Insert as Section 9.9 **{version:2}**

9.9 Fine-grain process control

A **process** is a built-in class that allows one process to access and control another process once it has started. Users can declare variables of type process and safely pass them through tasks or incorporate them into other objects. The prototype for the **process** class is:

```
class process;
    enum state { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };

    static function process self();
    function state status();
    task kill();
    task await();
    task suspend();
    task resume();
endclass
```

Objects of type **process** are created internally when processes are spawned. Users may not create objects of type **process**; attempts to call new shall not create a new process, and instead result in an error. The **process** class cannot be extended. Attempts to extend it shall result in a compilation error. Objects of type **process** are unique; they become available for reuse once the underlying process terminates and all references to the object are discarded.

The **`self`**`()` function returns a handle to the current process, that is, a handle to the process making the call.

The **`status`**`()` function returns the process status, as defined by the `state` enumeration:

- **FINISHED** Process terminated normally.
- **RUNNING** Process is currently running (not in a blocking statement).
- **WAITING** Process is waiting in a blocking statement.
- **SUSPENDED** Process is stopped awaiting a resume.
- **KILLED** Process was forcibly killed (via kill or disable).

The **`kill`**`()` task terminates the given process and all its sub-processes, that is, processes spawned using fork statements by the process being killed. If the process to be terminated is not blocked waiting some other condition, such as an event, **wait** expression, or a delay then the process shall be terminated at some unspecified time in the current time step.

The **`await`**`()` task allows one process to wait for the completion of another process. It shall be an error to call this task on the current process, i.e., a process may not wait for its own completion.

The **`suspend`**`()` task allows a process to suspend either its own execution or that of another process. If the process to be suspended is not blocked waiting some other condition, such as an event, **wait** expression, or a delay then the process shall be suspended at some unspecified time in the current time step. Calling this method more than once on the same (suspended) process has no effect.

The **`resume`**`()` task restarts a previously suspended process. Calling resume on a process that was suspended while blocked on another condition shall re-sensitize the process to the event expression, or wait for the **wait** condition to become true, or for the delay to expire. If the **wait** condition is now true or the original delay has transpired, the process is scheduled onto the Active or Reactive region, so as to continue its execution in the current time step. Calling resume on a process that suspends itself causes the process to continue to execute at the statement following the call to **suspend**.

The example below starts an arbitrary number of processes, as specified by the task argument N. Next, the task waits for the last process to start executing, and then waits for the first process to terminate. At that point the parent process forcibly terminates all forked processes that have not completed yet .

```
task do_n_way( int N );
    process job[1:N];

    for( int j = 1; j <= N; j++ )
        fork
            begin job[j] = process::self(); ... ; end
        join_none

    wait( job[N] != null );     // wait for last process to start

    job[1].await();             // wait for first process to finish

    for( int k = 1; k <= N; k++ ) begin
        if( job[k].status != process::FINISHED )
            job[k].kill();
    end
endtask
```