SANDIA REPORT

SAND2017-3777
Unlimited Release
Printed April 13, 2017

SIERRA Code Coupling Module: Arpeggio User Manual – Version 4.44

Samuel R. Subia, James R. Overfelt, David G. Baur

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2017-3777
Unlimited Release
Printed April 13, 2017

SIERRA Code Coupling Module: Arpeggio User Manual – Version 4.44

Samuel R. Subia, James R. Overfelt, David G. Baur
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

The SNL Sierra Mechanics code suite is designed to enable simulation of complex multiphysics scenarios. The code suite is composed of several specialized applications which can operate either in standalone mode or coupled with each other. Arpeggio is a supported utility that enables loose coupling of the various Sierra Mechanics applications by providing access to Framework services that facilitate the coupling. More importantly Arpeggio orchestrates the execution of applications that participate in the coupling. This document describes the various components of Arpeggio and their operability. The intent of the document is to provide a fast path for analysts interested in coupled applications via simple examples of its usage.

Contents

Contents 5

List of Figures 7

1 Introduction 9

1.1 Coupled Physics Approaches . . . 9

1.2 Sierra Mechanics Coupling . . . 10

1.3 Communication of Data (Transfer Services) . . . 11

1.4 Solution Control . . . 11

1.5 Coupling Strategies . . . 12

1.6 Coupling with Arpeggio . . . 14

1.7 Outline of the Manual . . . 15

2 Getting Started 17

2.1 Setting The Environment-Users External to Sandia Labs . . . 17

2.2 Setting The Environment-Users at Sandia Labs . . . 17

2.3 Running Arpeggio . . . 17

2.4 Arpeggio Environment Overview . . . 18

2.5 Overview of the Input File Structure . . . 19

2.6 Fields . . . 23

2.7 User Fields . . . 24

3 Model Definition 25

3.1 Model Overview . . . 25

3.2 Finite Element Model . . . 26

3.3 Parameters For Block . . . 30

3.4 Global Constants . . . 32

3.5 Definition For Function . . . 34

3.6 Values . . . 41

3.7 Restart Overview . . . 42

4 Solution Control Reference 43

4.1 Overview . . . 43
4.2 Solution Control Description . . . 47
4.3 System . . . 48
4.4 Transient . . . 52
4.5 Nonlinear . . . 55
4.6 Subcycle . . . 59
4.7 Sequential . . . 62
4.8 Initialize . . . 65
4.9 Parameters For . . . 66

5 Transfer Reference 75

5.1 Overview . . . 75
5.2 Transfer . . . 79

6 Input Output Region Reference 87

6.1 Input_Output Region Overview . . . 87
6.2 Input_Output Region . . . 88

7 Examples 93

7.1 One-Way Coupling From File . . . 93
7.2 One-Way Coupling Using Transfer From Different Mesh . . . 94
7.3 One-Way Coupling Using Transfer . . . 95
7.4 Two-Way Coupling With Transfer . . . 95
7.5 estack Regression Test . . . 96
7.6 tv Regression Test . . . 96

References 97

Index 99

List of Figures

1.1 Loose Coupling Schematic (Z Scheme). 10

1.2 Sierra Mechanics Data Types. 11

1.3 One-way Loose Coupling At Same Time Step. 13

1.4 Deferred One-way Loose Coupling At Same Time Step. 13

1.5 One-way Loose Coupling with Subcycling Schematic. 14

1.6 Two-way Loose Coupling Schematic. 14

1.7 Thermal-Mechanical With Thermal One-Way Element Death. 15

1.8 Thermal-Mechanical With Two-Way Element Death. 15

2.1 Schematic UML class diagram for the Expression subsystem. 18

5.1 Valid Transfer Operations 76

5.2 Invalid Transfer Operation 77

Chapter 1

Introduction

The SNL Sierra Mechanics code suite is designed to enable numerical simulations of complex multi-physics scenarios. The code suite is composed of specialized applications which can operate either in standalone mode or in a coupled mode with other Sierra Mechanics applications. Arpeggio is a supported utility that enables loose coupling of the various Sierra Mechanics applications by providing access to Framework services that facilitate application coupling. Utilizing these services Arpeggio is able orchestrate the execution of applications that participate in code coupling. This document describes the Framework services used by Arpeggio for coupling and the inter-operability of these services for coupling of Sierra SM and Sierra TF applications. Through the use of simple examples, the document also provides a resource for analysts interested performing in coupled-physics simulations.

1.1 Coupled Physics Approaches

When modelling tightly-coupled physics, the numerical representation of all PDEs within a region of interest are often combined a single system matrix and solved using a nonlinear solution strategy specific to the application. This approach to solving coupled-physics problems is available for a limited set of physics in the Sierra Mechanics TF module. Relaxing the notion of tight-coupling one could alternatively obtain solutions for each of the physics independently and patch the individual solutions together in some prescribed manner, this is the essence of loosely-coupled physics simulations.

The numerical analysis community has long recognized the need to include results from various physics in a single simulation. However, the fact that most application codes are often developed around single physics often limits the extent to which coupled-physics simulations can be achieved. Early approaches to coupled-physics simulations often simplified the coupling by level of importance by assigning primary physics and secondary physics roles. Here the primary physics depended upon secondary physics and the dependence of secondary physics upon primary physics was deemed less important. Under this assumption coupled physics simulations can be realized by first performing independent simulations of the secondary physics followed by a simulation of the primary physics utilizing results of the initial simulation. Figure 1.1 illustrates the coupling approach for a quasi-static solution step from a state t_n to state t_{n+1}. Broadly speaking, loose-coupling strategies are classified as Z-methods, since a Z describes the basic pattern of data communication between the physics applications. The one-way view of loosely-coupled physics lends itself to file-based approaches where single state results are obtained on a common spatially meshed discretization. Here the problem solutions are generally obtained at cell vertices (nodes) or cell centers (elements). Quite often each physics simulation lends itself to a particular spatial discretization and this gave rise to the introduction of an intermediate mapping step whereby the secondary physics results were mapped onto the primary physics discretization as in the MAPVAR utility [1]. For transient coupled-physics simulations best results are obtained when sharing a common time discretization but in many cases this is impractical and the coupling is based upon closest-time matched solutions or interpolations of solutions in time.

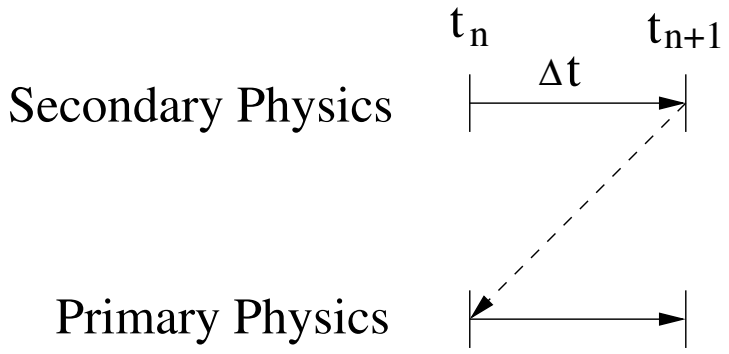

Figure 1.1. Loose Coupling Schematic (Z Scheme).

1.2 Sierra Mechanics Coupling

Sierra Mechanics physics applications deal with solving PDEs on a physical geometric domain, a Region. In defining a coupled physics problem, users configure one or more Regions corresponding to some particular physics. Each Region considers one or more PDEs to be solved on either the entire input mesh or on a portion of the mesh. When the Region physics are coupled one can elect whether to solve the physics in a tightly-coupled manner in a single application or by loosely coupling individual Region results. Here we note that for loose coupling the physical geometry and spatial discretization must overlap but need not be identical in each of the participating Regions.

In the context of Sierra Mechanics, loosely-coupled physics nonlinear solutions are obtained on each of the Regions and then combined to form an overall coupled solution. Not surprisingly there are numerous ways one can approach loose-coupling since different strategies are appropriate to different problem sets. That is, the solution for one Region may depend strongly upon the solution in another Region but not vice-versa (one-way coupled), or the the solution for each Region may depend upon the solution the other Region (two-way coupling). The goal of Sierra Mechanics is to provide services which enable one to easily perform variants of a multi-physics coupling.

Some considerations which are relevant to the loose-coupling solution strategies include

- Communication of data from one Region to another Region (**Transfer**),
- Initialization of the individual Regions,
- Solution for the individual Regions (**Advance**),
- Time stepping or pseudo-time stepping for the individual Regions,
- Time synchronization of participating Regions,
- Conditional convergence,
- Drive mesh adaptivity,
- Sequencing for all of the above.

Within Sierra Mechanics communication of data between application Regions is handled by the Framework Transfer service and all aspects of solution behavior are managed by another Framework service, Solution Control. Mesh Adaptivity is managed through the Percept library.

1.3 Communication of Data (Transfer Services)

In Sierra Mechanics application data is generally associated with nodes, elements, faces or edges of a meshed discretization as shown in Figure 1.2. A loose-coupling between applications implies the dependence of one application on data supplied from some external source. Since the physical location of data on the external source may or may not map geometrically onto the the other application solution, provisions must be made to perform this data mapping in a flexible manner. It is important to note that these mappings can be accomplished both for the case of different mesh and different element types. Within Sierra Mechanics this responsibility is handled by Framework Transfer services. Here it is important to note that Framework Transfer services enable the external source data to include element, face and edge data as well as nodal data.

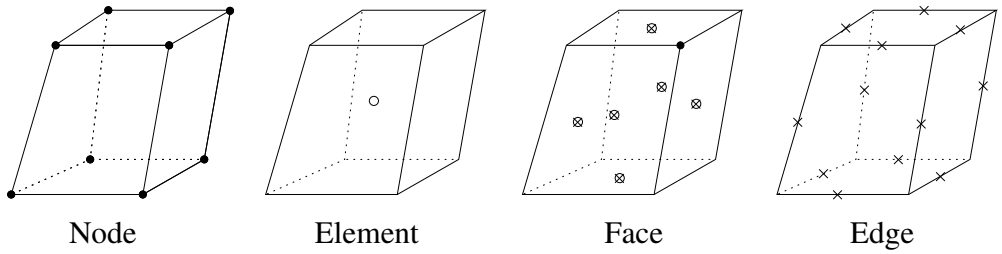

Figure 1.2. Sierra Mechanics Data Types.

1.4 Solution Control

The Solution Control subsystem controls the execution of coupled multi-physics applications. Solution control provides two basic operations for controlling the solution of a multi-physics system by defining the order for object execution and by setting parametric values on the controlled objects at the proper time. For *transient* problems this approach enables the applications to easily transition through the designated time periods. The same system can also service steady-state simulations by treating them in a *sequential* manner. The solution controllers are able to initialize Region data, set parameter values, advance Regions, execute transfers, call events and send notifications based on the input file specifications.

1.4.1 Region Initialization

When beginning execution, all applications require some baseline initialization operations at the Region level. When performing some loose-coupling simulations the dependence of data may may require that initialization of data be performed in some specific manner. Here the manner in which initialization occurs is determined by how the application solution variables are defined and the application code implementation of initialization. As an example for a thermal-mechanical coupling one might initialize the reference temperature state in the solid before any temperature change in the solid were allowed to occur. Solution Control provides a means for performing various types of non-standard data initialization.

1.4.2 Solution

Each set of coupled physics represents a *System* of equations which must be solved. While participating in loose-coupling an application physics will attempt to advance its solution to a later state. In the parlance

of Solution Control this step is known as an **advance** event. Here the details of code operations associated with advancing the solution are controlled entirely by the physics application. Additionally, because the advance can occur conditionally it provides flexibility in how the coupling is performed.

1.4.3 Time Stepping

Within Sierra Mechanics each application is allowed to define its own notion of solution time. The Solution Control time step controller probes the individual application solution time and uses that information to determine how time should be advanced for the coupled physics. For couplings of transient simulations with quasi-static applications, the time step controller manages a unified notion of pseudo-time and physical time seamlessly, even when the time step selection is adaptive.

1.4.4 Conditional Events

In loosely-coupled simulations the need often arises to perform some high level operations conditionally. Here Solution Control is able to probe the application for current states or variables to determine whether whether some coupling action should occur. These conditionals can be applied to both the data transfer or advance Solution Control events. Examples of conditional events are included in Chapter 4.

1.5 Coupling Strategies

Using the Solution Control one can easily define loose couplings between two or more Regions. For example, some or all of a solution from one Region may be transferred to another Region where it is treated as a constant, external field. The aggregate nonlinear problem including the contributions from all of the Regions may be iterated to convergence. The details of which physics are solved in each Region and the nonlinear solution strategy used within and between Regions is completely specified through the input file. Furthermore, a Sierra Mechanics user may pick a simple, minimal algorithm without needing to fit it into an overly-generalized worst-case scenario that represents the union of all possible algorithms.

Dynamically-specified loose coupling has many potential advantages that users may leverage to obtain solutions. First, the resulting linear system is considerably smaller than a fully-coupled system and contains far fewer off-diagonal contributions which can significantly improve the performance of linear solvers. Furthermore the resulting linear system may have a more desirable mathematical properties, such as being symmetric positive-definite, this permits the use of tailored iterative solutions techniques. Other extensions to loose coupling include subcycling of transient simulations where each Region may advance in time with its own time step size and in-core coupling to other applications based upon the Sierra framework.

The simplest loose-coupling strategy is a one-way coupling between two applications, App I and App II, is shown schematically in Figure 1.3. Here it is assumed that information (data) from App I is needed by App II but App I is independent (decoupled) from App II. Furthermore it is assumed that the applications can proceed at the same time step. In this case the solution for each application can proceed in locked step.

A variant of the simplest loose-coupling would be the case where the dependence of App II solutions on App I data is such that update of the App I data can be deferred for several steps. This type coupling behavior can be enforced using a conditional advance event in Solution Control. As an example, a data transfer event defined for every two time steps of each application is shown schematically in Figure 1.4.

In some couplings the temporal response of one application physics, App I, is much faster than that of

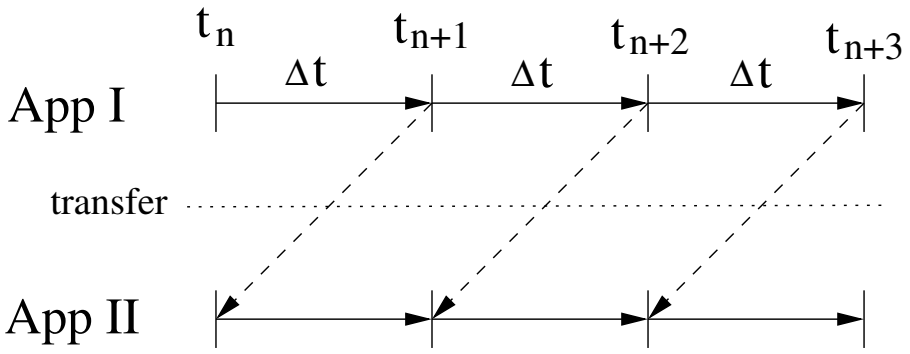

Figure 1.3. One-way Loose Coupling At Same Time Step.

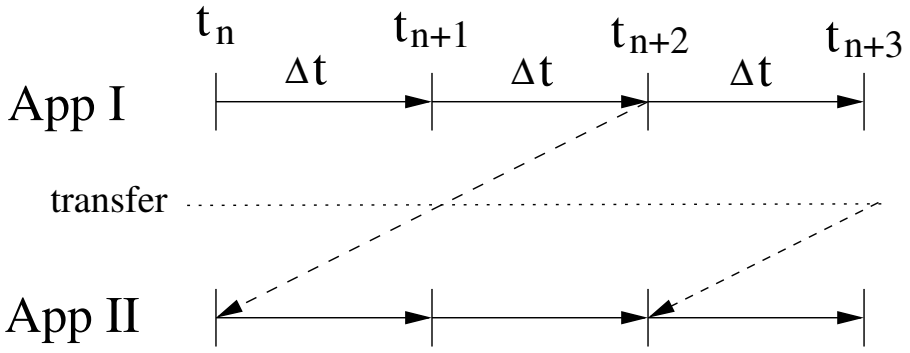

Figure 1.4. Deferred One-way Loose Coupling At Same Time Step.

another physics, App II. Here one may wish to advance the App I physics many time steps before requiring an update of its contribution to the App II information, Figure 1.5. Here Solution Control provides a facility denoted as *subcycling* to invoke this behavior.

When the coupling between App I and App II is circular in nature, (i.e. App I solutions depend upon App II and vice versa) the coupling can be achieved by adding an additional Transfer step to the one-way coupling approach. However, if the coupling dependency is fairly strong it may be prudent to ascertain a converged solution between the physics models before advancing to the solution step. Here the conditional event aspect of Solution Control can be employed to iterate App I and App II until a solution of the desired quality is obtained. The strategy is depicted in Figure 1.6 and is supported as the Nonlinear option within Solution Control.

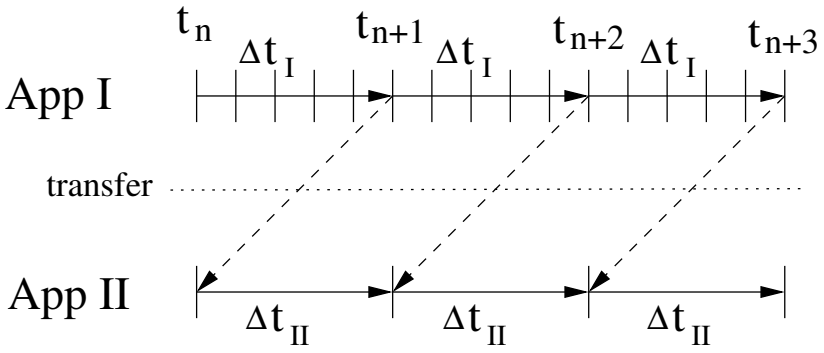

Figure 1.5. One-way Loose Coupling with Subcycling Schematic.

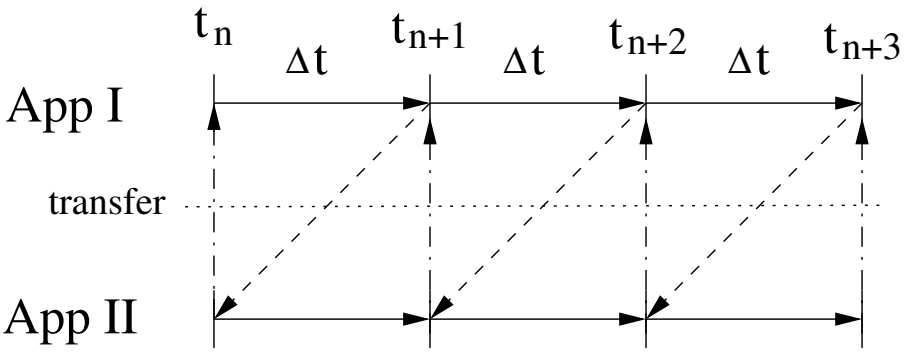

Figure 1.6. Two-way Loose Coupling Schematic.

1.6 Coupling with Arpeggio

While the previous sections have described the component utilities needed to enable coupled physics simulations but little has been said of existing tools composed of these utilities. Previous efforts in the development of Sierra Mechanics focused upon thermal-mechanical coupling of the Calore and Adagio applications with the Calagio utility to analyze problems of thermal stress. Here Sierra Mechanics utilities were used to solve the temperature state, then initializing the reference temperature state in Adagio followed by subsequent solves and transfer of the temperature state to Adagio to obtain a thermal stress state in the deformed configuration. Within the Calagio utility extra efforts were made to obscure the use of Framework utilities lying outside the realm of Calore and Adagio. Early one-way coupling efforts were later followed by two-way couplings where the deformed configuration was communicated to Calore and the heat transfer problem could be solved in the updated geometry. Although couplings with Calagio were largely successful it was recognized that incremental improvements in coupling capability came with a high price in terms of code development effort both to alter the predefined coupling strategies and to hide the underlying implementation from the analyst within the application code. Moreover, the predefined coupling strategy approach prevented the analyst from fully exploiting the resources available within Sierra Mechanics and the applications themselves. These shortcomings provided a motivation for creation of the Arpeggio utility in which the analyst fully specifies details of the coupling strategy.

1.6.1 Coupling Including Element Death

Coupling strategies in predecessors of Arpeggio precluded the possibility of simulations that required syncronization of the meshed discretization such as element death. Here the transfer capability in conjunction with a consistent notion of an application code indicator of element death (**Death_Status**) enables coupled simulations that include element death. Prevalent uses of this capability are one-way coupled thermal-mechanical simulations with thermally-driven element death 1.7 and two-way coupled thermal-mechanical simulations with element death driven by either application 1.8. For both types of coupling the mechanical code behavior is essentially the same as for a two-way coupling. On the other hand, in the case of two-way coupling the one-way coupling thermal invocation of a death criteria test is altered by the addition of an Aria Region level command line: **Transfer Element Death**.

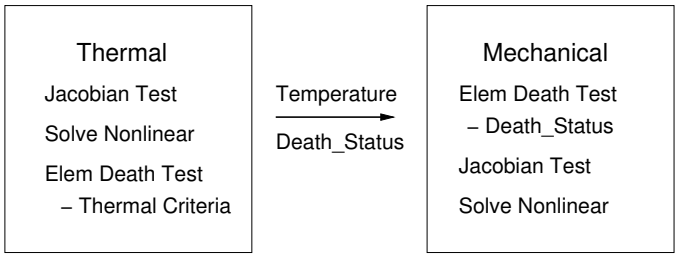

Figure 1.7. Thermal-Mechanical With Thermal One-Way Element Death.

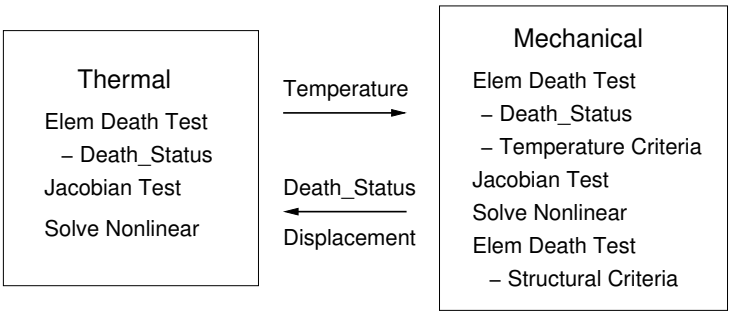

Figure 1.8. Thermal-Mechanical With Two-Way Element Death.

1.7 Outline of the Manual

Chapter 2 discusses the overall Sierra Mechanics environment for running Arpeggio, including the layout for the Arpeggio input file. Sierra Mechanics users familiar with the overall environment need only browse the input file structure and move directly to the sections describing Framework Transfer 5 and Solution Control 3.7. Experienced Sierra Mechanics users may opt to move directly to examples of coupling in Chapter 7

Chapter 2

Getting Started

2.1 Setting The Environment-Users External to Sandia Labs

To access Sierra/Arpeggio one will likely need to setup the user environment. This setup will differ upon location and the local system administrator can provide information on setting up your local environment.

2.2 Setting The Environment-Users at Sandia Labs

The environment for using Arpeggio is the same as for individual Sierra applications and can be configured by module files. The modules ensure that the look and feel of running Sierra applications is the same across a multitude of compute platforms. To obtain the proper environment for code execution one simply runs:

```
% module load sierra
```

2.3 Running Arpeggio

This section includes some very simple examples of how to run Arpeggio. For more information on running on some of Sandia's clusters, etc. see [2].

In its simplest form, Arpeggio can be run like this:

```
% sierra arpeggio -i myrun.i
```

In this example, `myrun.i` is the Arpeggio input file. The output – nonlinear iterations, time step information, etc. – will be written to a file called `myrun.log`. So, you can monitor the progress of the simulation by watching the log file. Alternatively, you can have all of the output sent to the display by using the `-l` *`logfile`* command line option. If you set the log file to be `-` (a single "minus" character) all of the output will be sent to the standard output (usually your display):

```
% sierra arpeggio -i myrun.i -l -
```

If you would like to use `aprepro` in your input file, add the `-a` command line option to have your input file automatically processed:

```
% sierra arpeggio -i myrun.i -l - -a
```

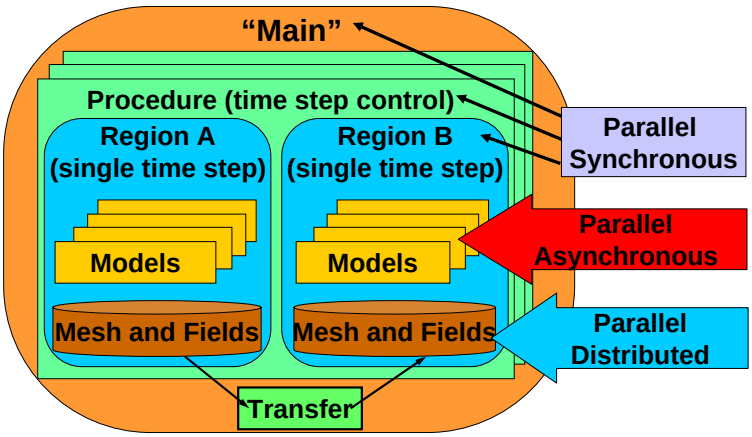

Figure 2.1. Schematic UML class diagram for the Expression subsystem.

Oftentimes we want to run Arpeggio remotely or locally in a batch mode, save any standard output and perhaps even logout from a session. Unfortunately, termination of the session through either voluntary (interactive) or involuntary (timeout) logout may in effect terminate the Arpeggio job. In this case one can prevent the job from terminating by using the Unix nohup command in conjunction with the standard execution command line.

```
% nohup sierra arpeggio -i myrun.i -l YourLogFile -a
```

If one wishes to run the job in a background mode the nohup command should be terminated with & at the end of the command line.

2.4 Arpeggio Environment Overview

The Sierra Mechanics code suite is composed of several specialized applications which can operate either in standalone mode or coupled with each other. The various application models and algorithms are integrated into the Sierra framework through the architecture illustrated in Figure 2.1. A Sierra-based application has four layers of code: Domain, Procedure, Region, and Model/Algorithm.

The outermost layer of an application is the Domain, or "main" program of the application. This domain layer is implemented by the Sierra Framework to manage the startup/shutdown of an application, and to orchestrate the execution of an application-proved set of procedures.

Code at the Procedure level is responsible for evolving one or more loosely coupled set of physics through a sequence of steps. This sequence may be a set of time steps, nonlinear solver iterations, or some combinations

of these or other types of steps.

An application mauy define multiple procedures to implement hand-off coupling between physics within the same main program. In hand-off coupling the first (or preceding) procedure completes execution, mesh and field data is transferred to a succeeding procedure, and the succeeding procedure continues the simulation with a different set of physics. For example, the first thermal procedure could calculate a temperature distribution inside a differentially heated fluid, and the second procedure could simulate natural convection of the fluid due to the density gradients set up by the resulting temperature field.

Code at the Region level is responsible for evolving a tightly coupled set of physics throug a single step. Loose coupling of Regions is supported by the advanced transfer services provided by the Sierra framework.

Each Region owns (1) a set of models or algorithms that implement its tightly coupled set of physics and solvers and (2) an in-memory parallel distributed mesh and field database. This mesh and field data is fully distributed among parallel processors via domain decomposition.

2.5 Overview of the Input File Structure

An Arpeggio model is described by commands contained in an ASCII input file. The structure of the input file follows a nested hierarchy. The topmost level of this hierarchy is named the domain. Below the domain lies a level named procedure, followed by the region level as depicted in Figure 2.1.

The domain level contains one or more procedures. At the domain level, one will also find commands associated with describing the finite element mesh, the linear solver set-up, material properties associated with a defined material, and user functions associated with source terms and boundary conditions that are added into Arpeggio's intrinsic set of functions.

The procedure level contains one or more regions. The procedure level is also used to specify the time stepping parameters, and interactions between regions, such as data transfers. Essentially at the procedure level, loose coupling algorithms are specified. Loose coupling here is defined within the context of Arpeggio's implicitly full-coupled paradigm. Whenever an independent variables's interaction with other variables in the solution procedure is not fully represented in the global matrix, the algorithm for loose coupling of that variable and its associated equation will be described at the procedure level. This loose coupling algorithm is known as a "solution control description". The procedure level contains a command block specifying the solution control procedure. An analogy to this block in simpler codes would be top level loop. For example in time dependent applications, the solution control description block would involve a block to solve the time dependent problem repeated for each time step until the desired solution time is reached.

The Region level is used to specify details about the physics to be solved. Details related to the solve include boundary conditions and initial conditions, where materials models are applied, and where surface and volumetric source terms are applied. Here the meshed discretization and material properties described at the domain level are tied into the problem statement by virtue of their names.

Global constraint equations are also specified at the region level. At the region level, specification of information written to the output file and the frequency at which output occurs. Additional post-processing associated with the output is specified. For example, additional volumetric fields which are functions of the independent variables may be specified to be added to the output file.

There are two types of commands in the input file. The first type is referred to as a block command. A block command is a grouping mechanism. A block command contains a set of commands made up of other block commands and line commands. A line command is the second type of command. The domain, procedure, and region levels are all parsed as block commands. A block command is defined in the input file by a matching pair of Begin and End lines. For example,

```
Begin SIERRA myJob
   ... block commands
End SIERRA myJob
```

A set of key words for the block command follows the "Begin" and "End" keywords. In most cases a user-specified name is added to the block commands. In the example above the keywords, SIERRA myJob, are added. Optionally, the keyword may be left off of the end of the block.

The second type of command is the line command. A line command is used to specify parameters within a given block command. In the remaining chapters and sections of this manual, the scope of each block and line command is identified, along with summaries of the meanings. Note that the ordering of any commands within a command block is arbitrary. Thus,

```
Begin Finite Element model fluid
  Database name is pipeflow2d.g
  Use Material water for block_1
End Finite Element model fluid
```

will have the same effect as

```
Begin Finite Element model fluid
  Use Material water for block_1
  Database name is pipeflow2d.g
End Finite Element model fluid
```

And the ordering of command blocks within the domain/procedure/region blocks are arbitrary–allowing you conderable freedom to collect and arrange commands. Note that the terms "command block" and "block command" are interchangeable.

The Sierra command block must contain a block for a procedure containing at least one Region. For a case where only an Aria Region is being used:

```
Begin procedure  myProcedureName
  .
  Begin Aria region myRegionName
   .
  End   Aria region myRegionName
End procedure myProcedure
```

and similarly for a case using both Aria and Adagio Regions:

```
Begin procedure  myProcedureName
  .
  Begin Adagio region myAdagioRegionName
   .
  End   Adagio region myAdagioRegionName
  .
  Begin Aria region myAriaRegionName
   .
  End   Aria region myAriaRegionName
End procedure myProcedure
```

The procedure command block is used to contain all of the application code commands that are associated with a solution procedure defined for a set of Regions. The *myProcedureName* and *name* keywords of the procedure and region blocks are left up to you. Note that the procedure command block must be present in the input file and must contain at least one application code Region command block. The procedure command block also contains other important command blocks such as the SOLUTION CONTROL block.

2.5.1 Syntax Conventions for Commands

In this section we describe the conventions used in presenting all the command descriptions in the remainder of this manual. There are four basic kinds of tokens, or words, that an application code expects to find as it parses an input file. These are *keywords, names, parameters* and *delimiters*.

Keywords

The words which distinguish one block command, or line command, from another we term keywords. Keywords are denoted in this manual in the monospaced font, for example, `BOUNDARY CONDITION`.

Names

The word, or words, that you supply on the same line of the `begin` line of a block command, is the *name*. Many times you may need to supply this *name* as a character parameter in a separate line command. Names are denoted in italics, *name* , as are parameters.

It is worth noting that the interpreter used to process standard input command lines is also used to process lines defining algebraic operations. This means that a "-" appearing within a name would be interpreted as a subtraction operation and as a consequence, the use of "-" within a *name* is not allowed. Thus instead of

```
Begin Adagio region name-1
```

one could perhaps use

```
Begin Adagio region name_1.
```

Parameters

There are three types of input parameters one will need to supply to line commands: character strings, integers, and real numbers. These are denoted in the documentation as (C), (R), and (I), respectively.

In most cases character strings may be specified in a free format. One exception to this paradigm is when a string begins a number. In this case the character string must be specified within quotation marks in order to be properly interpreted.

Real numbers may be entered in decimal form or exponential form. For example 0.0001, .1E-3, 10.0d-5 are all equivalent. Furthermore, if a real(R) is expected, an integer can be used.

Integer values (I) need not include a decimal point in their specification.

Multiple Parameters

For the case when a list of one or more paremeters is allowed, or required, for a command, (C,...) denotes a list of character strings, (I,...) a list of integers, and (R, ...) a list of real numbers. For a list of character strings, the separator between the strings must be one or more spaces or tab characters. Therefore, phrases with multiple spaces and words in them are tokenized into multiple character parameters before being processed by the application. For a list of real or integer numbers the comma can also be used as a separator.

Enumerated Parameters

Certain commands have predefined parameters, called *enumerations*, which are listed within {}. Each parameter in the list is separated using | . The default parameter for the list of parameters is enclosed by <>.

Delimiters

The keywords of a line command are often required to be separated from the parameters by a delimiter. You have a choice of delimiters to use: the equal sign, =, or a word. In this manual, we denote the choices surrounded by {}, and separated by |. You may use any one of the delimiters from those listed. For example, the line command to specify the density within the Calore Material Block command is

`Density` {= |IS} (R)

Examples of valid forms you could write in the input file are

```
Begin Property Specification for Calore Material water
  ...
  Density = 1.0E-3  # kg/m\^3 at 20C
  ...
End
```

and

```
Begin Aria Material water
  ...
  Density is constant   rho = 1.0E-3  # kg/m\^3 at 20C
  ...
End
```

White Space

Command keywords, names, and parameters and delimiters must have spaces around them.

Indentation

All leading spaces and/or tab characters are ignored in the input file. Of course, we recommend that you use indentation to improve the readability for yourself and others that may need to see your files.

Case Sensitivity

None of the command keywords, parameters, or delimiters read from the input file are case sensitive. For example, the following two lines are equivalent:

```
Use Material water for block_1
.
```

and

```
USE material wATer for blOCK_1
.
```

The exception to this rule are file names used for input and output, because the current operating systems on which SIERRA applications are run are based on UNIX, where file names are case sensitive.

Comments and Line Continuation

You may place comments in the input file starting with either the $ or # character. All further characters on a line following a comment character are ignored.

You can continue a command in the input file to the next line by using the line continuation character $, or you may optionally following it with a comment#. All further characters on the same line following a line continuation character $ are ignored, and the characters on the following line are joined and parsing continues. An example is the line command used to specify the title of a thermal model:

```
Begin SIERRA Job_Indentifier
# This thermal model for Calore simulates a convective heat transfer

Title   The title command is used to set the analysis title $\
  Convective heat transfer to a part. The analysis $\
  makes use of conjugate heat transfer to account for  $\
  cooling of a part due to flowing water.
...
End SIERRA Job_Indentifier
```

Checking the Syntax

Errors in the input deck can be checked by adding the command, "–check-syntax" to the aria command line. For example,

```
sierra arpeggio --check-syntax -i input.i
```

This command will print the code echo of the input deck and any syntax errors within it to the display.

2.6 Fields

Fields are defined as variables which are distributed on mesh objects (e.g. nodes, elements, faces or edges). The mesh object and Field data may be distribued among parallel processors via a domain decomposition

algorithm. Each application registers Fields by name on its own Region. In a coupled-physics simulation Framework transfer services may be called on to communicate these Fields to another application. For example, the temperature Field in one application may be communicated to a solid mechanics application in order to perform a thermal-stress analysis.

2.7 User Fields

Situations often arise where one wishes to provide Field data storage so that data can be transferred into or out of the application. Each of the application codes provide some mechanism for enabling this type of data access. Additionally, User Fields are often used to as additional storage needed in user supplied subroutines.

Chapter 3

Model Definition

3.1 Model Overview

Sierra Framework services provide overall control of input commands, discretization input data and output data, IO. Additionally they provide a directed interaction of Framework services at the so-called Domain level with with the application code at the Region level. This controlled interaction is enabled by commands that follow.

The model discretization (mesh) and the mesh components to be used in the model are defined at the Domain level and are later referenced by the application at the Region level. The association of material properties with portions of the mesh are also defined here within the Finite Element Model command block/s. For some couplings using the same mesh a single Finite Element Model may be used but for most cases one will use separate Finite Element Model command blocks for each Region. A sample outline of a setup for coupling of a solid mechanics application **sm** and a thermal-fluid **tf** is shown below.

```
Begin Sierra myJob
  .
  Begin Finite Element Model my_fem_model_sm
   .
  End
  .
  Begin Finite Element Model my_fem_model_tf
   .
  End

  Begin Global Constants
   .
  End
  .
  Model definition commands

    - Material definitions for sm
    - Function definitions for sm
    - Local Coordinate Systems for sm
    .
    - Material definitions for tf
    - Function definitions for tf
    - Local Coordinate Systems for tf
  .
```

```
  Begin Procedure My_Procedure
     .
     procedural commands
       - Solution Control Description
       - Transfer operations
     .
     Begin Adagio Region My_Adagio_Region
      .
      use Finite Element Model my_fem_model_sm
      .
      - sm Region level commands
      .
     End
     .
     Begin Aria Region My_Aria_Region
      .
      use Finite Element Model my_fem_model_tf
      .
      - tf Region level commands
      .
     End
     .
  End
  .
End Sierra myJob
```

Note that a given application may not support the entire set of available options available in the Finite Element Model command block, particularly in the Parameters for Block section. Rather than attempting to include the entire set of command lines available in the Finite Element command block, only a small subset of key command lines are shown here. One should consult documentation for the specific application to find a complete listing of the relevant Finite Element Model command lines.

3.2 Finite Element Model

Scope: Sierra

Begin Finite Element Model *Label*

- Alias *DatabaseName* As *InternalName*
- Component Separator Character *Option Separator*
- Create *GroupType NewSurfaceName* Add *SurfaceName...*
- Coordinate System {=|are|is} *CoordinateSystem*
- Database Name {=|are|is} *StreamName*
- Database Type {=|are|is} *DatabaseTypes*
- Decomposition Method {=|are|is} *Method*
- Global Id Mapping Backward Compatibility *Option1 Option2*
- Omit Block *BlockList...*
- Omit Volume *VolumeList...*
- Time Scale Factor *Option Scale*

```
Use Generic Names
Use Material MaterialName For VolumeList...
Begin Parameters For Block Blockname
End
Begin Parameters For Phase Phase Name
End
Begin Parameters For Surface Surface_Name
End
End
```

Summary Describes the location and type of the input stream used for defining a geometry model for the enclosing region.

3.2.1 Alias

Scope: Finite Element Model

`Alias` *DatabaseName* `As` *InternalName*

Parameter	Value	Default
DatabaseName	`string`	`undefined`
InternalName	`string`	`undefined`

Summary Name the database entity "DatabaseName" as "InternalName"

Description This "InternalName" may then be referenced in the data file in addition to the original name.

3.2.2 Component Separator Character

Scope: Finite Element Model

`Component Separator Character` *Option Separator*

Parameter	Value	Default
Separator	`string`	`undefined`

Summary The separator is the single character used to separate the output variable basename (e.g. "stress") from the suffices (e.g. "xx", "yy") when displaying the names of the individual variable components. For example, the default separator is "_", which results in names similar to "stress_xx", "stress_yy", ... "stress_zx". To eliminate the separator, specify an empty string ("") or NONE.

3.2.3 Create

Scope: Finite Element Model

`Create` *`GroupType NewSurfaceName`* `Add` *`SurfaceName...`*

Parameter	Value	Default
NewSurfaceName	`string`	`undefined`
SurfaceName	`string...`	`undefined`

Summary Create a new set (node, edge, face, element, side/surface) as the union of two or more existing sets. The sets must exist in the mesh database or have been created by a previous CREATE command.

3.2.4 Coordinate System

Scope: Finite Element Model

`Coordinate System {=|are|is}` *`CoordinateSystem`*

Parameter	Value	Default
CoordinateSystem	{`axisymmetric`|`barycentric`|`cartesian`|`cyclidic`|`cylindrical`|`polar`|`quadriplanar`|`skew`|`spherical`|`toroidal`|`trilinear`}	`undefined`

Summary The interpretation of the geometry data stored in this database. Optional. Defaults to Cartesian.

3.2.5 Database Name

Scope: Finite Element Model

`Database Name {=|are|is}` *`StreamName`*

Parameter	Value	Default
StreamName	`string`	`undefined`

Summary The base name of the database containing the output results. If the filename begins with the '/' character, it is an absolute path; otherwise, the path to the current directory will be prepended to the name. If this line is omitted, then a filename will be created from the basename of the input file with a ".g" suffix appended.

3.2.6 Database Type

Scope: Finite Element Model

`Database Type {=|are|is}` *`DatabaseTypes`*

Parameter	Value	Default
DatabaseTypes	{`catalyst`|`dof`|`dof_exodus`|`exodus`|`exodusii`|`generated`|`genesis`|`parallel_exodus`|`xdmf`}	`undefined`

Summary The database type/format used for the mesh.

3.2.7 Decomposition Method

Scope: Finite Element Model

Summary The decomposition algorithm to be used to partition elements to each processor in a parallel run.

3.2.8 Global Id Mapping Backward Compatibility

Scope: Finite Element Model

Summary (Unsupported, do not use)

3.2.9 Omit Block

Scope: Finite Element Model

`Omit Block` *`BlockList...`*

Parameter	Value	Default
BlockList	`string...`	`undefined`

Summary Specifies that the element blocks named in the blockList be omitted from the analysis.

Description If an element block is omitted, then it is illegal to refer to it later in the input file e.g an initial condition may not be specified on an omitted element block. The elements, faces, etc are never created and it is as if the omitted element blocks did not exist in the mesh file. If a surface is completely determined by the omitted element block, then it is illegal to specify boundary conditions on that surface. However, if the surface spans multiple element blocks, boundary conditions may be applied on the portion of the surface supported by the element blocks that are not omitted.

3.2.10 Omit Volume

Scope: Finite Element Model

`Omit Volume` *`VolumeList...`*

Parameter	Value	Default
VolumeList	`string...`	`undefined`

Summary Specifies that the volumes named in the volumeList be omitted from the analysis.

Description If a volume is omitted, then it is illegal to refer to it later in the input file e.g an initial condition may not be specified on an omitted volume. The elements, faces, etc are never created and it is as if the omitted volumes did not exist in the mesh file. If a surface is completely

determined by the omitted volume, then it is illegal to specify boundary conditions on that surface. However, if the surface spans multiple volumes, boundary conditions may be applied on the portion of the surface supported by the volumes that are not omitted.

3.2.11 Time Scale Factor

Scope: Finite Element Model

`Time Scale Factor` *`Option Scale`*

Parameter	Value	Default
Scale	`real`	`undefined`

Summary The scale factor to be applied to the times on the mesh database. If the scale factor is 20 and the times on the mesh database are 0.1, 0.2, 0.3, then the application will see the mesh times as 2, 4, 6.

3.2.12 Use Generic Names

Scope: Finite Element Model

Summary If this command is present then the name of all blocks and sets in the mesh will be of the form "type_"+id. For example, an element block with id=42 will be named "block_42"; a sideset with id 314 will be named "surface_314". If there are any names in the mesh file, those names will be aliases for the blocks and sets. If this command is not present, then if a name is in the mesh file, it will be used as the name and the generic generated name will be an alias. This is used a a workaround in codes that do not correctly handle named blocks and sets or as a workaround in meshes which contain non-user-specified names.

3.2.13 Use Material

Scope: Finite Element Model

`Use Material` *`MaterialName`* `For` *`VolumeList...`*

Parameter	Value	Default
MaterialName	`string`	`undefined`
VolumeList	`string...`	`undefined`

Summary Associate the given volumes with the indicated material name.

3.3 Parameters For Block

Scope: Finite Element Model

`Begin Parameters For Block` *`Blockname`*

`Include All Blocks`

`Local Coordinate System {=|are|is}` *`Mesh Entities`*

```
    Material MatName
    Material = MatName
    Phase PhaseLabel {=|are|is} MaterialName
    Remove Block {=|are|is} ExcludeBlockList...
End
```

Summary Specifies analysis parameters associated with each element block.

3.3.1 Include All Blocks

Scope: Parameters For Block

Summary Use this parameters definition for all blocks.

When using this option within the FINITE ELEMENT MODEL command block the PARAMETERS FOR BLOCK will not use a Blockname.

3.3.2 Local Coordinate System

Scope: Parameters For Block

`Local Coordinate System {=|are|is} Mesh Entities`

Parameter	Value	Default
Mesh Entities	`string`	`undefined`

Summary Associate coordinate system with mesh entity.

Description Specify the local coordinate system to be used in conjunction with given element blocks.

3.3.3 Material

Scope: Parameters For Block

`Material MatName`

Parameter	Value	Default
MatName	`string`	`undefined`

Summary Associates this element block with its material properties.

3.3.4 Material =

Scope: Parameters For Block

`Material = MatName`

Parameter	Value	Default
MatName	`string`	`undefined`

Summary Associates this element block with its material properties.

3.3.5 Phase

Scope: Parameters For Block

`Phase` *PhaseLabel* {=|are|is} *MaterialName*

Parameter	Value	Default
PhaseLabel	`string`	`undefined`
MaterialName	`string`	`undefined`

Summary Associate phase PhaseLabel with material Material_Name on this block.

3.3.6 Remove Block

Scope: Parameters For Block

`Remove Block` {=|are|is} *ExcludeBlockList...*

Parameter	Value	Default
ExcludeBlockList	`string...`	`undefined`

Summary List of blocks to exclude.

3.4 Global Constants

Scope: Sierra

```
Begin Global Constants empty
    Gravity Vector {=|are|is} Gravity_1 Gravity_2 Gravity_3
    Ideal Gas Constant {=|are|is} Sigma
    K-E Turbulence Model Parameter Param {=|are|is} Value
    K-W Turbulence Model Parameter Param {=|are|is} Value
    Les Turbulence Model Parameter Param {=|are|is} Value
    Stefan Boltzmann Constant {=|are|is} Sigma
    Turbulence Model Param Number {=|are|is} Value
End
```

Summary Set of universal constants for a simulation.

3.4.1 Gravity Vector

Scope: Global Constants

`Gravity Vector {=|are|is}` *Gravity*$_1$ *Gravity*$_2$ *Gravity*$_3$

Parameter	Value	Default
Gravity	`real_1 real_2 real_3`	`undefined`

Summary Gravity constant in vector form, acceleration components.

3.4.2 Ideal Gas Constant

Scope: Global Constants

`Ideal Gas Constant {=|are|is}` *Sigma*

Parameter	Value	Default
Sigma	`real`	`undefined`

Summary Ideal gas constant. extbfNOTE: Another ideal gas constant value can be specified while using certain code capabilities. This global constants value will be discarded for any other specified ideal gas constant values.

3.4.3 K-E Turbulence Model Parameter

Scope: Global Constants

`K-E Turbulence Model Parameter` *Param* `{=|are|is}` *Value*

Parameter	Value	Default
Param	`string`	`undefined`
Value	`real`	`undefined`

Summary $k - \epsilon$ RANS turbulence model parameters.

3.4.4 K-W Turbulence Model Parameter

Scope: Global Constants

`K-W Turbulence Model Parameter` *Param* `{=|are|is}` *Value*

Parameter	Value	Default
Param	`string`	`undefined`
Value	`real`	`undefined`

Summary $k - \omega$ RANS turbulence model parameters.

3.4.5 Les Turbulence Model Parameter

Scope: Global Constants

`Les Turbulence Model Parameter` *Param* {=|are|is} *Value*

Parameter	Value	Default
Param	string	undefined
Value	real	undefined

Summary LES turbulence model parameters.

3.4.6 Stefan Boltzmann Constant

Scope: Global Constants

`Stefan Boltzmann Constant` {=|are|is} *Sigma*

Parameter	Value	Default
Sigma	real	undefined

Summary Stefan-Boltzmann constant. Depending on the units involved in the specific problem by the user, this value will differ.

3.4.7 Turbulence Model

Scope: Global Constants

`Turbulence Model` *Param* `Number` {=|are|is} *Value*

Parameter	Value	Default
Param	string	undefined
Value	real	undefined

Summary Turbulence model Schmidt and Prandtl numbers

3.5 Definition For Function

Scope: Sierra

`Begin Definition For Function` *FunctionName*

`Abscissa` {=|are|is} *Name...*

`Abscissa Offset` {=|are|is} *Abscissa_offset*

`Abscissa Scale` {=|are|is} *Abscissa_scale*

`At Discontinuity Evaluate To` *Option*

`Column Titles` $Titles_1$ $Titles_2$...

`Data File =` *filename* `[ X From Column` *xcol* `Y From Column` *ycol* `]`

`Debug` {=|are|is} *Option*

```
    Differentiate Expression {=|are|is} Expr
    Evaluate Expression {=|are|is} Expr
    Evaluate From x0 To x1 By Dx
    Expression Variable:  Expr = VarType value_var_name...
    Expression Variable:  Expr
    Ordinate {=|are|is} Name...
    Ordinate Offset {=|are|is} Ordinate_offset
    Ordinate Scale {=|are|is} Ordinate_scale
    Scale By x
    Type {=|are|is} Type
    X Offset {=|are|is} X_offset
    X Scale {=|are|is} X_scale
    Y Offset {=|are|is} Y_offset
    Y Scale {=|are|is} Y_scale
    Begin Expressions empty
    End

    Begin Values empty
    End
End
```

Summary Defines a function in terms of its type and values.

3.5.1 Abscissa

Scope: Definition For Function

```
Abscissa {=|are|is} Name...
```

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Specifies a string identifier for the independent variable. Optionally specify a scale and/or offset value which transforms the abscissa values into scaled_abscissa = scale * (abscissa + abscissa_offset).

3.5.2 Abscissa Offset

Scope: Definition For Function

```
Abscissa Offset {=|are|is} Abscissa_offset
```

Parameter	Value	Default
Abscissa_offset	`real`	`undefined`

Summary Alias for X OFFSET

3.5.3 Abscissa Scale

Scope: Definition For Function

`Abscissa Scale {=|are|is}` *`Abscissa_scale`*

Parameter	Value	Default
Abscissa_scale	`real`	`undefined`

Summary Alias for X SCALE

3.5.4 At Discontinuity Evaluate To

Scope: Definition For Function

Summary Control the behavior of a piecewise constant function when evaluated at a discontinuity (plus or minus a small tolerance). The default behavior is to take the value to the right of the discontinuity. If "Left" is specified, the value to the left of the discontinuity is taken instead.

3.5.5 Column Titles

Scope: Definition For Function

`Column Titles` *`Titles`*$_1$ *`Titles`*$_2$`...`

Parameter	Value	Default
Titles	`string_1 string_2...`	`undefined`

Summary Name the columns (and also defined the expected number of columns) for Multicolumn Piecewise Linear tabular data.

3.5.6 Data File

Scope: Definition For Function

`Data File =` *`filename`* `[ X From Column` *`xcol`* `Y From Column` *`ycol`* `]`

Parameter	Value	Default
filename	`string`	`undefined`

Summary Function will read tabular data from an input file. Compatible with the piecewise linear function type. File must be of form like:

EXAMPLE FILE 1.099 1191 1.101 221 5.9011 133.1

Lines headed by a # are considered comments and will be ignored. Data itself must by in tabular columns seperated by whitespace or commas.

3.5.7 Debug

Scope: Definition For Function

Summary Prints functions to the log file.

3.5.8 Differentiate Expression

Scope: Definition For Function

`Differentiate Expression {=|are|is}` ***Expr***

Parameter	Value	Default
Expr	`(expression)`	`undefined`

Summary Specifies the expression of derivative of evaluation expression.

3.5.9 Evaluate Expression

Scope: Definition For Function

`Evaluate Expression {=|are|is}` ***Expr***

Parameter	Value	Default
Expr	`(expression)`	`undefined`

Summary Specifies the expression to evaluate.

Description This will greatly help with manufactured solutions, and be useful for other purposes as well.

This first implementation goes like this:

```
begin definition for function pressure
type is analytic
evaluate expression is "x <= 0.0 ?  0.0 :  (x < 0.5 ?  x*200.0 :  (x <
1.0 ?  (x - 0.5) *50.0 + 100.00 :  150.0));"
# type is piecewise linear
# begin values
# 0.0 0.0
# 0.5 100.0
# 1.0 150.0
# end values
end definition for function pressure
```

Also, notice that semicolon at the end. Be sure to put it there for now. You can actually provide multiple expressions to be evaluated, each terminated with a semicolon. This will be handy when multi-dependent variable come into the fold.

The following functions are currently implemented.

Operators All C-language operators are supported, e.g. $+ - */ \| ? :$ etc

Parens ()

Math Functions

abs(x) absolute value of x

mod(x, y) modulus of x|y

ipart(x) integer part of x

fpart(x) fractional part of x

min(x0, x1, ...) minimum value of xn

max(x0, x1, ...) maximum value of xn

Power functions

pow(x, y) x to the y power

sqrt(x) square root of x

Trig functions

sin(x) sine of x

sinh(x) hyperbolic sine of x

asin(x) arcsine of x

cos(x) cosine of x

cosh(x) hyperbolic cosine of x

acos(x) arccosine of x

tan(x) tangent of x

tanh(x) hyperbolic tangent of x

atan(x) arctangent of x

atan2(y, x) arctangent of y/x, signs of x and y determine quadrant (see atan2 man page)

Logarithm functions

log(x) natural logarithm of x

ln(x) natural logarithm of x

exp(x) e to the x power

logn(x, y) the y base logarithm of x

Rounding functions

ceil(x) smallest integral value not less than x

floor(x) largest integral value not greater than x

Random functions

rand(x) random number between 0.0 and 1.0, not including 1.0

srand(x) seeds the random number generator

Conversion routines

deg(x) converts radians to degrees

rad(x) converts degrees to radians

recttopolr(x, y) maginitude of vector x, y

recttopola(x, y) angle of vector x, y

poltorectx(r, theta) x coordinate of angle theta at distance r

poltorecty(r, theta) y coordinate of angle theta at distance r

3.5.10 Evaluate From

Scope: Definition For Function

`Evaluate From` *x0* `To` *x1* `By` *Dx*

Parameter	Value	Default
x0	`real`	`undefined`
x1	`real`	`undefined`
Dx	`real`	`undefined`

Summary Specifies the range and evaluation interval.

3.5.11 Expression Variable:

Scope: Definition For Function

`Expression Variable:` *Expr* `=` *VarType value_var_name...*

Parameter	Value	Default
Expr	`string`	`undefined`
value_var_name	`string...`	`undefined`

Summary Specifies what the arguments of an expression correspond to. For example:

BEGIN DEFINITION FOR FUNCTION dx_shear TYPE = ANALYTIC EXPRESSION variable: mx = NODAL model_coordinates(x) EXPRESSION variable: my = NODAL model_coordinates(y) EXPRESSION variable: time = GLOBAL time EVALUATE EXPRESSION = "(time/termTime)*(stretchx*(
- 0.0) + ((my-0.25)/0.5)*stretchxy)" END

Assuming the above expression is being evaluated on nodes the current values for x and y model coordinates would be placed into mx and my and current analysis time placed into time

3.5.12 Expression Variable:

Scope: Definition For Function

`Expression Variable:` *Expr*

Parameter	Value	Default
Expr	`string`	`undefined`

Summary Specifies what the arguments of an expression exists, but does not define it correspond to. For example:

BEGIN DEFINITION FOR FUNCTION dx_shear TYPE = ANALYTIC EXPRESSION variable: mx EXPRESSION variable: my EXPRESSION variable: time EVALUATE EXPRESSION = "(time/termTime)*(stretchx*(mx - 0.0) + ((my-0.25)/0.5)*stretchxy)" END

Call function must determine what each variable actually is is based off of the string name

3.5.13 Ordinate

Scope: Definition For Function

`Ordinate {=|are|is} Name...`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Specifies a string identifier for the dependent variable. Optionally specify a scale and/or offset value which transforms the ordinate values into scaled_ordinate = scale * (ordinate + ordinate_offset).

3.5.14 Ordinate Offset

Scope: Definition For Function

`Ordinate Offset {=|are|is} Ordinate_offset`

Parameter	Value	Default
Ordinate_offset	`real`	`undefined`

Summary Alias for Y OFFSET

3.5.15 Ordinate Scale

Scope: Definition For Function

`Ordinate Scale {=|are|is} Ordinate_scale`

Parameter	Value	Default
Ordinate_scale	`real`	`undefined`

Summary Alias for Y SCALE

3.5.16 Scale By

Scope: Definition For Function

`Scale By x`

Parameter	Value	Default
x	`real`	`undefined`

Summary Specifies a scale factor to be applied.

3.5.17 Type

Scope: Definition For Function

Summary Specifies the type of function.

3.5.18 X Offset

Scope: Definition For Function

`X Offset {=|are|is}` *`X_offset`*

Parameter	Value	Default
X_offset	`real`	`undefined`

Summary Sets an offset for the x-axis

3.5.19 X Scale

Scope: Definition For Function

`X Scale {=|are|is}` *`X_scale`*

Parameter	Value	Default
X_scale	`real`	`undefined`

Summary Sets a scale factor for the x-axis

3.5.20 Y Offset

Scope: Definition For Function

`Y Offset {=|are|is}` *`Y_offset`*

Parameter	Value	Default
Y_offset	`real`	`undefined`

Summary Sets an offset for the y-axis

3.5.21 Y Scale

Scope: Definition For Function

`Y Scale {=|are|is}` *`Y_scale`*

Parameter	Value	Default
Y_scale	`real`	`undefined`

Summary Sets a scale factor for the y-axis

3.6 Values

Scope: Definition For Function

```
Begin Values empty
     Xyvalues...
End
```

Summary Lists the values of the function. The values should be listed one pair per line, independent variable first, with whitespace or comma as a separator.

3.6.1

Scope: Values

Xyvalues...

Parameter	Value	Default
Xyvalues	`real...`	`undefined`

Summary For a piecewise linear function, lists an x-y pair for the nth interpolation point.

3.7 Restart Overview

Sierra Framework services provide convenient utilities for restarting an analysis from previous results. The most general capability supplements the results of a previous analysis with internal state variables to continue an analysis. In this case the input mesh is supplied from the Input Database Name from the Finite Element Model command block 3.1 and the restart information is obtained from the the Input Database Name from the Restart Data command block. Continuation of a job using restart data output is invoked using the command line which follows.

3.7.1 Restart Time

Scope:

`Restart Time {=|are|is}` *Time*

Parameter	Value	Default
Time	`real`	`undefined`

Summary Specify restart file read at a specified time.

Description **NOTE:** This command must be placed at the Sierra scope of the input file.

Specify the time that the analysis will be restarted. In addition to this line command, each Region in the analysis (strictly, only the region(s) that will be restarted) must have a restart block specifying the database to read the restart state data. The restart 'time' must be greater than zero and less than or equal to the termination time.

By default, use of this command will cause previous output files (e.g., results, history, heartbeat, restart) to be overwritten. If this command is chosen, the onus is placed on the user to ensure that previous output files are not overwritten.

Chapter 4

Solution Control Reference

4.1 Overview

Aria uses the *solution control* (SC) library from the SIERRA Framework to orchestrate execution of simulations. All Aria input files must include a Solution Control Description block in the Procedure section of the input file. This description contains directives for executing either a steady-state (sequential) or transient analysis either of which can include nested nonlinear iteration or subcycling. Within the description one selects a named solution control system where the details of execution are more clearly spelled out. Because there are similarities between the Sequential, Transient, Nonlinear Iteration and Subcycling many operations are shared between these directives. However, each of these segments must be uniquely named internally so they can be properly managed under solution control.

Within each SC system, execution of a problem defined at the Region level corresponds to an Advance directive. Thus a steady-state analysis could conceivably be carried out with a single Advance directive. For transient analysis the system can contain several time blocks, each with a corresponding Advance directive. Examples of different control structures are given below.

An example the solution control command block for steady-state analysis would reflect the structure indicated below:

```
.
.

Begin Procedure myProcedure

  Begin Solution Control Description
     Use System Main
     Begin System Main
        Begin Sequential MySolveBlock
           Advance myRegion
        End
     End
  End

  Begin Aria Region myRegion
    .
    .
    .

  End Aria Region myRegion

End Procedure myProcedure
```

.
.

A solution control command block for steady-state analysis containing nonlinear iteration for Aria and Adagio would reflect the general structure indicated below. Note that advancement of the solution can be governed by a user specified criteria, `Parameters for Nonlinear Iteration`:

```
.
.

Begin Procedure myProcedure

  Begin Solution Control Description
     Use System Main
     Begin System Main
        Begin Sequential MySolveBlock
           Begin Nonlinear Iteration
             Advance myAriaRegion
             Advance myAdagioRegion
             transfer adagio_to_aria
           End Nonlinear Iteration
        End
     End

     Begin Parameters for Nonlinear Iteration
       converged [When-expression]
     End Parameters for Nonlinear Iteration

  End

  Begin Aria Region myAriaRegion
    .
    .
  End Aria Region myAriaRegion

  Begin Adagio Region myAdagioRegion
    .
    .
  End Adagio Region myAdagioRegion

End Procedure myProcedure
.
.
```

In the case of transient analysis the solution control command block will contain specification of times for which the analysis will be carried out. Additionally parameters defining the time integration must also be supplied by the user. Details concerning time integration parameters are included in the user manual for the application. A simple example the solution control command block for transient analysis would resemble the structure indicated below:

```
.
.
```

```
Begin Procedure My_Aria_Procedure

   Begin Solution Control Description

      Use System Main

      Begin System Main
         Simulation Start Time            = 0.0
         Simulation Termination Time      = 10.0
         Simulation Max Global Iterations = 1000

         Begin Transient Time_Block_1
            Advance My_Aria_Region
         End
         Begin Transient Time_Block_2
            Advance My_Aria_Region
         End

      End

      Begin Parameters For Transient Time_Block_1
         Start Time      = 0.0
         Number of steps = 8
         Begin Parameters For Aria Region My_Aria_Region
            Time Step Variation    = Fixed
            Initial Time Step Size = 0.001
         End
      End

      Begin Parameters For Transient Time_Block_2
         Begin Parameters For Aria Region My_Aria_Region
            Time Step Variation           = Adaptive
            Initial Time Step Size        = 0.001
            Predictor-Corrector Tolerance = 1e-3
            Minimum Time Step Size        = 1e-6
         End
      End

   End
   .
   .
```

Similarly subcycled iterations in a one-way coupling between Aria and Presto could also be carried out in a transient analysis. In this case Presto subcycles at a small time, Aria has a larger time step and Aria is advanced when the two time steps arrive at the same solution time.

```
.
.
Begin Procedure My_Aria_Procedure

   Begin Solution Control Description

      Use System Main
```

```
      Begin System Main
         Simulation Start Time            = 0.0
         Simulation Termination Time      = 10.0
         Simulation Max Global Iterations = 1000

         Begin Transient Time_Block_1
            Transfer Presto_to_Aria
            Advance My_Aria_Region
             Begin Subcycle PrestoSubcycle
                Advance PrestoRegion
             End
         End

      End

      Begin Parameters For Transient Time_Block_1
         Start Time      = 0.0
         Number of steps = 8

         Begin Parameters For Aria Region My_Aria_Region
            Time Step Variation    = Fixed
            Initial Time Step Size = 0.001
         End

         Begin Parameters for Presto Region PrestoRegion
            initial time step = 1.0e-6
            # time step scale factor = 1.0
            time step increase factor = 10.
            # step interval = 500
         End
      End

   End
    .
    .

  Begin Aria Region myAriaRegion
    .
    .
  End Aria Region myAriaRegion

  Begin Presto Region myPrestoRegion
    .
    .
  End Presto Region myPrestoRegion

End Procedure myProcedure
.
.
```

It is important to note that Solution Control can orchestrate the execution of one Region or the execution of many Regions. Within a loosely-coupled code analysis SC is also used to control the movement of data between the coupled codes using the Transfer subsystem.

The outline views of various couplings include both **Transfer** and **Advance** events. In the examples above the event will always occur in the sequence specified. Alternatively one can specify that the event be carried out conditionally subject to criteria described syntactically as a "C" language $[When - expression]$ where the expression criteria includes internal code variables or explicit evaluations. Here the input $[When - expression]$ is parsed and transformed into an executable "C" statement. While some of the internal code variables used by a $[When - expression]$ are intuitive (i.e. `CURRENT_TIME` and `CURRENT_STEP`) many others are application dependent. The most widely used explicit evaluations are measures of convergence based upon solution residuals `adagio.norm(0.0)` for solid mechanics applications and `aria.MaxResidualNorm(0.0)` for thermal-fluid applications. Several examples of $[When - expression]$ are given below noting that the "C" expression must be enclosed in quotes within the input file.

Convergence based upon comparison of application residuals:

```
Begin parameters for nonlinear converge_step_p1
  # following two lines shown must be a single input command line
  converged when $"(aria.MaxResidualNorm(0.0) < 1.e-6 && adagio.norm(0.0)
                   < 1.e-6) || CURRENT_STEP > 2000"
End parameters for nonlinear converge_step_p1
```

Transfer at first step and then every four steps:

```
Transfer aria_to_adagio when "(CURRENT_STEP == 1) || (CURRENT_STEP % 4 == 0)"
```

Advance the region at second step:

```
advance aria_region when "CURRENT_STEP == 2"
```

Additionally, one may also use application specific global variables in the $[When - expression]$ criteria. Global variables that are generally available for use are listed as such in the simulation log file. Unfortunately these variables may not be directly accessible to the user. Hence consultation with an application developer may be required in this regard.

In the Solution Control syntax described below it is implied that the optional $[When - expression]$s represent evaluations which will be interpreted at parse time.

4.2 Solution Control Description

Scope: Procedure

```
Begin Solution Control Description Name

    Use System Name

    Begin Initialize Name
    End

    Begin Parameters For
    End
```

```
    Begin System Name
    End

End
```

Summary Contains the commands needed to execute an analysis using the arpeggio procedure that utilizes Solver Control.

4.2.1 Use System

Scope: Solution Control Description

```
Use System Name
```

Parameter	Value	Default
Name	string	undefined

Summary This set the name of which system to use.

4.3 System

Scope: Solution Control Description

```
Begin System Name
    Adapt Region_name... Using Field_name... [ When When-expression ]
    Compute Indicator On Region_name... Using Indicator_name... [ When When-expression
    ]
    Event Name... [ When When-expression ]
    Execute Postprocessor Group Group_name... On Region_name... [ When When-expression
    ]
    Indicatemarkadapt Region_name Using Indicator Marker [ When When-expression ]
    Mark Region_name... Using Marker_name... [ When When-expression ]
    Markadapt Region_name Using Marker [ When When-expression ]
    Output Name [ When When-expression ]
    Simulation Max Global Iterations {=|are|is} Number
    Simulation Start Time {=|are|is} Number
    Simulation Termination Time {=|are|is} Number
    Transfer Name [ When When-expression ]
    Use Initialize Name
    Begin Adaptivity Name
    End

    Begin Sequential Name
    End

    Begin Transient Name
```

```
    End

End
```

Summary This block wraps a solver system for a given name. The NAME parameter is the name used to define the system. There can be more than one system block in the Solver Control Description block. The "use system NAME" line commmand controls which one is to be used.

4.3.1 Adapt

Scope: System

`Adapt` *`Region_name...`* `Using` *`Field_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Field_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.3.2 Compute Indicator On

Scope: System

`Compute Indicator On` *`Region_name...`* `Using` *`Indicator_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Indicator_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.3.3 Event

Scope: System

`Event` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.3.4 Execute Postprocessor Group

Scope: System

`Execute Postprocessor Group` *`Group_name...`* `On` *`Region_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Group_name	`string...`	`undefined`
Region_name	`string...`	`undefined`

Summary Used within a Solver Control block to cause the group named group_name to be executed on region region_name.

4.3.5 Indicatemarkadapt

Scope: System

`Indicatemarkadapt` *`Region_name`* `Using` *`Indicator Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Indicator	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Compute Indicator On ... Mark ... Adapt ...

4.3.6 Mark

Scope: System

`Mark` *`Region_name...`* `Using` *`Marker_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Marker_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.3.7 Markadapt

Scope: System

`Markadapt` *`Region_name`* `Using` *`Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Mark ... Adapt ...

4.3.8 Output

Scope: System

`Output` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Output line command which execute a perform I/O on the region.

4.3.9 Simulation Max Global Iterations

Scope: System

`Simulation Max Global Iterations {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`integer`	`undefined`

Summary The Total number of Solves.

4.3.10 Simulation Start Time

Scope: System

`Simulation Start Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Simulation starting time. (by default 0.0)

4.3.11 Simulation Termination Time

Scope: System

`Simulation Termination Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary The drop dead time.

4.3.12 Transfer

Scope: System

`Transfer` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.3.13 Use Initialize

Scope: System

```
Use Initialize Name
```

Parameter	Value	Default
Name	`string`	`undefined`

Summary This set the name of which initialization to use.

4.4 Transient

Scope: System

```
Begin Transient Name
    Adapt Region_name... Using Field_name... [ When When-expression ]
    Advance Name... [ When When-expression ]
    Compute Indicator On Region_name... Using Indicator_name... [ When When-expression
    ]
    Event Name... [ When When-expression ]
    Execute Postprocessor Group Group_name... On Region_name... [ When When-expression
    ]
    Indicatemarkadapt Region_name Using Indicator Marker [ When When-expression ]
    Involve Name
    Mark Region_name... Using Marker_name... [ When When-expression ]
    Markadapt Region_name Using Marker [ When When-expression ]
    Output Name [ When When-expression ]
    Transfer Name [ When When-expression ]
    Begin Adaptivity Name
    End

    Begin Nonlinear Name
    End

    Begin Subcycle Name
    End
End
```

Summary This block is used to wrap a time loop.

4.4.1 Adapt

Scope: Transient

`Adapt` *`Region_name...`* `Using` *`Field_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Field_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.4.2 Advance

Scope: Transient

`Advance` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

4.4.3 Compute Indicator On

Scope: Transient

`Compute Indicator On` *`Region_name...`* `Using` *`Indicator_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Indicator_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.4.4 Event

Scope: Transient

`Event` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.4.5 Execute Postprocessor Group

Scope: Transient

`Execute Postprocessor Group` *`Group_name...`* `On` *`Region_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Group_name	`string...`	`undefined`
Region_name	`string...`	`undefined`

Summary Used within a Solver Control block to cause the group named group_name to be executed on region region_name.

4.4.6 Indicatemarkadapt

Scope: Transient

`Indicatemarkadapt` *`Region_name`* `Using` *`Indicator Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Indicator	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Compute Indicator On ... Mark ... Adapt ...

4.4.7 Involve

Scope: Transient

`Involve` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

4.4.8 Mark

Scope: Transient

`Mark` *`Region_name...`* `Using` *`Marker_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Marker_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.4.9 Markadapt

Scope: Transient

`Markadapt` *Region_name* `Using` *Marker* `[ When` *When-expression* `]`

Parameter	Value	Default
Region_name	string	undefined
Marker	string	undefined

Summary Shortcut line command... equivalent to: Mark ... Adapt ...

4.4.10 Output

Scope: Transient

`Output` *Name* `[ When` *When-expression* `]`

Parameter	Value	Default
Name	string	undefined

Summary A Solver Control Output line command which execute a perform I/O on the region.

4.4.11 Transfer

Scope: Transient

`Transfer` *Name* `[ When` *When-expression* `]`

Parameter	Value	Default
Name	string	undefined

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.5 Nonlinear

Scope: Sequential

`Begin Nonlinear` *Name*

`Adapt` *Region_name...* `Using` *Field_name...* `[ When` *When-expression* `]`

`Advance` *Name...* `[ When` *When-expression* `]`

`Compute Indicator On` *Region_name...* `Using` *Indicator_name...* `[ When` *When-expression* `]`

`Event` *Name...* `[ When` *When-expression* `]`

`Execute Postprocessor Group` *Group_name...* `On` *Region_name...* `[ When` *When-expression* `]`

`Indicatemarkadapt` *Region_name* `Using` *Indicator Marker* `[ When` *When-expression* `]`

```
    Involve Name
    Mark Region_name... Using Marker_name... [ When When-expression ]
    Markadapt Region_name Using Marker [ When When-expression ]
    Output Name [ When When-expression ]
    Transfer Name [ When When-expression ]
    Begin Subcycle Name
    End
End
```

Summary This block is used to wrap a nonlinear solve loop.

4.5.1 Adapt

Scope: Nonlinear

```
Adapt Region_name... Using Field_name... [ When When-expression ]
```

Parameter	Value	Default
Region_name	string...	undefined
Field_name	string...	undefined

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.5.2 Advance

Scope: Nonlinear

```
Advance Name... [ When When-expression ]
```

Parameter	Value	Default
Name	string...	undefined

Summary Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

4.5.3 Compute Indicator On

Scope: Nonlinear

```
Compute Indicator On Region_name... Using Indicator_name... [ When When-expression ]
```

Parameter	Value	Default
Region_name	string...	undefined
Indicator_name	string...	undefined

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.5.4 Event

Scope: Nonlinear

`Event` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.5.5 Execute Postprocessor Group

Scope: Nonlinear

`Execute Postprocessor Group` *`Group_name...`* `On` *`Region_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Group_name	`string...`	`undefined`
Region_name	`string...`	`undefined`

Summary Used within a Solver Control block to cause the group named group_name to be executed on region region_name.

4.5.6 Indicatemarkadapt

Scope: Nonlinear

`Indicatemarkadapt` *`Region_name`* `Using` *`Indicator Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Indicator	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Compute Indicator On ... Mark ... Adapt ...

4.5.7 Involve

Scope: Nonlinear

`Involve` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

4.5.8 Mark

Scope: Nonlinear

`Mark` *`Region_name...`* `Using` *`Marker_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Marker_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.5.9 Markadapt

Scope: Nonlinear

`Markadapt` *`Region_name`* `Using` *`Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Mark ... Adapt ...

4.5.10 Output

Scope: Nonlinear

`Output` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Output line command which execute a perform I/O on the region.

4.5.11 Transfer

Scope: Nonlinear

`Transfer` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.6 Subcycle

Scope: Nonlinear

```
Begin Subcycle Name
    Adapt Region_name... Using Field_name... [ When When-expression ]
    Advance Name... [ When When-expression ]
    Compute Indicator On Region_name... Using Indicator_name... [ When When-expression
    ]
    Event Name... [ When When-expression ]
    Execute Postprocessor Group Group_name... On Region_name... [ When When-expression
    ]
    Indicatemarkadapt Region_name Using Indicator Marker [ When When-expression ]
    Involve Name
    Mark Region_name... Using Marker_name... [ When When-expression ]
    Markadapt Region_name Using Marker [ When When-expression ]
    Output Name [ When When-expression ]
    Transfer Name [ When When-expression ]
End
```

Summary This block is used to wrap a subcycle time loop.

4.6.1 Adapt

Scope: Subcycle

```
Adapt Region_name... Using Field_name... [ When When-expression ]
```

Parameter	Value	Default
Region_name	string...	undefined
Field_name	string...	undefined

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.6.2 Advance

Scope: Subcycle

```
Advance Name... [ When When-expression ]
```

Parameter	Value	Default
Name	string...	undefined

Summary Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

4.6.3 Compute Indicator On

Scope: Subcycle

`Compute Indicator On` *Region_name...* `Using` *Indicator_name...* `[ When` *When-expression* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Indicator_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.6.4 Event

Scope: Subcycle

`Event` *Name...* `[ When` *When-expression* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.6.5 Execute Postprocessor Group

Scope: Subcycle

`Execute Postprocessor Group` *Group_name...* `On` *Region_name...* `[ When` *When-expression* `]`

Parameter	Value	Default
Group_name	`string...`	`undefined`
Region_name	`string...`	`undefined`

Summary Used within a Solver Control block to cause the group named group_name to be executed on region region_name.

4.6.6 Indicatemarkadapt

Scope: Subcycle

`Indicatemarkadapt` *Region_name* `Using` *Indicator Marker* `[ When` *When-expression* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Indicator	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Compute Indicator On ... Mark ... Adapt ...

4.6.7 Involve

Scope: Subcycle

`Involve` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

4.6.8 Mark

Scope: Subcycle

`Mark` *`Region_name...`* `Using` *`Marker_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Marker_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.6.9 Markadapt

Scope: Subcycle

`Markadapt` *`Region_name`* `Using` *`Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Mark ... Adapt ...

4.6.10 Output

Scope: Subcycle

`Output` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Output line command which execute a perform I/O on the region.

4.6.11 Transfer

Scope: Subcycle

```
Transfer Name [ When When-expression ]
```

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.7 Sequential

Scope: System

```
Begin Sequential Name
    Adapt Region_name... Using Field_name... [ When When-expression ]
    Advance Name... [ When When-expression ]
    Compute Indicator On Region_name... Using Indicator_name... [ When When-expression
    ]
    Event Name... [ When When-expression ]
    Execute Postprocessor Group Group_name... On Region_name... [ When When-expression
    ]
    Indicatemarkadapt Region_name Using Indicator Marker [ When When-expression ]
    Involve Name
    Mark Region_name... Using Marker_name... [ When When-expression ]
    Markadapt Region_name Using Marker [ When When-expression ]
    Output Name [ When When-expression ]
    Transfer Name [ When When-expression ]
    Begin Adaptivity Name
    End

    Begin Nonlinear Name
    End
End
```

Summary This block is used to wrap a sequential solution. It is used to wrap a sequence of Non-Linear or pseudo time solve step solves.

4.7.1 Adapt

Scope: Sequential

```
Adapt Region_name... Using Field_name... [ When When-expression ]
```

Parameter	Value	Default
Region_name	`string...`	`undefined`
Field_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.7.2 Advance

Scope: Sequential

`Advance` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

4.7.3 Compute Indicator On

Scope: Sequential

`Compute Indicator On` *`Region_name...`* `Using` *`Indicator_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Indicator_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.7.4 Event

Scope: Sequential

`Event` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.7.5 Execute Postprocessor Group

Scope: Sequential

`Execute Postprocessor Group` *`Group_name...`* `On` *`Region_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Group_name	`string...`	`undefined`
Region_name	`string...`	`undefined`

Summary Used within a Solver Control block to cause the group named group_name to be executed on region region_name.

4.7.6 Indicatemarkadapt

Scope: Sequential

`Indicatemarkadapt` *`Region_name`* `Using` *`Indicator Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Indicator	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Compute Indicator On ... Mark ... Adapt ...

4.7.7 Involve

Scope: Sequential

`Involve` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

4.7.8 Mark

Scope: Sequential

`Mark` *`Region_name...`* `Using` *`Marker_name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string...`	`undefined`
Marker_name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a mesh adaptment on the specific block should be performed.

4.7.9 Markadapt

Scope: Sequential

`Markadapt` *`Region_name`* `Using` *`Marker`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Region_name	`string`	`undefined`
Marker	`string`	`undefined`

Summary Shortcut line command... equivalent to: Mark ... Adapt ...

4.7.10 Output

Scope: Sequential

`Output` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Output line command which execute a perform I/O on the region.

4.7.11 Transfer

Scope: Sequential

`Transfer` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.8 Initialize

Scope: Solution Control Description

```
Begin Initialize Name
    Advance Name...  [ When When-expression ]
    Event Name...  [ When When-expression ]
    Involve Name
    Transfer Name [ When When-expression ]
End
```

Summary This block wraps a initializer for a given name. The NAME parameter is the name used to define the initialization block. There can be more than one initialize block in the Solver Control Description block. The "use initialize NAME" line commmand controls which one is to be used.

4.8.1 Advance

Scope: Initialize

`Advance` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

4.8.2 Event

Scope: Initialize

`Event` *Name...* `[ When` *When-expression* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.8.3 Involve

Scope: Initialize

`Involve` *Name*

Parameter	Value	Default
Name	`string`	`undefined`

Summary Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

4.8.4 Transfer

Scope: Initialize

`Transfer` *Name* `[ When` *When-expression* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.9 Parameters For

Scope: Solution Control Description

`Begin Parameters For`

`Converged When` *Convergence-expression*

`Incremental Number Of Steps {=|are|is}` *Number*

`Initial Deltat {=|are|is}` *Number*

`Number Of Adaptivity Steps {=|are|is}` *Number*

`Number Of Steps {=|are|is}` *Number*

```
Reinitialize Transient
Start Time {=|are|is} Number
Termination Time {=|are|is} Number
Time Step Quantum {=|are|is} TimeStepQuantum
Time Step Style TimeStepStyle...
Total Change In Time {=|are|is} Number
Begin Parameters For Aria Region RegionName
End
End
```

Summary A Solver Control PARAMETERS block to set up control data for the SC_type parameter. Inside this block one sets the time step parameters or nonlinear parameters.

4.9.1 Converged When

Scope: Parameters For

```
Converged When Convergence-expression
```

Parameter	Value	Default
Convergence-expression	(expression)	undefined

Summary Set the convergence expression.

4.9.2 Incremental Number Of Steps

Scope: Parameters For

```
Incremental Number Of Steps {=|are|is} Number
```

Parameter	Value	Default
Number	integer	undefined

Summary The incremental number steps to run the time for nonlinear loop. Number of time steps to run after restarting. NUMBER OF STEPS is total number of steps to run

4.9.3 Initial Deltat

Scope: Parameters For

```
Initial Deltat {=|are|is} Number
```

Parameter	Value	Default
Number	real	undefined

Summary Assign an initial delta T

4.9.4 Number Of Adaptivity Steps

Scope: Parameters For

`Number Of Adaptivity Steps {=|are|is} Number`

Parameter	Value	Default
Number	`integer`	`undefined`

Summary The number steps to run the time or nonlinear loop

4.9.5 Number Of Steps

Scope: Parameters For

`Number Of Steps {=|are|is} Number`

Parameter	Value	Default
Number	`integer`	`undefined`

Summary The number steps to run the time for nonlinear loop

4.9.6 Reinitialize Transient

Scope: Parameters For

Summary Reset time and re-initialize regions each step of the adaptivity loop.

4.9.7 Start Time

Scope: Parameters For

`Start Time {=|are|is} Number`

Parameter	Value	Default
Number	`real`	`undefined`

Summary Assign a start time.

4.9.8 Termination Time

Scope: Parameters For

`Termination Time {=|are|is} Number`

Parameter	Value	Default
Number	`real`	`undefined`

Summary Assign a final time to stop

4.9.9 Time Step Quantum

Scope: Parameters For

`Time Step Quantum {=|are|is}` *`TimeStepQuantum`*

Parameter	Value	Default
TimeStepQuantum	`real`	`undefined`

Summary Set the time stepping quantum time for SNAP style stepping.

4.9.10 Time Step Style

Scope: Parameters For

`Time Step Style` *`TimeStepStyle...`*

Parameter	Value	Default
TimeStepStyle	{`clip`|`noclip`|`nosnap`|`snap`}	`undefined`

Summary Set the time stepping style.

When CLIP is specified, the time step size will be clipped at the last step of the transient loop so that it ends at the transient loop's end time. If clip is not specified, the last time is allowed to exceed to the transient loop's end time and the following transient loop will start at the exceeded end time.

When SNAP is specified, the time step is broken down into "quantum" time units. By default this quantum time is 12 orders of magnitude down from the difference between the start and end time for the transient loop. This value can be overridden using the TIME STEP QUANTUM line command. All time values are "snapped" to multiples of the quantum time by rounding to the nearest quantum multiple.

4.9.11 Total Change In Time

Scope: Parameters For

`Total Change In Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Use this number and the initial time to compute termination time.

4.9.12 Advance

Scope:

`Advance` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

4.9.13 Converged When

Scope:

`Converged When` *`Convergence-expression`*

Parameter	Value	Default
Convergence-expression	`(expression)`	`undefined`

Summary Set the convergence expression.

4.9.14 Event

Scope:

`Event` *`Name...`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string...`	`undefined`

Summary Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

4.9.15 Initial Deltat

Scope:

`Initial Deltat {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Assign an initial delta T

4.9.16 Involve

Scope:

`Involve` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

4.9.17 Number Of Adaptivity Steps

Scope:

`Number Of Adaptivity Steps {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`integer`	`undefined`

Summary The number steps to run the time or nonlinear loop

4.9.18 Number Of Steps

Scope:

`Number Of Steps {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`integer`	`undefined`

Summary The number steps to run the time for nonlinear loop

4.9.19 Output

Scope:

`Output` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Output line command which execute a perform I/O on the region.

4.9.20 Reinitialize Transient

Scope:

Summary Reset time and re-initialize regions each step of the adaptivity loop.

4.9.21 Simulation Max Global Iterations

Scope:

`Simulation Max Global Iterations {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`integer`	`undefined`

Summary The Total number of Solves.

4.9.22 Simulation Start Time

Scope:

`Simulation Start Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Simulation starting time. (by default 0.0)

4.9.23 Simulation Termination Time

Scope:

`Simulation Termination Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary The drop dead time.

4.9.24 Start Time

Scope:

`Start Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Assign a start time.

4.9.25 Termination Time

Scope:

`Termination Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Assign a final time to stop

4.9.26 Time Step Quantum

Scope:

`Time Step Quantum {=|are|is}` *`TimeStepQuantum`*

Parameter	Value	Default
TimeStepQuantum	`real`	`undefined`

Summary Set the time stepping quantum time for SNAP style stepping.

4.9.27 Time Step Style

Scope:

`Time Step Style` *`TimeStepStyle...`*

Parameter	Value	Default
TimeStepStyle	{`clip`|`noclip`|`nosnap`|`snap`}	`undefined`

Summary Set the time stepping style.

When CLIP is specified, the time step size will be clipped at the last step of the transient loop so that it ends at the transient loop's end time. If clip is not specified, the last time is allowed to exceed to the transient loop's end time and the following transient loop will start at the exceeded end time.

When SNAP is specified, the time step is broken down into "quantum" time units. By default this quantum time is 12 orders of magnitude down from the difference between the start and end time for the transient loop. This value can be overridden using the TIME STEP QUANTUM line command. All time values are "snapped" to multiples of the quantum time by rounding to the nearest quantum multiple.

4.9.28 Total Change In Time

Scope:

`Total Change In Time {=|are|is}` *`Number`*

Parameter	Value	Default
Number	`real`	`undefined`

Summary Use this number and the initial time to compute termination time.

4.9.29 Transfer

Scope:

`Transfer` *`Name`* `[ When` *`When-expression`* `]`

Parameter	Value	Default
Name	`string`	`undefined`

Summary A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

4.9.30 Use Initialize

Scope:

`Use Initialize` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary This set the name of which initialization to use.

4.9.31 Use System

Scope:

`Use System` *`Name`*

Parameter	Value	Default
Name	`string`	`undefined`

Summary This set the name of which system to use.

Chapter 5

Transfer Reference

5.1 Overview

Recall that Sierra Mechanics supports application data associated with nodes, elements, faces or edges of a meshed discretization as in Figure 1.2. The Sierra Transfer utility provides the means by which to communicate data between two Sierra application Regions. Generally speaking the same type of data is most often communicated but data movement need not be for the same type, e.g. nodal data can be communicated to element data and vice-versa.

The Transfer utility is fairly flexible as it provides the ability to move data directly onto another problem domain either by direct copy or by interpolation. Analysts without prior experience with transfer are often uncertain as to which type of transfer to use. The two capabilities function exactly as their names imply but understanding which method to use requires a basic understanding of how each method works.

Copy transfer assumes that the discretization for applications involved in the transfer are identical. Moreover, copy transfer also assumes that the mesh is identical so that global IDs of nodes and elements within each mesh are the same. Under these assumptions a geometric search of source to destination locations is not necessary and a simple algorithm is able to perform the data transfer in a straightforward manner.

Interpolation transfer is much more general than copy transfer since it assumes only that data from one application must be geometrically mapped for use in another application. A mathematical definition of this mapping is made possible using the results from a geometric search of points on the destination mesh and their image on the sending mesh. With regard to code performance copy transfer will always more efficient than interpolation transfer but is rarely applicable in mainstream simulations. Interpolate transfer is designed to deal with complications that arise in mapping data from one application to the other and is more reliable. As a rule, one should always use interpolation transfer and not copy transfer. At the same time an analyst should strategize model construction so as to offset some of the performance costs of interpolation transfer.

Even with a basic understanding of transfer users of what transfer operations should be defined. Several proper transfer source and destinations are illustrated in Figure 5.1, here the numbers on the figures correspond to the ExodusII global IDs of nodes or elements.

Problematic transfer source and destination configurations are illustrated in Figure 5.2. Once again the numbers on the figures correspond to the ExodusII global IDs of nodes or elements.

In using the transfer utility one must clearly define the sending region (where the data resides) and the the receiving region (the data destination). Additionally one must also specify the general geometric location of data sender and receiver based upon existing mesh entities (blocks or surfaces). Sender and receiver need not be of same topology but the source and target destinations should overlap geometrically. Clearly the definition mesh entities influences time spent in the geometric search process and should be a key consideration in model construction.

The following section outlines the commands to be used in setting up transfer operations. Special

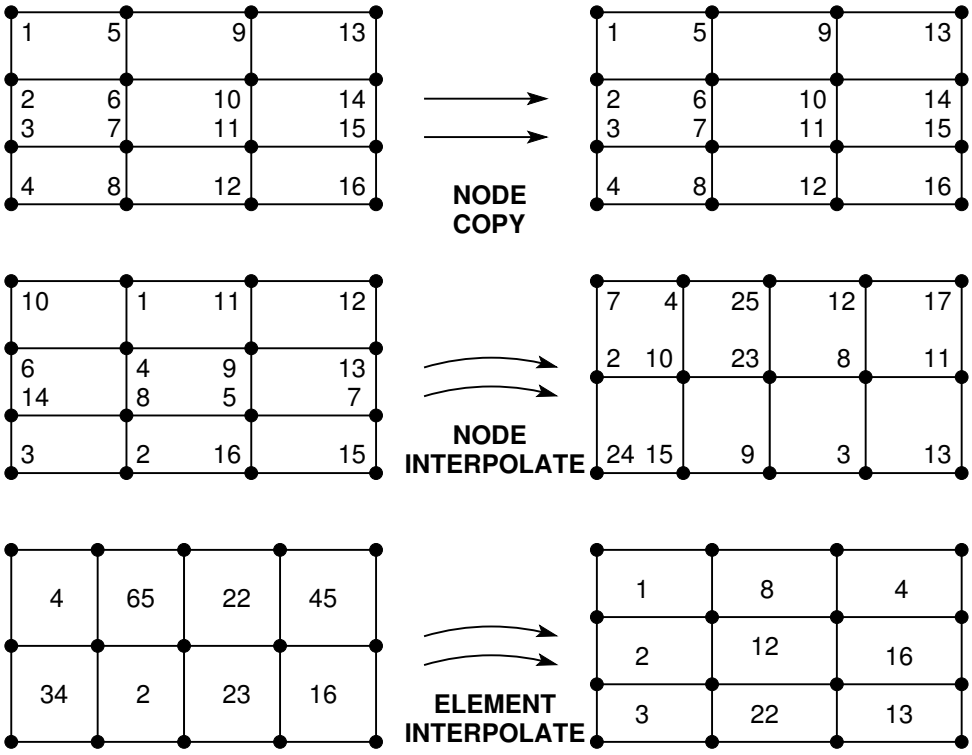

Figure 5.1. Valid Transfer Operations

attention should be paid to the syntax of the SEND command line since it differs between COPY and INTERPOLATION transfer.

Since several different uses of transfer can arise and several of those examples for steady problems are included below. The same basic setup of transfer would apply to transient problems as well.

A skeleton outline of one-way transfer from Region_1 to Region_2 in a steady-state problem would be:

```
Begin Sierra
  .
  Begin Transfer my_transfer
    .
    transfer commands for first_region to second_region
    .
  End
  .
  Begin Procedure My_Aria_Procedure
     .
    Begin Solution Control Description
       Use System Main
       Begin System Main
          Begin Sequential MySolveBlock
             Advance first_Region
             transfer my_transfer
             Advance second_Region
```

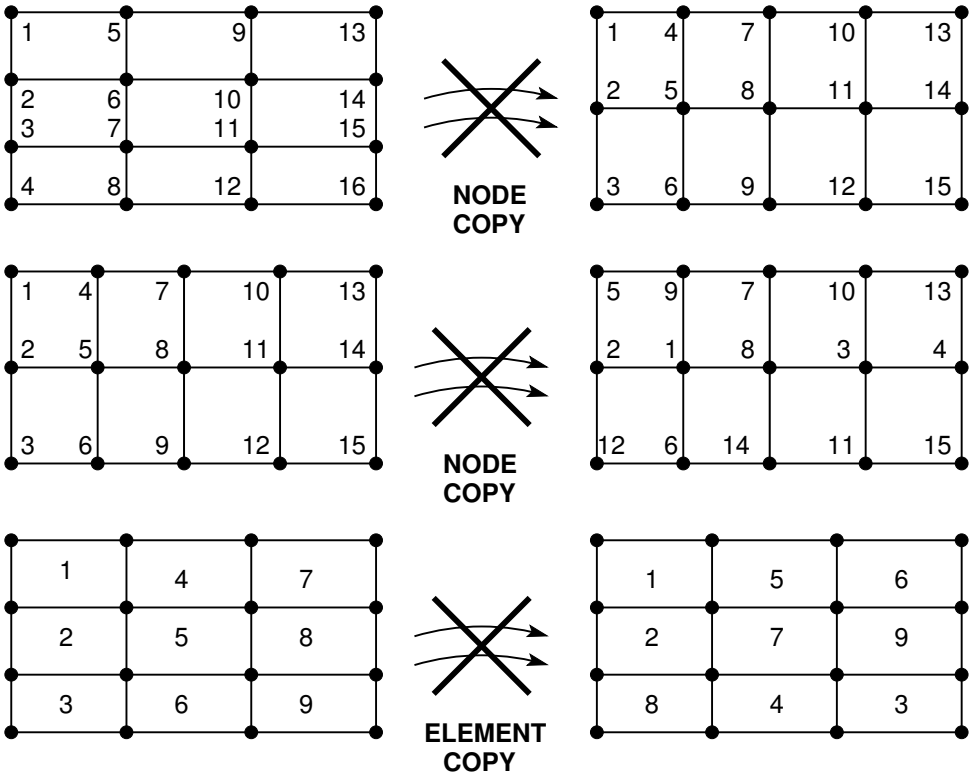

Figure 5.2. Invalid Transfer Operation

```
        End
      End
    End

    Begin Aria Region first_region
      .
      eq energy for temperature On block_1 using q1 with lumped_mass diff
      .
    End

    Begin Aria Region second_region
      .
      eq energy for temperature On block_1 using q1 with xfer
      .
    End

  End
  .
End Sierra
```

A skeleton outline of two-way transfer between Region_1 to Region_2 in a steady-state problem would be:

```
Begin Sierra
  .
  Begin Transfer my_first_transfer
    .
    transfer commands for first_region to second_region
    .
  End
  .
  Begin Transfer my_second_transfer
    .
    transfer commands for second_region to first_region
    .
  End
  .
  Begin Procedure My_Aria_Procedure
     .
    Begin Solution Control Description
       Use System Main
       Begin System Main
          Begin Sequential MySolveBlock
             Advance first_Region
             transfer my_first_transfer
             Advance second_Region
             transfer my_second_transfer
          End
       End
    End

    Begin Aria Region first_region
      .
      eq energy for temperature On block_1 using q1 with diff
      eq species_3 for temperature On block_1 using q1 with xfer
      .
    End

    Begin Aria Region second_region
      .
      eq energy for temperature On block_1 using q1 with xfer
      eq species_3 for species_3 On block_1 using q1 with diff
      .
    End

  End
  .
End Sierra
```

Assume an input mesh for an Input_Output Region 6.1 contains a nodal variable ConvCoeff. In this case a skeleton outline for one-way transfer of ConvCoeff to to Region_2 in a steady-state problem would be:

```
Begin Sierra
  .
  Begin Transfer my_first_transfer
    .
    transfer commands for input_output_region to second_region
    .
    SEND field hNd state none TO ConvCoeff state none
    .
  End
  .
  Begin Procedure My_Aria_Procedure
     .
    Begin Solution Control Description
       Use System Main
       Begin System Main
          Begin Sequential MySolveBlock
             Advance first_Region
             transfer my_first_transfer
             Advance second_Region
          End
       End
    End

    Begin Input_Output io_region
       USE FINITE ELEMENT MODEL my_input_transfer
    End

    Begin Aria Region second_region
      .
      USER FIELD REAL NODE SCALAR ConvCoeff on surface_1
      .
    End

  End
  .
End Sierra
```

5.2 Transfer

Scope: Procedure

Begin Transfer *Transfer_name*

Abort If Field Not Defined On Copy Transfer Send Or Receive Object

All Fields

Copy *Option1 Option2* From *From_region_name* To *To_region_name*

Distance Function Is Closest Receive Node To Send Centroid

Exclude Ghosted

```
    From Option1 To Option2

    Gauss Point Integration Order {=|are|is} Order

    Interpolate Option1 Option2 From From_region_name To To_region_name

    Interpolation Function User_Subroutine

    Nodes Outside Region {=|are|is} Option

    Search Coordinate Field Source_field_name State Option1 To Destination_field_name
    State Option2

    Search Geometric Tolerance {=|are|is} Geometric_tolerance

    Search Surface Gap Tolerance {=|are|is} Surface_gap_tolerance [ Or Less ]

    Search Type {=|are|is} [ Option1 Option2 Option3 ]

    Select One Receiver For Each Send Object

    Select One Unique Receiver For Each Send Object

    Send Predefined-transfer Fields

    Send Block From_blocks...  To To_blocks...

    Send Field Source_field_name State Option1 To Destination_field_name State Option2
    [ Lower Bound Lower_bound Upper Bound Upper_bound ]

    Begin Receive Blocks
    End

    Begin Send Blocks
    End

End
```

Summary transfer region/mesh information. the mechanics/variables information will get sorted out by the calling procedure.

5.2.1 Abort If Field Not Defined On Copy Transfer Send Or Receive Object

Scope: Transfer

Summary For testing purposes only. Normally mesh objects in the send or receive mesh which do not have the specified field defined on them are just ignored. This line command allows the construction of tests in which it is known that every mesh object should have the specified field defined on it and to abort if that field is not found.

5.2.2 All Fields

Scope: Transfer

Summary Select all fields for transfer that have same name and state for source and destination regions.

5.2.3 Copy

Scope: Transfer

`Copy` *`Option1 Option2`* `From` *`From_region_name`* `To` *`To_region_name`*

Parameter	Value	Default
From_region_name	`string`	`undefined`
To_region_name	`string`	`undefined`

Summary Copy transfer elements, nodes or constraints from one region to another. The copy transfer is very specific in that the sending and receiving mesh parts must have identical global ids for every element to be copied. The copy transfer works by iterating over all the mesh objects in the receiving mesh and using the global id of the receiving mesh object to find a mesh object in the sending mesh with the same global id. The field to transfer is then copied from the sending to receiving objects. There is no interpolation and the actual coordinates of the sending and receiving objects are not used and could be very different. The copy transfer is used in very special cases where the same mesh was read into both the sending and receiving meshes, there was no element death and there was no adaptivity. In this special case, a copy transfer can be much faster than an interpolation transfer.

5.2.4 Distance Function Is Closest Receive Node To Send Centroid

Scope: Transfer

Summary To be used in conjunction with "SELECT ONE UNIQUE RECEIVER FOR EACH SEND OBJECT". This helped in the case where the sending and receiving element blocks did not overlap and an element transfer was using element centroids for the distance computation. The elements were very distorted so that a centroid of a surface element could be far from the surface. It was wanted that the receiving element be the one close to the surface of the block and close to the sending element in the adjacent block. Using the corner nodes was enough since it was a tet mesh with plane faces. In this particular and unusual case this alternative method of matching sending and receiving elements was useful, but it is not expected to be used often or maybe never again.

5.2.5 Exclude Ghosted

Scope: Transfer

Summary exclude ghosted nodes from a copy transfer

5.2.6 From

Scope: Transfer

Summary Allows the send/receive mesh objects to be different.

5.2.7 Gauss Point Integration Order

Scope: Transfer

`Gauss Point Integration Order {=|are|is}` *`Order`*

Parameter	Value	Default
Order	`integer`	`undefined`

Summary Integration order to use when transferring to Gauss points.

5.2.8 Interpolate

Scope: Transfer

`Interpolate` *`Option1 Option2`* `From` *`From_region_name`* `To` *`To_region_name`*

Parameter	Value	Default
From_region_name	`string`	`undefined`
To_region_name	`string`	`undefined`

Summary Interpolate will transfer elements, nodes or constraints from one mesh to another. The interpolation transfer is very general in that the field values to transfer will be interpolated from the sending to receiving mesh based on the coordinates of the sending and receiving mesh objects.

Many line commands can be used to modify the behavior of the interpolation transfer but the basic algorithm is straightforward. Every mesh object in the receiving mesh is converted into a point. For elements this is the average of the nodal coordinates. An element in the sending mesh containing this point is found. If the field to transfer is nodal, the element shape functions are used to interpolate the nodal field to the receiving point. If the field to transfer is elemental, a bi-linear least squares fit based upon neighboring elements is first performed and then used to define the interpolation of the element field at the receiving point.

5.2.9 Interpolation Function

Scope: Transfer

`Interpolation Function` *`User_Subroutine`*

Parameter	Value	Default
User_Subroutine	`string`	`undefined`

Summary Allows an application defined subroutine to be used for the interpolation. Normally the interpolation transfer will determine the best type of interpolation to use: Basis functions for nodal fields and a neighborhood least squares fit for element fields. This line command can be used to override this if needed. It also allows an application to register it's own special interpolation functions that can then be used if the special name it was registered with is known.

5.2.10 Nodes Outside Region

Scope: Transfer

Summary This line command defines what to do when a receiving point is outside the scope of the sending mesh.

IGNORE - The receiving mesh object can be ignored and will receive no value. This is almost never a good idea as it can cause mesh objects just outside to have a zero value when the nodes just inside the mesh might have very large values. This can result in a discontinuous receiving field.

EXTRAPOLATE - This is the default behavior. The sending field is extrapolated beyond the bounds of the sending mesh. This can lead to extrapolation error, such as when a large gradient at the surface causes a negative values when only positive values are acceptable. If this happens to the upper and lower bounds that can be placed on the fields to be transferred with the SEND FIELD command.

TRUNCATE - The receiving coordinate is projected back to the surface of the sending mesh to determine a value. This ensures that the receiving value is outside of the field values in the sending mesh.

PROJECT - This option is similar to TRUNCATE in which the receiving coordinate is projected back to the surface of the sending mesh to determine a value. In this case more effort is made to make sure that the projection is normal to the surface in the sending mesh. Sometimes gives a better result than Truncate but is a little more expensive to compute.

If the PROJECT option is used in transferring of surface values, the sending mesh should envelope the receiving mesh. Failure to satisfy this condition will generally result in failure of the transfer.

5.2.11 Search Coordinate Field

Scope: Transfer

`Search Coordinate Field` *`Source_field_name`* `State` *`Option1`* `To` *`Destination_field_name`* `State` *`Option2`*

Parameter	Value	Default
Source_field_name	`string`	`undefined`
Destination_field_name	`string`	`undefined`

Summary Normally the interpolation transfers use the default coordinate field to determine geometry information. This line command can be used to specify an alternate field.

5.2.12 Search Geometric Tolerance

Scope: Transfer

`Search Geometric Tolerance {=|are|is}` *`Geometric_tolerance`*

Parameter	Value	Default
Geometric_tolerance	`real`	`undefined`

5.2.13 Search Surface Gap Tolerance

Scope: Transfer

`Search Surface Gap Tolerance {=|are|is}` *Surface_gap_tolerance* `[ Or Less ]`

Parameter	Value	Default
Surface_gap_tolerance	`real`	`undefined`

Summary

This is a tricky parameter best ignored, let it default to some small number. During the interpolation transfer there is a geometric search based on the coordinates of the send and receive objects. As part of this search, an axis aligned bounding box is contracted for each sending object and SEARCH GAP TOLERANCE is used to make this box bigger than just a tight bounding box. Lists of receiving points are then quickly found within these axis aligned boxes.

If all points in the receiving mesh are within at least one box, no additional searching needs to be done and the search algorithm is fast. If there are still points in the receiving mesh that were outside of EVERY box, then a warning message will be issued about an "expensive search for extrapolation" for these points. This 'expensive search" can be very costly if a large number of receiving objects fall into this category and this line command is provided for those special cases.

The OR LESS optional parameter is used when the tolerance must be set to large value for one part of the mesh but much of the mesh needs a much smaller value. In some cases it is neccessary for the tolerance to be set to the actual largest surface gap tolerance which may be far too large a gap for the rest of the mesh. Setting OR LESS allows the search tolerance to be reduced in areas of the mesh thus resulting in a faster search.

5.2.14 Search Type

Scope: Transfer

5.2.15 Select One Receiver For Each Send Object

Scope: Transfer

Summary

This option will cause each sending object to be used once and only once. This will have the side effect of some receiving objects not getting any value at all. If you use this option, you will also want to set NODES OUTSIDE REGION IGNORE The example which necessitated this option was a case in which there was a delta function defined on an element in the sending mesh. It was desirable that the delta functions be summed into the receiving mesh such that the total value of the sending was conserved. It was better to have only a single element on the receiving side have a non-zero value that was the sum of sending values and not worry about how close the receiving element was to the sending element. A check that this option is working is to use Encore to computer the sum of the values of the sending and receiving fields to make sure the total sum is the same.

5.2.16 Select One Unique Receiver For Each Send Object

Scope: Transfer

Summary

An unusual flag to get around an odd problem. Normally each receive object transfers from the nearest sending object so it is almost always the case that a send object will be used

multiple times to define a receiving value. This option will cause each sending object to be used only once. This will have the side effect of some receiving objects not getting any value at all. If you use this option, you will also want to set NODES OUTSIDE REGION IGNORE or else the uniqueness will be lost for nodes outside the sending region. The example which necessitated this option was a case in which there was a delta function defined on an element in the sending mesh. It was desirable that the delta function be defined on the receiving mesh for only a single element in the neighborhood of the sending element. The analysis was more sensitive to the number of delta functions on the receiving side than the location. So it was better to have only a single element on the receiving side have a non-zero value and not worry about how close the receiving element was to the sending element.

5.2.17 Send

Scope: Transfer

`Send` *`Predefined-transfer`* `Fields`

Parameter	Value	Default
Predefined-transfer	{}	`undefined`

Summary Use predefine transfer semantics provided by the specified name.

5.2.18 Send Block

Scope: Transfer

`Send Block` *`From_blocks...`* `To` *`To_blocks...`*

Parameter	Value	Default
From_blocks	`string...`	`undefined`
To_blocks	`string...`	`undefined`

Summary Add element blocks to a particular same mesh element copy transfer operator.

The copy transfer can have multiple of these lines to define many blocks, but each line sends a single block to a single block: SEND BLOCK block_1 TO block_1 SEND BLOCK block_101 TO block_101

The interpolation transfer can have only a single SEND BLOCK line, but can define many from/to blocks: SEND BLOCK block_3 block_5 block_6 TO block_3 block_5

5.2.19 Send Field

Scope: Transfer

`Send Field` *`Source_field_name`* `State` *`Option1`* `To` *`Destination_field_name`* `State` *`Option2`* `[ Lower Bound` *`Lower_bound`* `Upper Bound` *`Upper_bound`* `]`

Parameter	Value	Default
Source_field_name	`string`	`undefined`
Destination_field_name	`string`	`undefined`

Summary

Specifies the mapping between source and destination field names. Vector and tensor fields can be subscripted using parenthesis and 1's based or brackets and 0 based. Notes on subscripting: (0) Does not work for COPY transfers, only INTERPOLATION type transfers. (1) If the field name itself actually contains either parenthesis or brackets then we are in trouble and an error is going to be thrown due to a syntax error in index specification. (2) Only a single subscript is allowed so vectors of vectors or higher order tensors can not use double subscripts. But it should be possible to determine the correct offset within the field and pick out the correct value with a little effort. (3) Once subscripted, only a single value will be transferred. It is not possible to transfer multiple values starting at a certain index, instead multiple line commands must be used, as shown above. (4) The indexes can be 0 based with brackets or 1 based when using parenthesis. Although this could be very confusing if mixed within a single line command. (5) Both the from and to fields can be subscripted independently on the same line.

example SEND FIELD velocity TO velocity SEND FIELD temp TO temperature lower bound 0 SEND FIELD x TO y lower bound 10 upper bound 100 SEND FIELD A(2) TO B(3) lower bound 10 upper bound 100 SEND FIELD A[1] TO B[2] lower bound 10 upper bound 100

Chapter 6

Input Output Region Reference

6.1 Input_Output Region Overview

For some coupled simulations one can approximate part of the problem physics independent of the entire problem physics. In order to facilitate this type of loose application coupling the Sierra Framework provides the ability to input datasets that include the output of other simulations. An application can then make requests of information from these datasets. In fullfilling these requests, data can be extracted from these datasets and be copied or interpolated to another problem domain. Moreover these requests can be satisfied by data interpolated through time. The mechanism provided to achieve this end goal is known as the Input_Output Region and its usage is described in what follows.

The input_output region works in tandem with transfer 5.1 and solution control 4. Here transfer carries out the communication of data and solution control provides synchronization of the data transfer. Note that just like other Sierra Regions the input_output region must have its own Finite Element model command block defined.

As an example, let us assume that an input mesh for an Input_Output Region contains a nodal variable ConvCoeff that we wish to use in another Region. In this case an outline for one-way transfer of ConvCoeff to to a Region, *second_region*, in a steady-state problem would be:

```
Begin Sierra
  .
  Begin Finite Element Model input_transfer
   .
  End
  .
  Begin Transfer my_first_transfer
    .
    transfer commands for input_output_region to second_region
    .
    SEND field hNd state none TO ConvCoeff state none
    .
  End
  .
  Begin Procedure My_Aria_Procedure
     .
    Begin Solution Control Description
       Use System Main
       Begin System Main
          Begin Sequential MySolveBlock
             Advance io_region
             transfer my_first_transfer
```

```
            Advance second_Region
          End
       End
    End

    Begin Input_Output io_region
       USE FINITE ELEMENT MODEL my_input_transfer
    End

    Begin Aria Region second_region
      .
      use Finite Element Model input_transfer
      .
      USER FIELD REAL NODE SCALAR ConvCoeff on surface_1
      .
    End

  End
  .
End Sierra
```

6.2 Input_Output Region

Scope: Procedure

```
Begin Input_Output Region Parameter_block_name

    Create Element Field Field_name Of Type Option And Dimension Dimension [ Value {=
    |are|is} Number...  ]

    Create Nodal Field Field_name Of Type Option And Dimension Dimension [ Value {=|
    are|is} Number...  ]

    Fixed Time [ {=|are|is} Fixed_time ]

    Offset Time {=|are|is} Period_offset_time

    Periodicity Time {=|are|is} Periodicity_time

    Start Time {=|are|is} Start_time

    Use Finite Element Model ModelName [ Model Coordinates Are Nodal_variable_name ]

    Begin Results Output Label
    End

End
```

Summary BEGIN INPUT TRANSFER model_name USE FINITE ELEMENT MODEL fred START TIME is 0 OFFSET TIME is 1 PERIODICITY TIME is 10 END INPUT TRANSFER model_name

6.2.1 Create Element Field

Scope: Input_Output Region

`Create Element Field` *`Field_name`* `Of Type` *`Option`* `And Dimension` *`Dimension`* `[` *`Value`* `{=|are|is}` *`Number...`* `]`

Parameter	Value	Default
Field_name	`string`	`undefined`
Dimension	`integer`	`undefined`

Summary Creates a Element Field name field_name on the region.

6.2.2 Create Nodal Field

Scope: Input_Output Region

`Create Nodal Field` *`Field_name`* `Of Type` *`Option`* `And Dimension` *`Dimension`* `[` *`Value`* `{=|are|is}` *`Number...`* `]`

Parameter	Value	Default
Field_name	`string`	`undefined`
Dimension	`integer`	`undefined`

Summary Creates a Nodal Field name field_name on the region.

6.2.3 Fixed Time

Scope: Input_Output Region

Summary The line specifies that the database will be read for a single, fixed time. Specifying the actual time is optional. If the time is not specified, the final time plane in the database will be read.

NOTE: This option take precedence over the periodic specifications given by START TIME, PERIODICITY TIME, and OFFSET TIME.

if FIXED TIME is specified then if FIXED TIME value is given then (eg., FIXED TIME is 1.) DATABASE TIME = FIXED TIME else (eg., FIXED TIME) DATABASE TIME = last time in database else if PERIODICITY TIME greater than 0 then if APPLICATION TIME less than or equal to START TIME then DATABASE TIME = APPLICATION TIME else DATABASE TIME = START TIME + (APPLICATION TIME - START TIME) modulo PERIODICITY TIME else DATABASE TIME = APPLICATION TIME now add OFFSET TIME to the computed DATABASE TIME

6.2.4 Offset Time

Scope: Input_Output Region

`Offset Time {=|are|is}` *`Period_offset_time`*

Parameter	Value	Default
Period_offset_time	`real`	`undefined`

Summary This value is added to the application time to determine what database time slice to input. If OFFSET TIME were 15 than at application time 0 database time slice 15 would be read from the file and used for the initial values. At application time 1, database time slice 16 would be read. NOTE: The OFFSET TIME is added in after the START TIME and PERIODICITY TIME are used. The FIXED TIME option take precedence over this option.

if FIXED TIME is specified then if FIXED TIME value is given then (eg., FIXED TIME is 1.) DATABASE TIME = FIXED TIME else (eg., FIXED TIME) DATABASE TIME = last time in database else if PERIODICITY TIME greater than 0 then if APPLICATION TIME less than or equal to START TIME then DATABASE TIME = APPLICATION TIME else DATABASE TIME = START TIME + (APPLICATION TIME - START TIME) modulo PERIODICITY TIME else DATABASE TIME = APPLICATION TIME now add OFFSET TIME to the computed DATABASE TIME

6.2.5 Periodicity Time

Scope: Input_Output Region

`Periodicity Time {=|are|is}` *`Periodicity_time`*

Parameter	Value	Default
Periodicity_time	`real`	`undefined`

Summary START TIME and PERIODICITY TIME taken together give the time frame from the input database to use to initialize the application values. If START TIME is 25 and PERIODICITY TIME is 10, then time slices from 25 to 35 will be used over and over again as the application time runs from 0 to whatever. In general DATABASE TIME is (APPLICATION TIME - START TIME) modulo PERIODICITY TIME after the application time reaches the START TIME.

NOTE: The OFFSET TIME is added in after the START TIME and PERIODICITY TIME are used. The FIXED TIME option take precedence over this option.

if FIXED TIME is specified then if FIXED TIME value is given then (eg., FIXED TIME is 1.) DATABASE TIME = FIXED TIME else (eg., FIXED TIME) DATABASE TIME = last time in database else if PERIODICITY TIME greater than 0 then if APPLICATION TIME less than or equal to START TIME then DATABASE TIME = APPLICATION TIME else DATABASE TIME = START TIME + (APPLICATION TIME - START TIME) modulo PERIODICITY TIME else DATABASE TIME = APPLICATION TIME now add OFFSET TIME to the computed DATABASE TIME

6.2.6 Start Time

Scope: Input_Output Region

`Start Time {=|are|is}` *`Start_time`*

Parameter	Value	Default
Start_time	`real`	`undefined`

Summary The time in which to start applying PERIODICITY TIME. If PERIODICITY TIME is not specified then START TIME is ignored.

NOTES: The OFFSET TIME is added in after the START TIME and PERIODICITY TIME are used. The FIXED TIME option take precedence over this option.

if FIXED TIME is specified then if FIXED TIME value is given then (eg., FIXED TIME is 1.) DATABASE TIME = FIXED TIME else (eg., FIXED TIME) DATABASE TIME = last time in database else if PERIODICITY TIME greater than 0 then if APPLICATION TIME less than or equal to START TIME then DATABASE TIME = APPLICATION TIME else DATABASE TIME = START TIME + (APPLICATION TIME - START TIME) modulo PERIODICITY TIME else DATABASE TIME = APPLICATION TIME now add OFFSET TIME to the computed DATABASE TIME

6.2.7 Use Finite Element Model

Scope: Input_Output Region

`Use Finite Element Model` *`ModelName`* `[ Model Coordinates Are` *`Nodal_variable_name`* `]`

Parameter	Value	Default
ModelName	`string`	`undefined`

Summary Associates a predefined finite element model with this region.

Chapter 7

Examples

Sierra application code couplings with Arpeggio can be carried out in a variety of ways. In this chapter a few simple problems are used to demonstrate some of the coupling approaches.

Here we note that success in performing the coupling hinges upon defining a proper setup for each of the application codes participating in the coupling. Understandably the coupling becomes more straightforward if one begins by first setting up each of the independent application code problems (i.e. an application Region) and later unites the Regions under Arpeggio.

The purpose of the examples is simply to demonstrate the basics of how the problem setup will differ for various use cases. The examples given here illustrate the use cases most likely to occur:

- One-way coupling of TF with Adagio from file on same mesh 7.1,
- One-way coupling of TF with Adagio from file on different mesh 7.2,
- One-way coupling of TF with Adagio on same mesh using transfer 7.3,
- Two-way coupling of TF with Adagio on same mesh 7.4,
- One way coupling of TF with another TF, same mesh 7.6,
- One way coupling of TF with Presto on same mesh with subcycling 7.5,

7.1 One-Way Coupling From File

In many problems of coupled physics one of the physics (primary) is dependent upon the other physics (secondary) but not vice-versa. In this case the coupling is considered to be one-way and can be accomplished simply by supplying a secondary physics solution to the primary physics simulation. In the context of problem solutions one would first solve the secondary physics problem and then communicate the solution to a primary physics simulation. Perhaps the easiest way to carry out such a simulation is to supply the secondary physics solution to the primary physics via file. The following example describes the process as it might be carried out in Arpeggio.

7.1.1 Problem Statement

Consider a one-way coupled thermal structural analysis problem in which a body is free to expand as a response to gradual temperature change in time. Although the problem geometry is changing due to the structural deformation, the geometry change is assumed to have minimal effect upon heat transfer in the body. For each time step, a heat conduction problem was solved for the temperature distribution using the Aria code and the results were written to file. The Aria output file is then used as the input file for Adagio

where the temperatures are read into Adagio. Adagio subsequently solves for mechanical equilibrium which includes calculation of thermal strains due to changing temperatures.

Here we note that the thermal solution file time planes need not correspond to the Adagio time planes as the thermal solution will be interpolated in time to match the Adagio solution time. Furthermore, in this problem, an Aria results file is the Adagio input discretization so the problems correspond to the same mesh. Here it is important that the input Aria discretization contain the nodesets and sidesets needed to carry out the Adagio simulation. Problems in which one might wish to solve the Adagio problem on a different discretization can also be dealt with but in a slightly different manner.

7.1.2 Input File

7.2 One-Way Coupling Using Transfer From Different Mesh

In some coupled physics one of the physics (primary) is dependent upon the other physics (secondary) but not vice-versa. In this case the coupling is considered to be one-way and can be accomplished simply by supplying a secondary physics solution to the primary physics simulation. In the context of problem solutions one would first solve the secondary physics problem and then communicate the solution to a primary physics simulation. As previously demonstrated one way to carry out such a simulation is to supply the secondary physics solution to the primary physics via file 7.1. However, in some cases the secondary physics solution is available on a vastly different geometry. In this case the secondary physics solution must be interpolated onto the primary physics as needed. In Sierra Mechanics the communication step of such an analysis is carried out using **Solution Control** and **Transfer** operations. Here **Transfer** describes the information and **Solution Control** ensures sequencing of information to the primary physics. The following example describes the solution process to perform a coupled analysis using a precomputed thermal solution and Adagio.

7.2.1 Problem Statement

Consider a one-way coupled thermal structural analysis problem in which a body is free to expand as a response to gradual temperature change in time. Althought the problem geometry is changing due to the structural deformation, the geometry change is assumed to have minimal effect upon heat transfer in the body. For this situation a reasonable approach may be to precompute the heat transfer solution and then supply it to the mechanical simulation. Here a transient heat conduction problem on a full geometry was solved for the temperature distribution using the Aria code and the results were saved to file. Later on the previously computed temperature distribution was supplied to Adagio for solution of mechanical equilibrium which includes calculation of thermal strains due to changing temperatures. In this particular case the Adagio problem could be solved by invoking symmetry conditions so the model geometry is a subset of the thermal model geometry.

In this particular case the Adagio problem could be solved by invoking symmetry conditions so the model geometry is a subset of the thermal model geometry. During the simulation the transient thermal solution is read from file these results are then communicated to Adagio using a transfer operation. Once the Aria values are received by Adagio the structural problem is then solved. Since the thermal and structural model geometries are different, it is necessary to use the transfer **INTERPOLATE** operation. Note that the problem advances with the two applications lock stepped in time with the thermal solution is being interpolated in both space and time.

7.2.2 Input File

7.3 One-Way Coupling Using Transfer

In many problems of coupled physics one of the physics (primary) is dependent upon the other physics (secondary) but not vice-versa. In this case the coupling is considered to be one-way and can be accomplished simply by supplying a secondary physics solution to the primary physics simulation. In the context of problem solutions one would first solve the secondary physics problem and then communicate the solution to a primary physics simulation. One way to carry out such a simulation is to supply the secondary physics solution to the primary physics via file 7.1. However, in many instances it is more convenient to carry out both simulations simultaneously and directly communicate the secondary physics solution to the primary physics as needed. In Sierra Mechanics the communication step of such an analysis is carried out using **Solution Control** and **Transfer** operations. Here **Transfer** describes the information and **Solution Control** ensures sequencing of information to the primary physics. The following example describes the solution process to perform a coupled analysis using Aria and Adagio.

7.3.1 Problem Statement

Consider a one-way coupled thermal structural analysis problem in which a body is free to expand as a response to gradual temperature change in time. Althought the problem geometry is changing due to the structural deformation, the geometry change is assumed to have minimal effect upon heat transfer in the body. For each time step, a heat conduction problem was solved for the temperature distribution using the Aria code. Once the thermal solution has been obtained the temperature solution is communicated to Adagio via Transfer and Adagio then solves for mechanical equilibrium which includes calculation of thermal strains due to changing temperatures.

Note that the problem advances with the two applications lock stepped in time. In this problem the Aria input discretization is identical to that of Adagio. During the simulation an Aria solution is performed and Aria results are then communicated to Adagio using a transfer **COPY** operation. Once the Aria values are received by Adagio the structural problem is then solved. Problems in which one might wish to solve the Aria and Adagio problems on different discretizations can dealt with by making simple modifications to the input replacing the transfer **COPY** operation with a **INTERPOLATE** operation.

7.3.2 Input File

7.4 Two-Way Coupling With Transfer

7.4.1 Problem Statement

This is a test of solving a simple one-dimensional thermal diffusion problem with Dirichlet BCs. The test problem is shown schematically in Figure. Although the problem is one-dimensional we solve the problem in a three-dimensional setting. Once the diffusion problem has been solved numerically the temperature result is postprocessed to obtain a comparison with the analytical result and the distribution of diffusive heat flux. This test input also demonstrates the use tabular function and Encore function material property specification in Aria.

7.4.2 Input File

7.5 estack Regression Test

7.5.1 Problem Statement

This is a test of solving a simple one-dimensional thermal diffusion problem with Dirichlet BCs. The test problem is shown schematically in Figure. Although the problem is one-dimensional we solve the problem in a three-dimensional setting. Once the diffusion problem has been solved numerically the temperature result is postprocessed to obtain a comparison with the analytical result and the distribution of diffusive heat flux. This test input also demonstrates the use tabular function and Encore function material property specification in Aria.

7.5.2 Input File

7.6 tv Regression Test

7.6.1 Problem Statement

This is a test of solving a simple one-dimensional thermal diffusion problem with Dirichlet BCs. The test problem is shown schematically in Figure. Although the problem is one-dimensional we solve the problem in a three-dimensional setting. Once the diffusion problem has been solved numerically the temperature result is postprocessed to obtain a comparison with the analytical result and the distribution of diffusive heat flux. This test input also demonstrates the use tabular function and Encore function material property specification in Aria.

7.6.2 Input File

References

[1] Gerald W. Wellman. Mapvar: a computer program to transfer solution data between finite element meshes. SAND 1999-0466, Sandia National Laboratories, Albuquerque, NM, USA, March 1999. 1.1

[2] The SNTools Project. SNTools SourceForge Project. Online. 2.3

Index

A

Abort If Field Not Defined On Copy Transfer Send Or Receive Object, 79, 80
Abscissa, 34, 35
Abscissa Offset, 34, 35
Abscissa Scale, 34, 36
Adapt, 48, 49, 52, 53, 55, 56, 59, 62
Adaptivity, 48, 52, 62
Advance, 52, 53, 55, 56, 59, 62, 63, 65, 69
Alias, 26, 27
All Fields, 79, 80
At Discontinuity Evaluate To, 34, 36

B

barOneWayCoupleDifferentMesh
  Statement, 94
barOneWayCoupleFromDifferentMesh
  Input, 95
barOneWayCoupleFromFile
  Input, 94
  Statement, 93
barOneWayCoupleTransfer
  Input, 95
  Statement, 95

C

Column Titles, 34, 36
Component Separator Character, 26, 27
Compute Indicator On, 48, 49, 52, 53, 55, 56, 59, 60, 62, 63
Converged When, 66, 67, 70
Coordinate System, 26, 28
Copy, 79, 81
Create, 26, 28
Create Element Field, 88, 89
Create Nodal Field, 88, 89

D

Data File, 34, 36
Database Name, 26, 28
Database Type, 26, 28
Debug, 34, 37
Decomposition Method, 26, 29
Definition For Function, 34
Differentiate Expression, 35, 37
Distance Function Is Closest Receive Node To Send Centroid, 79, 81

E

estack
  Input, 96
  Statement, 96
Evaluate Expression, 35, 37
Evaluate From, 35, 39
Event, 48, 49, 52, 53, 55, 57, 59, 60, 62, 63, 65, 66, 70
Exclude Ghosted, 79, 81
Execute Postprocessor Group, 48, 50, 52, 54, 55, 57, 59, 60, 62, 63
Expression Variable:, 35, 39
Expressions, 35

F

Finite Element Model, 26
Fixed Time, 88, 89
From, 80, 81

G

gapClosure
  Input, 96
  Statement, 95
Gauss Point Integration Order, 80, 82
Global Constants, 32
Global Id Mapping Backward Compatibility, 26, 29
Gravity Vector, 32, 33

I

Ideal Gas Constant, 32, 33
Include All Blocks, 30, 31
Incremental Number Of Steps, 66, 67
Indicatemarkadapt, 48, 50, 52, 54, 55, 57, 59, 60, 62, 64
Initial Deltat, 66, 67, 70
Initialize, 47, 65
Input_Output Region, 88
Interpolate, 80, 82
Interpolation Function, 80, 82
Involve, 52, 54, 56, 57, 59, 61, 62, 64–66, 70

K

K-E Turbulence Model Parameter, 32, 33
K-W Turbulence Model Parameter, 32, 33

L

Les Turbulence Model Parameter, 32, 34
Local Coordinate System, 30, 31

M

Mark, 48, 50, 52, 54, 56, 58, 59, 61, 62, 64
Markadapt, 48, 50, 52, 55, 56, 58, 59, 61, 62, 64
Material, 31
Material =, 31

N

Nodes Outside Region, 80, 82
Nonlinear, 52, 55, 62
Number Of Adaptivity Steps, 66, 68, 71
Number Of Steps, 66, 68, 71

O

Offset Time, 88, 89
Omit Block, 26, 29
Omit Volume, 26, 29
Ordinate, 35, 40
Ordinate Offset, 35, 40
Ordinate Scale, 35, 40
Output, 48, 51, 52, 55, 56, 58, 59, 61, 62, 65, 71

P

Parameters For, 47, 66
Parameters For Aria Region, 67
Parameters For Block, 27, 30
Parameters For Phase, 27

Parameters For Surface, 27
Periodicity Time, 88, 90
Phase, 31, 32

R

Receive Blocks, 80
Reinitialize Transient, 67, 68, 71
Remove Block, 31, 32
Restart Time, 42
Results Output, 88

S

Scale By, 35, 40
Search Coordinate Field, 80, 83
Search Geometric Tolerance, 80, 83
Search Surface Gap Tolerance, 80, 83
Search Type, 80, 84
Select One Receiver For Each Send Object, 80, 84
Select One Unique Receiver For Each Send Object, 80, 84
Send, 80, 85
Send Block, 80, 85
Send Blocks, 80
Send Field, 80, 85
Sequential, 48, 62
Simulation Max Global Iterations, 48, 51, 71
Simulation Start Time, 48, 51, 72
Simulation Termination Time, 48, 51, 72
Solution Control Description, 47
Start Time, 67, 68, 72, 88, 90
Stefan Boltzmann Constant, 32, 34
Subcycle, 52, 56, 59
System, 48

T

Termination Time, 67, 68, 72
Time Scale Factor, 26, 30
Time Step Quantum, 67, 69, 72
Time Step Style, 67, 69, 73
Total Change In Time, 67, 69, 73
Transfer, 48, 51, 52, 55, 56, 58, 59, 61, 62, 65, 66, 73, 79
Transient, 48, 52
Turbulence Model, 32, 34
tv
    Input, 96
    Statement, 96
Type, 35, 40

U

Use Finite Element Model, 88, 91
Use Generic Names, 27, 30
Use Initialize, 48, 52, 74
Use Material, 27, 30
Use System, 47, 48, 74

V

Values, 35, 41, 42

X

X Offset, 35, 41
X Scale, 35, 41

Y

Y Offset, 35, 41
Y Scale, 35, 41

DISTRIBUTION:

1 MS 0899 Technical Library, 9536 (electronic copy)

Sandia National Laboratories