the simulation, for instance task priorities and worst-case execution times. This is stored in a file, in this case, *example1__spec*. Generally, file names start with the program name, and have an extension which depends on their contents. The actual analysis results are also stored in a file, here called *example1__anal*.

1.4. The *STRESS* Simulator

The next stage is to run the simulator. By default, the front end will run the simulator for 100 ticks. If this is too small, it can be arranged by selecting the *time* option and setting it to the required time, in the same manner as the *program* option was set to *example1*. For the example, however, 100 is reasonable.

The simulator is run by bringing up the same menu as was used to start the analysis tool, and to select the *simulate* option. The simulator will then run, counting through the ticks in the text window; when complete, it will report that it is done.

The simulation results are written to a file. Like the analysis output, the file name is based on the program file name, and in the example is called *example1__log*. Like all output files from the various tools, it is a text file, however the contents are encoded to keep the file size reasonable small, and it cannot be easily interpreted "by hand". The display tool, which is described next, provides a means of actually viewing the results of the simulation.

1.5. The *STRESS* Display Tool

The display tool is run by selecting the *display* option from the menu. Unlike the analysis and simulation tools, however, the display tool creates a new window, as shown in figure 4.

The display tool is described in more detail later. Briefly, however, it shows the execution of each task, with time running horizontally. Hatched boxes represent task execution, low-level circles task release, and high-level circles task completion. There are several others which do not appear in the example. The tasks are labelled with their name; the task named *idle* is a task which is added by the simulator, and which consumes processor time which would otherwise be idle.

To terminate the display tool, pressing the left-hand mouse button will bring up a menu, and the *quit* option should be selected. This must be confirmed by again clicking the left-hand button; clicking any other button will cancel the request to quit.

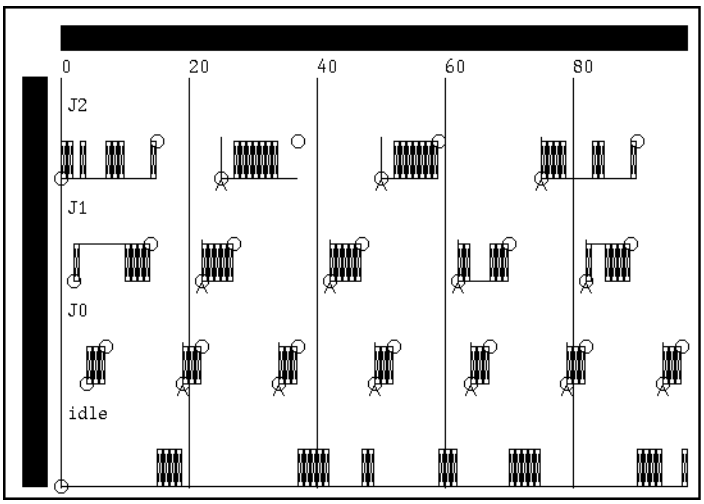

Figure 4: *STRESS* Display Tool

2. STRUCTURE OF *STRESS* SYSTEMS

This section describes the overall structure of systems which can be simulated using *STRESS*. The *STRESS* language which is used to describe and to program such systems is described in the next section, and should be referred to for specific descriptions of functionality.

Figure 5 shows the structure of *STRESS* systems. A key to the components appears in table 1, and the nesting in the diagram reflects the structure of the systems. In the table 1, an asterisk indicates that there may be any number of the component, and a plus sign that there should be one or more of the component; there may be only one network.

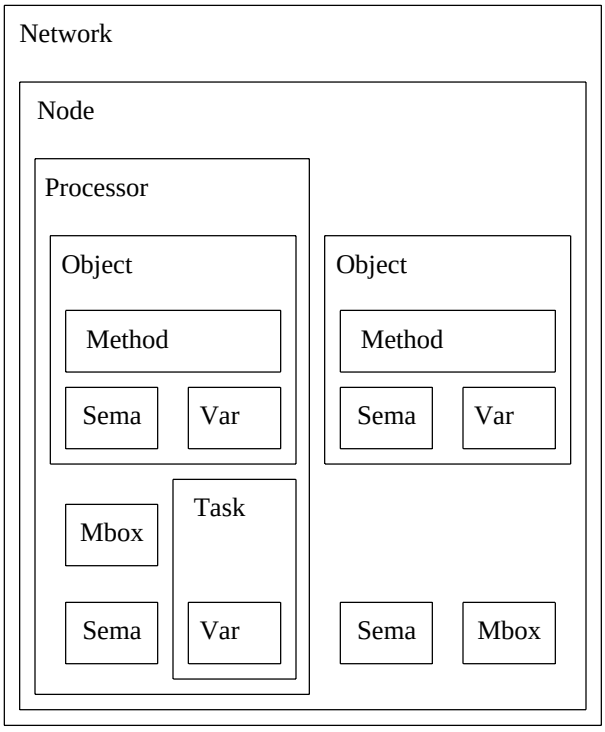

Figure 5: *STRESS* System Structure

A number of example programs appear in this section in order to illustrate the structure of

Network		Network connecting processing nodes
Node	+	Node containing processors
Processor	+	Physical processor
Object	*	Shared-memory communication object
Method	+	Access method to object
Sema	*	Semaphore
Var	*	Variable
Mbox	*	Communications mailbox
Task	+	Thread of execution

Table 1: *STRESS* System Components

STRESS.

2.1. A General Description

Starting from the outermost level, a *STRESS* system is a collection of processing nodes which are fully connected via a point-to-point network. Each processing node contains one or more processors, and may also have objects, semaphores and mailboxes which are global to that node. A wide variety of systems may be constructed using these components. For instance, a single processor system has a single processor on a single node; a set of uniprocessor nodes connected by a ring would be simulated as a set of nodes (each with a single processor) with communication restricted to a ring within the point-to-point network.

At the node level, objects provide a means of shared-memory communication between tasks running on processors on that node. An object cannot be accessed from any node other than the one on which it exists. Similarly, semaphores are the node level can be used to interlock between tasks. Mailboxes provide a message-based means of communication, where messages are sent to, and received from, the mailboxes. A message can be sent to a mailbox at the node level from anywhere on the network, however, messages can only be received by tasks running on the node on which the mailbox exists.

A processor represents a single processing unit, and runs one or more tasks. It may also contain local semaphores, mailboxes and objects. When these exist at the processor level, they can only be accessed by tasks running on that processor.

The task is the basic scheduling unit. At the lowest level, all tasks have a priority, and, on each processor, the simulator runs the highest priority task which is free to run. However, *STRESS* tasks may manipulate task priorities, so that it is possible to write schedulers and resource managers in the *STRESS* language itself.

2.2. The Network and Mailboxes

There are a wide variety of network structures which appear in real systems; for example rings, stars, busses, and partially connected point-to-point. Various protocols may be used on top of these; slotted and token rings, and contention sensed and time multiplexed busses.

In order that *STRESS* imposes as few limitations as possible, it allows total point-to-point connectivity between nodes. Clearly, any network topology can be simulated as a restriction on total connectivity, and by writing *STRESS* programs which use no more than the required connectivity. Hence, a ring is simulated by restricting network communication to use a single ring of communication links.

Actual communication over the network is represented by messages. A message is sent by

a task to a mailbox; and a task may receive a message from a mailbox. There are two restrictions; firstly, a message which is sent over the network (ie., from one node to another) may only be sent to a mailbox at the node level; and secondly, a task may only receive from a node level mailbox on the same node as it is executing one.

Mailboxes have an associated size, which by default is one, but which can be explicitly declared in the *STRESS* program. The size represents the number of messages which may be stored in the mailbox at any given time. If a message arrives at a mailbox when it is full, then the oldest message is silently discarded. Messages reception is handled on a first-in first-out basis. A message contains one or more value, set by the sender and received by the receiver. Note the number of messages which a mailbox can hold is **not** affected by the size of the messages.

A simple example program is shown in figure 6. The system comprises two nodes, each with two processors. On the first, a task periodically sends a message to a mailbox on the second node. There, the message arrival triggers the execution of a task, which receives the message.

```
system
    node node_1
        processor proc_1
            periodic task_1
                period   10
                deadline 10
                send 1 to node_2.mbox_2 delay [4,8]
            endper
        endpro
    endnod
    node node_2
        processor proc_2
            mailbox  box_2
            sporadic task_2
                arrival  4
                deadline 4
                trigger  mbox_2
                variable var_2
                recv mbox_2 to var_2 expiry 3
            endspo
        endpro
    endnod
endsys
```

Figure 6: Message Communication

When a message is transmitted, a communication delay (in ticks) is specified, to represent the time taken to cross the network. This is the significance of "delay [4,8]" in the **send** command. A delay is chosen randomly (with uniform distribution) in the interval [4,8] inclusive, and the message will not arrive at the mailbox for that time. There is a minimum delay time of one tick, except in one special case which is noted in the next section. Also, a timeout may be associated with a reception request, as by "expiry 3" in the example. If no message has been received within this time, then the task will be killed.

2.3. Nodes and Processors

Closely coupled multi-processors generally contain processors which share a common data bus, giving them access to global as well as local memory. This is simulated in *STRESS* using nodes which contain processors plus objects, semaphores and mailboxes.

At the node level, objects, which are described in detail later, provide a means of simulating the global memory. At this level, and object may be accessed from any task on any processor on the node. Since an object can contain local state information, data may be passed between tasks via the object.

As noted above, mailboxes at the node level can be used for communication between nodes, but *STRESS* does not prevent their use, if desired, for communication within the node. However, a mailbox which is located on a processor can only be accessed from that processor. The exception to the minimum message delay time of one occurs when a message is sent to a mailbox located on a processor (and hence is sent by a task on that processor); in this case, the delay time may be zero.

In *STRESS*, the use of multiple processors on a node is useful even when the real system being simulated has only a single processor on a node. For instance, if the real processors communicate over a token ring, where the ring interface is provided by an intelligent peripheral device, then that device can be simulated as a *STRESS* processor.

2.4. Tasks

A task is the basic *STRESS* scheduling unit. Tasks fall into three basic types; periodic, aperiodic and sporadic. Associated with each of these are various characteristics, shown in table 2.

periodic	period	Interval between successive releases
	offset	Time of first release
	deadline	Execution deadline from release
	priority	Basic execution priority
aperiodic	deadline	Execution deadline from release
	offset	Time before first possible release
	release	Release condition
	priority	Basic execution priority
sporadic	arrival	Minimum inter-arrival time
	deadline	Execution deadline from release
	offset	Time before earliest possible release
	release	Release condition
	priority	Basic execution priority

Table 2: Task Characteristics

There are actually two priority values associated with each task, the base priority **baspri**, and the effective priority **effpri**. The base priority can be specified in the *STRESS* program, or can be determined by the schedulability analyser, depending on the scheduling algorithm to be used. However, when a simulation is run, tasks are scheduled according to the effective priority; the task with the highest effective priority which is free to run is executed. The way in which the base priorities of tasks are manipulated to give the effective priorities is dependent on the scheduling algorithm. For instance, using simple priority inheritance, the effective priority of a task is equal to its base priority, except when it is blocking a task with a higher base priority, in which case the effective priority becomes the base priority of the blocked task. See the section on scheduling for more

details.

Note that, at the level of the *STRESS* simulator, all scheduling is performed locally to each processor, and there is no global scheduling.

Each task has associated with it code which it executes, and possibly some local variables. The value of any local variables is preserved between successive executions of the task. The task may also have exception handlers, which are invoked under certain conditions, such as deadline overrun, or message reception expiry.

2.5. Semaphores

Semaphores provide a basic method of implementing mutual exclusion between tasks on the same processor or node. Semaphores located on a node can be accessed from any processor on the node; a semaphore on a processor can only be accessed from that processor.

The basic mechanism is that of the Dijkstra semaphore, except that they may be initialised so that up to *n* tasks may lock the semaphore at any one time. If a semaphore is fully locked when a further task attempts to lock it, then that task becomes blocked on the semaphore. When a task releases a semaphore, **all** tasks blocked on that semaphore are released, and **all** will retry the locking request. This allows arbitrary rescheduling to be performed when a semaphore is released.

It is also possible to explicitly block a task on a semaphore, even if the task is not requesting the semaphore, or the semaphore is not fully locked. This is needed in order to implement resource control schemes such as the *priority ceiling protocol*.

2.6. Objects and Methods

Objects provide shared memory communication between tasks within a single node. An object cannot be accessed from a task running on a different node. Objects are accessed via methods, and can contain local variables and local semaphores.

Objects may be declared at the node level or at the processor level, the only difference being the scope. An object which is declared at the node level can be accessed from any task on any processor on that node; an object declared at the processor level can only be accessed from tasks on that processor.

A method invocation is effectively a function or subroutine call; it may be passed arguments, and may return a result. The method contains code in exactly the same manner a a task, and can access its arguments, the local variables and local semaphores. A method of an object at the node level can also access the mailboxes and semaphores declared on that node; a method of an object declared at the processor level can, in addition, access the mailboxes and semaphores declared on that processor.

Methods can be invoked from one-another, with the restriction that recursion amongst methods is not allowed. This is necessary in order that worst-case execution times can always be bounded. The scoping rules are the same as for access to mailboxes and semaphores; a method of an object at node level can access any method of any other object on that node, while a method of an object at processor level can also access methods of objects on that processor.

3. THE *STRESS* PROGRAMMING LANGUAGE

Systems to be simulated are specified using a special purpose programming language. This has two purposes; to describe the structure of the system to be simulated, and to specify the programs which the tasks execute.

3.1. Processors

At the *STRESS* language level, processors exist only as entities on which tasks, etc., may be placed. However, a list of all tasks on the processor may be obtained via the **tasklist** field, using the **of** operator; this is described in more detail in the section on expressions, but its use is shown in figure 7; the important part is "tasklist **of** proc_1". Similarly, a list of all semaphores declared at the processor level can be obtained through the **semalist** field.

```
processor proc_1
    semaphore s0
    semaphore s1
    periodic task_1
        ....
    endper
    periodic task_2
        period   10000
        deadline 1
        variable tptr
        variable sptr
        for tptr in tasklist of proc_1 max 32
            output effpri of tptr
        for sptr in semalist of proc_1 max 32
            output ceiling of sptr
endpro
```

Figure 7: Access to Processor Fields

3.2. Tasks

A task is specified by its characteristics (for instance, whether it is *periodic*, *aperiodic* or *sporadic*), and by the code which it will execute. A simple example is shown in figure 8.

```
processor proc_1
    semaphore s0
    semaphore s1
    periodic task_1
        ....
    endper
    periodic task_2
        period   10000
        deadline 1
        variable tptr
        variable sptr
        for tptr in tasklist of proc_1 max 32
            output effpri of tptr
        for sptr in semalist of proc_1 max 32
            output ceiling of sptr
endpro
```

Figure 8: An Simple Task

This is about the simplest possible task. It is periodic (indicated by the keywords **periodic** and **endper**, and has the name *task1*. It has a period of 10 ticks, and a deadline of 5 ticks. The statement **[2,2]** is the one and only line of code, and states that, when it executes, the task should consume two processor ticks.

Tasks can have local variables; these can be scalar values or arrays, and can contain either integers or references to tasks, processors, etc. There are also constructs for looping (the **loop** command) and for conditional execution (**if** ... **then** ... **else**. The task in figure 9 is again periodic. Each time it executes, it increments the variable *count* by one; when *count* reaches 10, it is reset to zero. Note that the values stored in local variables are preserved between successive executions of tasks. The initialisation of *count* to zero occurs just once, before the task ever executes.

```
periodic task1
    period   10
    deadline 5
    variable count = 0

    count := count + 1
    if count = 10 then count := 0
endper
```

Figure 9: Another Simple Task

Note that **:=** is used for assignment, and for equality (unlike C).

As noted above, tasks may be periodic, aperiodic or sporadic. Periodic tasks are embedded in the **periodic** and **endper** keywords, aperiodic tasks in **aperiodic** and **endape**, and sporadic tasks in **sporadic** and **endspo**. Depending on the task type, various characteristics may be set, some of which are optional. These are listed in the table below. The first column contains the associated keyword, the second indicates to which task types it applies (with P for periodic, etc.), the third gives the type of value associated with the keyword, and the fourth the meaning. An asterisk in the second column indicates that it is always required with the types of task to which it applies. Lastly, the characteristics should appear in the order given in the table (ie., **deadline** always appears before **offset**).

period	P*	integer	Period of task
arrival	S*	integer	Minimum inter-arrival time
chance	AS	integer	Chance of being released
trigger	AS	mailbox name	Release on message arrival
event†	AS	event name	Release on occurance of event
deadline	PAS*	integer	Execution deadline (from release time)
offset	PAS	integer	Release offset
priority	PAS	integer	Base priority of task
hidden	PAS	(none)	May not appear on display tool

Table 2: Task Characteristics

Periodic tasks are released at intervals specified by **period**. Strictly, the release times are at (offset), (offset+period), (offset+2*period), etc., with the offset defaulting to zero if not specified.

Aperiodic tasks can be released in one of three ways, (a) by chance; for example, **chance**

---

† For aperiodic and sporadic tasks, one of **chance**, **trigger** or **event** must be specified.

10 specifies that there is a one-in-ten chance of release at a given tick, (b) by message; hence, **trigger** *box77* indicates that the task should be released when a message arrives in the mailbox *box77* and (c) by event; **event** *startit* causes the task to be released whenever the event *startit* occurs (events are explained later). An offset is interpreted to mean that the task will definitely not execute until the offset time into the simulation.

Sporadic tasks are exactly as aperiodic tasks, except that **arrival** specifies a minimum time (the *minimum inter-arrival* time) between releases. When simulating a task set, this actually only applies to **chance** releases, and not to **trigger** or **event**; however, the schedulability analysis tool will assume that the minimum time always holds.

The **deadline**, **priority** and **hidden** options apply applied to all task types. **deadline** specifies the execution deadline of the task, starting from the time at which it is released. **priority** allows specific priorities to be assigned to tasks, where lower numbers represent higher priorities; however, the analysis tool can specify its own priorities, based on its evaluation of the task set. Finally, the **hidden** option (which does not have any associated value) can be used in order to prevent the task appearing when the display tool is used.

Following the specification of task characteristics, local variables can be declared. There are four variants on this, as shown in figure 10.

```
variable var1
variable var2 = 0
variable var3[10]
variable var4[10] = 0
```

Figure 10: Variable Declarations

Here, *var1* is a scalar variable whose initial value is undefined; *var2* is a scalar which is initialised to zero; *var3* is an array of 10 elements (indexed 0,1,...9), all of which are initially undefined; and *var4* is a similar array with its elements all initialised to zero. Note that initialisation is restricted to integer values. Also, as mentioned above, the variables are initialised once only when the simulation is started, and not each time the task is released.

A number of values associated with each task can be accessed by *STRESS* programs. **baspri** and **effpri**, mentioned earlier, are the base and effective (scheduling) priorities of the task; **deadline** is the absolute value of the next task deadline; **wcet** is the worst case execution time; **left** is the maximum execution time left (this is set to **wcet** when the task is released, and is decremented as the task executes); and **locking** is a list of all semaphores locked by the task. The details of access to these fields is described in the next section, and uses the **of** operator.

3.3. Variables, Expressions and Assignment

STRESS provides for variables, and the assignment of the values of expressions to variables. Expressions may also be used in various other contexts. Expressions are generally build up from the operators, variables and integer numbers (floating point is not supported). However, they may also contain method invocations, references to tasks, processors, etc., and to state information from tasks, etc.

The available operators are listed below. Their precedence (binding power) increases down the list, except that in the groups (equals, not equals), (less than, greater than, less than or equal, greater than or equal), (addition, subtraction) and (multiple, divide, remainder) the operators have the same precedence and bind from left to right.

||	logical or
&&	logical and
|	bitwise or
&	bitwise and
^	bitwise exclusive or
=	equals
!=	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
+	addition
-	subtraction
*****	multiple
/	divide
%	remainder
of	field of task, etc.

Table 3: Operators

Logical or and logical and are conditional. (*a*||*b*) means "if *a* then true else *b*", and (*a*&&*b*) means "if *a* then *b* else false". Hence, the second operand is only evaluated conditionally on the value of the first. Brackets can be used to force binding in the normal way. Expressions may also include method invocations. Objects and methods are described later, however it is worth noting here that a method invocation can be used in an expression with the same effect as a function call in other languages.

Figure 11 shows illustrates some expressions and assignments.

```
sum     := a + b + c
average := sum / 3
tptr    := mytask
mypri   := effpri of tptr
laxity  := deadline of mytask - tick - left of mytask
taskl   := tasklist of proc3
```

Figure 11: Some Expressions and Assignments

The first two assignments, to *sum* and *average*, simply calculate the sum and (integer) average of the variables *a*, *b* and *cv*. In the third, *mytask* is a special variable which points to the task executing the code, and after the assignment, the variable *tptr* also points to that task. The fourth assignment then assigns the effective priority of the task to the variable *mypri*. The first assignment calculates the laxity of the task; *tick* is a special variable which contains the current processor time, and **deadline** and **left** refer respectively to the deadline and execution time remaining of the task. Finally, the fifth assignment sets *taskl* to point to a list of all tasks on the processor named *proc3*.

The fields available using **of** are listed below. The three columns contain the field name, the type of thing to which to applies, and briefly its meaning. The significance of the fields is elaborated upon in the sections on tasks, etc.

baspri	task	Base priority of task
effpri	task	Effective priority of task
deadline	task	Deadline of task
wcet	task	Worst-case execution time of task
left	task	Execution time remaining in task
locking	task	List of semaphores locked by task
tasklist	proc	List of all tasks on processor
semalist	proc	List of all semaphores on processor (not embedded in an object)
cnt	sema	Semaphore free count
locking	sema	List of tasks locking semaphore
waiting	sema	List of tasks blocked on semaphore
ceiling	sema	Semaphore ceiling priority

Table 4: Fields **of** Components

There are a number of special variables, which give access to useful values. These are listed in the table 5.

rand_*n*	Random value in range 0...n-1
tick	Current simulation time
mytask	Refers to the task executing the code
myproc	Refers to the processor running the task
mynode	Refers to the node

Table 5: Special Variables

3.4. Control Constructs

The *STRESS* language provides three control constructs: **if**, **loop** and **for**.

if is used to provide conditional execution, as illustrated in figure 12. This also shows the use of braces **{** ... **}** to group together series of statements. Hence, if *x* is equal to zero, then *x* and *y* are both set to one; if *x* is not equal to zero then it is reset to zero. The **else** part is optional.

```
if x = 0 then
{     y := 1
      x := 1
}
else  x := 0
```

Figure 12: **if** ... **then** ... **else**

The second construct is **loop**, which allows bounded execution of one or more statements. An example is shown in figure 13. When this executes, the variable *tri* is set to the 10th triangle number. The **loop** body is executed while the condition *var* > 0 is true, except that the number of iterations is bounded to 20. Note that the bound must be an integer value; it is required so that the worst-case execution time can be calculated, and so that *STRESS* programs always execute in bounded time. There is no explicit indication of whether the **loop** terminated because of the condition or the bound; this must be tested for in the code. In this example, if the bound was not exceeded than *var* will be equal to zero.

The remaining construct is **for**, which allows lists to be scanned. These originate in expressions such as *tasklist* **of** *myproc* or *waiting* **of** *sema*. The example in figure 14 is the code for a least-laxity scheduler. *task* iterates over each of the tasks on the processor on

```
tri := 0
var := 10
loop var > 0 max 20
{   tri := tri + var * var
    var := var - 1
}
```

Figure 13: **loop**

which the code is running, and the effective priority of each task is set to its laxity.

```
for task in tasklist of myproc max 100
    effpri of task := deadline of task -
                tick -
                left of task
```

Figure 14: **for**

3.5. Semaphores

Three primitive operations are available for use with semaphores; these are the Dijkstra P- and V- operations, plus an explicit blocking command. Note that the P- and V- operations must refer to an explicit semaphore, however the **block** command evaluates an expression which must yield a pointer at a semaphore.

If a P- operation is executed, and the semaphores count is non-zero, then the count is decremented by one, and the task which executed the operation continues; if it is zero, then the task becomes blocked on the semaphore. The C- operation increments the count by one, and releases all tasks which are blocked on the semaphore; these will retry the P- operation on which they blocked, so that one of them will succeed. The use of the **p**(*sema)* and **v**(sema) appear in the example in the "Getting Started" section. **pop** and **vop** are synonyms for **p** and **v**.

The **block** operation is used, for example, in implementing the *priority ceiling* protocol. To implement this, it must be possible to block a task on a semaphore which it the task has not requested, and which may not be locked. Refer to the section on implementing resource management policies for more details.

A number of fields associated with each semaphore can be accessed by *STRESS* programs, using the **of** operator. These are **locking**, which is a list of all tasks currently locking the semaphore; **waiting**, which is a list of all tasks currently blocked on the semaphore; and **ceiling**, which is the priority ceiling of the semaphore (used by the priority ceiling protocol and determined by the analysis tool).

3.6. Mailboxes and Messages

Tasks interact with mailboxes using the **send** and **recv** commands, and also by arranging that aperiodic and sporadic tasks can be triggered by the arrival of a message in a mailbox. A short example is shown in figure 15.

The mailbox *box_1* is declared by the first line. Setting it to two indicates that if can hold two messages; if another message is placed in it when it already contains two messages, then the oldest message is discarded. By default, mailboxes can hold only a single message.

In this example, the sporadic task *counter* is **trigger**ed by the arrival of a message in the mailbox *box_1*. Once executing, it receives the message, storing the value contained in the

```
mailbox box_1 = 2

sporadic counter
        arrival  10
        deadline 5
        trigger  box_1
        variable count = 0
        variable incr

        recv box_c to incr
        count := count + incr
        [1,1]
        if count >= 10 then
        {       send count to proc_2.box_2 delay [3,6]
                count := 0
                [1,1]
        }
endspo
```

Figure 15: Mailboxes

message in the variable *incr*; this value is then added to *count*. Whenever *count* exceeds 10, its value is sent to mailbox *box_2* on processor *proc_2*, and *count* reset to zero. At two points in the code, the task consumes one tick of processor time.

When a message is sent, a delay is specified. This represents the time that the message takes to arrive at the destination mailbox. In the example, a delay will be chosen randomly with uniform distribution from the range 3 to 6 inclusive. Note that the minimum delay is one tick, unless the message is sent to a mailbox on the same processor as the sender, in which case the delay can be zero†.

Message reception can have an optional timeout, such that if the timeout expires, then the task is killed. The timeout is measured in ticks from the moment at which the receive command is executed. For example, a timeout of 10 ticks is given by "**recv** *box_c* **to** *incr* **expiry** 10". In the example, there is little point in adding a timeout, since the task will not execute until there is a message ready.

As well as simple scalar values being sent, arrays of values may also be transmitted. If the **send** command specifies the name of a variable which is an array, rather than specifying some expression, then the array of values is set. Similarly, a variable which is an array can be specified in the **recv** command. The only restriction is that both arrays must have the same number of elements.

3.7. Objects and Methods

In *STRESS*, objects and methods can be used for various purposes. A major use is to provide shared-memory communication between different tasks, including tasks on different processors (but on the same node). They can also be used to provide subroutines and functions.

An object is identified by a name, and can contain local variables, local semaphores, and one or more methods. Methods can have arguments, and can return a result. A simple example is shown in figure 16.

---

† There is an exception to this. If a message is sent to trigger the execution of another task, then there is a one tick delay before the task is triggered, even if it is on the same processor.

```
object counter
    semaphore lock
    variable  count = 0
    variable  temp
    method incby (inc)
        p (lock)
        count := count + inc
        [2,2]
        v (lock)
    endmet
    method getval ()
        p (lock)
        temp := count
        [1,1]
        v (lock)
        return temp
    endmet
endobj
```

Figure 16: Objects

The object contains two methods, *incby* and *getval*. The first increments the local variable *count* by the value of its argument *inc*; the second returns the value of *count*. Both methods lock the local semaphore *lock* while they are active.

A method is invoked by naming the object and the method, passing any arguments in brackets in the normal way. The results of the invocation is the return value from the method, and is undefined if no **return** command is executed. Invocation is illustrated in figure 17, using the object defined in figure 16.

```
counter::incby (10)
output (counter::getval ())
```

Figure 17: Method Invocation

3.8. Atomic Sections

It is sometimes necessary to write code sections which are executed atomically, that is, without the task being preempted. This can be done in *STRESS* using the **atomic** command. An example of this is shown in figure 18. In this example, the task cannot be preempted while executing the commands in the block of code preceeded by **atomic**.

```
        var1 := 10
        [1,1]
        atomic
        {       [2,2]
                var1 := 20
        }
        [3,3]
        var1 := 30
```

Figure 18: Method Invocation

Normally, a task can be preempted when executing a timing command, or just before an assignment or **output** command. These preemption points are disabled using the **atomic** command. Note that **atomic** does not stop a task being suspended if it becomes blocked on a semaphore or mailbox.

3.9. Output

STRESS programs can write to the logging output file using the **output** command. This takes a list of one or more arguments; the values of the arguments are output separated by spaces. For instance, the program in 19 generates a list of the squares of the integers one through nine..KF

```
idx := 1
loop idx < 10 max 10
{       output ("the square of", idx, "is", idx * idx)
        idx := idx + 1
}
```

Figure 19: The **output** command

The format of the logging file is described in an appendix. The output values can be extracted from the logging file using the *display* tool, executed as a command with the **-o** option:

/usr/drtee/stress3/bin/Sun/display -o *program_ident*

program is the name of the program which contained the **output** commands, and *ident* is the identifier set from the front end (and is null if none was set).

4. *STRESS* SCHEDULERS AND RESOURCE MANAGERS

This section describes scheduling in the *STRESS* system. It first describes the low-level scheduling provided by the actual simulator itself, and the discusses the construction of higher-level schedulers, and their analysis by the schedulability analyser.

4.1. Low-Level Scheduling in *STRESS*

At the lowest level, each processor is scheduled separately, and is not affected by events on other processors, other than through semaphores and mailboxes. Each task has associated with it two priorities, the base priority **baspri**, and the effective priority **effpri**.

The base priority may be set in one of three ways. Firstly, it may be set by use of the **priority** keyword. This sets it to a specific numerical value; priorities should be integers, where smaller numbers represent higher priorities. Secondly, it may be set by running the schedulability analyser; this will assign priorities based on the scheduling algorithm which is to be used. Thirdly, if no explicit priority is given, and the analysis tool is not used, then the base priority is set to the lowest-but-one possible priority.

Note that the simulator inserts an addition task, called *idle*, onto each processor. This has the lowest priority possible, and uses up all processor time which would otherwise be unused.

When the simulator is started, the effective priorities are set to the base priority values. Then, at each tick, the task with the highest effective priority, which is free to run, will be executed. If the effective priorities are left unchanged, the scheduling is effectively preemptive static priority, with semaphores access on a first-come first-served basis. In order to provide more sophisticated scheduling, it is necessary to write a scheduler in the *STRESS* itself. This is dealt with in the next section.

4.2. High-Level Scheduling in *STRESS*

As an example, a scheduler for the earliest-deadline algorithm will be shown. Under this algorithm, the task which is closest to its deadline (and free to run) will be executed. The code for this is shown in figure 20.

```
scheduler early
periodic  early_sched

    period   1
    deadline 0
    offset   0
    priority 0
    hidden
    variable task

    for task in tasklist of myproc max 999999
        if task != mytask
            then effpri of task := deadline of task

endper
```

Figure 20: Earliest Deadline Scheduler

The scheduler is implemented as a task, named *early_sched*, which runs on every tick, but does not consume any processor time. When it runs, it can adjust the effective priority values, and the terminate. This leaves the low-level scheduler free to run whichever other task is free to run and has the highest effective priority.

It is necessary to arrange that the scheduler task executes before any other task, in order

that it can set the effective priorities correctly. This is managed by setting to priority to zero. Provided that other tasks on the processor are not given priorities which are negative, the scheduler will have the highest priority, and will execute first.

When the scheduler task runs, it scans the list of tasks on the processor. *myproc* is a special variable which refers to the processor on which the task is running; this avoids needed to know the processor name. For each task, other than the scheduler task itself (*mytask* being another special variable which refers to the task executing the code), the effective priority is set to the next task deadline. The closer the deadline is to the current time, the smaller this value will be, and hence the higher priority. Note that it is not necessary to check whether the task is free to run or not, as the low-level scheduler will always ignore tasks which are not free to run.

Finally, since we probably do not want to see the scheduler task when a simulation run is displayed by the display tool, the task is marked **hidden**. The line "scheduler early" is used by the analysis tool, and is described in a later section.

As a second example, figure 21 shows a least laxity scheduler, where a tasks priority is set to the difference between the time interval until the next deadline, and the remaining computation time needed by the task. This works in a similar manner to the earlist deadline scheduler. *tick* is a special variable which gives the current simulation time.

```
scheduler least
periodic  least_sched

    period     1
    deadline 0
    offset     0
    priority 0
    hidden
    variable task

    for task in tasklist of myproc max 999999
        if task != mytask
            then effpri of task := deadline of task -
                                   left     of task - tick

endper
```

Figure 21: Least Laxity Scheduler

4.3. Resource Management in *STRESS*

This section describes how resource management algorithms can be implemented in *STRESS*. Like the scheduling algorithms, the are implemented using *STRESS* code. Two examples are shown, for simple priority inheritance, and for the priority ceiling protocol.

Figure 22 shows the code for simple priority inheritance. Essentially, it redefines **p** and **v** to invoke subroutines *respop* and *resvop* respectively, before actually performing the semaphore operation (**pop** and **vop** are synonyms for **p** and **v** respectively). Note that the subroutine call and the actual semaphore operation are place in an **atomic** block, so that they execute together. The subroutines are implemented as methods of the object **resman**.

The *respop* routine checks whether the semaphore is locked; if it is then *cnt* **of** *sema* will be zero. If it is locked, then the list of tasks locking the semaphore is scanned, and, if any has a lower effective priority Note the comparison, since lower numbers correspond to higher priorities.

```
#define p(sema) atomic { resman::respop(sema) pop(sema) }
#define v(sema) atomic { resman::resvop(sema) vop(sema) }

resource inherit
object   resman

    variable locker
    variable waiter
    variable myused

    method respop (sema)

        if cnt of sema = 0 then
            for locker in locking of sema max 999999
                if effpri of locker > effpri of mytask then
                    effpri of locker := effpri of mytask
    endmet

    method resvop (sema)

        effpri of mytask := baspri of mytask

        for myused in locking of mytask max 999999
            if myused != sema then
                for waiter in waiting of sema max 999999
                    if effpri of waiter < effpri of mytask then
                        effpri of mytask := effpri of waiter
    endmet

endobj
```

Figure 22: Simple Priority Inheritance

than the calling task (which is about to become blocked) then its effective priority is increased to that of the caller. Hence, if the caller is blocked by a lower priority task, than the effective priority of that task will be raised to that of the caller. Finally, on return from the routine, the actual semaphore operation is invoked.

The corresponding *resvop* routine initially resets the callers effective priority to its base priority. In then scans all semaphores locked by the caller (other than the semaphore which is being released), and each of the tasks blocked on such semaphores. The callers effective priority is raised if any blocked task has a higher effective priority.

The priority ceiling protocol example is shown in figure 23. This functions in a similar manner to the simple priority inheritance code, although the implementation is more complicated. The two main point to note is that the *respop* routine contains a retry, and that the **block** instruction is used.

The retry loop is required since all tasks blocked on a semaphore are released by the *STRESS* simulator when a task which was locking the semaphore releases it. Hence, it cannot be assumed that, simply because a task has been released after blocking on a semaphore, that it will be able to lock that semaphore. The **block** instruction is used since the priority inheritance protocol requires that sometimes a task may be blocked on a semaphore which it is not requesting, and which may not be locked.

```
#define p(sema) atomic { resman::respop(sema) pop(sema) }
#define v(sema) atomic { resman::resvop(sema) vop(sema) }
resource ceiling
object   resman
  variable  locker    variable  waiter    variable  myused
  variable  user      variable  star      variable  blocker
  variable  retry
  semaphore dummy
  method notuse (sema, task)
    for user in locking of sema max 999999
        if user = task then
            return    0
    return    1
  endmet
  method respop (sema)
    retry := 1
    loop retry max 999999
    { blocker := dummy
      star    := 0
      for sema in semalist of myproc max 999999
        if (cnt of sema = 0) && resman::notuse (sema, mytask) then
          if (star = 0) || (ceiling of sema < ceiling of star) then
            star := sema
      if cnt of sema = 0
        then blocker := sema
        else if (star != 0) && (effpri of mytask >= ceiling of star)
          then blocker := star
      if blocker != dummy then
      { for user in locking of blocker max 999999
          if effpri of user > effpri of mytask
            then effpri of user := effpri of mytask
        block    blocker
      }
      else retry    := 0
    }
  endmet
  method resvop (sema)
    effpri of mytask := baspri of mytask
    for myused in locking of mytask max 999999
      if myused != sema then
        for waiter in waiting of sema max 999999
          if effpri of waiter < effpri of mytask then
            effpri of mytask := effpri of waiter
  endmet
endobj
```

Figure 23: Priority Ceiling Protocol

4.4. Schedulability Analysis

The *STRESS* schedulability analyser takes a *STRESS* program, and reports on whether it can be feasibly scheduled. Some restrictions must be born in mind.

(a) The analyser obviously cannot determine scheduling and resource management algorithms from the *STRESS* program. For this reason, it must be explicitly informed which are being used.

(b) A limited number of algorithms, and combinations thereof are available.

By default, the scheduler assumes deadline-monotonic priority ordering, the priority ceiling protocol for semaphore access, and priority preemptive scheduling. However, these can be changed in a number of ways:

(a) The three menus on the front end (those header *program*, *early* and *ceiling* can be used. These actually invoke the schedulability analyser with appropriate arguments. See the manual page for the analyser for further details.

(b) The keywords **order**, **scheduler** and **resource** can be used within the *STRESS* program. The keyword is followed by the name of the algorithm required, as in table 6. These can be used at various levels in the program; at the **system** ... **endsys** level, they affect all processors; at the **node** ... **endnod** level, they affect all processors on the node; and at the **processor** ... **endpro** level, the affect just that processor.

order	program	Use **priority** in program
	dma	Deadline-monotonic ordering
	rma	Rate-monotonic ordering
	dynamic	A dynamic priority algorithm is used
scheduler	pripre	Priority preemptive
	early	Earliest deadline
	least	Least Laxity
resource	fcfs	First-come first-served
	inherit	Simple priority inheritance
	immediate	Priority ceiling inheritance
	ceiling	Priority ceiling protocol

Table 6: Algorithm Selection

Figure 24 shows a simple example. At the system level, the algorithms selected are rate-monotonic ordering, priority preemptive scheduling and first-come first-served resource management. These all apply to *node_1*; however, in *node_2*, deadline-monotonic replaces deadline-monotonic. Finally, just on processor *proc_1*, simple priority inheritance is used rather than first-come first-server.

```
system
        order rma
        scheduler pripre
        resource fcfs
        node node_1
                ...
        endnod
        node node_2
                order dma
                ...
                processor proc_1
                        resource inherit
                endpro
        endnod
endsys
```

Figure 24: Algorithm Selection

Note that, because the analyser is told which algorithm is to be used, and cannot deduce it from the code, there are no checks that the algorithm specified matches that which is actually used.

5. THE DISPLAY TOOL

This section describes the capabilities of the display tool in more detail. As an example, the display from the "Getting Started" section is used. This is shown again in figure 25.

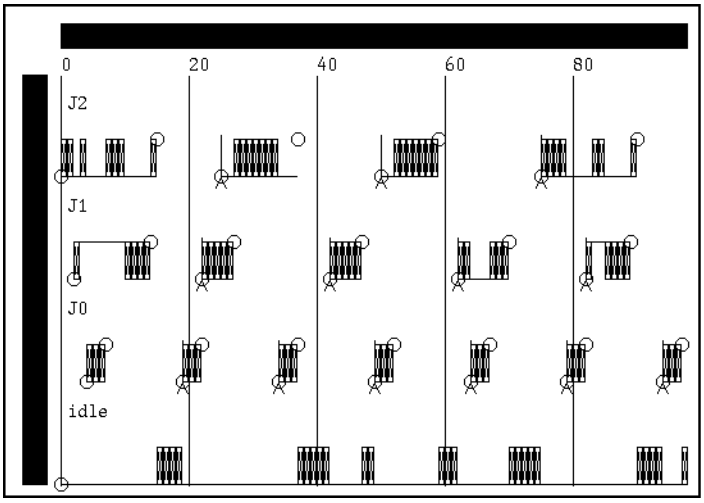

Figure 25: *STRESS* Display Tool

5.1. The Display

The dark bars at the top and left are sliders used to control the range respectively of time and tasks that are displayed. In this example, they both encompass the total possible range. They can be adjusted in two ways, under the control of the left hand mouse button. Pointing into the slider near the end of the black part, and pressing the left hand mouse button allows that end to be moved; pointing near the middle of the blak part allows the selected range to be slid back and forth. Hence, to display, say, the first half of the time range (about ticks 0 to 50), point to the right hand end of the time slider and press the left hand mouse button, and then move the pointer to the midpoint of the slider and release the button; the display will be redrawn in the new configuration.

Note that if there are too many tasks to be sensibly displayed, the display tool will limit the number that can appear on the display at any one time. Also, as the width of one tick is reduced, more and more detail is lost. On the other hand, if the tick width becomes sufficiently large, the the number of execution ticks in each "block" is displayed at the end of the block.

The remaining functions are available from a menu which appears when the left hand mouse button is pressed with the pointer outside the sliders. This has a number of entries, which can be selected in the normal way. These are described in the next section.

5.2. Display Markings

The symbols which are used on the display to represent task execution are listed below:

Task Execution

There are two display modes, which affect the way in which an active (executing) task is shown. In the default mode, when a task consumes a tick of processor time, is is shown as a box with a cross in it; this corresponds to the execution of a tick’s worth of a *[n,m]* statement. In the alternative mode, where successive ticks are consumed by the same task, a single elongated box is displayed (without the cross).

Note that there is no indication of task execution which does not consume any processor time.

A line at the bottom level, as for task *J2* at ticks 2, 4-6, ..., indicates that the task has been pre-empted by some other task (in this case, tasks *J1* and *J2* respectively). A line at the top level indicates that a task is blocked awaiting a semaphore, as for task *J1* during ticks 3-9.

Task Release and Termination

These are marked using circles. Release is indicated by a circle at the level of the bottom of the execution boxes, and termination by a circle at the level of the tops of the boxes. If the task fails for some reason, such as a deadline overrun, then the termination circle is filled in.

Task Deadline

A deadline is indicated by a vertical line with a $\wedge$ mark at the bottom. In the example in figure 25, task *J2* has deadlines at times 24, 28, ... Since this task has the same period and deadline, the release and deadline markers are coincident.

Message Traffic

The transmission of a message is indicated by a small outgoing arrow at the top level; the reception of a message is indicated by a small incoming arrow at the bottom level.

5.3. Controlling the Display

The following functions are available from the pop-up menu.

A4

When the display tool starts, it generates a window whose size is appropriate to the simulation requests which are being displayed. This is based on the number of tasks and ticks. However, this is unlikely to be a size which is suitable for generating A4 hardcopy output. Selecting the *A4* option resizes the window so that output can be generated using the *Pic* option.

Options

This option leads to a sub-menu which contains the following items. **All on** enables the display of release and completion, deadline and message marks; **All off** disables the display of these; **Release/done** toggles the display of release and completion marks; **Deadline** toggles deadline marks; **Message** toggles message marks; and **Drawing mode** toggles between execution drawing modes (individual crossed boxes and elongated boxes).

Pic

Hardcopy output can be generated corresponding to the current display. This takes a form that can be processed by the UNIX *pic* and *troff* commands. Selecting the *pic* option will result in a text box appearing. This prompts for the name of an output file; the default is *pic.out* but this can be changed. Pressing return or the left-hand mouse button confirms the file name; and other mouse button press will cancel the *pic* request.

Point

When this option has been selected, the pointer will change to a small arraw-and-circle pointer. If this is moved to a particular position in the display, and the left-hand mouse button clicked, then a window will appear which contains details of all events which occurred at that tick. This window will contain a slider similar to those on the main window which can be used to scroll through the events if there are two many events to fit in a reasonable sized window. The window contains a pop-up menu (on the left-hand mouse button) with a single **quit** entry.

Quit

This option provides the means of terminating the display tool. If must be confirmed by subsequently pressing the left hand mouse button again. Any other mouse button press will cancel the *quit* request.

Times

A specific time range may be selected using this option. A text box appears into which the range should be typed, in the form *n-m*, followed by return, or a futher press of the left-hand mouse button. The display will then be redrawn showing ticks *n* to *m*. Any other mouse button press will cancel the *times* request, and the display will be left unchanged.

Zoom

The *zoom* option also allows a time range to be selected, but using the mouse. Point to one end of the required time range, depress the left-hand mouse button, move the pointer to the other end of the time range, and release the mouse button. As the pointer is moved, a rectangular box will be drawn; the width of this indicates the selected range, but the height is ignored. Note that this option can only be used to reduce the time range, and not to increase it, as with the *times* option. Any other mouse button press will cancel the *zoom* request.

All the menu options other than *point* can be invoked using control key combinations as well as from the menu. The combinations are shown on the menu, and in the table below:

ctrl-A	*A4*
ctrl-P	*Pic*
ctrl-Q	*Quit*
ctrl-T	*Times*
ctrl-Z	*Zoom*

6. OPTIONS

This section describes the command line interface and various options available for the analysis, simulation and display tools. These can be used if the tool is invoked directly from the command line, or via the *anal opts*, *sim opts* and *disp opts* options on the front end. The front end sets some options automatically; these are noted below.

For completeness, this section also describes tool arguments which are not

Numerical options are represented using the # symbol. Hence, **-c***#* represents **-c** immediately followed by a number, for example **-c***4*. Similarly, a string option is represented by, for example, **-C***string*.

6.1. The Analysis Tool

There are two forms of command line:

analysis [-w] [-CDname[=val]] [-CUname] progf analf interf
pri res sched
analysis [-w] [-CDname[=val]] [-CUname] [-Oorder] [-Ssched] [-Mresrc]
progf analf interf

The second form is preferred, the first exists for historical reasons.

-C*string*

Arguments may be specified for the C preprocessor using the **-C** option. For example, **-C***DHARDER=TES* would cause the C preprocessor to define the symbol *HARDER* to **YES** (by passing the argument *-DHARDER=YES* to the preprocessor. Similarly, **-C***UHARDER* would undefine the symbol *HARDER*.

-M*resrc*

Normally, the schedulability analyser will assume that the priority ceiling protocol is being used, unless the *STRESS* program specifies otherwise. However, this default can be changed using the **-M** (resource **M**anager) option; *resrc* can be one of *fcfs*, *inherit*, *immediate* or *ceiling*, exactly as in a *STRESS* program. Note that the **-M** option is still overriden by explicit program commands.

-O*order*

This option selectes the default priority order assumed by the schedulability analyser, as for the **-M** option for the resource management. The available options are *program*, *rma*, *dma* or *dynamic*.

-S*sched*

Normally, the schedulability analyser will assume that the priority preemptive scheduling is being used, unless the *STRESS* program specifies otherwise. However, this default can be changed using the **-S** (**s**cheduler) option; *sched* can be one of *pripre*, *early* or *least*, exactly as in a *STRESS* program. Note that the **-S** option is still overriden by explicit program commands.

-w

If the **-w** option is set, then the anaysis tool waits for a return to be typed before it exits.

progf

The name of the file containing the program to be simulated is given by the **progf** argument.

analf

Analysis output is written to the file specified by the **analf** argument. If this output is not needed, then the file can be set to */dev/null*.

interf

The intermediate output file, which is subsequently used by the simulator, is written to the file specified by the **interf** argument.

pri

When the first form of the command is used, the default priority ordering is specified numerically via the **pri** argument: 0 -> program, 1 -> deadline monotonic, 2 -> rate monotonic and 3 -> dynamic

res

When the first form of the command is used, the default resource management policy is specified numerically via the **res** argument: 0 -> first-come first-served, 1 -> priority inheritance, 2 -> deadline inheritance, 3 -> priority ceiling and 4 -> immediate inheritance

sched

When the first form of the command is used, the default scheduling algorithm cen be specified numerically via the **sched** argument: 1 -> priority pre-emptive, 2 -> earliest deadline and 3 -> least laxity.

6.2. The Simulator

```
sim [-ABdDeLlrRsWw] [-CDname[=val]] [-CUname] [-ctime]
   [-Eeventf] [-ispecf] [-Iident] [-Oorder] [-ttime]
   [-Ssched] [-Mresrc] prog
```

Unless otherwise specified, if the stress program is contained in the file *stress*, then the intermediate specification file (output by the analysis tool) will be *stress__spec*, and any logging output will be written to *stress__log*.

-A

The **-A** (**A**lways) option causes sporadics which are released by chance to be released with the maximum possible frequency. In effect, they become preiodic tasks, where the period is the minimum inter-arrival time. This option can be used when accessing worst-case behaviour of a system

The front end sets this option is *fast spor* is set.

-B

If the **-B** (**B**est) option is selected, then the best-case time is always used whenever a timing statement is executed. Hence, a timing statement *[n,m]* always consumes *n* processor ticks. The **-W** option can be used to select the worst-case time.

The front end sets this option if *best case* is set.

-c*#*

Normally, task context switching takes zero time. However, the **-c#** option can be used to select a non-zero time. The exact behaviour is described in more detail on the *miscellany* section.

The front end sets this option if *context* is set.

-C*string*

Arguments may be specified for the C preprocessor using the **-C** option. For example, **-C***DHARDER=TES* would cause the C preprocessor to define the symbol *HARDER* to **YES** (by passing the argument *-DHARDER=YES* to the preprocessor. Similarly, **-C***UHARDER* would undefine the symbol *HARDER*.

-d

Selecting the **-d** (**d**isplay) option causes the program to be written to the standard output. If the simulator is invoked from the front end, this will be the window from which the front end itself was invoked. In the pregram output, all names will be converted to their full form. For example, a variable *var1* might be expanded to *var1@obj1@node1@<network>*, meaning variable *var1* on object *obj1* located at node *node1* on the *<network>*. This option is primarily meant for debugging purposes and may not behave in exactly this way.

-D

The *dangling-done* problem may be partially resolved using the **-D** (**D**angle) option. This problem is described in the *miscellany* section.

-e

Normally, the simulator continues until the end of the selected simulation time range. However, if the **-e** (**e**rror-stop) option is selected, then it will stop if any task overruns its deadline.

The front end sets this option is *error stop* is selected.

-E*file*

An event file to be used is specified using the **-E** (**E**vent) option. The format of an event file is described in the *miscellany* section.

The front end sets this option if a *scenario* is set.

-i*file*

Normally, the intermediate specification file, which contains information such as best-case and worst-case execution times, and semaphore ceilings, is read from a file whose name is based on the program file and the **-I***ident* option (if any). However, a specific file may be named using the **-I** (**I**ntermediate) option.

-I*idstring*

This option sets the identifier which is used in constructing file names.

The front end sets this option according to the value of *ident*.

-l

Simulation results logging is enabled using the **-l** (**l**ogging) option. Note that this option is initially set by the front end, but can be changed by the user.

-L

This option also enables logging, but only for **output** commands.

-M*resrc*

Normally, the schedulability analyser will assume that the priority ceiling protocol is being used, unless the *STRESS* program specifies otherwise. However, this default can be changed using the **-M** (resource **M**anager) option; *resrc* can be one of *fcfs, inherit, immediate* or *ceiling*, exactly as in a *STRESS* program. Note that the **-M** option is still overriden by explicit program commands.

The front end sets this optionm according to the selected resource management scheme.

-O*order*

This option selectes the default priority order assumed by the schedulability analyser, as for the **-M** option for the resource management. The available options are *program*, *rma*, *dma* or *dynamic*.

The front end sets this option according to the selected priority ordering.

-r

The front end sets this option is *record* is set.

-R

The front end sets this option is *replay* is set.

-S*sched*

Normally, the schedulability analyser will assume that the priority preemptive scheduling is being used, unless the *STRESS* program specifies otherwise. However, this default can be changed using the **-S** (**s**cheduler) option; *sched* can be one of *pripre*, *early* or *least*, exactly as in a *STRESS* program. Note that the **-S** option is still overriden by explicit program commands.

The front end sets this option according to the selected scheduling scheme.

-s

Using the **-s** (**s**hort) option, logging output uses the short form of names; names appear as they do in the program, rather than being extended to relect their position in the system hierarchey (eg., a variable *var* might be represented by *var@task@proc@node@<network>* in its extended form.

-T*kernf*

This option can be used to specify a file containing execution times for the semaphore and message operations. This is detailed in the *miscellany* option.

-t*# (or -t#1-#2).*

By default, the simulator executes for 100 ticks and, if logging is enabled, logs during that period. This can be changed using the **-t** (**t**imes) option. In the first form, **-t**$_{\#}$, the simulator executes for the specified number of ticks, and logs if logging is enabled. In the second form **-t***#1-#2*, the simulator executes for *#2* ticks, but only logs in the time interval *#1* to *#2* (again, if logging is enabled).

The front end sets this option on the basis of *sim time*.

-W

If the **-W** (**W**orst) option is selected, then the worst-case time is always used whenever a timing statement is executed. Hence, a timing statement *[n,m]* always consumes *m* processor ticks. The **-B** option can be used to select the best-case time.

The front end sets this option if *worst case* is set.

-w

Setting the **-w** option cases the simulator to wait for a return to be typed before it exits.

prog

prog is the name of the file containing the program to be simulated.

6.3. The Display Tool

-A

Using the **-A** (**A**4) option, the initial display size corresponds to A4 hardcopy output. If this option is not selected then the initial size is based on the number of tasks in the system.

-d

The **-d** (**d**isplay) option causes the display tool to output in text form a list of all events in the log file, and stops it from producing any graphical output. Note that this option is generally only useful if the display tool is invoked directly from the command line.

-j

By default, the display tool starts in the mode where task execution is shown by a separated crossed box for each tick consumed. The **-j** (**j**oined) option selects the mode where several consecutive ticks used by the same task is shown as an elongated box, without a cross.

-n

If the **-n** (**n**one) option is selected, then hidden tasks are not shown on the display.

-N

The -**N** (**N**one) option is similar to the **-n** option, except that neither hidden tasks, nor the idle task, are displayed.

-o

The **-o** (**o**utput) option has the same effect as the **-d** option, except that only output events are displayed.

-s

By default, task names appear on the display in exactly the form that they appear in the logging file. Unless the **-s** option to the simulator was selected, this will be the extended form. If the display tool **-s** (**s**hort) option is selected, then names will be truncated to their short form (hence, *task@proc@node@<network>* will appear simply as *task*).

6.4. The Front-End Driver

-A*file*

The analysis tool contained in the file *file* is used, rather that the standard tool.

-D*file*

The display tool contained in the file *file* is used, rather that the standard tool.

-d

The commands generated to run the various tools are reflected to the output window immediately before the tool in invoked.

-S*file*

The simulator contained in the file *file* is used, rather that the standard simulator

6.5. An Example

The example shown in figure 26 shows a simple script which runs the analysis tool, the simulator and then the display tool. It assumes that the program is being run on a Sun3 under Sun Windows or under Mux.

```
/usr/drtee/stress3/bin/Sun/analysis -Oprogram stressprog /dev/null program__spec
/usr/drtee/stress3/bin/Sun/sim -t1000 -Oprogram -D stressprog
/usr/drtee/stress3/bin/Sun/display -n -j stressprog_
```

Figure 26: Example Script to run Stress

7. MISCELLANY

This section is a collection of miscellaneous points which could not be found a reasonable home in any other section.

7.1. The *Dangling Done* Problem

A problem may sometimes be observed when a task completes its execution in the tick immediately before its deadline, and some other higher priority task becomes free to run at the next tick. Since scheduling occurs immediately after a tick has been consumed, the former task may be preempted *before* it terminates, even though it has no further work to do, and need consume no further processor time.

This is referred to as the *dangling done* problem. By default, the simulator will consider that the task has overrun its deadline, and will kill it at the next tick. However, the **-D** option to the simulator may be used to change this behaviour. Setting this option has the effect that the task will terminate correctly provided that, after its last timing *[n,n]* operation, it executes only **atomic**, **fail**, **loop**, **for**, **if** and **return** operations, evaluates expressions and invokes methods.

7.2. Invoking the Tools from the Command Line

Although the tools can be easily driven using the front-end, thay can also be executed directly from the command line. The default tools are located in the directory */usr/drtee/stress2/bin*. See the *options* section for details of their arguments and options.

7.3. Kernel Execution Time

The *STRESS* simulator in effect simulates a kernel which provides various primitives, such as semaphores. By default, no processor time is consumed when these primitives are invoked. If this is not adequate, and kernel time cannot be subsumed into task execution times, then the execution of various primitives can be specified via the **-T***kernf* option to the simulator.

kernf is the name of a file containing the times for each operation. An example is shown in figure 27. **P** and **V** refer to the semaphore operations, and **send** and **recv** to the mailbox operations.

```
P       2
V       1
send    2
recv    1
```

Figure 27: Example kernel times file

The semantics is such that, when a kernel operation is executed, it consumes that number of ticks. During that time, there will be no task switches, and the ticks will be attributed to the calling task. Note that the **P** and **recv** operations will consume the time when they are first executed; they will not consume any further time even if they block the calling task.

7.4. Context Switch Time

By default, task-to-task context switching takes zero time. For example, if task *A* is executes for some tick, and task *B* is released at the start of the next tick, then task *B* will execute immediately in the latter tick. This is obviously an idealised situation.

In some circumstances, this is not important, and the context switch time can be subsumed into task exeuction times using the *[n,m]* command. However, if this is not adequate,

context switch times can be explicitely given, using the **-c** option to the simulator.

If a non-zero context switch time is given, then, once a context switch has been found to be necessary on a processor, then that processors will not execute any task code for the context switch time. During such a period, other events may occur which themselves require another task switch (ie., a task release time occurs); in this case, there may be another context switch immediately after the current one.

7.5. The Event File

An event file, specified to the simulator using the **-E** option, lists times (in ticks) at which named events occur. These can be used to start event-triggered sporadic and aperiodic tasks.

The file is a simple text file, with a separate line for each occurance. The line comprises the time in ticks, and the event name, in that order and separated by a tab character. The lines must be in order of increasing tick number. An event may occur any number of times, and may even occur more than once on a particular tick. One (time,event) pair is "consumed" for each task release.

7.6. Recording and Replay

7.7. Different Machines

The directory */usr/drtee/stress3/bin* contains a number of subdirectories, each of which contain the command *analysis*, *sim*, etc. The following may be available:

Sun	Sun3 using Sun Windows or Mux
Sun_X	Sun3 using X windows
Sparc	Sun Sparc using X windows

Sha90. L. Sha, R. Rajkumar and J. P. Lehoczky, ‘‘Priority Inheritance Protocols: An Approach to Real-Time Synchronisation’’, *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).