SIMULIA

3DEXPERIENCE

ABAQUS 2026 FD04
COUPLED SIMULATION INTERFACE GUIDE

DASSAULT SYSTEMES

Contents

Trademarks and Legal Notices....4
Coupled Simulation Interface....5
....6
Introduction....7
About SIMULIA Co-simulation Services....8
SIMULIA Co-Simulation Services....8
Typical Applications....8
Intended Users....8
Licensing....9
SIMULIA Co-Simulation Services Interactions....9
SIMULIA Co-Simulation Services API Kit....10
SIMULIA Co-Simulation Services API Kit Contents....10
Release Compatibility....11
Extracting the SIMULIA Co-Simulation Services API Kit....12
Compiling and Linking the SIMULIA Co-Simulation Services....13
Compiling your Application with SIMULIA Co-Simulation Services....13
Linking the SIMULIA Co-Simulation Services to Your Application....13
Running a Coupled Simulation....14
Setting up the Runtime Environment....14
Starting the SIMULIA Co-Simulation Services Director....14
Technical Overview....16
SIMULIA Co-Simulation Services Director....17
Communication Services....18
Serial vs. Parallel SIMULIA Co-Simulation Services....19
Embedding the SIMULIA Co-Simulation Services in a Serial Manner....19
Embedding the SIMULIA Co-Simulation Services in a Parallel Manner....19
Mapper....20
Domain Coupling....20
Neighborhood Search....20
Mapping Algorithm....21
Synchronization....23
Coupling Algorithms....24
Explicit Coupling Schemes....24
Implicit-Iterative Coupling (Basic)....25
Implicit-Iterative Coupling with Interface Treatment....25
Proprietary Coupling Algorithms....26
SIMULIA Co-Simulation Services Configuration File....27
About the SIMULIA Co-Simulation Services Configuration File....28
The Configuration File Content....28
Configuration File Consistency Requirements....28
Configuration File Elements....30
The Root Element of the Co-Simulation Configuration File....31

Components....32
Component Instances....34
Connectors....36
Connections and Connection Sets....38
Execution....39
Mapping Specification....50
Modulations....52
Transformation....53
Field Operations....54
SIMULIA Co-Simulation Services API....55
Core Services....56
Initialization and Finalization....57
Establishing connection to and disconnecting from the SIMULIA Co-Simulation Services Director....59
Defining the Interface Model....61
Control API....70
Data API....74
Message Handler....77
Add-On Services....79
System-Level Modeling....80
Parallel Interface Definition....82
Importing a Remote Client's Mesh....86
General Services for Advanced Coupling Algorithms....89
Results Provider....90
Results Provider....91
Developing a Results Provider....91
Results Provider Template....91
Release Compatibility....91
Launching the Results Provider....91
Services for Results Provider....94
Data Model....94
Registering the Callback Services....94
Querying Required Fields....94
Results Provider Control API....94
Results Provider Data Services....95
Deep Dive into the Results Provider Template....96
Parsing the Command Line Arguments....96
Opening the Results File and Defining Your Database....97
Initializing the SIMULIA Co-Simulation Services....97
Establishing the Connections....97
Registering the Sequential Callback Services....98
Registering the Interface Model....98
Extracting and Sending Field Data....98
Disconnecting and Finalizing the SIMULIA Co-Simulation Services....99
putFieldFnWithTime....99
putFramePtsForField....100
Examples....102
Dual Mass-Spring Systems....103

Problem Description....103
Compiling the C++ Simulators....104
Linking the Simulators....104
Running the Coupled Simulation Using Two C++ Simulators....104
Results....105
Files....106
Aeroelastic Supersonic Flow over Long, Thin-Walled Cylinders....108
Problem Description....108
Third-Order Piston Theory....109
Compiling the Third-Order Piston Simulator....110
Linking the Third-Order Piston Simulator....110
Running the Coupled Simulation....110
Results....111
Files....111
References....112
ASCII File Results Provider....113
Problem Description....113
Compiling the Results Provider....114
Linking the Results Provider....114
Executing the Sequential Simulation....114
Results....115
Files....115
Appendix: Predefined Field Types....116
Mechanical Fields....116
Thermal Fields....116
Electrical Fields....117
Pore Pressure Fields....117
General Fields....117

Trademarks and Legal Notices

Trademarks

Abaqus, **3D**EXPERIENCE®, the 3DS logo, the Compass icon, IFWE, 3DEXCITE, 3DVIA, BIOVIA, CATIA, CENTRIC PLM, DELMIA, ENOVIA, GEOVIA, MEDIDATA, NETVIBES, OUTSCALE, SIMULIA and SOLIDWORKS are commercial trademarks or registered trademarks of Dassault Systèmes, a European company (Societas Europaea) incorporated under French law, and registered with the Versailles trade and companies registry under number 322 306 440, or its subsidiaries in the United States and/or other countries. All other trademarks are owned by their respective owners. Use of any Dassault Systèmes or its subsidiaries trademarks is subject to their express written approval.

Legal Notices

Abaqus and this documentation may be used or reproduced only in accordance with the terms of the software license agreement signed by the customer, or, absent such agreement, the then current software license agreement to which the documentation relates.

This documentation and the software described in this documentation are subject to change without prior notice.

Dassault Systèmes or its Affiliates shall not be responsible for the consequences of any errors or omissions that may appear in this documentation.

© Dassault Systèmes Americas Corp., 2026.

For a full list of the third-party software contained in this release, please go to the Legal Notices in the Abaqus 2026 HTML documentation, which can be obtained from a documentation installation, or in the SIMULIA Established Products 2026 Program Directory, which is available on *www.3ds.com*.

Coupled Simulation Interface

The Coupled Simulation Interface guide describes the SIMULIA Co-Simulation Services that enables you to perform coupled multiphysics and multiscale simulations between your application and the Dassault Systèmes physics solvers.

This guide is a part of the Abaqus® documentation collection, which describes all the capabilities of the SIMULIA Co-Simulation Services. It is intended for developers who want to embed the SIMULIA Co-Simulation Services into their applications. End users who perform coupled simulations might find the technical information in this guide useful.

This guide covers all aspects of SIMULIA Co-Simulation Services and consists of the following:

Introduction	Provides an introduction to the SIMULIA Co-Simulation Services describing its intended users, licensing requirements, installation, and how to get started.
Technical Overview	Provides a technical overview about the SIMULIA Co-Simulation Services' software services.
Configuration File	Provides essential information for understanding the SIMULIA Co-Simulation Services Configuration file. The Configuration file describes the configuration of the coupled problem, and is required by the SIMULIA Co-Simulation Services Director.
SIMULIA Co-Simulation Services API	Provides a reference to the SIMULIA Co-Simulation Services Application Programming Interface.
Developing a Results Provider	Provides details about developing a Results Provider. Results Providers can read your results file and transfer and map field data to Abaqus for sequential simulations.
Examples	Provides coding examples of embedding the SIMULIA Co-Simulation Services API into an application.

What's New

This page describes recent changes in Coupled Simulation Interface.

2026 FD04

PDF Guide Available

You can download a PDF version of this guide from the guide overview.

Benefits: You can easily access the PDF version of this guide.

For more information, see *Coupled Simulation Interface*.

2026 FD03

New Guide

This is the first release of the Coupled Simulation Interface guide.

Benefits: The Coupled Simulation Interface guide describes the SIMULIA Co-Simulation Services that enables you to perform coupled multiphysics and multiscale simulations between your application and the Dassault Systèmes physics solvers.

Introduction

SIMULIA Co-Simulation Services provides code-coupling technology for performing multiphysics and multiscale simulations between your application and the Dassault Systèmes physics solvers.

In this section:

- *About SIMULIA Co-simulation Services*
- *SIMULIA Co-Simulation Services API Kit*
- *Extracting the SIMULIA Co-Simulation Services API Kit*
- *Compiling and Linking the SIMULIA Co-Simulation Services*
- *Running a Coupled Simulation*

About SIMULIA Co-simulation Services

Overview

SIMULIA Co-Simulation Services provides code-coupling technology for performing multiphysics and multiscale simulations between your application and the Dassault Systèmes physics solvers.

SIMULIA Co-Simulation Services

SIMULIA Co-Simulation Services is an open architecture enabling coupled simulation among any number of applications, including physical systems and logical systems. SIMULIA Co-Simulation Services is built on a modern distributed memory parallel architecture for high-performance computation. It provides services for efficient communication, field mapping between meshes, coupling algorithms, and field operations (for example, unit conversion, modulations, and transformations).

SIMULIA Co-Simulation Services is widely deployed with the Dassault Systèmes physics solvers and with partner solvers. It supports the Functional Mockup Interface 2.0 for coupling to logical systems. In addition, Fraunhofer SCAI (makers of the Mesh-based Parallel Code Coupling Interface, MpCCI) provides an adapter that indirectly links solvers using the SIMULIA Co-Simulation Services with MpCCI compliant solvers.

Typical Applications

SIMULIA Co-Simulation Services extends the simulation capabilities of a particular physics application by coupling solutions with other physics applications either sequentially or concurrently.

SIMULIA Co-Simulation Services:

- provides a general capability for importing fields into a sequential coupled analysis through the use of Results Providers;
- can be used to solve complex multiphysics simulations, such as fluid-structure interaction, by concurrently coupling physics domain solvers;
- can be used to solve multi-domain simulations, such as coupling implicit and explicit dynamic solvers or coupling finite element analysis with multi-body systems;
- can be used for system-level modeling between logical and physical systems (for example, coupling Abaqus with Functional Mockup Unit, FMU, representing a control system); and
- can be used for submodeling simulation, importing results from a global analysis into a local analysis.

Intended Users

SIMULIA Co-Simulation Services is intended for the following users:

- Software vendors to couple their commercial applications to Dassault Systèmes applications.

  Software vendors who develop an interface must join the SIMULIA Alliance Program.
- Customers to couple their in-house solvers to Dassault Systèmes applications.

  Customers who intend to distribute their solution to a third-party must join the SIMULIA Alliance Program to distribute the SIMULIA Co-Simulation Services.

When embedding the SIMULIA Co-Simulation Services in your application, the programmer should:

- have proficient programming skills;
- have an in-depth knowledge of the Dassault Systèmes application being coupled; and
- be familiar with segregated solution methods when performing a co-simulation.

We encourage contacting customer support to discuss your project before embarking on any programming efforts.

Licensing

The SIMULIA Co-Simulation Services Director is a license-managed component. You must install the appropriate license server and features. The number of license tokens consumed depends on the total number of applications involved in the coupled simulation and whether these are Dassault Systèmes applications or third-party applications.

SIMULIA Co-Simulation Services Interactions

For clarity, we define different levels of interactions with the SIMULIA Co-Simulation Services:

End User

The end user interaction represents the interaction an analyst has while conducting a coupled simulation. This interaction might include authoring, launching, and monitoring a coupled simulation.

This guide provides a high-level description of the information that a user might need to provide to conduct a coupled simulation. For detailed information, refer to the application's documentation on its authoring and launching tools.

Client

The client represents one of the applications (for example, physics solver, FMU, or Results Provider) in the coupled simulation, the client's internal data and execution flow, including domain decomposition for parallel-distributed memory applications.

When this guide refers to the client level, we generalize the concepts to most common physics applications. The aim of the SIMULIA Co-Simulation Services is to provide a flexible environment to facilitate interaction between different applications.

API

The SIMULIA Co-Simulation Services API defines the application programing interface.

This guide provides details about the SIMULIA Co-Simulation Services API, configuration file, and software components.

System Level

The system level represents the computational environment (for example, computer platform, operating systems, compilers, queue system, etc.).

This guide provides guidance on the system level; however, you might need to consult your system (operating system, compiler, etc.) manuals to obtain specific information.

SIMULIA Co-Simulation Services API Kit

SIMULIA Co-Simulation Services API Kit Contents

The SIMULIA Co-Simulation Services API Kit includes the following:

SIMULIA Co-Simulation Services Director

The SIMULIA Co-Simulation Services Director controls the coupled simulation. For more information, see *SIMULIA Co-Simulation Services Director*.

The SIMULIA Co-Simulation Services Director executable, `SMACseCtlMain`, is located in the `/bin` folder.

Simulation Runtime Environment

The Simulation Runtime Environment (SRE) is the SIMULIA Co-Simulation Services component embedded into a client. It is responsible for all services including communication, mapping, and field operations.

The SRE consists of dynamic link libraries: `.dll` files on Windows and `.so` files on Linux. The dynamic link libraries are located in the `/lib` folder and are required at run time.

Header files

The header files provide the function signatures of the SIMULIA Co-Simulation Services API and are required for compiling your application.

Table 1 summarizes the header files. The header files are located in the `/include` folder.

XML document style and definitions files

The XML documentation style and definition files are required to perform XSLD transformations of the co-simulation configuration file and are required at run time by the SIMULIA Co-Simulation Services Director.

These files are located in the `/include` folder.

Dictionary files

The dictionary files are required by the SIMULIA Co-Simulation Services Director and the clients to parse XML.

The dictionary files are located in the `/dictionary` folder. You must set the `CATDICTIONARYPATH` environment variable to point to the `/dictionary` folder.

Results Provider

A template file to start to develop a Results Provider is located in the `/resultsProvider` folder. For information about how to develop Results Provider, see *Developing a Results Provider*.

Examples

Examples are provided to help illustrate how to use the SIMULIA Co-Simulation Services API kit. For more information, see *Examples*.

The coding examples are located in the `/examples` folder.

Table 1: Header file descriptions.

Header File	Description
`SMACseApi.h`	Contains the function signatures for the core functionality.
`SMACseApiEnums.h`	Defines the enumerated symbols.

Header File	Description
`SMACseDmp.h`	Defines the interface definition for distributed memory parallel implementation. This file is required when embedding the services in distributed parallel mode.
`SMACseApiSignalInterface.h`	Contains the function signatures for the signal interface class. This file is required when coupling to FMU's.
`SMACseApiMesh.h`	Defines the function signatures for the mesh interface services to import remote meshes. This file is required when providing your own mapping services.
`SMACseApiResultsProvider.h`	Define additional functions for the results provider. This file is required when developing a results provider.
`SMACseApiService.h`	Defines the function signatures for the service interface. The service interface is used to define the advanced coupling algorithm.

Release Compatibility

Third-party applications might use an earlier release of the SIMULIA Co-Simulation Services than the release of the Dassault Systèmes application being coupled. If the communication protocol is compatible between the releases, this should not cause any issues.

The SIMULIA Co-Simulation Services has a versioning mechanism to ensure the communication protocol is compatible between all applications in the coupled simulation. The SIMULIA Co-Simulation Services Director prints out the communication protocol version and issues an error message if one of the clients uses an incompatible protocol. In this case, you must upgrade the libraries.

It is highly recommended to update the SIMULIA Co-Simulation Services with every major release.

Extracting the SIMULIA Co-Simulation Services API Kit

You must extract the SIMULIA Co-Simulation Services API Kit from the Abaqus media.

To extract the SIMULIA Co-Simulation Services API from the Abaqus media, use the **extractapi** execution procedure. The extraction procedure extracts and packages the required libraries for a given Abaqus release on the platform where the command is executed. On Windows, the java path and the dumpbin utility (distributed by Microsoft Visual Studio) are required.

To extract SIMULIA Co-Simulation Services API from the Abaqus media, use the following command:

```
abaqus extractapi cse -jre <path-to-java>
```

Set the java path to the java installation folder without the trailing backward slash and within double quotes if the path contains spaces (for example, `"C:\Program Files\Semeru\jdk-8.0.402.6-openj9"`).

Compiling and Linking the SIMULIA Co-Simulation Services

You must compile and link the SIMULIA Co-Simulation Services to your application.

The SIMULIA Co-Simulation Services API is written in C, which allows you to easily develop interfaces to other programing languages.

Compiling your Application with SIMULIA Co-Simulation Services

The compiler needs to access the SIMULIA Co-Simulation Services header files. Either install the header files into a folder where the compiler can include them automatically, or set the compiler option to the folder that contains the header files. Consult your compiler reference guide for compiler flags to include the header files.

The SIMULIA Co-Simulation Services are compiled using the same compiler flags as Abaqus. If you have Abaqus installed, you can use the following command to list the compile flags that Abaqus uses:

```
abaqus -info environment
```

The variable `compile_cpp` defines the compiler flags that Abaqus uses.

Linking the SIMULIA Co-Simulation Services to Your Application

Common approaches to loading the dynamic libraries at run time include the following:

- Using the `dlopen` function on Linux or the `LoadLibraryA` function on Windows to load the SIMULIA Co-Simulation Services dynamic libraries at run time.

  This process is more involved because you must create function pointers.
- Loading the libraries from a particulate folder at run time (for example, the Abaqus `/code/bin` release directory).

  This approach is simpler. You must specify the location of the libraries during the linking phase and execution phases.

The linker needs access to the dynamic libraries. Either install the dynamic libraries into a location where the linker can include them automatically, or set the linker option to the folder that contains the dynamic libraries. Consult your linker user reference guide for link options. If you have Abaqus installed, you can use the following command to list the link flags used by Abaqus:

```
abaqus -info environment
```

The `link_exe` variable lists the common link options used by Abaqus.

Running a Coupled Simulation

Setting up the Runtime Environment

All clients and the SIMULIA Co-Simulation Services Director require the SIMULIA Co-Simulation Services libraries; the operating system needs to know where the libraries are located. There are several methods for the operating system to properly load the SIMULIA Co-Simulation Services libraries:

- If you are a member of the SIMULIA Alliance program, you can distribute the SIMULIA Co-Simulation Services libraries with your application media. In this case, you can load the libraries at run time from the same location as your application's dynamic libraries.
- The end user might extract the SIMULIA Co-Simulation Services and install them in a specific folder. On Windows, prepend this folder to the system path defined by the `PATH` environment variable. On Linux, set the `LD_LIBRARY_PATH` environment variable to include the folder containing the libraries.
- If Abaqus is installed on your system, you can point directly to the `/code/bin` directory in the Abaqus release folder to use the libraries that Abaqus is using. On Windows, prepend the `/code/bin` directory to the system path, `PATH`, environment variable. On Linux, include the `/code/bin` path in the `LD_LIBRARY_PATH` environment variable.

In addition to any license server settings, you need to define the following environment variables for the SIMULIA Co-Simulation Services Director process:

- Set the `CATDICTIONARYPATH` environment variable to point to the `/dictionary` folder. This is required for XML document parsing.
- Set `SMACSE_CFG` to point to the `/site` folder. This is required for upgrading the Co-Simulation Configuration file.

Starting the SIMULIA Co-Simulation Services Director

To start the SIMULIA Co-Simulation Services Director, provide a Co-Simulation Configuration file and a unique listener port, both of which are associated with a particular coupled simulation. The SIMULIA Co-Simulation Services Director can be started directly with `SMACseCtlMain` (which is located in the `/bin` folder of the SIMULIA Co-Simulation Services API Kit) or by using the Abaqus execution procedure.

Starting the SIMULIA Co-Simulation Services Director Directly

When starting the SIMULIA Co-Simulation Services Director directly, you must set the environment variables discussed in the previous section.

The command line summary:

`SMACseCtlMain`**job =** ***cosim-job-name*****configure =** ***configuration file-name*****listenerport =** ***listener port-number*****datacheckinteractive**

Starting the SIMULIA Co-Simulation Services Director Using the Abaqus Execution Procedure

If Abaqus is installed, you can start the SIMULIA Co-Simulation Services Director using the Abaqus execution procedure as follows:

```
abaqus -cse -job <log_file> -configure <configure_file> -listenerport <port>
```

This is the preferred approach because the Abaqus execution procedure defines the necessary environment settings. For more information, see *Director Execution*.

Technical Overview

SIMULIA Co-simulation Services consists of two parts: the SIMULIA Co-Simulation Services Director, which is in charge of controlling the coupled simulation, and services for communication, field mapping, synchronization, and coupling algorithms embedded in the client.

You can embed SIMULIA Co-simulation Services in a serial or parallel manner into your application. This section provides a technical overview of the SIMULIA Co-Simulation Services Director and the services.

In this section:

- *SIMULIA Co-Simulation Services Director*
- *Communication Services*
- *Serial vs. Parallel SIMULIA Co-Simulation Services*
- *Mapper*
- *Synchronization*
- *Coupling Algorithms*

SIMULIA Co-Simulation Services Director

The SIMULIA Co-Simulation Services Director is a separate executable that is in charge of controlling the coupled simulation.

The SIMULIA Co-Simulation Services Director is a finite-state machine that uses State Chart XML (SCXML) to define the states, events, and transitions. SCXML is able to describe complex finite-state machines based on substates, parallel states, and concurrency, which allows specialization of the advanced coupling algorithm.

Writing and interpreting SCXML markup language is complex, especially if multiple concurrent states are involved. For more details of SCXML, see *https://www.w3.org/TR/scxml/*.

To simplify the creation of the SCXML state machine for the most common coupled simulations, we use the Co-Simulation Configuration file. This file describes the configuration of the coupled simulation and is required as part of a coupled simulation. The SIMULIA Co-Simulation Services Director converts the configuration file into an SCXML state machine for the coupled simulation. For more information, about the Co-Simulation Configuration file, see *About the SIMULIA Co-Simulation Services Configuration File*.

Communication Services

SIMULIA Co-Simulation Services supports multiple communication channels.

The communication channels are characterized as control and data channels:

- The control channels exchange control messages between the clients and the SIMULIA Co-Simulation Services Director.
- The data channels exchange bulk data (for example, mesh, field, and signal data) between clients.

The communication channels can be further characterized as intercommunication and intracommunication channels:

- The intercommunication channels use TCP/IP sockets. They operate between the clients and between the clients and the SIMULIA Co-Simulation Services Director.
- The intracommunication channels use the client's Message Passing Interface, MPI. They operate within a client when you embed SIMULIA Co-Simulation Services in a parallel manner.

Serial vs. Parallel SIMULIA Co-Simulation Services

Embedding the SIMULIA Co-Simulation Services in a Serial Manner

If your client is a serial application, you must embed the SIMULIA Co-Simulation Services in a serial manner.

If your client is a distributed memory parallel application, you can embed the SIMULIA Co-Simulation Services in a serial manner. In this case, the main process (also referred to as the root process) calls the API. Mesh and field data are transferred on the root process to the remote client, and the client is responsible for fanning-in and fanning-out mesh and field data between its processes.

Embedding the SIMULIA Co-Simulation Services in a Parallel Manner

If your client is a distributed memory parallel application, you should embed the SIMULIA Co-Simulation Services in a parallel manner. In this case, each compute process registers its partition of the mesh, and the SIMULIA Co-Simulation Services fans-in and fans-out mesh and field data to the appropriate compute processes. Field mapping is performed in parallel.

When you embed the SIMULIA Co-Simulation Services in a parallel manner, you must provide an implementation of the SIMULIA Co-Simulation Services interface as discussed in *Parallel Interface Definition*.

Mapper

SIMULIA Co-Simulation Services provide mapping services to conservatively map fields between points, surfaces, and volumetric domains.

Mapping consists of two operations:

- a search where an association is made between members (nodes, elements, or integration points) between source and target meshes; and
- a calculation where field values are transferred from the source mesh to the target mesh using interpolation and/or integration algorithms.

Domain Coupling

When coupling numerical domains in multiphysics and multiscale simulations, you must consider whether the physics domains are connected through discrete points, abut one another, or overlap.

Domain coupling includes the following:

Point coupling	Point coupling occurs at discrete locations associated with a body. Coupling a physics application with a reduced-order model is an example of point coupling. In this case, the points of the reduced-order model must be co-located with the physics model nodes. Point coupling is different from logical coupling (for example, sensors and actuators), in which case there are no coordinates associated with the sensor or actuator.
Surface coupling	Surface coupling involves distinct domains that abut one another, where the common interface is a surface. Fluid-structure interaction is an example of surface coupling, where the fluid and structure domains are distinct, and field data is exchanged across the common interface surface.
Volume coupling	Volume coupling involves domains that partially or fully overlap. A sequential thermal-stress simulation, where temperatures are exchanged over the volume of a body, is an example of volume coupling.

The mapper supports mapping between similar domain types (for example, point-to-point, surface-to-surface, and volume-to-volume).

Neighborhood Search

The search algorithm establishes a one-to-one association between members of the source mesh and target mesh. Members can be nodes, elements, or integration points depending on their field position. The mapping algorithm determines how this association is formed and whether members of interest are searched on the source mesh or the target mesh.

When coupling surface domains, the surface normal is used to distinguish the bottom and top faces of membrane and shell elements. The normal direction is defined by the right-hand rule of the surface facet connectivity. The search is performed such that the surface normal of the source and target regions are facing each other.

Mapping Algorithm

A mapping category defines a set of similar mapping operations performed when mapping a field from the source mesh to the target mesh. *Table 1* shows the four possible mapping categories. The following factors determine the mapping category:

- the field's physics properties (for example, intensive quantity if the property does not depend on the amount of matter, or extensive if the property depends on the amount of matter), and
- the position where the field is evaluated.

Within a mapping category, there might be one or more algorithms.

Table 1: Mapping Categories

Category	Source Field			Target Field		
	Nodal		Integration Point	Nodal		Integration Point
	Intensive	Extensive	Intensive	Intensive	Extensive	Intensive
1						
2						
3						
4						

The four mapping categories are described below. To simplify the discussion, the term "element" is used to represent an element for a volume domain and a surface facet for a surface domain.

Mapping Category 1: Interpolation based on the element's shape function

Applicable when mapping intensive nodal fields from the source mesh to either intensive nodal or integration-point fields on the target mesh. Examples include mapping nodal displacements between meshes or mapping nodal velocity on a structural mesh to the cell-center velocity of a fluid mesh.

If the source mesh is element-based, the target node or integration point is projected onto the source mesh. The search algorithm finds the closest source element and the location within the source element where the target point is projected. The target field is then interpolated using the source element's shape function. If the source mesh is discrete, a similar algorithm is employed using a point-cloud search algorithm.

Mapping Category 2: Interpolation based on the target element's shape function

Applicable when mapping extensive nodal fields to extensive fields; for example, mapping nodal forces between source and target meshes.

If the source mesh is element based, the source node is projected onto the target mesh. The search algorithm finds the closest target element and the location within the target element where the source node is projected. The source field is then distributed using the target element's shape function.

If the target mesh is discrete, a similar algorithm is employed using a point-cloud search algorithm. The point-cloud search algorithm finds the closest target nodes and forms a virtual element to distribute the field values. In this case, the total quantity is conserved across the interface; however, the distribution might be less accurate.

Mapping Category 3: Extrapolation followed by interpolation on the source element	Applicable when mapping intensive integration-point fields to either intensive nodal or integration-point fields on the target mesh. An example is mapping stresses between source and target meshes. The source mesh must be element based. The field is first extrapolated to each element's nodes on the source mesh. The search algorithm projects the target node or integration point onto the source mesh and finds the closest source element. The target field is then interpolated using the source element's shape function.
Mapping Category 4: Integration of intensive to extensive fields	This category is applicable when mapping intensive fields to extensive fields (for example, mapping surface traction vector to nodal forces). This mapping category uses one of two available integration algorithms: the subelement or the global conservative algorithms. The subelement algorithm provides conservative mapping with a smooth distribution. It requires that the target mesh is element based. Each target element is divided into subelements. For each subelement, this algorithm locates the source field values according to the procedure described in Mapping Category 1 and sums the values over all the subelements. The number of subelements is determined automatically to provide accurate results. The global conservative algorithm provides conservative results over the entire region. It requires that the target mesh is element based. This algorithm is computationally less expensive than the subelements algorithm; however, the mapped field might show an uneven distribution in some cases. The global conservative algorithm makes the source field conserved by integrating the field over the source elements and then, for every source node, computes field values according to the procedure described in Mapping Category 2.

The mapping algorithm is determined automatically by the SIMULIA Co-Simulation Services based on the source and target fields. You can overwrite the default setting by providing a mapping specification in the Co-Simulation Configuration file (see *Mapping Specification*).

Synchronization

Synchronization ensures that a coupled simulation evolves in unison and determines the frequency at which solutions are exchanged.

The simulation is divided into a number of coupling steps. A coupling step defines the frequency of exchanges between the clients, which directly influences the stability and accuracy of the coupled solution. The coupling step size can adapt during the course of the simulation. The SIMULIA Co-Simulation Services Director establishes the coupling step size based on the clients' preferred increment sizes and a user-specified negotiation scheme.

A client can perform several increments, referred to as subcycling, within the coupling step. Subcycling allows a client to use its own increment size to reach the target time; it allows a client to cut back and reduce the time increment due to nonlinear events. However, subcycling comes at the expense of solution stability and accuracy.

Certain coupling schemes require the client's increment size to be equal to the coupling step size, referred to as lockstep. Lockstep is best from a solution stability and accuracy point of view. However, it might not be practical for solving industrial problems because one client might force an unnecessarily small increment on another client, or it might prevent a client from cutting back when deemed necessary.

The `SMACseGetTargetTime` function (see *Synchronizing and Establishing the Next Coupling Step Size*) is responsible for establishing the coupling step size and ensuring that all clients are synchronized.

Coupling Algorithms

The coupling algorithm defines the sequencing of field exchanges between the clients and any interface calculations performed.

When deciding on the coupling algorithm, you should consider solution stability and accuracy, as well as the impact on your computing resources. The SIMULIA Co-Simulation Services provides an open framework for implementing and customizing your own coupling algorithms.

The SIMULIA Co-Simulation Services coupling algorithms fall into three broad categories:

Basic coupling algorithms	The basic coupling algorithms are classical coupling techniques (for example, Gauss-Seidel and Jacobi scheme) with no interface treatment. These algorithms are suitable for physics that have weak-to-moderate coupling strength. The basic coupling algorithms serve as a starting point to customize more sophisticated algorithms, where a particular client can control and provide interface treatments. For more information, see *Explicit Coupling Schemes* and *Implicit-Iterative Coupling (Basic)*.
Advanced coupling algorithms	The advanced coupling algorithms are more sophisticated coupling algorithms and might include interface calculations performed by the SIMULIA Co-Simulation Services. These algorithms are suitable for physics that have moderate-to-strong coupling strength. For more information, see *Implicit-Iterative Coupling with Interface Treatment*.
Proprietary coupling algorithm	The proprietary coupling algorithms are highly specialized coupling algorithms and require interface computation to be performed by the clients. These algorithms are suitable for physics that have very strong coupling. Proprietary algorithms are not available to third-party applications. However, a third-party might develop their own proprietary coupling using the SIMULIA Co-Simulation Services For more information, see *Proprietary Coupling Algorithms*.

No matter which coupling algorithm you use, the SIMULIA Co-Simulation Services API is embedded into the client in a “neutral” manner; that is, the client should not have any coding logic specific to the number of clients it is coupling with, the physics being solved, the coupling scheme, and the interface treatment.

Coupling algorithms are specified in the Co-Simulation Configuration file through the model of computation element (see *Model of Computation*).

Explicit Coupling Schemes

Explicit coupling schemes define coupling algorithms where the fields are exchanged once per coupling step, and no convergence check is performed prior to advancing to the next coupling step. These coupling schemes are suited for physics with weak-to-moderate coupling strength; for example, an aeroelasticity simulation where you have air interacting with a relatively stiff structure.

The clients can be executed concurrently (commonly referred to as the Jacobi scheme) or sequentially (commonly referred to as the Gauss-Seidel scheme). The Jacobi scheme might use computing resources more effectively; however, it is less stable because both solutions lag by one coupling step. The Gauss-Seidel scheme is more stable because only one client lags by one coupling step.

Explicit coupling is numerically less costly because no coupling iterations are performed. However, it requires smaller coupling step sizes to reduce the lag between the solutions to remain stable. The Dual Mass-Spring example (see *Dual Mass-Spring Systems*) provides a study of coupling step sizes for explicit coupling.

Implicit-Iterative Coupling (Basic)

Implicit-iterative coupling is applicable if all clients use implicit time integration. The clients are executed in a staggered manner, and one analysis leads while the other analysis lags at the first exchange. At target times, multiple exchanges (referred to as coupling iterations) are performed until converge is achieved. The additional coupling iterations remove the lag between the solutions, prior to advancing to the next coupling step.

Implicit-iterative coupling requires multiple solution passes with updated interface conditions. The benefit is that a larger coupling step size can be used. Implicit-iterative coupling is recommended for physics with moderate-to-strong-coupled physics.

In the basic form, specify a maximum number of coupling exchanges prior to proceeding to the next coupling step. A client might end the coupling exchanges prior to reaching the maximum limit and implement its own interface treatment.

Implicit-Iterative Coupling with Interface Treatment

The SIMULIA Co-Simulation Services support enhanced implicit-iterative coupling algorithms with extrapolation, accelerators, and field convergence checks.

Extrapolation

Extrapolation uses the previous coupling step solution, , to provide an initial guess for the current coupling step, , where the superscript 0 refers to the initial guess and the subscript designates the coupling step. You can select between zero-order, first-order, and second-order extrapolation:

Zero-order extrapolation uses the previous coupling step solution as an initial guess:

First-order extrapolation uses a linear extrapolation based on the solution of the previous two coupling steps:

Second-order extrapolation uses a second-order extrapolation based on the solution of the previous three coupling steps:

Accelerators

An accelerator enlarges the convergence radius from a stability point of view and reduces the number of coupling iterations. The SIMULIA Co-Simulation Services support two classes of accelerators: relaxation methods and quasi-Newton methods.

Relaxation Accelerator Methods

Relaxation is a simple numerical technique employed commonly to improve the stability of a computation. Using a relaxation factor less than one improves the stability at the expense of slower convergence. Relaxation works well for weak-to-moderate-strength physics coupling.

The SIMULIA Co-Simulation Services support the constant relaxation and Aitken’s relaxation methods. Constant relaxation uses a constant user-specified relaxation factor. Constant relaxation is always used for the initial exchange with the Aitken and Quasi-Newton methods. It is available for use for all coupling exchanges (for completeness); however, you should use the Aitken's method or one of the Quasi-Newton methods instead. In the Aitken's method, the relaxation factor is adapted based on the residuals from past coupling iterations. The Aitken's relaxation method works well for moderate-strength physics coupling, and the interface computation is less expensive than the Quasi-Newton methods.

Quasi-Newton Relaxation Methods

The quasi-Newton relaxation methods approximate an inverse Jacobian at the interface. If sufficient past residual information is retained, the quasi-Newton method can deliver close to an exact inverse Jacobian. The quasi-Newton

methods incur a larger interface cost but can significantly reduce the number of coupling iterations. The methods are best suited for strongly coupled physics.

The SIMULIA Co-Simulation Services support the Anderson and the Broyden implementations. The Broyden implementation can deliver a close to exact inverse Jacobian if sufficient past residual information is provided. The Broyden implementation involves an advanced Singular Value Decomposition (SVD) factorization to limit memory requirements and avoid computations of full SVD. The Anderson method requires marginally more coupling iterations; however, it is computationally less intensive.

Convergence Checks

The advanced implicit-iterative schemes support field convergence checks performed by the SIMULIA Co-Simulation Services. The convergence criteria are specified for outgoing fields participating in iterative exchanges.

The following convergence criteria are available:

Absolute Criterion

The absolute criterion prevents the co-simulation from failing if there is no correction. This might occur at the start of a coupled simulation when there is insignificant load (for example, when a smoothed ramping curve is used):

Relative Criterion

The relative criterion applies when the coupled simulation is established.

There are two relative criteria referred to as the initial norm and step norm criteria. The initial norm criterion normalizes the correction with respect to the first coupling iteration. This ensures convergence of mathematical properties; specifically, the comparison is performed with respect to a constant value:
The step norm criterion normalizes the correction with the total solution increment, which might evolve during the iterations:

The solution is converged when either the absolute criterion or both of the relative criteria have converged:

The field criteria should not be more severe than the solver's solution criteria. You can change the convergence criteria on a field-by-field basis in the Co-Simulation Configuration file (see *Field Tolerances*).

Proprietary Coupling Algorithms

You can create specialized coupling algorithms by using SCXML state machine programming and writing solver-specific services. These algorithms are proprietary and require specialized services written for the solvers. Examples of proprietary coupling algorithms include the enhanced subcycling method used by Abaqus/Standard-to-Abaqus/Explicit and Abaqus-to-Simpack co-simulation. These algorithms represent strong physics coupling, where each client uses different time integration methods. Specialized treatment at the interface, especially if constraints exist, is implemented by the solvers.

Defining specialized coupling is beyond the scope of this guide and will require a service project.

SIMULIA Co-Simulation Services Configuration File

The Co-Simulation Configuration file defines the configuration of the coupled simulation. It is required input for the SIMULIA Co-Simulation Services Director.

In this section:

- *About the SIMULIA Co-Simulation Services Configuration File*
- *Configuration File Elements*

About the SIMULIA Co-Simulation Services Configuration File

The Co-Simulation Configuration file specifies the layout of the coupled simulation (the subsystems involved and the fields exchanged between the subsystems), the synchronization and coupling scheme, as well as mapper controls.

Developers need to author the Co-Simulation Configuration file for their coupled simulation product. Experienced end users might benefit from the discussion in case they need to modify algorithmic settings for their coupled simulation.

You should have:

- Prerequisite knowledge of XML (refer to the World Wide Web Consortium (W3C) site, for information on XML); and
- Knowledge of the capabilities and features of the SIMULIA Co-Simulation Services.

The SIMULIA Co-Simulation Services is comprised of software libraries used by the clients (referred to as subsystem simulators) and an independent directing process (referred to as the SIMULIA Co-Simulation Services Director). For example, the subsystem simulators communicate through the SIMULIA Co-Simulation Services API with a collection of runtime libraries called the Simulation Runtime Environment (SRE). The subsystem simulators are then directed in their simulation behaviors through the SRE, which operates under the direction of the SIMULIA Co-Simulation Services Director. The SIMULIA Co-Simulation Services Director is a finite state machine, as described in *SIMULIA Co-Simulation Services Director*. You define the coupled physics properties and behaviors to the SIMULIA Co-Simulation Services Director by providing the Co-Simulation Configuration file. The SIMULIA Co-Simulation Services Director then generates an SCXML state machine for the coupled simulation.

The Configuration File Content

Coupled simulation requires the interaction of two or more subsystems simulators, perhaps from different physics disciplines. The Co-Simulation Configuration file provides a general and extensible interface to define the coupled simulation, the synchronization and coupling algorithm, and mapper settings.

Any information related to the computing environment or file system resources is not defined in the Co-Simulation Configuration file. Examples of such information include the following:

- file system or database resource locators for subsystem input data (for example, Abaqus/Standard input files);
- CPU counts for execution of subsystem simulation (for example, the user's directive to run Abaqus/Explicit using 16 CPUs); and
- execution path definitions and environment setting for subsystem simulators.

Configuration File Consistency Requirements

The configuration file is created in XML. You must satisfy the following requirements for the Co-Simulation Configuration file to be accepted by the SIMULIA Co-Simulation Services Director. These requirements are verified by the SIMULIA Co-Simulation Services Director at run time. Optionally, the SIMULIA Co-Simulation Services Director can perform all but the final check using the **datacheck** execution option.

XML well-formedness	Your file must be written in proper XML format. In addition to using the SIMULIA Co-Simulation Services Director to confirm that your file is well-formed, virtually any XML authoring tool or facility provides similar services.
Conformance to the Co-Simulation Services schema	The majority of the rules and constraints defined in this section are described formally in the file `SMACseConfig.xsd`, which is included in the `/site` folder of the SIMULIA Co-Simulation Services API Kit. This file is written in the XML Schema Definition Language (XSD). In addition to use of the SIMULIA Co-Simulation Services

Director to confirm the schema compliance of your file, a significant number of XML authoring tools provide similar schema validating services.

Internal consistency of references

The Co-Simulation Configuration file defines references among items defined elsewhere in the file. For example, a “connection set” refers to a pair of “connectors” and the connector names referred to in the connection set must be defined in the file. The SIMULIA Co-Simulation Services Director enforces constraints such as these.

Topological and algorithmic consistency

The Co-Simulation Configuration file provides information that can be considered correct only in the context of the topology of the problem described (for example, are input variables paired with output variables?) and the algorithms used (for example, does a configuration description result in a deadlock situation?). The SIMULIA Co-Simulation Services Director performs these consistency checks.

External consistency of references

At run time, the subsystem simulators, through use of SIMULIA Co-Simulation Services API calls, register names with the SIMULIA Co-Simulation Services Director. These names must have accompanying entries in the Co-Simulation Configuration file. The SIMULIA Co-Simulation Services Director enforces these constraints during run time; they cannot be enforced during the data check phase.

Configuration File Elements

The configuration file defines the simulation properties of the subsystems, their interactions, and the numerical methods employed to solve the coupled simulation.

In addition to the information provided in this guide, you might find it useful to refer directly to the schema definition file, `SMACseConfig.xsd`, which is provided in the `/site` folder of the SIMULIA Co-simulation Services API Kit. The contents of the`SMACseConfig.xsd` file always supersedes the information in this guide because it contains the latest schema.

In this section:

- *The Root Element of the Co-Simulation Configuration File*
- *Components*
- *Component Instances*
- *Connectors*
- *Connections and Connection Sets*
- *Execution*
- *Mapping Specification*
- *Modulations*
- *Transformation*
- *Field Operations*

The Root Element of the Co-Simulation Configuration File

The <CoupledMultiphysicsSimulation> element is the root element of the Co-Simulation Configuration file, which uniquely identifies that the file is relevant to the SIMULIA Co-Simulation Services.

The <CoupledMultiphysicsSimulation> root element appears after the XML Prolog (that is, the first line that identifies the document as an XML compliant document). All content to define the configuration of the coupled simulation is specified within the <CoupledMultiphysicsSimulation> element.

The <header> element contains optional metadata related to the file (for example, schema version, title, description, author, creation and modification dates, etc.). The schema version is mandatory because you must specify the schema version with which this XML document complies. The SIMULIA Co-Simulation Services Director automatically upgrades the Co-Simulation Configuration file to conform with any future schema changes.

Example

The example below illustrates the XML Prolog, the <CoupledMultiphysicsSimulation> root element, and the <header> element.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<CoupledMultiphysicsSimulation noNamespaceSchemaLocation="http://www.simulia.com
 SMACseConfig.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <header>
      <!-- provide metadata -->
      <SchemaVersion>1.3</SchemaVersion>
                    :
  </header>
                  :
  <!-- any further elements to describe the configuration of the coupled
simulation -->
                  :
</CoupledMultiphysicsSimulation>
```

Components

Components are the subsystem applications in the coupled simulation.

The <components> element is a wrapper element that defines all the subsystems solvers participating in the coupled simulation. Your Co-Simulation Configuration file must contain one or more <component> elements, each defining a subsystem solver in the coupled simulation.

<component> Element

Use the <component> element to define the individual component (for example, physics solver, FMU, or Results Provider) participating in the coupled simulation. A coupled simulation must have at least one component (for example, you could perform a coupled simulation between two Abaqus/Standard analyses).

You must identify whether the component uses the top-down or bottom-up implementation:

<bottomUpImplementation>

In the bottom-up implementation, the component informs the SIMULIA Co-Simulation Services Director when it has completed a compute phase.

Use the <bottomUpImplementation> element if the component uses the SIMULIA Co-Simulation Services API.

Specify the <codeName> element to define the external reference to the application. *Table 1* defines reserved code names for Dassault Systèmes applications.

<topDownImplementation>

In the top-down implementation, the SIMULIA Co-Simulation Services Director controls one or more components through the FMU wrapper. The FMU uses the Functional Mockup Interface to communicate with one or more FMUs.

Use the <topDownImplementation> element for an FMU. The FMU must be of type co-simulation and adhere to the Functional Mockup Interface 1.0 or 2.0 standard. For more information on the FMI standard, see .

Define only one top-down component, even if multiple FMUs participate in the coupled simulation.

Table 1: Reserved code names for Dassault Systèmes apps.

Code Name	Application
Abaqus/Standard	Abaqus/Standard implicit structural mechanics solver
Abaqus/Explicit	Abaqus/Explicit explicit dynamic solver
CFD	Navier-Stokes fluid dynamic solver on **3D**EXPERIENCE
CSTStudio	CST Studio Suite electromagnetic solvers
SIMPACK	Simpack multibody dynamics solver
SMACseSimReader	SIM Results File Provider. The following can generate a SIM results file: • Abaqus/Standard • Abaqus/Explicit • Fluid Scenario Creation app (Navier-Stokes flow solver) • Plastic Injection app

Code Name	Application
CSTStudioReader	CST Results Provider for results generated by CST Studio Suite

Example

The following example illustrates how to define subsystem applications:

```
<components>
      <component name="StructureSolver">
            <bottomUpImplementation>
                <codeName>Abaqus/Standard</codeName>
            </bottomUpImplementation>
       </component>
       <component name="FluidSolve">
            <bottomUpImplementation>
                <codeName>CFD</codeName>
            </bottomUpImplementation>
       </component>
                     :
       <component name="ControlsFMU">
            <topDownImplementation>
                <!-- this will invoke the CSS FMU adapter -->
            </topDownImplementation>
       </component>
</components>
```

Component Instances

Component instances define the subsystem models participating in the coupled simulation.

The <componentInstances>> wrapper element defines the subsystem models (that is, the analysis jobs) participating in the coupled simulation. Use the <componentInstance> element to define an individual subsystem model.

<componentInstance> Element

Use the <componentInstance> element to define each subsystem model in the coupled simulation.

The name attribute is an internal reference, for cross-referencing purposes within the Co-Simulation Configuration file. The <component> element references the component instance to a component by its name.

Set the modelIdentifier attribute on each <componentInstance> element equal to the job name or the *FMU* file name (without its extension). The job name must correspond to the job name the application registers with the SMACseConnect function (see *Establishing connection to and disconnecting from the SIMULIA Co-Simulation Services Director*).

If the component instance uses the bottom-up implementation, certain basic assertions about the simulator behaviors can be modified:

- You can use the <initialConditions> element to specify how the initial conditions exchange is performed by the subsystem models.

  By default, each subsystem model sends initial fields when the SMACseInitialConditionStart function is called and receives initial conditions when the SMACseInitialConditionEnd function is called.

  In some cases, a subsystem might require the initial condition from another subsystem to compute its initial conditions. In this case, you can reverse the send and receive order; set the <sendBeforeReceive> element to false to have the client receive the initial conditions first.

  A declaration of false by both component instances would result in deadlock failure; such a declaration does not physically make sense and results in a consistency error.
- You can force a client to march in lockstep.

  Specify the <timeIncrementation> element, and set the <lockstep> element to true to force the client to march in lockstep. Otherwise, the component instance is allowed to subcycle.

Example

The following example illustrates how to define subsystem models:

```
<componentInstances>
     <componentInstance modelIdentifier="fan" name="Structure">
           <component>StructuralSolver</component>
           <timeIncrementation>
                 <lockstep>true</lockstep>
           </timeIncrementation>
           <initialConditions>
                 <sendBeforeReceive>true</sendBeforeReceive>
           </initialConditions>
     </componentInstance>
     <componentInstance modelIdentifier="air" name="Fluid">
           <component>FluidSolver</component>
           <timeIncrementation>
                 <lockstep>true</lockstep>
```

```
            </timeIncrementation>
            <initialConditions>
                  <sendBeforeReceive>false</sendBeforeReceive>
            </initialConditions>
      </componentInstance>
      <componentInstance modelIdentifier="PID_controller" name="PID">
           <component>ControlsFMU</component>
       </componentInstance>
</componentInstances>
```

Connectors

Connectors define incoming and outgoing fields/signals on a particular subsystem model.

The <connectors> wrapper element defines incoming and outgoing fields/signals on a subsystem model. A subsystem model can have one or more connectors.

Define separate connectors for each interface region if the subsystem contains multiple interface regions. You commonly have a number of fields entering or leaving a connector; in general, we recommend that you define a connector such that the exchange variables form work-conjugate pairs (for example, heat flux and temperature, force and displacement, etc.) that appear together in the same connector.

When deciding how to assign fields to a connector, consider the following:

- how the connector is paired in a connection (see *Connections and Connection Sets*);
- separate connectors must be defined for fields and signals;
- separate connectors must be defined for each region;
- use a separate connector if a component instance communicates with multiple subsystem models (for example, a connector on component instance A cannot communicate with both a connector on component instance B and a connector on component instance C; you must duplicate the connector on component instance A with another name); and
- how you might apply further configuration information (for example, mapping specification, field operations, etc.) to the connection definitions.

<connector> Element

Use the <connector> element to define a connector.

The name attribute is an internal reference name, for cross-referencing purposes within the Co-Simulation Configuration file. The <componentInstance> element refers to a component instance through its name.

Use the <input> and/or <output> elements to define incoming and outgoing fields or signals on the connector. Fields are specified by the field type name and region name, delimited with a period (for example, “velocity. Region A”) within a <variable> element. The field type must correspond to the field type registered by the subsystem solver. The field type name and region name are case sensitive. When specifying multiple fields, the order in which you list the fields is important, as discussed in *Connections*.

Example

The following example illustrates a connector definition:

```
<connectors>
  <connector name="Connector-1">
    <componentInstance>Structure</componentInstance>
    <variables>
         <input>
           <variable>temperature.SURF_A</variable>
         </input>
         <output>
           <variable>heat_flux.SURF_A</variable>
           <variable>heat_capacitance.SURF_A</variable>
          </output>
    </variables>
  </connector>

  <connector> name="Connector-2">
    <componentInstance>Fluid</componentInstance>
```

```
    <variables>
        <input>
           <variable>heat_flux.wetted.exteriorSurface</variable>
           <variable>heat_capacitance.exteriorSurface</variable>
        </input>
        <output>
           <variable>temperature.exteriorSurface</variable>
        </output>
      </variables>
  </connector>
</connectors>
```

Connections and Connection Sets

A connection associates two connectors on different subsystem models, and a connection set provides a grouping of connections.

The `<connectionSets>` wrapper element defines connection sets. A connection set provides a convenient way to group multiple connections used by the numerical methods to define the analysis phase when the fields are exchanged. A simple example of this utility is the case where you want a certain set of fields to be exchanged only during the initial conditions, while others exchange during the time integration loop.

`<ConnectionSet>` Element

Use the `<ConnectionSet>` element to define a connection set.

The `name` attribute is an internal reference, for cross-referencing purposes within the Co-Simulation Configuration file.

Set the `type` attribute to `FIELD` if fields are exchanged, and set it to `SIGNAL` if signals are exchanged. All connections within a connection set must be of the same type.

A connection set can contain only one connection.

`<connection>` Element

Use the `<connection>` element to define a connection.

If a connector has multiple variables, the connectors must be complementary; that is, one connector's incoming fields must match the other connector's outgoing fields, and vice versa, in the order in which the variables are defined. The corresponding variable names are not required to match by field type name. Therefore, field type names can be different between the subsystems; for example, a temperature field might be represented by field type “NT” in one client and “TEMP” in the other client.

You can assign a mapping specification to a connection (see *Mapping Specification*). To add a mapping specification, set the `mapSpec` attribute equal to the mapping specification reference name.

You can use the `mapOn` attribute to specify which client performs the mapping. Set `mapOn` equal to one of the following:

- `INPUT` to map incoming fields on the connection (default),
- `OUTPUT` to map outgoing fields on the connection,
- the component instance name to map both incoming and outgoing variables on a particular component instance, or
- `NOMAP` when importing a remote mesh and the component provides its own mapping services.

Example

The following examples illustrates how to define connection sets and connections:

```
<connectionSets>
   <connectionSet name="ConnectionSet-1" type="FIELD">
       <connection mapOn=”INPUT”>
            <connector>Connector-1</connector>
            <connector>Connector-2</connector>
       </connection>
   </connectionSet>
</connectionSets>
```

Execution

Execution defines all details of the synchronization and the coupling algorithm employed.

In this section:

- *Defining the coupling algorithm*
- *Model of Computation*
- *Connection Groups*
- *Scenario*

Defining the coupling algorithm

The Configuration File provides are hierchical arangement of component instances to which a numerical method can be applied.

The design of the Configuration File is generalized to enable a hierarchical arrangement of an arbitrary number of component instances, and to associate these with a particular numerical method. As a result of this generalization, we introduce an additional concept, that of an actor:

- An atomic actor refers to a single subsystem model (that is, component instance) in the coupled simulation.
- A composite actor refers to a grouping of atomic actors associated with a numerical method.

For now, it is sufficient to associate the atomic actor with a component instance, and use a single composite actor definition.

The `<execution>` element defines the composite actors, the connection groups (*Connection Groups*, and the duration of the coupled simulation (*Scenario*). Use `<compositeActors>` element within the `<execution>` element to define a composite actor. Currently, only a single composite actor definition is supported.

Use the `<compositeActor>` element within the `<compositeActors>` element to define the atomic actors and an associated computational model. The `name` attribute is an internal reference name. Use the `<actors>` and `<atomicActor>` elements to specify the component instances belonging to the composite actor definition. The order the atomic actors are listed defines the staggering sequence; the first entry is the lead actor, and subsequent entries are the lagging actors, which are sequenced in the order specified.

Use the `<modelOfComputation>` element within the `<compositeActor>` element to define the computational model, discussed in *Model of Computation*.

Example

The following example illustrates the `<compositeActors>` element:

```
<execution>
    <compositeActors>
        <compositeActor name="twoCodeContinuousTime">
            <actors>
                <atomicActor>cht_circuitboard_std</atomicActor>
                <atomicActor>cht_circuitboard_cfd</atomicActor>
            </actors>

            <modelOfComputation>
               <!-- specify model of computation -->
            </modelOfComputation>
        </compositeActor>
    </compositeActors>

    <connectionGroups>
       <!-- specify connection groups -->
    </connectionGroups>

    <scenario>
          <!-- scenario definition -->>
    </scenario>
</execution>
```

Model of Computation

The model of computation defines the synchronization method and coupling algorithm associated with a composite actor.

To define a computational model for a composite actor, use the `<modelOfComputation>` element within `<compositeActor>` element (see *Defining the coupling algorithm*). All computational models are time-based; therefore, specify the `<continuesTime>` element within the `<modelOfComputation>` element. Within the `<continuesTime>` element, define the negotiation method and coupling algorithm as discussed below.

Negotiation Method

The negotiation method defines how the next coupling step size is established. For more information, see*Synchronization* . Set the `<negotiationMethod>` element to one of the methods defined in *Table 1*. For some negotiation methods you may have to define a negotiation actor using the `<negotiationActor>` element or provide a constant coupling step size using the `<constantDt>` element.

Table 1: Negotiation methods.

Method	Description	Requirement
MIN	Computes the minimum of all actors preferred time increments.	none
MAX	Computes the maximum of all actors preferred time increments.	none
ACTOR-DRIVEN	One of the actors determines the coupling step size.	Use the `<negotiationActor>` element to define the negotiation actor.
CONSTANTDT	Sets the coupling step size to a user-specified constant value.	Use the `<constantDt>` element to define the constant coupling step size.
CONSTANT-MULTIPLE	An actor's coupling step size is an integral multiple of the constant coupling step size defined by the negotiation actor.	Use the `<negotiationActor>` element to define the negotiation actor.

The following example illustrates how to specify the negotiation method:

```
<execution>

   <compositeActors>

      <compositeActor, name="twoCodeContinuousTime">
         <actors>
                <!-- list the subsystem models associated with the model of
computation -->
         </actors>

         <modelOfComputation>
            <continuousTime>

               <negotiationMethod>ACTOR-DRIVEN</negotiationMethod>
               <negotiationActor>cht_circuitboard_std</negotiationActor>

            </continuousTime>
         </modelOfComputation>

                        :

       </compositeActor>
```

```
    </compositeActors>
                  :
</execution>
```

Coupling Algorithm

The coupling algorithm defines the sequencing of the atomic agents, and any interface for implicit-iterative coupling scheme. For clarity, we divide this discussion into two parts:

- coupling scheme, and
- interface treatment for implicit iterative coupling scheme.

Coupling Scheme

The coupling scheme defines the sequencing of the atomic agents. Use the `<algorithm>` element to select one of the coupling algorithms defined in *Table 2*. For more information about coupling algorithms, see *Coupling Algorithms*.

Table 2: Coupling algorithms.

Algorithm	Description	Comments
JACOBI	Executes the agents concurrently.	Not applicable for implicit iterative coupling.
GAUSS-SEIDEL	Executes the agents in a sequential manner defined by the staggering sequence, and exports fields at `NotifyEnd`.	
GAUSS-SEIDEL-PREDICTOR	Executes the agents in a sequential manner defined by the staggering sequence, and exports predictor fields at `NotifyStart`.	Transfers predictor fields taking account the next coupling step size.
GANDCENHANCED	Uses the enhanced stabilized subcycling method.	Proprietary coupling algorithm for structure-to-structure and structure-to-mbd co-simulation.
SEQUENTIAL	Uses sequential analysis method when coupling with a Results Provider.	

By default, an explicit coupling sequencing is assumed. Each atomic actor exchanges solution fields when it reaches the coupling step target time, and then the simulation proceeds to the next coupling step.

When coupling implicit solvers, you can select an implicit-iterative coupling scheme by specifying the `<iterativeCoupling>` element. In this case, the atomic actors exchange and correct the solutions at the coupling step target time before proceeding to the next coupling step. Use the `<exchangeLimit>` element to provide an upper limit of additional exchanges. The upper bound ends the iterative coupling loop, whether or not the interface solution converges. Either client may end the coupling iteration prior based on its computations (see *Notifying that the Client Completed an Implicit Coupling Iteration*).

Example

The following example illustrates how to specify an implicit-iterative coupling algorithm:

```
<execution>

   <compositeActors>

      <compositeActor, name="twoCodeContinousTime">
         <actors>
```

```
                <!-- list the subsystem models associated with the model of
computation -->
         </actors>

         <modelOfComputation>
            <continuousTime>

               <negotiationMethod>MAX</negotiationMethod>

               <algorithm>GAUSS-SEIDEL</algorithm>
               <iterativeCoupling>
                  <exchangeLimit>30</exchangeLimit>
               </iterativeCoupling>

            </continuousTime>
         </modelOfComputation>

                        :

       </compositeActor>
    </compositeActors>
                  :
</execution>
```

Interface Treatment for Implicit-Iterative Coupling

The SIMULIA Co-Simulation Services provides advanced numerical methods for implicit-iterative coupling scheme, referred to as accelerators, as well as extrapolation and convergence checks. Accelerators can improve the convergence of a coupled solution and reduce the computational cost. Accelerators fall into two categories: relaxation methods and quasi-Newton methods (see *Implicit-Iterative Coupling with Interface Treatment*).

Use the `<accelerator>` element within the `<iterativeCoupling>` element to specify the advanced numerical methods.

With all accelerator methods you can specify an extrapolation method (see *Extrapolation*). Use the `<extrapolationOrder>`element to specify the extrapolation defined in *Table 3*.

Table 3: Extrapolation order.

Element	Description
`<extrapolationOrder>`	Extrapolation order. Set to 0 to use the previous coupling step, set to 1 for linear extrapolation, and set to 2 for second-order extrapolation.

Relaxation Methods

Two relaxation methods are available, constant relaxation and Aitken relaxation.

To specify the constant relaxation method, use the `<relaxation>` element within the `<accelerator>`element, and define the parameters listed in *Table 4*.

Table 4: Settings for constant relaxation method.

Element	Description	Default
`<extrapolationOrder>`	Extrapolation order. Set to 0 to use the previous coupling step, set to 1 for linear extrapolation, and set to 2 for second-order extrapolation.	2
`<relaxationFactor>`	Constant relaxation factor. The value should be between 0.0 and 1.0.	0.5

The following example illustrates how to define the constant relaxation method:

```
<iterativeCoupling>
    <accelerator>
        <relaxation>
              <extrapolationOrder>2</extrapolationOrder>
              <relaxationFactor>0.5</relaxationFactor>
         </relaxation>
         <!-- specify tolerances -->
     </accelerator>
     <exchangeLimit>30</exchangeLimit>
</iterativeCoupling>
```

Use the `<aitken>` element to specify the Aitken relaxation method, and define the parameters listed in *Table 5*.

Table 5: Settings for Aitken relaxation method.

Element	Description	Default
`<relaxationFactor>`	Initial relaxation factor. The value should be between 0.0 and 1.0.	0.5
`<relaxLowerLimit>`	Lower limit on computed relaxation factor.	0.0
`<relaxUpperLimit>`	Upper limit on the computed relaxation factor.	1.0

The following example illustrates how to define the Aitken relaxation method:

```
<iterativeCoupling>
     <accelerator>
           <extrapolationOrder>2</extrapolationOrder>
           <aitkens>
              <extrapolationOrder>2</extrapolationOrder>
              <relaxationFactor>0.5</relaxationFactor>
              <relaxLowerLimit>0.1</relaxLowerLimit>
              <relaxUpperLimit>0.9</relaxUpperLimit>
           </aitkens>
           <!-- specify tolerances -->
     </accelerator>
</iterativeCoupling>
```

Quasi-Newton Methods

Two quasi-Newton methods are available: the Anderson and the Broyden methods.

To specify either quasi-Newton method, use the <quasiNewton> element. You can also modify the default extrapolation order and the initial relaxation factor.

To select the Anderson method, use the <method> and <anderson> elements within the <quasiNewton> element. Set the type attribute on the <anderson> element to define how the accelerator handles the data set:

- Set type to all to retain all previous and current solutions.
- Set type to reuse (default) to specify the number of previous solutions to retain in the data set.
- Set type to filtered to use a QR filter.

The Anderson method parameters are listed in *Table 6*

Table 6: Settings for Anderson quasi-Newton method.

Element	Description	Default
<nbOfReusedTimeSteps>	Number of reused time steps. The value should be zero or any positive integer.	10
<nbOfReusedColumns>	Number of reused solutions per time step.	100

The following example illustrates how to define the Anderson quasi-Newton method:

```
<iterativeCoupling>
     <accelerator>
         <quasiNewton>
             <extrapolationOrder>2</extrapolationOrder>
             <relaxationFactor>0.5</relaxationFactor>
             <method>
              <anderson type="REUSE">
                   <nbOfReusedTimeSteps>1</nbOfReusedTimeSteps>
                   <nbOfReusedColumns>100</nbOfReusedColumns>
               </anderson>
             </method>
             <!-- filtering method -->
         </quasiNewton>
     </accelerator>
</iterativeCoupling>
```

To select the Broyden method, specify the <method> and <broyden> elements within the <quasiNewton> element. The Broyden method parameters are listed in *Table 7*.

Table 7: Settings for Broyden quasi-Newton method.

Element	Description	Default
<sizeChunk>	Sets the threshold number before updating the SVD.	10
<svdMaxRank>	Sets the maximum SVD rank. Set to 0 to keep all (full rank).	0

With either quasi-Newton method, you can apply a filter to orthogonalize the data set to avoid a poorly conditioned matrix. Use the <filtering> element to define the filter type and parameters listed in *Table 8*.

Table 8: Filter parameters.

Element	Description	Default
`<epsFiltering>`	Filtering threshold for QR1 and QR2, and the maximum threshold for QRA. The values should be between 0.0 and 1.0.	0.01
`<epsInit>`	Initial filtering threshold for the QRA method. The values should be between 0 and 1.0.	1.0×10^{-6}

Use the `type` attribute on the `<filtering>` element to define the filtering method:

- Set `type` to `QR1` to perform a QU factorization when a vector is deleted.
- Set `type` to `QR2` to perform vector deletion during QU factorization (default).
- Set `type` to `QRA` to perform vector deletion during QU where the criterion is adapted in order to loosen the filtering as much as possible.

The following example illustrates how to add filtering to the data set:

```
<iterativeCoupling>
     <accelerator>
         <quasiNewton>
               <extrapolationOrder>2</extrapolationOrder>
               <relaxationFactor>0.5</relaxationFactor>
               <method>
                   <!-- specify Broyden or Anderson method -->
               </method>
               <filtering type="QR2">
                   <epsFiltering>1.0e-2</epsFiltering>
               </filtering>
           </quasiNewton>
     </accelerator>
</iterativeCoupling>
```

Field Tolerances

You can use field tolerance to override the default field criteria (see *Coupling Algorithms*) . Use the `<convergence>` element within the `<iterativeCoupling>` element to specify field tolerances. The `startTime` attribute specifies the time when the tolerances are enforced. The default value is zero, signifying the start of the co-simulation event. You might consider delaying the enforcement if you are ramping up the fields at the start of the simulation. The `minNumberOfIterations` attribute imposes a minimum number of coupling iterations to perform, irrespective of field tolerance criteria.

The convergence criteria are defined for outgoing fields. Use the `<convergenceParameters>` element to identify the field and define the convergence criteria. For each field, use the `<field>` element to define the criteria, and define the attributes in *Table 9*.

Table 9: Attributes for the `<field>` element.

Attribute	Description	Default
`componentInstance`	Component instance for the outgoing field.	
`name`	Outgoing field type name.	

Attribute	Description	Default
absConvEps	Absolute criterion.	1.0×10^{-9}
relConvEps	Relative initial norm.	5.0×10^{-4}
relStepConvEps	Relative step norm.	5.0×10^{-3}

The following example illustrates how to define convergence criteria:

```
<iterativeCoupling>
         <accelerator>
                :
         </accelerator>
         <convergence startTime="0.0" minNumberOfIterations="2">
             <convergenceParameters>
                 <field componentInstance="SimulatorA" name="force"
                                absConvEps="1.0e-08"
                                relConvEps="5.0e-03"
                                relStepConvEps="5.0e-03">
                  </field>
                  <field componentInstance="SimulatorB" name="displacement"
                                absConvEps="1.0e-08"
                                relConvEps="5.0e-03"
                                relStepConvEps="5.0e-03">
                   </field>
              </convergenceParameters>
         </convergence>
         <exchangeLimit>30</exchangeLimit>
     </iterativeCoupling>
```

Connection Groups

A connection group defines at what compute phase fields are exchanged.

Use the `<connectionGroups>` element within the `<execution>` element to define the active connection categories (for example, computed phases such as initial conditions, coupling iterations, and coupling step) when fields are exchanged.

Use the `<connectionCategory>` within the `<connectionGroups>` element for each active compute phase when fields are exchanged. The `name` attribute specifies the compute phase, as listed in *Table 1*

The following example illustrates how you can define connection groups.

```
<execution>
    <!-- define composite actors and associate to computational model >

    <connectionGroups>
       <connectionCategory name="InitialConditions">
              <connectionSet>ConnectionSet-1</connectionSet>
              <connectionSet>ConnectionSet-2</connectionSet>
         </connectionCategory>
         <connectionCategory name="CouplingStep">
              <connectionSet>ConnectionSet-2</connectionSet>
         </connectionCategory>
         <connectionCategory name="CouplingIteration">
              <connectionSet>ConnectionSet-2</connectionSet>
         </connectionCategory>
    </connectionGroups>

  </execution>
```

Table 1: Connection categories.

Computation Phase	Description
`InitialConditions`	Connection sets are exchanged during the initial condition exchange. For more information, see *Initial Conditions*.
`CouplingStep`	Connection sets are exchanged after a coupling step is completed. For more information, see *Notifying that the Client is Ready to Start Its Increment* and *Notifying that the Client Completed Its Increment*.
`CouplingIteration`	Connection sets are exchanged during coupling iterations. For more information, see *Notifying that the Client Completed an Implicit Coupling Iteration*.

Within the `<connectionGroups>` element, use the `<connectionSet>` element to identify one or more connection sets (see *<ConnectionSet> Element*) active.

Scenario

The scenario defines the duration of the coupled event.

Scenario Elements

Use the `<scenario>` and `<duration>` elements to define the duration when exchanges are performed. If the duration is shorter than the analysis procedure time, the imported fields remain constant for the remaining period.

Example

The following example illustrates scenario definition:

```
<execution>
    <scenario>
          <duration>20.0</duration>
    </scenario>
</execution>
```

Mapping Specification

A mapping specification allows you to change the mapper search criteria, the integration algorithm, and how missing members are treated.

The field mapping is discussed in *Mapper*. By default, the most appropriate mapping algorithm is selected based on the source and target fields. In certain cases, you can change the neighborhood search criteria and the transfer operation (for example, the mapping integration method and how missing members are handled).

Mapping Specification

Use the `<mappingSpecifications>` wrapper element to specify one or more mapping specifications.

Use the `<mapSpec>` element to provide a mapping specification. The `name` attribute is an internal reference name, used to assign a mapping specification to a connector definition.

Search Criteria

Use the `<search>` element to control the neighborhood search.

By default, search tolerances are relative to the characteristic element size of the source and target meshes. In some cases, it might be more appropriate to specify absolute distances.

Set the `isAbsolute` attribute to true if search tolerances are provided in units of length. If the source and target meshes have different unit systems, use meters (SI units) for the absolute tolerances in the search criteria.

Use the `<normal>` and `<tangential>` elements to define search tolerances in the normal and tangential directions, respectively. Only members that fall within this search distance are considered. If any target member could not find a source member within the given search tolerance, it is marked as a missing member.

For surface facets, the search is performed in the opposing normal direction to distinguish between the top and bottom faces of membrane and shell elements. You can apply search tolerances in the outward direction, the inward direction, or both directions as defined in *Table 1*.

Table 1: Normal search parameters.

Element	Description
`<outward>`	Search tolerance on a surface facet in the outward normal direction.
`<inward>`	Search tolerance on a surface facet in the inward normal direction.
`<bidirectional>`	Search tolerances for volumetric elements.

Example

The following example illustrates how to specify search tolerances:

```
<mappingSpecifications>
      <mapSpec name="MapSpec-1"”>
         <search isAbsolute=”false”>
            <normal>
               <bidirectional>0.05</bidirectional>
               <outward>0.05</outward>
               <inward>0.05</inward>
            </normal>
            <tangential>0.05</tangential>
         </search>
      </mapSpec>
```

```
          :
</mappingSpecifications>
```

Transfer

The transfer specification allows you to change the mapping algorithm's conservation principle and to specify how missing members are treated.

Use the `<transfer>` element to modify the default settings.

Use the `<flux>` element to specify the conservation algorithm. By default, a subelement algorithm is employed. The subelement algorithm provides conservative mapping with a smooth distribution. Alternatively, you can select a global conservation algorithm where global conservation of the quantity is ensured, potentially at the expense of the smoothness of the distribution. To select the global conservation method, set the `<globalConservation>` element to true.

Missing members are nodes or elements where the search algorithm failed to find a corresponding member. Use this capability with caution; it is intended when there are a few failures due to complicated geometry. Use the `<orphanTreatment>` element, and specify a treatment listed in *Table 2*.

Table 2: Transfer settings.

Entity	Description
`<nearestNeighbor>`	Computes the missing member value based on the nearest neighbor value.
`<constantValue>`	Lets you specify a constant value to be used for missing members (for example, a zero flux or ambient temperature value).

Example

The following example illustrates how to select the nearest neighbor treatment.

```
<mappingSpecifications>
      <mapSpec name="defaultMapSpec">
               :
         <transfer>
            <orphanTreatment>
               <nearestNeighbor></nearestNeighbor>
            </orphanTreatment>
         </transfer>
      </mapSpec>
 </mappingSpecifications>
```

Modulations

A modulation modifies the magnitude of an incoming field and is commonly employed at the start of a coupled simulation to avoid introducing shocks when the subsystrems are not in dynamic equilibrium.

Use the `<modulationSpecifications>` wrapper element to define modulations.

Modulation Definition

Use the `<modSpec>` element to specify the modulation.

The `name` attribute is an internal reference name.

Use the `type` attribute to specify the modulation type. You can apply only a smooth modulation; therefore, set the `type` attribute to `SMOOTH`.

Use the `<parameter>` element to specify the start time, t1, and the end time, t2, for the modulation.

Use the `<connector>` element to specify the connector, and use the `<variables>` element to specify the incoming field variables to be modified.

Example

The following example illustrates how to add a modulation:

```
<modulationSpecifications>
      <modSpec name="Modulation-1" type="SMOOTH">
         <parameters>
            <parameter name="t1" value="0.1"></parameter>
            <parameter name="t2" value="1"></parameter>
         </parameters>
         <connector>Connector-4</connector>
         <variables>
            <variable>traction_vector</variable>
         </variables>
      </modSpec>
   </modulationSpecifications>
```

Transformation

You can reposition the source mesh to the target mesh if they are not co-located. Transformation also rotates the vector and tensor fields accordingly.

Use the `<transformSpecifications>` wrapper element to specify transformations to move the source mesh to the target mesh.

Use the `<transformSpec>` element to specify the translation and/or rotation to move the source to a target mesh. The `name` attribute is an internal reference name. Specify the component instance using the `<componentInstance>` element, and use the `<region>` element to identify the interface region to be repositioned.

Specify a translation vector using the `<translation>` element and/or a rotation matrix using the `<rotationMatrix>` element.

Example

The following example illustrates how to reposition a mesh:

```
<transformSpecifications>
      <transformSpec name = "MeshTransformation-1">
            <componentInstance>Actor-2</componentInstance>
            <region>WettedSurface</region>
            <translation>0, 12, 0</translation>
            <rotationMatrix>0.819152, -0.573576, 0, 0.573576, 0.819152, 0, 0,
0, 1</rotationMatrix>
      </transformSpec>
</transformSpecifications>
```

Field Operations

Field Operations Elements

Use the `<fieldOperationSpecifications>` wrapper element to apply field operations to scale incoming fields.

Use the `<fieldOpSpec>` element to apply a field operation.

The `name` attribute on the `<fieldOpSpec>` element is an internal reference name.

You can define field operations on incoming fields, which are identified by using the `<connector>` and `<variables>` elements. The `type` attribute specifies the operation. Set the `type` attribute to `SCALE` to scale the incoming field.

Use the `parameters` element to specify the scaling factors.

Example

The following example illustrates how to scale an incoming field:

```
<fieldOperationSpecifications>
       <fieldOpSpec name = "FieldOp-1" type = "SCALE">
           <parameters>
               <parameter name = "scaleFactor" value = "2.5"/>
           </parameters>
           <connector>Connector-1</connector>
           <variables>
               <variable>field_variable_02.EALL</variable>
           </variables>
       </fieldOpSpec>
   </fieldOperationSpecifications>
```

SIMULIA Co-Simulation Services API

The SIMULIA Co-Simulation Services API provides a C language interface to the Co-Simulation Services.

The C language is used because it is portable to different computing systems, and interfaces to other programming languages can be developed easily.

The API can be broken down into two types of services: core services and add-on services. Core services provide the essential functionality. Add-on services provide extensions that might be required for certain applications.

In this section:

- *Core Services*
- *Add-On Services*

Core Services

This section describes the essential functionality of the SIMULIA Co-Simulation Services API.

In this section:

- *Initialization and Finalization*
- *Establishing connection to and disconnecting from the SIMULIA Co-Simulation Services Director*
- *Defining the Interface Model*
- *Control API*
- *Data API*
- *Message Handler*

Initialization and Finalization

Initialization

The SIMULIA Co-Simulation Services can operate in serial or parallel mode.

In serial mode, all functions are called from the main process even if the client uses a parallel distributed memory implementation. In this case, the client is responsible for fan-in/out mesh and field data between its compute ranks.

In parallel mode, the functions are called collectively on every compute rank, and the SIMULIA Co-Simulation Services is responsible for distributing the mesh and field data to the appropriate compute ranks. You must provide an implementation for the SIMULIA Co-Simulation ServicesDMP interface (see *Parallel Interface Definition*) and register it.

Use the `SMACseInitialize` function to initialize the SRE. This function takes the following argument:

`iDMPDLLPath` This argument is required when embedding the SIMULIA Co-Simulation Services in parallel mode. Set it to the dynamic link library containing your implementation of the distributed memory interface when using parallel mode; otherwise, set it to `NULL`.

When you embed the SIMULIA Co-Simulation Services in parallel mode, you must call `SMACseInitialize` on all compute ranks.

Finalization

Finalizing ensures that any memory allocated by theSRE is deallocated. This is especially important for an interactive application that might run multiple simulations.

Use the `SMACseFinalize` function to finalize the co-simulation services session. The`SMACseFinalize` function has no arguments.

When you embed the SIMULIA Co-Simulation Services in parallel mode, you must call `SMACseFinalize` on all compute ranks.

Example

The following example illustrates how to initialize and finalize the co-simulation services session:

```
// CSS Header files
#include <SMACseApi.h>
#include <SMACseApiEnums.h>

int main(int argc, char *argv[])
{
                :
// Initialize the SIMULIA CO-SIMULATION SERVICES
#if defined (_WINDOWS_SOURCE)
    SMACseInitialize("myDmpModule.dll");
#elif defined (_LINUX_SOURCE)
    SMACseInitialize("myDmpModule.so");
#endif
                :

// Finalize the SIMULIA CO-SIMULATION SERVICES
    SMACseFinalize( );
             :
```

```
    exit(0);
}
```

Establishing connection to and disconnecting from the SIMULIA Co-Simulation Services Director

Establishing Connections

When establishing connections, each client connects to the SIMULIA Co-Simulation Services Director process in charge of the coupled simulation by specifying the host the SIMULIA Co-Simulation Services Director is running on and the port number used for the particular coupled simulation.

Use the `SMACseConnect` function to establish connections. The `SMACseConnect` function has the following arguments:

`iCodeName`	Name of your application. It must correspond to one of the components in the Co-Simulation Configuration file.
`iJobName`	Analysis job name. It must correspond to one of the component instances in the Co-Simulation Configuration file.
`iWorkDir`	Folder where the SIMULIA Co-Simulation Services writes the diagnostics.
`iDirectorConnection`	SIMULIA Co-Simulation Services Director process in charge of the coupled simulation. The connection is specified as a host:port pair, where the host specifies the host name or IPV4/IPV6TCP/IP address the SIMULIA Co-Simulation Services Director is running on, and the port is the listener port of the SIMULIA Co-Simulation Services Director.
`iTimeout`	Timeout period in seconds. `SMACseConnect` returns a failure if the clients do not connect within the specified timeout period.
`iStartTime`	Set this argument to zero.
`iUserPutFieldFunction`	`putField` call-back function (see *Transferring Field Data from the Client to the Co-Simulation Services*).
`iUserData`	Your `userData` object. This is an opaque pointer that is passed to the `putField` callback function to allow access to your client's data.
`iUserMessageHandlerFunction`	Message handler callback function (see *Message Handler*). The SIMULIA Co-Simulation Services uses this function to raise any informational, warning, and error messages.

`SMACseConnect` returns with success if all participants in a coupled simulation have established the required communication channels within a specified timeout period.

In serial mode, call `SMACseConnect` on the root process; in parallel mode, call `SMACseConnect` on all compute ranks.

Disconnecting

When the end of the simulation is reached, or an error occurs, you should disconnect the client to inform the other clients that your application ended the coupled simulation.

You can provide a message to be sent to the other clients. The exception message can be communicated to the remote clients only if all connections are established. If a client should fail prior to establishing connections with the remote clients (for example, due to input file errors, client error, core dump, etc.), the message cannot be communicated.

Use the `SMACseDisconnectExt` function to end the coupled simulation. The `SMACseDisconnectExt` function takes the following argument:

`iExceptionMessage`	Constant character string containing an exception message, or NULL if no message should be sent.

In serial mode, call `SMACseDisconnectExt` on the root process; in parallel mode, call `SMACseDisconnectExt` collectively on all compute ranks. Only the message on the root process is conveyed by the SIMULIA Co-Simulation Services Director.

Example

The following example illustrates how to connect and disconnect the client:

```
// Co-Simulation Services header files
#include <SMACseApi.h>
#include <SMACseApiEnums.h>

int main(int argc, char** argv)
{
            :
   // Initialize the CSE API
   SMACseInitialize( NULL );

   // Connect to Co-Simulation Services
   const char* codeName = "myApplication";
   const char* jobName = "myJob";
   const char* workDir = ".//";
   const char* CSSDirectorConn = "localhost:21000";
   const int timeout = 300;
   double startTime = 0.0;
   void* userData = NULL;
             :
   SMACseConnectionStatus cStatus = SMACseConnect(codeName, jobName, workDir,
           directorConnection, timeout, startTime,
           myPutField, userData, myMessageHandler);
   if (cStatus == SMACseConnectStatus_failure) {
      // You do not need to disconnect, since connection were never established.

       SMACseFinalize( );
       exit(1);
   }
            :

   // End of analysis; disconnect and finalize
   SMACseDisconnectExt("The coupled simulation completed successfully.");
   SMACseFinalize( );
             :
}
```

Defining the Interface Model

The interface model describes the interface regions and fields that interact in a coupled simulation.

The SIMULIA Co-Simulation Services uses the concept of mesh and fields to define the interface model. The interface model consists of one or more interface regions on which one or more fields are exchanged.

Creating and Registering Interface Regions

An interface region defines a physics domain being coupled. The domains are required to be co-located. For a two-way co-simulation, the source and target regions must have the same extent for proper spatial mapping. For one-way co-simulation or a sequential simulation, the source region can be larger than the target region.

You can couple domains as follows:

- through discrete locations,
- through a common interface surface adjoining two non-overlapping domains, and
- over the entire domain if the physics domain overlap.

The interface regions can be of different region types; however, you can couple common region types only (for example, point-to-point, surface-to-surface, and volume-to-volume regions).

At the API level, a mesh defines an interface region. A mesh has a reference name; it contains one or more node collections and zero or more element collections. Here, the term “element” refers to either a surface facet for a surface region or an element for a volume region. Element collections are used to group different element classes. Generally, a mesh has a single node collection and one or more element collections. It can have zero element collections when the interface region consists of discrete points only.

To define an interface region, you must do the following:

- create a mesh object;
- add one or more node collections;
- add element collections, if required; and
- register the mesh.

Creating a Mesh Object

Use the `SMACseCreateMesh` function to create a mesh object. The `SMACseCreateMesh` function takes the following argument:

`iReferenceName`	Interface mesh reference name. The name is case sensitive and must correspond to the region name specified in the Co-Simulation Configuration file.

`SMACseCreateMesh` returns an opaque handle to the mesh object. The mesh object handle is used to add node and element collections and to register the mesh object.

When you embed the SIMULIA Co-Simulation Services in parallel mode, call `SMACseCreateMesh` on any compute rank that contains part of the interface region.

Adding Node Collections

A mesh typically has at least one node collection. However, you can define multiple node collections to provide further grouping (for example, grouping nodes per numerical domains present on a compute rank). Do not register any node collection when importing a remote mesh (see *Importing a Remote Client's Mesh*).

A node collection has a reference name and is associated with a spatial dimensionality. It also contains a list of node labels and coordinates. The node labels must be unique across all compute ranks.

Use the `SMACseAddNodeCollection` function to add a node collection. The `SMACseAddNodeCollection` function takes the following arguments:

`iMesh` Mesh handle.

`iCollectionRefName` Node collection reference name.

The reference name must be unique name within the region.

`iRegionDim` `SMACseDim_2d` for a two-dimensional mesh, `SMACseDim_3d` for a three-dimensional mesh, or `SMACseDim_Axi` for an axisymmetric mesh.

`iNNodes` Number of nodes in the collection.

`iLabels` Vector containing the node labels.

`iCoordinates` Vector containing the nodal coordinates. The coordinates are listed consecutively for every node in the collection given the spatial dimensionality.

Node labels can start with zero and can be any unsigned integer value. You can provide the node labels in any order; they do not need to be defined consecutively. However, the node labels must be unique for the entire model. Nodal fields are exchanged in the order in which the nodes are defined in the collection.

When you embed the SIMULIA Co-Simulation Services in parallel mode, call `SMACseAddNodeCollection` if the compute rank contains nodes belonging to the mesh. You should register only the nodes belonging to the compute rank.

Adding Element Collections

A mesh consists of zero or more element collections. If no element collections are specified, the mesh definition consists purely of nodes without element topology information; this represents discrete points. For surface coupling, each surface facet is represented as an element. Do not register any element collections when importing a remote mesh (see *Importing a Remote Client's Mesh*).

An element collection has a reference name; it is associated with an element class and refers to a node collection. It also contains a list of element labels and connectivity data. The connectivity data is in terms of the node labels provided in the node collection. In most cases, multiple element collections might refer to a single node collection. The element labels must be unique across all compute ranks.

Use the `SMACseAddElemCollection` function to add an element collection. The `SMACseAddElemCollection` function takes the following arguments:

`iMesh` Mesh handle.

`iElemCollectionRefName` Element collection reference name.

`iNodeCollectionRefName` Node collection reference name associated with the element collection.

`iElemClass` String defining the element class/type defined in *Table 1*.

Table 1: Element class.

String	Description
`SMACseSurfTopo_Line2`	2D Surface facet, 2-node line

String	Description
SMACseSurfTopo_Line3	2D Surface facet, 3-node quadratic line
SMACseSurfTopo_Tri3	3D Surface facet, 3-node linear triangle
SMACseSurfTopo_Tri6	3D Surface facet, 6-node quadratic triangle
SMACseSurfTopo_Quad4	3D Surface facet, 4-node linear quadrilateral
SMACseSurfTopo_Quad8	3D Surface facet, 8-node quadratic quadrilateral
SMACseVolTopo_Tri3	2D Volume element, 3-node linear triangle
SMACseVolTopo_Tri6	2D Volume element, 6-node quadratic triangle
SMACseVolTopo_Quad4	2D Volume element, 4-node linear quadrilateral
SMACseVolTopo_Quad8	2D Volume element, 8-node quadratic quadrilateral
SMACseVolTopo_Tet4	3D Volume element, 4-node linear tetrahedron
SMACseVolTopo_Tet10	3D Volume element, 10-node quadratic tetrahedron
SMACseVolTopo_Pyr5	3D Volume element, 5-node pyramid
SMACseVolTopo_Wedge6	3D Volume element, 6-node linear wedge
SMACseVolTopo_Wedge15	3D Volume element, 15-node quadratic wedge
SMACseVolTopo_Hex8	3D Volume element, 8-node linear hexahedron
SMACseVolTopo_Hex20	3D Volume element, 20-node quadratic hexahedron

`iNElems` Number of elements in the collection.

`iLabels` Vector containing the element labels.

The element labels must be unique across a compute domain.

`iConnectivity` Vector containing the nodal connectivity data for each element in the collection.

Specify one element collection for each element class.

To provide further grouping to access field data, you can define multiple element collections for the same element class. The element class name identifies the number of nodes per element. Element connectivity data is specified by the node labels; it is provided as the connectivity data for the first element, followed by the second element, etc. For each element, the node labels are provided in the order illustrated in *Figure 1*, *Figure 2*, and *Figure 3*. For surface elements (facets) the connectivity defines the outward normal according to the right-hand rule convention. The outward normals of each subsystem model must point toward each other for the mapper to distinguish between the top and bottom faces of membrane and shell elements.

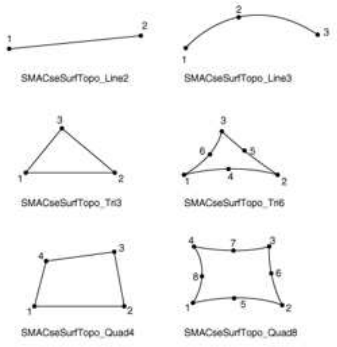


Figure 1: 2D and 3D surface element topology.

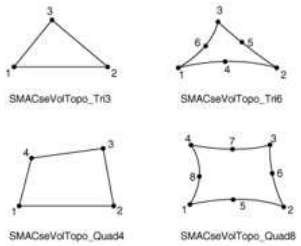


Figure 2: 2D volume element topology.

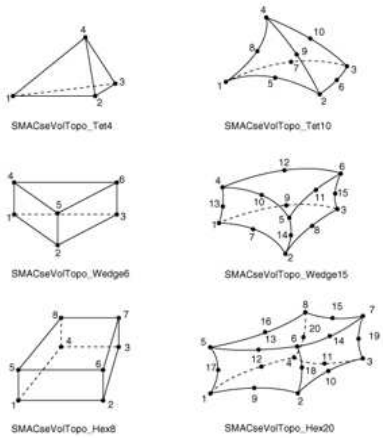


Figure 3: 3D volume element topology.

When you embed the SIMULIA Co-Simulation Services in parallel mode, each element is unique to a compute domain; therefore, each element can exist only on one compute domain. Call the `SMACseAddElemCollection` function if the interface region contains elements on the compute rank.

Registering the Mesh Object

When all node and element collections are added to the mesh object, use the `SMACseRegisterMesh` function to register the mesh. The `SMACseRegisterMesh` function takes the following argument:

`iRegion` The name of the region.

At this stage, you cannot modify the mesh object anymore. However, you can define and register additional interface regions.

When you embed the SIMULIA Co-Simulation Services in parallel mode, register the mesh on each compute domain that processes part of the interface.

Example

The following example illustrates how to create and register an interface region:

```
//
// Create a mesh object with name “Mesh A”
//
SMACseMesh* mesh = SMACseCreateMesh("Mesh A");

//
// Add a node collection with nodes numbered sequentially between 1 – 9.
// Node labels do not need to be defined in any particular order.
//
unsigned int numNodes = 9;
unsigned int labels[ ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
double coords[ ] = { 0., 0., 0.,
                     1., 0., 0.,
                     2., 0., 0.,
                     2., 1., 0.,
                     0., 2., 0.,
                     1., 2., 0.,
                     2., 2., 0. };


SMACseAddNodeCollection( mesh, "myNodeCollection", SMACseDim_3d, numNodes,
labels, coords);
//
// Add element collections with 2 quads and 4 triangles
// Element labels do not have to be defined in any particular order.
unsigned int elemLabels_1 [ ] = { 1, 5 };
unsigned int elemLabels_2 [ ] = { 3, 4, 10, 7 };

unsigned int connectivity_1[ ] = { 2, 5, 4, 1,
                                   5, 8, 7, 4 };

unsigned int connectivity_2[ ] = { 2, 3, 5,
                                   3, 6, 5,
                                   5, 6, 8,
                                   6, 9, 8 };

SMACseAddElemCollection( mesh, "MyQuads",     "MyNodeCollection",
   "SMACseSurfTopo_Quad4", 2, elemLabels_1, connectivity_1);
```

```
SMACseAddElemCollection( mesh, "MyTriangles", "MyNodeCollection",
   "SMACseSurfTopo_Tri3",  4, elemLabels_2, connectivity_2);


//
// Register Mesh
//
SMACseRegisterMesh( mesh );
```

Creating Field Types and Registering Fields

A field is a field type (for example, displacement, temperature, heat flux, etc.) spanning a region. A field has a causality; it can be incoming or outgoing. Incoming fields are required by the application to advance its solution, while outgoing fields are computed and are sent to the remote applications.

At the client level, a field type is associated with a physics property, a unit dimensionality, a position, an algebraic type, and a data type.

A field has either intensive or extensive physics properties:

- Extensive fields are integrated values over the surface or volume of the body, where the amount of matter within the body determines the field values.

  Examples of extensive fields include force, concentrated heat flux, and mass that define a total quantity.
- Intensive fields are fields where the amount of matter within the body does not influence the field values.

  Examples of intensive fields include pressure, surface heat flux density, and mass density, which are quantities defined per unit area or volume. Displacement, velocity, and temperature are also examples of intensive fields because these fields do not depend on the amount of matter within the body.

The position defines the mesh location where the field is evaluated. For example, velocity is evaluated at nodes for structural finite element solvers, while velocities are typically evaluated at element centroids for computational fluid dynamics finite volume solvers.

The data type specifies whether the field is single or double precision. Only double precision is supported.

At the API level, a field is an association between a field type and a region. You must register the field as either incoming or outgoing. To define fields:

- create a field type definition,
- associate the field type with a region, and
- declare the field causality.

Creating a Field Type

The SIMULIA Co-Simulation Services provides two methods for creating field types: a general method where you can define any field type, and a predefined field method where the physics property and unit dimensionality of commonly used field types is registered.

The field reference name is case sensitive and must match the field type name used in the Co-Simulation Configuration file. The field type names between clients do not have to be the same (for example, a structural solver might declare a velocity field with the reference name “V,” while a CFD solver might declare it as “velocity”). An association is made when defining the connection in the Co-Simulation Configuration file (see *<connection> Element*).

Creating a General Field Definition

Use the `SMACseCreateGeneralFieldDefinition` function to create a general field definition. The `SMACseCreateGeneralFieldDefinition` function takes the following arguments:

iFieldType Field type reference name.

This name is case sensitive and must match the field type name in the Co-Simulation Configuration file.

iProperty Field type physics property.

Set to `SMACseFieldProperty_IntensiveFlux` if the field has intensive physics properties, and set to `SMACseFieldProperty_ExtensiveFlux` if the field has extensive physics properties.

iPosition Position where the field is evaluated on the mesh.

Set to `SMACseFieldPos_atNode` for fields evaluated at nodes, and set to `SMACseFieldPos_atElemCentroid` for fields evaluated at the element centroid.

iAlgebraicType Algebraic type of the field. Set to:

- `SMACseFieldAlgType_Scalar` for scalar quantities
- `SMACseFieldAlgType_Vector2d` for 2D vector quantities
- `SMACseFieldAlgType_Vector3d` for 3D vector quantities
- `SMACseFieldAlgType_Tensor2d` for 2D tensor quantities
- `SMACseFieldAlgType_Tensor3d` for 3D tensor quantities

iDataType Defines the field data type. Set this argument to `SMACseFieldDataType_Double` for double precision.

iUnitDim Unit dimensionality of the field.

Set to a character string specifying the unit dimensionality in terms of the fundamental units; for example, velocity is defined as “LT-1,” where the base fundamental and supplementary unit designators are outlined in *Table 2*.

Table 2: Field dimensionality designator.

Symbol	Unit
T	time
L	length
M	mass
E	electric current
K	temperature
A	amount substance
C	candela
R	angle

Creating a Predefined Field Definition

Use the `SMACseCreateFieldDefinition` to define a predefined field type as listed in *Appendix: Predefined Field Types*. The `SMACseCreateFieldDefinition` function takes the following arguments:

`iFieldType` Field type reference name.

This name is case sensitive and must match the field type name in the Co-Simulation Configuration file.

`iPosition` Position where the field is evaluated on the mesh.

Set to `SMACseFieldPos_atNode` for fields evaluated at nodes, and set to `SMACseFieldPos_atElemCentroid` for fields evaluated at the element centroid.

`iAlgebraicType` Algebraic type of the field. Set to:

- `SMACseFieldAlgType_Scalar` for scalar quantities
- `SMACseFieldAlgType_Vector2d` for 2D vector quantities
- `SMACseFieldAlgType_Vector3d` for 3D vector quantities
- `SMACseFieldAlgType_Tensor2d` for 2D tensor quantities
- `SMACseFieldAlgType_Tensor3d` for 3D tensor quantities

`iDataType` Field data type.

Set to `SMACseFieldDataType_Double` for double precision.

When you embed the SIMULIA Co-Simulation Services in parallel mode, you must register the fields on all compute domains associated with the field.

Associating the Field Type with a Mesh, and Declaring Its Causality

To declare a field, you must assign a field type to a region and then declareits causality, whether it is incoming or outgoing.

When you embed the SIMULIA Co-Simulation Services in parallel mode, you must associate the field type with a mesh on all compute domains associated with the field.

Incoming Fields

Use the `SMACseRegisterIncomingField` function to register an incoming field. The `SMACseRegisterIncomingField` function takes the following arguments:

`iFieldType` Field type reference name.

`iRegion` Interface region reference name.

Outgoing Fields

Use the `SMACseRegisterOutgoingField` function to register an outgoing field. The`SMACseRegisterOutgoingField` function takes the following arguments:

`iFieldType` Field type reference name.

`iRegion` Interface region reference name.

Example

The following example illustrates how to define fields:

```
//
// Field definitions for pressure and displacement
//
SMACseCreateFieldDefinition( "traction_vector",
                             SMACseFieldPos_atElemCentroid,
                             SMACseFieldAlgType_Vector3d
                             SMACseFieldDataType_Double );
SMACseCreateFieldDefinition( "displacement",
                             SMACseFieldPos_atNode,
                             SMACseFieldAlgType_Vector3d,
                             SMACseFieldDataType_Double );

//
// Register Incoming and Outgoing Fields
//
SMACseRegisterIncomingField( "displacement",   "Mesh A" );
SMACseRegisterOutgoingField( "traction_vector","Mesh A" );

//Register the co-simulation model
SMACseRegisterModel( );
```

Registering the Interface Model

Once all the interface regions and fields are registered, you need to register the co-simulation model with the SIMULIA Co-Simulation Services. At this point the interface model is finalized, and you can proceed to the computation phase.

Use the `SMACseRegisterModel` function to register the interface model. The `SMACseRegisterModel` function takes no arguments.

When you embed the SIMULIA Co-Simulation Services in parallel mode, you must call `SMACseRegisterModel` collectively on all compute ranks, whether or not it contains an interface region.

Control API

The control API is responsible for sequencing the coupled simulation, ensuring synchronization while advancing the coupled simulation.

The control API should be called after defining the interface model.

The control API is embedded in a client in an opaque manner; that is, it is embedded in the client independent of how many clients are part of the coupled simulation and what numerical algorithms are employed (all of which is controlled through the SIMULIA Co-Simulation Services Director).

The control API can be split into two main tasks:

- exchange of initial conditions at the start of the coupled simulation; and
- solution advancement of the coupled simulation during the time integration loop.

Initial Conditions

Initial conditions need to be exchanged at the start of a coupled simulation. Use the `SMACseInitialConditionsStart` and `SMACseInitialConditionsEnd` functions to exchange initial conditions. Both functions take no arguments.

All fields that are exchanged must be defined using the initial condition category (*Connection Groups*). You must call both functions whether or not initial conditions are exchanged.

In general, `SMACseInitialConditionsStart` sends the client's initial outgoing fields, and `SMACseInitialConditionsEnd` receives a remote client's fields.

In some cases, a client might need the remote client's initial fields to compute its initial conditions. In this case, you can use the `<initCondSeqIsSendRecv>` element in the Configuration file (see *Component Instances*) to reverse the send and receive operations. Only one of the clients can receive initial fields first.

When you embed the SIMULIA Co-Simulation Services in parallel mode, you must call these functions collectively on all compute ranks.

Example

The following example illustrates the exchange of initials conditions:

```
//
// After the initial fields have been computed by the application, exchange
// the initial fields at the start of the simulation.
//
if ( SMACseInitialConditionsStart(  ) ==
SMACseSolutionStatus_InitialConditionsReceived) {
   // May use SMACseGetField to retrieve fields from CSS.
};
                    :
if ( SMACseInitialConditionsEnd(  ) ==
SMACseSolutionStatus_InitalConditionsReceived) {
     // May use SMACseGetField to retrieve fields from CSS
}
```

Solution Advancement

A coupled simulation consists of a series of coupling steps. The coupling step is the period between two consecutive exchanges, which directly influences the solution stability and accuracy.

The coupling step size can evolve during the simulation, taking into account a client's solution stability and accuracy. A client can march in lockstep manner, or it can subcycle. In lockstep, the client's increment size must be equal to the coupling step size that the SIMULIA Co-Simulation Services Director specifies. When subcycling, a client can take one or more increments per coupling step.

For every client's increment, whether or not a coupling step is reached, the client must call the following functions in the order listed below:

SMACseGetTargetTime	Synchronizes and establishes a new coupling step size if required.
SMACseNotifyStart	Notifies the SIMULIA Co-Simulation Services Director it is ready to begin an increment.
SMACseNotifyEnd	Notifies the SIMULIA Co-Simulation Services Director that it has completed an increment.

Optionally, implicit solvers can call the `SMACseNotifyIteration` function to enable implicit-iterative coupling.

Synchronizing and Establishing the Next Coupling Step Size

Use the `SMACseGetTargetTime` function to synchronize and establish the next coupling step size and target time. The `SMACseGetTargetTime` function takes the following arguments:

iIncrStartTime	Start time of the client's increment.
iClientDt	Client's preferred time step size for the next increment.
oTargetTime	Target time for the current coupling step. A new target time is computed when a coupling step is reached. Otherwise, the existing target time is returned while the client is subcycling.
oPreferredDt	Preferred time step size that the SIMULIA Co-Simulation Services Director suggests. For most negotiation methods, `oPreferredDt` never exceeds the `iClientDt` value. However, it can be smaller than the `iClientDt` value when the remote client enforces a smaller coupling step size for a certain negotiation method or when the SIMULIA Co-Simulation Services Director smooths out increment sizes such that they are evenly distributed when the client is subcycling.
oLockstep	If the value is one, the client needs to use `oPreferredDt` for its increment size to march in lockstep. If the client cannot adhere to `oPreferredDt`, it must raise an error and end the simulation. The lockstep behavior is controlled through the Co-Simulation Configuration file (see *Component Instances*).
ioControls	Control flags array that specifies controls at the beginning of a coupling step to be enforced at the upcoming coupling step target time. Set `ioControls[0]` to 1 to synchronize restart at the next coupling step. If `ioControls[0]` is 1 on return, write restart information at the upcoming target time. The `ioControls` argument is a vector..

The `SMACseGetTargetTime` function returns a solution status indicating whether the client has reached a target time, the client is subcycling, the co-simulation event is exceeded, or a system error occurred.

When you run in serial mode, `SMACseGetTargetTime` is called from the main process if the client is parallel. If you run in parallel mode, call `SMACseGetTargetTime` collectively on all ranks. In this case, make sure that `iIncrStartTime` and `iClientDt` agree on all compute ranks.

Notifying that the Client is Ready to Start Its Increment

Use the `SMACseNotifyStart` function to notify the SIMULIA Co-Simulation Services Director that the client is ready to start its next increment. This function is blocking at target time; the client's execution is blocked until all incoming fields are available for it to proceed to the next coupling step. The `SMACseNotifyStart` function takes no arguments.

The `SMACseNotifyStart` function returns a solution status indicating whether the client has reached a target time, the client is subcycling, the co-simulation event is exceeded, or a system error occurred.

In serial mode, you can use this flag to distribute incoming fields to all compute ranks if the application is parallel.

When you run in serial mode, you should call the `SMACseNotifyStart` function from the main process if the client is parallel. If you run in parallel mode, call `SMACseNotifyStart` on all compute ranks.

Notifying that the Client Completed Its Increment

Use the `SMACseNotifyEnd` function to notify the SIMULIA Co-Simulation Services Director that the client has completed the current increment. Most coupling algorithms do not receive fields at `SMACseNotifyEnd`; therefore, it is typically non-blocking.

The `SMACseNotifyEnd` function takes the following argument:

`iIncrEndTime`	Increment end time.

The `SMACseNotifyEnd` function returns a solution status indicating whether the client has reached a target time, the client is subcycling, the co-simulation event is exceeded, or a system error occurred.

When you run in serial mode, you should call `SMACseNotifyEnd` from the main process if the client is parallel. If you run in parallel mode, call `SMACseNotifyEnd` on all compute processes. In this case, the client must make sure that `iIncrEndTime` is the same on all compute ranks.

Notifying that the Client Completed an Implicit Coupling Iteration

Implicit-iterative coupling employs multiple exchanges at the coupling step target time until global convergence is achieved, before proceeding with the next coupling step.

Implicit-iterative coupling is relevant between implicit solvers only.

Call the `SMACseNotifyIteration` function when the client’s solution has converged. This function is blocking at target times; the execution is blocked until all incoming fields are updated to continue with additional coupling iterations. The `SMACseNotifyIteration` takes the following arguments:

`iIncrEndTime`	Increment end time.
`iTerminateIteration`	Set to 1 to end the coupling iteration before the exchange limit is reached or tolerances are achieved.

The `SMACseNotifyIteration` function returns a solution status indicating whether the client should perform further coupling iterations, whether to complete the time step, or a system error occurred.

The `SMACseNotifyIteration` function is called after the local convergence (the client’s convergence criteria) is achieved. If the return status indicates to perform further iterations, the incoming fields are corrected based on the latest outgoing fields. In this case, continue performing further iterations with the new incoming fields. If the return status is to complete the increment, finalize the increment and proceed to `SMACseNotifyEnd` to complete the coupling step.

There are several mechanisms to end the coupling iterations:

- the simplest method is to specify the maximum exchanges in the Co-Simulation Configuration file;

- any client can end the iteration by setting `iTerminateIteration` to 1; and
- specify convergence tolerances in the Co-Simulation Configuration file when the advanced implicit-iterative coupling algorithm (see *Interface Treatment for Implicit-Iterative Coupling* is used. The maximum number of exchanges is still in effect.

When you run in serial mode, you should call `SMACseNotifyIteration` from the main process if the client is parallel. If you run in parallel mode, call `SMACseNotifyIteration` on all compute processes. In this case, the client must make sure that `iIncrEndTime` and `iTerminateIteration` are the same on all compute ranks.

Example

The following example illustrates the control API:

```
//
// Iteration or time step loop
//
while (SMACseGetTargetTime(t, dt, &targetTime, &dT, &lockstep, controlFlags)
!= == SMACseSolutionStatus_CosimulationEventExceeded)
{
   // Notify CSE that you are ready to start an time step
   status = SMACseNotifyStart( );

// iteration loop
   do {
      // get field at end of increment
                :
      // Compute solution
                :
    } while (SMACseNotifyIteration(t+dt,terminate) ==
SMACseSolutionStatus_DoFurtherIterations);
}
```

Data API

The data API is responsible for transferring field data between the client and the SIMULIA Co-Simulation Services.

You access a field by specifying the interface region, collection, and field type that defines the field.

The field values are specified in a memory buffer in the order in which the members (nodes or elements) are defined within a collection. The memory required to store the field depends on the number of members in the collection, the multiplicity, and the data type.

The algebraic type specifies whether the field is a scalar, vector, or tensor field.

The algebraic type, along with the spatial dimension, defines a field's multiplicity. The multiplicity is the number of field values stored per member.

All vector and tensor fields are transferred in the global Cartesian coordinate system. The client is responsible for providing the proper transformation to and from the global Cartesian coordinate system.

There are two data transactions:

- transferring field data from the SIMULIA Co-Simulation Services to the client; and
- transferring field data from the client to the SIMULIA Co-Simulation Services.

Transferring Field Data from the Co-Simulation Services to the Client

Use the `SMACseGetField` function to retrieve an incoming field from the SIMULIA Co-Simulation Services. The `SMACseGetField` function takes the following arguments:

`iFieldName` Field type of the incoming field.

`iRegionName` Region name that the field spans.

`iCollectionName` Node or element collection name depending on whether it is a nodal- or element-based field.

If the collection name is not specified for nodal fields, the field data for all nodes on the region is provided.

`iNMembers` Number of members in `iMembers`. This argument is reserved for future use.

`iMembers` Member list. This argument is reserved for future use.

`iTimeStamp` Time the field is to be retrieved. Field values are interpolated between the previous coupling step and the target time.

The time should be between the previous target time and the current target time.

`oAppBuffer` Address of the client's memory buffer into which to copy field data.

`iMaxSizeInBytes` Memory buffer size in bytes.

`oMissingMembers` Byte array indicating missing members.

`iFieldName`, `iRegionName`, and `iCollectionName` identify the subset of the field.

The client must allocate a memory buffer for `oAppBuffer`, such that the SIMULIA Co-Simulation Services can copy the field data to the client's memory. The maximum size parameter, `iMaxSizeInBytes`, ensures that the SIMULIA Co-Simulation Services does not overwrite your memory. The required memory is the product of the number

of members in the collection, the multiplicity of the field type, and the data type size. For example, assume you have a nodal 3D vector field using double precision. If there are 20 nodes in the collection, the buffer size needs to be 20 members × 3 values per member × 8 bytes for double precision.

Missing members are nodes or elements for which the mapper neighborhood search failed; therefore, field data cannot be mapped at these locations. Missing member treatment is intended for a small number of failures. Depending on the field type, there are various ways of providing a localized treatment. For example, when mapping displacements, you can choose to interpolate values from neighboring nodes; when mapping fluxes, you can assign a zero value at a local spot without sacrificing global conservation across the interface.

There are several methods of treating missing members:

- If `oMissingMembers` is NULL and the Co-Simulation Configuration file does not specify how to treat missing members, an error is issued and the simulation is ended.
- If `oMissingMembers` is NULL and the Co-Simulation Configuration file specifies how to treat missing members, the appropriate treatment is applied. For more information, see *Mapping Specification*.
- If an `oMissingMembers` vector is provided, its entries are set to one to indicate the member is missing members. In this case, a warning is issued and the client is responsible for providing a field value for the missing member.

Example

The following example illustrates the `SMACseGetField` function:

```
                                                :
        unsigned int buffSizeInBytes = nInterfaceDof*sizeof(double);
        SMACseGetField( "force", "Mesh B", "", mesh_B.getNumNodes(),
mesh_B.labels(), t+dT, f_b, buffSizeInBytes, NULL);
                                                :
```

Transferring Field Data from the Client to the Co-Simulation Services

The SIMULIA Co-Simulation Services calls the client's `putFieldFunction` to copy field data from the client's memory to the SIMULIA Co-Simulation Services memory buffer. `putFieldFunction` is a call-back function; it is a function that you need to write to copy your application's field data to the SIMULIA Co-Simulation Services memory.

Both `putFieldFunction` and `userData` are registered when you establish a connection with the SIMULIA Co-Simulation Services (see *Establishing Connections*). The `userData` variable is an opaque pointer to an application's data object. You can define `userData` to provide access to your client's data without having to use global variables.

`putFieldFunction` is called by the SIMULIA Co-Simulation Services for each outgoing field on a collection basis. `putFieldFunction` needs to fill the SIMULIA Co-Simulation Services memory buffer with the field values in the order in which the members are defined in the collection.

Example

The following example illustrates the `putFieldFunction` function:

```
void myPutField(const char* fieldType, const char* meshName, const char*
collectionName,
                const unsigned int nMembers, int* members, void* targetBuffer,
 void * userData)
{
    double* target = (double *) targetBuffer;
    if (strcasecmp(fieldType,"pressure") == 0) {
      for (unsigned int i1 = 0; i1 < nMembers; i1++) target[i1] =  ….
           // copy pressure from application memory space
```

```
          // you can use userData to get access to the application's pressures


    } if (strcasecmp(fieldType,"temperature") == 0) {
      for (unsigned int i1 = 0; i1 < nMembers; i1++) target[i1] =  ….
           // copy temperature from application memory space
          // you can use userData to get access to the application's pressures

    }
}
```

Message Handler

Message Handler Callback Function

The client must register a message handler with the SIMULIA Co-Simulation Services (see *Establishing Connections*). The message handler is a callback function called by the SIMULIA Co-Simulation Services to convey information, warning, and error messages to the client's message handler.

Member labels are provided separately from the message. This allows the client to translate the node or element numbers into a different number or label space, if needed, prior to presenting them to the user or to graphically present them in a CAE environment.

The message handler takes the following arguments:

`iSeverity` Indicates whether this is an information, warning, or error message.

`iMsg` Constant character string containing the message.

`iNumNodes` If the message pertains to nodes, defines how many nodes are returned in the `nodes` argument.

`iNodes` List of nodes.

`iNumElements` If the message pertains to elements, defines how many elements are returned in the `elements` argument.

`iElements` List of elements.

Example

The following example illustrates a basic message handler:

```
void myMessageHandler ( const SMAMpfMessageHandlerSeverity severity,
                        const char* msg,
                        const int numNodes,
                        const int* nodes,
                        const int numElements,
                        const int* elements)
{
    switch (severity) {
      case SMACseMsgHandler_InformationalMessage:  // -- Informational Message,
 continue execution
            if (numNodes > 0) {
                // convert nodes to client's user node labels if needed,
               // print message and node labels using client's message routine

                // possible propagate message and nodes to CAE application for
 visualization
            }else if (numElems > 0) {
                // convert elements to client's user element labels if needed
(e.g. element id and face id for surface facets),
                // print message and element labels using client's message
routine
                // possible propagate message and elements to CAE application
for visualization
            } else {
```

```
                // print message only.
            }
            break;
       case SMACseMsgHandler_WarningMessage:  // -- Warning Message, continue
execution
           if (numNodes > 0) {
                // convert nodes to client's user node labels if needed,
               // print message and node labels using client's message routine

                // possible propagate message and nodes to CAE application for
 visualization
            }else if (numElems > 0) {
                // convert elements to client's user element labels if needed
(e.g. element id and face id for surface facets),
                // print message and element labels using client's message
routine
                // possible propagate message and elements to CAE application
for visualization
            } else {
                // print message only.
            }
       case SMACseMsgHandler_ErrorMessage:  // -- Error Message, terminate
execution
            if (numNodes > 0) {
                // convert nodes to client's user node labels if needed,
               // print message and node labels using client's message routine

                // possible propagate message and nodes to CAE application for
 visualization
            }else if (numElems > 0) {
                // convert elements to client's user element labels if needed
(e.g. element id and face id for surface facets),
                // print message and element labels using client's message
routine
                // possible propagate message and elements to CAE application
for visualization
            } else {
                // print message only.
            }
            break;
   }
```

Add-On Services

The add-on services provide additional functionality beyond the core services.

In this section:

- *System-Level Modeling*
- *Parallel Interface Definition*
- *Importing a Remote Client's Mesh*
- *General Services for Advanced Coupling Algorithms*

System-Level Modeling

System-level modeling refers to the modeling of systems that can include both physical (for example, structural, thermal, electromagnetic, etc.) and logical systems.

Logical modeling refers to a large class of modeling abstractions often encountered in engineering practice. You can designate a part of a system as using a logical modeling abstraction when most (if not all) of the geometry of the part is removed.

Because there is no geometry associated with a logical component, we refer to the inputs and outputs as signals. A signal is analogous to a field; however, it is a single scalar value and lacks geometric information. We use the term actuator to represent an incoming signal and sensor to represent an outgoing signal. The association between signals between clients is done by name referencing in the Co-Simulation Configuration file. Signals use the top-down interface as discussed in *<component> Element*.

The SIMULIA Co-Simulation Services support logical models defined by a Functional Mockup Unit (FMU) conforming to the Functional Mockup Interface (FMI) defined by the Modelisar organization. The SIMULIA Co-Simulation Services support the FMI 1.0 and FMI 2.0 standards. The FMU has to be of type co-simulation. For more information on the FMI standard or for a list of tools that provide the FMU file export capability, see *http://fmi-standard.org*.

The header file, `SMACseApiSignal.h`, defines additional services required to incorporate communication with logical systems. You should call these services after establishing the connection and before registering the co-simulation model.

The signal interface is serial; therefore, the client should call the function described in this section from the main process.

Creating a Signal Interface Class

Use the `SMACseCreateSignalInterface` function to create a signal interface class that contains all the signals. The

`SMACseCreateSignalInterface` takes the following arguments:

`iName` Name of the signal interface class.

`iPutSensorFn` Pointer to `putSensor`callback function.

`iUserData` Opaque pointer where you can define a user object to retrieve sensor data from your applications.

Adding Sensors to a Signal Class

Use the `SMACseAddSensor` function to add a sensor to the signal interface class. The `SMACseAddSensor` function takes the following arguments:

`iHd` Signal interface handle.

`iSensorName` Sensor name.

Adding an Actuator to a Signal Class

Use the `SMACseAddActuator` function to add an actuator to the signal interface class. The `SMACseAddActuator` function takes the following arguments:

iHd Signal interface handle.

iActuatorName Actuator name.

Registering the Signal Class

Use the `SMACseRegisterSignalInterface` function to register the signal interface after you have registered all sensors and actuators. The `SMACseRegisterSignalInterface` function takes no arguments.

Data Handling for Actuators

Use the `SMACseGetActuator` function to retrieve the value of an actuator. The `SMACseGetActuator` function takes the following arguments:

iActuatorName Name of the actuator to retrieve its value.

iTimeStamp Simulation time to extract the signal.

oValue Scalar double value of the signal.

Data Handling of Sensors

The `putSensorFunction` is a callback function, called by the SIMULIA Co-Simulation Services, to copy the sensor data from your application to the SIMULIA Co-Simulation Services memory buffer.

The `putSensorFunction` function has the following arguments:

iSensorName Sensor's name.

oValue Scalar value of the sensor.

iUserData Opaque pointer to user data object to retrieve sensor information.

This argument is initialized with`SMACseCreateSignalInterface`.

Example

The following example illustrates how to create a signal class and register sensors and actuators:

```
SMACseSignalInterface* signalInterfaceHd = SMACseCreateSignalInterface("Signal",
 myPutSensor, &solutionVars);

SMACseAddActuator(signalInterfaceHd, "Displacement at Mass 2");
SMACseAddActuator(signalInterfaceHd, "Velocity at Mass 1");
SMACseAddSensor(signalInterfaceHd,   "Compensation at Mass 1");
SMACseAddSensor(signalInterfaceHd,   "Compensation at Mass 2");
SMACseRegisterSignalInterface(signalInterfaceHd);

SMACseRegisterModel();
```

Parallel Interface Definition

You must provide an implementation of the parallel interface to embed the SIMULIA Co-Simulation Services in parallel.

You can embed the SIMULIA Co-Simulation Services in distributed memory parallel (DMP) mode if your application is parallel and uses the Message Passing Interface (MPI).



Warning: This feature is intended for advanced users only who are familiar with MPI.

We recommend that you embed and test the SIMULIA Co-Simulation Services in serial mode first, and then progress to parallel mode once the serial implementation works.

The header file `SMACseDmp.h` provides the interface definition for a subset of MPI used by the SIMULIA Co-Simulation Services. You must provide an implementation based on your client's MPI.

You must compile the implementation and create a dynamic library (`.dll` on Windows and `.so` on Linux). This library must be included with your client. When you initialize the SIMULIA Co-Simulation Services, you must provide the path and file name as an argument to the `SMACseInitialize` function (see *Initialization*).

Initialization and Finalization of the DMP Interface Implementation

Provide an implementation of the `SMACseDmpInitialize` interface. This function gets called at the start of the coupled simulation. It should duplicate your application's MPI_COMM and initialize any parameters for your DMP interface implementations. The `SMACseDmpInitialize` function takes no arguments.

Provide an implementation of the `SMACseDmpFinalize` interface. This function gets called at the end of the coupled simulation. It should clean-up memory, if any, used by your DMP implementation. The `SMACseDmpFinalize` function takes no arguments.

Get the Process Rank and Number of Processes

Provide an implementation of the `SMACseDmpGetRank` interface. This function returns the rank of the process from which it is called. The `SMACseDmpGetRank` function takes no arguments.

Provide an implementation of the `SMACseDmpGetNumCpus` interface. This function returns the number of CPUs used by the application. The `SMACseDmpGetNumCpus` function takes no arguments.

Non-Blocking Point-to-Point Send and Receive

Provide an implementation of the `SMACseDmpIsend` interface. This function corresponds to the non-blocking point-to-point send, `MPI_Isend`. The `SMACseDmpIsend` function takes the following arguments:

`iBuff`	Address of send buffer.
`iCount`	Number of elements in send buffer.
`iDatatype`	Datatype enumerator.
`iDest`	Rank of destination.
`iTag`	Message tag.
`oRequest`	Request handle.

The `SMACseDmpIsend` function returns 0 if OK; otherwise, it returns 1.

Provide an implementation of the SMACseDmpIrecv interface. This function corresponds to the non-blocking point-to-point receive, MPI_Irecv. The SMACseDmpIrecvfunction takes the following arguments:

iBuff	Address of receive buffer.
iCount	Number of elements in send buffer.
iDatatype	Datatype enumerator.
iSource	Source rank.
iTag	Message tag.
oRequest	Request handle.

The SMACseDmpIrecv function returns 0 if OK; otherwise, it returns 1.

Blocking Point-to-Point Send and Receive

Provide an implementation of the SMACseDmpSend interface. This function corresponds to the blocking point-to-point send, MPI_Send. The SMACseDmpSend function takes the following arguments:

iBuff	Address of send buffer.
iCount	Number of elements in send buffer.
iDatatype	Datatype enumerator.
iDest	Rank of destination.
iTag	Message tag.

The SMACseDmpSend function returns 0 if OK; otherwise, it returns 1.

Provide an implementation of the SMACseDmpRecv interface. This function corresponds to the blocking point-to-point receive, MPI_Recv. The SMACseDmpRecv function takes the following arguments:

iBuff	Address of receive buffer.
iCount	Number of elements in send buffer.
iDatatype	Datatype enumerator.
iSource	Source rank.
iTag	Message tag.
oStat	Status.

The SMACseDmpRecv function returns 0 if OK; otherwise, it returns 1.

Additional MPI Functions for Testing Blocking and Non-Blocking Tasks

Provide an implementation of the SMACseDmpIprobe interface. This function corresponds to the non-blocking test, MPI_IProbe. The SMACseDmpIprobe function takes the following arguments:

iSource	Source rank or CSEDMP_ANY_SOURCE .
iTag	Tag value or CSEDMP_ANY_TAG .
oFlag	Equal to 1 if a message with the specified tag received.
oStat	Status.

Provide an implementation of the `SMACseDmpTestall` interface. This function tests for the completion of all previously initiated requests, `MPI_Testall`. The `SMACseDmpTestall` function takes the following arguments:

iCount	List length.
iReqIndxs	Array of requests.
oFlags	Equal to 1 if all requests have been completed.
oStat	Status.

Provide an implementation of the `SMACseDmpWaitall` interface. This function waits for the completion of all previously initiated requests, `MPI_Waitall`. The `SMACseWaitall` function takes the following arguments:

iCount	List length.
iReqIndxs	Array of requests.
oStat	Status.

Broadcast, Gather, and Reduce

Provide an implementation of the `SMACseDmpBcast` interface. This function broadcasts a message from the process with rank "root" to all other processes on the communicator, `MPI_Bcast`. The `SMACseDmpBcast` function takes the following arguments:

ioBuff	Buffer address.
iCount	Number of elements in send buffer.
iDatatype	Datatype enumerator.
iRoot	Rank of broadcast root.

Provide an implementation of the `SMACseDmpGather` interface. This function gathers data from every process on the root process. The `SMACseDmpGather` function takes the following arguments:

iSendBuff	Send buffer address.
iSendCount	Number of elements in send buffer.
iSendtype	Send datatype enumerator.
oRecvBuff	Starting address of the receive buffer on the root process.
iRecvCount	Number of elements in the receive buffer.
iRecvType	Receive datatype enumerator.
iRoot	Rank where the message is received.

Provide an implementation of the `SMACseDmpAllGather` interface. This function gathers data on all ranks. The `SMACseDmpAllGather` function takes the following arguments:

iSendBuff	Send buffer address.
iSendCount	Number of elements in the send buffer.
iSendtype	Send datatype enumerator.
oRecvBuff	Starting address of the receive buffer on the root process.
iRecvCount	Number of elements in the receive buffer.

iRecvType	Receive datatype enumerator.

Provide an implementation of the `SMACseDmpReduce` interface. This function reduces a message at the "root", `MPI_Reduce`. The `SMACseDmpReduce` function takes the following arguments:

iBuff	Send buffer address.
oBuff	Receive buffer address.
iCount	Number of elements in the send buffer.
iDatatype	Datatype enumerator.
iOp	Operation enumerator.
iRoot	Rank where message is reduced.

Importing a Remote Client's Mesh

A client can bypass the SIMULIA Co-Simulation Services mapper and provide its own mapping services.

In some cases, the client might prefer to perform its own field mapping and to bypass the SIMULIA Co-Simulation Services mapper. Typical cases include:

- a client using non-standard element types that the SIMULIA Co-Simulation Services mapper does not support; or
- particle and Lattice Boltzmann applications where the mapping is best performed by the solver because these methods use specialized mapping algorithms. In addition, the amount of data transfer is reduced.

The header file, `SMACseApiMesh.h`, defines additional services to import a remote mesh. The services to import a remote mesh are serial; you should call them from the main compute process, whether using serial or parallel mode.

Creating a Local Mesh to Represent the Remote Mesh

Use the `SMACseCreateMesh` function to create a local mesh object (see *Creating a Mesh Object*). This mesh object is used to hold the remote mesh; do not assign any node or element collection to it. The mesh name used by the client importing the remote mesh can be different from the remote mesh name; the association between the meshes is done through the Co-Simulation Configuration file. You must set the `<mapOn>` element to “NOMAP” on the `<connection>` element (see *<connection> Element*).

Use the `SMACseRegisterMesh` function to register the mesh object (see *Registering the Mesh Object*). Then, define the incoming and outgoing fields (see *Creating a Field Type*).

Once you register the interface model (see *Registering the Interface Model*), you can retrieve the remote mesh's node and element topology.

Example

The following example illustrates how to create a local mesh to represent a remote mesh and how to assign incoming and outgoing fields to the local mesh:

```
   // Create local mesh object

   SMACseMesh* mesh = SMACseCreateMesh("myMeshName");

   // Do NOT define any element and node collections

   // Register Mesh
   SMACseRegisterMesh( mesh );

   // You can now define fields on the mesh
   SMACseCreateFieldDefinition( "displacement",    SMACseFieldPos_atNode,

                                SMACseFieldAlgType_Vector3d,
SMACseFieldDataType_Double );
   SMACseCreateFieldDefinition( "force",           SMACseFieldPos_atNode,

                                SMACseFieldAlgType_Vector3d,
SMACseFieldDataType_Double );

    // Register Incoming and Outgoing Fields
    SMACseRegisterOutgoingField( "force",           "myMeshName");
    SMACseRegisterIncomingField( "displacement",    "myMeshName");

    // Register the interface model
    SMACseRegisterInterface();
```

Accessing the Remote Mesh Definition

The header file, SMACseApiMesh.h, defines additional services to retrieve and access the topology information of the remote mesh.

Retrieving the Remote Mesh

After you register the interface model, use the SMACseGetRemoteMesh function to retrieve the remote mesh's topology. The SMACseGetRemoteMesh function takes the following arguments:

iLocalMeshName	Local mesh name representing the remote mesh.
oNNodes	Number of nodes on the remote mesh.
oNElemCollections	Number of element collections on the remote mesh.
oNDim	Dimensionality of the remote mesh. Set to: • SMACseDim_2d for a 2D mesh • SMACseDim_3d for a 3D mesh • SMACseDim_Axi for an axisymmetric mesh

The SMACseGetRemoteMesh function returns a mesh handle to retrieve the topology information.

Unpacking the Remote Mesh Node Collection

Use the SMACseGetNodeCollection function to retrieve the node collection. The remote mesh can have only a single node collection. The SMACseGetNodeCollection function takes the following arguments:

iMeshHd	Mesh handle.
oLabels	Vector containing the node labels.
oCoordinates	Vector containing the nodal coordinates.

The vector size for oLabels and oCoordinates is based on the oNNodes and oNDim provided by SMACseGetRemoteMesh. Nodal field data is transferred in the order in which the nodes are defined in the node collection.

Unpacking the Remote Mesh Element Collections

Use the SMACseQueryElemCollection and SMACseGetElemCollection functions to retrieve the element collection information.

The SMACseQueryElemCollection function takes the following arguments:

iMeshHd	Mesh handle.
iCollection	Collection index. The collection index is 0-based (that is, the first collection has an index equal to 0).
oNElems	Number of elements in the collection.
oNodesPerElem	Number of nodes per element.

The SMACseQueryElemCollection function returns a character string identifying the element topology.

The SMACseGetElemCollection function takes the following arguments:

iMeshHd	Mesh handle.
iCollection	Collection index. The collection index is 0-based (that is, the first collection has an index equal to 0).
oLabels	Vector to node labels.
oConnectivity	Element connectivity.

Element field data is stored the order in which the elements are defined in the collection.

Example

The following example illustrates how to extract the topology information of a remote mesh:

```
   unisgned int nNodes = 0;
   unsigned int nElemCollections = 0;
   unsigned int dimension = 0;
   SMACseMesh* mesh = SMACseGetRemoteMesh( "myMeshName", nNodes,
nElemCollections, dimension);

   // Process the node labels and coordinates
   unsigned int* nodeLabels = (unsigned int *) malloc(nNodes*sizeof(int));
   double* coords = (double *) malloc(nNodes*dimension*sizeof(double));
   SMACseGetNodeCollection (mesh,nodeLabels,coords);

   // loop over all element collection
   for (unsigned int jCollection = 0; jCollection <nElemCollections;
++jCollection) {

       unsigned int nElems = 0;
       unsigned int nNodesPerElem = 0;
       const char* topology = SMACseQueryElemCollection( mesh, jCollection,
&nElems, &nNodesPerElem );

       if (strcmp(topology,"SMACseSurfTopo_Tri3") == 0) {
            unsigend int* Tri3Labels = (unsigned int *)
malloc(nElems*sizeof(int));
            unsigned int* Tri3Connectivity = (unsigned int *)
malloc(nElems*nNodesPerElem*sizeof(int));
            SMACseGetElemCollection ( mesh, jCollection, myDb->Tri3Labels,
myDb->Tri3Connectivity );
       } else if (strcmp(topology,"SMACseSurfTopo_Tri6") == 0) {
                           :
       }
   } // for element collections
```

General Services for Advanced Coupling Algorithms

Registering a Service

Use the `SMACseRegisterService` function to register a call-back function. The `SMACseRegisterService` function takes the following arguments:

`iServiceName`	Service name as specified in the SCXML state chart.
`iFunction`	Service function.
`iUserData`	Opaque point to user data to be accessed by the service function.
`iCollective`	Set to 1 if the function is called collectively on all compute ranks; otherwise, set to 0 to be called by the main process.
`iPersistant`	Set to 0.

You should call `SMACseRegisterService` prior to registering your model.

Example

The following example illustrates how to register services:

```
                              :
  SMACseRegisterService("myServiceTask1",computeLagrangeMultipliers, NULL, 0,
0);
  SMACseRegisterService("myServiceTask2",SolverInterfaceSystem, NULL, 0, 0);
                              :
  SMACseRegisterModel( );
```

Results Provider

A results provider is a process that reads an upstream results file and uses the Co-Simulation Services to transfer and map fields to a downstream analysis.

In this section:

- *Results Provider*
- *Services for Results Provider*
- *Deep Dive into the Results Provider Template*

Results Provider

A results provider is a process that reads an upstream results file and uses the SIMULIA Co-Simulation Services to transfer and map fields to a downstream analysis.

You can use a results provider only with the Abaqus solvers. Abaqus provides a keyword interface and generates the configuration and specification files. The Abaqus execution driver launches the results provider and the SIMULIA Co-Simulation Services Director processes.

In general, a results provider can import fields into any application that supports the SIMULIA Co-Simulation Services; if a solver can import a field for a co-simulation, it can import the field in a sequential analysis with minimal amount of coding effort. Typically, such coding effort encompasses a user interface, authoring the configuration and specification files, and launching the result provider and SIMULIA Co-Simulation Services Director processes.

Developing a Results Provider

Developing a results provider requires familiarity with Abaqus and the SIMULIA Co-Simulation Services API. Results providers are easier to develop compared to embedding the SIMULIA Co-Simulation Services in a solver for the purpose of co-simulation because of the following:

- template code is provided as a starting point;
- results providers require only a subset of the SIMULIA Co-Simulation Services API;
- results providers are generally simpler applications than solvers; and
- you do not need to worry about stability of the segregated solution approach.

However, the programmer should have the expertise outlined in *Intended Users*.

Results Provider Template

A results provider template is provided in the SIMULIA Co-Simulation Services API kit. This template provides the necessary SIMULIA Co-Simulation Services calls. Typically, you need to provide code only for extracting data from your results file, maintaining your data in memory, and casting it to the form required by SIMULIA Co-Simulation Services.

Extract the SIMULIA Co-Simulation Services API kit from the Abaqus media (see *Extracting the SIMULIA Co-Simulation Services API Kit*). Copy the file `ResultsProviderTemplate.cpp` from the `/ResultsProvider` folder, and use it as a starting point to develop a results provider for your application.

If you have Abaqus installed, you can also use the following command to fetch the results provider template:

```
abaqus -fetch -job ResultsProviderTemplate.cpp
```

Release Compatibility

Release compatibility is discussed in *Release Compatibility*. Typically, the results provider is launched from the Abaqus execution driver; therefore, the Abaqus execution driver uses the Co-Simulation libraries provided with the Abaqus release. We recommend updating the Co-Simulation Services with every major Abaqus release and ensuring that the SIMULIA Co-Simulation Services API header files have not changed between releases.

Launching the Results Provider

You can launch the results provider automatically from the Abaqus execution driver, or you can start all processes manually.

Command Line Arguments

The Abaqus execution driver launches the results provider with the arguments specified below. We recommend that your results provider adheres to the same arguments; otherwise, you must start all processes manually.

The following command line arguments are required for automatic launching:

job
The job name of the upstream analysis without its extension.

The job name reflects the value of the FILE parameter (without its extension) on the **EXTERNAL FIELD* option when launched from Abaqus.

file
The results file path and name with extension. You read the upstream results from this file.

The file name reflects the value of the FILEparameter on the **EXTERNAL FIELD* option when launched from Abaqus.

spec
The results provider specification file with its extension. Abaqus generates the specification file.

The file name is the upstream job name with the `_spec` and `.xml` extension. For example, if the upstream job is `UpstreamJob`, the specification file produced by Abaqus is `UpstreamJob_spec.xml`.

csedirector
The host and listener port of the SIMULIA Co-Simulation Services Director.

You must use a colon between the host and port.

timeout
An optional timeout period in seconds.

Launching the Results Provider from Abaqus

The Abaqus execution procedure automatically launches the results provider and the SIMULIA Co-Simulation Services Director. In this case, do not specify the **csedirector** argument on the Abaqus command line. The READER parameter on the **EXTERNAL FIELD* option specifies the results provider executable name. You must provide the path and executable name (including its extension). The Abaqus preprocessor generates the configuration and specification files. The Abaqus execution driver then sets the required environment variables as discussed in *Setting up the Runtime Environment*.

Launching the Results Provider Manually

You can start the results provider manually. This is useful when developing and debugging the results provider, or when your results provider uses different arguments from what the Abaqus execution procedure uses. In this case, provide the **csedirector** argument on the Abaqus command line, and start the SIMULIA Co-Simulation Services Director and the results provider. In addition, you must define the required environment variables as outlined in *Setting up the Runtime Environment*.

Generating the Configuration and Specification Files

Typically, the results provider requires a specification file and the SIMULIA Co-Simulation Services Director requires a Co-Simulation Configuration file. An easy way to author these files is to run an Abaqus **datacheck** execution procedure. The Abaqus**datacheck** procedure generates both files based on the **EXTERNAL FIELD*, **FIELD MAPPER CONTROLS*, and **FIELD OPERATIONS* options. These files are valid as long as you do not make any changes to the Abaqus input file or rename the Abaqus job.

Use the following command to generate the configuration and specification files using the Abaqus**datacheck** execution procedure:

```
abaqus -job downstream -datacheck
```

Running the Analysis

You must start the Abaqus analysis, the SIMULIA Co-Simulation Services Director, and the results provider to run the analysis. The examples below assumes that the results provider uses the same arguments that the Abaqus execution driver uses. The downstream job refers to the analysis importing the results, and the upstream job refers to the analysis exporting the results.

To continue the Abaqus analysis after the **datacheck** using the co-simulation director running on a local host with listener port 91000:

```
abaqus -continue -job downstream -csedirector localhost:91000
```

To start the director process using the configuration file `downstream_config.xml` and the listener port 91000:

```
abaqus -cse -configure downstream_config.xml -listenerport localhost:91000
```

To start `myResultsProvider` reading from the file `upstream.dat` with the specification file `upstream_spec.xml`:

```
myResultsProvider -job upstream -file upstream.dat -spec upstream_spec.xml
-csedirector localhost:91000
```

Services for Results Provider

The required services for results provider are designed for performance and to facilitate the development of a results provider.

The SIMULIA Co-Simulation Services API is defined in *SIMULIA Co-Simulation Services API*. In addition, you need the add-on services defined by the `SMACseApiResultsProvider.h` header file. These add-on services reduce the data transfer and simplify the coding effort for the SIMULIA Co-Simulation Services API.

Data Model

The results provider uses the concept of frames. A frame contains field data at a specific time. Frames do not have to be uniformly distributed in time; however, the frames should be ordered in ascending order of time. Different fields can be stored at different frequencies; therefore, the number of frames can vary between field types. As the downstream application progresses in time, the results provider transfers and maps the field data on demand when the analysis time exceeds the previous transferred frames.

`putFramePtsForField` and `putFieldFnWithTime` are callback functions that you must provide. The functions are written in terms of your data model, but they are called by theSIMULIA Co-Simulation Services. You must register these callback functions.

Registering the Callback Services

Use the `SMACseRegisterSequentialCallback` function to register the callback functions `putFramePtsForField` and `putFieldFnWithTime`.

The `SMACseRegisterSequentialCallback` function takes the following arguments:

`iUserPutFieldFunctionWithTime`	Function pointer to `putFieldFnWithTime`.
`iUserPutFramePointForField`	Function pointer to `putFramePointForField`.

Querying Required Fields

Use the `SMACseQueryRequiredFields` function to obtain a list of fields required from the results provider. This optional utility helps you retrieve essential information from the specification file without having to parse XML.

The `SMACseQueryRequiredFields` function takes the following argument:

`iSpecFile`	File name of the specification file. The specification file name is defined as a command line argument of the results provider.

The `SMACseQueryRequiredFields` function returns a constant character string of a comma separated list of fields. The fields are defined by the field type and region name separated with a period. Multiple fields are separated by commas. For example, `displacement.Region A, force.Region A, temperature.Region B`. You must parse this string and register the regions and fields (see *Defining the Interface Model*).

Results Provider Control API

Use the `SMACseProvideResults` function to notify the SIMULIA Co-Simulation Services Director that the results provider is ready to provide field data. The SIMULIA Co-Simulation Services Director then calls `putFramePtsForField` and `putFieldFnWithTime` to retrieve required information as the consumer application progresses in time. There is no time advancement in the reader process; therefore, the time-based APIs are meaningless.

The `SMACseProvideResults` function takes no arguments.

Results Provider Data Services

The SIMULIA Co-Simulation Services call the `putFramePointsForField` and `putFieldFnWithTime` callback functions to retrieve data from your results provider. These services are called by the SIMULIA Co-Simulation Services when new information is required to advance the downstream simulation.

The `putFramePointsForField` function is called by the SIMULIA Co-Simulation Services to retrieve the frame number at the specified time for a particular field or to retrieve the frame number before and after the specified time for a particular field. The SIMULIA Co-Simulation Services send field data only when the before and after frame time points change. The `putFramePointsForField` function takes the following arguments:

`iFieldType`	Field type name to retrieve the frame time points.
`iMeshName`	Region name for which the frame points are retrieved.
`iMaxNumberOfFrames`	Maximum number of time frames. The value returned is either 1 or 2.
`iTime`	Solution time to extract field data.
`iUserData`	Pointer to user data object. The `userData` object is defined by `SMACseConnect`.
`oNumberOfFrames`	Number of frame time points the client has for the specified time. Set this value either to 1 or 2.
`oFrameTimePoints`	Vector of frame time points.

The `putFieldFnWithTime` function is called by the SIMULIA Co-Simulation Services to retrieve the field at the requested time point. The `putFieldFnWithTime`function takes the following parameters:

`iFieldType`	Field type name to be retrieved.
`iMeshName`	Region name for the field to be retrieved.
`iCollectionName`	Collection name. The collection name is either a node or element collection depending on the field position.
`iNMembers`	Number of members in the collection.
`iMembers`	List of member identifiers.
`iTime`	Solution time to extract the field.
`iUserData`	Pointer to user data object. The `userData` object is defined by the `SMACseConnect` function.
`oTargetBuffer`	Target buffer to which to write field data.

Deep Dive into the Results Provider Template

The results provider templates provides a good starting point to develop your own results provider.

The template illustrates the necessary calls to the SIMULIA Co-Simulation Services. You need to provide the coding to read your results file, manage your data, and cast your data to what the SIMULIA Co-Simulation Services require.

To obtain the results provider template, follow the instructions in *Results Provider Template*. The template is in C++ and uses the C++ Template Library (STL). The C++ STL library makes it easy to manipulate strings and read from a file.

Parsing the Command Line Arguments

When Abaqus launches the results provider, the Abaqus execution driver spawns the results provider with predefined arguments (see *Command Line Arguments*). We recommend that you adhere to these arguments such that you can automate the execution process from Abaqus.

Example

The following example illustrates how to read and parse the command line arguments:

```
int main(int argc, char** argv) {

               :

   // Process command line arguments
   //
   // The Results Provider is invoked by Abaqus with the following arguments.
We recomend using
   // these arguments so that Abaqus can automatically start the Results
Provider. If your
   // choose different arguments, you must start the Results Provider and CSS
Director manually.
   //
   int timeout        = 3600;                 // timeout in seconds; exist and
 error if process do not connect after timeout
   string codeName    = "myResultsProvider";  // INSTRUCTIONS: replace with
executable name without extension.
   string workDir     = ".//";                // CSE diagnostic files will be
written to the current work directory
   string cssDirector = "";                   // "host:port" of CSS director
   string jobName     = "";                   // This is the results file name
 without it extension
   string specFile;                          // Results provider specification
 file generate by Abaqus
   string resultsFile;                       // Results file name with file
extension.

   if ( (argc < 10) || (strcmp(argv[1], "-h") == 0) || (strcmp(argv[1],"-help")
 == 0) ) {
      cout << "Usage: myResultsProvider -job <jobname> -file <results filename>
 -spec <spec filename> -csedirector control_machine:port -timeout seconds" <<
endl;
      exit(0);
   }

   int iArg = 1;
```

```
   while (iArg < argc) {
     string argument = argv[iArg++];
     if (argument == "-csedirector") {
        cssDirector = argv[iArg++];
     } else if (argument == "-timeout") {
        timeout = atof(argv[iArg++]);
  } else if (argument == "-job") {
        jobName = argv[iArg++];
  } else if (argument == "-file") {
        resultsFile = argv[iArg++];
  } else if (argument == "-spec") {
        specFile = argv[iArg++];
     }
   }

              :

}
```

Opening the Results File and Defining Your Database

After parsing the command line arguments, you should check if the results file exists. If it does not exist, report an error and exit.

The template uses a pointer `myDb` to point to any object that contains information about your results file. The `myDb` object can be as simple as a file pointer; or it can be more complex, containing information about your database.

Example

The following example illustrates how to define your database structure:

```
   // INSTRUCTIONS: You can create any structure/class to provide access to
   // your data. Assign this to an opaque handle, which will be passed to
SMACSeConnect.
   // You will then be able to access your database in the myPutFieldFnWithTime
 and
   // myPutFramePtsForField services called by the CSE.
   void* myDb = static_cast<void*>(....);
```

Initializing the SIMULIA Co-Simulation Services

You must initialize the SIMULIA Co-Simulation Services before calling any services. Use the `SMACseInitialize` function to initialize the SIMULIA Co-Simulation Services (see *Initialization*). Typically the results provider is a serial process, and you do not need to provide an implementation of the DMP interface.

Establishing the Connections

Establish the connection with the SIMULIA Co-Simulation Services Director and Abaqus. Use the `SMACseConnect` function to establish the connection (see *Establishing Connections*) .

Example

The following example illustrates how to establish the connections:

```
    // Initialize the CSS Runtime environment
   SMACseInitialize(NULL);
```

```
// Connect Results Provider with CSS Director and Abaqus
double startTime   = 0.0;
int cStatus = SMACseConnectStatus_failure;
cStatus = SMACseConnect( codeName.c_str(), jobName.c_str(), workDir.c_str(),

                         cssDirector.c_str(), timeout,
                         startTime, NULL, myDd, myMessageHandler );
if ( cStatus != SMACseConnectStatus_success ) {
   cerr << "Error: Failed to connect with CSS Director." << endl;
   exit(1);
}
```

Registering the Sequential Callback Services

After establishing the connection, you must register the `putFieldFnWithTime` and `putFramePtsForField` callback functions.

Use the `SMACseRegisterSequentialCallbacks` function to register the functions.

Example

The following example illustrates how to register the sequential callback services:

```
// Register CSS callback functions
SMACseRegisterSequentialCallbacks(PutFieldFnWithTime, PutFramePtsForField);
```

Registering the Interface Model

Register the interface regions and the outgoing fields as discussed in *Defining the Interface Model*. The region and field names that you must register are defined in the results provider specification file. The specification file is an XML document. You can either parse the XML document yourself, or you can use the `SMACseQueryRequiredFields` utility to return a comma-separated string containing the field type and region name separated by a period.

For each region, you define and register a mesh. For each of the fields, you define the field type and register it as an outgoing field. When all the regions and outgoing fields are registered, you must register the interface model using `SMACseRegisterModel`.

Extracting and Sending Field Data

After registering the model, the results provider is ready to extract and send field data.

First, call `SMACseInitialConditionsStart` and `SMACseInitialConditionsEnd` to send the initial fields to the downstream application. Then, call `SMACseProvideResults`. `SMACseProvideResults` is a blocking call and provides the results to the downstream application as the analysis progresses. It returns once the downstream analysis completes. During this call, the `putFieldFnWithTime` and `putFramePtsForField` functions are called, requesting updated information when required.

Example

The following example illustrates the main body exchanging field data:

```
SMACseSolutionStatus  cssStatus;
cssStatus = SMACseInitialConditionsStart( );
cssStatus = SMACseInitialConditionsEnd( );
cssStatus = SMACseProvideResults();
```

Disconnecting and Finalizing the SIMULIA Co-Simulation Services

When `SMACseProvideResults` returns, you can disconnect and finalize theSIMULIA Co-Simulation Services, and then you can end the results provider.

Example

The following example illustrates how to disconnect and finalize the SIMULIA Co-Simulation Services:

```
   SMACseDisconnectExt("Results provider terminated succefully." );
   SMACseFinalize ( );
```

putFieldFnWithTime

The `putFieldFnWithTime` function extracts data from your results file and copies it into a SIMULIA Co-Simulation Services memory buffer to be transferred and mapped to the downstream application. The SIMULIA Co-Simulation Services call the `putFieldFnWithTime` function for each field type, mesh region, and collection for a particular analysis time.

Example

The following example illustrates the use of the `putFieldFnWithTime` function:

```
//
// PutFieldFnWithTime
//
// PutFieldFnWithTime service: This function is called by the CSE. It extracts
 field data from the
// client and copies it to CSE memory for export. Field data is to be copied
in the order the
// members are defined in a collection when you define your mesh.
//
void myPutFieldFnWithTime(const char* fieldType, const char* meshName, const
char* collectionName,
                     unsigned int nMembers, int* members, double* time, void*
userData,
                          void* targetBuffer)
{
    // INSTRUCTIONS:
    //
    // Recast userData to your dabase structure
    double* bufferT = static_cast<double*>(targetBuffer);

    vector<double> &timeFrames = mydbPtr->timeFrames;
    int numFrames = timeFrames.size();

    // if time before first time point, use first time point
    if ( time <= timeFrames[0] )
    {
        // INSTRUCTIONS: copy field data for time point 0 from your memory
       //              to the CSS memory buffer. Field data is in order members

        //               are register in a collection.
        return;
    }
```

```
    // if time is after last time point, use last time point
    if ( time >= timeFrames[numFrames-1] )
    {
        // INSTRUCTIONS: copy field data for the last time point from your
memory
       //             to the CSS memory buffer. Field data is in order members

        //               are register in a collection.
        return;
    }

    // if at time point or between time frame points
    for (int  i =0; i < numFrames - 1; ++i )
    {
        if (time == timeFrames[i] ) {
        // INSTRUCTIONS: copy field data at time point from your memory
       //             to the CSS memory buffer. Field data is in order members

        //               are register in a collection.
            return;
        }
        else if ( time > timeFrames[i] &&  time < timeFrames[i+1])
        {
        // INSTRUCTIONS: interpolate and copy field data from your memory
       //             to the CSS memory buffer. Field data is in order members

        //               are register in a collection.
            return;
        }
    }
}
```

putFramePtsForField

The `putFramePtsForField` function extracts the time points in the database for a given field. You can provide a maximum of two time points. Provide one time point if the time of the results frame matches the given time. Otherwise, provide two time points that sandwich the given time.

Example

The following example illustrates the use of the `putFramePtsForField` function:

```
//
// PutFramePtsForField
//
// PutFramePtsForField service is called by the CSS. For the given field and
time, it extracts the time frames
// points the database. A maximum of only two time frame points can be provided
 if time falls
// between two time frames.

void myPutFramePtsForField(const char* iFieldType, const char* iMeshName, const
 int iMaxNumberOfFrames,
                      int* oNumberOfFrames, double* oframeTimePoints, const
double* time,
                           void* userData)
{

    // Provide one time point if the time of results frame matches the given
```

```
time (*oNumberOfFrames =1),
    // otherwise, provide two time points that sandwich the given time
(*oNumberOfFrames = 2).
    // Fill the oframeTimePoints with time points

    // Recast opaque pointers to proper datapointers; useData was registered
in SMACseConnect
    struct DataBaseStruc *mydbPtr =  static_cast<struct
DataBaseStruc*>(userData);

    vector<double> &timeFrames = mydbPtr->timeFrames;
    int numFrames = timeFrames.size();

    double iTime = *time;
    *oNumberOfFrames = 0;

    // if time before first time point, use first time point
    if ( iTime <= timeFrames[0] )
    {
        *oNumberOfFrames = 1;
        oframeTimePoints[0] = timeFrames[0];
        if (debug) cout << "myPutFramePtsForField: time=" << iTime << " Time
Frame = " << 0 << endl;
        return;
    }

    // if time is after last time point, use last time point
    if ( iTime >= timeFrames[numFrames-1] )
    {
        *oNumberOfFrames = 1;
        oframeTimePoints[0] = timeFrames[numFrames-1];
        if (debug) cout << "myPutFramePtsForField: time=" << iTime << " Time
Frame = " << numFrames-1 << endl;
        return;
    }

    // provide time points at, or before and after
    for (int  i =0; i < (numFrames - 1); ++i )
    {
        double t = timeFrames[i];
        if (abs(iTime - t) < 1.e-6) {
            *oNumberOfFrames = 1;
            oframeTimePoints[0] = t;
            if (debug) cout << "myPutFramePtsForField: time=" << iTime << "
Time Frame = " << i << endl;
            return;
        } else if ( iTime > timeFrames[i] &&  iTime < timeFrames[i+1]) {
            *oNumberOfFrames = 2;
            oframeTimePoints[0] = timeFrames[i];
            oframeTimePoints[1] = timeFrames[i+1];
            if (debug) cout << "myPutFramePtsForField: time=" << iTime << "
Time Frame = " << i << ", " << i+1 << endl;
            return;
        }
    }
}
```

Examples

This section provides examples of embedding the SIMULIA Co-Simulation Services into a client.

The coding examples are kept simple and are intended to illustrate how to embed the Co-Simulation ServicesAPI. In many cases, things have been “hardwired” for the particular example; no error checking is performed, etc.

In this section:

- *Dual Mass-Spring Systems*
- *Aeroelastic Supersonic Flow over Long, Thin-Walled Cylinders*
- *ASCII File Results Provider*

Dual Mass-Spring Systems

This example illustrates how to embed the SIMULIA Co-Simulation Services API in a simple client that computes the dynamic response of a coupled dual mass-spring system.

The dual mass-spring example serves as a good starting point.

You learn how to:

- Embed the SIMULIA Co-Simulation Services API into a client.
- Compile and link the SIMULIA Co-Simulation Services to the client.
- Run a co-simulation between two mass-spring clients.
- Run a co-simulation between a mass-spring client and Abaqus/Standard.
- Author the Co-Simulation Configuration file for the advanced iterative coupling schemes.

This example is designed to illustrate stability issues associated with the segregated solution approach. We encourage you to experiment by changing the stiffness and mass distribution ratios between the subsystems, coupling step sizes, and coupling algorithms.

Problem Description

Figure 1 illustrates the dual mass-spring system. The two mass-spring systems are tied together, such that displacements of both masses are constrained to move together. Each mass-spring system is modeled by a separate simulator: Simulator A and Simulator B. For Simulator A, we provide C++ code. For Simulator B, we provide both C++ code and an Abaqus model.

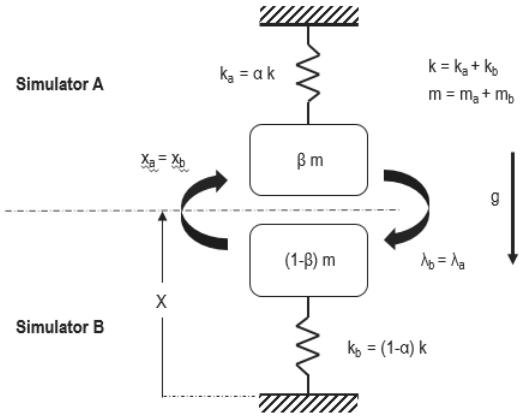


Figure 1: Dual mass spring systems.

Initially, both systems are at rest, and gravity is turned on at the start of the simulation such that the spring system oscillates. We define a constraint force, , between both systems to enforce that both masses move together. The system of equations:
with constraints and initial conditions:

A stiffness distribution factor, , defines the relative stiffness distribution between System A and System B compared to a total stiffness, , for an equivalent single system:

A mass distribution factor, , defines the relative mass distribution between System A and System B compared to a total mass, , for an equivalent single system:

The radial frequency, , is defined as .

Simulator A receives the displacement, , from Simulator B and sets its displacement, . Simulator A then computes the acceleration, , and solves for the interface force, , according to

Simulator B receives the interface force, , from Simulator A and computes the acceleration according to

It then computes the velocity and displacement using Newmark's time integration method. Its displacements, , are then exported to Simulator A.

Compiling the C++ Simulators

If you have not already done so, extract the SIMULIA Co-Simulation Services API kit as described in *Extracting the SIMULIA Co-Simulation Services API Kit*. The compiler needs access to the SIMULIA Co-Simulation Services header files, as discussed in *Compiling your Application with SIMULIA Co-Simulation Services*.

Typical compile options for compiling `DualMassSpring_SystemA.cpp` are as follows:

On Windows:

```
cl /c /W0 /MD /TP /I<location of CSS header files> DualMassSpring_SystemA.cpp
```

On Linux:

```
g++ -c -fPIC -w -Wno-deprecated /I <location of CSS header files>
DualMassSpring_SystemA.cpp
```

Use similar options to compile `DualMassSpring_SystemB.cpp`.

Linking the Simulators

Depending on how you load the dynamic libraries, the linker might need access to the SIMULIA Co-Simulation Services dynamic libraries (see *Linking the SIMULIA Co-Simulation Services to Your Application*).

Typical link options for linking `DualMassSpring_SystemA` are as follows:

On Windows:

```
link /nologo /INCREMENTAL:NO /subsystem:console /machine:AMD64 /STACK:20000000
 /NODEFAULTLIB:LIBC.LIB /DEFAULTLIB:kernel32.lib /DEFAULTLIB:user32.lib
/DEFAULTLIB:advapi32.lib /FIXED:NO /LARGEADDRESSAWARE
/out:.\DualMassSpring_SystemA.exe .\DualMassSpring_SystemA.obj /LIBPATH:<location
 of CSS .dll> ABQSMACseModules.lib msvcrt.lib
```

On Linux:

```
g++ -fPIC -Wl,-Bdynamic -Wl,--add-needed -o DualMassSpring_SystemA
./DualMassSpring_SystemA.o -L<path to CSS .so> -lABQSMACseModules
```

Use similar options to link `DualMassSpring_SystemB`.

Running the Coupled Simulation Using Two C++ Simulators

You run three processes for the coupled simulation:

- The SIMULIA Co-Simulation Services Director;
- Simulator A; and
- Simulator B

We recommend that you run these processes in separate command shells, so that you can monitor each process during the development and debugging stages.

The SIMULIA Co-Simulation Services Director process and its command line arguments are described in *SIMULIA Co-Simulation Services Director*. You can start the SIMULIA Co-Simulation Services Director by either starting the `SMACseCtlMain.exe` function or by using the Abaqus execution procedure.

The Simulator command line arguments are as follows:

`csedirector` Director in charge of the coupled simulation.

Specify the `host:port` where `host` is the host name or IP address of the machine from which the SIMULIA Co-Simulation Services Director is launched, and `port` is the listener port used by the SIMULIA Co-Simulation Services Director for this particular analysis.

`alpha` Stiffness distribution factor.

This factor should be between 0.0 and 1.0.

`beta` Mass distribution factor.

This factor should be between 0.0 and 1.0.

The commands below start the coupled simulation using the two simulator programs with the explicit Gauss-Seidel coupling scheme with the stiffness distribution factor set to 0.5 and the mass distribution factor set to 0.1. The SIMULIA Co-Simulation Services Director is launched on host `earth` with listener port `91000`.

```
SMACseCtrlMain -config DualMassSpring_GaussSeidel_config.xml -listenerport 91000
SimulatorA -alpha 0.5 -beta 0.1 -csedirector earth:91000
SimulatorB -alpha 0.5 -beta 0.1 -csedirector earth:91000
```

The commands below start the dual mass-spring example using Simulator A and Abaqus with the implicit iterative coupling using Aitkens relaxation. The stiffness distribution factor is 0.5, and the mass distribution factor is 0.1. The SIMULIA Co-Simulation Services Director is launched using the Abaqus execution procedure on host `mars` with listener port `95000`. The job name `SimulatorB` refers to the component instance name in the Co-Simulation Configuration file and differs from the input file name.

```
abaqus -cse -config DualMassSpring_Aitkens_Abaqus_config.xml -listenerport 95000
SimulatorA -alpha 0.5 -beta 0.1 -csedirector mars:95000
abaqus -job SimulatorB -input DualMassSpring_SystemB -csedirector mars:95000
```

Results

The solution of an equivalent single system is given by the differential equation:
The closed form solution is

In this example, we have chosen a total stiffness equal to 35,000.0, a total mass equal to 2.5, a stiffness distribution factor of 0.5, and a mass distribution factor equal to 0.1. The closed-form solution for displacement, velocity, and acceleration is given above with a period of 0.0531.

The simulation time is set to produce 10 cycles. A coupling step size equal to 1/20th of the period is selected. From a stability point of view, the coupled solution becomes unstable for explicit coupling, as shown in *Figure 2* where an increase in amplitude is observed from cycle to cycle.

Figure 2: Displacement response using explicit coupling.

In some cases, selecting a smaller coupling step size might alleviate the stability issue. However, the solution cost increases and other numerical problems might arise with a smaller coupling step size. We recommend explicit coupling only for weakly coupled physics.

Figure 3 shows the displacement response for implicit iterative coupling using the constant relaxation, Aitken relaxation, Anderson quasi-Newton, and Broyden quasi-Newton accelerators.

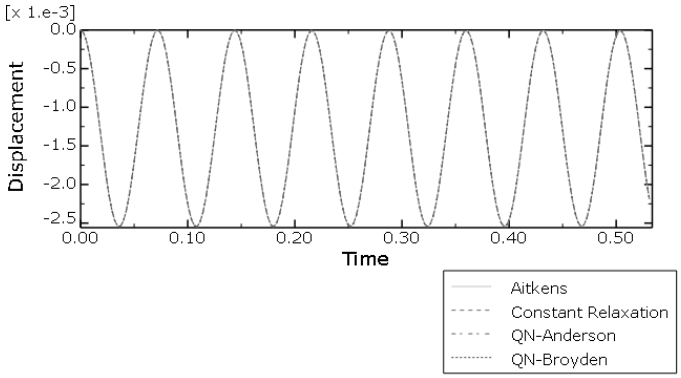


Figure 3: Implicit coupling using accelerators for a coupling size of P/20.

The response is accurate and stable for the large coupling step size P/20. The constant relaxation method requires the most iterations to reach the specified convergence tolerance. In fact, after 30 iterations it was ended. The Aitken relaxation method takes on average 3 iterations, and the Anderson and Broyden quasi-Newton methods take an average of 4 iterations to converge. In more complex industrial problems, we expect the quasi-Newton methods to have fewer iterations than the Aitken relaxation method.

Files

The C++ code for the simulators, the Abaqus input file, and the configuration files are available in the `examples/` folder in the SIMULIA Co-Simulation Services API kit (see *SIMULIA Co-Simulation Services API Kit*). If you have Abaqus installed, you can also use the **abaqus fetch** execution procedure.

DualMassSpring_SystemA.cpp	Code for spring-mass system A.
DualMassSpring_SystemB.cpp	Code for spring-mass system B.
DualMassSpring_SystemB.inp	Abaqus input file for System B.
DualMassSpring_GaussSeidel_config.xml	Configuration file using a Gauss-Seidel explicit coupling algorithm. Use this configuration file when coupling both simulators.
DualMassSpring_GaussSeidel_Abaqus_config.xml	Configuration file using Gauss-Seidel explicit coupling algorithm. Use this configuration file when coupling Simulator A with Abaqus.
DualMassSpring_ConstantRelaxation_config.xml	Configuration file using implicit iterative coupling with constant relaxation. Use this configuration file when coupling both simulators.
DualMassSpring_ConstantRelaxation_Abaqus_config.xml	Configuration file using implicit iterative coupling with constant relaxation. Use this configuration file when coupling Simulator A with Abaqus.
DualMassSpring_Aitkens_config.xml	Configuration file using implicit iterative coupling with Aitken relaxation. Use this configuration file when coupling both simulators.
DualMassSpring_Aitkens_Abaqus_config.xml	Configuration file using implicit iterative coupling with Aitken relaxation. Use this Configuration file when coupling Simulator A with Abaqus.

DualMassSpring_QNAnderson_config.xml Configuration file using implicit iterative coupling and the quasi-Newton Anderson method. Use this configuration file when coupling both simulators.

DualMassSpring_QNAnderson_Abaqus_config.xml Configuration file using implicit iterative coupling and the quasi-Newton Anderson method. Use this configuration file when coupling Simulator A with Abaqus.

DualMassSpring_QNBroyden_config.xml Configuration file using implicit iterative coupling and the quasi-Newton Broyden method. Use this configuration file when coupling both simulators.

DualMassSpring_QNBroyden_Abaqus_config.xml Configuration file using implicit iterative coupling and the quasi-Newton Broyden method. Use this configuration file when coupling Simulator A with Abaqus.

Aeroelastic Supersonic Flow over Long, Thin-Walled Cylinders

This is a more complex example illustrating how to embed the SIMULIA Co-Simulation Services API in a client that computes pressures associated with hypersonic flow on cylindrical panels.

This example studies the flutter behavior of infinitely long thin-walled cylinder subjected to aero-elastic supersonic flow. Such simulations are important to understand the complex nonlinear behavior that is present in high-speed flight vehicles such as aircraft, rockets, and launch vehicles, where one side of the structure is exposed to supersonic flow and the other side to still air.

Problem Description

The supersonic flow is computed using the Third-order Piston Theory. Third-order Piston Theory is a popular modeling technique used for supersonic and hypersonic aero-elastic analysis. We provide a standalone program, `thirdOrderPiston`, that computes the pressure as a function of the time and deformation computed by Abaqus/Explicit. We compare the results to analytical derived results from *L. Librescu, et. al. (2002)*.

The structure consists of an aluminum cylindrical panel with a curvature 0.1 m and a thickness of 0.01 m. The finite element model consists of a symmetric section with an arc length of 1 m. The material properties for the panels are as follows:

Young's modulus	7×10^{10} N/m^2
Poisson's ratio	0.3
Density	3000 kg/m^3

The third-order piston program uses the following fluid properties to model air:

Isentropic gas coefficient	1.4
Speed of sound	340.3 m/s
Density	1.225 kg/m^3

We assume an absolute pressure of 101328 N/m^2 . You can vary the speed by changing the variable, `M_INF`, in the third-order piston program.

The mesh used by the third-order piston program is generated and can be varied by modifying the variables `myModel->numElemsMesh` and `myModel->numLayerMesh` in the program. The upper and lower edges of the panel are pinned. Due to symmetry, the displacements in the axial directions are fixed.

Figure 1 shows the Abaqus mesh. There are 40 S4R elements in the circumferential direction.



Figure 1: Abaqus mesh of the panel segment.

You could have used one layer of elements in the cylindrical direction with a symmetry boundary condition; however, we use four layers of elements to illustrate mapping.

The upper and lower edges of the panel are pinned. Due to symmetry, the displacements in the axial directions are fixed. To get an initial nonzero pressure field, we introduce an initial imperfection in the first analysis step by applying a single load at the panel center. In the second step, we import the pressure computed by the third-order piston program.

Since Abaqus/Explicit cannot import pressure loads directly, we convert the pressure into a traction vector. The mapper then integrates the traction to concentrated force for Abaqus/Explicit.

Third-Order Piston Theory

The pressure field is computed according to third-order piston theory
where
and is the free stream pressure of the undisturbed flow (Pa), is the isentropic gas coefficient, and is the undisturbed flight Mach number.

The spatial derivative is approximated by the central difference method:
where is the transverse displacement, and is the element edge length in the -direction shown in *Figure 2*.

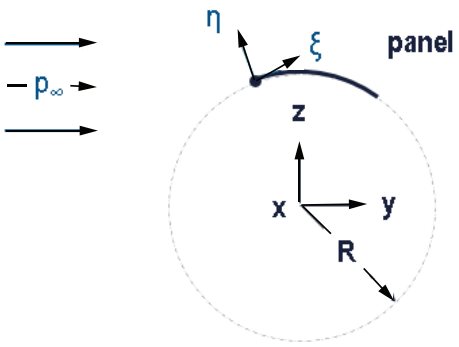


Figure 2: Modeling schematic.

Compiling the Third-Order Piston Simulator

If you have not already done so, extract the SIMULIA Co-Simulation Services API kit as described in *Extracting the SIMULIA Co-Simulation Services API Kit*. For information about compiling your client, see *Compiling your Application with SIMULIA Co-Simulation Services*. Typical compile options for the client are as follows:

On Windows:

```
cl /c /W0 /MD /TP /I<location of CSS header files> thirdOrderPiston.cpp
```

On Linux:

```
g++ -c -fPIC -w -Wno-deprecated -I <location of CSS header files>
thirdOrderPiston.cpp
```

Linking the Third-Order Piston Simulator

Depending on how you load the dynamic libraries, the linker might need access to the SIMULIA Co-Simulation Services dynamic libraries. For information about linking your client, see *Linking the SIMULIA Co-Simulation Services to Your Application*. Typical link options for the client are as follows:

On Windows:

```
link /nologo /INCREMENTAL:NO /subsystem:console /machine:AMD64 /STACK:20000000
 /NODEFAULTLIB:LIBC.LIB /DEFAULTLIB:kernel32.lib /DEFAULTLIB:user32.lib
/DEFAULTLIB:advapi32.lib /FIXED:NO /LARGEADDRESSAWARE /out:.\thirdOrderPiston.exe
 .\thirdOrderPiston.obj /LIBPATH:<location of CSS .dll> ABQSMACseModules.lib
msvcrt.lib
```

On Linux:

```
g++ -fPIC -Wl,-Bdynamic -Wl,--add-needed -o thirdOrderPiston ./thirdOrderPiston.o
 -L<path to CSS .so> -lABQSMACseModules
```

Running the Coupled Simulation

You run three processes for the coupled simulation:

- the SIMULIA Co-Simulation Services Director;
- the third-order piston simulator that computes the pressure; and
- Abaqus/Explicit analysis.

We recommend that you run these in three separate command shells so that you can monitor each process during the development and debugging stages.

The SIMULIA Co-Simulation Services Director process and its command line arguments are described in *SIMULIA Co-Simulation Services Director*. You can start the SIMULIA Co-Simulation Services Director process by starting the `SMACseCtlMain.exe` function or by using the **abaqus cse** execution procedure.

The third-order piston simulator's command line arguments are as follows:

`csedirector` Director in charge of the coupled simulation.

Specify the `host:port` where `host` is the name of the machine where the SIMULIA Co-Simulation Services Director port is launched, and `port` is the listener port used by the SIMULIA Co-Simulation Services Director.

`speed` Mach number for the undisturbed speed.

Use the commands below to start the coupled simulation. The SIMULIA Co-Simulation Services Director is launched on host `mars` with listener port `92000`.

```
abaqus -cse -config thirdOrderPiston_xpl_config.xml -listenerport 92000
thirdOrderPiston -speed 4.16 -csedirector mars:92000
abaqus -job thirdOrderPiston_xpl -double_both -csedirector mars:92000
```

Results

The co-simulation computes the Mach flutter of $M_{Fnumerical}$ 4.17. The numerical solution matches the analytical expectations according to *L. Librescu, et. al. (2002)* within less than one percent.

To determine the numerical Mach flutter, $M_{Fnumerical}$, we plot the displacements in the global z-direction of the center node of the panel. *Figure 3* shows a stable behavior of the panel for a Mach number of M = 4.16.

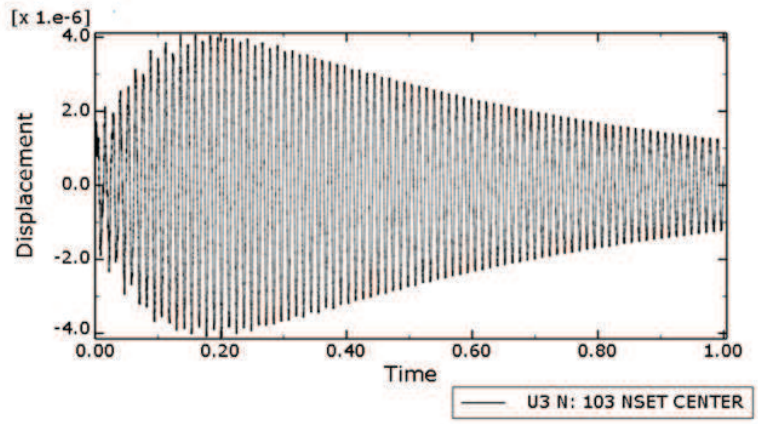


Figure 3: Center node displacement for M = 4.16.

On the contrary, the displacements for a Mach number of M = 4.18 (shown in *Figure 4*) do not damp out, which indicates that the numerical computed flutter speed is $M_{Fnumerical}$ 4.17.

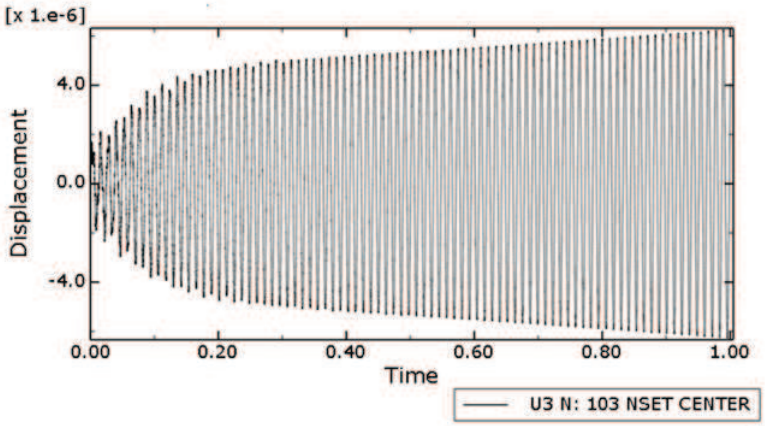


Figure 4: Center node displacement for M = 4.18.

Files

The C++ code for the Third-order Piston Theory, the Abaqus input file, and the configuration files are available in the `examples/` folder in the SIMULIA Co-Simulation Services API kit. If you have Abaqus installed, you can also use the **abaqus fetch** execution procedure .

thirdOrderPiston.cpp	C++ code for third-order piston program.
thirdOrderPiston_xpl.inp	Abaqus/Explicit input file for third-order piston model.
thirdOrderPiston_xpl_config.xml	Configuration file for third-order piston example.

References

1. Librescu, L., P. Marzocca, and W. A. Silva, "Supersonic/Hypersonic Flutter and Postflutter of Geometrically Imperfect Circular Cylindrical Panels," *Journal of Spacecraft and Rockets*, vol. 39, no. 5, 2002.

ASCII File Results Provider

Problem Description

This example is adapted from the Simpack tutorial on flexible bodies using co-simulation (see `SIMPACK_Tutorial_flex_abaqcosim.zip` on the Simpack media). You do not need Simpack because we provide the generated results file with this example.

The Simpack model consists of an automotive front axle with a flexible control arm as shown in *Figure 1*. During the first 0.25 sec, the front axle is accelerated to a velocity of 10 m/sec, and then the speed is kept constant.

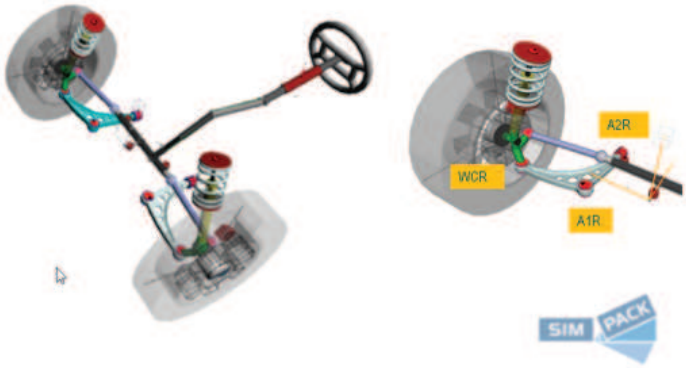


Figure 1: Simpack model and detail showing location where fields are transferred.

We perform a detailed finite element analysis of the control arm, where displacements and rotations of the left control arm computed by Simpack are imported and applied at three discrete locations shown in *Figure 2*. The imported displacement and rotation are applied using distributing constraints in Abaqus.

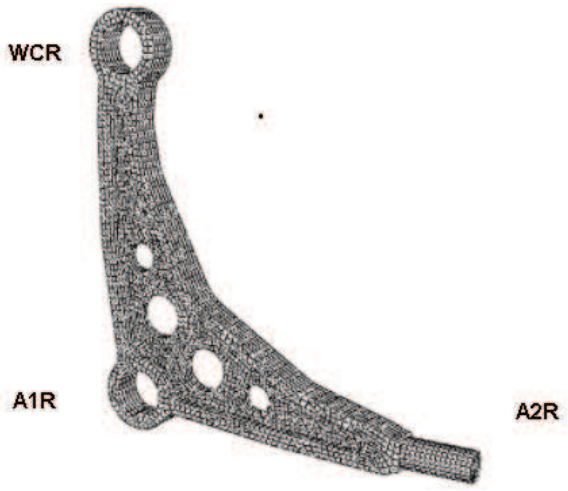


Figure 2: Abaqus model of control arm.

Compiling the Results Provider

If you have not already done so, extract the SIMULIA Co-Simulation Services API kit as described in *Extracting the SIMULIA Co-Simulation Services API Kit*. The compiler needs access to the SIMULIA Co-Simulation Services header files. Typical compile options for `MBDControlArm_Reader.cpp` are as follows (see *Compiling your Application with SIMULIA Co-Simulation Services*):

On Windows:

```
cl /c /W0 /MD /TP /I<location of CSS header files> MBDControlArm_Reader.cpp
```

On Linux:

```
g++ -c -fPIC -w -Wno-deprecated -I <location of CSS header files>
MBDControlArm_Reader.cpp
```

Linking the Results Provider

Depending on how you load the dynamic libraries, the linker might need access to the SIMULIA Co-Simulation Services dynamic libraries. Typical link options for `MBDControlArm_Reader` are as follows (see *Linking the SIMULIA Co-Simulation Services to Your Application*):

On Windows:

```
link /nologo /INCREMENTAL:NO /subsystem:console /machine:AMD64 /STACK:20000000
 /NODEFAULTLIB:LIBC.LIB /DEFAULTLIB:kernel32.lib /DEFAULTLIB:user32.lib
/DEFAULTLIB:advapi32.lib /FIXED:NO /LARGEADDRESSAWARE
/out:.\MBDControlArm_Reader.exe .\MBDControlArm_Reader.obj /LIBPATH:<location
of CSS .dll> ABQSMACseModules.lib msvcrt.lib
```

On Linux:

```
g++ -fPIC -Wl,-Bdynamic -Wl,--add-needed -o MBDControlArm_Reader
./MBDControlArm_Reader.o -L<path to CSS .so> -lABQSMACseModules
```

Executing the Sequential Simulation

The results provider and the SIMULIA Co-Simulation Services Director are started from the Abaqus execution procedure; therefore, you only need to run the Abaqus job. The configuration and results specification file are generated by the Abaqus preprocessor.

```
abaqus -job MBDControlArm_Abaqus
```

When starting the processes manually, we recommend that you run all processes in separate command shells during the development and debugging stages. First, obtain the configuration and specification files by running a **datacheck** analysis:

```
abaqus -job MBDControlArm_Abaqus -datacheck
```

Then, start the three processes in separate command shells:

- Start the Abaqus job by providing the `-csedirector` command line argument. You should wait until the Abaqus preprocessor completes because it might overwrite the existing configuration and specification files.

  ```
  abaqus -job MBDControlArm_Abaqus -csedirector venus:93000 -int
  ```

- Start the `MBDControlArm Results Provider`.

  ```
  MBDControlArm_Reader -job MBDControlArm_Simpack -file MBDControlArm_Simpack.dat
   -spec MBDControlArm_Simpack_spec.xml -csedirector venus:93000 -timeout 300
  ```

- Start the SIMULIA Co-Simulation Services Director process

```
abaqus -cse -job MBDControlArm_Abaqus_css -configure
MBDControlArm_Abaqus_config.xml -listenerport 93000 -timeout 300
```

Results

This example illustrates how to write a results provider to read an ASCII data file. In this case the data file is a Simpack report file. *Figure 3* and *Figure 4* show the deformation and stresses, respectively, of the control arm under the imposed loading at time=1.0.

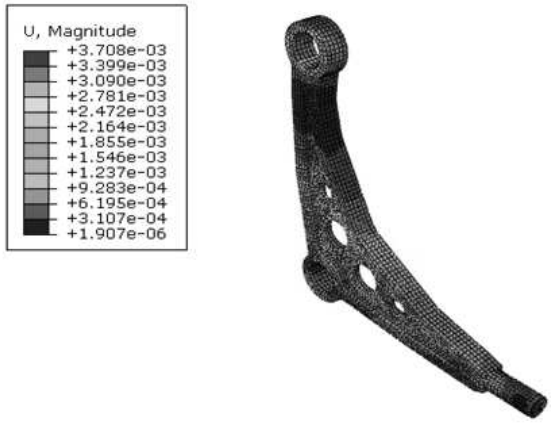


Figure 3: Abaqus deformation contour of control arm.

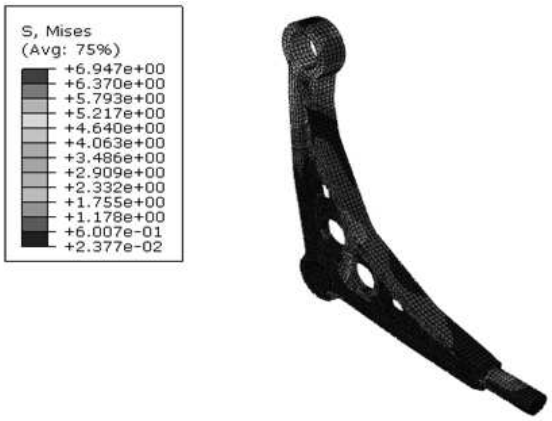


Figure 4: Abaqus stress contour of control arm.

Files

The C++ code for the results provider, the Abaqus input file, and the Simpack report file are available in the `examples/` folder in the SIMULIA Co-Simulation Services API kit. If you have Abaqus installed, you can use the **abaqus fetch** execution procedure to obtain the files.

MBDControlArm_Reader.cpp	C++ code for the Results Provider.
MBDControlArm_Abaqus.inp	Abaqus input file for the control arm.
MBDControlArm_Simpack.dat	Results report file generated from the Simpack analysis containing deformation and rotation at three discrete locations.

Appendix: Predefined Field Types

This section lists the predefined field types when using `SMACseDefineField`.

The predefined fields are listed by physics. If a predefined field is not available, you can use the `SMACseCreateGeneralFieldDefinition` function to define your own field types.

Mechanical Fields

Table 1: Mechanical field definitions.

Field Type	Physics Property	Unit Dimensionality
acceleration	intensive	LT-2
body_force	intensive	ML-2T-2
displacement	intensive	L
displacement_predictor	intensive	L
coordinates	intensive	L
force	extensive	MLT-2
moment	extensive	ML2T-2
pressure	intensive	ML-1T-2
velocity	intensive	LT-1
velocity_predictor	intensive	LT-1
rotation	intensive	R
rotation_predictor	intensive	R
rotational_acceleration	intensive	RT-2
rotational_velocity	intensive	RT-1
traction_vector	intensive	ML-1T-2

Thermal Fields

Table 2: Thermal field definitions.

Field Type	Physics Property	Unit Dimensionality
body_heat_flux_density	intensive	ML-1T-3
film_coefficient	intensive	MT-3K-1
heat_capacitance	extensive	ML2T-2K-1
heat_flux	extensive	ML2T-3

Field Type	Physics Property	Unit Dimensionality
sink_temperature	intensive	K
surface_heat_flux	intensive	MT-3
temperature	intensive	K

Electrical Fields

Table 3: Electrical field definitions.

Field Type	Physics Property	Unit Dimensionality
electric_charge	extensive	ET
electric_current	intensive	E
electric_current_density	intensive	EL-2
electric_potential	extensive	ML2T-3E-1

Pore Pressure Fields

Table 4: Pore pressure field definitions.

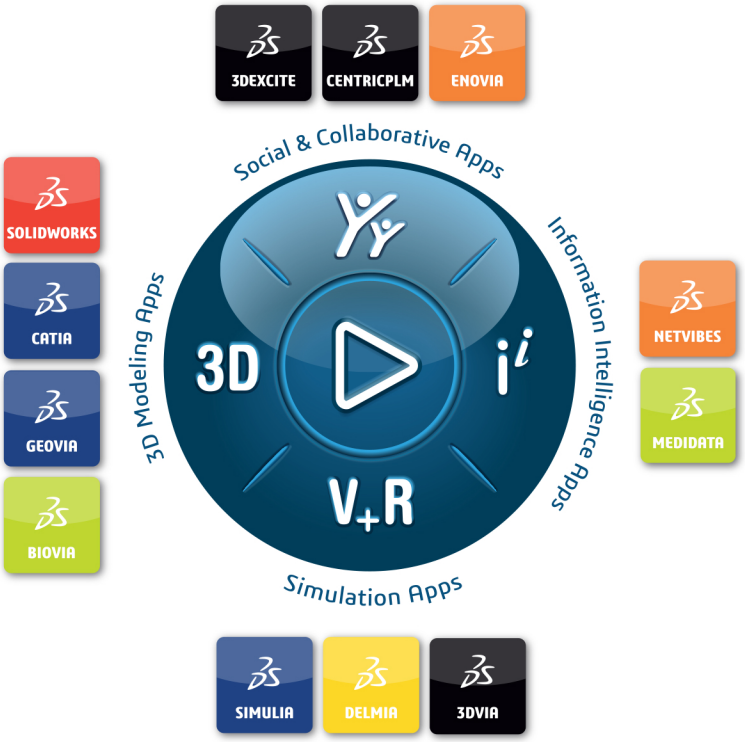
Field Type	Physics Property	Unit Dimensionality
concentrated_flow	extensive	L3T-1
sink_pore_pressure	intensive	ML-1T-2
pore_pressure	intensive	ML-1T-2
seepage	intensive	LT-1
seepage_coeff	intensive	M-1L2T
void_ratio	intensive	

General Fields

Table 5: General field definitions.

Field Type	Physics Property	Unit Dimensionality
field_variable_01 through field_variable_19	intensive	
mass	extensive	M

Field Type	Physics Property	Unit Dimensionality
rotary_inertia	extensive	ML2
temperature	intensive	K



Our **3D**EXPERIENCE® platform powers our brand applications, serving 12 industries, and provides a rich portfolio of industry solution experiences.

Dassault Systèmes is a catalyst for human progress. We provide business and people with collaborative virtual environments to imagine sustainable innovations. By creating virtual twin experiences of the real world with our **3D**EXPERIENCE platform and applications, our customers can redefine the creation, production and life-cycle-management processes of their offer and thus have a meaningful impact to make the world more sustainable. The beauty of the Experience Economy is that it is a human-centered economy for the benefit of all – consumers, patients and citizens.

Dassault Systèmes brings value to more than 300,000 customers of all sizes, in all industries, in more than 150 countries. For more information, visit **www.3ds.com**.

Europe/Middle East/Africa
Dassault Systèmes
10, rue Marcel Dassault
CS 40501
78946 Vélizy-Villacoublay Cedex
France

Asia-Pacific
Dassault Systèmes
17F, Foxconn Building,
No. 1366, Lujiazui Ring Road
Pilot Free Trade Zone, Shanghai 200120
China

Americas
Dassault Systèmes
175 Wyman Street
Waltham, Massachusetts
02451-1223
USA

DASSAULT SYSTEMES

©2026 Dassault Systèmes. All rights reserved. **3DEXPERIENCE**, the 3DS logo, the Compass icon, IFWE, 3DEXCITE, 3DVIA, BIOVIA, CATIA, CENTRIC PLM, DELMIA, ENOVIA, GEOVIA, MEDIDATA, NETVIBES, OUTSCALE, SIMULIA and SOLIDWORKS are commercial trademarks or registered trademarks of Dassault Systèmes, a European company (Societas Europaea) incorporated under French law, and registered with the Versailles trade and companies registry under number 322 306 440, or its subsidiaries in the United States and/or other countries.