# SystemC Process Control Extensions

*Bishnupriya Bhattacharya, John Rose, and Stuart Swan*

May, 2009

## 1 Problem Statement and Scope

Starting from version 1.0, SystemC provides the concept of processes (methods and threads) that are scheduled and activated under the control of a SystemC kernel scheduler. The scheduling is determined by the sensitivity of a process, either static or dynamic. When a method process is activated, it executes from beginning to end without interruption, and then returns control to the kernel. A thread process in each activation executes from the beginning or from a `wait()` statement to another `wait()` statement, or to the end of the function – when a `wait()` statement is encountered, or a function returns, control returns to the kernel. A thread process terminates when it reaches the end of its function body. A method process never terminates; it triggers repeatedly, depending on its sensitivity.

This process execution infrastructure is sufficient for many applications, but it clearly lacks a more direct API for controlling one process from another (e.g., kill/reset, or disable-enable). This is a well known deficiency in SystemC (see 3), compared to other languages like SytemVerilog in the verification space, or Verilog in the design space, which do provide some flavors of process control, albeit, in a very limited capacity. Process control constructs are required in a variety of applications, including RTOS modeling, testbench modeling, and abstract (high level) modeling of HW systems.

After analyzing the current deficiencies in SystemC, especially with respect to the application areas described above, in this document, we identify a fundamental and general purpose set of SystemC APIs and kernel extensions that shall enable effective modeling of such applications. Our work is partially based on the OSCI SystemC 3.0 specification 2, 3. Part of the SystemC 3.0 specification has already been made a language standard in IEEE 1666 1, namely

- creating dynamic processes via *sc_spawn()*
- obtaining a *sc_process_handle* for a dynamic or static process instance
- allowing a process to wait for another process to terminate.

Our contribution is to

- build upon the process control constructs described in 2, 3 – namely suspend, resume, kill, and throw a C++ exception in a process
- extend the specifications in 2, 3 with additional process control constructs for completeness – namely disable, enable, reset, sync_reset_on and sync_reset_off

In addition, we provide the crucial details missing from the 3.0 specification

- precise semantics of each construct
- precise semantics of how the constructs interact with each other, including their relative priorities

## 2 Specification

The sc_process_handle class is documented in IEEE 1666 as a representation of an underlying process instance. For modeling process control constructs in SystemC, new member methods shall be added to the sc_process_handle class, as described below.

### 2.1 Suspending a process

```
enum sc_descendant_inclusion_info
{
  SC_NO_DESCENDANTS,
  SC_INCLUDE_DESCENDANTS
};

void sc_process_handle::suspend(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

suspend() shall suspend a process such that it is not eligible to run again till it is resumed. If the include_descendants flag is set to SC_INCLUDE_DESCENDANTS, the suspension shall apply to the entire process hierarchy tree rooted at the target process handle, in a bottom up fashion.

During suspension, a process shall notice, and remember any triggering of sensitivity that was in effect when it got suspended. suspend() shall add an event (referred to as the resume_event) to the effective sensitivity list of the target process, where resume_event is notified by resume(). Thus, the effect of suspending a process shall be equivalent to changing the effective sensitivity of the process as follows:

```
{effective sensitivity after suspension} =
{effective sensitivity before suspension} && resume_event
```

If the effective sensitivity before suspension is a timeout – like wait(100, SC_NS)- then after suspension, the effective sensitivity shall be the process's timeout event and-ed with its resume_event. This is equivalent to the process having said:

```
sc_event e; e.notify(100, SC_NS); wait(e && resume_event);
```

There shall be no history associated with who suspended a process – anyone shall be able to suspend a process, and anyone shall be able to resume a process. Suspending a process that is already suspended shall be like a no-op. If multiple suspends are invoked on the same process, it shall only take one resume() to resume it.

If a process suspends itself, a method shall complete its current execution before the suspension takes effect. For a thread, the suspension shall take effect immediately, and its effective sensitivity shall be its resume_event.

Suspending a terminated process shall be like a no-op, and if the include_descendants flag is set to SC_INCLUDE_DESCENDANTS, the process hierarchy rooted at the target process shall be traversed, and action shall be taken depending on the live-ness of each process.

Suspending a process before simulation has started (e.g., from one of the callback phases) shall have the following effect on the process's sensitivity:

- If the process was declared with `dont_initialize`, then its effective sensitivity shall be modified by and-ing its `resume_event` with its existing effective sensitivity (i.e. with its static sensitivity)
- If the process was not declared with `dont_initialize`, then its effective sensitivity shall be set to its `resume_event`; this implies the process shall trigger as soon as the `resume_event` triggers

## 2.2 Resuming a process

```
void sc_process_handle::resume(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

`resume()` shall lift any previous suspensions on the process and make it eligible to run again. If the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, then the `resume()` shall apply to the entire process hierarchy tree rooted at the target process, in a bottom up fashion.

The effect of `resume()` shall be the same as notifying the `resume_event` that the target process was sensitive to, with a delta cycle delay.

A thread shall be resumed from where it last left off; a method shall be resumed from the beginning.

Calling `resume()` on a process that is not suspended shall have no effect.

If a process resumes itself, it shall have no effect, since the current process is already executing, but may not be completely pointless if the `include_descendants` flag is set, since some of its descendants maybe suspended.

Resuming a process before simulation has started shall not be a special case.

Resuming a terminated process shall be like a no-op, and if the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the process hierarchy rooted at the target process shall be traversed and action shall be taken depending on the live-ness of each process.

This suspend-resume semantics imply: if the effective sensitivity of the suspended process triggered during the time the process was suspended, then the process shall execute as soon as it is resumed. If the effective sensitivity of the process did not trigger while the process was suspended, then `resume()` shall not make the process execute, it shall still wait till its effective sensitivity triggers.

## 2.3 Disabling a process

```
void sc_process_handle::disbale(
  sc_descendant_inclusion_info
```

```
  include_descendants = SC_NO_DESCENDANTS
);
```

`disable()` shall disable a process such that it is not eligible to run again till it is enabled. If the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the disability shall apply to the entire process hierarchy tree rooted at the target process handle, in a bottom up fashion.

During its disability, a process shall not notice any triggering of its sensitivity, and shall ignore such triggering. So, a process's sensitivity shall be "frozen" during the time it is disabled.

There shall be no history associated with who disabled a process – anyone can disable a process, and anyone can enable a process. Disabling a process that is already disabled shall be like a no-op. If multiple `disable()` calls are invoked on the same process, it shall only take one `enable()` to enable it.

If a process is scheduled to run and gets disabled, or if a process disables itself, it shall complete its current execution before the `disable()` call takes effect. In other words, a process shall "consume" the last sensitivity triggering that caused it to get scheduled, and shall be able to set up its effective sensitivity for the next triggering, before getting disabled.

Disabling a terminated process shall be like a no-op, and if the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the process hierarchy rooted at the target process shall be traversed, and action shall be taken depending on the live-ness of each process.

Disabling a process before simulation has started (e.g., from one of the callback phases) shall not schedule the process to execute at time 0.

### 2.4 Enabling a process

```
void sc_process_handle::enable(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

`enable()` shall lift any previous disabilities on the process and shall make it eligible to run again. If the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, then the `enable()` shall apply to the entire process hierarchy tree rooted at the target process, in a bottom up fashion.

Unlike `resume()`, `enable()` shall never cause a disabled process to execute immediately. The process shall always wait for its sensitivity to trigger before it executes. It is possible that the process has missed a timeout event during the time it was disabled. If the timeout event consisted of the process's entire effective sensitivity (e.g., from a `wait(100, SC_NS)` call), then it is possible the process shall never run again. An implementation may choose to issue a warning message from an `enable()` call, if the process has missed any timeout event while it was disabled.

When the process executes again, a thread shall execute from where it last left off, and a method shall execute from the beginning.

Calling `enable()` on a process that is not disabled shall have no effect.

If a process enables itself, it shall have no effect, since the current process is already executing, but may not be completely pointless if the `include_descendants` flag is set, since some of its descendants maybe disabled.

Enabling a process before simulation has started shall not be a special case.

Enabling a terminated process shall be like a no-op, and if the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the process hierarchy rooted at the target process shall be traversed and action shall be taken depending on the live-ness of each process.

## 2.5 Comparison of suspend-resume and disable-enable

The distinction between *suspend-resume* and *disable-enable*, and also the compelling reason for having both forms, is demonstrated by the example of a thread process sensitive to a clock that triggers at times 10, 20, 30, and so on. If the process is suspended at time 15, and resumed at time 55, it will immediately execute at time 55. However, if the process is disabled at time 15, and enabled at time 55, it will not execute at time 55, instead it will execute at time 60 when the next clock comes

## 2.6 Killing a process

```
void sc_process_handle::kill(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

`kill()` shall kill a process such that, if it is in the middle of its function stack, then the process shall exit its function, and its stack shall unwind and all local objects shall get destructed. The process shall not be eligible to run again ever. Control shall then return to the initiator. Stack unwinding of the target process shall be accomplished by throwing a `sc_kill_exception` in the target process's stack. If the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, then all processes in the hierarchy tree rooted at the target process shall be killed, in a bottom up fashion.

`kill()` has an immediate/asynchronous effect. For a target thread, the target thread shall immediately exit its function, and return control to the initiator process. No other processes shall execute between the time the `kill()` of a target process is ordered and the target obliges. For a target method process, it is never in the middle of its function stack unless it is currently executing. Note that the killed process should at some point be removed from the simulation, but that point is implementation dependent, all that shall be guaranteed after the `kill()` call is that the process never runs again.

It shall be illegal to do any kind of scheduling or blocking calls (like `wait()` or `next_trigger()`) while getting killed. For example, the destructors of local objects in the killed thread shall get executed when the process is getting killed, and those destructors shall return immediately.

If a process kills itself, both for a thread and a method, it shall be immediately made to exit its associated function, and control shall go to the next process eligible to run.

Killing a process before simulation has started, shall make the process never execute.

Killing a terminated process shall be like a no-op, and if the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the process hierarchy rooted at the target process shall be traversed and action shall be taken depending on the live-ness of each process.

It is possible that when a `kill()` call finishes, and its time to return to the killer, the killer may not be in a state to execute. In course of executing the `kill()` call, the killer may have been killed or suspended, for example, due to side effects of stack unwinding. In that case, control shall go to the next process ready to run.

If a process so desires, it can provide a handler for `sc_kill_exception` in its function body, for example, to perform some exit cleanup. In such cases, the handler shall throw back the `sc_kill_exception` so that the implementation can also handle the exception and take the right action.

```
SC_MODULE(m) {
public:
  SC_CTOR(m) : { SC_THREAD(run); }
  void run() {
    try {
       ……..
    }
    catch (sc_kill_exception& ex) {
      // perform exit cleanup
      throw ex;
    }
  }
  …
};
```

## 2.7 Resetting a process (asynchronous)

```
void sc_process_handle::reset(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

`reset()` shall be the same as killing the process in terms of exiting the associated function. In addition, `reset()` shall nullify any “state” the target process may have built up during simulation, and restore it to its “state” at time 0.

For a target thread process, the semantics are

- the thread shall be switched in and killed by throwing an exception that shall unwind its stack
- any dynamic sensitivity shall be cancelled and its static sensitivity shall be restored
- the thread shall be executed from the beginning
- control shall go back to the initiator after the thread yields to the kernel (by executing a wait() statement or by returning from its function)

For a target method process, the semantics are

- any dynamic sensitivity shall be cancelled and its static sensitivity shall be restored
- control shall go back to the initiator

If the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, then all processes in the hierarchy tree rooted at the target process shall be reset in a bottom up fashion. `reset()` shall be immediate (and asynchronous) like `kill()`.

If a process resets itself, after the `reset()` finishes, control shall go to the next process ready to run. The process handle of the target process shall still be valid after `reset()`.

Resetting a process before simulation has started, or before the process has executed for the first time, shall have no effect.

Resetting a terminated process shall be like a no-op, and if the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the process hierarchy rooted at the target process shall be traversed and action shall be taken depending on the live-ness of each process.

It is possible that when a `reset()` call finishes, and its time to return to the initiator, the initiator may not be in a state to execute. In course of executing the `reset()` call, the initiator may have been killed or suspended, for example, due to side effects of stack unwinding. In that case, control shall go to the next process ready to run.

## 2.8 Resetting a process (synchronous)

There shall be two different constructs to synchronously reset a process

- Explicit synchronous reset through sync_reset_on() and sync_reset_off()
- Implicit synchronous reset through reset_signal_is() and async_reset_signal_is()

Applying an explicit or an implicit reset shall place the target process in a synchronous reset state. The target process shall be released from the synchronous reset state when neither an explicit reset, nor an implicit reset is active.

### 2.8.1 Explicit reset

```
void sc_process_handle::sync_reset_on(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);

void sc_process_handle::sync_reset_off(
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

`sync_reset_on()` shall place the target process in an explicit synchronous reset state – it shall set up the state of the target process such that every time it wakes up due to its sensitivity having triggered, the process shall be reset:

- the process shall be killed by throwing a `sc_kill_exception` in its stack
- any dynamic sensitivity shall be cancelled and its static sensitivity shall be restored

- process execution shall be restarted from the beginning

`sync_reset_off()` shall nullify the effect of previous `sync_reset_on()` calls, if any, and shall release the process from explicit synchronous reset state. This may allow the process to operate normally again, provided there are no active implicit resets on the target process (see Sections 2.8.2. and 2.8.3 for further details).

If the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, then all processes in the hierarchy tree rooted at the target process shall have the `sync_reset_on()` or `sync_reset_off()` call issued on their process handles, in a bottom up fashion.

Unlike `reset()`, `sync_reset_on()` or `sync_reset_off()` shall not have immediate effect. These shall only take effect when the process wakes up due to its effective sensitivity having triggered.

There shall be no history associated with who issued `sync_reset_on()` on a process – anyone shall be able to `issue sync_reset_on`() on a process, and anyone shall be able to issue `sync_reset_off()` on a process. Issuing `sync_reset_on`() on a process that is already under the influence of a previous `sync_reset_on()` call is like a no-op. If multiple `sync_reset_on()` calls are invoked on the same process, it shall only take one `sync_reset_off()` call to turn it off.

`sync_reset_on`() shall only be applicable to thread processes, it shall be an error to invoke `sync_reset_on()` on a method process, because the semantics of `sync_reset_on()` does not apply to method processes.

Calling `sync_reset_off()` on a process that is not currently in a `sync_reset_on`() state, shall have no effect.

If a process issues a `sync_reset_on()` or a `sync_reset_off()` call on itself, it shall complete its current execution, and the calls shall take effect the next time the process wakes up.

Issuing `sync_reset_on()` on a process before simulation has started, or before the process has executed for the first time, shall have no effect.

Issuing `sync_reset_on()` on a terminated process shall be a no-op, and if the `include_descendants` flag is set to `SC_INCLUDE_DESCENDANTS`, the process hierarchy rooted at the target process shall be traversed and action shall be taken depending on the live-ness of each process.

### 2.8.2 Implicit reset

IEEE 1666 specifies `reset_signal_is`() only for a `SC_CTHREAD` process. Here, we shall generalize it to all kinds of processes, spawned and unspawned, threads and methods and cthreads. In addition we shall define a new construct `async_reset_signal_is()`.

```
void sc_module::reset_signal_is(
  const sc_in<bool>& port, bool level
);

void sc_module::reset_signal_is(
  const sc_signal<bool>& sig, bool level
);
```

```
void sc_module::async_reset_signal_is(
  const sc_in<bool>& port, bool level
);

void sc_module::async_reset_signal_is(
  const sc_signal<bool>& sig, bool level
);

void sc_spawn_options::reset_signal_is(
  const sc_in<bool>& port, bool level
);

void sc_spawn_options::reset_signal_is(
  const sc_signal<bool>& sig, bool level
);

void sc_spawn_options::async_reset_signal_is(
  const sc_in<bool>& port, bool level
);

void sc_spawn_options::async_reset_signal_is(
  const sc_signal<bool>& sig, bool level
);
```

`sc_module::reset_signal_is()` shall specify a reset port or a reset signal along with the reset edge for the process that was last declared using the process macros `SC_THREAD`, or `SC_CTHREAD`. Whenever the reset port or the reset signal attains the reset value, the process shall be put in an implicit synchronous reset state – every time the target process wakes up due to its sensitivity having triggered, the process shall get reset:

- the process shall be killed by throwing a `sc_kill_exception` in its stack
- any dynamic sensitivity shall be cancelled and its static sensitivity shall be restored
- process execution shall be restarted from the beginning

When the reset port or reset signal attains the non-reset value, the target process shall be released from the implicit synchronous reset state. This may allow the process to operate normally again, provided there are no active explicit resets on the target process (see Sections 2.8.2. and 2.8.3 also).

It shall be an error to specify `reset_signal_is`() for a method process.

`sc_module::async_reset_signal_is()` shall specify a reset port or a reset signal along with the reset edge for the process that was last declared using the process macros `SC_THREAD`, `SC_CTHREAD`, or `SC_METHOD`. For a thread or cthread process, the semantics shall be the same as `reset_signal_is()`; in addition, the process shall also get reset as soon as the reset port or reset signal attains the reset value.

For a method process, the semantics of `async_reset_signal_is()` shall be to reset the method process as soon as the reset port or the reset signal attains its reset value:

- any dynamic sensitivity shall be cancelled and its static sensitivity shall be restored

For a method process, the async_reset_signal_is() behavior shall be implemented using the core process control construct reset().

The sc_spawn_options member methods are equivalent ways to achieve the same functionality for spawned processes.

Note that the capability to specify reset_signal_is() on a SC_THREAD process sensitive to a clock, makes SC_CTHREADs redundant and allows SC_CTHREADs to be removed from SystemC altogether.

### 2.8.3 Multiple explicit and implicit resets

It shall be legal to specify multiple reset_signal_is() and multiple async_reset_signal_is() constructs for the same target process. At the same, it is possible that explicit reset using sync_reset_on() is also specified on the same target process. In such cases, the semantics shall be as follows:

A process shall be put in a synchronous reset state either by an explicit synchronous reset or by an implicit synchronous reset, i.e. under the following circusmstances:

- When a sync_reset_on() call is issued on the target process
- When ANY of the reset ports or reset signals specified in reset_signal_is() constructs for the target process has attained it's reset value
- When ANY of the reset ports or reset signals specified in async_reset_signal_is() constructs for the target process has attained it's reset value; in addition, the target process shall also be immediately (asynchronously) reset using reset()

A process shall be released from a synchronous reset state when neither an explicit reset, nor an implicit reset is active, i.e. under the following circumstances

- When the target process is not under the influence of a pending sync_reset_on() call, and ALL of the reset ports and reset signals specified in reset_signal_is() and async_reset_signal_is() constructs for the target process have attained their non-reset values

## 2.9 Throwing an exception in a process

```
template <typename T>
void sc_process_handle::throw_it(
  const T& user_defined_exception,
  sc_descendant_inclusion_info
  include_descendants = SC_NO_DESCENDANTS
);
```

A thrower process shall issue a request to throw an exception in a throwee process using the above API. The exception to be thrown shall be provided as the first argument, and the second argument shall specify if the throw affects just the throwee process or all its descendants too – if the include_descendants flag is set to SC_INCLUDE_DESCENDANTS, the throw shall apply to the entire hierarchy tree rooted at the throwee process handle, in a bottom up fashion.

The throw_it() call is asynchronous and shall have immediate effect. The thrower process shall be suspended, and the throwee thread shall be switched in, and the specified exception

shall be thrown in the throwee's stack. It is expected that the throwee shall catch the exception in its associated function. After it catches the exception, the throwee can either return from its function (terminate) or it can issue a `wait` statement (suspend). In either case, once the throwee yields control to the kernel, the kernel shall switch back in the thrower process. No other process shall run in between. If the throwee thread does not catch the exception, it is an error - the kernel shall catch the uncaught exception and report it as an `UNKNOWN EXCEPTION`, and the default behavior shall be to exit the simulation.

The currently executing process shall be able to throw an exception in itself. In that case, no context switching between thrower and throwee shall take place, and control shall go to the next process eligible to run.

Throwing an exception in a process shall be meaningful only when the throwee process has started execution and is at some point in its associated stack. This implies that the throwee is either currently executing or it is suspended (i.e. it is at a `wait()` statement, or its process handle had `suspend()` or `disable()` invoked). Thus, it is meaningless to throw an exception in a method process, which has exited its associated function stack the last time it ran. The currently executing method process can be considered as an exception, however, for the sake of consistency and simplicity, all method processes shall be treated the same including the currently executing one. Hence, throwing an exception in a method process shall have no effect and shall generate a warning from the simulator if the `include_descendants` flag is not set. If the `include_descendants` flag is set, it is possible the exception throw shall apply to descendant thread processes, hence the hierarchy shall be traversed and appropriate action shall be taken for each descendant process. Note that if the currently executing method process wants to thrown an exception in itself, the simple way to achieve that is through a direct exception throw – `throw my_exception` – rather than through an indirect exception throw

```
sc_get_current_process_handle().throw_it(my_exception);
```

For similar reasons, throwing an exception in a terminated process shall be an error if the `include_descendants` flag is not set. If the `include_descendants` flag is set, the hierarchy shall be traversed and appropriate action shall be taken for each descendant process.

Throwing an exception in a process that has not started execution yet shall have no effect for the same reasons as above, and shall be an error. Setting up the `include_descendants` flag has no associated semantics in this case, because the throwee process is guaranteed to not have any descendants, as it has never executed, and hence, has never spawned any children.

Throwing an exception when simulation is not running (before simulation start or after simulation end) shall have no effect and shall be an error.

The thrower shall always be a process. It shall be an error to throw an exception from any other context, for example from a phase callback or from the `update()` routine of a user defined channel.

If a throwee is in the middle of a thrown exception, then throwing another exception in the same throwee process shall be an error.

It shall be illegal to do any kind of scheduling or blocking calls (like `wait()` or `next_trigger()`) while stack unwinding occurs during a throw. For example, if the throwee was suspended in the middle of a function call, then throwing an exception in it shall cause

that functions' stack to unwind. The destructors of local objects going out of scope during stack unwinding shall get executed, and those destructors shall return immediately.

### 2.10 Nested calls: kill, reset and throw_it

It shall be legal to have nested exception throwing, killing and resetting calls. A throwee shall be able to catch an exception and then throw the same exception or a different exception in another process, before yielding control to the kernel. Also, destructors of objects going out of scope during any stack unwinding shall be able to throw an exception in another process, or can kill/reset another process. Such nested calls shall be legal and shall be handled appropriately.

## 3 Interactions Among the Process Control Constructs

Among the different process control constructs discussed above, their relative priorities shall be as listed below, from highest to lowest.

1. `kill()` and `reset()` and `throw_it()`
2. `disable()` and `enable()`
3. `suspend()` and `resume()`
4. `sync_reset_on()` and `sync_reset_off()`

The asynchronous constructs shall have higher priority over the non-asynchronous constructs due to the immediate semantics of the former. For example, if a process is suspended, and an exception is thrown in it, the `throw_it()` call shall prevail.

A suspended or disabled process can be killed or reset and an exception can be also be thrown in it. But these shall not lift the suspension or the disability. For example, if a suspended or disabled thread is reset, it shall restart from the beginning, and after it yields to the kernel via a `wait()` statement, the thread shall go back to its suspended or disabled state. If an exception is thrown in a suspended or disabled thread, it shall wake up and throw the exception, and after catching the exception and yielding back to the kernel via a `wait()` statement, the thread shall go back to its suspended or disabled state.

Among the non-asynchronous constructs, `disable()`-`enable()` is more powerful than `suspend()`-`resume()`. For example, if a process is suspended, and then disabled, the `disable()` call shall prevail, such that if the process gets its sensitivity triggered and the process also gets resumed during its disability, it shall miss those, and shall not execute. It shall go back to its suspended state, after it is enabled.

`sync_reset_on()` and `sync_reset_off()` have the lowest priority among all the process control constructs. For example, if `sync_reset_on()` is issued on a suspended or disabled process , it shall have no immediate effect. The `sync_reset_on()` call shall take effect only after the process is resumed or enabled and it tries to wakes up.

# 4 Examples

## 4.1 AC power to battery mode in laptop

Consider a graphics co-processor inside a laptop computer in the middle of some graphics processing, when the laptop goes from AC power to battery power. The graphics processor may immediately need to switch into a computationally less intensive low power mode. Fig. 1 shows how a user defined exception can be used to effectively model the asynchronous event of the laptop going from AC power to battery power. Any change in power is signaled by notifying the event *power_change_event*, which wakes up the *handle_power_change* process, which throws the appropriate exception (*low_power_ex* or *hi_power_ex*) in the *main_loop* process. The *main_loop* process is likely to be suspended at a *wait()* statement when the power change happens; it immediately wakes up due to the thrown exception, handles it, loops back to the beginning and switches to low power processing mode.

```
SC_MODULE(graphics_coprocessor) {
  class low_power_ex { };
  class hi_power_ex { };
  sc_event power_change_event;
  sc_process_handle main_loop_handle;

  SC_CTOR(graphics_coprocessor) {
    SC_THREAD(handle_power_change);
    SC_THREAD(main_loop);
    main_loop_handle =
      sc_get_current_process_handle();
  } // CTOR

  void handle_power_change() {
    while (1) {
      wait(power_change_event);
      if (low_power())
      { main_loop_handle.throw_it(low_power_ex()); }
      else { main_loop_handle.throw_it(hi_power_ex()); }
    } // while (1)
  } // handle_power_change

  void main_loop {
    bool low_pwr_mode = false;
    while (1) {
      try {
        if (low_pwr_mode) {… // low power processing }
        else { … // regular power processing }
      } // try
      catch (low_power_ex& x) {low_pwr_mode = true;}
      catch (hi_power_ex& x) {low_pwr_mode = false;}
    } // while (1)
  } // main_loop
}; // module graphics_coprocessor
```

**Fig. 1. Example of *throw_it()* in graphics co-processor.**

### 4.2 Traffic generator in testbench

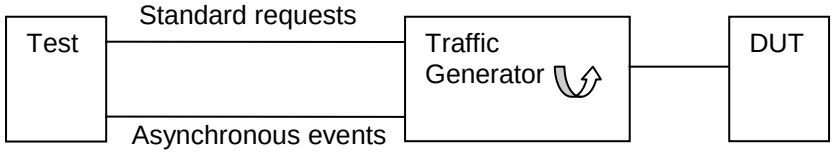


**Fig. 2a A traffic generator testbench.**

Consider a traffic generator for some bus protocol, as shown in Fig. 2a. In this kind of typical case, the traffic generator is a free running process which takes requests from the test (maybe many at a time) and the process runs free. The testbench pseudo code is shown in Fig. 2b. The *traffic_gen* thread is spawned by the thread *run*. To handle asynchronous events like bus reset, each component in the testbench needs explicit monitoring code (shown in bold), at each point where it can block via a *wait()* statement. This example has three such blocking points, and at each blocking point, the *reset_high* event has to be "or"-ed with the regular event list, and reset status needs to be checked on coming out of the *wait()*. In a thread with many blocking points, it is cumbersome and error-prone to model an asynchronous activity in this fashion.

```
SC_MODULE(testbench) {
  SC_CTOR(testbench) { SC_THREAD(run); }
  void run() {
    // spawn traffic_gen as a thread process
    sc_process_handle h = spawn_traffic_gen();
  }

  void traffic_gen() {
    while(1) {
      data_t t;
      if (reset()) wait(reset_low);
      request_fifo.get(t); send_traffic(t);
    } // while(1)
  } // traffic_gen

  void send_traffic (const data_t& d) {
    while(still_have_data) {
       wait (clock | reset_high);
       if (reset()) { reset_the_bus(); return; }
       send_next_chunk_of_data(d);
    } // while(still_have_data)
  } // send_traffic

  void send_next_chunk_of_data(data_t& d) {
     setup_request_on_bus(d);
     wait(acknowledge | timeout | reset_high);
     if (reset()) { reset_the_bus(); return; }
     end_request_on_bus();
     wait(end | reset_high);
     if (reset()) { reset_the_bus(); return; }
  } // send_next_chunk_of_data
}; // module testbench
```

**Fig. 2b. The testbench code with external reset.**

Using the kernel extensions described in Section 2, it is possible to model this in a more modular and isolated fashion, by modifying the process *run* to kill the *traffic_gen* process when reset is asserted, and to spawn the *traffic_gen* process again when reset is de-asserted. This is shown in Fig. 2c. This simplifies modeling of the rest of the testbench components, and allows the code in bold – "or" of *reset_high* and "if" check – to be removed from Fig. 2b.

```
void run() {
    while(1) {
      sc_process_handle h = spawn_traffic_gen();
      wait(reset_high);
      h.kill();
      reset_the_bus();
      wait(reset_low);
    }
}
```

**Fig. 2c. A single process monitoring external reset.**

## 5 References


1. 1666-2005, IEEE SystemC LRM, available at www.systemc.org, 2005.

2. Requirements for Software Modeling in SystemC 3.0, SystemC Language Working Group, Version 1.6, 2002.
3. T. Grotker, "Modeling Software with SystemC 3.0", 6th European SystemC Users Group Presentations, 2002.