Simulation Integration Manager
Version 0.1.0

IllinoisRocstar LLC

October 10, 2016

www.illinoisrocstar.com

License

The software package sources and executables referenced within are developed and supported by Illinois Rocstar LLC, located in Champaign, Illinois.The software and this document are licensed by the University of Illinois/NCSA Open Source License (see `opensource.org/licenses/NCSA`). The license is included below.

For more information regarding the software, its documentation, or support agreements, please contact Illinois Rocstar at:

- **tech@illinoisrocstar.com**
- **sales@illinoisrocstar.com**

0.1 Overview

The *Simulation Integration Manager* (*SIM*) is an orchestrator implementation in the *IMPACT* suite. It is the front-end of the code that directly interacts with end-developers and end-users of coupled simulations. *SIM* is a higher-level infrastructure, built on top of the *COM* integration framework, but it is possible to use it independently of *COM*. The overall objectives of *SIM* are to provide **high-level abstractions** of the operations in multiphysics coupling, facilitate **flexible** and **easy construction** of complex coupling algorithms from predefined building blocks, and deliver **readability** of the orchestration module.

0.2 Capabilities

The next-generation of *SIM* will support the following capabilities, and allow potential extensions:

- different types of applications
  - unsteady-state calculations (with time-marching schemes)
  - steady-state calculations
- different types of coupling (from simplest to most complex)
  - stand-alone fluid or solid
    - with or without combustion and flexible control of initial ignition
    - with or without surface regression and time zooming
    - thermal boundary conditions, chemical reactions, etc.
    - with or without propagation constraints
  - fully-coupled fluid-solid interaction
    - with or without predictor-corrector iterations
    - with or without subcycling
    - flexible execution order (e.g., alternating orders and concurrent executions)
  - fully-coupled fluid-solid-combustion interaction
    - all suboptions listed above
- numerical and geometrical capabilities
  - mesh modification
  - surface propagation
  - sliding interfaces
- infrastructure runtime support
  - flexible, user-friendly control mechanism in choosing different options
  - concurrent execution of independent tasks, possibly on different sets of processors
  - exact restart in all coupling modes
  - asynchronous I/O using different file formats, and separation of visualization and restart files

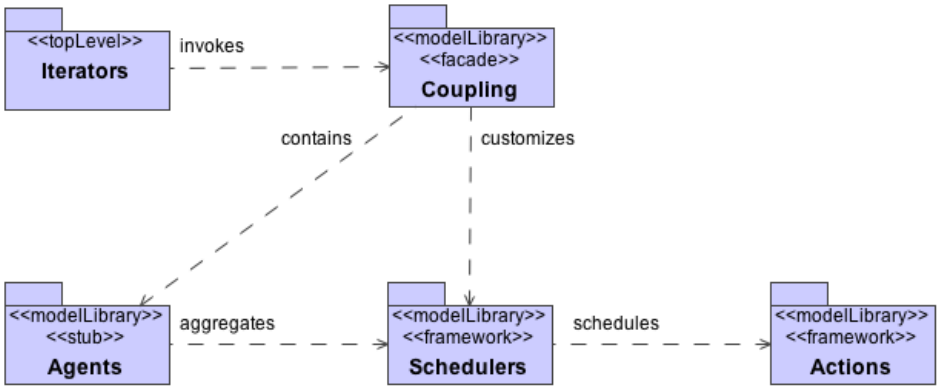

Figure 1: Overview of system architecture of *SIM*.

0.2.1 Design Features

To facilitate the diverse needs of different applications and coupling schemes, we design a new software infrastructure for *SIM*, which has the following features:

- Tiered architecture: the layers are *top-level iterations*, *coupling schemes* (with or without predictor-corrector iterations), *agents* for physics modules, and manipulation of *jump conditions*;
- Action-centric specification: coupling schemes are described as *actions*, with well-defined input and output to specify the *data flow*;
- Automatic scheduling: based on the data flow among actions, the *control flow* is derived automatically to determine the scheduling of actions by the runtime system, which allows potential concurrent execution of actions;
- Visual aid: *SIM* will provide *visualization of data flow* of actions, to help users comprehend and debug coupling algorithms.

0.3 System Architecture

SIM contains five types of key components, as depicted in Figure Figure 1: top-level iterations, coupling schemes (with or without predictor-corrector iterations), agents for physics modules, schedulers, and actions. We will explain these components in the following subsections.

0.3.1 Top-level Iterations

Time-marching Schemes For unsteady-state calculations, the time-marching scheme is a simple driver code, which can be used with different types of coupling schemes. The pseudo-code in Procedure 1 outlines the time-marching procedure, whose core is the coupling schemes.

Steady-state Iterations Steady-state calculations are sometimes used in fluid-solid interactions and for stand-alone simulations, and hence are desirable features for *SIM*. The pseudo-code in Procedure 2 outlines the top-level iterations of a steady-state calculation, which is very similar to the time-marching scheme.

Procedure 1 Top-level time-marching scheme.

construct coupling object
invoke input of coupling object
invoke initialization of coupling object
while not yet reached designated time **do**
  invoke time integration of coupling scheme by passing in current time and obtaining new time
  if reached time for restart dumps **then**
    invoke restart output (as well as visualization data) of coupling scheme
  else if reached time for visualization dump **then**
    invoke visualization output of coupling scheme
  end if
end while
invoke finalization of coupling object

Procedure 2 Top-level iteration for steady-state calculations.

construct coupling object
invoke input of coupling object
invoke initialization of coupling object
while not yet converged **do**
  invoke solver of coupling scheme
  if reached stage for restart dumps **then**
    invoke restart output (as well as visualization data) of coupling scheme
  else if reached stage for visualization dump **then**
    invoke visualization output of coupling scheme
  end if
end while
invoke finalization of coupling object

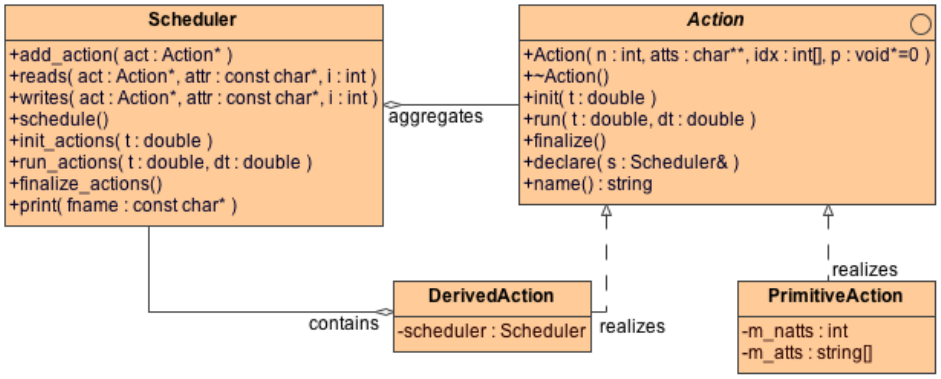

Figure 2: Overview of actions and schedulers.

0.3.2 Actions and Schedulers

As depicted in the class diagram in Figure 2, actions and schedulers are interdependent on each other. An *action* is a functional object. There are two types of actions: *primitive* actions, which perform some basic calculations, and *derived* actions, which are composed of collections of subactions. An action has a constructor, and three core procedures: init(), run(), and finalize(), for performing initialization, execution, and finalization, respectively. The constructor typically takes a number of DataItem names and their corresponding *time indices* as arguments. The time indices are important to denote whether the DataItem is from a previous iteration of the top-level loop. Optionally, the constructor may also take a void pointer as its final "wildcard" argument, which can point to a specific structure to encapsulate the additional input and output of the action.

The initialization procedure allocates the intermediate data needed to perform the action, and finalization deallocates these intermediate data. In general, when the constructor is invoked, the DataItem names listed in the arguments may not have been associated with actual DataItems; when initialization is invoked, they will have been associated. Depending on whether the action is primitive or derived, its execution procedure performs some basic calculations, or invokes the execution of subactions. In addition, an action has two procedures for interacting with schedulers: declare(), which registers the read and write operations performed on the DataItems passed into the constructor, and name(), which provides a descriptive name for the action for visualization.

A *scheduler* is a container of actions, and is responsible for determining the orders of initialization, execution, and finalization of its actions. It is also part of a derived action for scheduling the subactions. A scheduler provides a procedure add_action() to its user for registering actions. The scheduler invokes the declare() member function of an action when it is being added, and declare() registers data access with the scheduler by calling the reads() and writes() member functions of the scheduler. After all the actions have been registered with a scheduler, it then constructs a directed acyclic graph (DAG) for the actions, in which each edge is identified by a DataItem name and its corresponding time index. Typically, the pair of DataItem and time indices should identify one unique provider, but a DataItem may be used by multiple actions.

If a DataItem has multiple providers, the scheduler will try to identify the actual provider using other dependencies, and execution will abort if this identification process fails. The schedule() member function determines the orders of initialization, execution, and finalization of its actions, and recursively invokes the scheduling of the derived actions. In a parallel computation, the schedulers on all processes must return the same orders. The member functions init_actions(), run_actions(), and finalize_actions() invoke the

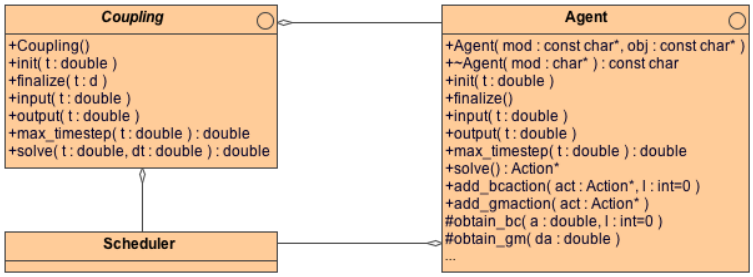

Figure 3: Overview of agents and coupling schemes.

corresponding procedures of the actions. Furthermore, the scheduler provides a print() function, for printing out the DAG and its scheduling for visualization.

In general, an action is present in at least one scheduler, and typically is in only one scheduler. The user (such as a derived action) of a scheduler should create actions by calling the C++ new operator, but the owner schedulers of the actions will be responsible for deleting the actions when they are no longer in use. If an action is used by multiple schedulers, the user must ensure the action is *reentrant*, in the sense that its initialization and finalization procedures do not allocate and deallocate intermediate data multiple times, and it is safe for multiple instances of the action to run concurrently.

0.3.3 Agents and Coupling Schemes

An *agent* serves a physics module. The most basic task of an agent is to load a physics module ComponentInterface (CI) into *COM* in the constructor. A more critical set of tasks of an agent is to manage the persistent buffer data on behalf of the physics module. These data are typically defined on the mesh of the physics module, may be shared by multiple actions, and need to be saved for visualization or restart. The constructor of an agent defines the buffer data, and its initialization procedure is responsible for allocating these buffer data, after invoking the initialization of the physics module.

The finalization of an agent deallocates these buffers and invokes the finalization of the module. Since these buffer data may need to be read and written along with the data of the physics module, the agent is responsible for providing the input and output functions, to be called by the coupling schemes.

In addition to providing data services for a physics module, the agent also interacts with scheduler on its behalf, and and feeds boundary conditions to the physics module. An agent usually provides one action to coupling schemes, for invoking the solver of the module. It also provides a function for obtaining the maximum time step constrained by the module.

For supply boundary conditions, it provides two callback procedures to the physics module, one for obtaining boundary conditions, and the other for obtaining grid motion. These procedures may be called by the physics module during its iterations of calculation (such as subcycling or Runge-Kutta iterations) without returning the control back to the coupling scheme. These callback procedures are implemented as derived actions in the agent, whose subactions are defined by the coupling schemes and registered by calling add_bcaction() and add_gmaction(), respectively. The scheduling of these callback procedures should be performed when the solve action is being scheduled by the main scheduler.

A *coupling scheme* is composed of a number of agents and a scheduler. Its constructor invokes the constructors of the agents and the scheduler to define the coupling algorithm, and then determines the orders that must be followed for invoking initialization, execution, and finalization. The initialization (init) and

Procedure 3 Coupling with predictor-corrector iterations.

```
for i = 1 to maxiter do
  invoke time integration of base coupling scheme
  check convergence
  if converged then
    store current solution
    break from loop
  else
    restore previous solution
  end if
end for

if i ≥maxiter and not converged then
  throw exception
end if
```

finalization (finalize) procedures of the coupling scheme initialize and finalize the agents and actions in the scheduled order, respectively. Its execution (solve) procedure takes the current time and the pre-determined time step, runs the actions, and then returns the new time. The coupling scheme also provides interfaces to the top-level iterators for input, output, and obtaining the maximum time step, which invoke their counterparts of the agents.

0.3.4 Predictor-corrector Iterations

Coupling with predictor-corrector (PC) iterations generalizes the basic coupling scheme described above. Besides the standard actions, coupling with PC iterations requires an inner loop to repeat the base coupling algorithm, until the interface conditions have converged, or a maximum number of iterations have been reached. Its execution requires some additional services, such as checking convergence and storing and restoring data. Procedure 3 describe the sequence of such a PC iteration, in which different base coupling schemes can be plugged.

0.4 Predefined Actions

To support implementations of coupling schemes, we define four types of actions: the execution of a physics module, interpolation of boundary conditions, manipulation of jump conditions, and operations for supporting PC iterations.

.

0.4.1 Solve

This action is physics dependent, and is defined in an agent. It takes the input and output of the physics module as the DataItem lists of the arguments, and takes a pointer to the parent agent as its wildcard argument.

0.4.2 Interpolate

This action in general handles interpolation or extrapolation in time for one DataItem. Except for time zooming, it is the sole building block for update_bc() and update_gm() functions of an Agent instance. There can be different types of interpolation schemes. In general, its constructor takes following arguments:

- The interpolation scheme, passed in as the wildcard argument, which can be one of the EXTRAP_LINEAR, INTERP_LINEAR, EXTRAP_CENTRAL, INTERP_CENTRAL, INTERP_CONST;
- Three DataItem names: current data, interpolated data, and old data. The old DataItem is optional and needed only for INTERP_CENTRAL and INTERP_LINEAR. For INTERP_LINEAR and EXTRAP_LINEAR, the action may construct a gradient DataItem internally.

0.4.3 Jump Conditions

These are the high-level abstractions of the jump conditions in each type of coupling, such as motion transfer, load transfer, heat transfer, mass transfer, momentum transfer, etc..

0.4.4 Actions for PCCoupling

PCCoupling requires some special services, including checking convergence and storing and restoring data. These special services are defined as protected classes within PCCoupling. These actions are manually scheduled and are called directly by the Procedure 3, instead of by a scheduler.

PCService PCService serves as the base class for the implementation of the other services in this subsection. Its constructor takes a list of DataItem names. The given DataItems may or may not belong to the same user window. The init() operation groups the DataItems received by the constructor based on their owner windows, and creates one buffer window for each owner window by using the connectivity tables (to replicate the panes) and then cloning the listed DataItems in it. The buffer data created will not be saved for restart. The action provides a protected utility function copy(dir) to help the implementation of subclasses. The function copies data DataItems from user windows to buffer windows or vice versa, depending on the argument.

CheckConvergence The class CheckConvergence is a subclass of PCService, in charge of managing memory space for backing up user-specified interface quantities, to check whether PC iterations have converged. Besides inheriting the init() operation from PCService, it has two public interface functions:

- set_tolerance(attr, tol, norm): sets the tolerance of a given DataItem for a specific norm.
- check(ipc) takes the current iteration index of the PC-iterations as input, and compares the norms of the DataItems against the preset tolerances. If all tolerances are met, it returns true; otherwise, it copies the current solution for later comparison and then returns false.

Backup The class Backup is also a subclass of PCService, in charge of managing memory space for backing up user-specified data DataItems after PC iterations have converged, and recovering data if not yet converged. Besides inheriting the init() operation from PCService, it has two public interface functions:

- store(): copies the data from CI windows to buffer windows;
- restore(): copies data from buffer windows back to CI windows.

0.5 Schedulers

0.5.1 Sequential

The simplest type of scheduler is to schedule the operations of the actions based on the order of their registration. It does not require constructing DAGs. It is particularly convenient for implementing simple derived actions, and should suffice for many simpler coupling schemes.

0.5.2 Concurrent

More sophisticated schedulers can allow execution of multiple independent actions concurrently and on different sets of processors. The concurrent scheme requires construction of DAGs, and may make use of sophisticated scheduling algorithms for optimal performance.

0.5.3 Interprocess

The most sophisticated schedulers can allow execution of independent actions to run on different processes, and potentially allow migration of actions.

0.6 Predefined Agents

Agents represent each of the physics modules which are typically written in a separate library in Fortran 90. Agents provide support to initialize the physics modules and drive their simulations. The base Agent class implements common features required by all physics modules including file I/O (using *SimIO*), and initialization of *COM* function handles.

In general, a given physics code will present a CI with a set of windows. The functions and data presented in application-specific CI are, in general, arbitrary. This situation necessitates multiple Agents. An application-specific Agent derives from a domain-specific Agent and uses the application-specific CI to present the desired interface to the *SIM* coupling constructs. Domain-specific agents are discussed below.

0.6.1 Fluid agent

The class FluidAgent is designed to present a computational fluid dynamics (CFD) interface that can be used in the advanced *SIM* coupling constructs. An arbitrary CFD application publishes its native data and functions through the CI window, and an application-specific Agent derives from FluidAgent and massages the application's CI so that it conforms. Together, the application-specific Agent and the FluidAgent create all necessary *COM* windows for boundary condition data that are used in coupled simulations. It defines the subroutine to write restart data files using *SimIO/SimOUT*. It also creates window for convergence check.

0.6.2 Solid agent

The class SolidAgent is designed to present a computational structural mechanics (CSM) interface that can be used in the advanced *SIM*. It creates all necessary *COM* windows for boundary condition data that are used in coupled simulations. It defines the subroutine to write restart data files using Rocout. It also compute integrals for conservation check.

0.6.3 Burn agent

The class BurnAgent is designed to work with transient thermal solvers, ignition, and burn-rate providers. It creates all necessary *COM* windows for boundary condition data that are used in coupled simulations. It defines the subroutine to write restart data files using *SimIO*.

0.7 Predefined Coupling Schemes

The class Couple implements the basic functions of a coupling scheme, which is composed of a number of agents and a scheduler. The definition of a particular coupling scheme is defined in the constructors of a derived Couple class. The physics details of each coupling scheme are defined in an (upcoming) manual - Numerical Coupling Interface in *SIM*.

0.7.1 Fluid-alone

Fluid alone simulation includes fluid only with or without combustion.

0.7.2 Solid-alone

Solid alone without combustion. It is implemented in derived class SolidAlone.

0.7.3 Fluid-solid interaction

Fluid solid interaction includes both fluid and solid modules, but without combustion. The following derived classes are implemented: SolidFluidSPC for solid, fluid, no combustion, simple staggered scheme with P-C; SolidFluidBurnSPC for solid, fluid, combustion and simple staggered scheme with P-C; FluidSolidISS for fluid, solid and no combustion using improved staggered scheme.

0.7.4 Fluid-solid-combustion interaction

Similar to fluid-solid interaction but with combustion. It is implemented in derived class SolidFluidBurnSPC for simple staggered scheme with P-C, and SolidFluidBurnEnergySPC with simple staggered scheme with burn energy.