Multi-core Debugger Integration and Suspend/Resume

Peter de Jager

Intel Corporation

SYSTEMC EVOLUTION DAY
OCT 28, 2021 | VIRTUAL WORKSHOP

Copyright Permission

- A non-exclusive, irrevocable, royalty-free copyright permission is granted by **Intel Corporation** to use this material in developing all future revisions and editions of the resulting draft and approved Accellera Systems Initiative **SystemC** standard, and in derivative works based on the standard.

Outline

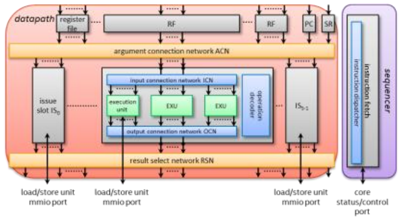
- Background & Motivation
- Problem statement
- Previous solution & other approaches
- Synchronization control
- Generic applicability
- Generic simulation control
- Conclusion, Proposal & Discussion

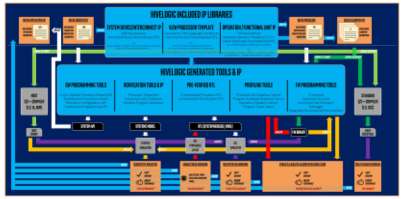
Background & Motivation

- About the author:
  - Located @ Intel Eindhoven, Silicon Hive team
  - Group develops tools (HiveLogic) to create cores and systems
  - Technology has been used in a variety of products for a variety of application domains, including :
    - video coding
    - video post-processing
    - imaging
    - communications

Silicon Hive technology: Four key elements

Design-time configurable processor & system architecture templates supported by elaborate libraries of hand-optimized, fully parameterized processor & peripheral building blocks

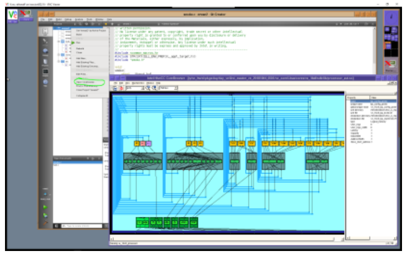

datapath
register file
RF
RF
PC
SR
argument connection network ACN
input connection network ICN
issue slot IS_0
execution unit
EXU
EXU
operation decoder
IS_{S-1}
output connection network OCN
result select network RSN
instruction fetch
instruction dispatcher
sequencer
load/store unit mmio port
load/store unit mmio port
load/store unit mmio port
core status/control port

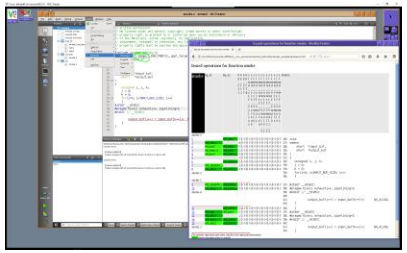

HIVELOGIC INCLUDED IP LIBRARIES
SYSTEM DEVICE/INTERCONNECT IP
VLIW PROCESSOR TEMPLATE
OPERATION/FUNCTIONAL UNIT IP
HIVELOGIC GENERATED TOOLS & IP
SW PROGRAMMING TOOLS
VERIFICATION TOOLS & IP
PRE-VERIFIED RTL
PROFILING TOOLS
FW PROGRAMMING TOOLS

A unique methodology for fast & vast design space exploration at processor and system-level, supported by highly abstract design entry through high-level languages

HIVELOGIC

A fully automated flow and corresponding tools for (multi-) processor & system hardware generation





A fully retargetable programming tool suite based on ANSI-C source entry

Slide courtesy of Jeroen Leijten, Sr. Principal Engineer, Intel Corporation

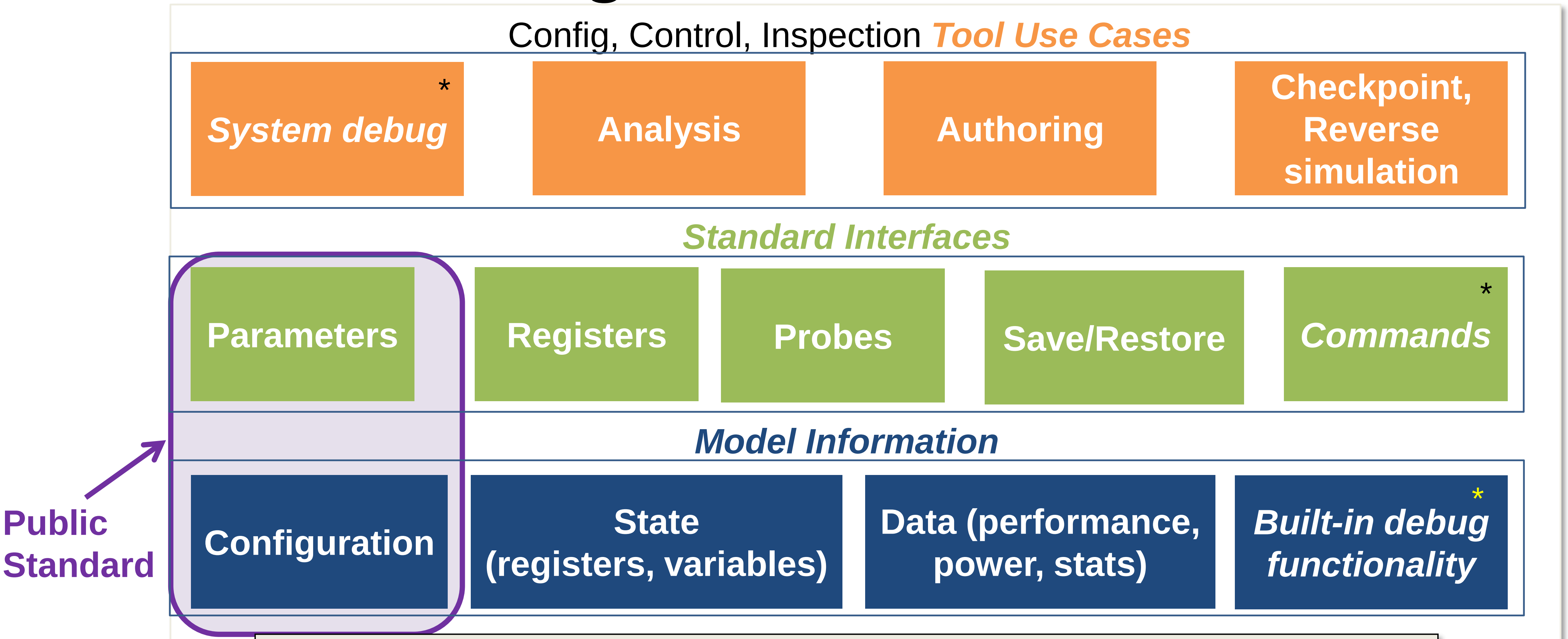
Background & Motivation

- Products that use our technology are
  - Multi-core
  - Heterogeneous
  - Application-specific (dsp, vector, vliw, custom memories, …)

- standard SystemC/TLM used as basis for System-Simulation technology

- A generic mechanism to support application-software debugging is not available in SystemC reference implementation and CCI 1.0

Background & Motivation



Config, Control, Inspection *Tool Use Cases*

System debug *

Analysis

Authoring

Checkpoint, Reverse simulation

Standard Interfaces

Parameters

Registers

Probes

Save/Restore

Commands *

Model Information

Configuration

State (registers, variables)

Data (performance, power, stats)

Built-in debug functionality *

Public Standard

Goal: Standardizing interfaces between models and tools

Problem statement

System-simulation with *n*-core models needs support for multi-core debugging

- REQ. 1 MUST be able to simultaneously connect {0..*n*} debug-connections, each to a separate core
- REQ. 2 MUST provide full (normal) debug functionality per attached debugger, irrespective of other debuggers being connected
- REQ. 3 MUST suspend system-simulation completely at end of current delta-cycle in case of
    - Breakpoint hit (in application code, breakpoint set via debugger)
    - Error triggered (due to application-code)
    - User-break request (via debugger)
- REQ. 4 MUST resume system-simulation only when all attached debuggers have issued (or still are in) '*continue*'-command
- REQ. 5 MUST suspend system-simulation when debug-connection is established during simulation
- REQ. 6 MUST remove debug-connection from current list of 'simulation blockers' when debug-connection is detached
    - When number of 'simulation blockers' is 0, simulation shall resume
- REQ. 7 MUST be able to attach debugger when system in 'suspended'-state (due to other debug-connection)
- REQ. 8 MUST be able to user-break the 'continue'-command in a debugger when system in 'suspended'-state
- REQ. 9 MUST function with official SystemC (currently 2.3.3) distribution
- OPEN How to handle connections to/from other simulators? How do these 'see' that this part is 'suspended'?

RED: not supported with previous solution

Previous solution

As discussed in presentation SystemC Evolution Day 2020

- Parallel debug-thread & simulation-thread
  - Debug-thread uses boost::asio threads to handle multiple connections

- simulation-thread is locked on interrupt/user-break/bp-hit
  - Per iss-model: quite complex handling of step/run commands with locks/mutexes/conditions

- When simulation-thread is locked, new connections & user-break in other debug-connection not possible (since that requires a reaction from the model)

  ➔ prohibits inspection of application code on other cores

Previous solution

As discussed in presentation SystemC Evolution Day 2020

Conclusion last year:

- Move control on SystemC thread stop/continue into global DebugService handling the pausing/resuming of simulation

- Keep administration on corestates & debuggers
  - Intercept userbreak when SystemC-thread is already stopped
  - Continue only when all cores in ‘broken’-state have received continue-command

Other approaches

- (Un)Suspend(able) – Mark burton, SCED-2019
  - Proposes extension to SystemC api ➔ breaks Req. 9
    - sc_suspend_all(sim_context)/sc_unsuspend_all(sim_context)
    - sc_suspendable()/sc_unsuspendable()
  - Primarily aimed at synchronization of time between hybrid simulations (multiple os-processes), snapshotting
  - Using async_update_request, sc_unsuspendable ()/sc_suspendable() a b_transport can be triggered from outside system (temporarily unblocking the simulation) ➔ breaks Req. 3/4

Other approaches

- B. Farkas, Standard Compliant Snapshotting for SystemC VPs, 2019
  - Uses sc_pause to enable the save_state function, thereby ensuring that the event queue is empty
  - The queue will be refilled upon restarting of the simulation and restoring the previous state of the attached models
  - Mentions possibility of snapshotting based on certain events/conditions

- IEEE 1666-2011 Standard SystemC section 4.3.4.2
  - Function **sc_pause** shall cause the scheduler to cease execution at the end of the current delta cycle such that the scheduler can be resumed again later
    - control is returned from sc_start to sc_main again
    - sc_start may be called again to resume simulation
    - Note: sc_start may only be called from within sc_main

GREEN: major difference with previous solution

Synchronization control

- Parallel SimulationControl-thread & Simulation-thread
  - SimulationControl-thread uses boost::asio to handle one or more control-connections

- Use sc_core::sc_pause() to suspend simulation when required
  - Call sc_core::sc_start() again to resume simulation
    Requires control of sc_main implementation

- Simulation-thread is paused on condition in the target: interrupt/user-break/bp-hit
  - Main loop in simulation-thread: simplified handling of pause/resume using 1 mutex/lock and 1 condition to interact with simulation-control thread
  - When simulation-thread is paused & locked, new connections & user-break in other control-connection are possible (since that does not require a reaction from the model anymore)

GREEN: major difference with previous solution

Synchronization control

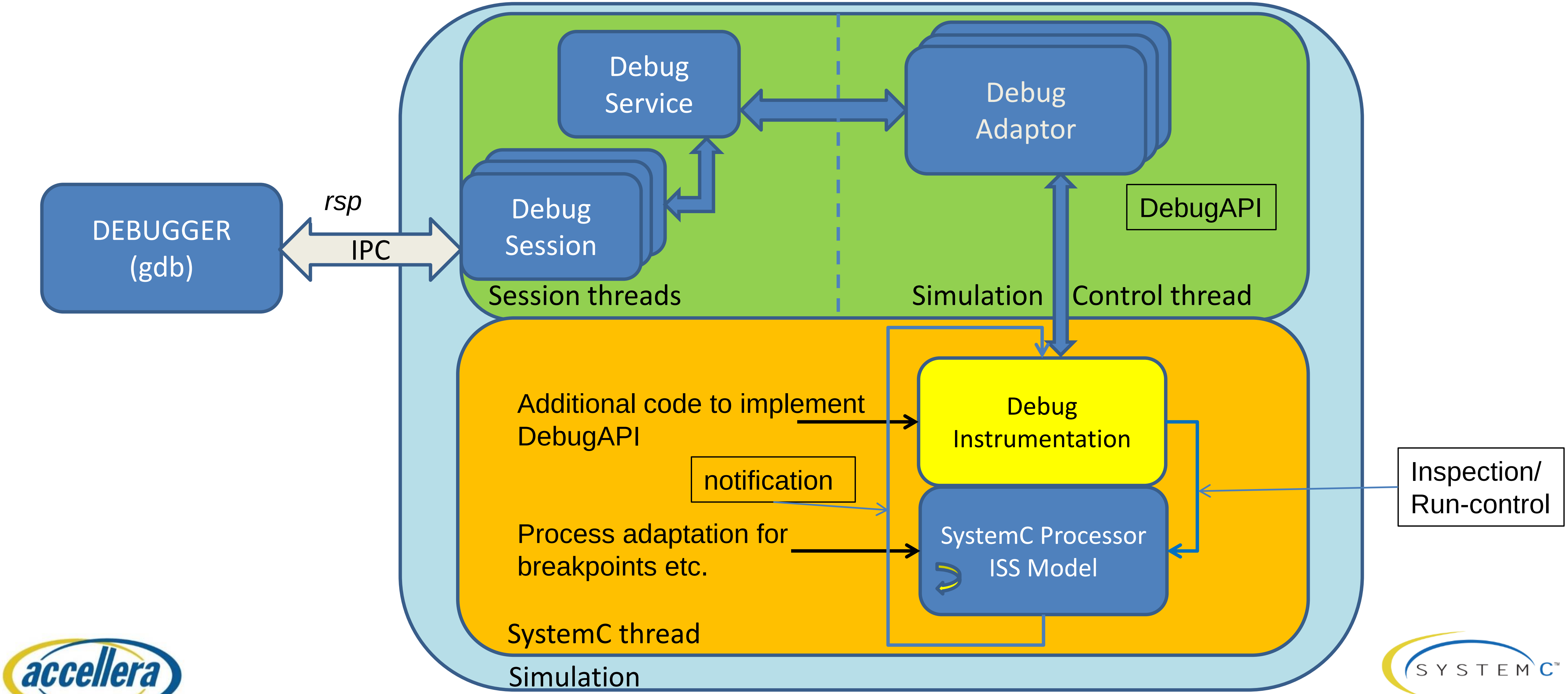
- SimulationControl-thread is responsible for
  - #connections, #simulation_blockers

  - Increase #simulation_blockers on attach/user-break,
    - Pause (suspend) simulation on #simulation_blockers == 1
      ➔ controller will get correct response automatically

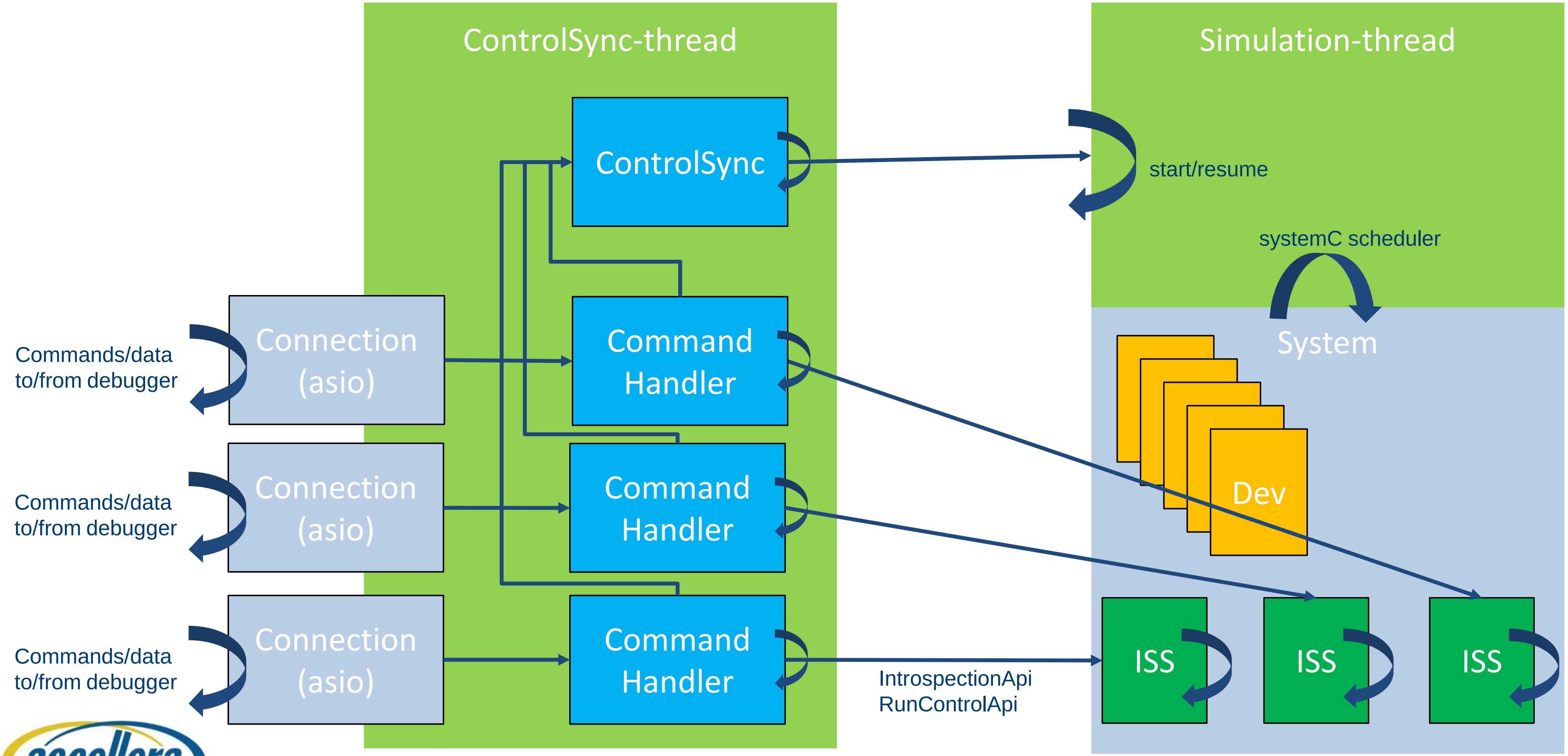
    - If simulation was already paused (suspended)
      ➔ create & send artificial 'interrupted'-response to debugger

  - Decrease #simulation_blockers on continue
    ➔ resume simulation when #simulation_blockers==0

Synchronization control

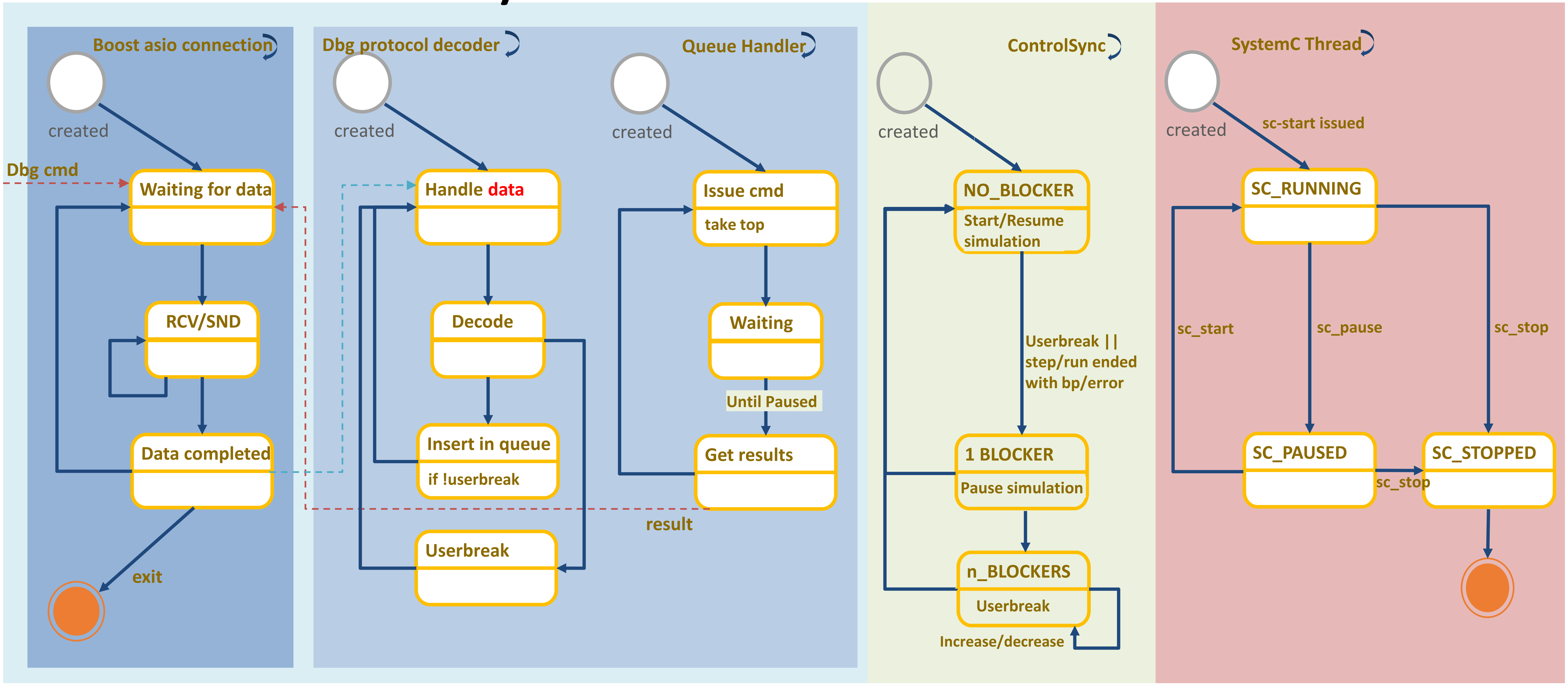


Synchronization control



```
while (true) {
  execute();
  if (condition) {
    sc_pause();
    sc_wait(SC_ZERO_TIME);
  }
}
```

Synchronization control



Synchronization control

Code for sc_main (replacement for sc_start())

```
if (allowDebug) {
    DebugService::getInstance().createMonitors(dbg_port); // create the sessions
    std::thread debugService(debug_task, &DebugService::getInstance().io_service);
    debugService.detach(); // Do not block execution.
}
std::thread systemSimulation(simulation_task, global_quantum_value); // calls sc_start()
systemSimulation.join(); // wait until simulation finishes
if (allowDebug) {
    debugService::getInstance().io_service.stop(); // cleanup resources
}
```

Boost asio

TLM global quantum

Synchronization control

Code for debug task

```
// The function we want to execute on the new thread.
void debug_task(boost::asio::io_service* io_service)
{
    io_service->run();
}
```

Synchronization control

Code for simulation task (simplified)

```
void simulation_task(uint64_t quantum_value) {
    … /* Initialize the Global Quantum Keeper */
    bool stopped(false);
    while (!stopped) {
        stopped = run_sim();
        if (!stopped) {
            // resume again if all controllers want to continue
            ControlSync::instance().waitForCommand();
        }
    }
}
```

Synchronization control

Code for simulation task (simplified, without exception-handling)

```
/* returns false for paused, true for stopped and/or error */
bool run_sim() {
   sc_core::sc_start();
   ControlSync::instance().notifyControllers();
   return (sc_core::sc_get_status() != sc_core::SC_PAUSED);
}
```

Generic applicability

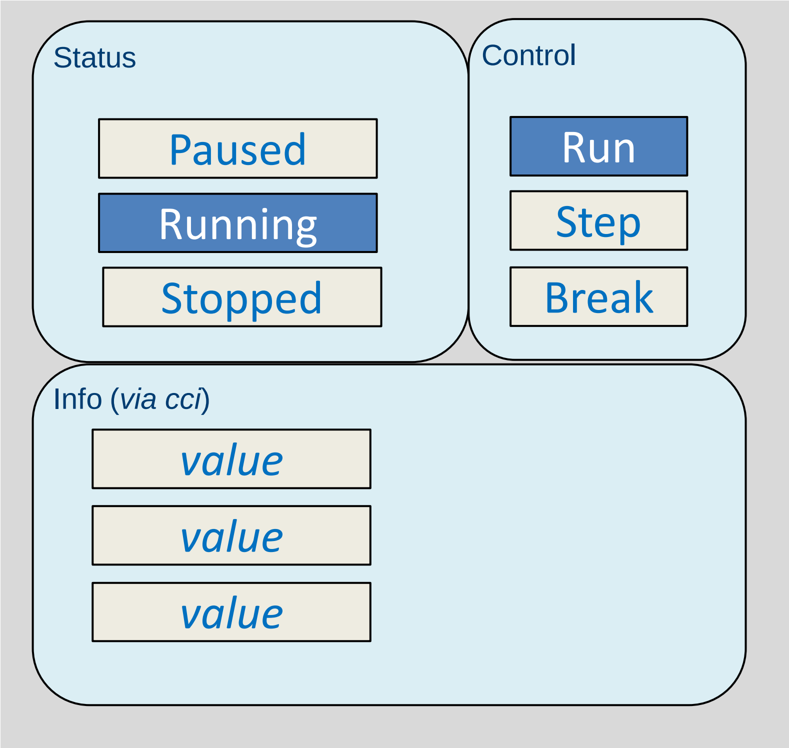
What if

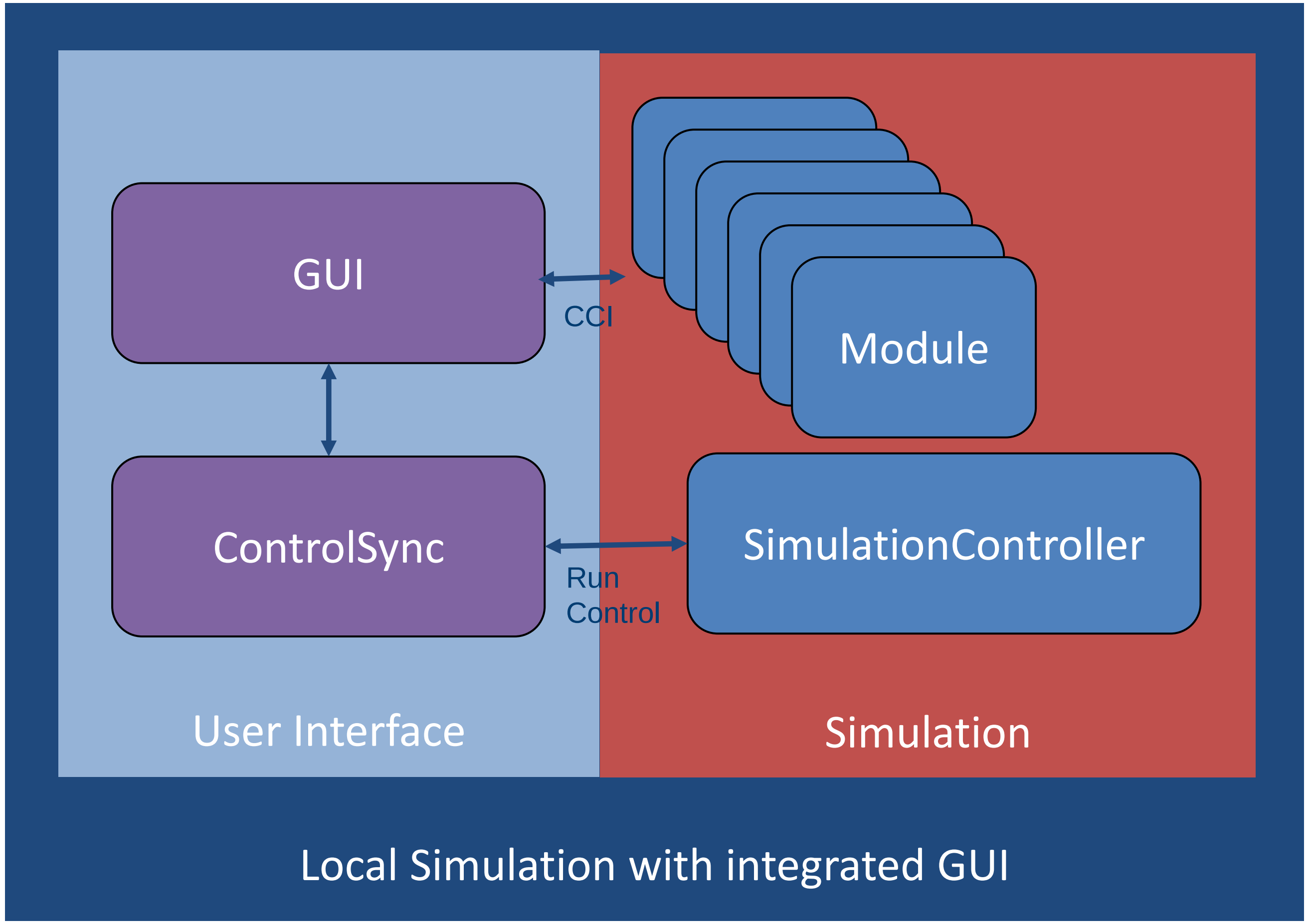
- we use the previous concepts also for simulations without ISS-models?
  - Generic system-simulation controller (api)
  - Replace gdb rsp with remote-cci protocol (tbd)

- we apply the same ideas to hybrid/distributed simulations?
  - State-synchronization across multiple simulators

Generic applicability

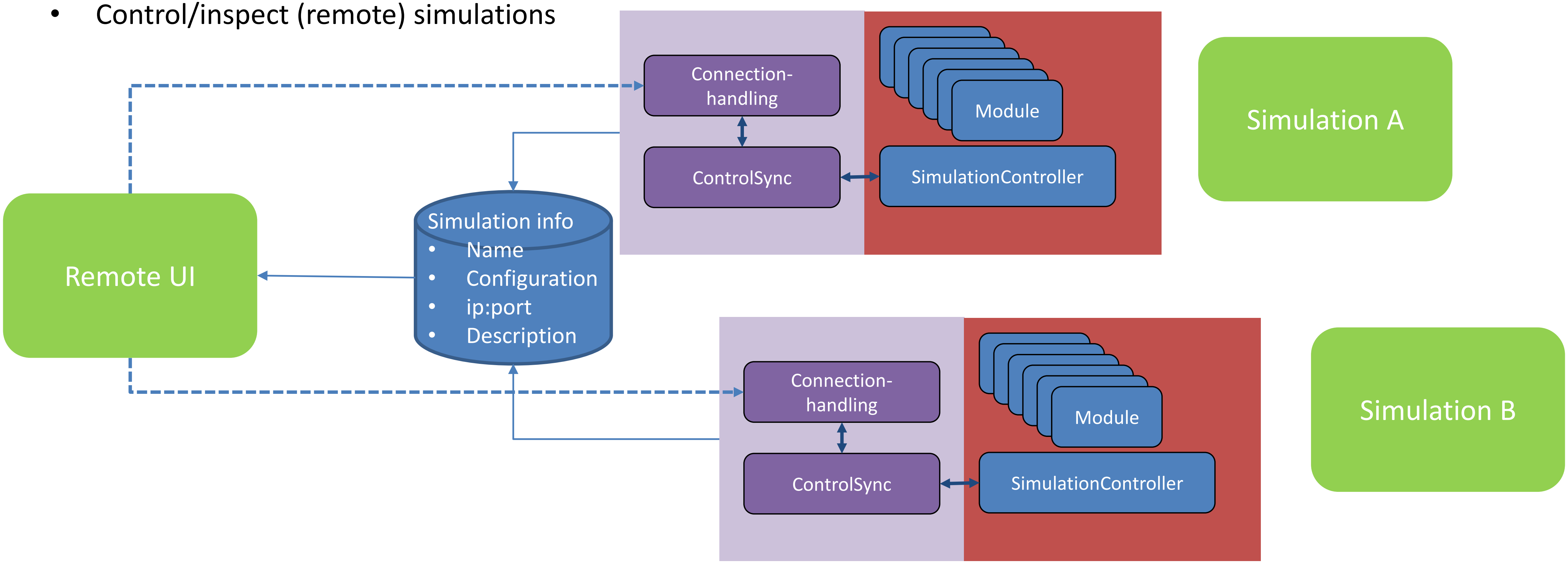
- Simple CCI/Control-GUI for simulation





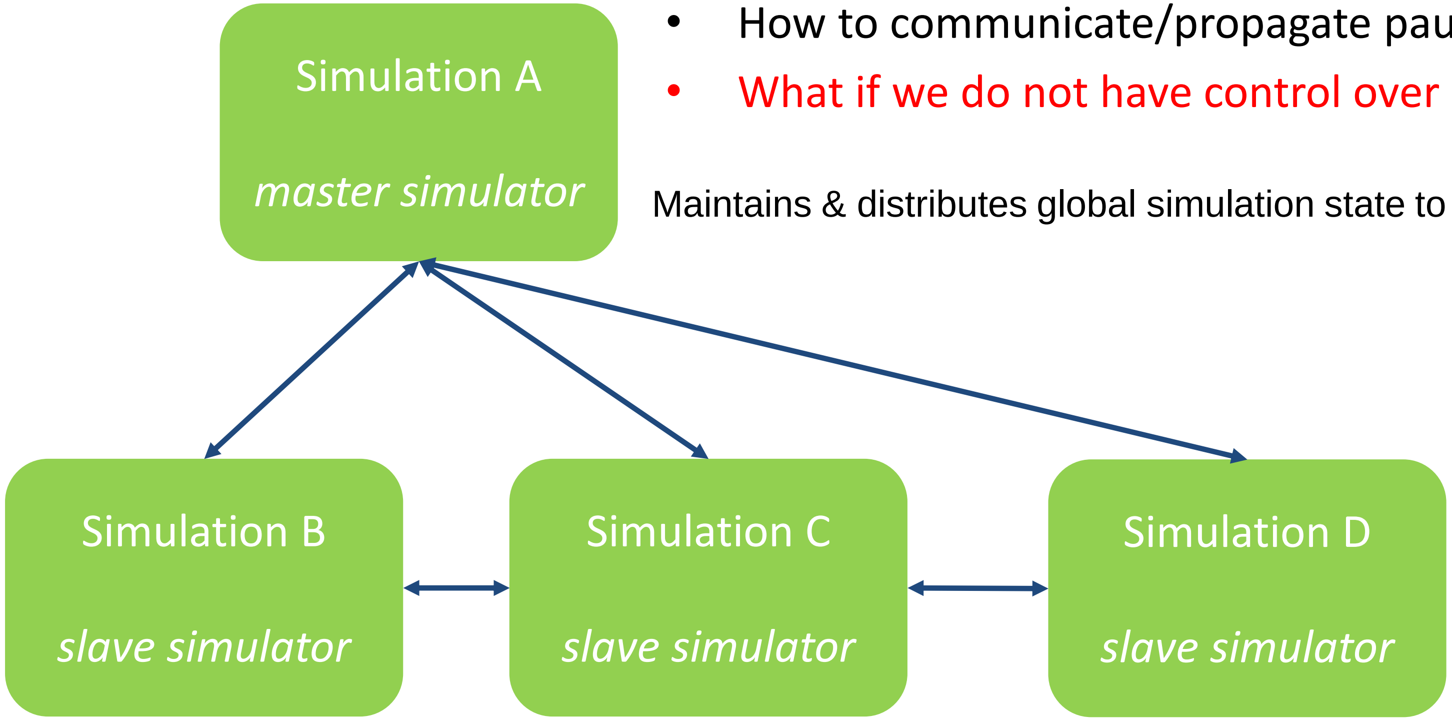
Generic applicability

- Control/inspect (remote) simulations



Generic applicability

- SystemC-simulation combined with other simulator(s): hybrid/distributed simulation
  - Assumption: other simulators have a similar ‘paused’-state
  - How to communicate/propagate paused-state to other simulators?
  - What if we do not have control over sc_main?

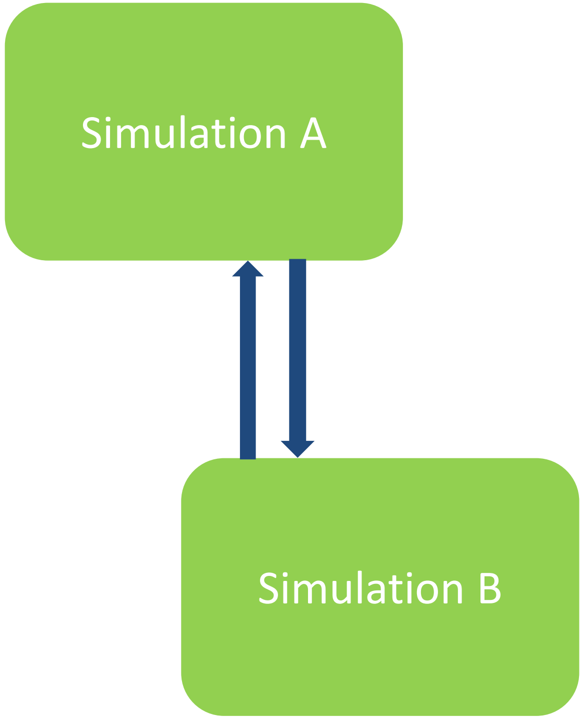


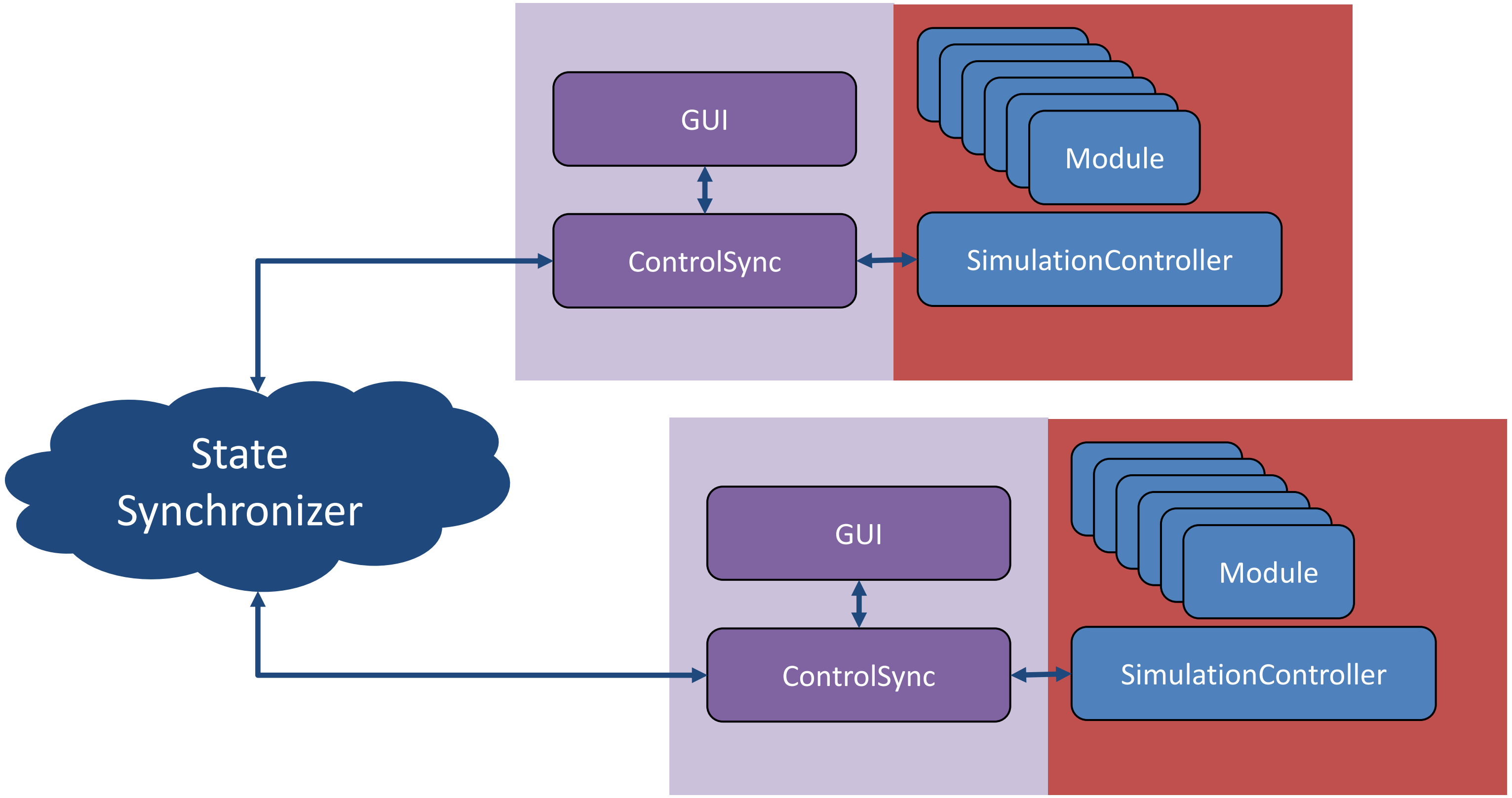
Maintains & distributes global simulation state to slave simulators

Report local simulation state changes to master simulator
Need specialized ControlSync instances in slaves to handle simulation state-control (only master can pause/resume)

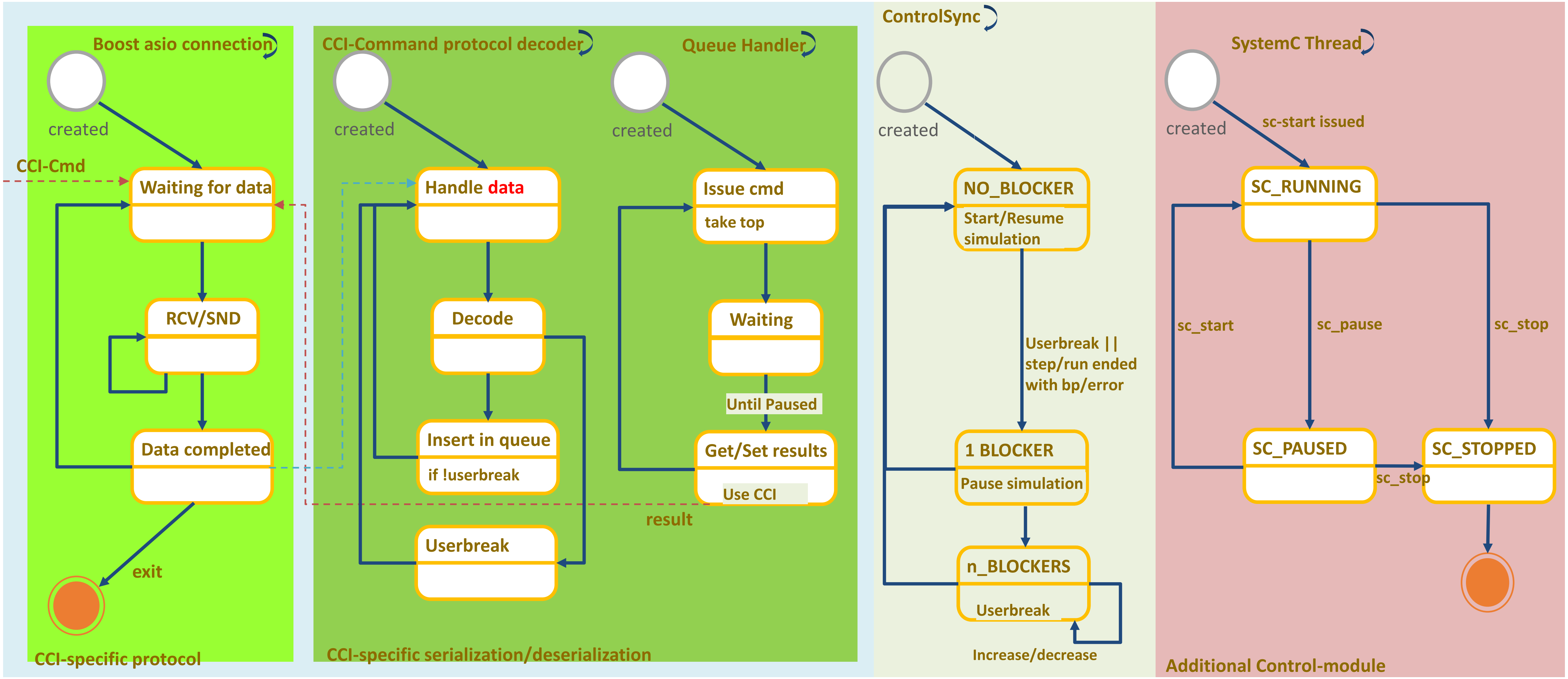
Generic applicability

- Distributed simulation





Generic simulation-control



accellera SYSTEMS INITIATIVE

SYSTEMC EVOLUTION DAY
OCT 28, 2021 | VIRTUAL WORKSHOP

Generic simulation-control

Using generic SystemC-module implementing RunControlApi

```
class RunControlApi {
public:
    /// constructor & destructor
    RunControlApi() {};
    ~RunControlApi() {};
    /// runcontrol
    virtual void attachController()    = 0;
    virtual void detachController()   = 0;
    virtual void step (const sc_core::sc_time& timeStep)  = 0;
    virtual void runUntilBreak()         = 0;
    virtual void userBreak()                 = 0; //< break current run/step
    /// allow controller to inspect simulation before simulation end
    virtual void endOfSimulationEvent()    = 0;
};
```

Generic simulation-control

```
void SimulationControl::main_thread() {
    while (true) {
        wait(m_attachEvent);
        while (m_controller) {
            if (m_stepping)  wait(m_stepTime, m_detachEvent|m_userBreakEvent);
            else wait(m_detachEvent|m_userBreakEvent);
            if (m_controller) {
                ControlSync::instance().controlBreak(this); sc_pause(); wait(SC_ZERO_TIME);
            }
        }
    }
}
```

Conclusion

- Current implementation (using sc_pause-mechanism) implements all requirements under condition that:
  - We have full control over sc_main implementation
  - No distributed/hybrid simulation scenarios are required
- In case a model is integrated by someone else, we cannot use this solution
  - We have no control over sc_main implementation,
    or it is not even used (running under direct control of the kernel, section 4.3.5 IEEE-SystemC)
    Can we use sc_pause in absence of sc_start/sc_main? It would seem not..
- To become a full solution, we need some changes

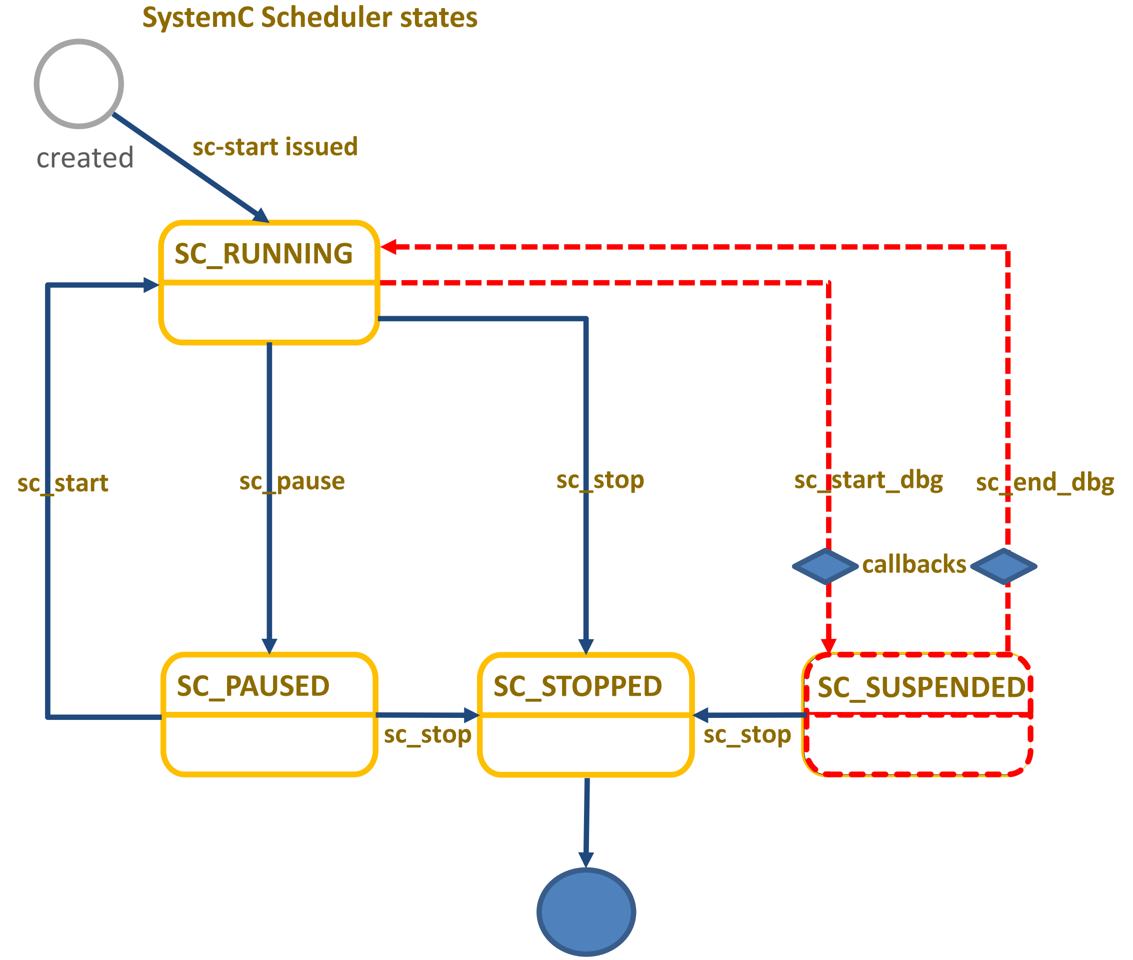
Proposal & Discussion

- Extend kernel scheduler state-machine with additional state SC_SUSPENDED
  - Like SC_PAUSED, but does not return to sc_main
  - Enable callbacks on transitions to/from SC_SUSPENDED
    to enable messaging to other simulators

- New api functions
  - sc_start_debug()/sc_end_debug()
    == sc_suspend_all()/sc_unsuspend_all with priority level

- Adapt implementation of ControlSync to use new api
  - Similar way as proposed in ‘(Un)Suspend(able)’) (patch merged 09/21/2021)
  - Prepare patch

Proposal & Discussion


SystemC Scheduler states
created
sc-start issued
SC_RUNNING
sc_start
sc_pause
sc_stop
sc_start_dbg
sc_end_dbg
callbacks
SC_PAUSED
SC_STOPPED
SC_SUSPENDED
sc_stop
sc_stop