DAKOTA,  
A Multilevel Parallel Object-Oriented Framework for Design Optimization,  
Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis  
Version 3.0 Developers Manual  

Michael S. Eldred, Anthony A. Giunta, Bart G. van Bloemen Waanders,  
Steven F. Wojtkiewicz, Jr., William E. Hart, and Mario P. Alleva
DAKOTA, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis

Version 3.0 Developers Manual

Michael S. Eldred, Anthony A. Giunta, and Bart G. van Bloemen Waanders
Optimization and Uncertainty Estimation Department

Steven F. Wojtkiewicz, Jr.
Structural Dynamics Research Department

William E. Hart
Optimization and Uncertainty Estimation Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-0847

Mario P. Alleva
Compaq Federal
Albuquerque, New Mexico 87109-3432

Abstract

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible and extensible interface between simulation codes and iterative analysis methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods; uncertainty quantification with sampling, analytic reliability, and stochastic finite element methods; parameter estimation with nonlinear least squares methods; and sensitivity analysis with design of experiments and parameter study methods. These capabilities may be used on their own or as components within advanced strategies such as surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment for design and performance analysis of computational models on high performance computers.

This report serves as a developers manual for the DAKOTA software and describes the DAKOTA class hierarchies and their interrelationships. It derives directly from annotation of the actual source code and provides detailed class documentation, including all member functions and attributes.
## Contents

1 DAKOTA Developers Manual ........................................... 9
   1.1 Introduction ................................................. 9
   1.2 Overview of DAKOTA ........................................ 9
   1.3 Related Components ...................................... 13
   1.4 Additional Resources .................................... 15

2 DAKOTA Hierarchical Index .......................................... 17
   2.1 DAKOTA Class Hierarchy .................................. 17

3 DAKOTA Compound Index ........................................... 21
   3.1 DAKOTA Compound List .................................... 21

4 DAKOTA File Index .................................................. 25
   4.1 DAKOTA File List ........................................... 25

5 DAKOTA Page Index .................................................. 27
   5.1 DAKOTA Related Pages .................................... 27

6 DAKOTA Class Documentation ....................................... 29
   6.1 AllMergeredVarConstraints Class Reference .......... 29
   6.2 AllMergeredVariables Class Reference .................. 32
   6.3 AllVarConstraints Class Reference ..................... 36
   6.4 AllVariables Class Reference ............................ 39
   6.5 AnalysisCode Class Reference ............................ 43
   6.6 ANNSurf Class Reference ................................ 46
   6.7 ApplicationInterface Class Reference ................. 48
   6.8 ApproximationInterface Class Reference ............. 58
   6.9 BaseConstructor Struct Reference ..................... 61
   6.10 BranchBndStrategy Class Reference .................. 62
   6.11 CommandLineHandler Class Reference ................ 64
6.12 CommandShell Class Reference ........................................... 65
6.13 ConcurrentStrategy Class Reference ........................................ 67
6.14 CONMINOptimizer Class Reference ......................................... 69
6.15 CtrlRegexp Class Reference .................................................. 76
6.16 DACEIterator Class Reference .............................................. 78
6.17 DakotaApproximation Class Reference ...................................... 82
6.18 DakotaArray Class Template Reference .................................... 87
6.19 DakotaBaseVector Class Template Reference ................................ 90
6.20 DakotaBiStream Class Reference ........................................... 93
6.21 DakotaBoStream Class Reference .......................................... 96
6.22 DakotaGraphics Class Reference ........................................... 99
6.23 DakotaInterface Class Reference ........................................... 101
6.24 DakotaIterator Class Reference ........................................... 107
6.25 DakotaList Class Template Reference ..................................... 113
6.26 DakotaMatrix Class Template Reference ................................... 117
6.27 DakotaModel Class Reference ............................................. 119
6.28 DakotaNonD Class Reference ............................................... 130
6.29 DakotaOptimizer Class Reference ......................................... 133
6.30 DakotaResponse Class Reference .......................................... 138
6.31 DakotaResponseRep Class Reference ...................................... 142
6.32 DakotaStrategy Class Reference ........................................... 147
6.33 DakotaString Class Reference ............................................ 152
6.34 DakotaVarConstraints Class Reference .................................. 155
6.35 DakotaVariables Class Reference ......................................... 160
6.36 DakotaVector Class Template Reference .................................. 166
6.37 DataInterface Class Reference ............................................ 170
6.38 DataMethod Class Reference .............................................. 174
6.39 DataResponses Class Reference ........................................... 181
6.40 DataVariables Class Reference ........................................... 184
6.41 DirectFnApplicInterface Class Reference ................................ 190
6.42 DOTOptimizer Class Reference ............................................ 194
6.43 ErrorTable Struct Reference ............................................... 198
6.44 ForkAnalysisCode Class Reference ....................................... 199
6.45 ForkApplicInterface Class Reference ..................................... 201
6.46 FunctionCompare Class Template Reference ............................. 204
6.47 FundamentalVarConstraints Class Reference ............................ 205
<table>
<thead>
<tr>
<th>Section</th>
<th>Class Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.48</td>
<td>FundamentalVariables Class Reference</td>
<td>208</td>
</tr>
<tr>
<td>6.49</td>
<td>GetLongOpt Class Reference</td>
<td>212</td>
</tr>
<tr>
<td>6.50</td>
<td>HermiteSurf Class Reference</td>
<td>216</td>
</tr>
<tr>
<td>6.51</td>
<td>HierLayeredModel Class Reference</td>
<td>218</td>
</tr>
<tr>
<td>6.52</td>
<td>KrigApprox Class Reference</td>
<td>222</td>
</tr>
<tr>
<td>6.53</td>
<td>KrisingSurf Class Reference</td>
<td>230</td>
</tr>
<tr>
<td>6.54</td>
<td>LayeredModel Class Reference</td>
<td>232</td>
</tr>
<tr>
<td>6.55</td>
<td>MARSurf Class Reference</td>
<td>236</td>
</tr>
<tr>
<td>6.56</td>
<td>MergedVarConstraints Class Reference</td>
<td>238</td>
</tr>
<tr>
<td>6.57</td>
<td>MergedVariables Class Reference</td>
<td>241</td>
</tr>
<tr>
<td>6.58</td>
<td>MultilevelOptStrategy Class Reference</td>
<td>245</td>
</tr>
<tr>
<td>6.59</td>
<td>NestedModel Class Reference</td>
<td>248</td>
</tr>
<tr>
<td>6.60</td>
<td>NoDBBaseConstructor Struct Reference</td>
<td>254</td>
</tr>
<tr>
<td>6.61</td>
<td>NonDAdvMeanValue Class Reference</td>
<td>255</td>
</tr>
<tr>
<td>6.62</td>
<td>NonDOptStrategy Class Reference</td>
<td>262</td>
</tr>
<tr>
<td>6.63</td>
<td>NonDPCE Class Reference</td>
<td>264</td>
</tr>
<tr>
<td>6.64</td>
<td>NonDProbability Class Reference</td>
<td>267</td>
</tr>
<tr>
<td>6.65</td>
<td>NonDSampling Class Reference</td>
<td>271</td>
</tr>
<tr>
<td>6.66</td>
<td>NPSOLOptimizer Class Reference</td>
<td>275</td>
</tr>
<tr>
<td>6.67</td>
<td>ParallelLibrary Class Reference</td>
<td>279</td>
</tr>
<tr>
<td>6.68</td>
<td>ParamResponsePair Class Reference</td>
<td>288</td>
</tr>
<tr>
<td>6.69</td>
<td>ParamStudy Class Reference</td>
<td>291</td>
</tr>
<tr>
<td>6.70</td>
<td>ProblemDescDB Class Reference</td>
<td>295</td>
</tr>
<tr>
<td>6.71</td>
<td>RespSurf Class Reference</td>
<td>301</td>
</tr>
<tr>
<td>6.72</td>
<td>SGOPTApplication Class Reference</td>
<td>303</td>
</tr>
<tr>
<td>6.73</td>
<td>SGOPTOptimizer Class Reference</td>
<td>306</td>
</tr>
<tr>
<td>6.74</td>
<td>SingleMethodStrategy Class Reference</td>
<td>309</td>
</tr>
<tr>
<td>6.75</td>
<td>SingleModel Class Reference</td>
<td>310</td>
</tr>
<tr>
<td>6.76</td>
<td>SNLLOptimizer Class Reference</td>
<td>312</td>
</tr>
<tr>
<td>6.77</td>
<td>SortCompare Class Template Reference</td>
<td>318</td>
</tr>
<tr>
<td>6.78</td>
<td>SurrBasedOptStrategy Class Reference</td>
<td>319</td>
</tr>
<tr>
<td>6.79</td>
<td>SurrLayeredModel Class Reference</td>
<td>323</td>
</tr>
<tr>
<td>6.80</td>
<td>SurrogateDataPoint Class Reference</td>
<td>328</td>
</tr>
<tr>
<td>6.81</td>
<td>SysCallAnalysisCode Class Reference</td>
<td>330</td>
</tr>
<tr>
<td>6.82</td>
<td>SysCallApplcInterface Class Reference</td>
<td>332</td>
</tr>
<tr>
<td>6.83</td>
<td>TaylorSurf Class Reference</td>
<td>334</td>
</tr>
</tbody>
</table>
CONTENTS

6.84 VariablesUtil Class Reference ........................................ 336

7 DAKOTA File Documentation ........................................... 339
  7.1 keywordtable.C File Reference ..................................... 339
  7.2 main.C File Reference .................................................. 340
  7.3 restart_util.C File Reference ........................................ 342

8 Recommended Practices for DAKOTA Development ................... 345
  8.1 Introduction ........................................................... 345
  8.2 Style Guidelines ........................................................ 345
  8.3 File Naming Conventions ............................................. 347
  8.4 Class Documentation Conventions .................................. 348

9 Instructions for Modifying DAKOTA's Input Specification .......... 349
  9.1 Modify dakota.input.spec ............................................. 349
  9.2 Rebuild IDR ............................................................ 350
  9.3 Update keywordtable.C in $DAKOTA/src .......................... 350
  9.4 Update ProblemDescDB.C in $DAKOTA/src ......................... 350
  9.5 Update Corresponding Data Classes ................................ 353
  9.6 Use get_<data_type>() Functions .................................. 354
  9.7 Update the Documentation .......................................... 354

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
Chapter 1

DAKOTA Developers Manual

Author:
   Michael S. Eldred, Anthony A. Giunta, Bart G. van Bloemen Waanders, Steven F. Wojtkiewicz, Jr.,
   William E. Hart, Mario P. Alleva

1.1 Introduction

The DAKOTA (Design Analysis Kit for Optimization and Terascale Applications) toolkit provides a flexible, extensible interface between analysis codes and iteration methods. DAKOTA contains algorithms for optimization with gradient and nongradient-based methods, uncertainty quantification with sampling, analytic reliability, and stochastic finite element methods, parameter estimation with nonlinear least squares methods, and sensitivity/primary effects analysis with design of experiments and parameter study capabilities. These capabilities may be used on their own or as components within advanced strategies for surrogate-based optimization, mixed integer nonlinear programming, or optimization under uncertainty. By employing object-oriented design to implement abstractions of the key components required for iterative systems analyses, the DAKOTA toolkit provides a flexible and extensible problem-solving environment as well as a platform for rapid prototyping of advanced solution methodologies.

The Developers Manual focuses on documentation of the class structures used by the DAKOTA system. It derives directly from annotation of the actual source code. For information on input command syntax, refer to the Reference Manual, and for a tour of DAKOTA features and capabilities, refer to the Users Manual.

1.2 Overview of DAKOTA

In the DAKOTA system, the strategy creates and manages iterators and models. In the simplest case, the strategy creates a single iterator and a single model and executes the iterator on the model to perform a single study. In a more advanced case, a hybrid optimization strategy might manage a global optimizer operating on a low-fidelity model in coordination with a local optimizer operating on a high-fidelity model. And on the high end, a surrogate-based optimization under uncertainty strategy would employ an uncertainty quantification iterator nested within an optimization iterator and would employ truth models layered
within surrogate models. Thus, iterators and models provide both stand-alone capabilities as well as building blocks for more sophisticated studies.

A model contains a set of variables, an interface, and a set of responses, and the iterator operates on the model to map the variables into responses using the interface. Each of these components is a flexible abstraction with a broad array of specializations for supporting a variety of studies. In a DAKOTA input file, the user specifies these components through strategy, method, variables, interface, and responses keyword specifications.

The extensive use of class hierarchies provides a clear direction for extensibility in DAKOTA components. In each of the various class hierarchies, adding a new capability typically involves deriving a new class and providing a small number of virtual function redefinitions. These redefinitions define the coding portions specific to the new derived class, with the common portions already defined at the base class. Thus, with a small amount of new code, the existing facilities can be extended, reused, and leveraged for new purposes.

The software components are presented in the following sections using a top-down order.

1.2.1 Strategies

Class hierarchy: DakotaStrategy.

Strategies provide a control layer for creation and management of iterators and models. Specific strategies include:

- **SingleMethodStrategy**: the simplest strategy. A single iterator is run on a single model to perform a single study.
- **MultilevelOptStrategy**: hybrid optimization using a succession of iterators employing a succession of models of varying fidelity. The best results obtained are passed from one iterator to the next.
- **SurrBasedOptStrategy**: surrogate-based optimization. Employs a single iterator with a LayeredModel (either data fit or hierarchical). A sequence of approximate optimizations is performed, each of which involves build, optimize, and verify steps.
- **NonDOptStrategy**: optimization under uncertainty (OUU). Employs a single iterator with a NestedModel. In OUU approaches involving surrogates, NestedModels and LayeredModels can be chained together in a variety of ways using recursion in sub-models.
- **BranchBndStrategy**: mixed integer nonlinear programming using the PICO library for parallel branch and bound. Employs a single iterator with a single model, but runs multiple instances of the iterator concurrently for different variable bounds within the model.
- **ConcurrentStrategy**: two similar algorithms are available: (1) multi-start iteration from several different starting points, and (2) pareto set optimization for several different multiobjective weightings. Employs a single iterator with a single model, but runs multiple instances of the iterator concurrently for different settings within the model.

1.2.2 Iterators

Class hierarchy: DakotaIterator.

The iterator hierarchy contains a variety of iterative algorithms for optimization, uncertainty quantification, nonlinear least squares, design of experiments, and parameter studies.

- **Optimization**: DakotaOptimizer inherits from DakotaIterator and provides a base class for DOTOptimizer, CONMINOptimizer, NPSOLOptimizer, SNLLOptimizer, and SGOPTOptimizer.
1.2 Overview of DAKOTA

- Uncertainty quantification: DakotaNonD inherits from DakotaIterator and provides a base class for NonDProbability, NonDAdvMeanValue, and NonDPCE.
- Parameter estimation: a Gauss-Newton least squares solver is provided as part of SNLLOptimizer.
- Design of experiments: DACEIterator inherits directly from DakotaIterator. NonDProbability from the uncertainty quantification branch can also be used for this purpose.
- Parameter studies: ParamStudy inherits directly from DakotaIterator.

1.2.3 Models

Class hierarchy: DakotaModel.

The model classes are responsible for mapping variables into responses when an iterator makes a function evaluation request. There are several types of models, some supporting sub-iterators and sub-models for enabling layered and nested relationships. When sub-models are used, they may be of arbitrary type so that a variety of recursions are supported.

- **SingleModel**: variables are mapped into responses using a single DakotaInterface object. No sub-iterators or sub-models are used.
- **LayeredModel**: variables are mapped into responses using an approximation. The approximation is built and/or corrected using data from a sub-model (the truth model) and the data may be obtained using a sub-iterator (a design of experiments iterator). LayeredModel has two derived classes: SurrLayeredModel for data fit surrogates and HierLayeredModel for hierarchical models of varying fidelity. The relationship of the sub-iterators and sub-models is considered to be ”layered” since they are not used in the response evaluation for the top level model, but rather used in separate build and correction steps.
- **NestedModel**: variables are mapped into responses using a combination of an optional DakotaInterface and a sub-iterator/sub-model pair. The relationship of the sub-iterators and sub-models is considered to be ”nested” since they are executed on every evaluation of the top level model as part of the response computation.

1.2.4 Variables

Class hierarchy: DakotaVariables.

The DakotaVariables class hierarchy manages design, uncertain, and state variable types for continuous and discrete domain types. This hierarchy is specialized according to various views of the data.

- **FundamentalVariables**: variable and domain type distinctions are retained, i.e. separate arrays for design, uncertain, and state variables types and for continuous and discrete domains.
- **AllVariables**: variable types are combined and domain type distinction is retained, i.e. design, uncertain, and state variable types combined into a single continuous variables array and a single discrete variables array.
- **MergedVariables**: variable type distinction is retained and domain types are combined, i.e. continuous and discrete variables merged into continuous arrays for design, uncertain, and state variable types.
- **AllMergedVariables**: variable and domain types are combined, i.e. design, uncertain, and state variables types combined (all) and continuous and discrete domain types combined (merged). The result is a single array of continuous variables.

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch (©) 1997-2001
The DakotaVarConstraints hierarchy contains the same specializations for managing linear and bound constraints on the variables (see FundamentalVarConstraints, AllVarConstraints, MergedVarConstraints, AllMergedVarConstraints).

1.2.5 Interfaces

Class hierarchy: DakotaInterface.

Interfaces provide access to simulation codes or, conversely, approximations based on simulation code data. In the simulation case, an ApplicationInterface is used. ApplicationInterface is specialized according to the simulation invocation mechanism, for which the following nonintrusive approaches are supported:

- **SysCallApplicInterface**: the simulation is invoked using a system call (the C function system()). Asynchronous invocation utilizes a background system call. Utilizes the SysCallAnalysisCode class to define syntax for input filter, analysis code, output filter, or combined spawning, which in turn utilize the CommandShell overloaded operator definitions.
- **ForkApplicInterface**: the simulation is invoked using a fork (the fork/exec/wait family of functions). Asynchronous invocation utilizes a nonblocking fork. Utilizes the ForkAnalysisCode class for lower level fork operations.
- **XMLApplicInterface**: the simulation is invoked using an XML packet passed across a socket communication. This capability is experimental and still under development.

and the following semi-intrusive approach

- **DirectFnApplicInterface**: the simulation is linked into the DAKOTA executable and is invoked using a procedure call. Asynchronous invocation utilizes a nonblocking thread (capability not yet available).

are supported. Scheduling of jobs for asynchronous local, message passing, and hybrid parallelism approaches is performed in the ApplicationInterface class, with job initiation and job capture specifics implemented in the derived classes.

In the approximation case, global, multipoint, or local approximations to simulation code response data can be built and used as surrogates for the actual, expensive simulation. The interface class providing this capability is

- **ApproximationInterface**: builds an approximation using data from a truth model and then employs the approximation for mapping variables to responses. This class contains an array of DakotaApproximation objects, one per response function, which allows mixing of approximation types (using the DakotaApproximation derived classes: ANNSurf, KrigingSurf, MARSSurf, RespSurf, HermiteSurf, and TaylorSurf).

1.2.6 Responses

Class: DakotaResponse.

Responses provide an abstract data representation of response functions, their gradients, and their Hessians which can be interpreted as an objective function and constraints (optimization data set), residual functions
(least squares data set), or generic response functions (uncertainty and parameter study data sets). This class is not currently part of a class hierarchy, since the abstraction has been sufficiently general and has not required specialization.

1.3 Related Components

A variety of services are provided in DAKOTA for parallel computing, failure capturing, restart, graphics, etc. In addition, the execution of function evaluations is a core component of DAKOTA involving several class hierarchies. An overview of the classes and member functions involved in performing function evaluations is included below.

1.3.1 Services

- Multilevel parallel computing: DAKOTA supports up to 4 nested levels of parallelism: a strategy can manage concurrent iterators, each of which manages concurrent function evaluations, each of which contains concurrent analyses executing on multiple processors. Partitioning of these levels with MPI communicators is managed in ParallelLibrary and scheduling routines for the levels are part of ConcurrentStrategy, ApplicationInterface, and ForkApplicationInterface.

- Parsing: DAKOTA employs the Input Deck Reader (IDR) parser to retrieve information from user input files. Parsing options are processed in CommandLineHandler and parsing occurs in main.C. IDR populates data within the ProblemDescDB support class, which maintains the strategy specification and lists of DataMethod, DataVariables, DataInterface, and DataResponses specifications. Instructions for modifying the parsing subsystem are described in Instructions for Modifying DAKOTA's Input Specification.

- Failure capturing: Simulation failures can be trapped and managed using exception handling in ApplicationInterface and its derived classes.

- Restart: DAKOTA maintains a record of all function evaluations both in memory (for capturing any duplication) and on the file system (for restarting runs). Restart options are processed in CommandLineHandler, restart file management occurs in main.C, and restart file insertions occur in ApplicationInterface. The dakota_restart_util executable, built from restart_util.C, provides a variety of services for interrogating, converting, repairing, concatenating, and post-processing restart files.

- Memory management: DAKOTA employs the techniques of reference counting and representation sharing through the use of letter-envelope and handle-body idioms (Coplien, "Advanced C++"). The former idiom provides for memory efficiency and enhanced polymorphism in the following class hierarchies: DakotaStrategy, DakotaIterator, DakotaModel, DakotaVariables, DakotaVarConstraints, DakotaInterface, and DakotaApproximation. The latter idiom provides for memory efficiency in heavily used classes which do not involve a class hierarchy. Currently, only the DakotaResponse class uses this idiom.

- Graphics: DAKOTA provides 2D graphics using Motif widgets and 3D graphics from the PLPLOT package. Graphics data can also be catalogued in a tabular data file for post-processing with 3rd party tools such as Matlab, Tecplot, etc. All of these capabilities are encapsulated within the DakotaGraphics class.
1.3.2 Performing function evaluations

Performing function evaluations is one of the most critical functions of the DAKOTA software. It can also be one of the most complicated, as a variety of scheduling approaches and parallelism levels are supported. This complexity manifests itself in the code through a series of cascaded member functions, from the top level model evaluation functions, through various scheduling routines, to the low level details of performing a system call, fork, or direct function invocation. This section provides an overview of the primary classes and member functions involved.

For a synchronous (i.e., blocking) mapping of parameters to responses, an iterator invokes DakotaModel::compute_response() to perform a function evaluation. This function is all that is seen from the iterator level, as underlying complexities are isolated. The binding of this top level function with lower level functions is as follows:

- DakotaModel::compute_response() utilizes DakotaModel::derived_compute_response() for portions of the response computation specific to derived model classes.
- DakotaModel::derived_compute_response() directly or indirectly invokes DakotaInterface::map().
- DakotaInterface::map() utilizes ApplicationInterface::derived_map() for portions of the mapping specific to derived application interface classes.

For an asynchronous (i.e., nonblocking) mapping of parameters to responses, an iterator invokes DakotaModel::asynch_compute_response() multiple times to queue asynchronous jobs and then invokes either DakotaModel::synchronize() or DakotaModel::synchronize_nowait() to schedule the queued jobs in blocking or nonblocking fashion. Again, these functions are all that is seen from the iterator level, as underlying complexities are isolated. The binding of these top level functions with lower level functions is as follows:

- DakotaModel::asynch_compute_response() utilizes DakotaModel::derived_asynch_compute_response() for portions of the response computation specific to derived model classes.
- This derived model class function directly or indirectly invokes DakotaInterface::map() in asynchronous mode, which adds the job to a scheduling queue.
- DakotaModel::synchronize() or DakotaModel::synchronize_nowait() utilize DakotaModel::derived_synchronize() or DakotaModel::derived_synchronize_nowait() for portions of the scheduling process specific to derived model classes.
- These derived model class functions directly or indirectly invoke DakotaInterface::synch() or DakotaInterface::synch_nowait().
- For application interfaces, these interface synchronization functions are responsible for performing evaluation scheduling in one of the following modes:
  - asynchronous local mode (using ApplicationInterface::asynchronous_local_evaluations() or ApplicationInterface::asynchronous_local_evaluations_nowait())
  - message passing mode (using ApplicationInterface::self_schedule_evaluations() or ApplicationInterface::static_schedule_evaluations() on the iterator master and ApplicationInterface::serve_evaluations_synch() or ApplicationInterface::serve_evaluations_peer() on the servers)
  - hybrid mode (using ApplicationInterface::self_schedule_evaluations() or ApplicationInterface::static_schedule_evaluations() on the iterator master and ApplicationInterface::serve_evaluations_asynch() on the servers)
- These scheduling functions utilize ApplicationInterface::derived_map() and ApplicationInterface::derived_map_asynch() for portions of asynchronous job launching specific to derived application interface classes, as well as ApplicationInterface::derived_synch() and ApplicationInterface::derived_synch_nowait() for portions of job capturing specific to derived application interface classes.
This covers the parallelism level of concurrent function evaluations serving an iterator. For the parallelism level of concurrent analyses serving a function evaluation, similar schedulers are involved (ForkApplicInterface::synchronous_local_analyses(), ForkApplicInterface::asynchronous_local_analyses(), ApplicationInterface::self_schedule_analyses(), ApplicationInterface::serve_analyses_synch(), ForkApplicInterface::serve_analyses_asynch()) to support synchronous local, asynchronous local, message passing, and hybrid modes. Not all of the schedulers are elevated to the ApplicationInterface level since the system call and direct function interfaces do not yet support nonblocking local analyses (and therefore support synchronous local and message passing modes, but not asynchronous local or hybrid modes). Fork interfaces, however, support all modes of analysis parallelism.

1.4 Additional Resources

Additional development resources include:

- Recommended Practices for DAKOTA Development
- Instructions for Modifying DAKOTA's Input Specification
Chapter 2

DAKOTA Hierarchical Index

2.1 DAKOTA Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

<table>
<thead>
<tr>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnalysisCode</td>
<td>43</td>
</tr>
<tr>
<td>ForkAnalysisCode</td>
<td>199</td>
</tr>
<tr>
<td>SysCallAnalysisCode</td>
<td>330</td>
</tr>
<tr>
<td>BaseConstructor</td>
<td>61</td>
</tr>
<tr>
<td>CommandShell</td>
<td>65</td>
</tr>
<tr>
<td>CtelRegexp</td>
<td>76</td>
</tr>
<tr>
<td>DakotaApproximation</td>
<td>82</td>
</tr>
<tr>
<td>ANNSurf</td>
<td>46</td>
</tr>
<tr>
<td>HermiteSurf</td>
<td>216</td>
</tr>
<tr>
<td>KrigingSurf</td>
<td>230</td>
</tr>
<tr>
<td>MARSSurf</td>
<td>236</td>
</tr>
<tr>
<td>RespSurf</td>
<td>301</td>
</tr>
<tr>
<td>TaylorSurf</td>
<td>334</td>
</tr>
<tr>
<td>DakotaArray&lt;T&gt;</td>
<td>87</td>
</tr>
<tr>
<td>DakotaBaseVector&lt;T&gt;</td>
<td>90</td>
</tr>
<tr>
<td>DakotaVector&lt;T&gt;</td>
<td>166</td>
</tr>
<tr>
<td>DakotaBaseVector&lt;DakotaBaseVector&lt;T&gt;&gt;</td>
<td>90</td>
</tr>
<tr>
<td>DakotaMatrix&lt;T&gt;</td>
<td>117</td>
</tr>
<tr>
<td>DakotaBiStream</td>
<td>93</td>
</tr>
<tr>
<td>DakotaBoStream</td>
<td>96</td>
</tr>
<tr>
<td>DakotaGraphics</td>
<td>99</td>
</tr>
<tr>
<td>DakotaInterface</td>
<td>101</td>
</tr>
<tr>
<td>ApplicationInterface</td>
<td>48</td>
</tr>
<tr>
<td>DirectFnApplicInterface</td>
<td>190</td>
</tr>
<tr>
<td>ForkApplicInterface</td>
<td>201</td>
</tr>
<tr>
<td>SysCallApplicInterface</td>
<td>332</td>
</tr>
<tr>
<td>ApproximationInterface</td>
<td>58</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>107</td>
</tr>
<tr>
<td>DACEIterator</td>
<td>78</td>
</tr>
<tr>
<td>DakotaNonD</td>
<td>130</td>
</tr>
<tr>
<td>NonDAdvMeanValue</td>
<td>255</td>
</tr>
<tr>
<td>Class/Function</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>NonDPCE</td>
<td>264</td>
</tr>
<tr>
<td>NonDProbability</td>
<td>267</td>
</tr>
<tr>
<td>NonDSampling</td>
<td>271</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
<td>133</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>275</td>
</tr>
<tr>
<td>SGOPTOptimizer</td>
<td>306</td>
</tr>
<tr>
<td>SNLLOptimizer</td>
<td>312</td>
</tr>
<tr>
<td>ParamStudy</td>
<td>291</td>
</tr>
<tr>
<td>DakotaList&lt;T&gt;</td>
<td>113</td>
</tr>
<tr>
<td>DakotaModel</td>
<td>119</td>
</tr>
<tr>
<td>LayeredModel</td>
<td>232</td>
</tr>
<tr>
<td>HierLayeredModel</td>
<td>218</td>
</tr>
<tr>
<td>SurrLayeredModel</td>
<td>323</td>
</tr>
<tr>
<td>NestedModel</td>
<td>248</td>
</tr>
<tr>
<td>SingleModel</td>
<td>310</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>138</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>142</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>147</td>
</tr>
<tr>
<td>BranchBndStrategy</td>
<td>62</td>
</tr>
<tr>
<td>ConcurrentStrategy</td>
<td>67</td>
</tr>
<tr>
<td>MultilevelOptStrategy</td>
<td>245</td>
</tr>
<tr>
<td>NonDOptStrategy</td>
<td>262</td>
</tr>
<tr>
<td>SingleMethodStrategy</td>
<td>309</td>
</tr>
<tr>
<td>SurrBasedOptStrategy</td>
<td>319</td>
</tr>
<tr>
<td>DakotaString</td>
<td>152</td>
</tr>
<tr>
<td>DakotaVarConstraints</td>
<td>155</td>
</tr>
<tr>
<td>AllMergedVarConstraints</td>
<td>29</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>FundamentalVarConstraints</td>
<td>205</td>
</tr>
<tr>
<td>MergedVarConstraints</td>
<td>238</td>
</tr>
<tr>
<td>DakotaVariables</td>
<td>160</td>
</tr>
<tr>
<td>AllMergedVariables</td>
<td>32</td>
</tr>
<tr>
<td>AllVariables</td>
<td>39</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>208</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>241</td>
</tr>
<tr>
<td>DataInterface</td>
<td>170</td>
</tr>
<tr>
<td>DataMethod</td>
<td>174</td>
</tr>
<tr>
<td>DataResponses</td>
<td>181</td>
</tr>
<tr>
<td>DataVariables</td>
<td>184</td>
</tr>
<tr>
<td>ErrorTable</td>
<td>198</td>
</tr>
<tr>
<td>FunctionCompare&lt;T&gt;</td>
<td>204</td>
</tr>
<tr>
<td>GetLongOpt</td>
<td>212</td>
</tr>
<tr>
<td>CommandLineHandler</td>
<td>64</td>
</tr>
<tr>
<td>KrigApprox</td>
<td>222</td>
</tr>
<tr>
<td>NoDBBaseConstructor</td>
<td>254</td>
</tr>
<tr>
<td>ParallelLibrary</td>
<td>279</td>
</tr>
<tr>
<td>ParamResponsePair</td>
<td>288</td>
</tr>
<tr>
<td>ProblemDescDB</td>
<td>295</td>
</tr>
<tr>
<td>SGOPTApplication</td>
<td>303</td>
</tr>
<tr>
<td>SortCompare&lt;T&gt;</td>
<td>318</td>
</tr>
<tr>
<td>Class Name</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>SurrogateDataPoint</td>
<td>328</td>
</tr>
<tr>
<td>VariablesUtil</td>
<td>336</td>
</tr>
<tr>
<td>AllMergerdVarConstraints</td>
<td>29</td>
</tr>
<tr>
<td>AllMergerdVariables</td>
<td>32</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>AllVariables</td>
<td>39</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>208</td>
</tr>
<tr>
<td>MergedVarConstraints</td>
<td>238</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>241</td>
</tr>
</tbody>
</table>
Chapter 3

DAKOTA Compound Index

3.1 DAKOTA Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

- **AllMergerConstraints** (Derived class within the DakotaVarConstraints hierarchy which combines the all and merged data views) ................................................................. 29
- **AllMergerVariables** (Derived class within the DakotaVariables hierarchy which combines the all and merged data views) ................................................................. 32
- **AllVarConstraints** (Derived class within the DakotaVarConstraints hierarchy which employs the all data view) ................................................................. 36
- **AllVariables** (Derived class within the DakotaVariables hierarchy which employs the all data view) ................................................................. 39
- **AnalysisCode** (Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations) ................................................................. 43
- **ANNSurf** (Derived approximation class for artificial neural networks) ................................................................. 46
- **ApplicationInterface** (Derived class within the interface class hierarchy for supporting interfaces to simulation codes) ................................................................. 48
- **ApproximationInterface** (Derived class within the interface class hierarchy for supporting approximations to simulation-based results) ................................................................. 58
- **BaseConstructor** (Dummy struct for overloading letter-envelope constructors) ................................................................. 61
- **BranchBndStrategy** (Strategy for mixed integer nonlinear programming using the PICO parallel branch and bound engine) ................................................................. 62
- **CommandLineHandler** (Utility class for managing command line inputs to DAKOTA) ................................................................. 64
- **CommandShell** (Utility class which defines convenience operators for spawning processes with system calls) ................................................................. 65
- **ConcurrentStrategy** (Strategy for multi-start iteration or pareto set optimization) ................................................................. 67
- **CONMINOptimizer** (Wrapper class for the CONMIN optimization library) ................................................................. 69
- **CtelRegexp** ................................................................. 76
- **DACEIterator** (Wrapper class for the DDACE design of experiments library) ................................................................. 78
- **DakotaApproximation** (Base class for the approximation class hierarchy) ................................................................. 82
- **DakotaArray<T>** (Template class for the Dakota bookkeeping array) ................................................................. 87
- **DakotaBaseVector<T>** (Base class for the DakotaMatrix and DakotaVector classes) ................................................................. 90
- **DakotaBiStream** (The binary input stream class. Overloads the > operator for all data types) ................................................................. 93
- **DakotaBoStream** (The binary output stream class. Overloads the << operator for all data types) ................................................................. 96
- **DakotaGraphics** (Single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc) ................................................................. 99
- **DakotaInterface** (Base class for the interface class hierarchy) ................................................................. 101
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DakotaIterator</td>
<td>Base class for the iterator class hierarchy</td>
<td>107</td>
</tr>
<tr>
<td>DakotaList&lt; T &gt;</td>
<td>Template class for the Dakota bookkeeping list</td>
<td>113</td>
</tr>
<tr>
<td>DakotaMatrix&lt; T &gt;</td>
<td>Template class for the Dakota numerical matrix</td>
<td>117</td>
</tr>
<tr>
<td>DakotaModel</td>
<td>Base class for the model class hierarchy</td>
<td>119</td>
</tr>
<tr>
<td>DakotaNonD</td>
<td>Base class for all nondeterministic iterators (the DAKOTA/UQ branch)</td>
<td>130</td>
</tr>
<tr>
<td>DakotaOptimizer</td>
<td>Base class for the optimizer branch of the iterator hierarchy</td>
<td>133</td>
</tr>
<tr>
<td>DakotaResponse</td>
<td>Container class for response functions and their derivatives. DakotaResponse provides the handle class</td>
<td>138</td>
</tr>
<tr>
<td>DakotaResponseRep</td>
<td>Container class for response functions and their derivatives. DakotaResponseRep provides the body class</td>
<td>142</td>
</tr>
<tr>
<td>DakotaStrategy</td>
<td>Base class for the strategy class hierarchy</td>
<td>147</td>
</tr>
<tr>
<td>DakotaString</td>
<td>DakotaString class, used as main string class for Dakota</td>
<td>152</td>
</tr>
<tr>
<td>DakotaVarConstraints</td>
<td>Base class for the variable constraints class hierarchy</td>
<td>155</td>
</tr>
<tr>
<td>DakotaVariables</td>
<td>Base class for the variables class hierarchy</td>
<td>160</td>
</tr>
<tr>
<td>DakotaVector&lt; T &gt;</td>
<td>Template class for the Dakota numerical vector</td>
<td>166</td>
</tr>
<tr>
<td>DataInterface</td>
<td>Container class for interface specification data</td>
<td>170</td>
</tr>
<tr>
<td>DataMethod</td>
<td>Container class for method specification data</td>
<td>174</td>
</tr>
<tr>
<td>DataResponses</td>
<td>Container class for responses specification data</td>
<td>181</td>
</tr>
<tr>
<td>DataVariables</td>
<td>Container class for variables specification data</td>
<td>184</td>
</tr>
<tr>
<td>DirectFnApplicInterface</td>
<td>Derived application interface class which spawns simulation codes and testers using direct procedure calls</td>
<td>190</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>Wrapper class for the DOT optimization library</td>
<td>194</td>
</tr>
<tr>
<td>ErrorTable</td>
<td>Data structure to hold errors</td>
<td>198</td>
</tr>
<tr>
<td>ForkAnalysisCode</td>
<td>Derived class in the AnalysisCode class hierarchy which spawns simulations using forks</td>
<td>199</td>
</tr>
<tr>
<td>ForkApplicInterface</td>
<td>Derived application interface class which spawns simulation codes using forks</td>
<td>201</td>
</tr>
<tr>
<td>FunctionCompare&lt; T &gt;</td>
<td></td>
<td>204</td>
</tr>
<tr>
<td>FundamentalVarConstraints</td>
<td>Derived class within the DakotaVarConstraints hierarchy which employs the default data view (no variable or domain type array merging)</td>
<td>205</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>Derived class within the DakotaVariables hierarchy which employs the default data view (no variable or domain type array merging)</td>
<td>208</td>
</tr>
<tr>
<td>GetLongOpt</td>
<td>GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France)</td>
<td>212</td>
</tr>
<tr>
<td>HermiteSurf</td>
<td>Derived approximation class for Hermite polynomials (global approximation)</td>
<td>216</td>
</tr>
<tr>
<td>HierLayeredModel</td>
<td>Derived model class within the layered model branch for managing hierarchical surrogates (models of varying fidelity)</td>
<td>218</td>
</tr>
<tr>
<td>KrigApprox</td>
<td>Utility class for kriging interpolation</td>
<td>222</td>
</tr>
<tr>
<td>KrigingSurf</td>
<td>Derived approximation class for kriging interpolation</td>
<td>230</td>
</tr>
<tr>
<td>LayeredModel</td>
<td>Base class for the layered models (SurrLayeredModel and HierLayeredModel)</td>
<td>232</td>
</tr>
<tr>
<td>MARSSurf</td>
<td>Derived approximation class for multivariate adaptive regression splines</td>
<td>236</td>
</tr>
<tr>
<td>MergedVarConstraints</td>
<td>Derived class within the DakotaVarConstraints hierarchy which employs the merged data view</td>
<td>238</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>Derived class within the DakotaVariables hierarchy which employs the merged data view</td>
<td>241</td>
</tr>
<tr>
<td>MultilevelOptStrategy</td>
<td>Strategy for hybrid optimization using multiple optimizers on multiple models of varying fidelity</td>
<td>245</td>
</tr>
<tr>
<td>NestedModel</td>
<td>Derived model class which performs a complete sub-iterator execution within every evaluation of the model</td>
<td>248</td>
</tr>
<tr>
<td>NoDBBaseConstructor</td>
<td>Dummy struct for overloading constructors used in on-the-fly instantiations</td>
<td>254</td>
</tr>
<tr>
<td>NonDAdvMeanValue</td>
<td>Class for the analytical reliability methods within DAKOTA/UQ</td>
<td>255</td>
</tr>
<tr>
<td>NonDOptStrategy</td>
<td>Strategy for optimization under uncertainty (robust and reliability-based design)</td>
<td>262</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>NonDPCE</td>
<td>(Stochastic finite element approach to uncertainty quantification using polynomial chaos expansions)</td>
<td>264</td>
</tr>
<tr>
<td>NonDPProbability</td>
<td>(Wrapper class for the LHS library)</td>
<td>267</td>
</tr>
<tr>
<td>NonDSampling</td>
<td>(Wrapper class for the Fortran 90 LHS library)</td>
<td>271</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>(Wrapper class for the NPSOL optimization library)</td>
<td>275</td>
</tr>
<tr>
<td>ParallelLibrary</td>
<td>(Class for managing partitioning of multiple levels of parallelism and message passing within the levels)</td>
<td>279</td>
</tr>
<tr>
<td>ParamResponsePair</td>
<td>(Container class for a variables object, a response object, and an evaluation id)</td>
<td>288</td>
</tr>
<tr>
<td>ParamStudy</td>
<td>(Class for vector, list, centered, and multidimensional parameter studies)</td>
<td>291</td>
</tr>
<tr>
<td>ProblemDescDB</td>
<td>(The database containing information parsed from the DAKOTA input file)</td>
<td>295</td>
</tr>
<tr>
<td>RespSurf</td>
<td>(Derived approximation class for quadratic polynomial regression)</td>
<td>301</td>
</tr>
<tr>
<td>SGOPTApplication</td>
<td>(Maps the evaluation functions used by SGOPT algorithms to the DAKOTA evaluation functions)</td>
<td>303</td>
</tr>
<tr>
<td>SGOPTOptimizer</td>
<td>(Wrapper class for the SGOPT optimization library)</td>
<td>306</td>
</tr>
<tr>
<td>SingleMethodStrategy</td>
<td>(Simple fall-through strategy for running a single iterator on a single model)</td>
<td>309</td>
</tr>
<tr>
<td>SingleModel</td>
<td>(Derived model class which utilizes a single interface to map variables into responses)</td>
<td>310</td>
</tr>
<tr>
<td>SNLLOptimizer</td>
<td>(Wrapper class for the OPT++ optimization library)</td>
<td>312</td>
</tr>
<tr>
<td>SortCompare&lt;T&gt;</td>
<td></td>
<td>318</td>
</tr>
<tr>
<td>SurrBasedOptStrategy</td>
<td>(Strategy for provably-convergent surrogate-based optimization)</td>
<td>319</td>
</tr>
<tr>
<td>SurrLayeredModel</td>
<td>(Derived model class within the layered model branch for managing data fit surrogates (global and local))</td>
<td>323</td>
</tr>
<tr>
<td>SurrogateDataPoint</td>
<td>(Simple container class encapsulating basic parameter and response data for defining a &quot;truth&quot; data point)</td>
<td>328</td>
</tr>
<tr>
<td>SysCallAnalysisCode</td>
<td>(Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls)</td>
<td>330</td>
</tr>
<tr>
<td>SysCallApplicInterface</td>
<td>(Derived application interface class which spawns simulation codes using system calls)</td>
<td>332</td>
</tr>
<tr>
<td>TaylorSurf</td>
<td>(Derived approximation class for 1st order Taylor series (local approximation))</td>
<td>334</td>
</tr>
<tr>
<td>VariablesUtil</td>
<td>(Utility class for the Dako Variables and DakoVarConstraints hierarchies which provides convenience functions for variable vectors and label arrays for combining design, uncertain, and state variable types and merging continuous and discrete variable domains)</td>
<td>336</td>
</tr>
</tbody>
</table>
Chapter 4

DAKOTA File Index

4.1 DAKOTA File List

Here is a list of all documented files with brief descriptions:

- \texttt{keywordtable.C} (File containing keywords for the strategy, method, variables, interface, and responses input specifications from \texttt{dakota.input.spec}) \hfill \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \phantom{339}
- \texttt{main.C} (File containing the main program for DAKOTA) \hfill \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \phantom{340}
- \texttt{restart_util.C} (File containing the DAKOTA restart utility main program) \hfill \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \phantom{342}
Chapter 5

DAKOTA Page Index

5.1 DAKOTA Related Pages

Here is a list of all related documentation pages:

- Recommended Practices for DAKOTA Development ........................................ 345
- Instructions for Modifying DAKOTA's Input Specification ............................... 349
Chapter 6

DAKOTA Class Documentation

6.1 AllMergedVarConstraints Class Reference

Derived class within the DakotaVarConstraints hierarchy which combines the all and merged data views.

#include <AllMergedVarConstraints.H>

Inheritance diagram for AllMergedVarConstraints::

```
DakotaVarConstraints  VariablesUtil
                   |                |
                   |                |
                   v                v
AllMergedVarConstraints
```

Public Methods

- **AllMergedVarConstraints** (const ProblemDescDB &problem_db)  
  *constructor.*

- **~AllMergedVarConstraints** ()  
  *destructor.*

- const DakotaRealVector & **continuous_lower_bounds** () const  
  *return the active continuous variable lower bounds.*

- void **continuous_lower_bounds** (const DakotaRealVector &c_l_bnds)  
  *set the active continuous variable lower bounds.*

- const DakotaRealVector & **continuous_upper_bounds** () const  
  *return the active continuous variable upper bounds.*

- void **continuous_upper_bounds** (const DakotaRealVector &c_u_bnds)
set the active continuous variable upper bounds.

- const DakotaIntVector & \texttt{discrete\_lower\_bounds} () const
  return the active discrete variable lower bounds.

- void \texttt{discrete\_lower\_bounds} (const DakotaIntVector &d\_lbnds)
  set the active discrete variable lower bounds.

- const DakotaIntVector & \texttt{discrete\_upper\_bounds} () const
  return the active discrete variable upper bounds.

- void \texttt{discrete\_upper\_bounds} (const DakotaIntVector &d\_ubnds)
  set the active discrete variable upper bounds.

- void \texttt{write} (ostream &s) const
  write a variable constraints object to an ostream.

- void \texttt{read} (istream &s)
  read a variable constraints object from an istream.

Private Attributes

- DakotaRealVector \texttt{allMergedLowerBnds}
  a continuous lower bounds array combining design, uncertain, and state variable types and merging continuous and discrete domains. The order is continuous design, discrete design, uncertain, continuous state, and discrete state.

- DakotaRealVector \texttt{allMergedUpperBnds}
  a continuous upper bounds array combining design, uncertain, and state variable types and merging continuous and discrete domains. The order is continuous design, discrete design, uncertain, continuous state, and discrete state.

- DakotaIntVector \texttt{emptyIntVector}
  an empty int vector returned in get functions when there are no variables corresponding to the request.

6.1.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which combines the all and merged data views.

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllMergedVarConstraints derived class combines design, uncertain, and state variable types (all) and continuous and discrete domain types (merged). The result is a single continuous lower bounds array (allMergedLowerBnds) and a single continuous upper bounds array (allMergedUpperBnds). No iterators/strategies currently use this approach; it is included for completeness and future capability.

6.1.2 Constructor & Destructor Documentation
6.1.2.1 AllMergedVarConstraints::AllMergedVarConstraints (const ProblemDescDB & problem db)

constructor.

Extract fundamental variable bounds and combine them into allMergedLowerBnds and allMergedUpperBnds using utilities from VariablesUtil.

The documentation for this class was generated from the following files:

- AllMergedVarConstraints.H
- AllMergedVarConstraints.C
6.2 AllMergedVariables Class Reference

Derived class within the DakotaVariables hierarchy which combines the all and merged data views.

#include <AllMergedVariables.H>

Inheritance diagram for AllMergedVariables::

```
DakotaVariables | VariablesUtil

              | AllMergedVariables
```

Public Methods

- **AllMergedVariables ()**
  
  *default constructor.*

- **AllMergedVariables (const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **~AllMergedVariables ()**
  
  *destructor.*

- **size_j tv () const**
  
  *Returns total number of vars.*

- **size_j cv () const**
  
  *Returns number of active continuous vars.*

- **size_j dv () const**
  
  *Returns number of active discrete vars.*

- **const DakotaRealVector & continuous_variables () const**
  
  *return the active continuous variables.*

- **void continuous_variables (const DakotaRealVector &c_vars)**
  
  *set the active continuous variables.*

- **const DakotaIntVector & discrete_variables () const**
  
  *return the active discrete variables.*

- **void discrete_variables (const DakotaIntVector &d_vars)**
  
  *set the active discrete variables.*

- **const DakotaStringArray & continuous_variable_labels () const**
return the active continuous variable labels.

- void `continuous_variable_labels` (const DakotaStringArray &cv_labels) 
  set the active continuous variable labels.

- const DakotaStringArray & `discrete_variable_labels` () const
  return the active discrete variable labels.

- void `discrete_variable_labels` (const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

- const DakotaRealVector & `inactive_continuous_variables` () const
  return the inactive continuous variables.

- void `inactive_continuous_variables` (const DakotaRealVector &ic-vars)
  set the inactive continuous variables.

- const DakotaIntVector & `inactive_discrete_variables` () const
  return the inactive discrete variables.

- void `inactive_discrete_variables` (const DakotaIntVector &id-yars)
  set the inactive discrete variables.

- size_t acv () const
  returns total number of continuous vars.

- size_t adv () const
  returns total number of discrete vars.

- DakotaRealVector `all_continuous_variables` () const
  returns a single array with all continuous variables.

- DakotaIntVector `all_discrete_variables` () const
  returns a single array with all discrete variables.

- void `read` (istream &s)
  read a variables object from an istream.

- void `write` (ostream &s) const
  write a variables object to an ostream.

- void `read_annotated` (istream &s)
  read a variables object in annotated format from an istream.

- void `write_annotated` (ostream &s) const
  write a variables object in annotated format to an ostream.

- void `read` (DakotaBiStream &s)
  read a variables object from the binary restart stream.


- void write (DakotaBoStream &s) const
  
  write a variables object to the binary restart stream.

- void read (UnPackBuffer &s)
  
  read a variables object from a packed MPI buffer.

- void write (PackBuffer &s) const
  
  write a variables object to a packed MPI buffer.

**Private Methods**

- void copy_rep (const DakotaVariables *vars_rep)
  
  Used by copy() to copy the contents of a letter class.

**Private Attributes**

- DakotaRealVector allMergedVars
  
  a continuous array combining design, uncertain, and state variable types and merging continuous and discrete domains. The order is continuous design, discrete design, uncertain, continuous state, and discrete state.

- DakotaStringArray allMergedLabels
  
  an array containing labels for continuous design, discrete design, uncertain, continuous state, and discrete state variables.

- DakotaRealVector emptyRealVector
  
  an empty real vector returned in get functions when there are no variables corresponding to the request.

- DakotaIntVector emptyIntVector
  
  an empty int vector returned in get functions when there are no variables corresponding to the request.

- DakotaStringArray emptyStringArray
  
  an empty label array returned in get functions when there are no variables corresponding to the request.

**Friends**

- int operator== (const AllMergedVariables &vars1, const AllMergedVariables &vars2)
  
  equality operator.

**6.2.1 Detailed Description**

Derived class within the DakotaVariables hierarchy which combines the all and merged data views.

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllMergedVariables derived class combines design, uncertain,
and state variable types (all) and continuous and discrete domain types (merged). The result is a single array of continuous variables (allMergedVars). No iterators/strategies currently use this approach; it is included for completeness and future capability.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AllMergedVariables::AllMergedVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels and combine them into allMergedVars and allMergedLabels using utilities from VariablesUtil.

The documentation for this class was generated from the following files:

- AllMergedVariables.H
- AllMergedVariables.C
6.3 AllVarConstraints Class Reference

Derived class within the DakotaVarConstraints hierarchy which employs the all data view.

```cpp
#include <AllVarConstraints.H>
```

Inheritance diagram for AllVarConstraints:

```
DakotaVarConstraints  VariablesUtil
                    /
                   /
                  /
                 /
                /
               /
              /
             /   
            /    
           /     
          /      
         /       
        /        
       /         
      AllVarConstraints
```

Public Methods

- **AllVarConstraints** (const ProblemDescDB &problem_db)
  
  constructor.

- **~AllVarConstraints** ()

  destructor.

- const DakotaRealVector & **continuous_lower_bounds** () const

  return the active continuous variable lower bounds.

- void **continuous_lower_bounds** (const DakotaRealVector &c_lbnds)

  set the active continuous variable lower bounds.

- const DakotaRealVector & **continuous_upper_bounds** () const

  return the active continuous variable upper bounds.

- void **continuous_upper_bounds** (const DakotaRealVector &c_ubnds)

  set the active continuous variable upper bounds.

- const DakotaIntVector & **discrete_lower_bounds** () const

  return the active discrete variable lower bounds.

- void **discrete_lower_bounds** (const DakotaIntVector &d_lbnds)

  set the active discrete variable lower bounds.

- const DakotaIntVector & **discrete_upper_bounds** () const

  return the active discrete variable upper bounds.

- void **discrete_upper_bounds** (const DakotaIntVector &d_ubnds)

  set the active discrete variable upper bounds.

- void **write** (ostream &s) const
6.3 AllVarConstraints Class Reference

write a variable constraints object to an ostream.

- void read (istream &s)
  read a variable constraints object from an istream.

Private Attributes

- DakotaRealVector allContinuousLowerBnds
  a continuous lower bounds array combining continuous design, uncertain, and continuous state variable types (all view).

- DakotaRealVector allContinuousUpperBnds
  a continuous upper bounds array combining continuous design, uncertain, and continuous state variable types (all view).

- DakotaIntVector allDiscreteLowerBnds
  a discrete lower bounds array combining discrete design and discrete state variable types (all view).

- DakotaIntVector allDiscreteUpperBnds
  a discrete upper bounds array combining discrete design and discrete state variable types (all view).

- size_t numCDV
  number of continuous design variables.

- size_t numDDV
  number of discrete design variables.

- size_t numUV
  number of uncertain variables.

- size_t numCSV
  number of continuous state variables.

- size_t numDSV
  number of discrete state variables.

6.3.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which employs the all data view.

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllVarConstraints derived class combines design, uncertain, and state variable types but separates continuous and discrete domain types. The result is combined continuous bounds arrays (allContinuousLowerBnds, allContinuousUpperBnds) and combined discrete bounds arrays (allDiscreteLowerBnds, allDiscreteUpperBnds). Parameter and DACE studies currently use this approach (see DakotaVariables::get_variables(problem_db) for variables type selection; variables type is passed to the DakotaVarConstraints constructor in DakotaModel).
6.3.2 Constructor & Destructor Documentation

6.3.2.1 AllVarConstraints::AllVarConstraints (const ProblemDescDB & problem_db)

constructor.

Extract fundamental lower and upper bounds and combine them into allContinuousLowerBnds, allContinuousUpperBnds, allDiscreteLowerBnds, and allDiscreteUpperBnds using utilities from VariablesUtil.

The documentation for this class was generated from the following files:

- AllVarConstraints.H
- AllVarConstraints.C
6.4 AllVariables Class Reference

Derived class within the DakotaVariables hierarchy which employs the all data view.

```cpp
#include <AllVariables.h>
```

Inheritance diagram for AllVariables::

```
DakotaVariables  |  VariablesUtil
                /  
               /    
     AllVariables
```

Public Methods

- **AllVariables ()**
  
  *default constructor.*

- **AllVariables (const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **~AllVariables ()**
  
  *destructor.*

- **size_t tv () const**
  
  *Returns total number of vars.*

- **size_t cv () const**
  
  *Returns number of active continuous vars.*

- **size_t dv () const**
  
  *Returns number of active discrete vars.*

- **const DakotaRealVector & continuous_variables () const**
  
  *return the active continuous variables.*

- **void continuous_variables (const DakotaRealVector &c_vars)**
  
  *set the active continuous variables.*

- **const DakotaIntVector & discrete_variables () const**
  
  *return the active discrete variables.*

- **void discrete_variables (const DakotaIntVector &d-vars)**
  
  *set the active discrete variables.*

- **const DakotaStringArray & continuous_variable_labels () const**
  
  *set the active variable labels.*
return the active continuous variable labels.

- void **continuous_variable_labels** (const DakotaStringArray &cv_labels)
  
  *set the active continuous variable labels.*

- const DakotaStringArray & **discrete_variable_labels** () const
  
  *return the active discrete variable labels.*

- void **discrete_variable_labels** (const DakotaStringArray &dv_labels)
  
  *set the active discrete variable labels.*

- const DakotaRealVector & **inactive_continuous_variables** () const
  
  *return the inactive continuous variables.*

- void **inactive_continuous_variables** (const DakotaRealVector &ic-vars)
  
  *set the inactive continuous variables.*

- const DakotaIntVector & **inactive_discrete_variables** () const
  
  *return the inactive discrete variables.*

- void **inactive_discrete_variables** (const DakotaIntVector &id-vars)
  
  *set the inactive discrete variables.*

- size_t **acv** () const
  
  *returns total number of continuous vars.*

- size_t **adv** () const
  
  *returns total number of discrete vars.*

- DakotaRealVector **all_continuous_variables** () const
  
  *returns a single array with all continuous variables.*

- DakotaIntVector **all_discrete_variables** () const
  
  *returns a single array with all discrete variables.*

- void **read** (istream &s)
  
  *read a variables object from an istream.*

- void **write** (ostream &s) const
  
  *write a variables object to an ostrom.*

- void **read_annoted** (istream &s)
  
  *read a variables object in annotated format from an istream.*

- void **write_annoted** (ostream &s) const
  
  *write a variables object in annotated format to an ostrom.*

- void **read** (DakotaBiStream &s)
  
  *read a variables object from the binary restart stream.*
6.4 AllVariables Class Reference

- **void** write (DakotaBoStream &s) const
  
  write a variables object to the binary restart stream.

- **void** read (UnPackBuffer &s)
  
  read a variables object from a packed MPI buffer.

- **void** write (PackBuffer &s) const
  
  write a variables object to a packed MPI buffer.

**Private Methods**

- **void** copy_rep (const DakotaVariables *vars_rep)
  
  Used by copy() to copy the contents of a letter class.

**Private Attributes**

- DakotaRealVector allContinuousVars
  
  a continuous array combining continuous design, uncertain, and continuous state variable types (all).

- DakotaIntVector allDiscreteVars
  
  a discrete array combining discrete design and discrete state variable types (all).

- DakotaStringArray allContinuousLabels
  
  a label array combining continuous design, uncertain, and continuous state variable types (all).

- DakotaStringArray allDiscreteLabels
  
  a label array combining discrete design and discrete state variable types (all).

- DakotaRealVector emptyRealVector
  
  an empty real vector returned in get functions when there are no variables corresponding to the request.

- DakotaIntVector emptyIntVector
  
  an empty int vector returned in get functions when there are no variables corresponding to the request.

- `size_t` numCDV
  
  number of continuous design variables.

- `size_t` numDDV
  
  number of discrete design variables.

- `size_t` numUV
  
  number of uncertain variables.

- `size_t` numCSV
  
  number of continuous state variables.

- `size_t` numDSV
  
  number of discrete state variables.
Friends

- int operator==(const AllVariables &vars1, const AllVariables &vars2)
  
  equality operator.

6.4.1 Detailed Description

Derived class within the DakotaVariables hierarchy which employs the all data view.

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The AllVariables derived class combines design, uncertain, and state variable types but separates continuous and discrete domain types. The result is a single array of continuous variables (allContinuousVars) and a single array of discrete variables (allDiscreteVars). Parameter and DACE studies currently use this approach (see DakotaVariables::get_variables(problem_db)).

6.4.2 Constructor & Destructor Documentation

6.4.2.1 AllVariables::AllVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels and combine them into allContinuousVars, allDiscreteVars, allContinuousLabels, and allDiscreteLabels using utilities from VariablesUtil.

The documentation for this class was generated from the following files:

- AllVariables.H
- AllVariables.C
6.5 AnalysisCode Class Reference

Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations.

```
#include <AnalysisCode.H>
```

Inheritance diagram for AnalysisCode::

```
AnalysisCode

ForkAnalysisCode
SysCallAnalysisCode
```

Public Methods

- `void define_filenames (const int id)`
  
  define modified filenames from user input by handling Unix temp file and tagging options.

- `void write_parameters_file (const DakotaVariables &vars, const DakotaIntArray &asv, const int id)`
  
  write the variables and active set vector objects to the parameters file in either standard or aprepro format.

- `void read_results_file (DakotaResponse &response, const int id)`
  
  read the response object from the results file.

- `const DakotaStringList & program_names () const`
  
  return programNames.

- `const DakotaString & input_filter_name () const`
  
  return iFilterName.

- `const DakotaString & output_filter_name () const`
  
  return oFilterName.

- `const DakotaString & modified_parameters_filename () const`
  
  return modifiedParamsFileName.

- `const DakotaString & modified_results_filename () const`
  
  return modifiedResFileName.

- `const DakotaString & results_fname (const int id) const`
  
  return the entry in resultsFNameList corresponding to id.

- `void quiet_flag (const short flag)`
  
  set quietFlag.
- short quiet_flag () const
  
  return quietFlag.

**Protected Methods**

- **AnalysisCode** (const ProblemDescDB &problem_db)
  
  constructor.

- virtual ~AnalysisCode ()
  
  destructor.

**Protected Attributes**

- short quietFlag
  
  flag set by master processor to quiet output from slave processors.

- short verboseFlag
  
  flag for additional analysis code output if method verbosity is set.

- short fileTagFlag
  
  flags tagging of parameter/results files.

- short fileSaveFlag
  
  flags retention of parameter/results files.

- short apreproFlag
  
  flags use of the APREPRO (the Sandia "A PRE PROcessor" utility) format for parameter files.

- DakotaString iFilterName
  
  the name of the input filter (input\_filter user specification).

- DakotaString oFilterName
  
  the name of the output filter (output\_filter user specification).

- DakotaStringList programNames
  
  the names of the analysis code programs (analysis\_drivers user specification).

- size_t numPrograms
  
  the number of analysis code programs (length of programNames list).

- DakotaString parametersFileName
  
  the name of the parameters file from user specification.

- DakotaString modifiedParamsFileName
  
  the parameters file name actually used (modified with tagging or temp files).

- DakotaString resultsFileName
the name of the results file from user specification.

- **DakotaString modifiedResFileName**
  the results file name actually used (modified with tagging or temp files).

- **DakotaStringList parametersFNameList**
  list of parameters file names used in spawning function evaluations.

- **DakotaStringList resultsFNameList**
  list of results file names used in spawning function evaluations.

- **DakotaIntList fileNameKey**
  stores function evaluation identifiers to allow key-based retrieval of file names from parametersFNameList and resultsFNameList.

### Private Attributes

- **ParallelLibrary & parallelLib**
  reference to the `ParallelLibrary` object. Used in `define_filenames()`.

### 6.5.1 Detailed Description

Base class providing common functionality for derived classes (SysCallAnalysisCode and ForkAnalysisCode) which spawn separate processes for managing simulations.

The AnalysisCode class hierarchy provides simulation spawning services for ApplicationInterface derived classes and alleviates these classes of some of the specifics of simulation code management. The hierarchy does not employ the letter-envelope technique since the ApplicationInterface derived classes instantiate the appropriate derived AnalysisCode class directly.

The documentation for this class was generated from the following files:

- AnalysisCode.H
- AnalysisCode.C
## 6.6 ANNSurf Class Reference

Derived approximation class for artificial neural networks.

```cpp
#include <ANNSurf.H>
```

Inheritance diagram for ANNSurf::

```
DakotaApproximation

ANNSurf
```

### Public Methods

- **`ANNSurf (const ProblemDescDB &problem_db)`**
  *constructor.*

- **`~ANNSurf ()`**
  *destructor.*

### Protected Methods

- **`int required_samples (int num_vars)`**
  *return the minimum number of samples required to build the derived class approximation type in `num_vars` dimensions.*

- **`void find_coefficients ()`**
  *calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.*

- **`Real get_value (const DakotaRealVector &x)`**
  *retrieve the approximate function value for a given parameter vector.*

### Private Attributes

- **`ANNAprox * annObject`**
  *pointer to the ANNAprox object (see VendorPackages/ann for class declaration).*

### 6.6.1 Detailed Description

Derived approximation class for artificial neural networks.
The ANNSurf class uses a layered-perceptron artificial neural network. Unlike most neural networks, it does not employ a back-propagation approach to training. Rather it uses a direct training approach developed by Prof. David Zimmerman of the University of Houston and modified by Tom Paez and Chris O’Gorman of Sandia. It is more computationally efficient that back-propagation networks, but relative accuracy can be a concern.

The documentation for this class was generated from the following files:

- ANNSurf.H
- ANNSurf.C
6.7 ApplicationInterface Class Reference

Derived class within the interface class hierarchy for supporting interfaces to simulation codes.

```cpp
#include <ApplicationInterface.H>
```

Inheritance diagram for ApplicationInterface::

```
DakotaInterface

ApplicationInterface

DirectFnApplicInterface ForkApplicInterface SysCallApplicInterface
```

Protected Methods

- **ApplicationInterface** (const ProblemDescDB &problemdb, const size_t &num_fns)
  constructor.

- **~ApplicationInterface** ()
  destructor.

- void **init_communicators** (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  allocate communicators for the evaluation and analysis parallelism levels.

- void **free_communicators** ()
  deallocate communicators for the evaluation and analysis parallelism levels.

- int **asynch_local_evaluation_concurrency** () const
  return asynchLocalEvalConcurrency.

- DakotaString **interface_synchronization** () const
  return interfaceSynchronization.

- void **map** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, const int asynch_flag=0)
  Provides a “mapping” of variables to responses using a simulation. Protected due to DakotaInterface letter-envelope idiom.

- void **manage_failure** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int failed_eval)
  manages a simulation failure using abort/retry/recover/continuation.

- const DakotaArray < DakotaResponse > & synch ()
executes a blocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns all jobs.

- const DakotaList& DakotaResponse & synch_nowait()
  executes a nonblocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns a partial list of completed jobs.

- void serve_evaluations()
  run on evaluation servers to serve the iterator master.

- void stop_evaluation_servers()
  used by the iterator master to terminate evaluation servers.

- virtual void derived_map (const DakotaVariables & vars, const DakotaIntArray & asv, DakotaResponse & response, int fn_eval_id)=0
  Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- virtual void derived_map_asynch (const ParamResponsePair & pair)=0
  Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- virtual void derived_synch (DakotaList< ParamResponsePair > & prp_list)=0
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- virtual void derived_synch_nowait (DakotaList< ParamResponsePair > & prp_list)=0
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- virtual void clear_bookkeeping()
  clears any bookkeeping in derived classes.

- void self_schedule_analyses()
  blocking self-schedule of all analyses within a function evaluation using message passing.

- void serve_analyses_synch()
  serve the master analysis scheduler and manage one synchronous analysis job at a time.

- virtual int derived_synchronous_local_analysis (const int & analysis_id)=0
  Execute a particular analysis (identified by analysis_id) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve_analyses_synch().

Protected Attributes

- ParallelLibrary & parallelLib
  reference to the ParallelLibrary object used to manage MPI partitions for the concurrent evaluations and concurrent analyses parallelism levels.
- **short evalMessagePass**
  flags use of message passing at the level of evaluation scheduling.

- **short analysisMessagePass**
  flags use of message passing at the level of analysis scheduling.

- **short suppressOutput**
  flag for suppressing output on slave processors.

- **int asyncLocalAnalysisConcurrency**
  limits the number of concurrent analyses in asynchronous local scheduling and specifies hybrid concurrency when message passing.

- **short asyncLocalAnalysisFlag**
  flag for asynchronous local parallelism of analyses.

- **int worldSize**
  size of `MPI_COMM_WORLD`.

- **int iteratorCommSize**
  size of `iteratorComm`.

- **int evalCommSize**
  size of `evalComm`.

- **int analysisCommSize**
  size of `analysisComm`.

- **int worldRank**
  processor rank within `MPI_COMM_WORLD`.

- **int iteratorCommRank**
  processor rank within `iteratorComm`.

- **int evalCommRank**
  processor rank within `evalComm`.

- **int analysisCommRank**
  processor rank within `analysisComm`.

- **int evalServerId**
  evaluation server identifier.

- **int analysisServerId**
  analysis server identifier.

- **short evalDedMasterFlag**
  flag for dedicated master partitioning at the level of evaluation scheduling.


- short multiProcAnalysisFlag
  
  flag for multiprocessor analysis partitions.

- DakotaStringList analysisDrivers
  
  the set of analyses within each function evaluation (from the analysis_drivers interface specification).

- int numAnalysisDrivers
  
  length of analysisDrivers list.

- int numAnalysisServers
  
  number of analysis servers.

- MPI_Comm evalComm
  
  intracomm for fn eval: partition of iteratorComm.

- MPI_Comm analysisComm
  
  intracomm for analysis; partition of evalComm.

- int lenVarsMessage
  
  length of a ByteBuffer containing a DakotaVariables object; computed in DakotaModel::initCommunicators().

- int lenVarsASVMessage
  
  length of a ByteBuffer containing a DakotaVariables object and an active set vector object; computed in DakotaModel::initCommunicators().

- int lenResponseMessage
  
  length of a ByteBuffer containing a DakotaResponse object; computed in DakotaModel::initCommunicators().

- int lenPRPairMessage
  
  length of a ByteBuffer containing a ParamResponsePair object; computed in DakotaModel::initCommunicators().

### Private Methods

- int duplication_detect (const DakotaVariables &vars, DakotaResponse &response, const int asynch_flag)

  checks data pairs and beforeSynchPRPList to see if the current evaluation request has already been performed or queued.

- void self_schedule_evaluations ()

  blocking self-schedule of all evaluations in beforeSynchPRPList using message passing; executes on iteratorComm master.

- void static_schedule_evaluations ()

  blocking static schedule of all evaluations in beforeSynchPRPList using message passing; executes on iteratorComm master.

- void asynchronous_local_evaluations (DakotaList< ParamResponsePair > &prp_list)
perform all jobs in prp_list using asynchronous approaches on the local processor.

- void synchronous_{local}evaluations (DakotaList< ParamResponsePair > &prp_list)
  perform all jobs in prp_list using synchronous approaches on the local processor.

- void asynchronous_{local}evaluations_{nowait} (DakotaList< ParamResponsePair > &prp_list)
  launch new jobs in prp_list asynchronously (if capacity is available), perform nonblocking query of all running jobs, and process any completed jobs.

- void serve_{evaluations}_{synchronous} ()
  serve the evaluation message passing schedulers and perform one synchronous evaluation at a time.

- void serve_{evaluations}_{asynchronous} ()
  serve the evaluation message passing schedulers and manage multiple asynchronous evaluations.

- void serve_{evaluations}_{peer} ()
  serve the evaluation message passing schedulers and perform one synchronous evaluation at a time as part of the 1st peer.

- const ParamResponsePair & get_{source}_{pair} (const DakotaVariables &target_vars)
  convenience function for the continuation approach in manage_failure() for finding the nearest successful "source" evaluation to the failed "target".

- void continuation (const DakotaVariables &target_vars, const DakotaIntArray &asv, DakotaResponse &response, const ParamResponsePair &source_pair, int failed_eval_id)
  performs a 0th order continuation method to step from a successful "source" evaluation to the failed "target". Invoked by manage_failure() for failAction == "continuation".

**Private Attributes**

- int numEvalServers
  number of evaluation servers.

- int procsPerAnalysis
  processors per analysis servers.

- DakotaString evalScheduling
  user specification of evaluation scheduling algorithm (self, static, or no spec). Used for manual overrides of the auto-configure logic in ParallelLibrary::resolve_inputs().

- DakotaString analysisScheduling
  user specification of analysis scheduling algorithm (self, static, or no spec). Used for manual overrides of the auto-configure logic in ParallelLibrary::resolve_inputs().

- int asyncLocalEvalConcurrency
  limits the number of concurrent evaluations in asynchronous local scheduling and specifies hybrid concurrency when message passing.

- DakotaString interfaceSynchronization
  interface synchronization specification: synchronous (default) or asynchronous.
• short headerFlag  
  used by synch_nowait to manage output frequency (since this function may be called many times prior to any completions).

• short asvControl  
  used by map to manage the user’s setting for active set vector control. 1 = on = modify the ASV each evaluation as appropriate (default); 0 = off = ASV values are static so that the user need not check them on each evaluation.

• DakotaIntArray defaultASV  
  the static ASV values used when the user has selected asvControl = off.

• DakotaString failAction  
  mitigation action for captured simulation failures: abort, retry, recover, or continuation.

• int failRetryLimit  
  limit on the number of retries for the retry failAction.

• DakotaRealVector failRecoveryFnVals  
  the dummy function values used for the recover failAction.

• DakotaIntArray historyDuplicateIds  
  used to bookkeep fnEvalId of asynchronous evaluations which duplicate data pairs evaluations.

• DakotaList< DakotaResponse > historyDuplicateResponses  
  used to bookkeep response of asynchronous evaluations which duplicate data pairs evaluations.

• DakotaIntArray beforeSynchDuplicateIds  
  used to bookkeep fnEvalId of asynchronous evaluations which duplicate queued beforeSynchPRPList evaluations.

• DakotaSizeTList beforeSynchDuplicateIndices  
  used to bookkeep beforeSynchPRPList index of asynchronous evaluations which duplicate queued beforeSynchPRPList evaluations.

• DakotaList< DakotaResponse > beforeSynchDuplicateResponses  
  used to bookkeep response of asynchronous evaluations which duplicate queued beforeSynchPRPList evaluations.

• DakotaIntArray runningList  
  used by asynchronous_local_nowait to bookkeep which jobs are running.

• DakotaList< ParamResponsePair > beforeSynchPRPList  
  used to bookkeep vars/asv/response of nonduplicate asynchronous evaluations. This is the queue of jobs populated by asynchronous map() invocations which is later scheduled on a call to synch() or synch_nowait().
6.7.1 Detailed Description

Derived class within the interface class hierarchy for supporting interfaces to simulation codes.

ApplicationInterface provides an interface class for performing parameter to response mappings using simulation code(s). It provides common functionality for a number of derived classes and contains the majority of all of the scheduling algorithms in DAKOTA. The derived classes provide the specifics for managing code invocations using system calls, forks, direct procedure calls, or XML over sockets.

6.7.2 Member Function Documentation

6.7.2.1 `void ApplicationInterface::map (const DakotaVariables & vars, const DakotaIntArray & asv, DakotaResponse & response, const int asynch_flag = 0)` [protected, virtual]

Provides a "mapping" of variables to responses using a simulation. Protected due to DakotaInterface letter-envelope idiom.

The function evaluator for application interfaces. Called from derived `compute_response()` and derived `asynch_compute_response()` in derived DakotaModel classes. If asynch_flag is not set, perform a blocking evaluation (using `derived_map()`). If asynch_flag is set, add the job to the beforeSynchPRPList queue for execution by one of the scheduler routines in `synch()` or `synch_nowait()`. Duplicate function evaluations are detected with `duplication_detect()`.

Reimplemented from DakotaInterface.

6.7.2.2 `const DakotaArray< DakotaResponse > & ApplicationInterface::synch ()` [protected, virtual]

executes a blocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns all jobs.

This function provides blocking synchronization for all cases of asynchronous evaluations, including the local asynchronous case (background system call, nonblocking fork, & multithreads), the message passing case, and the hybrid case. Called from derived `synchronize()` in derived DakotaModel classes.

Reimplemented from DakotaInterface.

6.7.2.3 `const DakotaList< DakotaResponse > & ApplicationInterface::synch_nowait ()` [protected, virtual]

executes a nonblocking schedule for asynchronous evaluations in the beforeSynchPRPList queue and returns a partial list of completed jobs.

This function will eventually provide nonblocking synchronization for all cases of asynchronous evaluations, however it currently supports only the local asynchronous case since nonblocking message passing schedulers have not yet been implemented. Called from derived `synchronize_nowait()` in derived DakotaModel classes.

Reimplemented from DakotaInterface.
6.7.2.4  void ApplicationInterface::serve_evaluations ()  [protected, virtual]  

run on evaluation servers to serve the iterator master.

Invoked by the serve() function in derived DakotaModel classes. Passes control to serve_evaluations_asynch(), serve_evaluations_peer(), or serve_evaluations_synch() according to specified concurrency and self/static scheduler configuration.

Reimplemented from DakotaInterface.

6.7.2.5  void ApplicationInterface::stop_evaluation_servers ()  [protected, virtual]  

used by the iterator master to terminate evaluation servers.

This code is executed on the iteratorComm rank 0 processor when iteration on a particular model is complete. It sends a termination signal (tag = 0 instead of a valid fn evalid) to each of the slave analysis servers. NOTE: This function is called from the Strategy layer even when in serial mode. Therefore, use both USE_MPI and iteratorCommSize to provide appropriate fall through behavior.

Reimplemented from DakotaInterface.

6.7.2.6  void ApplicationInterface::self_schedule_analyses ()  [protected]  

blocking self-schedule of all analyses within a function evaluation using message passing.

This code is called from derived classes to provide the master portion of a master-slave algorithm for the dynamic self-scheduling of analyses among slave servers. It is patterned after self_schedule_evaluations(). It performs no analyses locally and matches either serve_analyses_asynch() or serve_analyses_synch() on the slave servers, depending on the value of asynchLocalAnalysisConcurrency. Self-scheduling approach assigns jobs in 2 passes. The 1st pass gives each server the same number of jobs (equal to asynchLocalAnalysisConcurrency). The 2nd pass assigns the remaining jobs to slave servers as previous jobs are completed. Single- and multilevel parallel use intra- and inter-communicators, respectively, for send/receive.

Specific syntax is encapsulated within ParallelLibrary.

6.7.2.7  void ApplicationInterface::serve_analyses_synch ()  [protected]  

serve the master analysis scheduler and manage one synchronous analysis job at a time.

This code is called from derived classes to run synchronous analyses on slave processors. The slaves receive requests (blocking receive), do local derived_map ac’s, and return codes. This is done continuously until a termination signal is received from the master. It is patterned after serve_evaluations_synch().

6.7.2.8  int ApplicationInterface::duplication_detect (const DakotaVariables & vars, DakotaResponse & response, const int asynch_flag)  [private]  

checks data_pairs and beforeSynchPRPList to see if the current evaluation request has already been performed or queued.

Check incoming evaluation request for duplication with content of data_pairs and beforeSynchPRPList. If duplication is detected, return true, else return false. Manage bookkeeping with historyDuplicate and beforeSynchDuplicate lists. Called from map().
### 6.7.2.9 void ApplicationInterface::self_schedule_evaluations () [private]

blocking self-schedule of all evaluations in beforeSynchPRPList using message passing; executes on iteratorComm master.

This code is called from `synch()` to provide the master portion of a master-slave algorithm for the dynamic self-scheduling of evaluations among slave servers. It performs no evaluations locally and matches either `serve_evaluations_synchron()` or `serve_evaluations_asynchronous()` on the slave servers, depending on the value of `asynchLocalEvalConcurrency`. Self-scheduling approach assigns jobs in 2 passes. The 1st pass gives each server the same number of jobs (equal to `asynchLocalEvalConcurrency`). The 2nd pass assigns the remaining jobs to slave servers as previous jobs are completed. Single- and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within `ParallelLibrary`.

### 6.7.2.10 void ApplicationInterface::static_schedule_evaluations () [private]

blocking static schedule of all evaluations in beforeSynchPRPList using message passing; executes on iteratorComm master.

This code runs on the iteratorCommRank 0 processor (the iterator) and is called from `synch()` in order to assign a static schedule. It matches `serve_evaluations_peer()` for any other processors within the 1st evaluation partition and `serve_evaluations_synchron()`/`serve_evaluations_asynchronous()` for all other evaluation partitions (depending on `asynchLocalEvalConcurrency`). It performs function evaluations locally for its portion of the static schedule using either `asynchronous_local_evaluations()` or `synchronous_local_evaluations()`. Single-level and multilevel parallel use intra- and inter-communicators, respectively, for send/receive. Specific syntax is encapsulated within `ParallelLibrary`. The iteratorCommRank 0 processor assigns the static schedule since it is the only processor with access to beforeSynchPRPList (it runs the iterator and calls synchronize). The alternate design of each peer selecting its own jobs using the modulus operator would be applicable if execution of this function (and therefore the job list) were distributed.

### 6.7.2.11 void ApplicationInterface::asynchronous_local_evaluations (DakotaList&lt;ParamResponsePair &gt; & prp_list) [private]

perform all jobs in `prp_list` using asynchronous approaches on the local processor.

This function provides blocking synchronization for the local async case (background system call, non-blocking fork, or threads). It can be called from `synch()` for a complete local scheduling of all asynchronous jobs or from `static_schedule_evaluations()` to perform a local portion of the total job set. It uses the `derived_map_asynchronous()` to initiate asynchronous evaluations and `derived_synchron()` to capture completed jobs, and mirrors the `self_schedule_evaluations()` message passing scheduler as much as possible (`derived_synchron()` is modeled after MPI_Waitsome()).

### 6.7.2.12 void ApplicationInterface::synchronous_local_evaluations (DakotaList&lt;ParamResponsePair &gt; & prp_list) [private]

perform all jobs in `prp_list` using synchronous approaches on the local processor.

This function provides blocking synchronization for the local synchronous case (foreground system call, blocking fork, or procedure call from `derived_map()`). It is called from `static_schedule_evaluations()` to perform a local portion of the total job set.
### 6.7.13 void ApplicationInterface::asynchronous_local_evaluations_nowait (DakotaList<
ParamResponsePair> & prp_list) [private]

launch new jobs in prp_list asynchronously (if capacity is available), perform nonblocking query of all running jobs, and process any completed jobs.

This function provides nonblocking synchronization for the local async case (background system call, nonblocking fork, or threads). It is called from synch_nowait() and passed the complete set of all asynchronous jobs (beforeSynchPRPList). It uses derived_map_async() to initiate asynchronous evaluations and derived_synch_nowait() to capture completed jobs in nonblocking mode. It mirrors a nonblocking message passing scheduler as much as possible (derived_synch_nowait() modeled after MPI_Testsome()). The results of this function are rawResponseList and completionList. Since rawResponseList is in no particular order, completionList must be used as a key. It is assumed that the incoming prp_list contains only active and new jobs - i.e., all completed jobs are cleared by synch_nowait().

### 6.7.14 void ApplicationInterface::serve_evaluations_synch () [private]

serve the evaluation message passing schedulers and perform one synchronous evaluation at a time.

This code is invoked by serve_evaluations() to perform one synchronous job at a time on each slave/peer server. The servers receive requests (blocking receive), do local synchronous maps, and return results. This is done continuously until a termination signal is received from the master (sent via stop_evaluation_servers()).

### 6.7.15 void ApplicationInterface::serve_evaluations_asynch () [private]

serve the evaluation message passing schedulers and manage multiple asynchronous evaluations.

This code is invoked by serve_evaluations() to perform multiple asynchronous jobs on each slave/peer server. The servers test for any incoming jobs, launch any new jobs, process any completed jobs, and return any results. Each of these components is nonblocking, although the server loop continues until a termination signal is received from the master (sent via stop_evaluation_servers()). In the master-slave case, the master maintains the correct number of jobs on each slave. In the static scheduling case, each server is responsible for limiting concurrency (since the entire static schedule is sent to the peers at start up).

### 6.7.16 void ApplicationInterface::serve_evaluations_peer () [private]

serve the evaluation message passing schedulers and perform one synchronous evaluation at a time as part of the 1st peer.

This code is invoked by serve_evaluations() to perform a synchronous evaluation in coordination with the iteratorCommRank 0 processor (the iterator) for static schedules. The bcast() matches the bcast() in synchronous_local_evaluations(), which is invoked by static_schedule_evaluations().

The documentation for this class was generated from the following files:

- ApplicationInterface.H
- ApplicationInterface.C
6.8 ApproximationInterface Class Reference

Derived class within the interface class hierarchy for supporting approximations to simulation-based results.

```cpp
#include <ApproximationInterface.H>
```

Inheritance diagram for ApproximationInterface:

```
DakotaInterface
    
ApproximationInterface
```

Public Methods

- `ApproximationInterface` (ProblemDescDB &problem_db, const size_t &num_cv, const size_t &num_fns)
  
  Constructor.

- `~ApproximationInterface` ()
  
  Destructor.

Protected Methods

- void `map` (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, const int asynch_flag=0)
  
  The function evaluator: provides an approximate "mapping" from the variables to the responses using functionSurfaces.

- int `minimum_samples` () const
  
  Returns minSamples.

- void `build_global_approximation` (DakotaIterator &dace_iterator)
  
  Builds a global approximation for use as a surrogate.

- void `build_local_approximation` (DakotaModel &actual_model)
  
  Builds a local approximation for use as a surrogate.

- void `update_approximation` (const DakotaRealVector &x_star, const DakotaResponse &response, const DakotaResponse &response)
  
  Updates an existing global approximation with new data.

- const DakotaArray< DakotaResponse > & synch ()
  
  Recovers data from a series of asynchronous evaluations (blocking).
6.8 Approximation Interface Class Reference

- const DakotaList&lt; DakotaResponse &gt;&amp; synch_nowait()
  
  recovers data from a series of asynchronous evaluations (nonblocking).

Private Attributes

- DakotaString dacemethodPointer
  
  string pointer to the dace iterator specified by the user in the global approximation specification.

- DakotaString actualInterfacePointer
  
  string pointer to the actual interface specified by the user in the local/multipoint approximation specifications.

- DakotaArray&lt; DakotaApproximation &gt; functionSurfaces
  
  list of approximations, one per response function.

- DakotaRealVector approxScale
  
  vector of approximation scalings from an external file.

- DakotaRealVector approxOffset
  
  vector of approximation offsets from an external file.

- DakotaString sampleReuse
  
  user selection of sample reuse type: all, region, or none (default).

- short graphicsFlag
  
  controls 3D graphics of approximation surfaces.

- int minSamples
  
  the minimum number of samples over all functionSurfaces.

- DakotaList&lt; DakotaResponse &gt; beforeSynchResponseList
  
  bookkeeping list to catalogue responses generated in map for use in synch() and synch_nowait(). This supports pseudo-asynchronous operations (approximate responses all always computed synchronously, but asynchronous virtual functions are supported through bookkeeping).

6.8.1 Detailed Description

Derived class within the interface class hierarchy for supporting approximations to simulation-based results.

Approximation Interface provides an interface class for building a set of global/local/multipoint approximations and performing approximate function evaluations using them. It contains a list of DakotaApproximation objects, one for each response function.

6.8.2 Member Data Documentation
6.8.2.1 **DakotaString** `ApproximationInterface::daceMethodPointer` [private]

String pointer to the dace iterator specified by the user in the global approximation specification.

This pointer is not used for building objects since this is managed in SurrLayeredModels. Its use in `ApproximationInterface` is currently limited to flagging dace contributions to data sets in `build_global_approximation()`.

6.8.2.2 **DakotaString** `ApproximationInterface::actualInterfacePointer` [private]

String pointer to the actual interface specified by the user in the local/multipoint approximation specifications.

This pointer is not used for building objects since this is managed in SurrLayeredModels. Its use in `ApproximationInterface` is currently limited to header output.

6.8.2.3 **DakotaArray** `<DakotaApproximation>` `ApproximationInterface::functionSurfaces` [private]

List of approximations, one per response function.

This formulation allows the use of mixed approximations (i.e., different approximations used for different response functions), although the input specification is not currently general enough to support it.

The documentation for this class was generated from the following files:

- `ApproximationInterface.H`
- `ApproximationInterface.C`
6.9 BaseConstructor Struct Reference

Dummy struct for overloading letter-envelope constructors.

#include <ProblemDescDB.H>

Public Methods

- BaseConstructor (int=0)
  
  C++ structs can have constructors.

6.9.1 Detailed Description

Dummy struct for overloading letter-envelope constructors.

BaseConstructor is used to overload the constructor for the base class portion of letter objects. It avoids
infinite recursion (Coplien p.139) in the letter-envelope idiom by preventing the letter from instantiating
another envelope. Putting this struct here (rather than in a header of a class that uses it) avoids problems
with circular dependencies.

The documentation for this struct was generated from the following file:

- ProblemDescDB.H
6.10 BranchBndStrategy Class Reference

Strategy for mixed integer nonlinear programming using the PICO parallel branch and bound engine.

```c
#include <BranchBndStrategy.H>
```

Inheritance diagram for BranchBndStrategy:

```
DakotaStrategy
    
BranchBndStrategy
```

Public Methods

- **BranchBndStrategy** (ProblemDescDB &problem_db)
  constructor.

- **~BranchBndStrategy**()
  destructor.

- **void run_strategy()**
  Performs the branch and bound strategy by executing selectedIterator on userDefinedModel multiple times in parallel for different variable bounds within the model.

Private Attributes

- **DakotaModel userDefinedModel**
  the model used by the iterator.

- **DakotaIterator selectedIterator**
  the iterator used by BranchBndStrategy.

- **int numIteratorServers**
  number of concurrent iterator partitions.

- **int numRootSamples**
  number of samples to perform at the root of the branching structure.

- **int numNodeSamples**
  number of samples to perform at each node of the branching structure.

- **MPI_Comm picoComm**
  MPI intracommuicator for PICO hub processors (strategy and iterator masters).
6.10 BranchBndStrategy Class Reference

- **int picoCommRank**
  
  processor rank in picoComm.

- **int picoCommSize**
  
  number of processors in picoComm.

- **int argC**
  
  dummy argument count passed to pico classes in init(), readAll(), and readAndBroadcast().

- **char ** argV**
  
  dummy argument vector passed to pico classes in init(), readAll(), and readAndBroadcast().

- **DoubleVector picoLowerBnds**
  
  global lower bounds for merged continuous & discrete design variables passed to PICO (copied from user-DefinedModel).

- **DoubleVector picoUpperBnds**
  
  global upper bounds for merged continuous & discrete design variables passed to PICO (copied from user-DefinedModel).

- **IntVector picoListOfIntegers**
  
  key to the discrete variables which have been relaxed and merged into the continuous variables and bounds arrays (indices in the combined arrays).

### 6.10.1 Detailed Description

Strategy for mixed integer nonlinear programming using the PICO parallel branch and bound engine.

This strategy combines the PICO branching engine with nonlinear programming optimizers from DAKOTA (e.g., DOT, NPSOL, OPT++) to solve mixed integer nonlinear programs. The discrete variables in the problem must support relaxation, i.e., they must be able to assume nonintegral values during the solution process. PICO selects solution “branches”, each of which constrains the problem to lie within different variable bounds. The series of branches selected is designed to drive integer variables to their integral values. For each of the branches, a nonlinear DAKOTA optimizer is used to solve the optimization problem and return the solution to PICO. If this solution has all of the integer variables at integral values, then it provides an upper bound on the true solution. This bound can be used to prune other branches, since there is no need to further investigate a branch which does not yet have integral values for the integer variables and which has an objective function worse than the bound. In linear programs, the bounding and pruning processes are rigorous and will lead to the exact global optimum. In nonlinear problems, the bounding and pruning processes are heuristic, i.e. they will find local optima but the global optimum may be missed. PICO supports parallelism between “hubs,” each of which drives a concurrent iterator partition in DAKOTA (and each of these iterator partitions may have lower levels of nested parallelism). This complexity is hidden from PICO through the use of picoComm, which contains the set of master iterator processors, one from each iterator partition. Thus, PICO can schedule jobs among single-processor hubs in its normal manner, unaware of the nested parallelism complexities that may occur within each nonlinear optimization.

The documentation for this class was generated from the following files:

- BranchBndStrategy.H
- BranchBndStrategy.C
6.11 CommandLineHandler Class Reference

Utility class for managing command line inputs to DAKOTA.

#include <CommandLineHandler.H>

Inheritance diagram for CommandLineHandler:

```
CommandLineHandler
    ^
    | GetLongOpt
```

Public Methods

- **CommandLineHandler ()**
  constructor.

- **~CommandLineHandler ()**
  destructor.

- **void check_usage (int argc, char **argv)**
  Verifies that DAKOTA is called with the correct command usage. Prints a descriptive message and exits the program if incorrect.

- **const char * read_restart_stream ()**
  Returns the filename of the restart file as specified on the DAKOTA command line.

- **const char * write_restart_stream ()**
  Returns the filename of the restart file as specified on the DAKOTA command line.

- **int read_restart_evals ()**
  Returns the number of evaluations to be read from the restart file as specified on the DAKOTA command line.

6.11.1 Detailed Description

Utility class for managing command line inputs to DAKOTA.

CommandLineHandler provides additional functionality that is specific to DAKOTA's needs for the definition and parsing of command line options. Inheritance is used to allow the class to have all the functionality of the base class, GetLongOpt.

The documentation for this class was generated from the following files:

- CommandLineHandler.H
- CommandLineHandler.C
6.12 CommandShell Class Reference

Utility class which defines convenience operators for spawning processes with system calls.

```cpp
#include <CommandShell.H>
```

Public Methods

- **CommandShell ()**
  constructor.
- **~CommandShell ()**
  destructor.
- **CommandShell & operator<< (const char *string)**
  adds string to unixCommand.
- **CommandShell & operator<<(CommandShell &f)(CommandShell &)**
  allows passing of the flush function to the shell using `<<`.
- **CommandShell & flush ()**
  "flushes" the shell; i.e. executes the unixCommand.
- **void async_flag (const short flag)**
  set the asyncFlag.
- **short async_flag () const**
  get the asyncFlag.
- **void quiet_flag (const short flag)**
  set the quietFlag.
- **short quiet_flag () const**
  get the quietFlag.

Private Attributes

- **DakotaString unixCommand**
  the command string that is constructed through one or more `<<` insertions and then executed by flush.
- **short asyncFlag**
  flags nonblocking operation (background system calls).
- **short quietFlag**
  flags quiet operation (no command echo).
6.12.1 Detailed Description

Utility class which defines convenience operators for spawning processes with system calls.
The CommandShell class wraps the C system() utility and defines convenience operators for building a
command string and then passing it to the shell.

6.12.2 Member Function Documentation

6.12.2.1 CommandShell & CommandShell::flush ()

“flushes” the shell; i.e. executes the unixCommand.
Executes the unixCommand by passing it to system(). Appends an “&” if asynchFlag is set (background
system call) and echos the unixCommand to cout if quietFlag is not set.

The documentation for this class was generated from the following files:

- CommandShell.H
- CommandShell.C
6.13 ConcurrentStrategy Class Reference

Strategy for multi-start iteration or pareto set optimization.

```cpp
#include <ConcurrentStrategy.H>
```

Inheritance diagram for ConcurrentStrategy::

```
public
```

### Public Methods

- **ConcurrentStrategy (ProblemDescDB &problem_db)**
  
  constructor.

- **~ConcurrentStrategy ()**
  
  destructor.

- **void run_strategy ()**
  
  Performs the concurrent strategy by executing selectedIterator on userDefinedModel multiple times in parallel for different settings within the iterator or model.

### Private Attributes

- **DakotaModel userDefinedModel**
  
  the model used by the iterator.

- **DakotaIterator selectedIterator**
  
  the iterator used by the concurrent strategy.

- **int numIteratorServers**
  
  number of concurrent iterator partitions.

- **int numIteratorJobs**
  
  total number of iterator executions to schedule over the servers.

- **DakotaRealVectorArray iteratorParameterSets**
  
  an array of parameter set vectors (either multistart variable sets or pareto multiobjective weighting sets) to be performed.

- **short strategyDedicatedMasterFlag**
  
  signals ded. master partitioning.
6.13.1 Detailed Description

Strategy for multi-start iteration or pareto set optimization.

This strategy maintains two concurrent iterator capabilities. First, a general capability for running an iterator multiple times from different starting points is provided (often used for multi-start optimization, but not restricted to optimization). Second, a simple capability for mapping the "pareto frontier" (the set of optimal solutions in multiobjective formulations) is provided. This pareto set is mapped through running an optimizer multiple times for different sets of multiobjective weightings.

The documentation for this class was generated from the following files:

- ConcurrentStrategy.H
- ConcurrentStrategy.C
6.14 CONMINOptimizer Class Reference

Wrapper class for the CONMIN optimization library.

```cpp
#include <CONMINOptimizer.H>
```

Inheritance diagram for CONMINOptimizer:

```
DakotaIterator
  
DakotaOptimizer
  
CONMINOptimizer
```

Public Methods

- **CONMINOptimizer** (DakotaModel &model)
  
  *constructor.*

- **~CONMINOptimizer** ()
  
  *destructor.*

- **void findOptimum** ()
  
  *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

Private Methods

- **void allocateWorkspace** ()
  
  *Allocates workspace for the optimizer.*

Private Attributes

- **int conminInfo**
  
  *INFO from CONMIN manual.*

- **int printControl**
  
  *IPRINT from CONMIN manual (controls output verbosity).*

- **int optimizationType**
  
  *MINMAX from DOT manual (minimize or maximize).*
- **DakotaRealVector** `localConstraintValues`
  array of nonlinear constraint values passed to CONMIN.

- **DakotaSizeList** `constraintMappingIndices`
  a list of indices for referencing the corresponding DakotaResponse constraints used in computing the CONMIN constraints.

- **DakotaRealList** `constraintMappingMultipliers`
  a list of multipliers for mapping the DakotaResponse constraints to the CONMIN constraints.

- **DakotaRealList** `constraintMappingOffsets`
  a list of offsets for mapping the DakotaResponse constraints to the CONMIN constraints.

- **int** `N1`
  Size variable for CONMIN arrays. See CONMIN manual.

- **int** `N2`
  Size variable for CONMIN arrays. See CONMIN manual.

- **int** `N3`
  Size variable for CONMIN arrays. See CONMIN manual.

- **int** `N4`
  Size variable for CONMIN arrays. See CONMIN manual.

- **int** `N5`
  Size variable for CONMIN arrays. See CONMIN manual.

- **int** `NFDG`
  Finite difference flag.

- **int** `IPRINT`
  Flag to control amount of output data.

- **int** `ITMAX`
  Flag to specify the maximum number of iterations.

- **Real** `FDCH`
  Relative finite difference step size.

- **Real** `FDCHM`
  Absolute finite difference step size.

- **Real** `CT`
  Constraint thickness parameter.

- **Real** `CTMIN`
  Minimum absolute value of CT used during optimization.

- **Real** `CTL`
Constraint thickness parameter for linear and side constraints.

- Real **CTLMIN**
  Minimum value of CTL used during optimization.

- Real **DELFUN**
  Relative convergence criterion threshold.

- Real **DABFUN**
  Absolute convergence criterion threshold.

- Real * **conminDesVars**
  Array of design variables used by CONMIN (length N1 = numdv+2).

- Real * **conminLowerBnds**
  Array of lower bounds used by CONMIN (length N1 = numdv+2).

- Real * **conminUpperBnds**
  Array of upper bounds used by CONMIN (length N1 = numdv+2).

- Real **S**
  Internal CONMIN array.

- Real **G1**
  Internal CONMIN array.

- Real **G2**
  Internal CONMIN array.

- Real **B**
  Internal CONMIN array.

- Real **C**
  Internal CONMIN array.

- int * **MS1**
  Internal CONMIN array.

- Real * **SCAL**
  Internal CONMIN array.

- Real * **DF**
  Internal CONMIN array.

- Real * **A**
  Internal CONMIN array.

- int * **ISC**
  Internal CONMIN array.
6.14.1 Detailed Description

Wrapper class for the CONMIN optimization library.

The CONMINOptimizer class provides a wrapper for CONMIN, a Public-domain Fortran 77 optimization library written by Gary Vanderplaats under contract to NASA Ames Research Center. The CONMIN User’s Manual is contained in NASA Technical Memorandum X-62282, 1978. CONMIN uses a reverse communication mode, which avoids the static function and static attribute issues that arise with function pointer designs (see NPSOLOptimizer and SNLLOptimizer).

The user input mappings are as follows: max_iterations is mapped into CONMIN’s ITMAX parameter, max_function_evaluations is implemented directly in the find optimum() loop since there is no CONMIN parameter equivalent, convergence_tolerance is mapped into CONMIN’s DELFUN and DABFUN parameters, output verbosity is mapped into CONMIN’s IPRINT parameter (verbose: IPRINT = 4; quiet: IPRINT = 2), gradient mode is mapped into CONMIN’s NFDG parameter, and finite difference step size is mapped into CONMIN’s FDCH and FDCHM parameters. Refer to [Vanderplaats, 1978] for additional information on CONMIN parameters.

6.14.2 Member Data Documentation

6.14.2.1 int CONMINOptimizer::conminInfo [private]
INFO from CONMIN manual.
Information requested by CONMIN: 1 = evaluate objective and constraints, 2 = evaluate gradients of objective and constraints.

6.14.2.2 int CONMINOptimizer::printControl [private]
IPRINT from CONMIN manual (controls output verbosity).
Values range from 0 (nothing) to 4 (most output). 0 = nothing, 1 = initial and final function information, 2 = all of #1 plus function value and design vars at each iteration, 3 = all of #2 plus constraint values and direction vectors, 4 = all of #3 plus gradients of the objective function and constraints, 5 = all of #4 plus proposed design vector, plus objective and constraint functions from the 1-D search

6.14.2.3 int CONMINOptimizer::optimizationType [private]
MINMAX from DOT manual (minimize or maximize).
Values of 0 or -1 (minimize) or 1 (maximize).

6.14.2.4 DakotaRealVector CONMINOptimizer::localConstraintValues [private]
array of nonlinear constraint values passed to CONMIN.
This array must be of nonzero length (sized with localConstraintArraySize) and must contain only one-sided inequality constraints which are $\leq 0$ (which requires a transformation from 2-sided inequalities and equalities).

### 6.14.2.5 DakotaSizeList CONMINOptimizer::constraintMappingIndices [private]

A list of indices for referencing the corresponding DakotaResponse constraints used in computing the CONMIN constraints.

The length of the list corresponds to the number of CONMIN constraints, and each entry in the list points to the corresponding DAKOTA constraint.

### 6.14.2.6 DakotaRealList CONMINOptimizer::constraintMappingMultipliers [private]

A list of multipliers for mapping the DakotaResponse constraints to the CONMIN constraints.

The length of the list corresponds to the number of CONMIN constraints, and each entry in the list contains a multiplier for the DAKOTA constraint identified with constraintMappingIndices. These multipliers are currently +1 or -1.

### 6.14.2.7 DakotaRealList CONMINOptimizer::constraintMappingOffsets [private]

A list of offsets for mapping the DakotaResponse constraints to the CONMIN constraints.

The length of the list corresponds to the number of CONMIN constraints, and each entry in the list contains an offset for the DAKOTA constraint identified with constraintMappingIndices. These offsets involve inequality bounds or equality targets, since CONMIN assumes constraint allowables = 0.

### 6.14.2.8 int CONMINOptimizer::N1 [private]

Size variable for CONMIN arrays. See CONMIN manual.

$N1 = \text{number of variables} + 2$

### 6.14.2.9 int CONMINOptimizer::N2 [private]

Size variable for CONMIN arrays. See CONMIN manual.

$N2 = \text{number of constraints} + 2*\text{(number of variables)}$

### 6.14.2.10 int CONMINOptimizer::N3 [private]

Size variable for CONMIN arrays. See CONMIN manual.

$N3 = \text{Maximum possible number of active constraints}$

### 6.14.2.11 int CONMINOptimizer::N4 [private]

Size variable for CONMIN arrays. See CONMIN manual.

$N4 = \text{Maximum}(N3, \text{number of variables})$
6.14.2.12 int CONMINOptimizer::N5 [private]

Size variable for CONMIN arrays. See CONMIN manual.
N5 = 2*(N4)

6.14.2.13 Real CONMINOptimizer::CT [private]

Constraint thickness parameter.
The value of CT decreases in magnitude during optimization.

6.14.2.14 Real* CONMINOptimizer::S [private]

Internal CONMIN array.
Move direction in N-dimensional space.

6.14.2.15 Real* CONMINOptimizer::G1 [private]

Internal CONMIN array.
Temporary storage of constraint values.

6.14.2.16 Real* CONMINOptimizer::G2 [private]

Internal CONMIN array.
Temporary storage of constraint values.

6.14.2.17 Real* CONMINOptimizer::B [private]

Internal CONMIN array.
Temporary storage for computations involving array S.

6.14.2.18 Real* CONMINOptimizer::C [private]

Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.14.2.19 int* CONMINOptimizer::MS1 [private]

Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.14.2.20 Real* CONMINOptimizer::SCAL [private]

Internal CONMIN array.
Vector of scaling parameters for design parameter values.
6.14.2.21 Real* CONMINOptimizer::DF [private]

Internal CONMIN array.
Temporary storage for analytic gradient data.

6.14.2.22 Real* CONMINOptimizer::A [private]

Internal CONMIN array.
Temporary 2-D array for storage of constraint gradients.

6.14.2.23 int* CONMINOptimizer::ISC [private]

Internal CONMIN array.
Array of flags to identify linear constraints. (not used in this implementation of CONMIN)

6.14.2.24 int* CONMINOptimizer::IC [private]

Internal CONMIN array.
Array of flags to identify active and violated constraints

The documentation for this class was generated from the following files:

- CONMINOptimizer.H
- CONMINOptimizer.C
6.15  CtelRegexp Class Reference

#include <CtelRegExp.H>

Public Types

- enum RStatus { GOOD = 0, EXP_TOO_BIG, OUT_OF_MEM, TOO_MANY_PAR, UNMATCH_PAR, STARPLUS_EMPTY, STARPLUS_NESTED, INDEX_RANGE, INDEX_-MATCH, STARPLUS NOTHING, TRAILING, INT_ERROR, BAD_PARAM, BAD_OPCODE }

   Error codes reported by the engine - Most of these codes never really occurs with this implementation.

Public Methods

- CtelRegExp (const std::string &pattern)
  Constructor - compile a regular expression.

- ~CtelRegExp ()
  Destructor.

- bool compile (const std::string &pattern)
  Compile a new regular expression.

- std::string match (const std::string &str)
  matches a particular string; this method returns a string that is a sub-string matching with the regular expression.

- bool match (const std::string &str, size_t *start, size_t *size)
  another form of matching; returns the indexes of the matching.

- RStatus getStatus ()
  Get status.

- const std::string & getStatusMsg ()
  Get status message.

- void clearErrors ()
  Clear all errors.

- const std::string & getRe ()
  Return regular expression pattern.

- bool split (const std::string &str, std::vector<std::string> &all_matches)
  Split.
Private Methods

- **CtelRegexp** (const CtelRegexp &)
  
  *Private copy constructor.*

- **CtelRegexp & operator=** (const CtelRegexp &)
  
  *Private assignment operator.*

Private Attributes

- std::string **strPattern**
  
  *STL string to hold pattern.*

- regexp * **r**
  
  *Pointer to regexp.*

- **RStatus status**
  
  *Return status, enumerated type.*

- std::string **statusMsg**
  
  *STL string to hold status message.*

6.15.1 Detailed Description

DESCRIPTION: Wrapper for the Regular Expression engine( regexp ) released by Henry Spencer of the University of Toronto.

The documentation for this class was generated from the following files:

- CtelRegExp.H
- CtelRegExp.C
6.16 DACEIterator Class Reference

Wrapper class for the DDACE design of experiments library.

```cpp
#include <DACEIterator.H>
```

Inheritance diagram for DACEIterator::

```
DakotaIterator

DACEIterator
```

**Public Methods**

- **DACEIterator (DakotaModel &model)**
  
  *primary constructor for building a standard iterator.*

- **DACEIterator (DakotaModel &model, int samples, int symbols, int seed, const DakotaString &sampling_method)**
  
  *alternate constructor for an iterator used for building approximations (inactive).*

- **~DACEIterator ()**
  
  *destructor.*

- **void run_iterator ()**
  
  *run the iterator.*

- **const DakotaVariables & iterator_variable_results () const**
  
  *return the final iterator solution (variables).*

- **const DakotaResponse & iterator_response_results () const**
  
  *return the final iterator solution (response).*

- **void print_iterator_results (ostream &s) const**
  
  *print the final iterator results.*

- **void sampling_reset (int min_samples, short all_data_flag, short stats_flag) const**
  
  *reset sampling iterator.*

- **const DakotaString & sampling_scheme () const**
  
  *return sampling name.*

- **void update_best (const DakotaRealVector &vars, const DakotaResponse &response, const int eval_num) const**
  
  *compares current evaluation to best evaluation and updates best.*
Private Methods

- void resolve_samples_symbols()
  convenience function for resolving number of samples and number of symbols from input.

Private Attributes

- DakotaString daceMethod
  oas, lhs, oa_lhs, random, box_behnken_design, central_composite_design, or grid.

- int numSamples
  number of samples to be evaluated.

- int numSymbols
  number of symbols to be used in generating the sample set (inversely related to number of replications).

- int randomSeed
  seed for the random number generator (allows repeatability of results).

- short allDataFlag
  flag which triggers the update of allV ars/allResponses for use by DakotaIterator::allvariables() and DakotaIterator::allResponses().

- DakotaVariables bestVariables
  best variables found during the study.

- DakotaResponse bestResponses
  best responses found during the study.

- Real bestObjectiveFn
  best objective function found during the study.

- Real bestViolations
  best constraint violations found during the study. In the current approach, constraint violation reduction takes strict precedence over objective function reduction.

- size_t numObjectiveFunctions
  number of objective functions. Used in update_best.

- size_t numNonlinearIneqConstraints
  number of nonlinear inequality constraints. Used in update_best.

- size_t numNonlinearEqConstraints
  number of nonlinear equality constraints. Used in update_best.

- DakotaRealVector multiObjWeights
  vector of multiobjective weights. Used in update_best.

- DakotaRealVector nonlinearIneqLowerBnds
6.16.1 Detailed Description

Wrapper class for the DDACE design of experiments library.

The DACEIterator class provides a wrapper for DDACE, a C++ design of experiments library from the Computational Sciences and Mathematics Research (CSMR) department at Sandia’s Livermore CA site. This class uses design and analysis of computer experiments (DACE) methods to sample the design space spanned by the bounds of a DakotaModel. It returns all generated samples and their corresponding responses as well as the best sample found.

6.16.2 Constructor & Destructor Documentation

6.16.2.1 DACEIterator::DACEIterator (DakotaModel & model)

primary constructor for building a standard iterator.
This constructor is called for a standard iterator built with data from probDescDB.

6.16.2.2 DACEIterator::DACEIterator (DakotaModel & model, int samples, int symbols, int seed, const DakotaString & sampling_method)

alternate constructor for an iterator used for building approximations (inactive).
This constructor is currently inactive, since the old DACEIterator instantiations within ApproximationInterface have been replaced with more general facilities within LayeredModel.

6.16.3 Member Function Documentation

6.16.3.1 void DACEIterator::run iterator () [virtual]

run the iterator.
This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.
Reimplemented from Dakotalterator.
6.16.3.2 void DACEIterator::resolve_samples_symbols () [private]

convenience function for resolving number of samples and number of symbols from input.

This function must define a combination of samples and symbols that is acceptable for a particular sampling algorithm. Users provide requests for these quantities, but this function must enforce any restrictions imposed by the sampling algorithms.

The documentation for this class was generated from the following files:

- DACEIterator.H
- DACEIterator.C
6.17 DakotaApproximation Class Reference

Base class for the approximation class hierarchy.

```cpp
#include <DakotaApproximation.H>
```

Inheritance diagram for DakotaApproximation::

```
DakotaApproximation

| ANNsurf | HermiteSurf | KrigingSurf | MARSSurf | RespSurf | TaylorSurf |
```

Public Methods

- **DakotaApproximation ()**
  default constructor.

- **DakotaApproximation (const DakotaString &approx_type, const ProblemDescDB &problem_db)**
  standard constructor for envelope.

- **DakotaApproximation (const DakotaApproximation &approx)**
  copy constructor.

- virtual **~DakotaApproximation ()**
  destructor.

- **DakotaApproximation operator= (const DakotaApproximation &approx)**
  assignment operator.

- virtual **void find_coefficients ()**
  calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

- virtual **Real get_value (const DakotaRealVector &x)**
  retrieve the approximate function value for a given parameter vector.

- virtual **const DakotaRealVector & get_gradient (const DakotaRealVector &x)**
  retrieve the approximate function gradient for a given parameter vector.

- virtual **int required_samples (int num_vars)**
  return the minimum number of samples required to build the derived class approximation type in num-vars dimensions.

- **void build (int nv, int ns, const DakotaRealVectorArray &vars_samples, const DakotaRealVector &fn_samples, const DakotaRealVectorArray &grad_samples)**
  build the surface from scratch. Populates currentPoints and invokes find_coefficients().
void add_point_rebuild (const DakotaRealVector &x, const Real &f, const DakotaRealVector &grad_f)
    add a new point to the approximation and rebuild it.

void set_bounds (const DakotaRealVector &lower, const DakotaRealVector &upper)
    set approximation lower and upper bounds (currently only used by graphics).

void draw_surface ()
    render the approximate surface using the 3D graphics (2 variable problems only).

const int & num_variables () const
    return the number of variables used in the approximation.

Protected Methods

DakotaApproximation (BaseConstructor, const ProblemDescDB &problem_db)
    constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

Protected Attributes

int numVars
    number of variables in the approximation.

int numCurrentPoints
    number of points in the currentPoints list.

int numSamples
    number of samples passed to build() to construct the approximation.

short gradientFlag
    flag signaling the use of gradient data in global approximation builds as indicated by the user’s use_gradients specification.

DakotaRealVector gradVector
    gradient of the approximation with respect to the variables.

DakotaList< SurrogateDataPoint > currentPoints
    list of samples used to build the approximation.

DakotaString approxType
    approximation type (long form for diagnostic I/O).
Private Methods

- DakotaApproximation * getApprox (const DakotaString &approxType, const ProblemDescDB &probdb)
  
  *Used only by the envelope constructor to initialize approxRep to the appropriate derived type.*

- void addPoint (const DakotaRealVector &x, const Real &f, const DakotaRealVector &grad f)
  
  *add a new point to the approximation (used by build & addPoint, rebuild).*

Private Attributes

- DakotaRealVector approxLowerBounds
  
  *approximation lower bounds (used only by 3D graphics).*

- DakotaRealVector approxUpperBounds
  
  *approximation upper bounds (used only by 3D graphics).*

- DakotaApproximation * approxRep
  
  *pointer to the letter (initialized only for the envelope).*

- int referenceCount
  
  *number of objects sharing approxRep.*

6.17.1 Detailed Description

Base class for the approximation class hierarchy.

The DakotaApproximation class is the base class for the data fit surrogate class hierarchy in DAKOTA. One instance of a DakotaApproximation must be created for each function to be approximated (a vector of DakotaApproximations is contained in ApproximationInterface). For memory efficiency and enhanced polymorphism, the approximation hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaApproximation) serves as the envelope and one of the derived classes (selected in DakotaApproximation::getApproximation()) serves as the letter.

6.17.2 Constructor & Destructor Documentation

6.17.2.1 DakotaApproximation::DakotaApproximation ()

default constructor.

The default constructor is used in List<DakotaApproximation> instantiations. approxRep is NULL in this case (problemDB is needed to build a meaningful DakotaModel object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.
6.17.2.2 DakotaApproximation::DakotaApproximation (const DakotaString & approx_type, const ProblemDescDB & problem_db)

standard constructor for envelope.
Envelope constructor only needs to extract enough data to properly execute get approx, since Dakota-Approximation(BaseConstructor, problem_db) builds the actual base class data for the derived approximations.

6.17.2.3 DakotaApproximation::DakotaApproximation (const DakotaApproximation & approx)

copy constructor.
Copy constructor manages sharing of approxRep and incrementing of referenceCount.

6.17.2.4 DakotaApproximation::~DakotaApproximation () [virtual]
destructor.
Destructor decrements referenceCount and only deletes approxRep when referenceCount reaches zero.

6.17.2.5 DakotaApproximation::DakotaApproximation (BaseConstructor, const ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor is the one which must build the base class data for all derived classes. get approx() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling get approx() again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ~Dakota-Approximation).

6.17.3 Member Function Documentation

6.17.3.1 DakotaApproximation DakotaApproximation::operator= (const DakotaApproximation & approx)

assignment operator.

6.17.3.2 DakotaApproximation * DakotaApproximation::get approx (const DakotaString & approx_type, const ProblemDescDB & problem_db) [private]

Used only by the envelope constructor to initialize approxRep to the appropriate derived type.
Used only by the envelope constructor to initialize approxRep to the appropriate derived type, as given by the approx_type parameter.
The documentation for this class was generated from the following files:

- DakotaApproximation.H
- DakotaApproximation.C
6.18 DakotaArray Class Template Reference

Template class for the Dakota bookkeeping array.

#include <DakotaArray.H>

Public Methods

- **DakotaArray ()**
  Default constructor.

- **DakotaArray (size_t size)**
  Constructor which takes an initial size.

- **DakotaArray (size_t size, const T &initial_val)**
  Constructor which takes an initial size and an initial value.

- **DakotaArray (const DakotaArray<T> &a)**
  Copy constructor.

- **DakotaArray (const T *p, size_t size)**
  Constructor, creates array of size, with initial value <T> p.

- **~DakotaArray ()**
  Destructor.

- **T & operator[] (ptrdiff_t i)**
  Index operator, returns the ith value of the array.

- **T & operator[] (size_t i)**
  Index operator, returns the ith value of the array.

- **const T & operator[] (ptrdiff_t i) const**
  Index operator const, returns the ith value of the array.

- **const T & operator[] (size_t i) const**
  Index operator const, returns the ith value of the array.

- **T & operator() (ptrdiff_t i)**
  Index operator, not bounds checked.

- **T & operator() (size_t i)**
  Index operator, not bounds checked.

- **const T & operator() (ptrdiff_t i) const**
  Index operator const, not bounds checked.
- `const T & operator[](size_t i) const`
  
  Index operator const, not bounds checked.

- `size_t length() const`
  
  Returns size of array.

- `void reshape(size_t sz)`
  
  Resizes array to size sz.

- `const T * data() const`
  
  Returns pointer to continuous data.

- `void testClass()`
  
  Class unit test method.

- `DakotaArray< T > & operator=(const DakotaArray< T > &a)`
  
  Normal const assignment operator.

- `DakotaArray< T > & operator=(DakotaArray< T > &a)`
  
  Normal assignment operator.

- `DakotaArray< T > & operator=(const T &ival)`
  
  Sets all elements in self to the value ival.

- `operator T * () const`
  
  Converts the DakotaArray to a standard C-style array. Use with care!

### Private Methods

- `void copy_array (const T *p, size_t size)`
  
  Deep copies the array pointed to by \{p\} into this array.

### 6.18.1 Detailed Description

`template<class T> class DakotaArray< T >`

Template class for the Dakota bookkeeping array.

An array class template that provides additional functionality that is specific to Dakota’s needs. The DakotaArray class adds additional functionality needed by Dakota to the inherited base array class. The DakotaArray class can inherit from either the STL or RW vector classes.

### 6.18.2 Constructor & Destructor Documentation
6.18.2.1 template <class T> DakotaArray < T >::DakotaArray (const T * p, size_t size)

Constructor, creates array of size, with initial value <T> p.
Assigns up to size values in array to p. Calls the private method copyArray

6.18.3 Member Function Documentation

6.18.3.1 template <class T> const T * DakotaArray < T >::data () const

Returns pointer T * to continuous data.
Returns a c style pointer to the data within the array. USE WITH CARE. Needed to mimick RW vector class, is used in the operator(). Uses the STL front method.

6.18.3.2 template <class T> void DakotaArray < T >::testClass ()

Class unit test method.
Unit test method for the DakotaArray class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an unexpected answer is received.

6.18.3.3 template <class T> DakotaArray < T > & DakotaArray < T >::operator= (const T & ival)

Sets all elements in self to the value ival.
Assigns all values of array to the value passed in as ival. For the Rogue Wave case utilizes base class operator=(ival), while for the ANSI case uses the STL assign() method.

6.18.3.4 template <class T> DakotaArray < T >::operator T * () const

Converts the {DakotaArray} to a standard C-style array. Use with care!
The operator() returns a c style pointer to the data within the array. Calls the data() method. USE WITH CARE.

6.18.3.5 template <class T> void DakotaArray < T >::copyArray (const T * p, size_t size)
[private]

Deep copies the array pointed to by {p} into this array.
Copy an array of type T into the DakotaArray Private function for {operator=(const T *p)} and the constructor {DakotaArray(const T * p, size_t size)}.
The documentation for this class was generated from the following file:

- DakotaArray.H
6.19 DakotaBaseVector Class Template Reference

Base class for the DakotaMatrix and DakotaVector classes.
#include <DakotaBaseVector.H>
Inheritance diagram for DakotaBaseVector::

```
DakotaBaseVector
```

Public Methods

- DakotaBaseVector ()
  Default constructor.

- DakotaBaseVector (size_t size)
  Constructor, creates vector of size.

- DakotaBaseVector (size_t size, const T &initial_val)
  Constructor, creates vector of size with initial value of initial_val.

- ~DakotaBaseVector ()
  Destructor.

- DakotaBaseVector (const DakotaBaseVector<T> &a)
  Default copy constructor.

- DakotaBaseVector (const T *p, size_t size)
  Alternate copy constructor.

- DakotaBaseVector<T> & operator= (const DakotaBaseVector<T> &a)
  Normal assignment operator.

- DakotaBaseVector<T> & operator= (const T &ival)
  Assigns all values of vector to ival.

- T & operator[] (ptrdiff_t i)
  Returns the object at index i, (can use as lvalue).

- T & operator[] (size_t i)
  Returns the object at index i, (can use as lvalue).

- const T & operator[] (ptrdiff_t i) const

```
6.19 DakotaBaseVector Class Template Reference

Returns the object at index i, const (can’t use as lvalue).

- const T & operator[](size_t i) const
  Returns the object at index i, const (can’t use as lvalue).

- T & operator() (ptrdiff_t i)
  Index operator, not bounds checked.

- T & operator() (size_t i)
  Index operator, not bounds checked.

- const T & operator() (ptrdiff_t i) const
  Index operator const, not bounds checked.

- const T & operator() (size_t i) const
  Index operator const, not bounds checked.

- size_t length () const
  Returns size of vector.

- void reshape (size_t sz)
  Resizes vector to size sz.

- const T * data () const
  Returns pointer to standard C array use with care.

Protected Attributes

- T * array_
  Protected data member to hold pointer to front of vector.

- size_t npts_
  Protected data member which holds number of points in array.

6.19.1 Detailed Description

**template<class T> class DakotaBaseVector<T>**

Base class for the DakotaMatrix and DakotaVector classes.

The DakotaBaseVector class is the base class for the DakotaMatrix class. It is used to define a common vector interface for both the STL and RW vector classes. If the STL version is based on the valarray class then some basic vector operations such as +, * are available.

6.19.2 Constructor & Destructor Documentation
6.19.2.1 template<class T> DakotaBaseVector<T>::DakotaBaseVector (size_t size, const T & initial_val)

Constructor, creates vector of size with initial value of initial_val.
Constructor which takes an initial size and an initial value, allocates an area of initial size and initializes it with input value. Calls base class constructor

6.19.3 Member Function Documentation

6.19.3.1 template<class T> void DakotaBaseVector<T>::reshape (size_t sz)

Resizes vector to size sz.
Resizes the array to size sz by calling the STL resize method, and sets the private data member npts_equal to sz. Needed to mimic the RW vector class

6.19.3.2 template<class T> const T * DakotaBaseVector<T>::data () const

Returns pointer to standard C array use with care.
Returns a c style pointer to the data within the array. USE WITH CARE. Needed to mimic RW vector class.
The documentation for this class was generated from the following file:

- DakotaBaseVector.H
6.20 DakotaBiStream Class Reference

The binary input stream class. Overloads the >> operator for all data types.

```c
#include <DakotaBinStream.H>
```

Public Methods

- **DakotaBiStream()**
  Default constructor, need to open.

- **DakotaBiStream(const char *s)**
  Constructor takes name of input file.

- **DakotaBiStream(const char *s, int flags)**
  Constructor takes name of input file, flags.

- **~DakotaBiStream()**
  Destructor, calls xdr_destroy to delete xdr stream.

- **DakotaBiStream & operator>>(DakotaString &)**
  Binary Input stream operator>>.

- **DakotaBiStream & operator>>(char *)**
  Input operator, reads char* from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(char &)**
  Input operator, reads char from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(int &)**
  Input operator, reads int* from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(long &)**
  Input operator, reads long from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(short &)**
  Input operator, reads short from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(double &)**
  Input operator, reads double from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(float &)**
  Input operator, reads float from binary stream DakotaBiStream.

- **DakotaBiStream & operator>>(unsigned char &)**
  Input operator, reads unsigned char* from binary stream DakotaBiStream.
- DakotaBiStream & operator>>(unsigned int &)
  \[ \text{Input operator, reads unsigned int from binary stream DakotaBiStream.} \]

- DakotaBiStream & operator>>(unsigned long &)
  \[ \text{Input operator, reads unsigned long from binary stream DakotaBiStream.} \]

- DakotaBiStream & operator>>(unsigned short &)
  \[ \text{Input operator, reads unsigned short from binary stream DakotaBiStream.} \]

**Private Attributes**

- XDR xdrInBuf
  \[ XDR input stream buffer. \]

- char inBuf [256]
  \[ Buffer to hold data as it is read in. \]

### 6.20.1 Detailed Description

The binary input stream class. Overloads the >> operator for all data types.

The DakotaBiStream class is a binary input class which overloads the >> operator for all standard data types (int, char, float, etc). The class relies on the methods within the ifstream base class. The DakotaBiStream class inherits from the ifstream class. The class also utilizes rpc/xdr to construct machine independent binary files. The Dakota restart files can now be moved from host to host. These motivations to develop these classes was to replace the Rogue wave classes which Dakota historically used for binary I/O.

### 6.20.2 Constructor & Destructor Documentation

#### 6.20.2.1 DakotaBiStream::DakotaBiStream ()

Default constructor, need to open.

Default constructor, allocates xdr stream, but does not call the open method. The open method must be called before stream can be read.

#### 6.20.2.2 DakotaBiStream::DakotaBiStream (const char * s)

Constructor takes name of input file.

Constructor which takes a char* filename. Calls the base class open method with the filename and no other arguments. Also allocates the xdr stream.
6.20 DakotaBiStream Class Reference

6.20.2.3 DakotaBiStream::DakotaBiStream (const char * s, int flags)

Constructor takes name of input file, flags.
Constructor which takes a char* filename and int flags. Calls the base class open method with the filename
and flags as arguments. Also allocates xdr stream.

6.20.2.4 DakotaBiStream::~DakotaBiStream ()

Destructor, calls xdr_destroy to delete xdr stream.
Destructor, destroys the xdr stream allocated in constructor

6.20.3 Member Function Documentation

6.20.3.1 DakotaBiStream & DakotaBiStream::operator>>(DakotaString & ds)

Binary Input stream operator>>.
The DakotaString input operator must first read both the xdr buffer size and the size of the string written.
Once these are read it can then read and convert the DakotaString correctly.

6.20.3.2 DakotaBiStream & DakotaBiStream::operator>>(char * s)

Input operator, reads char* from binary stream DakotaBiStream.
Reading char array is a special case. The method has no way of knowing if the length to the input array
is large enough, it assumes it is one char longer than actual string, (Null terminator added). As with the
DakotaString the size of the xdr buffer as well as the char array size written must be read from the stream
prior to reading and converting the char array.
The documentation for this class was generated from the following files:

- DakotaBinStream.H
- DakotaBinStream.C
6.21 DakotaBoStream Class Reference

The binary output stream class. Overloads the $<<$ operator for all data types.

```cpp
#include <DakotaBinStream.H>
```

**Public Methods**

- **DakotaBoStream ()**
  
  Default constructor, need to open.

- **DakotaBoStream (const char *s)**
  
  Constructor takes name of input file.

- **DakotaBoStream (const char *s, int flags)**
  
  Constructor takes name of input file, flags.

- **~DakotaBoStream ()**
  
  Destructor, calls xdr_destroy to delete xdr stream.

- **void testClass ()**
  
  Performs unit testing for the DakotaBoStream class.

- **DakotaBoStream & operator<< (const DakotaString &)**
  
  Binary Output stream operator$<<$.

- **DakotaBoStream & operator<< (const char *)**
  
  Output operator, writes char* TO binary stream DakotaBoStream.

- **DakotaBoStream & operator<< (const char &c)**
  
  Output operator, writes char to binary stream DakotaBoStream.

- **DakotaBoStream & operator<< (const int &i)**
  
  Output operator, writes int to binary stream DakotaBoStream.

- **DakotaBoStream & operator<< (const long &l)**
  
  Output operator, writes long to binary stream DakotaBoStream.

- **DakotaBoStream & operator<< (const double &d)**
  
  Output operator, writes double to binary stream DakotaBoStream.

- **DakotaBoStream & operator<< (const short &s)**
  
  Output operator, writes short to binary stream DakotaBoStream.

- **DakotaBoStream & operator<< (const float &f)**
  
  Output operator, writes float to binary stream DakotaBoStream.
DakotaBoStream & operator<<(const unsigned char &c)

Output operator, writes unsigned char to binary stream DakotaBoStream.

DakotaBoStream & operator<<(const unsigned int &i)

Output operator, writes unsigned int to binary stream DakotaBoStream.

DakotaBoStream & operator<<(const unsigned long &l)

Output operator, writes unsigned long to binary stream DakotaBoStream.

DakotaBoStream & operator<<(const unsigned short &s)

Output operator, writes unsigned short to binary stream DakotaBoStream.

Private Attributes

- XDR xdrOutBuf
  
  XDR output stream buffer.

- char outBuf [256]
  
  Buffer to hold converted data before it is written.

### 6.21.1 Detailed Description

The binary output stream class. Overloads the << operator for all data types.

The DakotaBoStream class is a binary output classes which overloads the << operator for all standard data types (int, char, float, etc). The class relies on the built in write methods within the ostream base classes. DakotaBoStream inherits from the ofstream class. The motivation to develop this class was to replace the Rogue wave class which Dakota historically used for binary I/O. The class also utilize rpc/xdr to construct machine independent binary files. The Dakota restart files can now be moved between hosts. These motivation to develop these classes was to replace the RW classes which Dakota historically used for binary I/O.

### 6.21.2 Constructor & Destructor Documentation

#### 6.21.2.1 DakotaBoStream::DakotaBoStream ()

Default constructor, need to open.

Default constructor, allocates xdr stream, but does not call the open() method. The open() method must be called before stream can be written to.

#### 6.21.2.2 DakotaBoStream::DakotaBoStream (const char * s)

Constructor takes name of input file.

Constructor, takes char * filename as argument. Calls base class open method with filename and no other arguments. Also allocates xdr stream.
### 6.21.2.3 DakotaBoStream::DakotaBoStream (const char * s, int flags)

Constructor takes name of input file, flags.
Constructor, takes char * filename and int flags as arguments. Calls base class open method with filename and flags as arguments. Also allocates xdr stream

### 6.21.3 Member Function Documentation

#### 6.21.3.1 void DakotaBoStream::testClass ()

Performs unit testing for the DakotaBoStream class.
Unit test method for the DakotaBinStream class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an expected answer is not received.

#### 6.21.3.2 DakotaBoStream & DakotaBoStream::operator<< (const DakotaString & ds)

Binary Output stream operator<<.
The DakotaString operator<< must first write the xdr buffer size and the original string size to the stream. The input operator needs this information to be able to correctly read and convert the DakotaString.

#### 6.21.3.3 DakotaBoStream & DakotaBoStream::operator<<(const char * s)

Output operator, writes char* TO binary stream DakotaBoStream.
The output of char* is the same as the output of the DakotaString. The size of the xdr buffer and the size of the string must be written first, then the string itself.
The documentation for this class was generated from the following files:

- DakotaBinStream.H
- DakotaBinStream.C
### 6.22 DakotaGraphics Class Reference

The DakotaGraphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.

```c
#include <DakotaGraphics.H>
```

#### Public Methods

- **DakotaGraphics()**  
  constructor.
- **~DakotaGraphics()**  
  destructor.
- **void create_plots_2d(const DakotaVariables &vars, const DakotaResponse &response)**  
  creates the 2d graphics window and initializes the plots.
- **void create_tabular_datstream(const DakotaVariables &vars, const DakotaResponse &response, const DakotaString &tabular_data_file)**  
  opens the tabular data file stream and prints the headings.
- **void add_datapoint(const DakotaVariables &vars, const DakotaResponse &response)**  
  adds data to the 2d graphics and tabular data file.
- **void show_data_3d(DakotaRealArray &X, DakotaRealArray &Y, DakotaRealMatrix &F)**  
  generate a new 3d plot for F(X,Y).

#### Private Attributes

- **Graphics2D * graphics2D**  
  pointer to the 2D graphics object.
- **short win2dOn**  
  flag to indicate if 2D graphics window is active.
- **short win3dOn**  
  flag to indicate if 3D graphics window is active.
- **int graphicsCntr**  
  used for x axis values in 2D graphics and for 1st column in tabular data.
- **short tabularDataFlag**  
  flag to indicate if tabular data stream is active.
- **ofstream tabularDataFStream**
file stream for tabulation of graphics data within compute_response.

6.22.1 Detailed Description

The DakotaGraphics class provides a single interface to 2D (motif) and 3D (PLPLOT) graphics as well as tabular cataloguing of data for post-processing with Matlab, Tecplot, etc.

There is only one DakotaGraphics object (dakotaGraphics) and it is global (for convenient access from strategies, models, and approximations).

6.22.2 Member Function Documentation

6.22.2.1 void DakotaGraphics::create_plots_2d (const DakotaVariables & vars, const DakotaResponse & response)

creates the 2d graphics window and initializes the plots.

Sets up a single event loop for duration of the dakotaGraphics object, continuously adding data to a single window. There is no reset. To start over with a new data set, you need a new object (delete old and instantiate new).

The documentation for this class was generated from the following files:

- DakotaGraphics.H
- DakotaGraphics.C
6.23 DakotaInterface Class Reference

Base class for the interface class hierarchy.

`#include <DakotaInterface.H>`

Inheritance diagram for DakotaInterface:

```
DakotaInterface
  `-- ApplicationInterface
  |    `-- ApproximationInterface
  |         `-- DirectFnApplicInterface
  |                  `-- ForkApplicInterface
  |                          `-- SysCallApplicInterface
```

Public Methods

- DakotaInterface ()
  default constructor.

- DakotaInterface (ProblemDescDB &problem_db, const size_t &num_acv, const size_t &num_fns)
  standard constructor for envelope.

- DakotaInterface (const DakotaInterface &interface)
  copy constructor.

- virtual ~DakotaInterface ()
  destructor.

- DakotaInterface operator= (const DakotaInterface &interface)
  assignment operator.

- virtual void map (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, const int asynch_flag=0)
  the function evaluator: provides a "mapping" from the variables to the responses.

- virtual const DakotaArray< DakotaResponse > & synch ()
  recovers data from a series of asynchronous evaluations (blocking).

- virtual const DakotaList< DakotaResponse > & synch_nowait ()
  recovers data from a series of asynchronous evaluations (nonblocking).

- virtual void serve_evaluations ()
  evaluation server function for multiprocessor executions.

- virtual void stop_evaluation_servers ()
send messages from iterator rank 0 to terminate evaluation servers.

- virtual void `init_communicators` (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  allocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

- virtual void `free_communicators` ()
  deallocate communicator partitions for concurrent evaluations within an iterator and concurrent multiprocessor analyses within an evaluation.

- virtual int `asynch_local_evaluation_concurrency` () const
  return the user-specified concurrency for asynch local evaluations.

- virtual DakotaString `interface_synchronization` () const
  return the user-specified interface synchronization.

- virtual int `minimum_samples` () const
  returns the minimum number of samples required to build a particular `ApproximationInterface` (used by `SurfLayeredModels`).

- virtual void `build_global_approximation` (DakotaIterator &dace_iterator)
  builds a global approximation for use as a surrogate.

- virtual void `build_local_approximation` (DakotaModel &actual_model)
  builds a local approximation for use as a surrogate.

- virtual void `update_approximation` (const DakotaRealVector &x_star, const DakotaResponse &response_star)
  updates an existing global approximation with new data.

- const DakotaIntList & `sync_nowait_completions` ()
  returns id’s matching response list from `sync_nowait()`.

- const DakotaString & `interface_type` () const
  returns the interface type.

- int `total_eval_counter` () const
  returns the total number of evaluations of the interface.

- int `new_eval_counter` () const
  returns the number of new (nonduplicate) evaluations of the interface.

- short `multi_proc_eval_flag` () const
  returns a flag signaling the use of multiprocessor evaluation partitions.

- short `iterator_dedicated_master_flag` () const
  returns a flag signaling the use of a dedicated master processor for iterator scheduling.
Protected Methods

- **DakotaInterface** (BaseConstructor, const ProblemDescDB &problem_db)
  
  *constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).*

Protected Attributes

- **DakotaString interfaceType**
  
  *interface type may be (1) application: system, fork, direct, or xml; or (2) approximation: ann, rsm, mars, hermite, ksm, mpa, taylor, or hierarchical.*

- int **fnEvalId**
  
  *total evaluation counter.*

- int **newFnEvalId**
  
  *new (non-duplicate) evaluation counter.*

- **DakotaIntList beforeSynchIdList**
  
  *bookkeeps fnEvalId’s of all asynchronous evaluations (new & duplicate).*

- **DakotaArray < DakotaResponse > rawResponseArray**
  
  *The complete array of responses returned after a blocking schedule of asynchronous evaluations.*

- **DakotaList< DakotaResponse > rawResponseList**
  
  *The partial list of responses returned after a nonblocking schedule of asynchronous evaluations.*

- **DakotaIntList completionList**
  
  *identifies the responses in rawResponseList for nonblocking schedules.*

- short **multiProcEvalFlag**
  
  *flag for multiprocessor evaluation partitions (evalComm).*

- short **iteratorDedMasterFlag**
  
  *flag for dedicated master partitioning at the iterator level.*

- short **verboseFlag**
  
  *flag for verbose interface output.*

- short **debugFlag**
  
  *flag for really verbose (debug) interface output.*

Private Methods

- **DakotaInterface * getInterface (ProblemDescDB &problem_db, const size_t &num_acv, const size_t &num_ins)**
  
  *Used by the envelope to instantiate the correct letter class.*
Private Attributes

- DakotaInterface * interfaceRep
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  number of objects sharing interfaceRep.

6.23.1 Detailed Description

Base class for the interface class hierarchy.

The DakotaInterface class hierarchy provides the part of a DakotaModel that is responsible for mapping a set of DakotaVariables into a set of DakotaResponses. The mapping is performed using either a simulation-based application interface or a surrogate-based approximation interface. For memory efficiency and enhanced polymorphism, the interface hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaInterface) serves as the envelope and one of the derived classes (selected in DakotaInterface::get_interface()) serves as the letter.

6.23.2 Constructor & Destructor Documentation

6.23.2.1 DakotaInterface::DakotaInterface ()

default constructor.

used in DakotaModel envelope class instantiations

6.23.2.2 DakotaInterface::DakotaInterface (ProblemDescDB & problem_db, const size_t & num_acv, const size_t & num_fns)

standard constructor for envelope.

Used in DakotaModel instantiation to build the envelope. This constructor only needs to extract enough data to properly execute get_interface, since DakotaInterface::DakotaInterface(BaseConstructor, problem_db) builds the actual base class data inherited by the derived interfaces.

6.23.2.3 DakotaInterface::DakotaInterface (const DakotaInterface & interface)

copy constructor.

Copy constructor manages sharing of interfaceRep and incrementing of referenceCount.

6.23.2.4 DakotaInterface::~DakotaInterface () [virtual]

destructor.

Destructor decrements referenceCount and only deletes interfaceRep if referenceCount is zero.
6.23.2.5 DakotaInterface::DakotaInterface (BaseConstructor, const ProblemDescDB & problem_db) [protected]

cConstructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

This constructor is the one which must build the base class data for all inherited interfaces. getInterface() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling getInterface() again). Since this is the letter and the letter IS the representation, interfaceRep is set to NULL (an uninitialized pointer causes problems in ~DakotaInterface).

6.23.3 Member Function Documentation

6.23.3.1 DakotaInterface DakotaInterface::operator= (const DakotaInterface & interface)

assignment operator.


6.23.3.2 DakotaInterface * DakotaInterface::getInterface (ProblemDescDB & problem_db, const size_t & num_acv, const size_t & num_fns) [private]

Used by the envelope to instantiate the correct letter class.

used only by the envelope constructor to initialize interfaceRep to the appropriate derived type, as given by the interfaceType attribute.

6.23.4 Member Data Documentation

6.23.4.1 DakotaArray <DakotaResponse> DakotaInterface::rawResponseArray [protected]

The complete array of responses returned after a blocking schedule of asynchronous evaluations.

The array is the raw set of responses corresponding to all asynchronous map calls. This raw array is postprocessed (i.e., finite difference gradients merged) in DakotaModel::synchronize() where it becomes responseArray.

6.23.4.2 DakotaList <DakotaResponse> DakotaInterface::rawResponseList [protected]

The partial list of responses returned after a nonblocking schedule of asynchronous evaluations.

The list is a partial set of completions which must be identified through the use of completionList. Post-processing from raw to combined form (i.e., finite difference gradient merging) is not currently supported in DakotaModel::synchronize_nowait().

The documentation for this class was generated from the following files:
- DakotaInterface.H
- DakotaInterface.C
6.24 DakotaIterator Class Reference

Base class for the iterator class hierarchy.

```
#include <DakotaIterator.H>
```

Inheritance diagram for DakotaIterator:

```
DakotaIterator

DACEIterator  DakotaNonD

NonDAdvMeanValue

NonDPCE

NonDProbability

NonDSampling

DakotaOptimizer

CONMINOptimizer

DOTOptimizer

NPSOLOptimizer

SGOPTOptimizer

SNLLOptimizer

ParamStudy
```

Public Methods

- **DakotaIterator ()**
  
  *default constructor.*

- **DakotaIterator (DakotaModel &model)**
  
  *standard constructor for envelope.*

- **DakotaIterator (const DakotaIterator &iterator)**
  
  *copy constructor.*

- virtual ~DakotaIterator ()
  
  *destructor.*

- DakotaIterator operator= (const DakotaIterator &iterator)
  
  *assignment operator.*

- virtual void run_iterator ()
  
  *run the iterator.*

- virtual const DakotaVariables & iterator_variable_results () const
  
  *return the final iterator solution (variables).*

- virtual const DakotaResponse & iterator_response_results () const
  
  *return the final iterator solution (response).*
virtual void print\_iterator\_results (ostream &s) const
    print the final iterator results.

virtual void multi\_objective\_weights (const DakotaRealVector &multi\_obj\_wts)
    set the relative weightings for multiple objective functions. Used by ConcurrentStrategy for Pareto set optimization.

virtual void sampling\_reset (int min\_samples, short all\_data\_flag, short stats\_flag)
    reset sampling iterator.

virtual const DakotaString & sampling\_scheme () const
    return sampling name.

void user\_defined\_model (const DakotaModel &the\_model)
    set the model.

DakotaModel & user\_defined\_model () const
    return the model.

const DakotaString & method\_name () const
    return the method name.

int maximum\_concurrency () const
    return the maximum concurrency supported by the iterator.

const DakotaRealVectorArray & all\_variables () const
    return the complete set of evaluated variables (used by ApproximationInterface::build\_approximation).

const DakotaArray< DakotaResponse > & all\_responses () const
    return the complete set of computed responses (used by ApproximationInterface::build\_approximation).

short is\_null () const
    function to check iteratorRep (does this envelope contain a letter).

Protected Methods

- DakotaIterator (BaseConstructor, DakotaModel &model)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

- Dakotalterator (NoDBBaseConstructor, DakotaModel &model)
  base class for iterator classes constructed on the fly (no DB queries).
Protected Attributes

- **DakotaModel & userDefinedModel**
  class member reference for the model passed into the constructor.

- **const ProblemDescDB & probDescDB**
  class member reference to the problem description database.

- **DakotaString methodName**
  name of the iterator (the user’s method spec).

- **int maxIterations**
  maximum number of iterations for the iterator.

- **int maxFunctionEvals**
  maximum number of fn evaluations for the iterator.

- **int numFunctions**
  number of response functions.

- **int maxConcurrency**
  maximum coarse-grained concurrency.

- **int numContinuousVars**
  number of active continuous vars.

- **int numDiscreteVars**
  number of active discrete vars.

- **int numVars**
  total number of vars. (active and inactive).

- **DakotaIntArray activeSetVector**
  this vector tracks the data requirements for the response functions. It uses a 0 value for inactive functions and, for active functions, sums 1 for value, 2 for gradient, and 4 for Hessian.

- **DakotaString gradientType**
  type of gradient data: "analytic", "numerical", "mixed", or "none".

- **DakotaString hessianType**
  type of Hessian data: "analytic" or "none".

- **DakotaString finiteDiffType**
  type of finite difference interval: "central" or "forward".

- **DakotaString methodSource**
  source of finite difference routine: "dakota" or "vendor".

- **Real finiteDiffStepSize**
  relative finite difference step size.
- DakotaIntList `mixedGradAnalyticIds`
  for mixed gradients, contains ids of functions with analytic gradients.

- DakotaIntList `mixedGradNumericalIds`
  for mixed gradients, contains ids of functions with numerical gradients.

- DakotaString `outputLevel`
  iterator verbosity: "verbose", "normal", "quiet", or "debug".

- short `verboseOutput`
  convenience flag for `outputLevel` == "verbose".

- short `debugOutput`
  convenience flag for `outputLevel` == "debug".

- short `asynchFlag`
  copy of the model's asynchronous evaluation flag.

- DakotaRealVectorArray `allVariables`
  array of all variables evaluated (used by `ApproximationInterface`).

- DakotaArray < DakotaResponse > `allResponses`
  array of all responses computed (used by `ApproximationInterface`).

**Static Protected Attributes**

- DakotaModel & `staticModel` = dummy\_model
  static model reference used by OPT++, NPSOL, NonDAMV.

**Private Methods**

- DakotaIterator * `get_iterator`(DakotaModel &model)
  Used by the envelope to instantiate the correct letter class.

- void `populate\_gradient\_vars`()
  Used only by constructor functions to define gradient variables for use within the iterator hierarchy.

**Private Attributes**

- DakotaIterator * `iteratorRep`
  pointer to the letter (initialized only for the envelope).

- int `referenceCount`
  number of objects sharing `iteratorRep`. 
6.24 DakotaIterator Class Reference

6.24.1 Detailed Description

Base class for the iterator class hierarchy.

The DakotaIterator class is the base class for one of the primary class hierarchies in DAKOTA. The iterator hierarchy contains all of the iterative algorithms which use repeated execution of simulations as function evaluations. For memory efficiency and enhanced polymorphism, the iterator hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaIterator) serves as the envelope and one of the derived classes (selected in DakotaIterator::get_iterator()) serves as the letter.

6.24.2 Constructor & Destructor Documentation

6.24.2.1 DakotaIterator::DakotaIterator ()

default constructor.

The default constructor is used in Vector<DakotaIterator> instantiations and for initialization of DakotaIterator objects contained in DakotaStrategy derived classes (see derived class header files). iteratorRep is NULL in this case (a populated problem_db is needed to build a meaningful DakotaIterator object). This makes it necessary to check for NULL pointers in the copy constructor, assignment operator, and destructor.

6.24.2.2 DakotaIterator::DakotaIterator (DakotaModel & model)

standard constructor for envelope.

Used in iterator instantiations within strategy constructors. Envelope constructor only needs to extract enough data to properly execute get_iterator, since DakotaIterator( BaseConstructor, model) builds the actual base class data inherited by the derived iterators.

6.24.2.3 DakotaIterator::DakotaIterator (const DakotaIterator & iterator)

copy constructor.

Copy constructor manages sharing of iteratorRep and incrementing of referenceCount.

6.24.2.4 DakotaIterator::~DakotaIterator () [virtual]

destructor.

Destructor decrements referenceCount and only deletes iteratorRep when referenceCount reaches zero.

6.24.2.5 DakotaIterator::DakotaIterator (BaseConstructor, DakotaModel & model)
[protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor builds the base class data for all inherited iterators. `get_iterator()` instantiates a derived class and the derived class selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling `get_iterator()` again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaIterator).

### 6.24.2.6 DakotaIterator::DakotaIterator (NoDBBaseConstructor, DakotaModel & model)

[protected]

base class for iterator classes constructed on the fly (no DB queries).

This constructor also builds base class data for inherited iterators. However, it is used for on-the-fly instantiations for which DB queries cannot be used (e.g., ApproximationInterface instantiation of DACEIterator or NonDProbability, AMV usage of optimizers, etc.). Therefore it only sets attributes taken from the incoming model.

### 6.24.3 Member Function Documentation

#### 6.24.3.1 DakotaIterator DakotaIterator::operator= (const DakotaIterator & iterator)

assignment operator.


#### 6.24.3.2 void DakotaIterator::run_iterator () [virtual]

run the iterator.

This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented in DACEIterator, DakotaNonD, DakotaOptimizer, and ParamStudy.

#### 6.24.3.3 DakotaIterator * DakotaIterator::get_iterator (DakotaModel & model) [private]

Used by the envelope to instantiate the correct letter class.

Used only by the envelope constructor to initialize iteratorRep to the appropriate derived type, as given by the methodName attribute.

#### 6.24.3.4 void DakotaIterator::populate_gradient_vars () [private]

Used only by constructor functions to define gradient variables for use within the iterator hierarchy.

Convenience function for constructors. Populates gradient and Hessian data attributes from the problem description database.

The documentation for this class was generated from the following files:

- DakotaIterator.H
- DakotaIterator.C
6.25 DakotaList Class Template Reference

Template class for the Dakota bookkeeping list.
#include <DakotaList.H>

Public Methods

- DakotaList ()
  Default constructor.
- DakotaList (const DakotaList<T> &a)
  Copy constructor.
- ~DakotaList ()
  Destructor.
- DakotaList<T> & operator=(const DakotaList<T> &)
  Assignment operator.
- void testClass ()
  Class unit test method.
- size_t entries () const
  Returns the number of items that are currently in the list.
- void append (const T &a)
  Adds the item a to the end of the list.
- T get ()
  Returns the last item on the list and removes it.
- T removeAt (size_t index)
  Removes and returns the item at the specified index.
- int remove (const T &a)
  Removes the specified item from the list.
- void insert (const T &a)
  Adds the item a to the end of the list.
- int contains (const T &a) const
  Returns TRUE if list contains object a, returns FALSE otherwise.
- int find (int(*testFun)(const T &, void *), void *d, T &k) const
  Returns TRUE if the list contains an object which the user defined function finds.
• `size_t index (int(*testFun)(const T &, void *), void *d) const`  
  Returns the index of object which the user defined test function finds.

• `void sort (int(*sortFun)(const T &, const T &))`  
  Sorts the list into an order based on the predefined sort function.

• `size_t index (const T &a) const`  
  Returns the index of the object.

• `size_t occurrencesOf (const T &a) const`  
  Returns the number of items in the list equal to object.

• `bool isEmpty ()`  
  Returns TRUE if list is empty, returns FALSE otherwise.

• `T & operator[] (size_t i)`  
  Returns the object at index i (can use as lvalue).

• `const T & operator[] (size_t i) const`  
  Returns the object at index i, const (can’t use as lvalue).

### 6.25.1 Detailed Description

**template<class T> class DakotaList<T>**

Template class for the Dakota bookkeeping list.

The DakotaList is the common list class for Dakota. It inherits from either the RW list class or the STL list class. Extends the base list class to add Dakota specific methods Builds upon the previously existing DakotaValList class

### 6.25.2 Member Function Documentation

#### 6.25.2.1 `template<class T> void DakotaList<T>::testClass ()`

Class unit test method.

Unit test method for the DakotaList class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an unexpected answer is received.

#### 6.25.2.2 `template<class T> void DakotaList<T>::append (const T & a)`

Adds the item a to the end of the list.

Insert item at the end of list, calls STL push_back method which places the object at the end of the list. Same as the insert() method.
6.25.2.3 template <class T> T DakotaList<T>::get()

Returns the last item on the list and removes it.
Remove and return item from front of list. Returns the object pointed to by the STL front iterator. It also deletes the first node by calling the STL erase method. The erase() method handles all aspects of removing a node from the list.

6.25.2.4 template <class T> T DakotaList<T>::removeAt(size_t index)

Removes and returns the item at the specified index.
Removes the item at the index specified. Uses a STL iterator to step to the appropriate position in the list, and then calls the STL erase() method(). The erase() method handles all aspects of removing a node from the list.

6.25.2.5 template <class T> int DakotaList<T>::remove(const T & a)

Removes the specified item from the list.
Remove the item from the list, uses STL iterator to find the object. It then calls the STL erase() method which handles all the aspects of doing a remove.

6.25.2.6 template <class T> void DakotaList<T>::insert(const T & a)

Adds the item a to the end of the list.
Insert item at the end of list, calls STL push_back method which places the object at the end of the list. Same as the append() method.

6.25.2.7 template <class T> int DakotaList<T>::contains(const T & a) const

Returns TRUE if list contains object a, returns FALSE otherwise.
Uses a STL iterator to step through the list. Returns a true as soon as the specified object is found.

6.25.2.8 template <class T> int DakotaList<T>::find(int(* testFun)(const T &, void *), void * d, T & k) const

Returns TRUE if the list contains an object which the user defined function finds.
The find method mimicks the RW find algorithm by utilizing the FunctionCompare class and the STL find_if algorithm. The find_if method returns an iterator. The iterator is then compared against the iterator that is returned by the method end() to determine if object was found.

6.25.2.9 template <class T> size_t DakotaList<T>::index(int(* testFun)(const T &, void *), void * d) const

Returns the index of object which the user defined test function finds.
The index method mimicks the RW index algorithm by utilizing the FunctionCompare class and the STL find_if algorithm. The find_if method returns an iterator. The iterator is then used to find the position of the object.
6.25.2.10 template<class T> void DakotaList<T>::sort (int (*sortFun)(const T &, const T &))

Sorts the list into an order based on the predefined sort function. The sort method utilizes the SortCompare class and the STL sort algorithm to sort a list based on the predefined function sortFun. Each type T should have a defined sort method if you wish to sort the specific data. Note: Not supported under SOLARIS!

6.25.2.11 template<class T> size_t DakotaList<T>::index (const T & a) const

Returns the index of the object. Returns the index of the item in the list, uses STL iterator to step through the list. Returns the index of the first instance of the object, there may be more than one in the list.

6.25.2.12 template<class T> size_t DakotaList<T>::occurrencesOf (const T & a) const

Returns the number of items in the list equal to object. Uses an STL iterator to step through the list and count the number of occurrences of the specified object.

6.25.2.13 template<class T> T & DakotaList<T>::operator[ ] (size_t i)

Returns the object at index i (can use as lvalue). Return item at position i of list, steps through using STL iterator Once object is found it returns the value pointed to by the iterator.

6.25.2.14 template<class T> const T & DakotaList<T>::operator[ ] (size_t i) const

Returns the object at index i, const (can’t use as lvalue). Return const item at position i of list, steps through the list using an STL iterator. Once object is found it returns the value pointed to by the iterator.

The documentation for this class was generated from the following file:

- DakotaList.H
6.26 DakotaMatrix Class Template Reference

Template class for the Dakota numerical matrix.

```cpp
#include "DakotaMatrix.H"
```

Inheritance diagram for DakotaMatrix::

![Inheritance diagram](image)

**Public Methods**

- **DakotaMatrix** (size_t num_rows=0, size_t num_cols=0)
  
  Constructor, takes number of rows, and number of columns as arguments.

- ~DakotaMatrix ()
  
  Destructor.

- DakotaMatrix<T> & operator= (const T &ival)
  
  Sets all elements in the matrix to ival.

- size_t num_rows () const
  
  Returns the number of rows for the matrix.

- size_t num_columns () const
  
  Returns the number of columns for the matrix.

- void reshape_2d (size_t num_rows, size_t num_cols)
  
  Resizes the matrix to num_rows by num_cols.

- void print (ostream &s) const
  
  Prints a DakotaMatrix to an output stream.

- void read (UnPackBuffer &s)
  
  Reads a DakotaMatrix from an UnPackBuffer after an MPI receive.

- void print (PackBuffer &s) const
  
  Prints a DakotaMatrix to a PackBuffer prior to an MPI send.

- void testClass ()
  
  Class unit test method.
6.26.1 Detailed Description

template<class T> class DakotaMatrix<T>

Template class for the Dakota numerical matrix.
A matrix class template to provide 2D arrays of objects. The matrix is zero-based, rows: 0 to (numRows-1) and cols: 0 to (numColumns-1). The class supports overloading of the subscript operator allowing it to emulate a normal built-in 2D array type. The DakotaMatrix relies on the DakotaBaseVector template class to manage the differences between the Rogue Wave vector class and the STL vector class.

6.26.2 Member Function Documentation

6.26.2.1 template<class T> DakotaMatrix<T> & DakotaMatrix<T>::operator= (const T & val)

Sets all elements in the matrix to ival.
calls base class operator=(ival)
Reimplemented from DakotaBaseVector.

6.26.2.2 template<class T> void DakotaMatrix<T>::testClass ()

Class unit test method.
verifies the basic functionality of the DakotaMatrix class. The assert function is used to test the correctness of results.
The documentation for this class was generated from the following file:

- DakotaMatrix.H
6.27 DakotaModel Class Reference

Base class for the model class hierarchy.

#include <DakotaModel.H>

Inheritance diagram for DakotaModel::

```
DakotaModel
    LayeredModel
    NestedModel
    SingleModel
    HierLayeredModel
    SurrLayeredModel
```

Public Methods

- **DakotaModel ()**
  
  *default constructor.*

- **DakotaModel (ProblemDescDB &problem_db)**
  
  *standard constructor for envelope.*

- **DakotaModel (const DakotaModel &model)**
  
  *copy constructor.*

- virtual ~DakotaModel ()
  
  *destructor.*

- **DakotaModel operator= (const DakotaModel &model)**
  
  *assignment operator.*

- virtual DakotaModel & subordinate_model ()
  
  *return the sub-model in nested and layered models.*

- virtual DakotaIterator & subordinate_iterator ()
  
  *return the sub-iterator in nested and layered models.*

- virtual int maximum_concurrency () const
  
  *used to return DACE iterator concurrency for SurrLayeredModels.*

- virtual void build_approximation ()
  
  *build the approximation in LayeredModels.*

- virtual void update_approximation (const DakotaRealVector &x_star, const DakotaResponse &response_star)
  
  *update the approximation in LayeredModels.*
update the approximation in SurrLayeredModels with new data.

- virtual void compute_correction (const DakotaResponse &truth_response, const DakotaResponse &approx_response, const DakotaRealVector &c_vars)
  compute correction factors for use in LayeredModels.

- virtual void auto_correction (short correction_flag)
  manages automatic application of correction factors in LayeredModels.

- virtual void apply_correction (DakotaResponse &approx_response, const DakotaRealVector &c_vars, short quiet_flag=0)
  apply correction factors to approx_response (for use in LayeredModels).

- virtual DakotaString local_eval_synchronization ()
  return derived model synchronization setting.

- virtual void free_communicators ()
  deallocate communicator partitions for a model.

- virtual void serve ()
  Service job requests received from the master. Completes when a termination message is received from stop_servers().

- virtual void stop_servers ()
  Executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete.

- virtual const DakotaIntList & synchronize_nowait_completions ()
  Return completion id’s matching response list from synchronize_nowait.

- virtual short derived_master_overload () const
  Return a flag indicating the combination of multiprocessor evaluations and a dedicated master iterator scheduling. Used in synchronous compute_response functions to prevent the error of trying to run a multi-processor job on the master.

- virtual int total_eval_counter () const
  Return the total evaluation count from the interface.

- virtual int new_eval_counter () const
  Return the new (non-duPLICATE) evaluation count from the interface.

- void compute_response ()
  Compute the DakotaResponse at currentVariables (default asv).

- void compute_response (const DakotaIntArray &asv)
  Compute the DakotaResponse at currentVariables (specified asv).

- void async_compute_response ()
  Spawn an asynchronous job (or jobs) that computes the value of the DakotaResponse at currentVariables (default asv).
void async_compute_response (const DakotaIntArray &asv)
  Spawn an asynchronous job (or jobs) that computes the value of the DakotaResponse at currentVariables (specified asv).

const DakotaArray< DakotaResponse > & synchronize ()
  Execute a blocking scheduling algorithm to collect the complete set of results from a group of asynchronous evaluations.

const DakotaList< DakotaResponse > & synchronize_nowait ()
  Execute a nonblocking scheduling algorithm to collect all available results from a group of asynchronous evaluations.

void init_communicators (const int &max_iterate_concurrency)
  allocate communicator partitions for a model.

size_t tv () const
  return total number of vars.

size_t cv () const
  return number of active continuous variables.

size_t dv () const
  return number of active discrete variables.

size_t num_functions () const
  return number of functions in currentResponse.

void active_variables (const DakotaVariables &vars)
  set the active variables in currentVariables.

const DakotaRealVector & continuous_variables () const
  return the active continuous variables from currentVariables.

void continuous_variables (const DakotaRealVector &c_vars)
  set the active continuous variables in currentVariables.

const DakotaIntVector & discrete_variables () const
  return the active discrete variables from currentVariables.

void discrete_variables (const DakotaIntVector &d_vars)
  set the active discrete variables in currentVariables.

const DakotaStringArray & continuous_variable_labels () const
  return the active continuous variable labels from currentVariables.

const DakotaStringArray & discrete_variable_labels () const
  return the active discrete variable labels from currentVariables.

void inactive_continuous_variables (const DakotaRealVector &i_c_vars)
  set the inactive continuous variables in currentVariables.
- void inactive_discrete_variables (const DakotaIntVector & inactive_vars)  
  set the inactive discrete variables in currentVariables.

- const DakotaRealVector & continuous_lower_bounds () const  
  return the active continuous variable lower bounds from userDefinedVarConstraints.

- void continuous_lower_bounds (const DakotaRealVector & continuous_lower_bnds)  
  set the active continuous variable lower bounds in userDefinedVarConstraints.

- const DakotaRealVector & continuous_upper_bounds () const  
  return the active continuous variable upper bounds from userDefinedVarConstraints.

- void continuous_upper_bounds (const DakotaRealVector & continuous_upper_bnds)  
  set the active continuous variable upper bounds in userDefinedVarConstraints.

- const DakotaIntVector & discrete_lower_bounds () const  
  return the active discrete variable lower bounds from userDefinedVarConstraints.

- void discrete_lower_bounds (const DakotaIntVector & discrete_lower_bnds)  
  set the active discrete variable lower bounds in userDefinedVarConstraints.

- const DakotaIntVector & discrete_upper_bounds () const  
  return the active discrete variable upper bounds from userDefinedVarConstraints.

- void discrete_upper_bounds (const DakotaIntVector & discrete_upper_bnds)  
  set the active discrete variable upper bounds in userDefinedVarConstraints.

- const size_t & num_linear_ineq_constraints () const  
  return the number of linear inequality constraints.

- const size_t & num_linear_eq_constraints () const  
  return the number of linear equality constraints.

- const DakotaRealMatrix & linear_ineq_constraint_coeffs () const  
  return the linear inequality constraint coefficients.

- const DakotaRealVector & linear_ineq_constraint_lower_bounds () const  
  return the linear inequality constraint lower bounds.

- const DakotaRealVector & linear_ineq_constraint_upper_bounds () const  
  return the linear inequality constraint upper bounds.

- const DakotaRealMatrix & linear_eq_constraint_coeffs () const  
  return the linear equality constraint coefficients.

- const DakotaRealVector & linear_eq_constraint_targets () const  
  return the linear equality constraint targets.

- const DakotaIntList & merged_integer_list () const
return the list of discrete variables merged into a continuous array in currentVariables.

- const DakotaIntArray & message_lengths () const
  return the array of MPI packed message buffer lengths (messageLengths).

- const DakotaVariables & current_variables () const
  return the current variables (currentVariables).

- const DakotaResponse & current_response () const
  return the current response (currentResponse).

- const ProblemDescDB & prob_desc_db () const
  return the problem description database (probDescDB).

- const DakotaString & model_type () const
  return the model type (modelType).

- short async_flag () const
  return the asynchronous evaluation flag (asyncEvalFlag).

- void async_flag (const short flag)
  set the asynchronous evaluation flag (asyncEvalFlag).

- void activate_model_auto_graphics ()
  set modelAutoGraphicsFlag to activate posting of graphics data within compute_response/synchronize functions (automatic graphics posting in the model as opposed to graphics posting at the strategy level).

- const DakotaString & gradient_method () const
  return the gradient method (gradType).

- int gradient_concurrency () const
  return the gradient concurrency for use in parallel configuration logic.

- short is_null () const
  function to check modelRep (does this envelope contain a letter).

### Protected Methods

- DakotaModel (BaseConstructor, ProblemDescDB & probDescDB)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

- virtual void derived_compute_response (const DakotaIntArray & asv)
  portion of compute_response() specific to derived model classes.

- virtual void derived_asynch_compute_response (const DakotaIntArray & asv)
  portion of async ComputeResponse() specific to derived model classes.

- virtual const DakotaArray & Derived_synchronize ()
portion of `synchronize()` specific to derived model classes.

- virtual const `DakotaList< DakotaResponse >` & derived `synchronize_nowait()` portion of `synchronize_nowait()` specific to derived model classes.

- virtual void `derived_init_communicators` (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency) `init_communicators()` specific to derived model classes.

Protected Attributes

- `DakotaVariables currentVariables`
  the set of current variables used by the model for performing function evaluations.

- `size_t numGradVars`
  the number of active continuous variables (used in the finite difference routines).

- `DakotaResponse currentResponse`
  the set of current responses that holds the results of model function evaluations.

- `size_t numFns`
  the number of functions in `currentResponse`.

- `DakotaVarConstraints userDefinedVarConstraints`
  Explicit constraints on variables are maintained in the `DakotaVarConstraints` class hierarchy. Currently, this includes linear constraints and bounds, but could be extended in the future to include other explicit constraints which (1) have their form specified by the user, and (2) are not catalogued in `DakotaResponse` since their form and coefficients are published to an iterator at startup.

Private Methods

- `DakotaModel * get_model (ProblemDescDB &problem_db)`
  Used by the envelope to instantiate the correct letter class.

- int `fd_gradients` (const DakotaIntArray &map_asv, const DakotaIntArray &fd_grad_asv, const DakotaIntArray &original_asv, const int asynch_flag)
  evaluate numerical gradients using finite differences. This routine is selected with "method-source dakota" (the default method-source) in the numerical gradient specification.

- void `synchronize_fd_gradients` (const DakotaArray< DakotaResponse > &fd_grad_responses, DakotaResponse &new_response, const DakotaIntArray &fd_grad_asv, const DakotaIntArray &asv)
  combine results from an array of finite difference response objects (fd_grad_responses) into a single response (new_response).

- void `update_response` (DakotaResponse &new_response, const DakotaIntArray &fd_grad_asv, const DakotaIntArray &asv, const short initial_map, DakotaRealVector &fn_vals_x0, DakotaRealMatrix &partial_fn_grads, const DakotaRealMatrix &new_fn_grads)
overlay results to update a response object.

- **void** manage\_asv (const DakotaIntArray &asv\_in, DakotaIntArray &map\_asv\_out, DakotaIntArray &fd\_grad\_asv\_out, int &use\_fd\_grad)
  
  Coordinates map() and fd\_gradients() calls given an asv\_in input.

### Private Attributes

- **DakotaModel** * modelRep
  
  pointer to the letter (initialized only for the envelope).

- **int** referenceCount
  
  number of objects sharing modelRep.

- **const ProblemDescDB &** probDescDB
  
  class member reference to the problem description database.

- **const ParallelLibrary &** parallelLib
  
  class member reference to the parallel library.

- **DakotaIntArray** messageLengths
  
  length of packed MPI buffers containing vars, vars and asv, response, and PRPair.

- **DakotaString** modelType
  
  type of model: single, nested, or layered.

- **short** asynchFDFlag
  
  flags use of fd\_gradients w/i async\_compute\_response.

- **short** asynchEvalFlag
  
  flags async evaluations (local or distributed).

- **short** modelAutoGraphicsFlag
  
  flag for posting of graphics data within compute\_response (automatic graphics posting in the model as opposed to graphics posting at the strategy level).

- **DakotaList\lt DakotaVariables \gt** varsList
  
  history of vars populated in async\_compute\_response() and used in synchronize().

- **DakotaList\lt DakotaIntArray \gt** asvList
  
  if asynchFDFlag is set, transfers asv requests to synchronize.

- **DakotaShortList** initialMapList
  
  transfers initial\_map flag values from fd\_gradients to synchronize\_fd\_gradients.

- **DakotaShortList** dbFnsList
  
  transfers db\_fns flag values from fd\_gradients to synchronize\_fd\_gradients.

- **DakotaList\lt DakotaResponse \gt** dbResponseList
transfers database captures from \texttt{fd\_gradients} to synchronize \texttt{fd\_gradients}.

- \texttt{DakotaRealList deltaList}
  transfers deltas from \texttt{fd\_gradients} to synchronize \texttt{fd\_gradients}.

- \texttt{DakotaIntList numMapsList}
  tracks the number of maps used in \texttt{fd\_gradients()}. Used in \texttt{synchronize()} as a key for combining finite difference responses into numerical gradients.

- \texttt{DakotaArray< DakotaResponse > responseArray}
  used to return an array of responses for asynchronous evaluations. This array has the responses in final concatenated form. The similar array in \texttt{DakotaInterface} contains the raw responses.

- \texttt{DakotaList< DakotaResponse > responseList}
  used to return a list of responses for asynchronous evaluations. This list has the responses in final concatenated form. The similar list in \texttt{DakotaInterface} contains the raw responses.

- \texttt{DakotaString gradType}
  
  gradient type: none, numerical, analytic, mixed.

- \texttt{DakotaString methodSrc}
  
  method source: dakota, vendor.

- \texttt{DakotaString intervalType}
  
  interval type: forward, central.

- \texttt{Real finiteDiffSS}
  
  relative finite difference step size.

- \texttt{DakotaIntList idAnalytic}
  
  analytic fn id's for mixed gradients.

6.27.1 Detailed Description

Base class for the model class hierarchy.

The DakotaModel class is the base class for one of the primary class hierarchies in DAKOTA. The model hierarchy contains a set of variables, an interface, and a set of responses, and an iterator operates on the model to map the variables into responses using the interface. For memory efficiency and enhanced polymorphism, the model hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaModel) serves as the envelope and one of the derived classes (selected in \texttt{DakotaModel::get\_model()}) serves as the letter.

6.27.2 Constructor & Destructor Documentation
6.27.2.1  DakotaModel::DakotaModel()

default constructor.
The default constructor is used in Vector<DakotaModel> instantiations and for initialization of Dakota-Model objects contained in DakotaStrategy derived classes (see derived strategy header files). modelRep is NULL in this case (a populated problem_db is needed to build a meaningful DakotaModel object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

6.27.2.2  DakotaModel::DakotaModel(ProblemDescDB & problem_db)

standard constructor for envelope.
Used in model instantiations within strategy constructors. Envelope constructor only needs to extract enough data to properly execute get_model, since DakotaModel(BaseConstructor, problem_db) builds the actual base class data for the derived models.

6.27.2.3  DakotaModel::DakotaModel(const DakotaModel & model)

copy constructor.
Copy constructor manages sharing of modelRep and incrementing of referenceCount.

6.27.2.4  DakotaModel::~DakotaModel() [virtual]
destructor.
Destructor decrements referenceCount and only deletes modelRep when referenceCount reaches zero.

6.27.2.5  DakotaModel::DakotaModel(BaseConstructor, ProblemDescDB & problem_db)
[protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor builds the base class data for all inherited models. get_model() instantiates a derived class and the derived class selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_model() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaModel).

6.27.3  Member Function Documentation

6.27.3.1  DakotaModel DakotaModel::operator=(const DakotaModel & model)

assignment operator.
6.27.3.2 **DakotaString** DakotaModel::local_eval_synchronization ()  
[virtual]

return derived model synchronization setting.

SingleModels and HierLayeredModels redefine this virtual function. A default value of "synchronous" prevents async local operations for:

- NestedModels: a subIterator can support message passing parallelism, but not async local. Also, probDescDB "interface.synchronization" will be bad if no optional interface (will contain last interface spec. parsed).
- SurrLayeredModels: while async evals on approximations will work due to some added bookkeeping, avoiding them is preferable.

Reimplemented in HierLayeredModel, and SingleModel.

6.27.3.3 **void** DakotaModel::init_communicators (const int & max_iterator_concurrency)

allocate communicator partitions for a model.

The init_communicators() and derived init_communicators() functions are structured to avoid performing the messageLengths estimation more than once. init_communicators() (not virtual) performs the estimation and then forwards the results to derived init_communicators (virtual) which uses the data in different contexts.

6.27.3.4 **DakotaModel** DakotaModel::get_model (ProblemDescDB & problem_db)  
[private]

Used by the envelope to instantiate the correct letter class.

Used only by the envelope constructor to initialize modelRep to the appropriate derived type, as given by the modelType attribute.

6.27.3.5 **int** DakotaModel::fd_gradients (const DakotaIntArray & map_asv, const DakotaIntArray & fd_grad_asv, const DakotaIntArray & original_asv, const int async_flag)  
[private]

evaluate numerical gradients using finite differences. This routine is selected with "method source dakota" (the default method source) in the numerical gradient specification.

Compute finite difference gradients, put the data in currentResponse, and return the number of maps used by fd_gradients. This return value is used by async_compute_response() and synchronize() to track response arrays and it could be used to improve management of max function evaluations within the iterators.

6.27.3.6 **void** DakotaModel::synchronize_fd_gradients (const DakotaArray & fd_grad_responses, DakotaResponse & new_response, const DakotaIntArray & fd_grad_asv, const DakotaIntArray & asv)  
[private]

combine results from an array of finite difference response objects (fd_grad_responses) into a single response (new_response).

Merge a vector of fd_grad_responses into a single new_response. This function is used both by compute_response() for the case of asynchronous fd_gradients() and by synchronize() for the case where one or more async_compute_response() calls has employed asynchronous fd_gradients().
6.27.3.7  

```cpp
void DakotaModel::update_response (DakotaResponse & new_response, const DakotaIntArray & fd_grad_asv, const DakotaIntArray & asv, const short initial_map, 
DakotaRealVector & fn_vals_x0, DakotaRealMatrix & partial_fn_grads, const DakotaRealMatrix & new_fn_grads) [private]
```

Overlay results to update a response object.
Overlay function value and numerical gradient data to populate new response as governed by initial_map flag and asv vectors. If initial_map occurred, then add to the partial response object created by the map. If initial_map was not used, then only new fn_grads should be present in the updated new response. Convenience function used by fd_gradients for the synchronous case and by synchronize fd_gradients for the asynchronous case.

6.27.3.8  

```cpp
void DakotaModel::manage_asv (const DakotaIntArray & asv_in, DakotaIntArray & map_asv_out, DakotaIntArray & fd_grad_asv_out, int & use_fd_grad) [private]
```

Coordinates map() and fd_gradients() calls given an asv_in input.
Splits asv_in total request into map_asv_out for use by map() and fd_grad_asv_out for use by fd_gradients(), as governed by gradient specification.
The documentation for this class was generated from the following files:

- DakotaModel.H
- DakotaModel.C
### 6.28 DakotaNonD Class Reference

Base class for all nondeterministic iterators (the DAKOTA/UQ branch).

```cpp
#include <DakotaNonD.H>
```

Inheritance diagram for DakotaNonD:

```
DakotaNonD
    |_________DakotaIterator
        |_________NonDAdvMeanValue
        |_________NonDPCE
        |_________NonDProbability
        |_________NonDSampling
```

#### Protected Methods

- **DakotaNonD** (**DakotaModel** &model)
  
  *constructor.*

- **DakotaNonD** (**NoDBaseConstructor**, **DakotaModel** &model)
  
  *alternate constructor for instantiations "on the fly".*

- **~DakotaNonD** ()
  
  *destructor.*

- void **run Iterator** ()
  
  *redefines the main iterator hierarchy virtual function to invoke quantify_uncertainty.*

- virtual void **quantify_uncertainty** ()=0
  
  *performs a forward uncertainty propagation of parameter distributions into response statistics.*

- void **vector_statistics** (int num_ran_var, int num_obs, const DakotaRealVectorArray &samples, const DakotaRealArray &thresh)
  
  *computes mean, standard deviation, and probability of threshold exceedance for the samples input, where samples may be input or output-related.*

- const **DakotaResponse** & **iterator_response_results** () const
  
  *return the final statistics from the nondeterministic iteration.*

#### Protected Attributes

- **DakotaRealVector** **normalMeans**
  
  *normal uncertain variable means.*
- DakotaRealVector `normalStdDevs`
  normal uncertain variable standard deviations.

- DakotaRealVector `normalDistLowerBnds`
  normal uncertain variable distribution lower bounds.

- DakotaRealVector `normalDistUpperBnds`
  normal uncertain variable distribution upper bounds.

- DakotaRealVector `lognormalMeans`
  lognormal uncertain variable means.

- DakotaRealVector `lognormalStdDevs`
  lognormal uncertain variable standard deviations.

- DakotaRealVector `lognormalErrFacts`
  lognormal uncertain variable error factors.

- DakotaRealVector `lognormalDistLowerBnds`
  lognormal uncertain variable distribution lower bounds.

- DakotaRealVector `lognormalDistUpperBnds`
  lognormal uncertain variable distribution upper bounds.

- DakotaRealVector `uniformDistLowerBnds`
  uniform uncertain variable distribution lower bounds.

- DakotaRealVector `uniformDistUpperBnds`
  uniform uncertain variable distribution upper bounds.

- DakotaRealVector `loguniformDistLowerBnds`
  loguniform uncertain variable distribution lower bounds.

- DakotaRealVector `loguniformDistUpperBnds`
  loguniform uncertain variable distribution upper bounds.

- DakotaRealVector `weibullAlphas`
  weibull uncertain variable alphas.

- DakotaRealVector `weibullBetas`
  weibull uncertain variable betas.

- DakotaStringList `histogramFilenames`
  histogram uncertain variable filenames.

- DakotaRealMatrix `uncertainCorrelations`
  uncertain variable correlation matrix (rank correlations for sampling and correlation coefficients for analytic reliability).

- `size_t numNormalVars`
number of normal uncertain variables.

- \texttt{size_t numLognormalVars}
  
  number of lognormal uncertain variables.

- \texttt{size_t numUniformVars}
  
  number of uniform uncertain variables.

- \texttt{size_t numLoguniformVars}
  
  number of loguniform uncertain variables.

- \texttt{size_t numWeibullVars}
  
  number of weibull uncertain variables.

- \texttt{size_t numHistogramVars}
  
  number of histogram uncertain variables.

- \texttt{size_t numUncertainVars}
  
  total number of uncertain variables.

- \texttt{size_t numResponseFunctions}
  
  number of response functions.

- \texttt{DakotaRealVector meanStats}
  
  means computed in \texttt{vector\_statistics(\)}.  

- \texttt{DakotaRealVector stdDevStats}
  
  std deviations computed in \texttt{vector\_statistics(\)}.  

- \texttt{DakotaRealVector probMoreThanRespThresh}
  
  probabilities that response functions are greater than respThresh (computed in \texttt{vector\_statistics(\)}).

- \texttt{DakotaResponse finalStatistics}
  

- \texttt{short correlationFlag}
  
  flag for indicating if correlation exists among the uncertain variables.

\subsection{6.28.1 Detailed Description}

Base class for all nondeterministic iterators (the DAKOTA/UQ branch).

The base class for nondeterministic iterators consolidates uncertain variable data and probabilistic utilities for inherited classes.

The documentation for this class was generated from the following files:

- DakotaNonD.H
- DakotaNonD.C
6.29 DakotaOptimizer Class Reference

Base class for the optimizer branch of the iterator hierarchy.

```cpp
#include <DakotaOptimizer.H>
```

Inheritance diagram for DakotaOptimizer:

```
DakotaIterator
 `  
DakotaOptimizer
|
|   CONMINOptimizer
|   DOTOptimizer
|   NPSOLOptimizer
|   SGOPTOptimizer
|   SNLLOptimizer
```

### Public Methods

- void `run_iterator()`
  - run the iterator.

- const DakotaVariables & `iterator_variable_results()` const
  - return the final iteration solution (variables).

- const DakotaResponse & `iterator_response_results()` const
  - return the final iteration solution (response).

- void `print_iterator_results` (ostream &s) const

- void `multi_objective_weights` (const DakotaRealVector &multi_obj_wts)
  - set the relative weightings for multiple objective functions. Used by ConcurrentStrategy for Pareto set optimization.

### Protected Methods

- DakotaOptimizer ()
  - default constructor.

- DakotaOptimizer (DakotaModel &model)
  - standard constructor.

- ~DakotaOptimizer ()
  - destructor.

- virtual void `find_optimum` ()=0
  - Used within the optimizer branch for computing the optimal solution. Redefines the `run_iterator` virtual function for the optimizer branch.
Static Protected Methods

- DakotaResponse multi_objective_modify (const DakotaResponse &raw_response)
  maps multiple objective functions to a single objective for single-objective optimizers.

Protected Attributes

- int realWorkSpaceSize
  size of realWorkSpace.

- int intWorkSpaceSize
  size of intWorkSpace.

- DakotaRealArray realWorkSpace
  real work space for f77 optimizers.

- DakotaIntArray intWorkSpace
  int work space for f77 optimizers.

- size_t numObjectiveFunctions
  number of objective functions.

- Real convergenceTol
  optimizer convergence tolerance.

- Real constraintTol
  optimizer constraint tolerance.

- size_t numNonlinearIneqConstraints
  number of nonlinear inequality constraints.

- DakotaRealVector nonlinearIneqLowerBnds
  nonlinear inequality constraint lower bounds.

- DakotaRealVector nonlinearIneqUpperBnds
  nonlinear inequality constraint upper bounds.

- Real bigBoundSize
  cutoff value for inequality constraint bounds.

- size_t numNonlinearEqConstraints
  number of nonlinear equality constraints.

- DakotaRealVector nonlinearEqTargets
  nonlinear equality constraint targets.

- int numNonlinearConstraints
  total number of nonlinear constraints.
- `int numConstraints`
  
  total number of linear and nonlinear constraints (for DOT/CONMIN).

- `size_t numLinearIneqConstraints`
  
  number of linear inequality constraints.

- `DakotaRealMatrix linearIneqConstraintCoeffs`
  
  linear inequality constraint coefficients.

- `DakotaRealVector linearIneqLowerBnds`
  
  linear inequality constraint lower bounds.

- `DakotaRealVector linearIneqUpperBnds`
  
  linear inequality constraint upper bounds.

- `size_t numLinearEqConstraints`
  
  number of linear equality constraints.

- `DakotaRealMatrix linearEqConstraintCoeffs`
  
  linear equality constraint coefficients.

- `DakotaRealVector linearEqTargets`
  
  linear equality constraint targets.

- `int numLinearConstraints`
  
  total number of linear constraints.

- `int localConstraintArraySize`
  
  used by Fortran optimizers for non-zero array sizing.

- `short speculativeFlag`
  
  flag for speculative optimization approach.

- `short vendorNumericalGradFlag`
  
  convenience flag for gradType=="numerical" && methodSource=="vendor".

- `DakotaVariables bestVariables`
  
  best variables found by optimizer.

- `DakotaResponse bestResponses`
  
  best response found by optimizer.

### Static Protected Attributes

- `size_t staticNumContinuousVars`
  
  static copy of numContinuousVars used in static functions passed by function pointer (NPSOL and OPT++).

- `size_t staticNumObjFns`
static copy of numObjectiveFns used in static functions passed by function pointer (NPSOL and OPT++).

- size_t staticNumNonlinearConstraints
  static copy of numNonlinearConstraints used in static functions passed by function pointer (NPSOL and OPT++).

Private Attributes

- DakotaRealVector multiObjWeights
  user-specified weights for multiple objective functions.

Static Private Attributes

- DakotaRealVector staticMultiObjWeights
  static copy of multiObjWeights used in static functions passed by function pointer (NPSOL and OPT++).

6.29.1 Detailed Description

Base class for the optimizer branch of the iterator hierarchy.
The DakotaOptimizer class provides common data and functionality for DOTOptimizer, NPSOLOptimizer, SNLLOptimizer, and SGLOPTOptimizer.

6.29.2 Constructor & Destructor Documentation

6.29.2.1 DakotaOptimizer::DakotaOptimizer (DakotaModel & model) [protected]

standard constructor.
This constructor extracts the inherited data for the optimizer branch and performs sanity checking on gradient and constraint settings.

6.29.3 Member Function Documentation

6.29.3.1 void DakotaOptimizer::run_iterator () [virtual]

run the iterator.
This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.
Reimplemented from DakotaIterator.
6.29.3.2 void DakotaOptimizer::print_iterator_results (ostream & s) const [virtual]

Redefines default iterator results printing to include optimization results (objective function and constraints).
Reimplemented from DakotaIterator.

6.29.3.3 DakotaResponse DakotaOptimizer::multi_objective_modify (const DakotaResponse & raw_response) [static, protected]

maps multiple objective functions to a single objective for single-objective optimizers.
This function is responsible for the mapping of multiple objective functions into a single objective for publishing to single-objective optimizers. Used in DOTOptimizer, NPSOLOptimizer, SNLLOptimizer, and SGOPTApplication on every function evaluation. The simple weighting approach (using staticMulti-ObjWeights) is the only technique supported currently. The weightings are used to scale function values, gradients, and Hessians as needed.

The documentation for this class was generated from the following files:

- DakotaOptimizer.H
- DakotaOptimizer.C
6.30 DakotaResponse Class Reference

Container class for response functions and their derivatives. DakotaResponse provides the handle class.

```cpp
#include <DakotaResponse.H>
```

Public Methods

- **DakotaResponse ()**
  
  *default constructor.*

- **DakotaResponse (int num_params, const ProblemDescDB &problem_db)**
  
  *standard constructor built from problem description database.*

- **DakotaResponse (int num_params, const DakotaIntArray &asv)**
  
  *alternate constructor using limited data.*

- **DakotaResponse (const DakotaResponse &response)**
  
  *copy constructor.*

- **DakotaResponse ()**
  
  *destructor.*

- **DakotaResponse (const DakotaResponse &response)**

  *assignment operator.*

- **int operator== (const DakotaResponse &response) const**

  *equality operator.*

- **size_t num_functions () const**

  *return the number of response functions.*

- **const DakotaIntArray & active_set_vector () const**

  *return the active set vector.*

- **void active_set_vector (const DakotaIntArray &asv)**

  *set the active set vector.*

- **const DakotaString & interface_id () const**

  *return the interface identifier.*

- **void interface_id (const DakotaString &id)**

  *set the interface identifier.*

- **const DakotaStringArray & fn_tags () const**

  *return the function identifier strings.*
• void fn_tags (const DakotaStringArray &tags) 
  set the function identifier strings.

• const DakotaRealVector & function_values () const 
  return the function values.

• void function_values (const DakotaRealVector &function_yals) 
  set the function values.

• const DakotaRealMatrix & function_gradients () const 
  return the function gradients.

• void function_gradients (const DakotaRealMatrix &function_grads) 
  set the function gradients.

• const DakotaRealMatrixArray & function_hessians () const 
  return the function Hessians.

• void function_hessians (const DakotaRealMatrixArray &function_hessians) 
  set the function Hessians.

• void read (istream &s) 
  read a response object from an istream.

• void write (ostream &s) const 
  write a response object to an ostrm.

• void read_annotated (istream &s) 
  read a response object in annotated format from an istream.

• void write_annotated (ostream &s) const 
  write a response object in annotated format to an ostrm.

• void write_tabular (ostream &s) const 
  write responseRep::functionValues in tabular format to an ostrm.

• void read (DakotaBiStream &s) 
  read a response object from the binary restart stream.

• void write (DakotaBoStream &s) const 
  write a response object to the binary restart stream.

• void read (UnPackBuffer &s) 
  read a response object from a packed MPI buffer.

• void write (PackBuffer &s) const 
  write a response object to a packed MPI buffer.

• DakotaResponse copy () const 
  a deep copy for use in history mechanisms.
- int data_size()
  handle class forward to corresponding body class member function.

- void read_data(double *response_data)
  handle class forward to corresponding body class member function.

- void write_data(double *response_data)
  handle class forward to corresponding body class member function.

- void overlay(const DakotaResponse &response)
  handle class forward to corresponding body class member function.

- void copy_results(const DakotaResponse &response)
  handle class forward to corresponding body class member function.

- void purge_inactive()
  handle class forward to corresponding body class member function.

- void reset()
  handle class forward to corresponding body class member function.

Private Attributes

- DakotaResponseRep * responseRep
  pointer to the body (handle-body idiom).

6.30.1 Detailed Description

Container class for response functions and their derivatives. DakotaResponse provides the handle class.

The DakotaResponse class is a container class for an abstract set of functions (functionValues) and their first
(functionGradients) and second (functionHessians) derivatives. The functions may involve objective and
constraint functions (optimization data set), least squares terms (parameter estimation data set), or generic
response functions (uncertainty quantification data set). It is not currently part of a class hierarchy, since
the abstraction has been sufficiently general and has not required specialization. For memory efficiency, it
employs the "handle-body idioms approach to reference counting and representation sharing (see Coplien
"Advanced C++", p. 58), for which DakotaResponse serves as the handle and DakotaResponseRep serves
as the body.

6.30.2 Constructor & Destructor Documentation
6.30.2.1 DakotaResponse::DakotaResponse ()

default constructor.

Need a populated problem description database to build a meaningful DakotaResponse object, so set the
responseRep=NULL in default constructor for efficiency. This then requires a check on NULL in the copy
constructor, assignment operator, and destructor.

The documentation for this class was generated from the following files:

- DakotaResponse.H
- DakotaResponse.C
6.31 DakotaResponseRep Class Reference

Container class for response functions and their derivatives. DakotaResponseRep provides the body class.  
#include <DakotaResponse.H>

Private Methods

- **DakotaResponseRep ()**  
  default constructor.

- **DakotaResponseRep (int num_params, const ProblemDescDB &problem_db)**  
  standard constructor built from problem description database.

- **DakotaResponseRep (int num_params, const DakotaIntArray &asv)**  
  alternate constructor using limited data.

- **~DakotaResponseRep ()**  
  destructor.

- **void read (istream &s)**  
  read a responseRep object from an istream.

- **void write (ostream &s) const**  
  write a responseRep object to an ostream.

- **void read_annotated (istream &s)**  
  read a responseRep object from an istream (annotated format).

- **void write_annotated (ostream &s) const**  
  write a responseRep object to an ostream (annotated format).

- **void write_tabular (ostream &s) const**  
  write functionValues to an ostream (tabular format).

- **void read (DakotaBiStream &s)**  
  read a responseRep object from a binary stream.

- **void write (DakotaBoStream &s) const**  
  write a responseRep object to a binary stream.

- **void read (UnPackBuffer &s)**  
  read a responseRep object from a packed MPI buffer.

- **void write (PackBuffer &s) const**  
  write a responseRep object to a packed MPI buffer.
- **int data_size ()**
  
  Return the number of doubles active in response. Used for sizing double* response data arrays passed into read_data and write_data.

- **void read_data (double *response_data)**
  
  Read from an incoming double* array.

- **void write_data (double *response_data)**
  
  Write to an incoming double* array.

- **void overlay (const DakotaResponse &response)**
  
  Add incoming response to functionValues/Gradients/Hessians.

- **void copy_results (const DakotaResponse &response)**
  
  Copy functionValues, functionGradients, & functionHessians data only. Do not copy ASV, tags, id's, etc. Used in place of assignment operator for retrieving results data from the data_pairs list without corrupting other data.

- **void purge_inactive ()**
  
  Purge extraneous data from the response object (used when a response object is returned from the database (desired_pair) with more data than needed by the search_pair ASV (see ApplicationInterface::map and DakotaModel::fd_gradients).

- **void reset ()**
  
  Resets functionValues, functionGradients, and functionHessians to zero.

### Private Attributes

- **int referenceCount**
  
  Number of handle objects sharing responseRep.

- **DakotaRealVector functionValues**
  
  Abstract set of functions.

- **DakotaRealMatrix functionGradients**
  
  First derivatives.

- **DakotaRealMatrixArray functionHessians**
  
  Second derivatives.

- **DakotaIntArray responseASV**
  
  Copy of DakotaIterator's activeSetVector needed for operator overloaded I/O.

- **DakotaStringArray fnTags**
  
  Function identifiers used to improve output readability.

- **DakotaString interfaceId**
  
  The interface used to generate this response object. Used in PRPair::vars_asv_compare.
Friends

- class DakotaResponse
  
  *the handle class can access attributes of the body class directly.*

### 6.31.1 Detailed Description

Container class for response functions and their derivatives. DakotaResponseRep provides the body class. The DakotaResponseRep class is the "representation" of the response container class. It is the "body" portion of the "handle-body idiom" (see Coplien "Advanced C++", p. 58). The handle class (DakotaResponse) provides for memory efficiency in management of multiple response objects through reference counting and representation sharing. The body class (DakotaResponseRep) actually contains the response data (functionValues, functionGradients, functionHessians, etc.). The representation is hidden in that an instance of DakotaResponseRep may only be created by DakotaResponse. Therefore, programmers create instances of the DakotaResponse handle class, and only need to be aware of the handle/body mechanisms when it comes to managing shallow copies (shared representation) versus deep copies (separate representation used for history mechanisms).

### 6.31.2 Constructor & Destructor Documentation

#### 6.31.2.1 DakotaResponseRep::DakotaResponseRep (int num_params, const ProblemDescDB & problem_db) [private]

standard constructor built from problem description database.

The standard constructor used by DakotaModelRep. An interfaceId identifies a set of results with the interface used in generating them, which allows vars\_asv\_compare to prevent duplicate detection on results from different interfaces.

#### 6.31.2.2 DakotaResponseRep::DakotaResponseRep (int num_params, const DakotaIntArray & asv) [private]

alternate constructor using limited data.

Used for building a response object of the correct size on the fly (e.g., by slave analysis servers performing execute() on a local response). fnTags and interfaceId are not needed for this purpose since they're not passed in the MPI send/recv buffers (NOTE: if interfaceId becomes needed, it could be set from an AppInt attribute passed from AppInt::serve()). However, NPSOLOptimizer's user-defined functions option uses this constructor to build bestResponses and bestResponses needs fnTags for I/O, so construction of fnTags has been added.

### 6.31.3 Member Function Documentation
6.31.3.1 void DakotaResponseRep::read (istream & s)  [private]

read a responseRep object from an istream.
ASCII version of read needs capabilities for capturing data omissions or formatting errors (resulting from user error or async race condition) and analysis failures (resulting from nonconvergence, instability, etc.).

6.31.3.2 void DakotaResponseRep::write (ostream & s) const  [private]

write a responseRep object to an ostream.
ASCII version of write.

6.31.3.3 void DakotaResponseRep::read_annotated (istream & s)  [private]

read a responseRep object from an istream (annotated format).
read_annotated version is used for neutral file translation of restart files. Since objects are built solely from this data, annotations are used. This version is currently identical to the DakotaBiStream version.

6.31.3.4 void DakotaResponseRep::write_annotated (ostream & s) const  [private]

write a responseRep object to an ostream (annotated format).
write_annotated version is used for neutral file translation of restart files. Since objects need to be build solely from this data, annotations are used. This version differs from the DakotaBoStream version only in the use of white space between fields.

6.31.3.5 void DakotaResponseRep::write_tabular (ostream & s) const  [private]

write functionValues to an ostream (tabular format).
write_tabular is used for output of functionValues in a tabular format for convenience in post-processing/plotting of DAKOTA results. Objects are not built from this data (there is no corresponding read_tabular).

6.31.3.6 void DakotaResponseRep::read (DakotaBiStream & s)  [private]

read a responseRep object from a binary stream.
Binary version differs from ASCII version in 2 primary ways: (1) it lacks formatting. (2) the DakotaResponse has not been sized a priori. In reading data from the binary restart file, a ParamResponsePair was constructed with its default constructor which called the DakotaResponse default constructor. Therefore, we must first read sizing data and resize the arrays.

6.31.3.7 void DakotaResponseRep::write (DakotaBoStream & s) const  [private]

write a responseRep object to a binary stream.
Binary version differs from ASCII version in 2 primary ways: (1) It lacks formatting. (2) In reading data from the binary restart file, ParamResponsePairs are constructed with their default constructor which calls
the `DakotaResponse` default constructor. Therefore, we must first write sizing data so that DakotaResponse-Rep::read(DakotaBoStream& s) can resize the arrays.

### 6.31.3.8 void DakotaResponseRep::read (UnPackBuffer & s) [private]

read a responseRep object from a packed MPI buffer.

UnpackBuffer version differs from `DakotaBiStream` version only in omission of interfaceId and default fnTags. Master processor retains tags and ids and communicates asv and response data only with slaves.

### 6.31.3.9 void DakotaResponseRep::write (PackBuffer & s) const [private]

write a responseRep object to a packed MPI buffer.

PackBuffer version differs from `DakotaBoStream` version only in omissions of interfaceId and flush. The master processor retains tags and ids and communicates asv and response data only with slaves.

The documentation for this class was generated from the following files:

- DakotaResponse.H
- DakotaResponse.C
6.32 DakotaStrategy Class Reference

Base class for the strategy class hierarchy.

```cpp
#include <DakotaStrategy.H>
```

Inheritance diagram for DakotaStrategy:

```
DakotaStrategy
  ├── BranchBndStrategy
  │      └── ConcurrentStrategy
  │          ├── MultiLevelOptStrategy
  │          │      └── NonDOptStrategy
  │          │          └── SingleMethodStrategy
  │          │              └── SurrBasedOptStrategy
```

**Public Methods**

- **DakotaStrategy ()**
  
  *default constructor (should not be used).*

- **DakotaStrategy (ProblemDescDB &problem_db)**
  
  *constructor.*

- **DakotaStrategy (const DakotaStrategy &strat)**
  
  *copy constructor.*

- **virtual ~DakotaStrategy ()**
  
  *destructor.*

- **DakotaStrategy operator= (const DakotaStrategy &strat)**
  
  *assignment operator.*

- **virtual void run_strategy ()**
  
  *the run function for the strategy: invoke the iterator(s) on the model(s). Called from main.C.*

- **void run_iterator (DakotaIterator &the_iterator, DakotaModel &the_model)**
  
  *Convenience function for invoking an iterator on a model and managing parallelism. Function must be public due to use by MINLPNode.*

- **MPI_Comm iterator_communicator () const**
  
  *return iteratorComm (used only by MINLPNode).*

- **int iterator_communicator_size () const**
  
  *return iteratorCommSize (used only by MINLPNode).*
Protected Methods

- **DakotaStrategy** *(BaseConstructor, ProblemDescDB &probDescDB)*
  
  Constructor initializes the base class part of letter classes *(BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).*

- void **initialize_graphics** (const DakotaModel &model)
  
  Convenience function for initialization of 2D graphics and data tabulation.

Protected Attributes

- **ProblemDescDB & probDescDB**
  
  Class member reference to the problem description database.

- **ParallelLibrary & parallelLib**
  
  Class member reference to the parallel library.

- **DakotaString strategyName**
  
  Type of strategy: single_method, multi_level, surrogate_based_opt, opt_under_uncertainty, branch_and_bound, multi_start, or pareto_set.

- int **worldRank**
  
  Processor rank in MPI_COMM_WORLD.

- int **worldSize**
  
  Size of MPI_COMM_WORLD.

- **MPI_Comm iteratorComm**
  
  The communicator defining the group of processors on which an iterator executes. Results from init_iterator_comms.

- int **iteratorCommRank**
  
  Processor rank in iteratorComm.

- int **iteratorCommSize**
  
  Number of processors in iteratorComm.

- short **graphicsFlag**
  
  Flag for using graphics in a graphics executable.

- short **tabularDataFlag**
  
  Flag for file tabulation of graphics data.

- **DakotaString tabularDataFile**
  
  Filename for tabulation of graphics data.
Private Methods

- DakotaStrategy * get_strategy (ProblemDescDB &problem_db)
  
  Used by the envelope to instantiate the correct letter class.

- ProblemDescDB & prob_desc_db () const
  
  returns the problem description database (probDescDB).

Private Attributes

- DakotaStrategy * strategyRep
  
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  
  number of objects sharing strategyRep.

6.32.1 Detailed Description

Base class for the strategy class hierarchy.

The DakotaStrategy class is the base class for the class hierarchy providing the top level control in DAKOTA. The strategy is responsible for creating and managing iterators and models. For memory efficiency and enhanced polymorphism, the strategy hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaStrategy) serves as the envelope and one of the derived classes (selected in DakotaStrategy::get_strategy()) serves as the letter.

6.32.2 Constructor & Destructor Documentation

6.32.2.1 DakotaStrategy::DakotaStrategy ()

default constructor (should not be used).

The default constructor should not be used. strategyRep is NULL in this case (a populated problem db is needed to build a meaningful DakotaStrategy object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

6.32.2.2 DakotaStrategy::DakotaStrategy (ProblemDescDB & problem_db)

constructor.

Used in main.C instantiation to build the envelope. This constructor only needs to extract enough data to properly execute get_strategy, since DakotaStrategy::DakotaStrategy(BaseConstructor, problem_db) builds the actual base class data inherited by the derived strategies.
6.32.3 DakotaStrategy::DakotaStrategy (const DakotaStrategy & strat)

copy constructor.
Copy constructor manages sharing of strategyRep and incrementing of referenceCount.

6.32.4 DakotaStrategy::~DakotaStrategy () [virtual]
destructor.
Destructor decrements referenceCount and only deletes strategyRep when referenceCount reaches zero.

6.32.5 DakotaStrategy::DakotaStrategy (BaseConstructor, ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor is the one which must build the base class data for all inherited strategies. get_strategy() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_strategy() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaStrategy).

6.32.3 Member Function Documentation

6.32.3.1 DakotaStrategy DakotaStrategy::operator= (const DakotaStrategy & strat)

assignment operator.

6.32.3.2 void DakotaStrategy::run_iterator (DakotaIter & the_iterator, DakotaModel & the_model)

Convenience function for invoking an iterator on a model and managing parallelism. Function must be public due to use by MINLPNode.
This is a convenience function for encapsulating the parallel features (init/serve/etc.) of running an iterator. It does not require a strategyRep forward since it is only used by letter objects. While it is currently a public function due to its use in MINLPNode, this usage still involves a strategy letter object.

6.32.3.3 void DakotaStrategy::initialize_graphics (const DakotaModel & model) [protected]

convenience function for initialization of 2D graphics and data tabulation.
This is a convenience function for encapsulating graphics initialization operations. It is not a public function for which a strategyRep forward would be needed, rather it is used exclusively by letter objects.
6.32.3.4 DakotaStrategy * DakotaStrategy::get_strategy (ProblemDescDB & problem_db) [private]

Used by the envelope to instantiate the correct letter class.
Used only by the envelope constructor to initialize strategyRep to the appropriate derived type, as given by the strategyName attribute.

6.32.3.5 ProblemDescDB & DakotaStrategy::prob_desc_db () const [inline, private]

returns the problem description database (probDescDB).
Used only by the copy constructor (otherwise strategyRep forward needed).
The documentation for this class was generated from the following files:

- DakotaStrategy.H
- DakotaStrategy.C
6.33 DakotaString Class Reference

DakotaString class, used as main string class for Dakota.

#include <DakotaString.h>

Public Methods

- DakotaString ()
  Default constructor.

- DakotaString (const DakotaString &a)
  Default copy constructor.

- DakotaString (const char *initial_val)
  Copy constructor from standard C char array.

- ~DakotaString ()
  Destructor.

- void testClass ()
  Class unit test method.

- DakotaString & toUpper ()
  Convert to upper case string.

- void upper ()
- DakotaString & toLower ()
  Convert to lower case string.

- void lower ()
- bool contains (const char *subString) const
  Returns true if DakotaString contains char* substring.

- bool isNull () const
  Returns true of DakotaString is empty.

- char * data () const
  Returns pointer to standard C char array.

- DakotaString & operator= (const DakotaString &)
  Normal assignment operator.

- DakotaString & operator= (const DAKOTA_BASE_STRING &)
  Assignment operator for base string.

- DakotaString & operator= (const char *)
6.33 DakotaString Class Reference

Assignment operator, standard C char*

- operator const char * () const
  
  The operator() returns pointer to standard C char array.

6.33.1 Detailed Description

DakotaString class, used as main string class for Dakota.

The DakotaString class is the common string class for Dakota. It provides a common interface for string operations whether inheriting from the STL basic_string or the Rogue Wave RWCSString class.

6.33.2 Member Function Documentation

6.33.2.1 void DakotaString::testClass ()

Class unit test method.

Unit test method for the DakotaString class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will abort if an unexpected answer is received.

6.33.2.2 void DakotaString::upper ()

Private method which converts DakotaString to upper. Utilizes a STL iterator to step through the string and then calls the STL toupper() method. Needs to be done this way because STL only provides a single char toupper method.

6.33.2.3 void DakotaString::lower ()

Private method which converts DakotaString to lower. Utilizes a STL iterator to step through the string and then calls the STL tolower() method. Needs to be done this way because STL only provides a single char tolower method.

6.33.2.4 bool DakotaString::contains (const char * subString) const

Returns true if DakotaString contains char* substring.

Returns true of the DakotaString contains the char* substring. Calls the STL rfind() method, then checks if substring was found within the DakotaString.

6.33.2.5 char * DakotaString::data () const

Returns pointer to standard C char array.

Returns a pointer to c style char array. Needed to mimic the Rogue Wave string class. USE WITH CARE.
6.33.2.6 DakotaString::operator const char * () const

The operator() returns pointer to standard C char array.

The operator() returns a pointer to a char string. Uses the STL c.str() method. This allows for the Dakota-String to be used in method calls without having to call the data() or c.str() methods.

The documentation for this class was generated from the following files:

- DakotaString.H
- DakotaString.C
6.34 DakotaVarConstraints Class Reference

Base class for the variable constraints class hierarchy.

#include <DakotaVarConstraints.H>

Inheritance diagram for DakotaVarConstraints::

```
DakotaVarConstraints
AllMergedVarConstraints  AllVarConstraints  FundamentalVarConstraints  MergedVarConstraints
```

Public Methods

- **DakotaVarConstraints ()**
  
  *default constructor.*

- **DakotaVarConstraints (const ProblemDescDB &problem_db, const DakotaString &vars_type)**
  
  *standard constructor.*

- **DakotaVarConstraints (const DakotaVarConstraints &vc)**
  
  *copy constructor.*

- **virtual ~DakotaVarConstraints ()**
  
  *destructor.*

- **DakotaVarConstraints operator= (const DakotaVarConstraints &vc)**
  
  *assignment operator.*

- **virtual const DakotaRealVector & continuous_lower_bounds () const**
  
  *return the active continuous variable lower bounds.*

- **virtual void continuous_lower_bounds (const DakotaRealVector &c_l_bnds)**
  
  *set the active continuous variable lower bounds.*

- **virtual const DakotaRealVector & continuous_upper_bounds () const**
  
  *return the active continuous variable upper bounds.*

- **virtual void continuous_upper_bounds (const DakotaRealVector &c_u_bnds)**
  
  *set the active continuous variable upper bounds.*

- **virtual const DakotaIntVector & discrete_lower_bounds () const**
  
  *return the active discrete variable lower bounds.*

- **virtual void discrete_lower_bounds (const DakotaIntVector &d_l_bnds)**
  
  *set the active discrete variable lower bounds.*
virtual const DakotaIntVector & discrete_upper_bounds() const
return the active discrete variable upper bounds.

virtual void discrete_upper_bounds(const DakotaIntVector &dubnds)
set the active discrete variable upper bounds.

virtual void write (ostream &s) const
write a variable constraints object to an ostream.

virtual void read (istream &s)
read a variable constraints object from an istream.

const size_t & num_linear_ineq_constraints() const
return the number of linear inequality constraints.

const size_t & num_linear_eq_constraints() const
return the number of linear equality constraints.

const DakotaRealMatrix & linear_ineq_constraint_coeffs() const
return the linear inequality constraint coefficients.

const DakotaRealVector & linear_ineq_constraint_lower_bounds() const
return the linear inequality constraint lower bounds.

const DakotaRealVector & linear_ineq_constraint_upper_bounds() const
return the linear inequality constraint upper bounds.

const DakotaRealMatrix & linear_eq_constraint_coeffs() const
return the linear equality constraint coefficients.

const DakotaRealVector & linear_eq_constraint_targets() const
return the linear equality constraint targets.

Protected Methods

- DakotaVarConstraints (BaseConstructor, const ProblemDescDB &problem_db)
  constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).

- void manage_linear_constraints (const ProblemDescDB &problem_db, const size_t &num_vars)
  perform checks on user input, convert linear constraint coefficient input to matrices, and assign defaults.

- size_t num_active_variables() const
  return number of active variables.
Protected Attributes

- DakotaString variablesType
  
  All, Merged, AllMerged, or Fundamental.

- short discreteFlag
  
  flags discrete variable mode.

- size numLinearIneqConstraints
  
  number of linear inequality constraints.

- size numLinearEqConstraints
  
  number of linear equality constraints.

- DakotaRealMatrix linearIneqConstraintCoeffs
  
  linear inequality constraint coefficients.

- DakotaRealMatrix linearEqConstraintCoeffs
  
  linear equality constraint coefficients.

- DakotaRealVector linearIneqConstraintLowerBnds
  
  linear inequality constraint lower bounds.

- DakotaRealVector linearIneqConstraintUpperBnds
  
  linear inequality constraint upper bounds.

- DakotaRealVector linearEqConstraintTargets
  
  linear equality constraint targets.

Private Methods

- DakotaVarConstraints * get_var_constraints (const ProblemDescDB &problem_db)
  
  Used only by the constructor to initialize varConstraintsRep to the appropriate derived type.

Private Attributes

- DakotaVarConstraints * varConstraintsRep
  
  pointer to the letter (initialized only for the envelope).

- int referenceCount
  
  number of objects sharing varConstraintsRep.
6.34.1 Detailed Description

Base class for the variable constraints class hierarchy.

The DakotaVarConstraints class is the base class for the class hierarchy managing linear and bound constraints on the variables. Using the variable lower and upper bounds arrays and linear constraint coefficients and bounds from the input specification, different derived classes define different views of this data. For memory efficiency and enhanced polymorphism, the variable constraints hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaVarConstraints) serves as the envelope and one of the derived classes (selected in DakotaVarConstraints::getVarConstraints()) serves as the letter.

6.34.2 Constructor & Destructor Documentation

6.34.2.1 DakotaVarConstraints::DakotaVarConstraints()

default constructor.

The default constructor: varConstraintsRep is NULL in this case (a populated problem\_db is needed to build a meaningful DakotaVarConstraints object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.

6.34.2.2 DakotaVarConstraints::DakotaVarConstraints(const ProblemDescDB & problem\_db, const DakotaString & vars\_type)

standard constructor.

The envelope constructor only needs to extract enough data to properly execute get\_var\_constraints, since the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived classes.

6.34.2.3 DakotaVarConstraints::DakotaVarConstraints(const DakotaVarConstraints & vc)

copy constructor.

Copy constructor manages sharing of varConstraintsRep and incrementing of referenceCount.

6.34.2.4 DakotaVarConstraints::~DakotaVarConstraints() [virtual]
destructor.

Destructor decrements referenceCount and only deletes varConstraintsRep when referenceCount reaches zero.

6.34.2.5 DakotaVarConstraints::DakotaVarConstraints(BaseConstructor, const ProblemDescDB & problem\_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor is the one which must build the base class data for all derived classes. `get_var_constraints()` instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid recursion in the base class constructor calling `get_var_constraints()` again). Since the letter IS the representation, its rep pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaVarConstraints).

### 6.34.3 Member Function Documentation

#### 6.34.3.1 DakotaVarConstraints DakotaVarConstraints::operator=(const DakotaVarConstraints & vc)

assignment operator.


#### 6.34.3.2 void DakotaVarConstraints::manage_linear_constraints (const ProblemDescDB & problem_db, const size_t & num_vars) [protected]

perform checks on user input, convert linear constraint coefficient input to matrices, and assign defaults.

Convenience function called from derived class constructors. The number of variables active for applying linear constraints is passed up from the particular derived class.

#### 6.34.3.3 DakotaVarConstraints * DakotaVarConstraints::get_var_constraints (const ProblemDescDB & problem_db) [private]

Used only by the constructor to initialize varConstraintsRep to the appropriate derived type.

Initializes varConstraintsRep to the appropriate derived type, as given by the variablesType attribute.

The documentation for this class was generated from the following files:

- DakotaVarConstraints.H
- DakotaVarConstraints.C
6.35 DakotaVariables Class Reference

Base class for the variables class hierarchy.

#include <DakotaVariables.H>

Inheritance diagram for DakotaVariables::

![Inheritance diagram](image)

### Public Methods

- **DakotaVariables ()**
  
  *default constructor.*

- **DakotaVariables (const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **DakotaVariables (const DakotaString &vars_type)**
  
  *alternate constructor.*

- **DakotaVariables (const DakotaVariables &vars)**
  
  *copy constructor.*

- **virtual ~DakotaVariables ()**
  
  *destructor.*

- **DakotaVariables operator= (const DakotaVariables &vars)**

  *assignment operator.*

- **virtual size_t tv () const**

  *Returns total number of vars.*

- **virtual size_t cv () const**

  *Returns number of active continuous vars.*

- **virtual size_t dv () const**

  *Returns number of active discrete vars.*

- **virtual const DakotaRealVector & continuous_variables () const**

  *return the active continuous variables.*

- **virtual void continuous_variables (const DakotaRealVector &c_vars)**
set the active continuous variables.

- virtual const DakotaIntVector & discrete_variables() const
  return the active discrete variables.

- virtual void discrete_variables(const DakotaIntVector &d_vars)
  set the active discrete variables.

- virtual const DakotaStringArray & continuous_variable_labels() const
  return the active continuous variable labels.

- virtual void continuous_variable_labels(const DakotaStringArray &cv_labels)
  set the active continuous variable labels.

- virtual const DakotaStringArray & discrete_variable_labels() const
  return the active discrete variable labels.

- virtual void discrete_variable_labels(const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

- virtual const DakotaRealVector & inactive_continuous_variables() const
  return the inactive continuous variables.

- virtual void inactive_continuous_variables(const DakotaRealVector &i_c_vars)
  set the inactive continuous variables.

- virtual const DakotaIntVector & inactive_discrete_variables() const
  return the inactive discrete variables.

- virtual void inactive_discrete_variables(const DakotaIntVector &i_d_vars)
  set the inactive discrete variables.

- virtual size_t acv() const
  returns total number of continuous vars.

- virtual size_t adv() const
  returns total number of discrete vars.

- virtual DakotaRealVector all_continuous_variables() const
  returns a single array with all continuous variables.

- virtual DakotaIntVector all_discrete_variables() const
  returns a single array with all discrete variables.

- virtual void read(istream &s)
  read a variables object from an istream.

- virtual void write(ostream &s) const
  write a variables object to an ostream.
- virtual void `read_annotated` (istream &s)  
  *read a variables object in annotated format from an stream.*

- virtual void `write_annotated` (ostream &s) const  
  *write a variables object in annotated format to an ostream.*

- virtual void `read` (DakotaBiStream &s)  
  *read a variables object from the binary restart stream.*

- virtual void `write` (DakotaBoStream &s) const  
  *write a variables object to the binary restart stream.*

- virtual void `read` (UnPackBuffer &s)  
  *read a variables object from a packed MPI buffer.*

- virtual void `write` (PackBuffer &s) const  
  *write a variables object to a packed MPI buffer.*

- void `write_tabular` (ostream &s) const  
  *write a variables object in tabular format to an ostream.*

- `DakotaVariables` `copy` () const  
  *for use when a true copy is needed (the representation is not shared).*

- const `DakotaIntList` & `merged_integer_list` () const  
  *returns the list of discrete variables merged into a continuous array.*

- const `DakotaString` & `variables_type` () const  
  *returns the variables type: All, Merged, AllMerged, or Fundamental.*

**Protected Methods**

- `DakotaVariables` (BaseConstructor, const ProblemDescDB &problem_db)  
  *constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).*

**Protected Attributes**

- `DakotaIntList` mergedIntegerList  
  *the list of discrete variables for which integrality is relaxed by merging them into a continuous array.*

- `DakotaString` variablesType  
  *All, Merged, AllMerged, or Fundamental.*

- short `apreproFlag`  
  *used to trigger special behavior in `write` (ostream &).*
Private Methods

- virtual void copy_rep (const DakotaVariables *vars_rep)
  
  Used by copy() to copy the contents of a letter class.

- DakotaVariables * get_variables (const ProblemDescDB &problem_db)

  Used by the standard envelope constructor to instantiate the correct letter class.

- DakotaVariables * get_variables (const DakotaString &vars_type) const

  Used by the alternate envelope constructor, by read functions, and by copy() to instantiate a new letter class.

Private Attributes

- DakotaVariables * variablesRep

  pointer to the letter (initialized only for the envelope).

- int referenceCount

  number of objects sharing variablesRep.

Friends

- int operator== (const DakotaVariables &vars1, const DakotaVariables &vars2)

  equality operator.

6.35.1 Detailed Description

Base class for the variables class hierarchy.

The DakotaVariables class is the base class for the class hierarchy providing design, uncertain, and state variables for continuous and discrete domains within a DakotaModel. Using the fundamental arrays from the input specification, different derived classes define different views of the data. For memory efficiency and enhanced polymorphism, the variables hierarchy employs the "letter/envelope idiom" (see Coplien "Advanced C++", p. 133), for which the base class (DakotaVariables) serves as the envelope and one of the derived classes (selected in DakotaVariables::get_variables()) serves as the letter.

6.35.2 Constructor & Destructor Documentation

6.35.2.1 DakotaVariables::DakotaVariables ()

default constructor.

The default constructor: variablesRep is NULL in this case (a populated problem_db is needed to build a meaningful DakotaVariables object). This makes it necessary to check for NULL in the copy constructor, assignment operator, and destructor.
6.35.2.2 DakotaVariables::DakotaVariables (const ProblemDescDB & problem_db)

standard constructor.
This is the primary envelope constructor which uses problem_db to build a fully populated variables object. It only needs to extract enough data to properly execute get_variables(problem_db), since the constructor overloaded with BaseConstructor builds the actual base class data inherited by the derived classes.

6.35.2.3 DakotaVariables::DakotaVariables (const DakotaString & vars_type)

alternate constructor.
This is the alternate envelope constructor for instantiations on the fly. Since it does not have access to problem_db, the letter class is not fully populated. This constructor executes get_variables(vars_type), which invokes the default constructor of the derived letter class, which in turn invokes the default constructor of the base class.

6.35.2.4 DakotaVariables::DakotaVariables (const DakotaVariables & vars)

copy constructor.
Copy constructor manages sharing of variablesRep and incrementing of referenceCount.

6.35.2.5 DakotaVariables::~DakotaVariables () [virtual]

destructor.
Destructor decrements referenceCount and only deletes variablesRep when referenceCount reaches zero.

6.35.2.6 DakotaVariables::DakotaVariables (BaseConstructor, const ProblemDescDB & problem_db) [protected]

constructor initializes the base class part of letter classes (BaseConstructor overloading avoids infinite recursion in the derived class constructors - Coplien, p. 139).
This constructor is the one which must build the base class data for all derived classes. get_variables() instantiates a derived class letter and the derived constructor selects this base class constructor in its initialization list (to avoid the recursion of the base class constructor calling get_variables() again). Since the letter IS the representation, its representation pointer is set to NULL (an uninitialized pointer causes problems in ~DakotaVariables).

6.35.3 Member Function Documentation

6.35.3.1 DakotaVariables DakotaVariables::operator= (const DakotaVariables & vars)

assignment operator.
6.35.3.2 DakotaVariables DakotaVariables::copy () const

for use when a true copy is needed (the representation is not shared).
Deep copies are used for history mechanisms such as bestVariables and data pairs since these must catalogue copies (and should not change as the representation within currentVariables changes).

6.35.3.3 DakotaVariables & DakotaVariables::get_variables (const ProblemDescDB & problem_db) [private]

Used by the standard envelope constructor to instantiate the correct letter class.
Initializes variablesRep to the appropriate derived type, as given by problem_db attributes. The standard derived class constructors are invoked.

6.35.3.4 DakotaVariables & DakotaVariables::get_variables (const DakotaString & vars_type) const [private]

Used by the alternate envelope constructor, by read functions, and by copy() to instantiate a new letter class.
Initializes variablesRep to the appropriate derived type, as given by the vars_type attribute. The default derived class constructors are invoked.

The documentation for this class was generated from the following files:

- DakotaVariables.H
- DakotaVariables.C
6.36 DakotaVector Class Template Reference

Template class for the Dakota numerical vector.

```cpp
#include <DakotaVector.H>
```

Inheritance diagram for DakotaVector:

```
DakotaBaseVector< T >
DakotaVector
```

Public Methods

- **DakotaVector ()**
  
  Default constructor.

- **DakotaVector (size_t size)**
  
  Constructor which takes an initial size.

- **DakotaVector (size_t size, const T &initial_val)**
  
  Constructor which takes an initial size and an initial value.

- **DakotaVector (const DakotaVector< T > &a)**
  
  Copy constructor.

- **DakotaVector (const T *p, size_t size)**
  
  Constructor, creates array of size, with initial value <T> p.

- **~DakotaVector ()**
  
  Destructor.

- **void testClass ()**
  
  Class unit test method.

- **void read (istream &s)**
  
  Reads a DakotaVector from an input stream.

- **void read (istream &s, DakotaArray< DakotaString > &label_array)**
  
  Reads a DakotaVector and associated label array from an input stream.

- **void read_partial (istream &s, size_t start_index, size_t num_items)**
  
  Reads part of a DakotaVector from an input stream.
- void `read_partial` (istream &s, size_t start_index, size_t num_items, DakotaArray & DakotaString > &label_array)
  
  Reads part of a DakotaVector and the corresponding labels from an input stream.

- void `read_annotated` (istream &s, DakotaArray < DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array in annotated from an input stream.

- void `print` (ostream &s) const
  
  Prints a DakotaVector to an output stream.

- void `print` (ostream &s, const DakotaArray < DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array to an output stream.

- void `print_partial` (ostream &s, size_t start_index, size_t num_items) const
  
  Prints part of a DakotaVector to an output stream.

- void `print_partial` (ostream &s, size_t start_index, size_t num_items, const DakotaArray < DakotaString > &label_array) const
  
  Prints part of a DakotaVector and the corresponding labels to an output stream.

- void `print_aprepro` (ostream &s, const DakotaArray < DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array to an output stream in aprepro format.

- void `print_partial_aprepro` (ostream &s, size_t start_index, size_t num_items, const DakotaArray < DakotaString > &label_array) const
  
  Prints part of a DakotaVector and the corresponding labels to an output stream in aprepro format.

- void `print_annotated` (ostream &s, const DakotaArray < DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array in annotated form to an output stream.

- void `read` (DakotaBiStream &s, DakotaArray < DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array from a binary input stream.

- void `print` (DakotaBoStream &s, const DakotaArray < DakotaString > &label_array) const
  
  Prints a DakotaVector and associated label array to a binary output stream.

- void `read` (UnPackBuffer &s)
  
  Reads a DakotaVector from a buffer after an MPI receive.

- void `read` (UnPackBuffer &s, DakotaArray < DakotaString > &label_array)
  
  Reads a DakotaVector and associated label array from a buffer after an MPI receive.

- void `print` (PackBuffer &s) const
  
  Writes a DakotaVector to a buffer prior to an MPI send.

- void `print` (PackBuffer &s, const DakotaArray < DakotaString > &label_array) const
  
  Writes a DakotaVector and associated label array to a buffer prior to an MPI send.

- DakotaVector < T > & operator=(const DakotaVector < T > &a)
  
  Normal const assignment operator.
DakotaVector\< T \> & operator= (const T &ival)

Sets all elements in self to the value ival.

operator T * () const

Converts the DakotaVector to a standard C-style array. Use with care!

**Private Methods**

- void copy_array (const T *p, size_t size)

Deep copies the array pointed to by \{p\} into this array. Private function for \{operator=(const T *p)\} and the constructor \{DakotaVector(const T *p, size_t size)\}.

### 6.36.1 Detailed Description

template<class T> class DakotaVector\< T \>

Template class for the Dakota numerical vector. The DakotaVector class is the numeric vector class. It inherits from the common vector class DakotaBase vector which provides the same interface for both the STL and RW vector classes. If the STL version of DakotaBaseVector is based on the valarray class then some basic vector operations such as +, * are available. This class adds functionality to read/print vectors in a variety of ways

### 6.36.2 Constructor & Destructor Documentation

#### 6.36.2.1 template<class T> DakotaVector\< T \>::DakotaVector (const T * p, size_t size)

Constructor, creates array of size, with initial value \<T> p. Assigns up to size values in array to p, uses the private copy_array method

### 6.36.3 Member Function Documentation

#### 6.36.3.1 template<class T> void DakotaVector\< T \>::testClass ()

Class unit test method. Unit test method for the DakotaVector class. Provides a quick way to test the basic functionality of the class. Utilizes the assert function to test for correctness, will fail if an unexpected answer is received.
6.36.3.2 template<class T> DakotaVector<T> & DakotaVector<T>::operator=(const T & ival)

Sets all elements in self to the value ival.
Assigns all values of array to ival. If STL, uses the vector assign method because there is no operator=(ival).
Reimplemented from DakotaBaseVector.

6.36.3.3 template<class T> void DakotaVector<T>::copy_array (const T * p, size_t size)
[private]

Deep copies the array pointed to by {p} into this array. Private function for {operator=(const T *p)} and the constructor {DakotaVector(const T* p, size_t size)}.
First resizes the vector to correct size and then assigns each value using the operator[ ](i).
The documentation for this class was generated from the following file:

- DakotaVector.H
6.37 DataInterface Class Reference

Container class for interface specification data.

#include <DataInterface.H>

Public Methods

- **DataInterface ()**
  constructor.

- **DataInterface (const DataInterface &)**
  copy constructor.

- **~DataInterface ()**
  destructor.

- **DataInterface & operator= (const DataInterface &)**
  assignment operator.

- **int operator== (const DataInterface &)**
  equality operator.

- **void write (ostream &s)**
  write a DataInterface object to an ostream.

- **void read (UnPackBuffer &s)**
  read a DataInterface object from a packed MPI buffer.

- **void write (PackBuffer &s)**
  write a DataInterface object to a packed MPI buffer.

Public Attributes

- **DakotaString interfaceType**
  the interface selection: application_system/fork/direct/xml or approximation_ann/rsn/mars/hermite/ksm/mpa/taylor/hierarchical.

- **DakotaString idInterface**
  string identifier for an interface specification data set (from the id_interface specification in InterfSetId).

- **DakotaString inputFilter**
  the input filter for a simulation-based interface (from the input_filter specification in InterfAppl).

- **DakotaString outputFilter**
the output filter for a simulation-based interface (from the output_filter specification in InterfAppic).

- **DakotaStringList analysisDrivers**
  the set of analysis drivers for a simulation-based interface (from the analysis_drivers specification in InterfAppic).

- **DakotaString parametersFile**
  the parameters file for system call and fork interfaces (from the parameters_file specification in InterfAppic).

- **DakotaString resultsFile**
  the results file for system call and fork interfaces (from the results_file specification in InterfAppic).

- **DakotaString analysisUsage**
  the analysis command usage string for a system call interface (from the analysis_usage specification in InterfAppic).

- **short apreproFormatFlag**
  the flag for aprepro format usage in the parameters file for system call and fork interfaces (from the aprepro specification in InterfAppic).

- **short fileTagFlag**
  the flag for file tagging of parameters and results files for system call and fork interfaces (from the file_tag specification in InterfAppic).

- **short fileSaveFlag**
  the flag for saving of parameters and results files for system call and fork interfaces (from the file_save specification in InterfAppic).

- **int procsPerAnalysis**
  processors per parallel analysis for a direct interface (from the processors_per_analysis specification in InterfAppic).

- **DakotaStringList xmlHostNames**
  names of host machines for an XML interface (from the hostnames specification in InterfAppic).

- **DakotaIntArray xmlProcsPerHost**
  processors per host machine for an XML interface (from the processors_per_host specification in InterfAppic).

- **DakotaString interfaceSynchronization**
  parallel mode for a simulation-based interface: synchronous or asynchronous (from the asynchronous specification in InterfAppic).

- **int asynchLocalEvalConcurrency**
  evaluation concurrency for asynchronous simulation-based interfaces (from the evaluation_concurrency specification in InterfAppic).

- **int asynchLocalAnalysisConcurrency**
  analysis concurrency for asynchronous simulation-based interfaces (from the analysis_concurrency specification in InterfAppic).
- **int evalServers**
  number of evaluation servers to be used in the parallel configuration (from the `evaluation_servers` specification in `InterApplic`).

- **DakotaString evalScheduling**
  the scheduling approach to be used for concurrent evaluations within an iterator (from the `evaluation_self_scheduling` and `evaluation_static_scheduling` specifications in `InterApplic`).

- **int analysisServers**
  number of analysis servers to be used in the parallel configuration (from the `analysis_servers` specification in `InterApplic`).

- **DakotaString analysisScheduling**
  the scheduling approach to be used for concurrent analyses within a function evaluation (from the `analysis_self_scheduling` and `analysis_static_scheduling` specifications in `InterApplic`).

- **DakotaString failAction**
  the selected action upon capture of a simulation failure: abort, retry, recover, or continuation (from the `failure_capture` specification in `InterApplic`).

- **int retryLimit**
  the limit on retries for captured simulation failures (from the `retry` specification in `InterApplic`).

- **DakotaRealVector recoveryFnVals**
  the function values to be returned in a recovery operation for captured simulation failures (from the `recover` specification in `InterApplic`).

- **short activeSetVectorFlag**
  active set vector: 1=variable (ASV control on), 0=constant (ASV control off) (from the `active_set_vector` specification in `InterApplic`).

- **DakotaString approxType**
  the selected approximation type: global, multipoint, local, or hierarchical.

- **DakotaString actualInterfacePtr**
  pointer to the interface specification for constructing the truth model used in building local and multipoint approximations (from the `actual_interface_pointer` specification in `InterApprox`).

- **DakotaString actualInterfaceResponsesPtr**
  pointer to the responses specification for constructing the truth model used in building local approximations (from the `actual_interface_responses_pointer` specification in `InterApprox`). This allows differences in gradient specifications between the responses used to build the approximation and the responses computed from the approximation.

- **DakotaString lowFidelityInterfacePtr**
  pointer to the low fidelity interface specification used in hierarchical approximations (from the `low_fidelity_interface_pointer` specification in `InterApprox`).

- **DakotaString highFidelityInterfacePtr**
  pointer to the high fidelity interface specification used in hierarchical approximations (from the `high_fidelity_interface_pointer` specification in `InterApprox`).
- **DakotaString approxDaceMethodPtr**  
  pointer to the design of experiments method used in building global approximations (from the dace\_method\_pointer specification in InterfApprox).

- **DakotaString approxSampleReuse**  
  sample reuse selection for building global approximations: none, all, or region (from the reuse\_samples specification in InterfApprox).

- **DakotaString approxCorrection**  
  correction approach selection for global and hierarchical approximations: offset, scaled, or beta (from the correction specification in InterfApprox).

- **short approxGradUsageFlag**  
  flags the use of gradients in building global approximations (from the use\_gradients specification in InterfApprox).

- **DakotaRealVector krigingCorrelations**  
  vector of correlations used in building a kriging approximation (from the correlations specification in InterfApprox).

### Private Methods

- **void assign (const DataInterface &data\_interface)**  
  convenience function for setting this objects attributes equal to the attributes of the incoming data\_interface object (used by copy constructor and assignment operator).

### 6.37.1 Detailed Description

Container class for interface specification data.

The DataInterface class is used to contain the data from a interface keyword specification. It is populated by ProblemDescDB::interface\_kwhandler() and is queried by the ProblemDescDB::get\_<datatype>() functions. A list of DataInterface objects is maintained in ProblemDescDB::interfaceList, one for each interface specification in an input file. Default values are managed in the DataInterface constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::interfaceList is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

The documentation for this class was generated from the following files:

- DataInterface.H
- DataInterface.C
6.38 DataMethod Class Reference

Container class for method specification data.
#include <DataMethod.H>

Public Methods

- **DataMethod ()**
  constructor.

- **DataMethod (const DataMethod &)**
  copy constructor.

- **~DataMethod ()**
  destructor.

- **DataMethod & operator= (const DataMethod &)**
  assignment operator.

- **int operator== (const DataMethod &)**
  equality operator.

- **void write (ostream &s)**
  write a DataMethod object to an ostream.

- **void read (UnPackBuffer &s)**
  read a DataMethod object from a packed MPI buffer.

- **void write (PackBuffer &s)**
  write a DataMethod object to a packed MPI buffer.

Public Attributes

- **DakotaString methodName**
  the method selection: one of the dot, npsol, opt++, apps, sgopt, nond, dace, or parameter study methods.

- **DakotaString idMethod**
  string identifier for the method specification data set (from the id_method specification in MethodIndControl).

- **DakotaString variablesPointer**
  string pointer to the variables specification to be used by this method (from the variables_pointer specification in MethodIndControl).

- **DakotaString interfacePointer**
• **DakotaString responsesPointer**
  string pointer to the responses specification to be used by this method (from the responses_pointer specification in MethodIndControl).

• **DakotaString modelType**
  model type selection: single, nested, or layered (from the model_type specification in MethodIndControl).

• **DakotaString subMethodPointer**
  string pointer to the sub-iterator used by nested models (from the sub_method_pointer specification in MethodIndControl).

• **DakotaString optionalInterfaceResponsesPointer**
  string pointer to the responses specification used by the optional interface in nested models (from the interface_responses_pointer specification in MethodIndControl).

• **DakotaRealVector primaryCoeffs**
  the primary mapping matrix used in nested models for weighting contributions from the sub-iterator responses in the top level (objective) functions (from the primary_mapping_matrix specification in MethodIndControl).

• **DakotaRealVector secondaryCoeffs**
  the secondary mapping matrix used in nested models for weighting contributions from the sub-iterator responses in the top level (constraint) functions (from the secondary_mapping_matrix specification in MethodIndControl).

• **DakotaString methodOutput**
  method verbosity control: quiet, verbose, debug, or normal (default) (from the output specification in MethodIndControl).

• **Real convergenceTolerance**
  iteration convergence tolerance for the method (from the convergence_tolerance specification in MethodIndControl).

• **Real constraintTolerance**
  tolerance for controlling the amount of infeasibility that is allowed before an active constraint is considered to be violated (from the constraint_tolerance specification in MethodIndControl).

• **int maxIterations**
  maximum number of iterations allowed for the method (from the max_iterations specification in MethodIndControl).

• **int maxFunctionEvaluations**
  maximum number of function evaluations allowed for the method (from the max_function_evaluations specification in MethodIndControl).

• **short speculativeFlag**
  flag for use of speculative gradient approaches for maintaining parallel load balance during the line search portion of optimization algorithms (from the speculative specification in MethodIndControl).
- **DakotaRealVector** `linearIneqConstraintCoefs`  
  Coefficient matrix for the linear inequality constraints (from the `linear_inequality_constraint_matrix` specification in `MethodIndControl`).

- **DakotaRealVector** `linearIneqLowerBnds`  
  Lower bounds for the linear inequality constraints (from the `linear_inequality_lower_bounds` specification in `MethodIndControl`).

- **DakotaRealVector** `linearIneqUpperBnds`  
  Upper bounds for the linear inequality constraints (from the `linear_inequality_upper_bounds` specification in `MethodIndControl`).

- **DakotaRealVector** `linearEqConstraintCoefs`  
  Coefficient matrix for the linear equality constraints (from the `linear_equality_constraint_matrix` specification in `MethodIndControl`).

- **DakotaRealVector** `linearEqTargets`  
  Targets for the linear equality constraints (from the `linear_equality_targets` specification in `MethodIndControl`).

- **DakotaString** `daceMethod`  
  The dace method selection: grid, random, oas, lhs, oa_lhs, box_behnken_design, or central_composite_design (from the dace specification in `MethodDACE`).

- **DakotaString** `minMaxType`  
  The optimization type specification in `MethodDOTDC`.

- **int** `verifyLevel`  
  The verify_level specification in `MethodNPSOLDC`.

- **Real** `functionPrecision`  
  The function_precision specification in `MethodNPSOLDC`.

- **Real** `lineSearchTolerance`  
  The linesearch_tolerance specification in `MethodNPSOLDC`.

- **DakotaString** `searchMethod`  
  The search_method specification for Newton and NIPS methods in `MethodOPTPPDC`.

- **Real** `gradientTolerance`  
  The gradient_tolerance specification in `MethodOPTPPDC`.

- **Real** `maxStep`  
  The max_step specification in `MethodOPTPPDC`.

- **DakotaString** `meritFn`  
  The merit_function specification for NIPS methods in `MethodOPTPPDC`.

- **DakotaString** `centralPath`  
  The central_path specification for NIPS methods in `MethodOPTPPDC`. 

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
- Real *stepLenToBoundary*
  
  *the* step length to boundary *specification for NIPS methods in MethodOPTPPDC.*

- Real *centeringParam*
  
  *the* centering parameter *specification for NIPS methods in MethodOPTPPDC.*

- Real *initialRadius*
  
  *the* initial radius *specification for ellipsoid methods in MethodOPTPPDC.*

- int *searchSchemeSize*
  
  *the* search scheme size *specification for PDS methods in MethodOPTPPDC.*

- Real *solnAccuracy*
  
  *the* solution accuracy *specification in MethodSGOPTDC.*

- Real *maxCPUTime*
  
  *the* max cpu time *specification in MethodSGOPTDC.*

- Real *crossoverRate*
  
  *the* crossover rate *specification for GA/EPSA methods in MethodSGOPTEA.*

- Real *mutationDimRate*
  
  *the* dimension rate *specification for mutation in GA/EPSA methods in MethodSGOPTEA.*

- Real *mutationPopRate*
  
  *the* population rate *specification for mutation in GA/EPSA methods in MethodSGOPTEA.*

- Real *mutationScale*
  
  *the* mutation scale *specification for GA/EPSA methods in MethodSGOPTEA.*

- Real *mutationMinScale*
  
  *the* min scale *specification for mutation in EPSA methods in MethodSGOPTEA.*

- Real *initDelta*
  
  *the* initial delta *specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.*

- Real *threshDelta*
  
  *the* threshold delta *specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.*

- Real *contractFactor*
  
  *the* contraction factor *specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.*

- int *populationSize*
  
  *the* population size *specification for GA/EPSA methods in MethodSGOPTEA.*

- int *newSolnsGenerated*
the new_solutions_generated specification for GA/EPSA methods in MethodSGOPTEA.

- **int numberRetained**
  the integer assignment to random, chc, or elitist in the replacement_type specification for GA/EPSA methods in MethodSGOPTEA.

- **int expandAfterSuccess**
  the expand_after_success specification for PS/SW methods in MethodSGOPTPS and MethodSGOPTSW.

- **int contractAfterFail**
  the contract_after_failure specification for the SW method in MethodSGOPTSW.

- **int mutationRange**
  the mutation_range specification for the pga_int method in MethodSGOPTEA.

- **int numPartitions**
  the num_partitions specification for EPSA methods in MethodSGOPTEA.

- **int totalPatternSize**
  the total_pattern_size specification for APPS/PS methods in MethodAPPSDC and MethodSGOPTPS.

- **int batchSize**
  the batch_size specification for the sMC method in MethodSGOPTSMC.

- **short nonAdaptiveFlag**
  the non_adaptive specification for the pga_real method in MethodSGOPTEA.

- **short randomizeOrderFlag**
  the stochastic specification for the PS method in MethodSGOPTPS.

- **short expansionFlag**
  the no_expansion specification for APPS/PS/SW methods in MethodAPPSDC, MethodSGOPTPS, and MethodSGOPTSW.

- **DakotaString selectionPressure**
  the selection_pressure specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString replacementType**
  the replacement_type specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString crossoverType**
  the crossover_type specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString mutationType**
  the mutation_type specification for GA/EPSA methods in MethodSGOPTEA.

- **DakotaString exploratoryMoves**
  the exploratory_moves specification for the PS method in MethodSGOPTPS.
- **DakotaString patternBasis**
  
  the pattern basis specification for APPS/PS methods in MethodAPPSDC and MethodSGOPTPS.

- **DakotaIntArray varPartitions**
  
  the partitions specification for sMC/PStudy methods in MethodSGOPTSMC and MethodPSMPS.

- **int randomSeed**
  
  the seed specification for S戈PT, NonD, & DACE methods.

- **int numSamples**
  
  the samples specification for NonD & DACE methods.

- **int numSymbols**
  
  the symbols specification for DACE methods.

- **int expansionTerms**
  
  the expansion terms specification in MethodNonDPCE.

- **int expansionOrder**
  
  the expansion order specification in MethodNonDPCE.

- **DakotaString sampleType**
  
  the sample type specification in MethodNonDMC and MethodNonDPCE.

- **DakotaString reliabilityMethod**
  
  the amv/c iterated amv/form/c sorm selection in MethodNonDAMV.

- **DakotaRealArray responseThresholds**
  
  the response_thresholds specification in MethodNonDMC and MethodNonDPCE.

- **DakotaRealArray responseLevels**
  
  the response_levels specification in MethodNonDAMV.

- **DakotaRealArray probabilityLevels**
  
  the probability_levels specification in MethodNonDAMV.

- **short allVarsFlag**
  
  the all_variables specification in MethodNonDMC.

- **int paramStudyType**
  
  the type of parameter study: list(-1), vector(1, 2, or 3), centered(4), or multidim(5).

- **DakotaRealVector finalPoint**
  
  the final_point specification in MethodPSVPS.

- **DakotaRealVector stepVector**
  
  the step_vector specification in MethodPSVPS.

- **Real stepLength**
  
  the step_length specification in MethodPSVPS.
• int numSteps
  
  the num_steps specification in MethodPSVP.

• DakotaRealVector listOfPoints
  
  the list_of_points specification in MethodPSLPS.

• Real percentDelta
  
  the percent_delta specification in MethodPSCP.

• int deltasPerVariable
  
  the deltas_per_variable specification in MethodPSCP.

### Private Methods

• void assign (const DataMethod &data_method)
  
  convenience function for setting this objects attributes equal to the attributes of the incoming data_method object (used by copy constructor and assignment operator).

### 6.38.1 Detailed Description

Container class for method specification data.

The DataMethod class is used to contain the data from a method keyword specification. It is populated by ProblemDescDB::method_kwhandler() and is queried by the ProblemDescDB::get_<datatype>() functions. A list of DataMethod objects is maintained in ProblemDescDB::methodList, one for each method specification in an input file. Default values are managed in the DataMethod constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::methodList is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

The documentation for this class was generated from the following files:

• DataMethod.H
• DataMethod.C
6.39 DataResponses Class Reference

Container class for responses specification data.

```cpp
#include <DataResponses.H>
```

**Public Methods**

- **DataResponses ()**
  
  constructor.

- **DataResponses (const DataResponses &)**
  
  copy constructor.

- **~DataResponses ()**
  
  destructor.

- **DataResponses & operator=(const DataResponses &)**
  
  assignment operator.

- **int operator==(const DataResponses &)**
  
  equality operator.

- **void write (ostream &s)**
  
  write a DataResponses object to an ostream.

- **void read (UnPackBuffer &s)**
  
  read a DataResponses object from a packed MPI buffer.

- **void write (PackBuffer &s)**
  
  write a DataResponses object to a packed MPI buffer.

**Public Attributes**

- **size_t numObjectiveFunctions**
  
  number of objective functions (from the num_objective_functions specification in RespFnOpt).

- **size_t numNonlinearIneqConstraints**
  
  number of nonlinear inequality constraints (from the num_nonlinear_inequality_constraints specification in RespFnOpt).

- **size_t numNonlinearEqConstraints**
  
  number of nonlinear equality constraints (from the num_nonlinear_equality_constraints specification in RespFnOpt).

- **size_t numLeastSquaresTerms**
number of least squares terms (from the num_least_squares_terms specification in RespFnLS).

- size_t numResponseFunctions
  number of generic response functions (from the num_response_functions specification in RespFngen).

- DakotaRealVector multiObjectiveWeights
  vector of multiobjective weightings (from the multi_objective_weights specification in RespFnopt).

- DakotaRealVector nonlinearIneqLowerBnds
  vector of nonlinear inequality constraint lower bounds (from the nonlinear_inequality_lower_bounds specification in RespFnopt).

- DakotaRealVector nonlinearIneqUpperBnds
  vector of nonlinear inequality constraint upper bounds (from the nonlinear_inequality_upper_bounds specification in RespFnopt).

- DakotaRealVector nonlinearEqTargets
  vector of nonlinear equality constraint targets (from the nonlinear_equality_targets specification in RespFnopt).

- DakotaString gradientType
  gradient type: none, numerical, analytic, or mixed (from the no_gradients, numerical_gradients, analytic_gradients, and mixed_gradients specifications in RespGrad).

- DakotaString hessianType
  Hessian type: none or analytic (from the no_hessians and analytic_hessians specifications in RespHess).

- DakotaString methodSource
  numerical gradient method source: dakota or vendor (from the method_source specification in RespGradNum and RespGradMixed).

- DakotaString intervalType
  numerical gradient interval type: forward or central (from the interval_type specification in RespGradNum and RespGradMixed).

- Real fdStepSize
  numerical gradient finite difference step size (from the fd_step_size specification in RespGradNum and RespGradMixed).

- DakotaIntList idNumerical
  mixed gradient numerical identifiers (from the id_numerical specification in RespGradMixed).

- DakotaIntList idAnalytic
  mixed gradient analytic identifiers (from the id_analytic specification in RespGradMixed).

- DakotaString idResponses
  string identifier for the responses specification data set (from the id_responses specification in RespSetId).
Private Methods

- void assign (const DataResponses &data\_responses)
  
  convenience function for setting this objects attributes equal to the attributes of the incoming data\_responses object (used by copy constructor and assignment operator).

6.39.1 Detailed Description

Container class for responses specification data.

The DataResponses class is used to contain the data from a responses keyword specification. It is populated by ProblemDescDB::responses\_kwhandler() and is queried by the ProblemDescDB::get\_<datatype>\_() functions. A list of DataResponses objects is maintained in ProblemDescDB::responsesList, one for each responses specification in an input file. Default values are managed in the DataResponses constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within ProblemDescDB since ProblemDescDB::responsesList is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

The documentation for this class was generated from the following files:

- DataResponses.H
- DataResponses.C
6.40 DataVariables Class Reference

Container class for variables specification data.

```cpp
#include <DataVariables.H>
```

### Public Methods

- DataVariables ()
  
  *constructor.*

- DataVariables (const DataVariables &)

  *copy constructor.*

- ~DataVariables ()

  *destructor.*

- DataVariables & operator= (const DataVariables &)

  *assignment operator.*

- int operator==(const DataVariables &)

  *equality operator.*

- void write (ostream &s)

  *write a DataVariables object to an ostream.*

- void read (UnPackBuffer &s)

  *read a DataVariables object from a packed MPI buffer.*

- void write (PackBuffer &s)

  *write a DataVariables object to a packed MPI buffer.*

- int design ()

  *return total number of design variables.*

- int uncertain ()

  *return total number of uncertain variables.*

- int state ()

  *return total number of state variables.*

- int num_continuous_variables ()

  *return total number of continuous variables.*

- int num_discrete_variables ()

  *return total number of discrete variables.*
• `size_t num_variables ()`
  
  return total number of variables.

Public Attributes

• `DakotaString idVariables`
  
  string identifier for the variables specification data set (from the `id_variables` specification in `VarSet-Id`).

• `size_t numContinuousDesVars`
  
  number of continuous design variables (from the `continuous_design` specification in `VarDV`).

• `size_t numDiscreteDesVars`
  
  number of discrete design variables (from the `discrete_design` specification in `VarDV`).

• `size_t numNormalUncVars`
  
  number of normal uncertain variables (from the `normal_uncertain` specification in `VarUV`).

• `size_t numLognormalUncVars`
  
  number of lognormal uncertain variables (from the `lognormal_uncertain` specification in `VarUV`).

• `size_t numUniformUncVars`
  
  number of uniform uncertain variables (from the `uniform_uncertain` specification in `VarUV`).

• `size_t numLoguniformUncVars`
  
  number of loguniform uncertain variables (from the `loguniform_uncertain` specification in `VarUV`).

• `size_t numWeibullUncVars`
  
  number of weibull uncertain variables (from the `weibull_uncertain` specification in `VarUV`).

• `size_t numHistogramUncVars`
  
  number of histogram uncertain variables (from the `histogram_uncertain` specification in `VarUV`).

• `size_t numContinuousStateVars`
  
  number of continuous state variables (from the `continuous_state` specification in `VarSV`).

• `size_t numDiscreteStateVars`
  
  number of discrete state variables (from the `discrete_state` specification in `VarSV`).

• `DakotaRealVector continuousDesignVars`
  
  initial values for the continuous design variables array (from the `cdv_initial_point` specification in `VarDV`).

• `DakotaRealVector continuousDesignLowerBnds`
  
  the continuous design lower bounds array (from the `cdv_lower_bounds` specification in `VarDV`).

• `DakotaRealVector continuousDesignUpperBnds`
  
  the continuous design upper bounds array (from the `cdv_upper_bounds` specification in `VarDV`).
• DakotaIntVector discreteDesignVars
  initial values for the discrete design variables array (from the ddv_initial_point specification in VarDV).

• DakotaIntVector discreteDesignLowerBnds
  the discrete design lower bounds array (from the ddv_lower_bounds specification in VarDV).

• DakotaIntVector discreteDesignUpperBnds
  the discrete design upper bounds array (from the ddv_upper_bounds specification in VarDV).

• DakotaStringArray continuousDesignLabels
  the continuous design labels array (from the cdv_descriptor specification in VarDV).

• DakotaStringArray discreteDesignLabels
  the discrete design labels array (from the ddv_descriptor specification in VarDV).

• DakotaRealVector normalUncMeans
  means of the normal uncertain variables (from the nuv_means specification in VarUV).

• DakotaRealVector normalUncStdDevs
  standard deviations of the normal uncertain variables (from the nuv_std_deviations specification in VarUV).

• DakotaRealVector normalUncDistLowerBnds
  distribution lower bounds for the normal uncertain variables (from the nuv_dist_lower_bounds specification in VarUV).

• DakotaRealVector normalUncDistUpperBnds
  distribution upper bounds for the normal uncertain variables (from the nuv_dist_upper_bounds specification in VarUV).

• DakotaRealVector lognormalUncMeans
  means of the lognormal uncertain variables (from the lnuv_means specification in VarUV).

• DakotaRealVector lognormalUncStdDevs
  standard deviations of the lognormal uncertain variables (from the lnuv_std_deviations specification in VarUV).

• DakotaRealVector lognormalUncErrFacts
  error factors for the lognormal uncertain variables (from the lnuv_error_factors specification in VarUV).

• DakotaRealVector lognormalUncDistLowerBnds
  distribution lower bounds for the lognormal uncertain variables (from the lnuv_dist_lower_bounds specification in VarUV).

• DakotaRealVector lognormalUncDistUpperBnds
  distribution upper bounds for the lognormal uncertain variables (from the lnuv_dist_upper_bounds specification in VarUV).

• DakotaRealVector uniformUncDistLowerBnds
DataVariable class reference

- **DakotaRealVector `uniformUncDistUpperBnds`**
  distribution upper bounds for the uniform uncertain variables (from the `uuv_dist_upper_bounds` specification in `VarUV`).

- **DakotaRealVector `loguniformUncDistLowerBnds`**
  distribution lower bounds for the loguniform uncertain variables (from the `luuv_dist_lower_bounds` specification in `VarUV`).

- **DakotaRealVector `loguniformUncDistUpperBnds`**
  distribution upper bounds for the loguniform uncertain variables (from the `luuv_dist_upper_bounds` specification in `VarUV`).

- **DakotaRealVector `weibullUncAlphas`**
  alpha factors for the weibull uncertain variables (from the `wuv_alphas` specification in `VarUV`).

- **DakotaRealVector `weibullUncBetas`**
  beta factors for the weibull uncertain variables (from the `wuv_betas` specification in `VarUV`).

- **DakotaRealVector `weibullUncDistLowerBnds`**
  distribution lower bounds for the weibull uncertain variables (from the `wuv_dist_lower_bounds` specification in `VarUV`).

- **DakotaRealVector `weibullUncDistUpperBnds`**
  distribution upper bounds for the weibull uncertain variables (from the `wuv_dist_upper_bounds` specification in `VarUV`).

- **DakotaRealVector `histogramUncDistLowerBnds`**
  distribution lower bounds for the histogram uncertain variables (from the `huv_dist_lower_bounds` specification in `VarUV`).

- **DakotaRealVector `histogramUncDistUpperBnds`**
  distribution upper bounds for the histogram uncertain variables (from the `huv_dist_upper_bounds` specification in `VarUV`).

- **DakotaStringList `histogramUncFilenames`**
  filenames containing the histograms for the histogram uncertain variables (from the `huv_filenames` specification in `VarUV`).

- **DakotaRealMatrix `uncertainCorrelations`**
  correlation matrix for all uncertain variables (from the `uncertain_correlation_matrix` specification in `VarUV`). This matrix specifies rank correlations for sampling methods (i.e., LHS) and correlation coefficients \( \rho_{ij} = \) normalized covariance matrix) for analytic reliability methods.

- **DakotaRealVector `uncertainVars`**
  array of values for all uncertain variables (built and initialized in `ProblemDescDB::variables_kwhandler()`).

- **DakotaRealVector `uncertainDistLowerBnds`**
distribution lower bounds for all uncertain variables (collected from nuv_dist_lower_bounds, lnuv_dist_lower_bounds, uuv_dist_lower_bounds, luuv_dist_lower_bounds, wuv_dist_lower_bounds, and huv_dist_lower_bounds specifications in VarUV).

- **DakotaRealVector** uncertainDistUpperBnds
  
  distribution upper bounds for all uncertain variables (collected from nuv_dist_upper_bounds, lnuv_dist_upper_bounds, uuv_dist_upper_bounds, luuv_dist_upper_bounds, wuv_dist_upper_bounds, and huv_dist_upper_bounds specifications in VarUV).

- **DakotaStringArray** uncertainLabels
  
  labels for all uncertain variables (collected from nuv_descriptor, lnuv_descriptor, uuv_descriptor, luuv_descriptor, wuv_descriptor, and huv_descriptor specifications in VarUV).

- **DakotaRealVector** continuousStateVars
  
  initial values for the continuous state variables array (from the csv_initial_state specification in VarSV).

- **DakotaRealVector** continuousStateLowerBnds
  
  the continuous state lower bounds array (from the csv_lower_bounds specification in VarSV).

- **DakotaRealVector** continuousStateUpperBnds
  
  the continuous state upper bounds array (from the csv_upper_bounds specification in VarSV).

- **DakotaIntVector** discreteStateVars
  
  initial values for the discrete state variables array (from the dsv_initial_state specification in Var-SV).

- **DakotaIntVector** discreteStateLowerBnds
  
  the discrete state lower bounds array (from the dsv_lower_bounds specification in VarSV).

- **DakotaIntVector** discreteStateUpperBnds
  
  the discrete state upper bounds array (from the dsv_upper_bounds specification in VarSV).

- **DakotaStringArray** continuousStateLabels
  
  the continuous state labels array (from the csv_descriptor specification in VarSV).

- **DakotaStringArray** discreteStateLabels
  
  the discrete state labels array (from the dsv_descriptor specification in VarSV).

### Private Methods

- void **assign** (const DataVariables &data_variables)
  
  convenience function for setting this objects attributes equal to the attributes of the incoming data_variables object (used by copy constructor and assignment operator).
6.40 DataVariables Class Reference

6.40.1 Detailed Description

Container class for variables specification data.

The DataVariables class is used to contain the data from a variables keyword specification. It is populated by `ProblemDescDB::variables_kwhandler()` and is queried by the `ProblemDescDB::get_<datatype>()` functions. A list of DataVariables objects is maintained in `ProblemDescDB::variablesList`, one for each variables specification in an input file. Default values are managed in the DataVariables constructor. Data is public to avoid maintaining set/get functions, but is still encapsulated within `ProblemDescDB` since `ProblemDescDB::variablesList` is private (a similar model is used with SurrogateDataPoint objects contained in DakotaApproximation and with ParallelismLevel objects contained in ParallelLibrary).

The documentation for this class was generated from the following files:

- DataVariables.H
- DataVariables.C
6.41 DirectFnApplicInterface Class Reference

Derived application interface class which spawns simulation codes and testers using direct procedure calls.

```cpp
#include <DirectFnApplicInterface.H>
```

Inheritance diagram for DirectFnApplicInterface:

```
    DakotaInterface
     |                  
    ApplicationInterface
     |                  
DirectFnApplicInterface
```

Public Methods

- **DirectFnApplicInterface** (const ProblemDescDB &problem_db, const size_t &num_fns)
  
  * constructor.

- **~DirectFnApplicInterface** ()
  
  * destructor.

- **void derived_map** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int fn_eval_id)
  
  Called by `map()` and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- **void derived_map_asynch** (const ParamResponsePair &pair)
  
  Called by `map()` and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- **void derived_synch** (DakotaList< ParamResponsePair > &prp_list)
  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- **void derived_synch_nowait** (DakotaList< ParamResponsePair > &prp_list)
  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- **int derived_synchronous_local_analysis** (const int &analysis_id)
  
  Execute a particular analysis (identified by `analysis_id`) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve_analyses_synch().
Private Methods

- `int derived_map_if (const DakotaString &if_name)`
  
  executes the input filter portion of a direct evaluation invocation.

- `int derived_map_ac (const DakotaString &ac_name)`
  
  executes an analysis code portion of a direct evaluation invocation.

- `int derived_map_of (const DakotaString &of_name)`
  
  executes the output filter portion of a direct evaluation invocation.

- `void set_local_data ()`
  
  convenience function for local test simulators which sets variable attributes and zeros response data.

- `void overlay_response (DakotaResponse &response)`
  
  convenience function for local test simulators which overlays response contributions from multiple analyses using MPI\textit{Reduce}.

- `int cylhead (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  the cylinder head constrained optimization test function.

- `int rosenbrock (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  the rosenbrock optimization and least squares test function.

- `int textbook (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  the textbook constrained optimization test function.

- `int textbook1 (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  portion of \textit{textbook()} evaluating the objective function and its derivatives.

- `int textbook2 (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  portion of \textit{textbook()} evaluating constraint 1 and its derivatives.

- `int textbook3 (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  portion of \textit{textbook()} evaluating constraint 2 and its derivatives.

- `int salinas (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response)`
  
  direct interface to the SALINAS structural dynamics simulation code.
Private Attributes

- **DakotaString iFilterName**
  
  name of the direct function input filter.

- **DakotaString oFilterName**
  
  name of the direct function output filter.

- **short gradFlag**
  
  signals use of fnGrads in direct simulator functions.

- **short hessFlag**
  
  signals use of fnHessians in direct simulator functions.

- **size_t numFns**
  
  number of functions in fnVals.

- **size_t numVars**
  
  total number of continuous and discrete variables.

- **size_t numGradVars**
  
  number of continuous variables.

- **DakotaRealVector xVect**
  
  continuous and discrete variable set used within direct simulator functions.

- **DakotaRealVector fnVals**
  
  response function values set within direct simulator functions.

- **DakotaRealMatrix fnGrads**
  
  response function gradients set within direct simulator functions.

- **DakotaRealMatrixArray fnHessians**
  
  response function Hessians set within direct simulator functions.

- **DakotaVariables directFnVars**
  
  class scope variables object.

- **DakotaIntArray directFnASV**
  
  class scope active set vector object.

- **DakotaResponse directFnResponse**
  
  class scope response object.
6.41.1 Detailed Description

Derived application interface class which spawns simulation codes and testers using direct procedure calls. DerivedFnApplicInterface uses a few linkable simulation codes and several internal member functions to perform parameter to response mappings.

The documentation for this class was generated from the following files:

- DirectFnApplicInterface.H
- DirectFnApplicInterface.C
6.42 DOTOptimizer Class Reference

Wrapper class for the DOT optimization library.

```cpp
#include <DOTOptimizer.H>
```

Inheritance diagram for DOTOptimizer:

```
DakotaIterator
    |           |
    |           |
    DakotaOptimizer
    |           |
    |           |
    DOTOptimizer
```

Public Methods

- **DOTOptimizer (DakotaModel &model)**
  
  *constructor.*

- **~DOTOptimizer ()**
  
  *destructor.*

- **void find_optimum ()**
  
  *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

Private Methods

- **void allocate_workspace ()**
  
  *Allocates workspace for the optimizer.*

Private Attributes

- **int dotInfo**
  
  *INFO from DOT manual.*

- **int dotFDInfo**
  
  *internal DOT parameter NGOTOZ.*

- **int dotMethod**
  
  *METHOD from DOT manual.*
6.42 DOTOptimizer Class Reference

- int printControl
  
  *IPRINT from DOT manual (controls output verbosity).*

- int optimizationType
  
  *MINMAX from DOT manual (minimize or maximize).*

- DakotaRealArray realCntlParmArray
  
  *RPRM from DOT manual.*

- DakotaIntArray intCntlParmArray
  
  *IPRM from DOT manual.*

- DakotaRealVector localConstraintValues
  
  *array of nonlinear constraint values passed to DOT.*

- DakotaSizeList constraintMappingIndices
  
  *a list of indices for referencing the corresponding DakotaResponse constraints used in computing the DOT constraints.*

- DakotaRealList constraintMappingMultipliers
  
  *a list of multipliers for mapping the DakotaResponse constraints to the DOT constraints.*

- DakotaRealList constraintMappingOffsets
  
  *a list of offsets for mapping the DakotaResponse constraints to the DOT constraints.*

### 6.42.1 Detailed Description

Wrapper class for the DOT optimization library.

The DOTOptimizer class provides a wrapper for DOT, a commercial Fortran 77 optimization library from Vanderplaats Research and Development. It uses a reverse communication mode, which avoids the static function and static attribute issues that arise with function pointer designs (see NPSOLOptimizer and SNLLOptimizer).

The user input mappings are as follows: `max_iterations` is mapped into DOT’s ITMAX parameter within its IPRM array, `max_function_evaluations` is implemented directly in the `find_optimum()` loop since there is no DOT parameter equivalent, `convergence_tolerance` is mapped into DOT’s DELOBJ parameter (the relative convergence tolerance) within its RPRM array, output verbosity is mapped into DOT’s IPRINT parameter within its function call parameter list (verbose: IPRINT = 7; quiet: IPRINT = 3), and `optimization_type` is mapped into DOT’s MINMAX parameter within its function call parameter list. Refer to [Vanderplaats Research and Development, 1995] for information on IPRM, RPRM, and the DOT function call parameter list.

### 6.42.2 Member Data Documentation

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch (©) 1997-2001
6.42.2.1 int DOTOptimizer::dotInfo  [private]
INFO from DOT manual.
Information requested by DOT: 0=optimization complete, 1=get values, 2=get gradients

6.42.2.2 int DOTOptimizer::dotFDsInfo  [private]
internal DOT parameter NGOTOZ.
the DOT parameter list has been modified to pass NGOTOZ, which signals whether DOT is finite-
differencing (nonzero value) or performing the line search (zero value).

6.42.2.3 int DOTOptimizer::dotMethod  [private]
METHOD from DOT manual.
For nonlinear constraints: 0/1 = dot_mmfd, 2 = dot_slp, 3 = dot_sqp. For unconstrained: 0/1 = dot_bfgs, 2
= dot_frcg.

6.42.2.4 int DOTOptimizer::printControl  [private]
IPRINT from DOT manual (controls output verbosity).
Values range from 0 (least output) to 7 (most output).

6.42.2.5 int DOTOptimizer::optimizationType  [private]
MINMAX from DOT manual (minimize or maximize).
Values of 0 or -1 (minimize) or 1 (maximize).

6.42.2.6 DakotaRealArray DOTOptimizer::realCntlParmArray  [private]
RPRM from DOT manual.
Array of real control parameters.

6.42.2.7 DakotaIntArray DOTOptimizer::intCntlParmArray  [private]
IPRM from DOT manual.
Array of integer control parameters.

6.42.2.8 DakotaRealVector DOTOptimizer::localConstraintValues  [private]
array of nonlinear constraint values passed to DOT.
This array must be of nonzero length (sized with localConstraintArraySize) and must contain only one-
sided inequality constraints which are <= 0 (which requires a transformation from 2-sided inequalities and
equalities).
6.42.2.9 DakotaSizeList DOTOptimizer::constraintMappingIndices [private]

A list of indices for referencing the corresponding DakotaResponse constraints used in computing the DOT constraints.

The length of the list corresponds to the number of DOT constraints, and each entry in the list points to the corresponding DAKOTA constraint.

6.42.2.10 DakotaRealList DOTOptimizer::constraintMappingMultipliers [private]

A list of multipliers for mapping the DakotaResponse constraints to the DOT constraints.

The length of the list corresponds to the number of DOT constraints, and each entry in the list contains a multiplier for the DAKOTA constraint identified with constraintMappingIndices. These multipliers are currently +1 or -1.

6.42.2.11 DakotaRealList DOTOptimizer::constraintMappingOffsets [private]

A list of offsets for mapping the DakotaResponse constraints to the DOT constraints.

The length of the list corresponds to the number of DOT constraints, and each entry in the list contains an offset for the DAKOTA constraint identified with constraintMappingIndices. These offsets involve inequality bounds or equality targets, since DOT assumes constraint allowables = 0.

The documentation for this class was generated from the following files:

- DOTOptimizer.H
- DOTOptimizer.C
6.43 ErrorTable Struct Reference

Data structure to hold errors.

Public Attributes

- CtelRegexp::RStatus rc
  
  Enumerated type to hold status codes.

- const char * msg
  
  Holds character string error message.

6.43.1 Detailed Description

Data structure to hold errors.

This module implements a C++ wrapper for Regular Expressions based on the public domain engine for regular expressions released by: Copyright (c) 1986 by University of Toronto. Written by Henry Spencer. Not derived from licensed software.

The documentation for this struct was generated from the following file:

- CtelRegExp.C
6.44 ForkAnalysisCode Class Reference

Derived class in the AnalysisCode class hierarchy which spawns simulations using forks.

```cpp
#include <ForkAnalysisCode.H>
```

Inheritance diagram for ForkAnalysisCode:

```
+----------------------------------+
| AnalysisCode                     |
+----------------------------------+
| ForkAnalysisCode                 |
```

### Public Methods

- **ForkAnalysisCode** (const ProblemDescDB &problem, db)
  
  *constructor.*

- **~ForkAnalysisCode** ()
  
  *destructor.*

- **pid_t fork_program** (const short block_flag)
  
  *spawn a child process using fork()/vfork()/execvp() and wait for completion using waitpid() if block_flag is true.*

- void **check_status** (const int status)
  
  *check the exit status of a forked process and abort if an error code was returned.*

- void **argument_list** (const int index, const DakotaString &arg)
  
  *set argList[index] to arg.*

- void **tag_argument_list** (const int index, const int tag)
  
  *append an additional tag to argList[index] (beyond that already present in the modified file names) for managing concurrent analyses within a function evaluation.*

### Private Attributes

- const char * **argList** [4]
  
  *an array of strings for use with execvp(const char * , char * const *) (an argList entry can be passed as the first argument, and the entire argList can be cast as the second argument).*
6.44.1 Detailed Description

Derived class in the AnalysisCode class hierarchy which spawns simulations using forks.

ForkAnalysisCode creates a copy of the parent DAKOTA process using fork() / vfork() and then replaces the copy with a simulation process using execvp(). The parent process can then use waitpid() to wait on completion of the simulation process.

6.44.2 Member Function Documentation

6.44.2.1 void ForkAnalysisCode::check_status (const int status)

check the exit status of a forked process and abort if an error code was returned.

Check to see if the 3-piece interface terminated abnormally (WIFEXITED(status)==0) or if either execvp or the application returned a status code of -1 (WIFEXITED(status)!=0 && (signed char)WEXITSTATUS(status)==-1). If one of these conditions is detected, output a failure message and abort. Note: the application code should not return a status code of -1 unless an immediate abort of dakota is wanted. If for instance, failure capturing is to be used, the application code should write the word ”FAIL” to the appropriate results file and return a status code of 0 through exit().

The documentation for this class was generated from the following files:

- ForkAnalysisCode.H
- ForkAnalysisCode.C
6.45  ForkApplicInterface Class Reference

Derived application interface class which spawns simulation codes using forks.

```cpp
#include <ForkApplicInterface.H>
```

Inheritance diagram for ForkApplicInterface:

```
DakotaInterface
   |
   |
ApplicationInterface
   |
   |
ForkApplicInterface
```

Public Methods

- **ForkApplicInterface** (const ProblemDescDB &problem_db, const size_t &num_fns)
  
  *constructor.*

- ~**ForkApplicInterface** ()
  
  *destructor.*

- void **derived_map** (const DakotaVariables &vars, const DakotaIntArray &asv, DakotaResponse &response, int fn_id)
  
  *Called by map() and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.*

- void **derived_map_asynch** (const ParamResponsePair &pair)
  
  *Called by map() and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.*

- void **derived_sync** (DakotaList< ParamResponsePair > &prp_list)
  
  *For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.*

- void **derived_sync_nowait** (DakotaList< ParamResponsePair > &prp_list)
  
  *For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.*

- int **derived_synchronous_local_analysis** (const int &analysis_id)
  
  *Execute a particular analysis (identified by analysis_id) synchronously on the local processor. Used for the derived class specifics within ApplicationInterface::serve_analyses_sync().*
Private Methods

- void derived\_synch\_kernel (DakotaList< ParamResponsePair > &prpList, const pid\_t pid)
  
  Convenience function for common code between derived\_synch() & derived\_synch\_nowait().

- pid\_t fork\_application (const short block\_flag)
  
  perform the complete function evaluation by managing the input filter, analysis programs, and output filter.

- void asynchronous\_local\_analyses (const int &start, const int &end, const int &step)
  
  execute analyses asynchronously on the local processor.

- void synchronous\_local\_analyses (const int &start, const int &end, const int &step)
  
  execute analyses synchronously on the local processor.

- void serve\_analyses\_async ()
  
  serve the analysis scheduler and execute analysis assignments asynchronously.

Private Attributes

- ForkAnalysisCode forkSimulator
  
  ForkAnalysisCode provides convenience functions for forking individual programs and checking fork exit status.

- DakotaList< pid\_t > processIdList
  
  list of process id’s for asynchronous evaluations; correspondence to evalldList used for mapping captured fork process id’s to function evaluation id’s.

- DakotaIntList evalldList
  
  list of function evaluation id’s for asynchronous evaluations; correspondence to processIdList used for mapping captured fork process id’s to function evaluation id’s.

6.45.1 Detailed Description

Derived application interface class which spawns simulation codes using forks.
ForkApplicInterface uses a ForkAnalysisCode object for performing simulation invocations.

6.45.2 Member Function Documentation

6.45.2.1 pid\_t ForkApplicInterface::fork\_application (const short block\_flag) [private]

perform the complete function evaluation by managing the input filter, analysis programs, and output filter.
Manage the input filter, 1 or more analysis programs, and the output filter in blocking or nonblocking mode as governed by block\_flag. In the case of a single analysis and no filters, a single fork is performed, while in other cases, an initial fork is reforked multiple times. Called from derived\_map().
with block_flag == BLOCK and from derived_map_asynch() with block_flag == FALL_THROUGH. Uses ForkAnalysisCode::fork_program() to spawn individual program components within the function evaluation.

### 6.45.2.2 void ForkApplicInterface::asynchronous_local_analyses (const int & start, const int & end, const int & step) [private]

execute analyses asynchronously on the local processor.

Schedule analyses asynchronously on the local processor using a self-scheduling approach (start to end in step increments). Concurrency is limited by asyncLocalAnalysisConcurrency. Modeled after ApplicationInterface::asynchronous_local_evaluations(). NOTE: This function should be elevated to ApplicationInterface if and when another derived interface class supports asynchronous local analyses.

### 6.45.2.3 void ForkApplicInterface::synchronous_local_analyses (const int & start, const int & end, const int & step) [private]

execute analyses synchronously on the local processor.

Execute analyses synchronously in succession on the local processor (start to end in step increments). Modeled after ApplicationInterface::synchronous_local_evaluations().

### 6.45.2.4 void ForkApplicInterface::serve_analyses_asynch () [private]

serve the analysis scheduler and execute analysis assignments asynchronously.

This code runs multiple async analyses on each server. It is modeled after ApplicationInterface::serve_evaluations_asynch(). NOTE: This fn should be elevated to ApplicationInterface if and when another derived interface class supports hybrid analysis parallelism.

The documentation for this class was generated from the following files:

- ForkApplicInterface.H
- ForkApplicInterface.C
6.46 FunctionCompare Class Template Reference

#include <DakotaList.H>

Public Methods

- FunctionCompare (int(*func)(const T &, void *), void *v)
  Constructor that defines the pointer to function and search value.

- bool operator() (T t) const
  The operator() must be defined. Calls the function testFunction.

Private Attributes

- int(* testFunction )(const T &, void *)
  Pointer to test function.

- void * search_val
  Holds the value to search for.

6.46.1 Detailed Description

template<class T> class FunctionCompare<T>

Internal Functor to mimick the RW find and index functions using the STL find_if() method. The class holds a pointer to the test function and the search value.

The documentation for this class was generated from the following file:

- DakotaList.H
6.47 FundamentalVarConstraints Class Reference

Derived class within the DakoVarConstraints hierarchy which employs the default data view (no variable or domain type array merging).

```
#include <FundamentalVarConstraints.H>
```

Inheritance diagram for FundamentalVarConstraints::

```
DakotaVarConstraints
  FundamentalVarConstraints
```

Public Methods

- `FundamentalVarConstraints (const ProblemDescDB &problem_db)`
  - constructor.

- `~FundamentalVarConstraints ()`
  - destructor.

- `const DakotaRealVector & continuous_lower_bounds () const`
  - return the active continuous variable lower bounds.

- `void continuous_lower_bounds (const DakotaRealVector &c_lower_bounds)`
  - set the active continuous variable lower bounds.

- `const DakotaRealVector & continuous_upper_bounds () const`
  - return the active continuous variable upper bounds.

- `void continuous_upper_bounds (const DakotaRealVector &c_upper_bounds)`
  - set the active continuous variable upper bounds.

- `const DakotaIntVector & discrete_lower_bounds () const`
  - return the active discrete variable lower bounds.

- `void discrete_lower_bounds (const DakotaIntVector &d_lower_bounds)`
  - set the active discrete variable lower bounds.

- `const DakotaIntVector & discrete_upper_bounds () const`
  - return the active discrete variable upper bounds.

- `void discrete_upper_bounds (const DakotaIntVector &d_upper_bounds)`
  - set the active discrete variable upper bounds.
• void write (ostream &s) const
  write a variable constraints object to an ostream.

• void read (istream &s)
  read a variable constraints object from an istream.

Private Attributes

• short nonDFlag
  this flag is set if uncertain variables are active (the default is design variables are active; see constructor for logic).

• DakotaRealVector continuousDesignLowerBnds
  the continuous design lower bounds array.

• DakotaRealVector continuousDesignUpperBnds
  the continuous design upper bounds array.

• DakotalIntVector discreteDesignLowerBnds
  the discrete design lower bounds array.

• DakotalIntVector discreteDesignUpperBnds
  the discrete design upper bounds array.

• DakotaRealVector uncertainDistLowerBnds
  the uncertain distribution lower bounds array.

• DakotaRealVector uncertainDistUpperBnds
  the uncertain distribution upper bounds array.

• DakotaRealVector continuousStateLowerBnds
  the continuous state lower bounds array.

• DakotaRealVector continuousStateUpperBnds
  the continuous state upper bounds array.

• DakotalIntVector discreteStateLowerBnds
  the discrete state lower bounds array.

• DakotalIntVector discreteStateUpperBnds
  the discrete state upper bounds array.

• DakotaRealVector emptyRealVector
  an empty real vector returned in get functions when there are no variables corresponding to the request.

• DakotalIntVector emptyIntVector
  an empty int vector returned in get functions when there are no variables corresponding to the request.
6.47 FundamentalVarConstraints Class Reference

6.47.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which employs the default data view (no variable or domain type array merging).

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The FundamentalVarConstraints derived class separates the design, uncertain, and state variable types as well as the continuous and discrete domain types. The result is separate lower and upper bounds arrays for continuous design, discrete design, uncertain, continuous state, and discrete state variables. This is the default approach, so all iterators and strategies not specifically utilizing the All, Merged, or AllMerged views use this approach (see DakotaVariables::get_2variables(problem_db) for variables type selection; variables type is passed to the DakotaVarConstraints constructor in DakotaModel).

6.47.2 Constructor & Destructor Documentation

6.47.2.1 FundamentalVarConstraints::FundamentalVarConstraints (const ProblemDescDB & problem_db)

Constructor.

Extract fundamental lower and upper bounds (VariablesUtil is not used).

The documentation for this class was generated from the following files:

- FundamentalVarConstraints.H
- FundamentalVarConstraints.C
6.48 FundamentalVariables Class Reference

Derived class within the DakotaVariables hierarchy which employs the default data view (no variable or domain type array merging).

#include <FundamentalVariables.H>

Inheritance diagram for FundamentalVariables::

```
DakotaVariables     VariablesUtil
                  |                        | FundamentalVariables
```

**Public Methods**

- `FundamentalVariables ()`
  
  default constructor.

- `FundamentalVariables (const ProblemDescDB &problem_db)`
  
  standard constructor.

- `~FundamentalVariables ()`
  
  destructor.

- `size_t tv () const`
  
  Returns total number of vars.

- `size_t cv () const`
  
  Returns number of active continuous vars.

- `size_t dv () const`
  
  Returns number of active discrete vars.

- `const DakotaRealVector & continuous_variables () const`
  
  return the active continuous variables.

- `void continuous_variables (const DakotaRealVector &c_vars)`
  
  set the active continuous variables.

- `const DakotaIntVector & discrete_variables () const`
  
  return the active discrete variables.

- `void discrete_variables (const DakotaIntVector &d_vars)`
  
  set the active discrete variables.
• `const DakotaStringArray & continuous_variable_labels() const`
  return the active continuous variable labels.

• `void continuous_variable_labels(const DakotaStringArray &cv_labels)`
  set the active continuous variable labels.

• `const DakotaStringArray & discrete_variable_labels() const`
  return the active discrete variable labels.

• `void discrete_variable_labels(const DakotaStringArray &dv_labels)`
  set the active discrete variable labels.

• `const DakotaRealVector & inactive_continuous_variables() const`
  return the inactive continuous variables.

• `void inactive_continuous_variables(const DakotaRealVector &i_c_vars)`
  set the inactive continuous variables.

• `const DakotaIntVector & inactive_discrete_variables() const`
  return the inactive discrete variables.

• `void inactive_discrete_variables(const DakotaIntVector &i_d_vars)`
  set the inactive discrete variables.

• `size_t acv() const`
  returns total number of continuous vars.

• `size_t adv() const`
  returns total number of discrete vars.

• `DakotaRealVector all_continuous_variables() const`
  returns a single array with all continuous variables.

• `DakotaIntVector all_discrete_variables() const`
  returns a single array with all discrete variables.

• `void read(istream &s)`
  read a variables object from an istream.

• `void write(ostream &s) const`
  write a variables object to an ostream.

• `void read_annotated(istream &s)`
  read a variables object in annotated format from an istream.

• `void write_annotated(ostream &s) const`
  write a variables object in annotated format to an ostream.

• `void read(DakotaBiStream &s)`
  read a variables object from the binary restart stream.
void write (DakotaBoStream &s) const
  write a variables object to the binary restart stream.

void read (UnPackBuffer &s)
  read a variables object from a packed MPI buffer.

void write (PackBuffer &s) const
  write a variables object to a packed MPI buffer.

Private Methods

void copy_rep (const DakotaVariables *vars_rep)
  Used by copy() to copy the contents of a letter class.

Private Attributes

short nonDFlag
  this flag is set if uncertain variables are active (the default is design variables are active; see constructor for logic).

DakotaRealVector continuousDesignVars
  the continuous design variables array.

DakotaIntVector discreteDesignVars
  the discrete design variables array.

DakotaRealVector uncertainVars
  the uncertain variables array.

DakotaRealVector continuousStateVars
  the continuous state variables array.

DakotaIntVector discreteStateVars
  the discrete state variables array.

DakotaStringArray continuousDesignLabels
  the continuous design variables label array.

DakotaStringArray discreteDesignLabels
  the discrete design variables label array.

DakotaStringArray uncertainLabels
  the uncertain variables label array.

DakotaStringArray continuousStateLabels
  the continuous state variables label array.
• DakotaStringArray discreteStateLabels
  *the discrete state variables label array.*

• DakotaRealVector emptyRealVector
  *an empty real vector returned in get functions when there are no variables corresponding to the request.*

• DakotaIntVector emptyIntVector
  *an empty int vector returned in get functions when there are no variables corresponding to the request.*

• DakotaStringArray emptyStringArray
  *an empty label array returned in get functions when there are no variables corresponding to the request.*

**Friends**

• int operator==(const FundamentalVariables &vars1, const FundamentalVariables &vars2)
  *equality operator.*

### 6.48.1 Detailed Description

Derived class within the *DakotaVariables* hierarchy which employs the default data view (no variable or domain type array merging).

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The FundamentaVariables derived class separates the design, uncertain, and state variable types as well as the continuous and discrete domain types. The result is separate arrays for continuous design, discrete design, uncertain, continuous state, and discrete state variables. This is the default approach, so all iterators and strategies not specifically utilizing the All, Merged, or AllMerged views use this approach (see DakotaVariables::get_variables(problem_db)).

### 6.48.2 Constructor & Destructor Documentation

#### 6.48.2.1 FundamentalVariables::FundamentalVariables (const ProblemDescDB & problem_db)

*standard constructor.*

Extract fundamental variable types and labels (*VariablesUtil* is not used).

The documentation for this class was generated from the following files:

• FundamentalVariables.H
• FundamentalVariables.C
6.49 GetLongOpt Class Reference

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France).

```c
#include <CommandLineHandler.H>
```

Inheritance diagram for GetLongOpt:

```
GetLongOpt

CommandLineHandler
```

**Public Types**

- `enum OptType { Valueless, OptionalValue, MandatoryValue }

  enum for different types of values associated with command line options.

**Public Methods**

- **GetLongOpt** (const char optmark='\'-\')
  
  Constructor.

- **~GetLongOpt** ()
  
  Destructor.

- **int parse** (int argc, char *const *argv)
  
  parse the command line args (argc, argv).

- **int parse** (char *const str, char *const p)
  
  parse a string of options (typically given from the environment).

- **int enroll** (const char *const opt, const OptType t, const char *const desc, const char *const val)
  
  Add an option to the list of valid command options.

- **const char * retrieve** (const char *const opt) const
  
  Retrieve value of option.

- **void usage** (ostream &outfile=cout) const
  
  Print usage information to outfile.

- **void usage** (const char *str)
  
  Change header of usage output to str.
Private Methods

- char * basename (char * const p) const
  extract the base name from a string as delimited by '/'.

- int setcell (Cell *c, char *valtoken, char *nexttoken, const char *p)
  internal convenience function for setting Cell::value.

Private Attributes

- Cell * table
  option table.

- const char * ustring
  usage message.

- char * pname
  program basename.

- char optmarker
  option marker.

- int enroll_done
  finished enrolling.

- Cell * last
  last entry in option table.

6.49.1 Detailed Description

GetLongOpt is a general command line utility from S. Manoharan (Advanced Computer Research Institute, Lyon, France).

GetLongOpt manages the definition and parsing of "long options." Command line options can be abbreviated as long as there is no ambiguity. If an option requires a value, the value should be separated from the option either by whitespace or an "=".

6.49.2 Constructor & Destructor Documentation

6.49.2.1 GetLongOpt::GetLongOpt (const char optmark = ' -')

Constructor.

Constructor for GetLongOpt takes an optional argument: the option marker. If unspecified, this defaults to ' -', the standard (?) Unix option marker.
6.49.3 Member Function Documentation

6.49.3.1 int GetLongOpt::parse (int argc, char *const *argv)

parse the command line args (argc, argv).
A return value < 1 represents a parse error. Appropriate error messages are printed when errors are seen.
parse returns the the optind (see getopt(3)) if parsing is successful.

6.49.3.2 int GetLongOpt::parse (char *const str, char *const p)

parse a string of options (typically given from the environment).
A return value < 1 represents a parse error. Appropriate error messages are printed when errors are seen.
parsing takes two strings: the first one is the string to be parsed and the second one is a string to be prefixed
to the parse errors.

6.49.3.3 int GetLongOpt::enroll (const char *const opt, const OptType t, const char *const desc,
const char *const val)

Add an option to the list of valid command options.
enroll adds option specifications to its internal database. The first argument is the option string. The second
is an enum saying if the option is a flag (Valueless), if it requires a mandatory value (MandatoryValue) or
if it takes an optional value (OptionalValue). The third argument is a string giving a brief description of
the option. This description will be used by GetLongOpt::usage. GetLongOpt, for usage-printing, uses {$val}
to represent values needed by the options. {<$val>} is a mandatory value and {[$val]} is an optional
value. The final argument to enroll is the default string to be returned if the option is not specified. For
flags (options with Valueless), use "" (empty string, or in fact any arbitrary string) for specifying TRUE
and 0 (null pointer) to specify FALSE.

6.49.3.4 const char * GetLongOpt::retrieve (const char *const opt, const

Retrieve value of option.
The values of the options that are enrolled in the database can be retrieved using retrieve. This returns a
string and this string should be converted to whatever type you want. See atoi, atof, atol, etc. If a "parse"
is not done before retrieving all you will get are the default values you gave while enrolling! Ambiguities
while retrieving (may happen when options are abbreviated) are resolved by taking the matching option
that was enrolled last. For example, -{v} will expand to {-verify}. If you try to retrieve something you
didn’t enroll, you will get a warning message.

6.49.3.5 void GetLongOpt::usage (const char * str) [inline]

Change header of usage output to str.
GetLongOpt::usage is overloaded. If passed a string "str", it sets the internal usage string to "str". Other-
wise it simply prints the command usage.
The documentation for this class was generated from the following files:
- CommandLineHandler.H
- CommandLineHandler.C


## 6.50 HermiteSurf Class Reference

Derived approximation class for Hermite polynomials (global approximation).

```cpp
#include <HermiteSurf.H>
```

Inheritance diagram for HermiteSurf:

```
DakotaApproximation

HermiteSurf
```

### Public Methods

- **HermiteSurf** (const `ProblemDescDB &prob_db`)
  
  *Constructor.*

- **~HermiteSurf** ()
  
  *Destructor.*

### Protected Methods

- **int required_samples** (int `num_vars`)
  
  *Return the minimum number of samples required to build the derived class approximation type in `num_vars` dimensions.*

- **void find_coefficients** ()
  
  *Find the Polynomial Chaos coefficients for the response surface.*

- **Real get_value** (const `DakotaRealVector &x`)
  
  *Retrieve the approximate function value for a given parameter vector.*

- **void get_num_chaos** ()
  
  *Calculate number of Chaos according to the highest order of Chaos.*

### Private Methods

- **DakotaRealVector get_chaos** (const `DakotaRealVector &x`, int `order`)
  
  *Calculate the Polynomial Chaos from variables.*
6.50 HermiteSurf Class Reference

Private Attributes

- DakotaRealVector chaosCoeffs
  numChaos entries.

- DakotaRealVectorArray chaosSamples
  numChaos×numCurrentPoints entries.

- int numChaos
  Number of terms in Polynomial Chaos Expansion.

- int highestOrder
  Highest order of Hermite Polynomials in Expansion.

Static Private Attributes

- int index = -1
  index determining which term in the series.

6.50.1 Detailed Description

Derived approximation class for Hermite polynomials (global approximation).

The HermiteSurf class provides a global approximation based on Hermite polynomials. It is used primarily for polynomial chaos expansions (for stochastic finite element approaches to uncertainty quantification).

The documentation for this class was generated from the following files:

- HermiteSurf.H
- HermiteSurf.C
## 6.51 HierLayeredModel Class Reference

Derived model class within the layered model branch for managing hierarchical surrogates (models of varying fidelity).

```cpp
#include <HierLayeredModel.H>
```

Inheritance diagram for HierLayeredModel:

```
DakotaModel
```
```
LayeredModel
```
```
HierLayeredModel
```

### Public Methods

- **HierLayeredModel** (ProblemDescDB &problem_db)
  
  constructor.

- `~HierLayeredModel()`
  
  destructor.

### Protected Methods

- void **derived_compute_response** (const DakotaIntArray &asv)
  
  portion of `compute_response()` specific to `HierLayeredModel`.

- void **derived_async_compute_response** (const DakotaIntArray &asv)
  
  portion of `async_compute_response()` specific to `HierLayeredModel`.

- const DakotaArray< DakotaResponse > & **derived_synchronize** ()
  
  portion of `synchronize()` specific to `HierLayeredModel`.

- const DakotaList< DakotaResponse > & **derived_synchronize_nowait** ()
  
  portion of `synchronize_nowait()` specific to `HierLayeredModel`.

- DakotaModel & **subordinate_model** ()
  
  return highFidelityModel to `SurrBasedOptStrategy`.

- void **build_approximation** ()
  
  use highFidelityModel to compute the truth values needed for correction of lowFidelityInterface results.

- DakotaString **local_eval_synchronization** ()
return lowFidelityInterface local evaluation synchronization setting.

- const DakotaIntArray & synchronize_nowait_completions ()
  return completion id’s matching response list from synchronize_nowait (request forwarded to lowFidelityInterface).

- short derived_master_overload () const
  flag which prevents overloading the master with a multiprocessor evaluation (request forwarded to lowFidelityInterface).

- void derived_init_communicators (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  portion of init_communicators() specific to HierLayeredModel (request forwarded to lowFidelityInterface).

- void free_communicators ()
  deallocate communicator partitions for the HierLayeredModel (request forwarded to lowFidelityInterface).

- void serve ()
  Service job requests received from the master. Completes when a termination message is received from stop_servers() (request forwarded to lowFidelityInterface).

- void stop_servers ()
  executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (request forwarded to lowFidelityInterface).

- int total_eval_counter () const
  return the total evaluation count for the HierLayeredModel (request forwarded to lowFidelityInterface).

- int new_eval_counter () const
  return the new evaluation count for the HierLayeredModel (request forwarded to lowFidelityInterface).

Private Attributes

- DakotaInterface lowFidelityInterface
  manages the approximate low fidelity function evaluations.

- DakotaModel highFidelityModel
  provides truth evaluations for computing corrections to the low fidelity results.

- DakotaResponse highFidResponse
  the high fidelity response is computed in build_approximation() and needs class scope for use in automatic surrogate construction in derived compute_response functions.

- DakotaIntArray evalIdList
  bookkeeps fnEvalId’s for correction of asynchronous low fidelity evaluations.
6.51.1 Detailed Description

Derived model class within the layered model branch for managing hierarchical surrogates (models of varying fidelity).

The HierLayeredModel class manages hierarchical models of varying fidelity. In particular, it uses a low fidelity model as a surrogate for a high fidelity model. The class contains a lowFidelityInterface which manages the approximate low fidelity function evaluations and a highFidelityModel which provides truth evaluations for computing corrections to the low fidelity results.

6.51.2 Member Function Documentation

6.51.2.1 void HierLayeredModel::derived_computeresponse (const DakotaIntArray & asv) 
[protected, virtual]

portion of compute_response() specific to HierLayeredModel.

Evaluate the approximate response using lowFidelityInterface, compute the high fidelity response with build_approximation() (if not performed previously), and, if correction is active, correct the low fidelity results.

Reimplemented from DakotaModel.

6.51.2.2 void HierLayeredModel::derived_asynch_computeresponse (const DakotaIntArray & asv) [protected, virtual]

portion of asynch_compute_response() specific to HierLayeredModel.

Evaluate the approximate response using an asynchronous lowFidelityInterface mapping and compute the high fidelity response with build_approximation() (for correcting the low fidelity results in derived_synchronize() and derived_synchronize_nowait()) if not performed previously.

Reimplemented from DakotaModel.

6.51.2.3 const DakotaArray< DakotaResponse > & HierLayeredModel::derived_synchronize ()

[protected, virtual]

portion of synchronize() specific to HierLayeredModel.

Perform a blocking retrieval of all asynchronous evaluations from lowFidelityInterface and, if automatic correction is on, apply correction to each response in the array.

Reimplemented from DakotaModel.

6.51.2.4 const DakotaList< DakotaResponse > & HierLayeredModel::derived_synchronize_nowait () [protected, virtual]

portion of synchronize_nowait() specific to HierLayeredModel.

Perform a nonblocking retrieval of currently available asynchronous evaluations from lowFidelityInterface and, if automatic correction is on, apply correction to each response in the list.

Reimplemented from DakotaModel.
The documentation for this class was generated from the following files:

- HierLayeredModel.H
- HierLayeredModel.C
6.52 KrigApprox Class Reference

Utility class for kriging interpolation.

#include <KSMSurf.H>

Public Methods

- `KrigApprox` (int, int, const DakotaRealVector &, const DakotaRealVector &, const DakotaRealVector &)  
  constructor.
- `~KrigApprox` ()  
  destructor.
- void `ModelBuild` (int, int, const DakotaRealVector &, const DakotaRealVector &, int)  
  Function to compute vector and matrix terms in the kriging surface.
- Real `ModelApply` (int, int, const DakotaRealVector &)  
  Function returns a response value using the kriging surface.

Private Attributes

- int `N1`  
  Size variable for CONMIN arrays. See CONMIN manual.
- int `N2`  
  Size variable for CONMIN arrays. See CONMIN manual.
- int `N3`  
  Size variable for CONMIN arrays. See CONMIN manual.
- int `N4`  
  Size variable for CONMIN arrays. See CONMIN manual.
- int `N5`  
  Size variable for CONMIN arrays. See CONMIN manual.
- int `conminSingleArray`  
  Array size parameter needed in interface to CONMIN.
- int `numcon`  
  CONMIN variable: Number of constraints.
- int `NFDG`
CONMIN variable: Finite difference flag.

- int IPRINT
  CONMIN variable: Flag to control amount of output data.

- int ITMAX
  CONMIN variable: Flag to specify the maximum number of iterations.

- Real FDCH
  CONMIN variable: Relative finite difference step size.

- Real FDCHM
  CONMIN variable: Absolute finite difference step size.

- Real CT
  CONMIN variable: Constraint thickness parameter.

- Real CTMIN
  CONMIN variable: Minimum absolute value of CT used during optimization.

- Real CTL
  CONMIN variable: Constraint thickness parameter for linear and side constraints.

- Real CTLMIN
  CONMIN variable: Minimum value of CTL used during optimization.

- Real DELFUN
  CONMIN variable: Relative convergence criterion threshold.

- Real DABFUN
  CONMIN variable: Absolute convergence criterion threshold.

- int conminInfo
  CONMIN variable: status flag for optimization.

- Real * S
  Internal CONMIN array.

- Real * G1
  Internal CONMIN array.

- Real * G2
  Internal CONMIN array.

- Real * B
  Internal CONMIN array.

- Real * C
  Internal CONMIN array.
- int * MS1
  *Internal CONMIN array.

- Real * SCAL
  *Internal CONMIN array.

- Real * DF
  *Internal CONMIN array.

- Real * A
  *Internal CONMIN array.

- int * ISC
  *Internal CONMIN array.

- int * IC
  *Internal CONMIN array.

- Real * conmin_theta_vars
  *Temporary array of design variables used by CONMIN (length N1 = numdv+2).

- Real * conmin_theta_lower_bnds
  *Temporary array of lower bounds used by CONMIN (length N1 = numdv+2).

- Real * conmin_theta_upper_bnds
  *Temporary array of upper bounds used by CONMIN (length N1 = numdv+2).

- Real ALPHAX
  *Internal CONMIN variable: 1-D search parameter.

- Real ABOBJ1
  *Internal CONMIN variable: 1-D search parameter.

- Real THETA
  *Internal CONMIN variable: mean value of push-off factor.

- Real PHI
  *Internal CONMIN variable: "participation coefficient".

- int NSIDE
  *Internal CONMIN variable: side constraints parameter.

- int NSCAL
  *Internal CONMIN variable: scaling control parameter.

- int NACMX1
  *Internal CONMIN variable: estimate of 1+(max # of active constraints).

- int LINOBJ
  *Internal CONMIN variable: linear objective function identifier (unused).
- int **ITRM**
  *Internal CONMIN variable: diminishing return criterion iteration number.*

- int **ICNDIR**
  *Internal CONMIN variable: conjugate direction restart parameter.*

- int **IGOTO**
  *Internal CONMIN variable: internal optimization termination flag.*

- int **NAC**
  *Internal CONMIN variable: number of active and violated constraints.*

- int **INFOG**
  *Internal CONMIN variable: gradient information flag.*

- int **ITER**
  *Internal CONMIN variable: iteration count.*

- int **iFlag**
  *Fortran77 flag for kriging computations.*

- Real **betaHat**
  *Estimate of the beta term in the kriging model.*

- Real **maxLikelihoodEst**
  *Error term computed via Maximum Likelihood Estimation.*

- int **numNewPts**
  *Size variable for the arrays used in kriging computations.*

- int **numSampQuad**
  *Size variable for the arrays used in kriging computations.*

- Real * **thetaVector**
  *Array of correlation parameters for the kriging model.*

- Real * **xMatrix**
  *A 2-D array of design points used to build the kriging model.*

- Real * **yValueVector**
  *Array of response values corresponding to the array of design points.*

- Real * **xNewVector**
  *A 2-D array of design points where the kriging model will be evaluated.*

- Real * **yNewVector**
  *Array of response values corresponding to the design points specified in xNewVector.*

- Real * **thetaLoBndVector**
Array of lower bounds in optimizer-to-kriging interface.

- Real * thetaUpBndVector
  
  Array of upper bounds in optimizer-to-kriging interface.

- Real * constraintVector
  
  Array of constraint values (used with optimizer).

- Real * rhsTermsVector
  
  Internal array for kriging Fortran77 code: matrix algebra result.

- int * iPivotVector
  
  Internal array for kriging Fortran77 code: pivot vector for linear algebra.

- Real * correlationMatrix
  
  Internal array for kriging Fortran77 code: correlation matrix.

- Real * invcorrelMatrix
  
  Internal array for kriging Fortran77 code: inverse correlation matrix.

- Real * fValueVector
  
  Internal array for kriging Fortran77 code: response value vector.

- Real * fRinvVector
  
  Internal array for kriging Fortran77 code: vector\times matrix result.

- Real * yfbVector
  
  Internal array for kriging Fortran77 code: vector arithmetic result.

- Real * yfbRinvVector
  
  Internal array for kriging Fortran77 code: vector\times matrix result.

- Real * rXhatVector
  
  Internal array for kriging Fortran77 code: local correlation vector.

- Real * workVector
  
  Internal array for kriging Fortran77 code: temporary storage.

- Real * workVectorQuad
  
  Internal array for kriging Fortran77 code: temporary storage.

- int * iworkVector
  
  Internal array for kriging Fortran77 code: temporary storage.

### 6.52.1 Detailed Description

Utility class for kriging interpolation.

The KrigApprox class provides utilities for the KrigingSurf class. It is based on the Ph.D. thesis work of Tony Giunta.
6.52.2 Member Function Documentation

6.52.2.1 Real KrigApprox::ModelApply (int numVars, int numCurrentPoints, const DakotaRealVector & x_array)

Function returns a response value using the kriging surface.
The response value is computed at the design point specified by the DakotaRealVector function argument.

6.52.3 Member Data Documentation

6.52.3.1 int KrigApprox::N1 [private]
Size variable for CONMIN arrays. See CONMIN manual.
N1 = number of variables + 2

6.52.3.2 int KrigApprox::N2 [private]
Size variable for CONMIN arrays. See CONMIN manual.
N2 = number of constraints + 2*(number of variables)

6.52.3.3 int KrigApprox::N3 [private]
Size variable for CONMIN arrays. See CONMIN manual.
N3 = Maximum possible number of active constraints.

6.52.3.4 int KrigApprox::N4 [private]
Size variable for CONMIN arrays. See CONMIN manual.
N4 = Maximum(N3,number of variables)

6.52.3.5 int KrigApprox::N5 [private]
Size variable for CONMIN arrays. See CONMIN manual.
N5 = 2*(N4)

6.52.3.6 Real KrigApprox::CT [private]
CONMIN variable: Constraint thickness parameter.
The value of CT decreases in magnitude during optimization.
6.52.3.7 Real* KrigApprox::S [private]
Internal CONMIN array.
Move direction in N-dimensional space.

6.52.3.8 Real* KrigApprox::G1 [private]
Internal CONMIN array.
Temporary storage of constraint values.

6.52.3.9 Real* KrigApprox::G2 [private]
Internal CONMIN array.
Temporary storage of constraint values.

6.52.3.10 Real* KrigApprox::B [private]
Internal CONMIN array.
Temporary storage for computations involving array S.

6.52.3.11 Real* KrigApprox::C [private]
Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.52.3.12 int* KrigApprox::MS1 [private]
Internal CONMIN array.
Temporary storage for use with arrays B and S.

6.52.3.13 Real* KrigApprox::SCAL [private]
Internal CONMIN array.
Vector of scaling parameters for design parameter values.

6.52.3.14 Real* KrigApprox::DF [private]
Internal CONMIN array.
Temporary storage for analytic gradient data.

6.52.3.15 Real* KrigApprox::A [private]
Internal CONMIN array.
Temporary 2-D array for storage of constraint gradients.
6.52.3.16 int KrigApprox::ISC [private]

Internal CONMIN array.
Array of flags to identify linear constraints. (not used in this implementation of CONMIN)

6.52.3.17 int KrigApprox::IC [private]

Internal CONMIN array.
Array of flags to identify active and violated constraints

6.52.3.18 int KrigApprox::iFlag [private]

Fortran77 flag for kriging computations.
iFlag=1 computes vector and matrix terms for the kriging surface, iFlag=2 computes the response value (using kriging) at the user-supplied design point.
The documentation for this class was generated from the following files:

- KSMSurf.H
- KSMSurf.C
**6.53 KrigingSurf Class Reference**

Derived approximation class for kriging interpolation.

```cpp
#include <KSMSurf.H>
```

Inheritance diagram for KrigingSurf:

```
  DakotaApproximation
       |        |
       v        v
KrigingSurf
```

**Public Methods**

- `KrigingSurf` (const `ProblemDescDB` &`problem_db`)  
  * constructor.

- `~KrigingSurf` ()
  * destructor.

**Protected Methods**

- void `findCoefficients` ()
  * calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

- int `required_samples` (int `num_vars`)
  * return the minimum number of samples required to build the derived class approximation type in `num_vars` dimensions.

- Real `getValue` (const `DakotaRealVector` &`x`)
  * retrieve the approximate function value for a given parameter vector.

**Private Attributes**

- `KrigApprox` * `krigObject`
  * Kriging Surface object declaration.

- `DakotaRealVector` `x` * `matrix`
  * A 2-d array of all sample sites (design points) used to create the kriging surface.

- `DakotaRealVector` `f` * `array`
  * An array of response values; one response value per sample site.
6.53 KrigingSurf Class Reference

- DakotaRealVector correlationVector
  
  An array of correlation parameter values used to build the kriging surface.

- int runConminFlag
  
  Flag to run CONMIN (value=1) or use user-supplied correlations (value=0).

6.53.1 Detailed Description

Derived approximation class for kriging interpolation.

The KrigingSurf class uses a the kriging approach to interpolate between data points. It is based on the Ph.D. thesis work of Tony Giunta.

The documentation for this class was generated from the following files:

- KSMSurf.H
- KSMSurf.C
### 6.54 LayeredModel Class Reference

Base class for the layered models (SurrLayeredModel and HierLayeredModel).

```c
#include <LayeredModel.h>
```

Inheritance diagram for LayeredModel:

```
  DakotaModel
  ↓
  LayeredModel
  ↓
  HierLayeredModel  SurrLayeredModel
```

**Protected Methods**

- **LayeredModel** (ProblemDescDB &problem_db)
  
  *constructor.*

- **~LayeredModel** ()
  
  *destructor.*

- void **compute_correction** (const DakotaResponse &truth_response, const DakotaResponse &approx_response, const DakotaRealVector &vars)
  
  *compute the correction required to bring approx_response into agreement with truth_response.*

- void **apply_correction** (DakotaResponse &approx_response, const DakotaRealVector &vars, short quiet_flag=0)
  
  *apply the correction computed in compute_correction() to approx_response.*

- void **check_submodel_compatibility** (const DakotaModel &sub_model)
  
  *verify compatibility between LayeredModel attributes and attributes of the submodel (SurrLayeredModel::actualModel or HierLayeredModel::highFidelityModel).*

- void **auto_correction** (short correction_flag)
  
  *sets autoCorrection to ON (1) or OFF (0).*

**Protected Attributes**

- **DakotaArray < DakotaResponse > correctedResponseArray**
  
  *array of corrected responses used in derived_synchronize() functions.*

- **DakotaList < DakotaResponse > correctedResponseList**
  
  *list of corrected responses used in derived_synchronize_nowait() functions.*
- **DakotaList** < DakotaRealVector > rawCVarsList
  
  *list of raw continuous variables used by applyCorrection(). DakotaModel::varsList cannot be used for this purpose since it does not contain lower level variables sets from finite differencing.*

- **DakotaString** correctionType
  
  *correction approach to be used: offset, scaled, or beta.*

- **int** approxBuilds
  
  *number of calls to buildApproximation().*

- **short** autoCorrection
  
  *a flag which controls the use of applyCorrection() in SurrLayeredModel and HierLayeredModel approximate response computations.*

**Private Attributes**

- **DakotaRealVector** offsetValues
  
  *values with which to offset the function values in an approximate response in order to apply a truth model correction.*

- **DakotaRealVector** scaleFactors
  
  *values with which to scale the function values, gradients, and Hessians in an approximate response in order to apply a truth model correction.*

- **short** correctionComputed
  
  *flag used to indicate whether or not a correction is available.*

- **short** badScalingFlag
  
  *flag used to indicate function values near zero for scaled corrections; triggers an automatic switch from scaled to offset correction.*

- **DakotaRealVector** betaFcnRatio
  
  *Beta-correction scaling term: ratio of high-fidelity model value to low fidelity value at x=x\_center.*

- **DakotaRealMatrix** betaGrads
  
  *Beta-correction scaling term: gradient of beta at x=x\_center.*

- **DakotaRealVector** betaCenterPt
  
  *Beta-correction scaling term: center point of the trust region.*

- **DakotaRealVector** betaApproxCenterVals
  
  *Function values at the center of the trust region which are needed as a fall back if the current function values are unavailable when applying the beta-correction.*

- **DakotaRealMatrix** betaApproxCenterGrads
  
  *Gradient values at the center of the trust region which are needed as a fall back if the current function gradients are unavailable when applying the beta-correction.*
6.54.1 Detailed Description

Base class for the layered models (SurrLayeredModel and HierLayeredModel).

The LayeredModel class provides common functions to derived classes for computing and applying corrections to approximations.

6.54.2 Member Function Documentation

6.54.2.1 void LayeredModel::compute_correction (const DakotaResponse & truth_response, const DakotaResponse & approx_response, const DakotaRealVector & c_yars) [protected, virtual]
compute the correction required to bring approx_response into agreement with truth_response.

Compute a correction for approximate responses based on an offset, scaled, or beta correction approach. The offset and scaled approaches will correct the approximate function values to match the truth function values at a single point in the parameter space (e.g., the center of a trust region). In the "beta" correction approach, the function value and the function gradient are matched at a single point. The beta-correction is similar to the scaled-correction method, but the scaled-correction uses a scalar value for each response function, whereas the beta-correction uses a **scaling function** for each response function that varies w.r.t. position in the parameter space.

Reimplemented from DakotaModel.

6.54.3 Member Data Documentation

6.54.3.1 int LayeredModel::approxBuilds [protected]
number of calls to build_approximation().

used as a flag to automatically build the approximation if one of the derived compute_response functions is called prior to build_approximation().

6.54.3.2 short LayeredModel::autoCorrection [protected]
a flag which controls the use of apply_correction() in SurrLayeredModel and HierLayeredModel approximate response computations.
the default is ON once `compute_correction()` has been called. However this should be overridden when a new correction is desired, since `compute_correction()` no longer automatically backs out an old correction.

The documentation for this class was generated from the following files:

- `LayeredModel.H`
- `LayeredModel.C`
6.55   MARSSurf Class Reference

Derived approximation class for multivariate adaptive regression splines.

```
#include <MARSSurf.H>
```

Inheritance diagram for MARSSurf::

```
DakotaApproximation
  
MARSSurf
```

**Public Methods**

- `MARSSurf` (const `ProblemDescDB` &`problem_db`)
  
  *constructor.*

- `~MARSSurf` ()
  
  *destructor.*

**Protected Methods**

- `int required_samples` (int `num_vars`)
  
  *return the minimum number of samples required to build the derived class approximation type in `num_vars` dimensions.*

- `void find_coefficients` ()
  
  *calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.*

- `Real get_value` (const `DakotaRealVector` &`x`)
  
  *retrieve the approximate function value for a given parameter vector.*

**Private Attributes**

- `int * flags`
  
  *variable type declarations (ordinal, excluded, categorical).*

- `Mars * marsObject`
  
  *pointer to the Mars object (MARS wrapper provided as part of DDACE).*
6.55.1 Detailed Description

Derived approximation class for multivariate adaptive regression splines.

The MARSSurf class provides a global approximation based on regression splines. It employs the C++ wrapper developed by the DDACE team for the Multivariate Adaptive Regression Splines (MARS) package from Prof. Jerome Friedman of Stanford University Dept. of Statistics.

The documentation for this class was generated from the following files:

- MARSSurf.H
- MARSSurf.C
6.56 MergedVarConstraints Class Reference

Derived class within the DakotaVarConstraints hierarchy which employs the merged data view.

#include <MergedVarConstraints.H>

Inheritance diagram for MergedVarConstraints::

```
DakotaVarConstraints   VariablesUtil
                        |       |
                        |       |
                        |       |
                        |
MergedVarConstraints
```

Public Methods

- **MergedVarConstraints** (const ProblemDescDB &problem_db)
  constructor.

- **~MergedVarConstraints** ()
  destructor.

- const DakotaRealVector & **continuous_lower_bounds** () const
  return the active continuous variable lower bounds.

- void **continuous_lower_bounds** (const DakotaRealVector &c_lbounds)
  set the active continuous variable lower bounds.

- const DakotaRealVector & **continuous_upper_bounds** () const
  return the active continuous variable upper bounds.

- void **continuous_upper_bounds** (const DakotaRealVector &c_ubounds)
  set the active continuous variable upper bounds.

- const DakotaIntVector & **discrete_lower_bounds** () const
  return the active discrete variable lower bounds.

- void **discrete_lower_bounds** (const DakotaIntVector &d_lbnds)
  set the active discrete variable lower bounds.

- const DakotaIntVector & **discrete_upper_bounds** () const
  return the active discrete variable upper bounds.

- void **discrete_upper_bounds** (const DakotaIntVector &d_ubnds)
  set the active discrete variable upper bounds.

- void **write** (ostream &s) const
write a variable constraints object to an ostream.

- void read (istream &s)
  read a variable constraints object from an istream.

Private Attributes

- DakotaRealVector mergedDesignLowerBnds
  a design lower bounds array merging continuous and discrete domains (integer values promoted to reals).

- DakotaRealVector mergedDesignUpperBnds
  a design upper bounds array merging continuous and discrete domains (integer values promoted to reals).

- DakotaRealVector uncertainDistLowerBnds
  the uncertain distribution lower bounds array (no discrete uncertain to merge).

- DakotaRealVector uncertainDistUpperBnds
  the uncertain distribution upper bounds array (no discrete uncertain to merge).

- DakotaRealVector mergedStateLowerBnds
  a state lower bounds array merging continuous and discrete domains (integer values promoted to reals).

- DakotaRealVector mergedStateUpperBnds
  a state upper bounds array merging continuous and discrete domains (integer values promoted to reals).

- DakotaIntVector emptyIntVector
  an empty int vector returned in get functions when there are no variables corresponding to the request.

6.56.1 Detailed Description

Derived class within the DakotaVarConstraints hierarchy which employs the merged data view.

Derived variable constraints classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MergedVarConstraints derived class combines continuous and discrete domain types but separates design, uncertain, and state variable types. The result is merged design bounds arrays (mergedDesignLowerBnds, mergedDesignUpperBnds), uncertain distribution bounds arrays (uncertainDistLowerBnds, uncertainDistUpperBnds), and merged state bounds arrays (mergedStateLowerBnds, mergedStateUpperBnds). The branch and bound strategy uses this approach (see DakotaVariables::get_variables(problem.db) for variables type selection; variables type is passed to the DakotaVarConstraints constructor in DakotaModel).

6.56.2 Constructor & Destructor Documentation
6.56.2.1 MergedVarConstraints::MergedVarConstraints (const ProblemDescDB & problem_db)

constructor.

Extract fundamental lower and upper bounds and merge continuous and discrete domains to create merged-DesignLowerBnds, mergedDesignUpperBnds, mergedStateLowerBnds, and mergedStateUpperBnds using utilities from VariablesUtil (uncertain distribution bounds do not require any merging).

The documentation for this class was generated from the following files:

- MergedVarConstraints.H
- MergedVarConstraints.C
6.57 MergedVariables Class Reference

Derived class within the DakotaVariables hierarchy which employs the merged data view.

```c++
#include <MergedVariables.H>
```

Inheritance diagram for MergedVariables::

```
DakotaVariables | VariablesUtil
    | MergedVariables
```

Public Methods

- **MergedVariables ()**
  
  *default constructor.*

- **MergedVariables (const ProblemDescDB &problem_db)**
  
  *standard constructor.*

- **~MergedVariables ()**
  
  *destructor.*

- **size_t tv () const**
  
  *Returns total number of vars.*

- **size_t cv () const**
  
  *Returns number of active continuous vars.*

- **size_t dv () const**
  
  *Returns number of active discrete vars.*

- **const DakotaRealVector & continuous_variables () const**
  
  *return the active continuous variables.*

- **void continuous_variables (const DakotaRealVector &c_vars)**
  
  *set the active continuous variables.*

- **const DakotaIntVector & discrete_variables () const**
  
  *return the active discrete variables.*

- **void discrete_variables (const DakotaIntVector &d_vars)**
  
  *set the active discrete variables.*

- **const DakotaStringArray & continuous_variable_labels () const**
return the active continuous variable labels.

- void **continuous_variable_labels** (const DakotaStringArray &cv_labels)
  set the active continuous variable labels.

- const DakotaStringArray & **discrete_variable_labels** () const
  return the active discrete variable labels.

- void **discrete_variable_labels** (const DakotaStringArray &dv_labels)
  set the active discrete variable labels.

- const DakotaRealVector & **inactive_continuous_variables** () const
  return the inactive continuous variables.

- void **inactive_continuous_variables** (const DakotaRealVector &ic_vars)
  set the inactive continuous variables.

- const DakotaIntVector & **inactive_discrete_variables** () const
  return the inactive discrete variables.

- void **inactive_discrete_variables** (const DakotaIntVector &id_vars)
  set the inactive discrete variables.

- size_t acv () const
  returns total number of continuous vars.

- size_t adv () const
  returns total number of discrete vars.

- DakotaRealVector **all_continuous_variables** () const
  returns a single array with all continuous variables.

- DakotaIntVector **all_discrete_variables** () const
  returns a single array with all discrete variables.

- void **read** (istream &s)
  read a variables object from an istream.

- void **write** (ostream &s) const
  write a variables object to an ostream.

- void **read_annotated** (istream &s)
  read a variables object in annotated format from an istream.

- void **write_annotated** (ostream &s) const
  write a variables object in annotated format to an ostream.

- void **read** (DakotaBiStream &s)
  read a variables object from the binary restart stream.
- void **write** (DakotaBoStream &s) const
  
  write a variables object to the binary restart stream.

- void **read** (UnPackBuffer &s)
  
  read a variables object from a packed MPI buffer.

- void **write** (PackBuffer &s) const
  
  write a variables object to a packed MPI buffer.

### Private Methods

- void **copy** (const DakotaVariables *vars_rep)
  
  Used by **copy**() to copy the contents of a letter class.

### Private Attributes

- DakotaRealVector **mergedDesignVars**
  
  a design variables array merging continuous and discrete domains (integer values promoted to reals).

- DakotaRealVector **uncertainVars**
  
  the uncertain variables array (no discrete uncertain to merge).

- DakotaRealVector **mergedStateVars**
  
  a state variables array merging continuous and discrete domains (integer values promoted to reals).

- DakotaStringArray **mergedDesignLabels**
  
  a label array combining continuous design and discrete design labels.

- DakotaStringArray **uncertainLabels**
  
  the uncertain variables label array (no discrete uncertain to combine).

- DakotaStringArray **mergedStateLabels**
  
  a label array combining continuous state and discrete state labels.

- DakotaIntVector **emptyIntVector**
  
  an empty int vector returned in get functions when there are no variables corresponding to the request.

- DakotaStringArray **emptyStringArray**
  
  an empty label array returned in get functions when there are no variables corresponding to the request.

### Friends

- int **operator==** (const MergedVariables &vars1, const MergedVariables &vars2)
  
  equality operator.
6.57.1 Detailed Description

Derived class within the DakotaVariables hierarchy which employs the merged data view.

Derived variables classes take different views of the design, uncertain, and state variable types and the continuous and discrete domain types. The MergedVariables derived class combines continuous and discrete domain types but separates design, uncertain, and state variable types. The result is a single continuous array of design variables (mergedDesignVars), a single continuous array of uncertain variables (uncertainVars), and a single continuous array of state variables (mergedStateVars). The branch and bound strategy uses this approach (see DakotaVariables::getVariables(problem_db)).

6.57.2 Constructor & Destructor Documentation

6.57.2.1 MergedVariables::MergedVariables (const ProblemDescDB & problem_db)

standard constructor.

Extract fundamental variable types and labels and merge continuous and discrete domains to create mergedDesignVars, mergedStateVars, mergedDesignLabels, and mergedStateLabels using utilities from VariablesUtil (uncertain variables and labels do not require any merging).

The documentation for this class was generated from the following files:

- MergedVariables.H
- MergedVariables.C
6.58 MultilevelOptStrategy Class Reference

Strategy for hybrid optimization using multiple optimizers on multiple models of varying fidelity.

```cpp
#include <MultilevelOptStrategy.H>
```

Inheritance diagram for MultilevelOptStrategy:

```
    DakotaStrategy
     |
    MultilevelOptStrategy
```

### Public Methods

- **MultilevelOptStrategy** (ProblemDescDB &problem_db)
  
  *constructor.*

- **~MultilevelOptStrategy**()
  
  *destructor.*

- **void run_strategy**()
  
  *Performs the hybrid optimization strategy by executing multiple iterators on different models of varying fidelity.*

### Private Methods

- **void run_coupled**()
  
  *run a tightly coupled hybrid.*

- **void run_uncoupled**()
  
  *run an uncoupled hybrid.*

- **void run_uncoupled_adaptive**()
  
  *run an uncoupled adaptive hybrid.*

### Private Attributes

- **DakotaString multiLevelType**
  
  *coupled, uncoupled, or uncoupled_adaptive.*

- **DakotaStringList methodList**
  
  *the list of method identifiers.*
• int numIterators
  number of methods in methodList.

• Real localSearchProb
  the probability of running a local search refinement within phases of the global optimization for coupled hybrids.

• Real progressMetric
  the amount of progress made in a single iterator++ cycle within an uncoupled adaptive hybrid.

• Real progressThreshold
  when the progress metric falls below this threshold, the uncoupled adaptive hybrid switches to the next method.

• DakotaArray< DakotaIterator > selectedIterators
  the set of iterators, one for each entry in methodList.

• DakotaArray< DakotaModel > userDefinedModels
  the set of models, one for each iterator.

6.58.1 Detailed Description

Strategy for hybrid optimization using multiple optimizers on multiple models of varying fidelity.

This strategy has three approaches to hybrid optimization: (1) the uncoupled hybrid runs one method to completion, passes its best results as the starting point for a subsequent method, and continues this succession until all methods have been executed; (2) the uncoupled adaptive hybrid is similar to the uncoupled hybrid, except that the stopping rules for the optimizers are controlled adaptively by the strategy instead of internally by each optimizer; and (3) the coupled hybrid uses multiple methods in close coordination, generally using a local search optimizer repeatedly within a global optimizer (the local search optimizer refines candidate optima which are fed back to the global optimizer). The uncoupled strategies only pass information forward, whereas the coupled strategy allows both feed forward and feedback. Note that while the strategy is targeted at optimizers, any iterator may be used so long as it defines the notion of a final solution which can be passed as the starting point for subsequent iterators.

6.58.2 Member Function Documentation

6.58.2.1 void MultilevelOptStrategy::run_coupled () [private]

run a tightly coupled hybrid.

In the coupled case, use is made of external hybridization capabilities, such as those available in the global/local hybrids from SGOPT. This function is responsible only for publishing the local optimizer selection to the global optimizer and then invoking the global optimizer; the logic of method switching is handled entirely within the global optimizer. Status: incomplete.
6.58.2.2 void MultilevelOptStrategy::run_uncoupled () [private]

run an uncoupled hybrid.
In the uncoupled nonadaptive case, there is no interference with the iterators. Each runs until its own convergence criteria is satisfied (using iterator.run_iterator()). Status: fully operational.

6.58.2.3 void MultilevelOptStrategy::run_uncoupled_adaptive () [private]

run an uncoupled adaptive hybrid.
In the uncoupled adaptive case, there is interference with the iterators through the use of the ++ overloaded operator. iterator++. runs the iterator for one cycle, after which a progress metric is computed. This progress metric is used to dictate method switching instead of each iterator’s internal convergence criteria. Status: incomplete.

The documentation for this class was generated from the following files:

- MultilevelOptStrategy.H
- MultilevelOptStrategy.C
### 6.59 NestedModel Class Reference

Derived model class which performs a complete sub-iterator execution within every evaluation of the model.

```
#include <NestedModel.H>
```

Inheritance diagram for NestedModel:

```
        DakotaModel
           |
           v
          NestedModel
```

#### Public Methods

- **NestedModel** (ProblemDescDB &problem_db)
  
  constructor.

- **~NestedModel** ()
  
  destructor.

#### Protected Methods

- void **derived_computeresponse** (const DakotaIntArray &asv)
  
  portion of `compute_response()` specific to NestedModel.

- void **derived_asynch Computeresponse** (const DakotaIntArray &asv)
  
  portion of `asynch_compute_response()` specific to NestedModel.

- const **DakotaArray** < DakotaResponse > & **derived_synchronize** ()
  
  portion of `synchronize()` specific to NestedModel.

- const **DakotaList** < DakotaResponse > & **derived_synchronize_nowait** ()
  
  portion of `synchronize_nowait()` specific to NestedModel.

- const DakotaList & **synchronize_nowait_completions** ()
  
  Return completion id’s matching response list from `synchronize_nowait`.

- **DakotaModel** & **subordinate_model** ()
  
  return a reference to the subModel.

- short **derived_master_overload** () const
  
  flag which prevents overloading the master with a multiprocessor evaluation (forwarded to subModel so that UQ portion of OUU can execute in parallel).
- void `derived init communicators` (const DakotalIntArray &message lengths, const int &max_iterator concurrency)
  
  portion of `init communicators()` specific to NestedModel.

- void `free communicators` ()
  
  deallocate communicator partitions for the NestedModel (forwarded to subModel so that UQ portion of OUU can execute in parallel).

- void `serve` ()
  
  Service job requests received from the master. Completes when a termination message is received from `stop servers()`. (forwarded to subModel so that UQ portion of OUU can execute in parallel).

- void `stop servers` ()
  
  Executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (forwarded to subModel so that UQ portion of OUU can execute in parallel).

- int `total eval counter` () const
  
  Return the total evaluation count for the NestedModel; forwarded to optionalInterface if present (placeholder for now).

- int `new eval counter` () const
  
  Return the new evaluation count for the NestedModel; forwarded to optionalInterface if present (placeholder for now).

Private Methods

- void `response mapping` (const DakotaResponse &interface response, const DakotaResponse &sub-iterator response, DakotaResponse &mapped response)
  
  combine the response from the optional interface evaluation with the response from the sub-iteration using the objCoeffs/constrCoeffs mappings to create the total response for the model.

- void `asv mapping` (const DakotalIntArray &mapped asv, DakotalIntArray &interface asv)
  
  define the evaluation requirements for the optional interface (interface asv) from the total model evaluation requirements (mapped asv).

Private Attributes

- DakotaIterator subIterator
  
  the sub-iterator that is executed on every evaluation of this model.

- DakotaModel subModel
  
  the sub-model used in sub-iterator evaluations.

- size_t `numSubIteratorIneqConstr`
  
  number of top-level inequality constraints mapped from the sub-iteration results.

- size_t `numSubIteratorEqConstr`
number of top-level equality constraints mapped from the sub-iteration results.

- DakotaInterface optionalInterface
  the optional interface contributes nonnested response data to the total model response.

- DakotaString interfacePointer
  the optional interface pointer from the nested model specification.

- DakotaResponse interfaceResponse
  the response object resulting from optional interface evaluations.

- size_t numInterfObjFs
  number of objective functions resulting from optional interface evaluations.

- size_t numInterfIneqConstr
  number of inequality constraints resulting from optional interface evaluations.

- size_t numInterfEqConstr
  number of equality constraints resulting from the optional interface evaluations.

- DakotaRealMatrix objCoeffs
  "primary" response mapping matrix applied to the sub-iterator response functions. For OUU, the matrix is applied to UQ statistics to create contributions to the top-level objective function(s).

- DakotaRealMatrix constrCoeffs
  "secondary" response mapping matrix applied to the sub-iterator response functions. For OUU, the matrix is applied to UQ statistics to create contributions to the top-level inequality and equality constraints.

- DakotaArray< DakotaResponse > responseArray
  dummy response array for derived_synchronize() prior to derived_async_compute_response() support.

- DakotaList< DakotaResponse > responseList
  dummy response list for derived_synchronize_nowait() prior to derived_async_compute_response() support.

- DakotaIntList completionList
  dummy completion list for synchronize_nowait_completions() prior to derived_async_compute_response() support.

6.59.1 Detailed Description

Derived model class which performs a complete sub-iterator execution within every evaluation of the model.

The NestedModel class nests a sub-iterator execution within every model evaluation. This capability is most commonly used for optimization under uncertainty, in which a nondeterministic iterator is executed on every optimization function evaluation. The NestedModel also contains an optional interface, for portions of the model evaluation which are independent from the sub-iterator, and a set of mappings for combining sub-iterator and optional interface data into a top level response for the model.
6.59.2 Member Function Documentation

6.59.2.1 `void NestedModel::derived::compute_response (const DakotaIntArray & asv)`

[protected, virtual]

portion of `compute_response()` specific to NestedModel.

Update subModel's inactive variables with active variables from currentVariables, compute the optional interface and sub-iterator responses, and map these to the total model response.

Reimplemented from `DakotaModel`.

6.59.2.2 `void NestedModel::derived::asynch_compute_response (const DakotaIntArray & asv)`

[protected, virtual]

portion of `asynch_compute_response()` specific to NestedModel.

Not currently supported by NestedModels (need to add concurrent iterator support). As a result, `derived::synchronize()`, `derived::synchronize_nowait()`, and `synchronize_nowait_completions()` are inactive as well.

Reimplemented from `DakotaModel`.

6.59.2.3 `const DakotaArray< DakotaResponse > & NestedModel::derived::synchronize ()`

[protected, virtual]

portion of `synchronize()` specific to NestedModel.

Asynchronous response computations are not currently supported by NestedModels. Return a dummy responseArray to satisfy the compiler.

Reimplemented from `DakotaModel`.

6.59.2.4 `const DakotaList< DakotaResponse > & NestedModel::derived::synchronize_nowait ()`

[protected, virtual]

portion of `synchronize_nowait()` specific to NestedModel.

Asynchronous response computations are not currently supported by NestedModels. Return a dummy responseList to satisfy the compiler.

Reimplemented from `DakotaModel`.

6.59.2.5 `const DakotaIntList & NestedModel::synchronize_nowait_completions () [inline, protected, virtual]`

Return completion id's matching response list from synchronize_nowait.

Asynchronous response computations are not currently supported by NestedModels. Return a dummy completionList to satisfy the compiler.

Reimplemented from `DakotaModel`. 
6.59.2.6  void NestedModel::derived_init_communicators (const DakotaIntArray &
message_lengths, const int & max_iterator_concurrency)  [inline, protected,
virtual]

portion of init_communicators() specific to NestedModel.

Asynchronous flags need to be initialized for the subModel. In addition, max_iterator_concurrency is the
outer level iterator concurrency, not the subIterator concurrency that subModel will see, and recomputing
the message_lengths on the subModel is probably not a bad idea either. Therefore, recompute everything
on subModel using init_communicators.

Reimplemented from DakotaModel.

6.59.2.7  void NestedModel::response_mapping (const DakotaResponse & interface_response,
const DakotaResponse & sub_iterator_response, DakotaResponse & mapped_response)
[private]

combine the response from the optional interface evaluation with the response from the sub-iteration using
the objCoeffs/constrCoeffs mappings to create the total response for the model.

In the OUU case,

optionalInterface functions = \{f\}, \{g\} (deterministic objectives & constraints)
subIterator functions = \{S\} (UQ response statistics)

Problem formulation for mapped functions:
minimize \{f\} + [W]{S}
subject to \{g\}_l <= \{g\} <= \{g\}_u
\{a\}_l <= [A]{S} <= \{a\}_u
\{g\} == \{g_t\}
[A]{S} == \{a_t\}

where [W] is the primary_mapping_matrix user input (objCoeffs class attribute), [A] is the secondary
mapping_matrix user input (constrCoeffs class attribute), \{{g}\}_l,\{{a}\}_l are the top level inequality
constraint lower bounds, \{{g}_u,\{a}_u\} are the top level inequality constraint upper bounds, and \{{g_t}\}_l,\{a_t\}_l are the top level equality constraint targets.

NOTE: optionalInterface/subIterator objectives overlap but optionalInterface/subIterator constraints do not.
The [W] matrix can be specified so as to allow

- some purely deterministic objective functions and some combined: [W] filled and [W].num_rows() < \{f\}.length() [combined first] or [W].num_rows() == \{f\}.length() and [W] contains rows of zeros
  [combined last]
- some combined and some purely stochastic objective functions: [W] filled and [W].num_rows() >
  \{f\}.length()
- separate deterministic and stochastic objective functions: [W].num_rows() > \{f\}.length() and [W]
  contains \{f\}.length() rows of zeros.

If the need arises, could change constraint definition to allow overlap as well: \{g\}_l <= \{g\} + [A]{S} <=
\{g\}_u with [A] usage the same as for [W] above.

In the UOO case, things are simpler, just compute statistics of each optimization response function: [W] =
[I]. \{f\}/\{g\}/[A] are empty.

6.59.3  Member Data Documentation
6.59.3.1 DakotaModel NestedModel::subModel [private]

the sub-model used in sub-iterator evaluations.

There are no restrictions on subModel, so arbitrary nestings are possible. This is commonly used to support surrogate-based optimization under uncertainty by having NestedModels contain LayeredModels and vice versa.

The documentation for this class was generated from the following files:

- NestedModel.H
- NestedModel.C
6.60 NoDBBaseConstructor Struct Reference

Dummy struct for overloading constructors used in on-the-fly instantiations.

```cpp
#include <ProblemDescDB.H>
```

### Public Methods

- **NoDBBaseConstructor** (int=0)
  
  *C++ structs can have constructors.*

#### 6.60.1 Detailed Description

Dummy struct for overloading constructors used in on-the-fly instantiations.

NoDBBaseConstructor is used to overload the constructor used for on-the-fly iterator instantiations in which ProblemDescDB queries cannot be used. Putting this struct here (rather than in a header of a class that uses it) avoids problems with circular dependencies.

The documentation for this struct was generated from the following file:

- ProblemDescDB.H
6.61 NonDAdvMeanValue Class Reference

Class for the analytical reliability methods within DAKOTA/UQ.

```cpp
#include <NonDAdvMeanValue.H>
```

Inheritance diagram for NonDAdvMeanValue:

```
DakotaIterator  
    ↓           
DakotaNonD  
    ↓           
NonDAdvMeanValue
```

Public Methods

- **NonDAdvMeanValue (DakotaModel &model)**
  - `constructor`.

- **~NonDAdvMeanValue ()**
  - `destructor`.

- void **quantify_uncertainty ()**
  - `performs an uncertainty propagation using analytical reliability methods which solve constrained optimization problems to obtain approximations of the cumulative distribution function of response.`

- void **print_iterator_results (ostream &s) const**
  - `print the approximate mean, standard deviation, and importance factors when using the mean value method (MV) or the CDF information when using other reliability methods (AMV, AMV+, FORM).`

Static Private Methods

- void **lin_approx_objective_eval (int &mode, int &n, Real *u, Real &f, Real *gradf, int &)**
  - `static function used by NPSOL as the objective function in the constrained optimization problems solved in the analytical reliability methods.`

- void **lin_approx_constraint_eval (int &mode, int &ncnl, int &n, int &nrowj, int *needc, Real *u, Real *c, Real *jac, int &nstate)**
  - `static function used by NPSOL as the constraint function in the constrained optimization problems solved in the analytical reliability methods.`

- void **transUTOX (const Epetra_SerialDenseVector &uncorr_normal_vars, Epetra_SerialDenseVector &random_vars)**
  - `Transformation Routine from u-space of random variables to x-space of random variables for Petra data types.`
- void transUToX (const DakotaRealVector &uncorr_normal_vars, DakotaRealVector &random_vars)
  
  **Transformation Routine from u-space of random variables to x-space of random variables for DakotaReal-Vector data types.**

- void transXToU (const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseVector &uncorr_normal_vars)
  
  **Transformation Routine from x-space of random variables to u-space of random variables for Petra data types.**

- void transXToZ (const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseVector &correlated_normal_vars)
  
  **Transformation Routine from x-space of random variables to z-space of random variables for Petra data types.**

- void transUToZ (const Epetra_SerialDenseVector &uncorr_normal_vars, Epetra_SerialDenseVector &correlated_normal_vars)
  
  **Transformation Routine from u-space of random variables to z-space of random variables for Petra data types.**

- void transZToU (Epetra_SerialDenseVector &correlated_normal_vars, Epetra_SerialDenseVector &random_vars)
  
  **Transformation Routine from z-space of random variables to u-space of random variables for Petra data types.**

- void jacXToZ (const Epetra_SerialDenseVector &random_vars, const Epetra_SerialDenseVector &correlated_normal_vars, Epetra_SerialDenseMatrix &jacobianXZ)
  
  **Jacobian of mapping from x to z random variable space.**

- void jacZToX (const Epetra_SerialDenseVector &correlated_normal_vars, const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseMatrix &jacobianZX)
  
  **Jacobian of mapping from z to x random variable space.**

- void jacXToU (const Epetra_SerialDenseVector &random_vars, const Epetra_SerialDenseVector &uncorr_normal_vars, Epetra_SerialDenseMatrix &jacobianXU)
  
  **Jacobian of mapping from x to u random variable space.**

- void jacUToX (const Epetra_SerialDenseVector &uncorr_normal_vars, const Epetra_SerialDenseVector &random_vars, Epetra_SerialDenseMatrix &jacobianUX)
  
  **Jacobian of mapping from u to x random variable space.**

- void transNataf (Epetra_SerialSymDenseMatrix &modcorr_matrix)
  
  **This procedure modifies the correlation matrix input by the user to be used in the Nataf distribution model.**

- void erfInverse (const double &p, double &z)
  
  **Inverse of error function used to invert cdf of normal random variables.**
Private Attributes

- **DakotaString reliabilityMethod**
  
  reliability method identifier specified by user specifies `amvFlag`.

- **DakotaRealArray responseLevelTargets**
  
  user specified targets for response levels.

- **DakotaRealArray probabilityLevelTargets**
  
  user specified targets for probability levels.

- **DakotaRealVector meanResponse**
  
  approximate mean values of response functions predicted by MV.

- **DakotaRealVector stdResponse**
  
  approximate standard deviations of response functions predicted by MV.

- **DakotaRealMatrix impFactor**
  
  importance factors predicted by MV.

Static Private Attributes

- **Epetra_SerialDenseVector staticFnVals**
  
  static copy of `DakotaResponseRep::functionValues`.

- **Epetra_SerialDenseMatrix staticFnGrads**
  
  static copy of `DakotaResponseRep::functionGradients`.

- **Epetra_SerialDenseMatrix staticGlobalGradsX**
  
  Gradient of Response function in x-space for each response level.

- **Epetra_SerialDenseMatrix staticGlobalGradsU**
  
  Gradient of Response function in u-space for each response level.

- **Epetra_SerialSymDenseMatrix petraCorrMatrix**
  
  petra copy of `uncertainCorrMatrix`.

- **Epetra_SerialDenseMatrix cholCorrMatrix**
  
  cholesky factor of `petraCorrMatrix`.

- **Epetra_SerialDenseMatrix mostProbPointX**
  
  Location of MPP in x space.

- **Epetra_SerialDenseMatrix mostProbPointU**
  
  Location of MPP in u space.

- **Epetra_SerialDenseVector ranVarMeans**
  
  Mean Vector of all uncertain random variables.
- Epetra_SerialDenseVector \texttt{ranVarSigm}as
  \textit{Standard Deviation Vector of all uncertain random variables.}

- Epetra_SerialDenseVector \texttt{petraRespLevels}
  \textit{Petra copy of responseLevelTargets specified by user.}

- Epetra_SerialDenseVector \texttt{petraProbLevels}
  \textit{Petra copy of probabilityLevelTargets specified by user.}

- Epetra_SerialDenseVector \texttt{correctedRespLevel}
  \textit{output response levels calculated.}

- int \texttt{respLevelCount}
  \textit{counter for which response level is being analyzed.}

- int \texttt{amvFlag}
  \textit{flag to represent which reliability method is being used.}

- size\_t \texttt{staticNumUncVars}
  \textit{static copy of numUncertainVars.}

- size\_t \texttt{staticNumFuncs}
  \textit{static copy of numFunctions.}

- DakotaIntVector \texttt{ranVarType}
  \textit{vector of indices indicating which type of uncertain variable.}

- DakotaRealVector \texttt{medianFnVals}
  \textit{vector of median values of functions used to determine which side of probability equal 0.5 the response level is.}

- DakotaRealVector \texttt{probLevels}
  \textit{computed probability values.}

\subsection{6.61.1 Detailed Description}

Class for the analytical reliability methods within DAKOTA/UQ.

The \texttt{NonDAdvMeanValue} class implements the following analytic reliability methods: advanced mean value method (AMV), iterated advanced mean value method (AMV+), first order reliability method (FORM), and second order reliability method (SORM). Each of these employ an optimizer (currently NPSOL) to perform a search for the most probable point (MPP).

\subsection{6.61.2 Member Function Documentation}
void NonDAdvMeanValue::lin_approx_objective_eval (int & mode, int & n, Real * u, Real & f, Real * gradf, int &)

static function used by NPSOL as the objective function in the constrained optimization problems solved in the analytical reliability methods.

Need to be static so that they can be passed in function pointers without having to restrict the recipient to functions from the NonDAdvMeanValue class (see Stroustrup, p.166 - pointers to member functions must use class scope operators which would restrict the generality of the NPSOLOptimizer "user functions" interface).

void NonDAdvMeanValue::lin_approx_constraint_eval (int & mode, int & ncnln, int & n, int & nrowj, int * needc, Real * u, Real * c, Real * jac, int & nstate)

static function used by NPSOL as the constraint function in the constrained optimization problems solved in the analytical reliability methods.

Need to be static so that they can be passed in function pointers without having to restrict the recipient to functions from the NonDAdvMeanValue class (see Stroustrup, p.166 - pointers to member functions must use class scope operators which would restrict the generality of the NPSOLOptimizer "user functions" interface).

void NonDAdvMeanValue::transUToX (const Epetra_SerialDenseVector & uncorr_normal_vars, Epetra_SerialDenseVector & random_vars)

Transformation Routine from u-space of random variables to x-space of random variables for Petra data types.

This procedure performs the transformation from u to x space

uncorr_normal_vars is the vector of random variables in standard normal space (u-space).

random_vars is the vector of the random variables in the user-defined x-space

void NonDAdvMeanValue::transXToU (const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseVector & uncorr_normal_vars)

Transformation Routine from x-space of random variables to u-space of random variables for Petra data types.

This procedure performs the transformation from x to u space

uncorr_normal_vars is the vector of random variables in standard normal space (u-space).

random_vars is the vector of the random variables in the user-defined x-space

void NonDAdvMeanValue::transXToZ (const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseVector & correlated_normal_vars)

Transformation Routine from x-space of random variables to z-space of random variables for Petra data types.

This procedure performs the transformation from x to z space

correlated_normal_vars is the vector of random variables in normal space with proper correlations(z-space).
random_vars is the vector of the random variables in the user-defined x-space

6.61.2.6 void NonDAdvMeanValue::transUToZ (const Epetra_SerialDenseVector & uncorr_normal_vars, Epetra_SerialDenseVector & correlated_normal_vars) [static, private]

Transformation Routine from u-space of random variables to z-space of random variables for Petra data types.
This procedure computes the transformation from u to z space.
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).

6.61.2.7 void NonDAdvMeanValue::transZToU (Epetra_SerialDenseVector & correlated_normal_vars, Epetra_SerialDenseVector & uncorr_normal_vars) [static, private]

Transformation Routine from z-space of random variables to u-space of random variables for Petra data types.
This procedure computes the transformation from z to u space.
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).

6.61.2.8 void NonDAdvMeanValue::jacXToZ (const Epetra_SerialDenseVector & random_vars, const Epetra_SerialDenseVector & correlated_normal_vars, Epetra_SerialDenseMatrix & jacobianXZ) [static, private]

Jacobian of mapping from x to z random variable space.
This procedure computes the jacobian of the transformation from x to z space.
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).
random_vars is the vector of the random variables in the user-defined x-space.

6.61.2.9 void NonDAdvMeanValue::jacZToX (const Epetra_SerialDenseVector & correlated_normal_vars, const Epetra_SerialDenseVector & random_vars, Epetra_SerialDenseMatrix & jacobianXZ) [static, private]

Jacobian of mapping from z to x random variable space.
This procedure computes the jacobian of the transformation from z to x space.
correlated_normal_vars is the vector of random variables in normal space with proper correlations (z-space).
random_vars is the vector of the random variables in the user-defined x-space.
6.61.2.10  void NonDAdvMeanValue::jacXToU (const Epetra_SerialDenseVector & random-vars,  
const Epetra_SerialDenseVector & uncorr_normal-vars, Epetra_SerialDenseMatrix &  
jacobianXU)  [static, private]

Jacobian of mapping from x to u random variable space.  
This procedure computes the jacobian of the transformation from x to u space.  
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).  
random_vars is the vector of the random variables in the user-defined x-space

6.61.2.11  void NonDAdvMeanValue::jacUToX (const Epetra_SerialDenseVector  
& uncorr_normal_vars, const Epetra_SerialDenseVector & random_vars,  
Epetra_SerialDenseMatrix & jacobianUX)  [static, private]

Jacobian of mapping from u to x random variable space.  
This procedure computes the jacobian of the transformation from u to x space.  
uncorr_normal_vars is the vector of random variables in standard normal space (u-space).  
random_vars is the vector of the random variables in the user-defined x-space

6.61.2.12  void NonDAdvMeanValue::transNataf (Epetra_SerialSymDenseMatrix &  
mod_corr_matrix)  [static, private]

This procedure modifies the correlation matrix input by the user to be used in the Nataf distribution model.  
This procedure modifies the correlation matrix input by the user to be used in the Nataf distribution model  
(der Kiureghian and Liu, ASCE JEM 112:1, 1986).  
R: the correlation coefficient matrix of the random variables  
mod_corr_matrix: modified correlation matrix  
Note: The modification is exact for log-log, normal-log, normal-normal, normal-uniform transformations  
(numerical precision)  
The uniform-uniform and uniform-log case are approximations obtained in the above reference.

6.61.3  Member Data Documentation

6.61.3.1  Epetra_SerialDenseVector NonDAdvMeanValue::correctedRespLevel  [static,  
private]

output response levels calculated.  
identical to responseLevelTargets for AMV+, FORM but will differ for AMV  
The documentation for this class was generated from the following files:

- NonDAdvMeanValue.H  
- NonDAdvMeanValue.C
6.62 NonDOptStrategy Class Reference

Strategy for optimization under uncertainty (robust and reliability-based design).

```c
#include <NonDOptStrategy.H>
```

Inheritance diagram for NonDOptStrategy:

```
DakotaStrategy
  |-- NonDOptStrategy
```

Public Methods

- **NonDOptStrategy (ProblemDescDB &problem_db)**
  
  constructor.

- **~NonDOptStrategy ()**
  
  destructor.

- **void run_strategy ()**
  
  Perform the strategy by executing optIterator (an optimizer) on designModel (a layered or nested model containing a nondeterministic iterator at a lower level).

Private Attributes

- **DakotaModel designModel**
  
  the nested or layered model interfaced with optIterator.

- **DakotaIterator optIterator**
  
  the top level optimizer.

6.62.1 Detailed Description

Strategy for optimization under uncertainty (robust and reliability-based design).

This strategy uses a NestedModel to nest an uncertainty quantification iterator within an optimization iterator in order to perform optimization using nondeterministic data. For OUU based on surrogates, LayeredModels are also employed, and the general recursion facilities supported by nested and layered models allow a broad array of OUU formulations. This class is very simple and is essentially identical to SingleMethodStrategy since all of the nested iteration mappings are contained within NestedModel::response::mapping().

The documentation for this class was generated from the following files:
- NonDOptStrategy.H
- NonDOptStrategy.C
6.63 NonDPCE Class Reference

Stochastic finite element approach to uncertainty quantification using polynomial chaos expansions.

```
#include <NonDPCE.H>
```

Inheritance diagram for NonDPCE:

```
DakotaIterator

DakotaNonD

NonDPCE
```

Public Methods

- `NonDPCE(DakotaModel &model)`
  
  constructor.

- `~NonDPCE()`
  
  destructor.

- `void quantify_uncertainty()`
  
  virtual function to perform uncertainty quantification using SFEM/PCE methods outputs coefficients of polynomial chaos expansions.

- `void print_iterator_results(ostream &s) const`
  
  print the final statistics and PCE coefficient array.

Static Public Attributes

- `DakotaRealVectorArray coeffArray`
  
  Array containing Polynomial Chaos coefficients.

Private Methods

- `void run_lhs()`
  
  generates the desired set of parameter samples from within user-specified probabilistic distributions.
Private Attributes

- **LatinHypercube * lhsSampler**
  
  pointer to the LatinHypercube object (responsible for generating the parameter samples).

- **DakotaRealVectorArray paramSamples**
  
  the set of parameter samples output from LHS. The double* array from Fortran (arranged head to tail by parameter: all observations for var 1 followed by all observations for var 2, etc.) is converted to this array of DakotaRealVectors.

- **DakotaRealVectorArray responseFnSamples**
  
  the response data (fn values only) for the parameter samples arranged as an array of DakotaRealVectors. This parallels paramSamples so that either may be used in vector_stats().

- **int numObservations**
  
  the number of samples.

- **int randomSeed**
  
  the random number seed.

- **DakotaString sampleType**
  
  the sample type: "lhs" or "random".

- **DakotaRealArray respThresh**
  
  response thresholds for computing failure probabilities.

- **short statsFlag**
  
  flags computation/output of statistics.

- **short allDataFlag**
  
  flags update of allVariables/allResponses.

- **size_t numActiveVars**
  
  total number of variables published to LHS.

- **int numX**
  
  Number of Xi's in Polynomial Chaos Expansion.

- **int highestOrder**
  
  Highest order of Hermite Polynomials in Expansion.

- **int numChaos**
  
  Number of terms in Polynomial Chaos Expansion.

### 6.63.1 Detailed Description

Stochastic finite element approach to uncertainty quantification using polynomial chaos expansions.
The NonDPCE class uses a polynomial chaos expansion (PCE) approach to approximate the effect of parameter uncertainties on response functions of interest. It utilizes the HermiteSurf and HermiteChaos classes to perform the PCE.

The documentation for this class was generated from the following files:

- NonDPCE.H
- NonDPCE.C
6.64 NonDProbability Class Reference

Wrapper class for the LHS library.

```cpp
#include <NonDProbability.H>
```

Inheritance diagram for NonDProbability:

![Inheritance Diagram](image)

Public Methods

- **NonDProbability (DakotaModel &model)**
  
  constructor.

- **NonDProbability (DakotaModel &model, int samples, int seed, const DakotaString &sampling_method, const short &opt_flag)**
  
  alternate constructor for instantiations "on the fly".

- **~NonDProbability ()**
  
  destructor.

- void **quantify_uncertainty ()**
  
  performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results.

- void **print_iterator_results (ostream &s) const**
  
  print the final statistics.

- void **sampling_reset (int min_samples, short all_data_flag, short stats_flag)**
  
  resets number of samples and sampling flags.

- const DakotaString & **sampling_scheme () const**
  
  return sampleType: "lhs" or "random".

Private Methods

- void **run_lhs ()**
  
  generates the desired set of parameter samples from within user-specified probabilistic distributions.
Private Attributes

- **LatinHypercube** * `lhsSampler`
  
  pointer to the LatinHypercube object (responsible for generating the parameter samples).

- **DakotaRealVectorArray** `paramSamples`
  the set of parameter samples output from LHS. The double* array from Fortran (arranged head to tail by parameter: all observations for var 1 followed by all observations for var 2, etc.) is converted to this array of DakotaRealVectors.

- **DakotaRealVectorArray** `responseFnSamples`
  the response data (fn values only) for the parameter samples arranged as an array of DakotaRealVectors. This parallels `paramSamples` so that either may be used in `vector::statistics()`.

- **int** `numObservations`
  the number of samples to evaluate.

- **int** `randomSeed`
  the random number seed.

- **DakotaString** `sampleType`
  the sample type: "lhs" or "random".

- **DakotaRealArray** `respThresh`
  response thresholds for computing failure probabilities.

- **short** `allVarsFlag`
  flags DACE mode using all variables.

- **short** `statsFlag`
  flags computation/output of statistics.

- **short** `allDataFlag`
  flags update of allVariables/allResponses.

- **size_t** `numActiveVars`
  total number of variables published to LHS.

- **size_t** `numDesignVars`
  number of design variables (treated as uniform distribution within design variable bounds for DACE usage of NonDProbability).

- **size_t** `numStateVars`
  number of state variables (treated as uniform distribution within state variable bounds for DACE usage of NonDProbability).
6.64 NonDProbability Class Reference

6.64.1 Detailed Description

Wrapper class for the LHS library.

The Latin Hypercube Sampling (LHS) package from Sandia Albuquerque’s Risk and Reliability organization provides comprehensive capabilities for Monte Carlo and Latin Hypercube sampling within a broad array of user-specified probabilistic parameter distributions. It enforces user-specified correlations through use of a mixing routine. The NonDProbability class provides a C++ wrapper for the LHS library and is used for performing forward propagations of parameter uncertainties into response statistics. The current LHS version in use dates back to a 1970’s vintage Fortran version that was converted to C using f2c and then recast as C++ classes. These classes appear in the LatinHypercube (main LHS class which generates a set of samples from parameter distributions), LHSInput (random variable and user input classes), and LHSVecMatUtil (vector/matrix utilities) files. These files are undocumented as this version is due to be replaced with the 1998 Fortran LHS version shortly.

6.64.2 Constructor & Destructor Documentation

6.64.2.1 NonDProbability::NonDProbability (DakotaModel & model)

constructor.

This constructor is called for a standard letter-envelope iterator instantiation. In this case, set dbList nodes has been called and probDescDB can be queried for settings from the method specification.

6.64.2.2 NonDProbability::NonDProbability (DakotaModel & model, int samples, int seed, const DakotaString & sampling_method, const short & opt_flag)

alternate constructor for instantiations "on the fly".

This alternate constructor (currently inactive) is used by ApproximationInterface for uniform sampling. It is not a letter-envelope instantiation and a set dbList nodes has not been performed. It is called with data from the approximation interface specification (not the method specification) passed through the constructor. This works because NoDBBaseConstructor is used and all of the relevant data for this NonDProbability usage can be taken from the incoming model. Thus, the following attributes are not initialized and should not be used: maxIterations, maxFunctionEvals, verboseOutput, methodName.

6.64.3 Member Function Documentation

6.64.3.1 void NonDProbability::quantify_uncertainty () [virtual]

performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results.

Loop over set of samples and compute responses. paramSamples contains an array ordered with all observations for parameter 1 followed by all observations for parameter 2, etc. After each response is computed, the value returned in current_response is compared to the threshold value, respThresh. The counter, less than thresh, is then incremented if the returned value is < respThresh.
Reimplemented from DakotaNonD.

6.64.3.2 void NonDProbability::sampling_reset (int min_samples, short all_data_flag, short stats_flag) [inline, virtual]

resets number of samples and sampling flags.

used by ApproximationInterface::build_global_approximation() to publish the minimum number of samples needed from the sampling routine (to build a particular global approximation) and to set allDataFlag and statsFlag. In this case, allDataFlag is set to true (vectors of variable and response sets must be returned to build the global approximation) and statsFlag is set to false (statistics computations are not needed).

Reimplemented from Dakotalterator.

The documentation for this class was generated from the following files:

- NonDProbability.H
- NonDProbability.C
6.65 NonDSampling Class Reference

Wrapper class for the Fortran 90 LHS library.

#include <NonDSampling.H>

Inheritance diagram for NonDSampling:

```
DakotaIterator
  \downarrow
DakotaNonD
  \downarrow
NonDSampling
```

Public Methods

- **NonDSampling** (DakotaModel &model)
  
  constructor.

- **NonDSampling** (DakotaModel &model, int samples, int seed, const DakotaString &sampling_method, const short &opt_flag)

  alternate constructor for instantiations "on the fly".

- ~NonDSampling ()

  destructor.

- **void quantify_uncertainty** ()

  performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results.

- **void print_iterator_results** (ostream &s) const

  print the final statistics.

- **void sampling_reset** (int min_samples, short all_data_flag, short stats_flag)

  resets number of samples and sampling flags.

- **const DakotaString & sampling_scheme** () const

  return sampleType: "lhs" or "random".

Private Methods

- **void run_lhs** ()

  generates the desired set of parameter samples from within user-specified probabilistic distributions.
void check_error (const int &err_code, const char *err_source) const

checks the return codes from LHS routines and aborts if an error is returned.

Private Attributes

- DakotaRealVectorArray paramSamples
  the set of parameter samples output from LHS. The double* array from Fortran (arranged head to tail by parameter: all observations for var 1 followed by all observations for var 2, etc.) is converted to this array of DakotaRealVectors.

- DakotaRealVectorArray responseFnSamples
  the response data (fn values only) for the parameter samples arranged as an array of DakotaRealVectors. This parallels paramSamples so that either may be used in vector_statistics().

- int numObservations
  the number of samples to evaluate.

- int randomSeed
  the random number seed.

- DakotaString sampleType
  the sample type: "lhs" or "random".

- DakotaRealArray respThresh
  response thresholds for computing failure probabilities.

- short allVarsFlag
  flags DACE mode using all variables.

- short statsFlag
  flags computation/output of statistics.

- short allDataFlag
  flags update of allVariables/allResponses.

- size_t numActiveVars
  total number of variables published to LHS.

- size_t numDesignVars
  number of design variables (treated as uniform distribution within design variable bounds for DACE usage of NonDSampling).

- size_t numStateVars
  number of state variables (treated as uniform distribution within state variable bounds for DACE usage of NonDSampling).
6.65 NonDSampling Class Reference

6.65.1 Detailed Description

Wrapper class for the Fortran 90 LHS library.

The Latin Hypercube Sampling (LHS) package from Sandia Albuquerque’s Risk and Reliability organization provides comprehensive capabilities for Monte Carlo and Latin Hypercube sampling within a broad array of user-specified probabilistic parameter distributions. It enforces user-specified rank correlations through use of a mixing routine. The NonDSampling class provides a C++ wrapper for the LHS library and is used for performing forward propagations of parameter uncertainties into response statistics. The current LHS version reflects the 1998 Fortran 90 LHS version (as documented in SAND98-0210), which was converted to a UNIX link library in 2001. Thus, the NonDSampling class supercedes NonDProbability, which used a 1970’s vintage LHS that had been f2c’d and converted to (incomplete) classes.

6.65.2 Constructor & Destructor Documentation

6.65.2.1 NonDSampling::NonDSampling (DakotaModel & model)

constructor.

This constructor is called for a standard letter-envelope iterator instantiation. In this case, setDBlistNodes has been called and probDescDB can be queried for settings from the method specification.

6.65.2.2 NonDSampling::NonDSampling (DakotaModel & model, int samples, int seed, const DakotaString & sampling_method, const short & opt_flag)

alternate constructor for instantiations "on the fly".

This alternate constructor (currently inactive) is used by ApproximationInterface for uniform sampling. It is a letter-envelope instantiation and a setDBlistNodes has not been performed. It is called with data from the approximation interface specification (not the method specification) passed through the constructor. This works because NoDBBaseConstructor is used and all of the relevant data for this NonDSampling usage can be taken from the incoming model. Thus, the following attributes are not initialized and should not be used: maxIterations, maxFunctionEvals, verboseOutput, methodName.

6.65.3 Member Function Documentation

6.65.3.1 void NonDSampling::quantifyUncertainty () [virtual]

performs a forward uncertainty propagation by using LHS to generate a set of parameter samples, performing function evaluations on these parameter samples, and computing statistics on the ensemble of results.

Loop over set of samples and compute responses. paramSamples contains an array ordered with all observations for parameter 1 followed by all observations for parameter 2, etc. After each response is computed, the value returned in currentResponse is compared to the threshold value, respThresh. The counter, lessThanRespThresh, is then incremented if the returned value is < respThresh.

Reimplemented from DakotaNonD.
resets number of samples and sampling flags.

used by ApproximationInterface::build_global_approximation() to publish the minimum number of samples needed from the sampling routine (to build a particular global approximation) and to set allDataFlag and statsFlag. In this case, allDataFlag is set to true (vectors of variable and response sets must be returned to build the global approximation) and statsFlag is set to false (statistics computations are not needed).

Reimplemented from DakotaIterator.

The documentation for this class was generated from the following files:

- NonDSampling.H
- NonDSampling.C
6.66 NPSOLOptimizer Class Reference

Wrapper class for the NPSOL optimization library.

```c
#include "NPSOLOptimizer.H"
```

Inheritance diagram for NPSOLOptimizer:

```
DakotaIterator

DakotaOptimizer

NPSOLOptimizer
```

Public Methods

- **NPSOLOptimizer (DakotaModel &model)**
  - *standard constructor.*

- **NPSOLOptimizer (const DakotaRealVector &initial_point, const DakotaRealVector &var_lower_bnds, const DakotaRealVector &var_upper_bnds, int num_lin_ineq, int num_lin_eq, int num_lnl_ineq, int num_lnl_eq, const DakotaRealMatrix &lin_ineq_coeffs, const DakotaRealVector &lin_ineq_lower_bnds, const DakotaRealVector &lin_ineq_upper_bnds, const DakotaRealMatrix &lin_eq_coeffs, const DakotaRealVector &lin_eq_targets, const DakotaRealVector &nonlin_ineq_lower_bnds, const DakotaRealVector &nonlin_ineq_upper_bnds, const DakotaRealVector &nonlin_eq_targets, void(*user_obj_eval)(int &, int &, Real *, Real &, Real *, int &), void(*user_con_eval)(int &), int derivative_level)**
  - *special constructor for instantiations "on the fly".*

- **~NPSOLOptimizer ()**
  - *destructor.*

- **void find_optimum ()**
  - *Used within the optimizer branch for computing the optimal solution. Redefines the run_iterator virtual function for the optimizer branch.*

Private Methods

- **void find_optimum_on_model ()**
  - *called by find_optimum for setUpType == "model".*

- **void find_optimum_on_user_functions ()**
  - *called by find_optimum for setUpType == "user_functions".*
void allocate_workspace ()

Allocates workspace for the optimizer. Private method for the NPSOLOptimizer constructors.

void augment_bounds (DakotaRealVector &augmented_lower_bnds, DakotaRealVector &augmented_upper_bnds)

augments variable bounds with linear and nonlinear constraint bounds.

**Static Private Methods**

void objective_feval (int &mode, int &num_parameters, Real *x, Real &f, Real *g, int &state)

OBJFUN in NPSOL manual: computes the value and first derivatives of the objective function (passed by function pointer to NPSOL).

void constraint_feval (int &mode, int &ncnln, int &n, int &nrwj, int *needc, Real *x, Real *c, Real *cjac, int &nstate)

CONFUN in NPSOL manual: computes the values and first derivatives of the nonlinear constraint functions (passed by function pointer to NPSOL).

**Private Attributes**

int linConstraintMatrixSize

used for non-zero array sizing (linear constraints).

DakotaRealArray cLambda

CLAMBD A from NPSOL manual: Langrange multipliers.

DakotaIntArray constraintState

IST A TE from NPSOL manual: constraint status.

int informResult

INFORM from NPSOL manual: optimization status on exit.

int numberIterations

ITER from NPSOL manual: number of (major) iterations performed.

int boundsArraySize

length of augmented bounds arrays (variable bounds plus linear and nonlinear constraint bounds).

double * linConstraintMatrixF77


double * upperFactorHessianF77


double * constraintJacMatrixF77

6.66 DakotaString setUpType

controls iteration mode: "model" (normal usage) or "user functions" (user-supplied functions mode for "on the fly" instantiations). NonDAdvMeanValue currently uses the user functions mode.

- DakotaRealVector initialPoint
  
holds initial point passed in for "user functions" mode.

- DakotaRealVector lowerBounds
  
holds variable lower bounds passed in for "user functions" mode.

- DakotaRealVector upperBounds
  
holds variable upper bounds passed in for "user functions" mode.

- void(* userObjectiveEval)(int &, int &, Real *, Real &, Real *, int &)
  
holds function pointer for objective function evaluator passed in for "user functions" mode.

- void(* userConstraintEval)(int &, int &, int &, int &, int *, Real *, Real *, Real *, Real *, int &)
  
holds function pointer for constraint function evaluator passed in for "user functions" mode.

Static Private Attributes

- int fnEvalCntr
  
  counter for testing against staticMaxFnEvals.

- int staticMaxFnEvals
  
  static copy of DakotaIterator::maxFunctionEvals.

- int staticVendorNumericalGradFlag
  
  static copy of DakotaOptimizer::vendorNumericalGradFlag.

6.66.1 Detailed Description

Wrapper class for the NPSOL optimization library.

The NPSOLOptimizer class provides a wrapper for NPSOL, a Fortran 77 sequential quadratic programming library from Stanford University marketed by Stanford Business Associates. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or static as well. To isolate the effect of these static requirements from the rest of the iterator hierarchy, static copies are made of many non-static attributes inherited from above.

The user input mappings are as follows: max_function_evaluations is implemented directly in NPSOLOptimizer’s evaluator functions since there is no NPSOL parameter equivalent, and max_iterations, convergence_tolerance, output verbosity, verify level, function precision, and linesearch tolerance are mapped into NPSOL’s “Major Iteration Limit”, “Optimality Tolerance”, “Major Print Level” (verbose: Major Print Level = 20; quiet: Major Print Level = 10), “Verify Level”, “Function Precision”, and “Linesearch Tolerance” parameters, respectively, using NPSOL’s npoptn() subroutine (as wrapped by npoptn2() from the npoptn wrapper.f file). Refer to [Gill,

The documentation for this class was generated from the following files:

- NPSOLOptimizer.H
- NPSOLOptimizer.C
6.67 ParallelLibrary Class Reference

Class for managing partitioning of multiple levels of parallelism and message passing within the levels.

```
#include <ParallelLibrary.H>
```

Public Methods

- `ParallelLibrary` (int &argc, char **&argv)
  
  constructor.

- `ParallelLibrary()`
  
  default constructor (used only for dummy_lib).

- `~ParallelLibrary()`
  
  destructor.

- `void init_iterator_communicators(const ProblemDescDB &problem_db)`
  
  split MPI_COMM_WORLD into iterator communicators.

- `void init_evaluation_communicators(int eval_servers, int procs_per_eval, int max_concurrency, int asynch_local_eval_concurrency, const DakotaString &eval_scheduling)`
  
  split an iterator communicator into evaluation communicators.

- `void init_analysis_communicators(int analysis_servers, int procs_per_analysis, int max_concurrency, int asynch_local_analysis_concurrency, const DakotaString &analysis_scheduling)`
  
  split an evaluation communicator into analysis communicators.

- `void free_iterator_communicators()`
  
  deallocate iterator communicators.

- `void free_evaluation_communicators()`
  
  deallocate evaluation communicators.

- `void free_analysis_communicators()`
  
  deallocate analysis communicators.

- `void print_configuration()`
  
  print the parallel configuration for all parallelism levels.

- `void send SI(PackBuffer &send_buffer, int dest, int tag)`
  
  blocking send at the strategy-iterator communication level.

- `void isend SI(PackBuffer &send_buffer, int dest, int tag, MPI_Request &send_request)`
  
  nonblocking send at the strategy-iterator communication level.

- `void recv SI(UnPackBuffer &recv_buffer, int source, int tag, MPI_Status &status)`
blocking receive at the strategy-iterator communication level.

- void **irecv** (UnPackBuffer &recv_buffer, int source, int tag, MPI_Request &recv_request)  
  nonblocking receive at the strategy-iterator communication level.

- void **send** (PackBuffer &send_buffer, int dest, int tag)  
  blocking send at the iterator-evaluation communication level.

- void **isend** (PackBuffer &send_buffer, int dest, int tag, MPI_Request &send_request)  
  nonblocking send at the iterator-evaluation communication level.

- void **recv** (UnPackBuffer &recv_buffer, int source, int tag, MPI_Status &status)  
  blocking receive at the iterator-evaluation communication level.

- void **irecv** (UnPackBuffer &recv_buffer, int source, int tag, MPI_Request &recv_request)  
  nonblocking receive at the iterator-evaluation communication level.

- void **send** (int &send_int, int dest, int tag)  
  blocking send at the evaluation-analysis communication level.

- void **isend** (int &send_int, int dest, int tag, MPI_Request &send_request)  
  nonblocking send at the evaluation-analysis communication level.

- void **recv** (int &recv_int, int source, int tag, MPI_Status &status)  
  blocking receive at the evaluation-analysis communication level.

- void **irecv** (int &recv_int, int source, int tag, MPI_Request &recv_request)  
  nonblocking receive at the evaluation-analysis communication level.

- void **bcast** (int &data, MPI_Comm comm)  
  broadcast an integer across a communicator.

- void **bcast** (PackBuffer &send_buffer, MPI_Comm comm)  
  send a packed buffer across a communicator using a broadcast.

- void **bcast** (UnPackBuffer &recv_buffer, MPI_Comm comm)  
  matching receive for a packed buffer broadcast.

- void **waitall** (int num_recv, MPI_Request * &recv_requests)  
  wait for all messages from a series of nonblocking receives.

- int **world_size** () const  
  return worldSize.

- int **world_rank** () const  
  return worldRank.

- short **parallelism_levels** () const  
  return parallelismLevels.
- short `strategy_dedicated_master_flag` () const
  
  `return strategyDedicatedMasterFlag.`

- short `strategy_iterator_split_flag` () const
  
  `return strIteratorSplitFlag.`

- short `iterator_master_flag` () const
  
  `return iteratorMasterFlag.`

- short `strategy_iterator_message_pass` () const
  
  `return strIteratorMessagePass.`

- `MPI_Comm` iterator_intra_communicator () const
  
  `return iteratorIntraComm.`

- `MPI_Comm` `strategy_iterator_inter_communicator` () const
  
  `return strIteratorInterComm.`

- `MPI_Comm` * `strategy_iterator_inter_communicators` () const
  
  `return strIteratorInterComms.`

- int `iterator_servers` () const
  
  `return numIteratorServers.`

- int `iterator_communicator_rank` () const
  
  `return iteratorCommRank.`

- int `iterator_communicator_size` () const
  
  `return iteratorCommSize.`

- int `iterator_server_id` () const
  
  `return iteratorServerId.`

- short `iterator_dedicated_master_flag` () const
  
  `return iteratorDedicatedMasterFlag.`

- short `iterator_eval_split_flag` () const
  
  `return iteratorEvalSplitFlag.`

- short `evaluation_master_flag` () const
  
  `return evalMasterFlag.`

- short `iterator_eval_message_pass` () const
  
  `return iteratorEvalMessagePass.`

- `MPI_Comm` evaluation_intra_communicator () const
  
  `return evalIntraComm.`

- `MPI_Comm` iterator_eval_inter_communicator () const
  
  `return iteratorEvalInterComm.`
- `MPI_Comm * iterator_eval_inter_communicators () const
  return iteratorEvalInterComms.``
- `int evaluation_servers () const
  return numEvalServers.``
- `int evaluation_communicator_rank () const
  return evalCommRank.``
- `int evaluation_communicator_size () const
  return evalCommSize.``
- `int evaluation_server_id () const
  return evalServerId.``
- `short evaluation_dedicated_master_flag () const
  return evalDedicatedMasterFlag.``
- `short eval_analysis_split_flag () const
  return evalAnalysisSplitFlag.``
- `short analysis_master_flag () const
  return analysisMasterFlag.``
- `short eval_analysis_message_pass () const
  return evalAnalysisMessagePass.``
- `MPI_Comm analysis_intra_communicator () const
  return analysisIntraComm.``
- `MPI_Comm eval_analysis_inter_communicator () const
  return evalAnalysisInterComm.``
- `MPI_Comm * eval_analysis_inter_communicators () const
  return evalAnalysisInterComms.``
- `int analysis_servers () const
  return numAnalysisServers.``
- `int analysis_communicator_rank () const
  return analysisCommRank.``
- `int analysis_communicator_size () const
  return analysisCommSize.``
- `int analysis_server_id () const
  return analysisServerId.``
Private Methods

- short `split_communicator.dedicated_master` (MPI_Comm parent_comm, const int &parent_comm_rank, const int &parent_comm_size, const int &num_servers, const int &procs_per_server, const int &proc_remainder, MPI_Comm &child_intra_comm, int &child_comm_rank, int &child_comm_size, MPI_Comm &child_inter_comm, MPI_Comm * &child_inter_comms, int &server_id, short &child_master_flag)
  
  *split a parent communicator into a dedicated master processor and num_servers child communicators.*

- short `split_communicator.peer_partition` (MPI_Comm parent_comm, const int &parent_comm_rank, const int &parent_comm_size, const int &num_servers, const int &procs_per_server, const int &proc_remainder, MPI_Comm &child_intra_comm, int &child_comm_rank, int &child_comm_size, MPI_Comm &child_inter_comm, MPI_Comm * &child_inter_comms, int &peer_id, short &child_master_flag)
  
  *split a parent communicator into num_servers child communicators (no dedicated master processor).*

- short `resolve_inputs` (int &num_servers, int &procs_per_server, const int &avail_procs, int &proc_remainder, const int &max_concurrency, const int &capacity_multiplier, const DakotaString &default_config, const DakotaString &scheduling_override)
  
  *Resolve user inputs into a sensible partitioning scheme.*

Private Attributes

- int `worldSize`

  *size of MPI_COMM_WORLD.*

- int `worldRank`

  *rank in MPI_COMM_WORLD.*

- short `parallelismLevels`

  *number of parallelism levels.*

- short `mpirunFlag`

  *flag for a parallel mpirun/yod launch.*

- short `dummyFlag`

  *prevents multiple MPI_Finalize calls due to dummy_lib.*

- Real `startCPUTime`

  *start reference for UTILIB CPU timer.*

- Real `startWCTime`

  *start reference for UTILIB wall clock timer.*

- Real `startMPITime`

  *start reference for MPI wall clock timer.*

- long `startClock`

  *start reference for local clock() timer measuring parent+child CPU.*
- short `strategyDedicatedMasterFlag`
  signals ded. master partitioning.

- short `stratIteratorSplitFlag`
  signals a communicator split was used.

- short `iteratorMasterFlag`
  identifies master iterator processors.

- short `stratIteratorMessagePass`
  flag for message passing at si level.

- `MPI_Comm` `iteratorIntraComm`
  intracomm for each iterator partition.

- `MPI_Comm` `stratIteratorInterComm`
  intercomm between an iterator & master strategy (on iterator partitions only).

- `MPI_Comm` * `stratIteratorInterComms`
  intercomm. array on master strategy.

- int `numIteratorServers`
  number of iterator servers.

- int `procsPerIterator`
  processors per iterator server.

- int `iteratorCommRank`
  rank in iteratorIntraComm.

- int `iteratorCommSize`
  size of iteratorIntraComm.

- int `iteratorServerId`
  identifier for an iterator server.

- short `iteratorDedicatedMasterFlag`
  signals ded. master partitioning.

- short `iteratorEvalSplitFlag`
  signals a communicator split was used.

- short `evalMasterFlag`
  identifies master evaluation processors.

- short `iteratorEvalMessagePass`
  flag for message passing at ie level.

- `MPI_Comm` `evalIntraComm`
intracomm for each fn. eval. partition.

- **MPI_Comm iteratorEvalInterComm**
  intercomm between a fn. eval. & master iterator (on fn. eval. partitions only).

- **MPI_Comm * iteratorEvalInterComms**
  intercomm array on master iterator.

- int **numEvalServers**
  number of evaluation servers.

- int **procsPerEval**
  processors per evaluation server.

- int **evalCommRank**
  rank in evalIntraComm.

- int **evalCommSize**
  size of evalIntraComm.

- int **evalServerId**
  identifier for an evaluation server.

- short **evalDedicatedMasterFlag**
  signals dedicated master partitioning.

- short **evalAnalysisSplitFlag**
  signals a communicator split was used.

- short **analysisMasterFlag**
  identifies master analysis processors.

- short **evalAnalysisMessagePass**
  flag for message passing at ea level.

- **MPI_Comm analysisIntraComm**
  intracomm for each analysis partition.

- **MPI_Comm evalAnalysisInterComm**
  intercomm between an analysis & master fn. eval. (on analysis partitions only).

- **MPI_Comm * evalAnalysisInterComms**
  intercomm array on master fn. eval.

- int **numAnalysisServers**
  number of analysis servers.

- int **procsPerAnalysis**
  processors per analysis server.
int analysisCommRank
rank in analysisIntraComm.

int analysisCommSize
size of analysisIntraComm.

int analysisServerId
identifier for an analysis server.

### 6.67.1 Detailed Description

Class for managing partitioning of multiple levels of parallelism and message passing within the levels.

The ParallelLibrary class encapsulates all of the details of performing message passing within multiple levels of parallelism. It provides functions for partitioning of levels according to user configuration input and functions for passing messages within and across MPI communicators for each of the parallelism levels. If support for other message-passing libraries beyond MPI becomes needed, then ParallelLibrary should become a class hierarchy with virtual functions to encapsulate the library-specific syntax.

### 6.67.2 Member Function Documentation

#### 6.67.2.1 void ParallelLibrary::init_iterator_communicators (const ProblemDescDB & problem_db)

split MPI_COMM_WORLD into iterator communicators.

Split MPI_COMM_WORLD into the specified number of subcommunicators to set up concurrent iterator partitions serving a strategy. This constructs new iterator intra-communicators and strategy-iterator inter-communicators. The init_iterator_communicators() and free_iterator_communicators() functions are both called from main.C, and init_iterator_communicators() is called prior to output and restart management since output and restart files are tagged based on iterator server id.

#### 6.67.2.2 void ParallelLibrary::init_evaluation_communicators (int eval_servers, int procs_per_eval, int max_concurrency, int async_local_eval_concurrency, const DakotaString & eval_scheduling)

split an iterator communicator into evaluation communicators.

Split iteratorIntraComm (=MPI_COMM_WORLD if no concurrency in iterators) as specified by the passed parameters to set up concurrent evaluation partitions serving an iterator. This constructs new evaluation intra-communicators and iterator-evaluation inter-communicators. init_evaluation_communicators() is called from ApplicationInterface::init_communicators() and free_evaluation_communicators() function is called from ApplicationInterface::free_communicators(). eval_servers, async_local_eval_concurrency, and eval_scheduling come from the interface keyword specification. procs_per_eval is not directly user-specified, rather it contains the minimum procs_per_eval required to support any lower level user requests (such as procs_per_analysis). max_concurrency is passed in via the function DakotalIter::max_concurrency(), which queries individual methods for their gradient configuration, population size, etc. These partitions can be reconfigured for each iterator/model pair within a strategy (e.g. interface 1 uses 4 by 256 while interface 2 uses 2 by 512) – see DakotaStrategy::run_iterator().
6.67.2.3  void ParallelLibrary::init_analysis_communicators (int analysis_servers, int procs_per_analysis, int max_concurrency, int async_local_analysis_concurrency, const DakotaString & analysis_scheduling)

split an evaluation communicator into analysis communicators.

Split evalIntraComm as indicated by the passed parameters to set up concurrent analysis partitions serving a function evaluation. This constructs new analysis intra-communicators and evaluation-analysis inter-communicators. init_analysis_communicators() is called from ApplicationInterface::init_communicators() following the call to init_evaluation_communicators() and free_analysis_communicators() is called from ApplicationInterface::free_communicators() preceding the call to free_evaluation_communicators(). The analysis_servers, procs_per_analysis, async_local_analysis_concurrency, and analysis_scheduling attributes come from the interface keyword specification, and max_concurrency contains the length of analysis_drivers from the interface keyword specification. The analysis partitions can be reconfigured for each iterator/model pair within a strategy.

6.67.2.4  short ParallelLibrary::resolve_inputs (int & num_servers, int & procs_per_server, const int & avail_procs, int & proc_remainder, const int & max_concurrency, const int & capacity_multiplier, const DakotaString & default_config, const DakotaString & scheduling_override) [private]

Resolve user inputs into a sensible partitioning scheme.

This function is responsible for the "auto-configure" intelligence of DAKOTA. It resolves a variety of inputs and overrides into a sensible partitioning configuration for a particular parallelism level. It also handles the general case in which a user’s specification request does not divide out evenly with the number of available processors for the level. If num_servers & procs_per_server are both nondefault, then the former takes precedence.

The documentation for this class was generated from the following files:

- ParallelLibrary.H
- ParallelLibrary.C
6.68  ParamResponsePair Class Reference

Container class for a variables object, a response object, and an evaluation id.

#include <ParamResponsePair.H>

Public Methods

- **ParamResponsePair ()**  
  default constructor.

- **ParamResponsePair (const DakotaVariables &vars, const DakotaResponse &response)**  
  alternate constructor for temporaries.

- **ParamResponsePair (const DakotaVariables &vars, const DakotaResponse &response, const int id)**  
  standard constructor for history uses.

- **ParamResponsePair (const ParamResponsePair &pair)**  
  copy constructor.

- **~ParamResponsePair ()**  
  destructor.

- **ParamResponsePair & operator= (const ParamResponsePair &pair)**  
  assignment operator.

- **void read (istream &s)**  
  read a ParamResponsePair object from an istream.

- **void write (ostream &s) const**  
  write a ParamResponsePair object to an ostream.

- **void read_annotated (istream &s)**  
  read a ParamResponsePair object in annotated format from an istream.

- **void write_annotated (ostream &s) const**  
  write a ParamResponsePair object in annotated format to an ostream.

- **void write_tabular (ostream &s) const**  
  write a ParamResponsePair object in tabular format to an ostream.

- **void read (DakotaBiStream &s)**  
  read a ParamResponsePair object from the binary restart stream.

- **void write (DakotaBoStream &s) const**  
  write a ParamResponsePair object to the binary restart stream.
- **void** read (UnPackBuffer &s)
  
  *read a ParamResponsePair object from a packed MPI buffer.*

- **void** write (PackBuffer &s) const
  
  *write a ParamResponsePair object to a packed MPI buffer.*

- **int** evalId () const
  
  *return the evaluation identifier.*

- **const** DakotaVariables & prp_parameters () const
  
  *return the parameters object.*

- **const** DakotaResponse & prp_response () const
  
  *return the response object.*

- **void** prp_response (const DakotaResponse &response)
  
  *set the response object.*

- **const** DakotaIntArray & active_set_vector () const
  
  *return the active set vector from the response object.*

- **void** active_set_vector (const DakotaIntArray &asv)
  
  *set the active set vector in the response object.*

- **const** DakotaString & interfaceId () const
  
  *return the interface identifier from the response object.*

### Private Attributes

- **DakotaVariables** prPairParameters
  
  *the set of parameters for the function evaluation.*

- **DakotaResponse** prPairResponse
  
  *the response set for the function evaluation.*

- **int** evalId
  
  *the function evaluation identifier (assigned from ApplicationInterface:fnEvalId).*

### Friends

- **int** operator==(const ParamResponsePair &pair1, const ParamResponsePair &pair2)
  
  *equality operator.*
6.68.1 Detailed Description

Container class for a variables object, a response object, and an evaluation id.

ParamResponsePair provides a container class for association of the input for a particular function evaluation (a variables object) with the output from this function evaluation (a response object), along with an evaluation identifier. This container defines the basic unit used in the data pairs list, in restart file operations, and in a variety of scheduling algorithm bookkeeping operations. With the advent of STL, replacement of this class with the pair<> template construct may be possible (using pair<int, pair<vars, response>>, for example), assuming that deep copies, I/O, alternate constructors, etc., can be adequately addressed.

6.68.2 Constructor & Destructor Documentation

6.68.2.1 ParamResponsePair::ParamResponsePair (const DakotaVariables & vars, const DakotaResponse & response) [inline]

alternate constructor for temporaries.

This constructor can use the standard DakotaVariables and DakotaResponse copy constructors to share representations since this constructor is used for search pairs (which are local instantiations that go out of scope prior to any changes to values; i.e., they are not used for history).

6.68.2.2 ParamResponsePair::ParamResponsePair (const DakotaVariables & vars, const DakotaResponse & response, const int id) [inline]

standard constructor for history uses.

This constructor cannot share representations since it involves a history mechanism (beforeSynchPRPList or data pairs). Deep copies must be made.

6.68.3 Member Data Documentation

6.68.3.1 int ParamResponsePair::evalId [private]

the function evaluation identifier (assigned from ApplicationInterface::fnEvalId).

evalId belongs here rather than in DakotaResponse since some DakotaResponse objects involve consolidation of several fn. evals. (e.g., synchronize fd gradients). The prPair, on the other hand, is used for storage of all low level fn. evals. that get evaluated, so evalId is meaningful.

The documentation for this class was generated from the following files:

- ParamResponsePair.H
- ParamResponsePair.C
6.69  ParamStudy Class Reference

Class for vector, list, centered, and multidimensional parameter studies.

```cpp
#include <ParamStudy.H>
```

Inheritance diagram for ParamStudy::

```
DakotaIterator
   |
   v
ParamStudy
```

**Public Methods**

- **ParamStudy** (DakotaModel &model)
  constructor.

- **~ParamStudy** ()
  destructor.

- void **run_iterator** ()
  run the iterator.

- const DakotaVariables & **iterator_variable_results** () const
  return the final iterator solution (variables).

- const DakotaResponse & **iterator_response_results** () const
  return the final iterator solution (response).

- void **print_iterator_results** (ostream &s) const
  print the final iterator results.

**Private Methods**

- void **compute_vector_steps** ()
  computes stepVector and numSteps from initialPoint, finalPoint, and either numSteps or stepLength (pStudy-Type is 1 or 2).

- void **vector_loop** (const DakotaRealVector &start, const DakotaRealVector &step_vect, const int &num_steps)
  performs the parameter study by looping from start in numSteps increments of step_vect. Total number of evaluations is num_steps + 1.

- void **sample** (const DakotaRealVector &list_of_points)
performs the parameter study by sampling from a list of points.

- void `centered_loop` (const DakotaRealVector &start, const Real &percentDelta, const int &deltasPerVariable)
  performs a number of plus and minus offsets for each parameter centered about start.

- void `multidim_loop` (const DakotaIntArray &varPartitions)
  performs vector loops recursively in multiple dimensions.

- void `recurse` (int nloop, int nindex, DakotaIntArray &currentIndex, const DakotaIntArray &maxIndex, const DakotaRealVector &start, const DakotaRealVector &stepVect)
  used by `multidim_loop` to enable a variable number of nested loops.

- void `update_best` (const DakotaRealVector &vars, const DakotaResponse &response, const int evalNum)
  compares current evaluation to best evaluation and updates best.

**Private Attributes**

- DakotaRealVector `listOfPoints`
  list of evaluation points for the list parameter study.

- DakotaRealVector `initialPoint`
  the starting point for vector and centered parameter studies.

- DakotaRealVector `finalPoint`
  the ending point for vector parameter study (a specification option).

- DakotaRealVector `stepVector`
  the n-dimensional increment in vector parameter study.

- int `numSteps`
  the number of times `stepVector` is applied in vector parameter study.

- int `pStudyType`
  internal code for parameter study type: -1 (list), 1, 2, 3 (different vector specifications), 4 (centered), or 5 (multidim).

- int `deltasPerVariable`
  number of offsets in the plus and the minus direction for each variable in a centered parameter study.

- short `nestedFlag`
  flag set by parameter studies which call other parameter studies in loops.

- short `recurseFlag`
  flag set after initial loop in a nested study (so `update_best()` works with `evalNum==0` multiple times).

- Real `stepLength`
  the Cartesian length of multidimensional steps in vector parameter study (a specification option).
- **Real** `percentDelta`  
  size of relative offsets in percent for each variable in a centered parameter study.

- **DakotaIntArray** `variablePartitions`  
  number of partitions for each variable in a multidim parameter study.

- **DakotaVariables** `bestVariables`  
  best variables found during the study.

- **DakotaResponse** `bestResponses`  
  best responses found during the study.

- **Real** `bestObjectiveFn`  
  best objective function found during the study.

- **Real** `bestViolations`  
  best constraint violations found during the study. In the current approach, constraint violation reduction takes strict precedence over objective function reduction.

- **size_t** `numObjectiveFunctions`  
  number of objective functions. Used in `update_best`.

- **size_t** `numNonlinearIneqConstraints`  
  number of nonlinear inequality constraints. Used in `update_best`.

- **size_t** `numNonlinearEqConstraints`  
  number of nonlinear equality constraints. Used in `update_best`.

- **DakotaRealVector** `multiObjWeights`  
  vector of multiobjective weights. Used in `update_best`.

- **DakotaRealVector** `nonlinearIneqLowerBnds`  
  vector of nonlinear inequality constraint lower bounds. Used in `update_best`.

- **DakotaRealVector** `nonlinearIneqUpperBnds`  
  vector of nonlinear inequality constraint upper bounds. Used in `update_best`.

- **DakotaRealVector** `nonlinearEqTargets`  
  vector of nonlinear equality constraint targets. Used in `update_best`.

- **DakotaRealVectorArray** `vectorOfVars`  

- **int** `psCounter`  
  class-scope counter (needed for asynchronous multidim loop).
6.69.1 Detailed Description

Class for vector, list, centered, and multidimensional parameter studies.

The ParamStudy class contains several algorithms for performing parameter studies of different types. It is not a wrapper for an external library, rather its algorithms are self-contained. The vector parameter study steps along an n-dimensional vector from an arbitrary initial point to an arbitrary final point in a specified number of steps. The centered parameter study performs a number of plus and minus offsets in each coordinate direction around a center point. A multidimensional parameter study fills an n-dimensional hypercube based on a specified number of intervals for each dimension. It is a nested study in that it utilizes the vector parameter study internally as it recurses through the variables. And the list parameter study provides for a user specification of a list of points to evaluate, which allows general parameter investigations not fitting the structure of vector, centered, or multidim parameter studies.

6.69.2 Member Function Documentation

6.69.2.1 void ParamStudy::run_iterator () [virtual]

run the iterator.

This function is the primary run function for the iterator class hierarchy. All derived classes need to redefine it.

Reimplemented from Dakotalterator.

The documentation for this class was generated from the following files:

- ParamStudy.H
- ParamStudy.C
6.70 ProblemDescDB Class Reference

The database containing information parsed from the DAKOTA input file.

#include <ProblemDescDB.H>

Public Methods

- **ProblemDescDB** (ParallelLibrary &parallel
  
  constructor.

- **~ProblemDescDB** ()
  
  destructor.

- void **check_input** ()
  
  verifies that there was at least one of each of the required keywords in the dakota input file.

- void **set_db_list_nodes** (const DakotaString &method_tag)
  
  set methodIndeX based on the method identifier string to activate a particular method specification in methodList and use pointers from this method specification to set the other list indices.

- void **set_db_list_nodes** (const int &method_index)
  
  set methodIndeX to activate a particular method specification in methodList and use pointers from this method specification to set the other list indices.

- int **get_db_list_nodes** () const
  
  return the current methodIndeX.

- void **set_db_interface_node** (const DakotaString &interface_tag)
  
  set interfaceIndeX based on the interface identifier string.

- void **set_db_interface_node** (const int &interface_index)
  
  set interfaceIndeX.

- int **get_db_interface_node** () const
  
  return the current interfaceIndeX.

- void **set_db_responses_node** (const DakotaString &responses_tag)
  
  set responsesIndeX based on the responses identifier string.

- void **set_db_model_type** (const DakotaString &model_type)
  
  set the model type.

- void **send_db_buffer** ()
  
  MPI send of a large buffer containing strategy specification attributes and all the objects in interfaceList, variablesList, methodList, and responsesList.
void receive_db_buffer ()

MPI receive of a large buffer containing strategy specification attributes and all the objects in interfaceList, variablesList, methodList, and responsesList.

ParalleLib & parallelLibrary () const

return the parallelLibrary reference.

const DakotaRealVector & get_drv (const DakotaString &entry_name) const
get a DakotaRealVector out of the database based on an identifier string.

const DakotaIntVector & get_div (const DakotaString &entry_name) const
get a DakotaIntVector out of the database based on an identifier string.

const DakotaRealArray & get_dra (const DakotaString &entry_name) const
get a DakotaRealArray out of the database based on an identifier string.

const DakotaIntArray & get_dia (const DakotaString &entry_name) const
get a DakotaIntArray out of the database based on an identifier string.

const DakotaRealMatrix & getdrm (const DakotaString &entry_name) const
get a DakotaRealMatrix out of the database based on an identifier string.

const DakotaIntList & get_dil (const DakotaString &entry_name) const
get a DakotaIntList out of the database based on an identifier string.

const DakotaStringArray & getdsa (const DakotaString &entry_name) const
get a DakotaStringArray out of the database based on an identifier string.

const DakotaStringList & getdsl (const DakotaString &entry_name) const
get a DakotaStringList out of the database based on an identifier string.

const DakotaString & get_string (const DakotaString &entry_name) const
get a DakotaString out of the database based on an identifier string.

const Real & get_real (const DakotaString &entry_name) const
get a Real out of the database based on an identifier string.

const int & get_int (const DakotaString &entry_name) const
get an int out of the database based on an identifier string.

const size_t & get_sizet (const DakotaString &entry_name) const
get a size_t out of the database based on an identifier string.

const short & get_short (const DakotaString &entry_name) const
get a short out of the database based on an identifier string.
**Static Public Methods**

- void `method_kwhandler` (const struct FunctionData *parsed_data)
  
  *method keyword handler called by IDR when a complete method specification is parsed.*

- void `variables_kwhandler` (const struct FunctionData *parsed_data)
  
  *variables keyword handler called by IDR when a complete variables specification is parsed.*

- void `interface_kwhandler` (const struct FunctionData *parsed_data)
  
  *interface keyword handler called by IDR when a complete interface specification is parsed.*

- void `responses_kwhandler` (const struct FunctionData *parsed_data)
  
  *responses keyword handler called by IDR when a complete responses specification is parsed.*

- void `strategy_kwhandler` (const struct FunctionData *parsed_data)
  
  *strategy keyword handler called by IDR when a complete strategy specification is parsed.*

**Private Methods**

- void `set_other_list_nodes` (const int &world_rank, const int &verbose_flag)

  *convenience function used by set_db_list_nodes(method_tag) and set_db_list_nodes(method_index) to set the other list indices once methodIndex is set (based on pointers from the method specification).*

**Static Private Methods**

- void `build_label` (DakotaString &label, const DakotaString &root_label, size_t tag)

  *create a label by appending tag to root_label.*

- void `build_labels` (DakotaStringArray &label_array, const DakotaString &root_label)

  *create an array of labels by tagging root_label with index in label_array. Uses build_label().*

**Private Attributes**

- `ParallelLibrary` & parallelLib

  *reference to the parallel_lib object passed from main.*

**Static Private Attributes**

- `DakotaString strategyType`

  *the strategy selection: multi, surrogate based, opt, opt under uncertainty, branch and bound, multi start, pareto set, or single method.*

- short `strategyGraphicsFlag`

  *flags use of graphics by the strategy (from the graphics specification in StratIndControl).*
- short **strategyTabularDataFlag**
  flags tabular data collection by the strategy (from the *tabular_graphics_data specification in StratIndControl*).

- **DakotaString strategyTabularDataFile**
  the filename used for tabular data collection by the strategy (from the *tabular_graphics_file specification in StratIndControl*).

- int **strategyIteratorServers**
  number of servers for concurrent iterator parallelism (from the *iterator_servers specification in StratIndControl*).

- **DakotaString strategyIteratorScheduling**
  type of scheduling (self or static) used in concurrent iterator parallelism (from the *iterator_self_scheduling and iterator_static_scheduling specifications in StratIndControl*).

- **DakotaString strategyMethodPointer**
  method identifier for the strategy (from the *opt_method_pointer specifications in StratSBO, StratOUU, StratBandB, and StratParetoSet and method_pointer specifications in StratSingle and StratMultiStart*).

- int **strategyBandBNumSamplesRoot**
  number of samples at the root for the branch and bound strategy (from the *num_samples_at_root specification in StratBandB*).

- int **strategyBandBNumSamplesNode**
  number of samples at each node for the branch and bound strategy (from the *num_samples_at_node specification in StratBandB*).

- **DakotaStringList strategyMultilevelMethodList**
  list of methods for the multilevel hybrid optimization strategy (from the *method_list specification in StratML*).

- **DakotaString strategyMultilevelType**
  the type of multilevel hybrid optimization strategy: uncoupled, uncoupled adaptive, or coupled (from the *uncoupled, adaptive, and coupled specifications in StratML*).

- **Real strategyMultilevelProgThresh**
  progress threshold for uncoupled adaptive multilevel hybrids (from the *progress_threshold specification in StratML*).

- **DakotaString strategyMultilevelGlobalMethodPointer**
  global method pointer for coupled multilevel hybrids (from the *global_method_pointer specification in StratML*).

- **DakotaString strategyMultilevelLocalMethodPointer**
  local method pointer for coupled multilevel hybrids (from the *local_method_pointer specification in StratML*).

- **Real strategyMultilevelLSProb**
  local search probability for coupled multilevel hybrids (from the *local_search_probability specification in StratML*).
- int `strategySBOMaxIterations`
  maximum number of iterations in the surrogate-based optimization strategy (from the `max_iterations` specification in `StratSBO`).

- Real `strategySBOTRInitSize`
  initial trust region size in the surrogate-based optimization strategy (from the `init_size` specification in `StratSBO`).

- Real `strategySBOTRContract`
  trust region contraction factor in the surrogate-based optimization strategy (from the `contraction_factor` specification in `StratSBO`).

- Real `strategySBOTRExpand`
  trust region expansion factor in the surrogate-based optimization strategy (from the `expansion_factor` specification in `StratSBO`).

- int `strategyConcurrentNumJobs`
  number of iterator jobs to perform in the concurrent strategy (from the `num_starts` and `num_optima` specifications in `StratMultiStart` and `StratParetoSet`).

- DakotaRealVector `strategyConcurrentParameterSets`
  number of parameter sets to evaluate in the concurrent strategy (from the `starting_points` and `multi-objective_weight_sets` specifications in `StratMultiStart` and `StratParetoSet`).

- DakotaList< DataMethod > `methodList`
  list of method specifications, one for each call to `method_handler` by the parser.

- DakotaList< DataVariables > `variablesList`
  list of variables specifications, one for each call to `variables_handler` by the parser.

- DakotaList< DataInterface > `interfaceList`
  list of interface specifications, one for each call to `interface_handler` by the parser.

- DakotaList< DataResponses > `responsesList`
  list of responses specifications, one for each call to `responses_handler` by the parser.

- int `methodIndex`
  index into `methodList` (identifies the active method specification).

- int `variablesIndex`
  index into `variablesList` (identifies the active variables specification).

- int `interfaceIndex`
  index into `interfaceList` (identifies the active interface specification).

- int `responsesIndex`
  index into `responsesList` (identifies the active responses specification).

- int `strategyIndex`
  used only in `check_input` (there is no strategy specification list) to verify that there is only one strategy specification.
6.70.1 Detailed Description

The database containing information parsed from the DAKOTA input file.

The ProblemDescDB class is a database for DAKOTA input file data that is populated by the Input Deck Reader (IDR) parser. When the parser reads a complete keyword (delimited by a newline), it calls the corresponding kwhandler function from this class, which (for method, variables, interface, or responses specifications) populates a data class object (DataMethod, DataVariables, DataInterface, or DataResponses) and appends the object to a linked list (methodList, variablesList, interfaceList, or responsesList). The strategy kwhandler is the exception to this, since the restriction of only allowing one strategy specification means there’s no need for a DataStrategy class or a strategyList (instead, strategy attributes are members of ProblemDescDB). For information on modifying the input parsing procedures, refer to Dakota/docs/spec_change_instructions.txt

6.70.2 Member Function Documentation

6.70.2.1 void ProblemDescDB::set_db_model_type (const DakotaString & model_type)

set the model type.

Used to avoid recursion in DakotaModel::get_model() by a sub model when get_string("method.model - type") is not reset by a sub iterator. Note: if more needs of this type arise, could add set_<type> member functions to parallel the existing get_<type> member functions.

The documentation for this class was generated from the following files:

- ProblemDescDB.H
- ProblemDescDB.C
6.71 RespSurf Class Reference

Derived approximation class for quadratic polynomial regression.

```cpp
#include <RespSurf.H>
```

Inheritance diagram for RespSurf:

```
DakotaApproximation
    |
    v
RespSurf
```

Public Methods

- `RespSurf` (const `ProblemDescDB &problem_db`)  
  *constructor.*

- `~RespSurf` ()  
  *destructor.*

Protected Methods

- `void findCoefficients ()`  
  *Least squares fit to data using a singular value decomposition.*

- `int requiredSamples (int num_vars)`  
  *return the minimum number of samples required to build the derived class approximation type in num_vars dimensions.*

- `Real getValue (const DakotaRealVector &x)`  
  *retrieve the approximate function value for a given parameter vector.*

- `const DakotaRealVector & getGradient (const DakotaRealVector &x)`  
  *retrieve the approximate function gradient for a given parameter vector.*

Private Attributes

- `int numRowsA`  
  *Number of rows in matrix A.*

- `int numColsA`  
  *Number of columns in matrix A.*
- int numRHS
  Number of right hand side vectors for least squares solution.

- int leadDimA
  The leading dimension of matrix A.

- int leadDimB
  The leading dimension of matrix B (always=1 here since numRHS always=1).

- Real rcond
  Flag to use machine precision to rank singular values of A.

- int rank
  The effective rank of matrix A.

- int lwork
  The length of the work vector.

- int info
  Output flag from DGELSS subroutine.

- Real * matrixTerms
  Matrix of quadratic polynomial terms unrolled into a vector.

- Real * responseValues
  Vector of response values that correspond to the samples in matrix A.

- Real * quadCoeffs
  Vector of quadratic polynomial coefficients.

- Real * matrixS
  Temporary storage for interface to Fortran77 LAPACK subroutines.

- Real * matrixW
  Temporary storage for interface to Fortran77 LAPACK subroutines.

### 6.71.1 Detailed Description

Derived approximation class for quadratic polynomial regression.

The RespSurf class assumes a quadratic polynomial fit to data which has \((n+1)(n+2)/2\) coefficients for \(n\) variables. A least squares estimation of the quadratic polynomial coefficients is performed using LAPACK’S linear least squares subroutine DGELSS which uses a singular value decomposition method.

The documentation for this class was generated from the following files:

- RespSurf.H
- RespSurf.C
6.72 SGOPTApplication Class Reference

Maps the evaluation functions used by SGOPT algorithms to the DAKOTA evaluation functions.

```c
#include "SGOPTApplication.H"
```

Public Methods

- **SGOPTApplication** (DakotaModel &model, DakotaResponse(*multiobj_mod_ptr)(const DakotaResponse &), int type)  
  constructor.
- **~SGOPTApplication** ()  
  destructor.
- **int DoEval** (OptPoint &pt, OptResponse *response, int synch_flag)  
  launch a function evaluation either synchronously or asynchronously.
- **int synchronize** ()  
  blocking retrieval of all pending jobs.
- **int nextEval** (int &id)  
  nonblocking query and retrieval of a job if completed.
- **void dakota_asynch_flag** (const short &asynch_flag)  
  set dakotaModelAsynchFlag.

Private Methods

- **void copy** (const DakotaResponse &, OptResponse &)  
  copy data from a DakotaResponse object to an SGOPT OptResponse object.

Private Attributes

- **DakotaModel & userDefinedModel**  
  reference to the SGOPTOptimizer’s model passed in the constructor.
- **DakotaIntArray activeSetVector**  
  copy/conversion of the SGOPT request vector.
- **short dakotaModelAsynchFlag**  
  a flag for asynchronous DAKOTA evaluations.
- **DakotaList< DakotaResponse > dakotaResponseList**
list of DAKOTA responses returned by synchronize_nowait().

- DakotaIntList dakotaCompletionList
  list of DAKOTA completions returned by synchronize_nowait_completions().

- size_t numObjFns
  number of objective functions.

- size_t numNonlinCons
  number of nonlinear constraints.

- DakotaResponse(* multiobjModifyPtr ) (const DakotaResponse &)
  function pointer to DakotaOptimizer::multi_objective_modify() for reducing multiple objective functions to a single function.

6.72.1 Detailed Description

Maps the evaluation functions used by SGOPT algorithms to the DAKOTA evaluation functions.

SGOPTApplication is a DAKOTA class that is derived from SGOPT’s AppInterface hierarchy. It redefines a variety of virtual SGOPT functions to use the corresponding DAKOTA functions. This is a more flexible algorithm library interfacing approach than can be obtained with the function pointer approaches used by NPSOLOptimizer and SNLLOptimizer.

6.72.2 Member Function Documentation

6.72.2.1 int SGOPTApplication::DoEval (OptPoint & pt, OptResponse * prob_response, int synch_flag)

launch a function evaluation either synchronously or asynchronously.

Converts SGOPT variables and request vector to DAKOTA variables and active set vector, performs a DAKOTA function evaluation with synchronization governed by synch_flag, and then copies the DakotaResponse data to the SGOPT response (synchronous) or bookkeeps the SGOPT response object (asynchronous).

6.72.2.2 int SGOPTApplication::synchronize ()

blocking retrieval of all pending jobs.

Blocking synchronize of asynchronous DAKOTA jobs followed by conversion of the DakotaResponse objects to SGOPT response objects.

6.72.2.3 int SGOPTApplication::next_eval (int & id)

nonblocking query and retrieval of a job if completed.

Nonblocking job retrieval. Finds a completion (if available), populates the SGOPT response, and sets id to the completed job’s id. Else set id = -1.
6.72.2.4  void SGOPTApplication::dakota_asynch_flag (const short & asynch_flag) [inline]

set dakotaModelAsynchFlag.

This function is needed to publish the iterator’s asynchFlag at run time (asynchFlag not available at construction).

The documentation for this class was generated from the following files:

- SGOPTApplication.H
- SGOPTApplication.C
6.73 SGOPTOptimizer Class Reference

Wrapper class for the SGOPT optimization library.

#include &lt;SGOPTOptimizer.H&gt;

Inheritance diagram for SGOPTOptimizer:

```
DakotaIterator
  
DakotaOptimizer
  
SGOPTOptimizer
```

Public Methods

- `SGOPTOptimizer (DakotaModel &model)`
  
  constructor.

- `~SGOPTOptimizer ()`
  
  destructor.

- `void find_optimum ()`
  
  Performs the iterations to determine the optimal solution.

Private Methods

- `void set_method_options ()`
  
  sets options for the methods based on user specifications.

Private Attributes

- `DakotaString exploratoryMoves`
  
  user input for desired pattern search algorithm variant.

- `short discreteAppFlag`
  
  convenience flag for integer vs. real applications.

- `PM_LCG * linConGenerator`
  
  Pointer to random number generator.

- `BaseOptimizer * baseOptimizer`
6.73 SGOPTOptimizer Class Reference

• AppInterface * sgoptApplication
  pointer to the SGOPTApplication object.

• RealOptProblem * realProblem
  pointer to RealOptProblem object.

• IntOptProblem * intProblem
  pointer to IntOptProblem object.

• PGAreal * pGAREalOptimizer
  pointer to PGAreal object.

• P GAint * pGAIntOptimizer
  pointer to P GAint object.

• EPSA * ePSAOptimizer
  pointer to EPSA object.

• PatternSearch * patternSearchOptimizer
  pointer to PatternSearch object.

• APPSOpt * aPPSOptimizer
  pointer to APPSOpt object.

• SWOpt * sWOptimizer
  pointer to SWOpt object.

• sMCreal * sMCrealOptimizer
  pointer to sMCreal object.

6.73.1 Detailed Description

Wrapper class for the SGOPT optimization library.

The SGOPTOptimizer class provides a wrapper for SGOPT, a Sandia-developed C++ optimization library of genetic algorithms, pattern search methods, and other nongradient-based techniques. It uses an SGOPTApplication object to perform the function evaluations.

The user input mappings are as follows: max_iterations, max_function_evaluations, convergence_tolerance, solution_accuracy and max_cpu_time are mapped into SGOPT’s max_iter, max_neval, ftol, accuracy, and max_time data attributes. An output setting of verbose is passed to SGOPT’s set_output() function and a setting of debug activates output of method initialization and sets the SGOPT debug attribute to 10000. SGOPT methods assume asynchronous operations whenever the algorithm has independent evaluations which can be performed simultaneously (implicit parallelism). Therefore, parallel configuration is not mapped into the method, rather it is used in SGOPTApplication to control whether or not an asynchronous evaluation request from the method is honored by the model (exception: pattern search exploratory moves is set to best all for parallel function evaluations). Refer to [Hart, W.E., 1997] for additional information on SGOPT objects and controls.
6.73.2 Constructor & Destructor Documentation

6.73.2.1 SGOPTOptimizer::SGOPTOptimizer (DakotaModel & model)

constructor.
The constructor allocates the objects and populates the class member pointer attributes.

6.73.2.2 SGOPTOptimizer::~SGOPTOptimizer ()

destructor.
The destructor deallocates the class member pointer attributes.

6.73.3 Member Function Documentation

6.73.3.1 void SGOPTOptimizer::find_optimum () [virtual]

Performs the iterations to determine the optimal solution.
find_optimum redefines the DakotaOptimizer virtual function to perform the optimization using SGOPT. It first sets up the problem data, then executes minimize() on the SGOPT algorithm, and finally catalogues the results.
Reimplemented from DakotaOptimizer.

6.73.3.2 void SGOPTOptimizer::set_method_options () [private]

sets options for the methods based on user specifications.
set_method_options propagates DAKOTA user input to the appropriate SGOPT objects.

6.73.4 Member Data Documentation

6.73.4.1 AppInterface* SGOPTOptimizer::sgoptApplication [private]

pointer to the SGOPTApplication object.
SGOPTApplication is a DAKOTA class derived from the SGOPT AppInterface class. It redefines the virtual SGOPT evaluation functions to use DAKOTA evaluation functions.
The documentation for this class was generated from the following files:

- SGOPTOptimizer.H
- SGOPTOptimizer.C
6.74 SingleMethodStrategy Class Reference

Simple fall-through strategy for running a single iterator on a single model.

```
#include <SingleMethodStrategy.H>
```

Inheritance diagram for SingleMethodStrategy::

```
DakotaStrategy
  ↓
SingleMethodStrategy
```

### Public Methods

- **SingleMethodStrategy** (ProblemDescDB &problem_db)
  
  *constructor.*

- **~SingleMethodStrategy** ()
  
  *destructor.*

- **void run_strategy** ()
  
  *Perform the strategy by executing selectedIterator on userDefinedModel.*

### Private Attributes

- **DakotaModel userDefinedModel**
  
  *the model to be iterated.*

- **DakotaIterator selectedIterator**
  
  *the iterator.*

### 6.74.1 Detailed Description

Simple fall-through strategy for running a single iterator on a single model.

This strategy executes a single iterator on a single model. Since it does not provide coordination for multiple iterators and models, it can considered to be a "fall-through" strategy in that it allows control to fall through immediately to the iterator.

The documentation for this class was generated from the following files:

- SingleMethodStrategy.H
- SingleMethodStrategy.C
6.75 SingleModel Class Reference

Derived model class which utilizes a single interface to map variables into responses.

```cpp
#include "SingleModel.H"
```

Inheritance diagram for SingleModel:

```
DakotaModel
   ↓
SingleModel
```

Public Methods

- `SingleModel (ProblemDescDB &problem_db)`
  
  *constructor.*

- `~SingleModel ()`
  
  *destructor.*

- `void derived_compute_response (const DakotaIntArray &asv)`
  
  *portion of compute_response() specific to SingleModel (invokes a synchronous map() on userDefined-Interface).*

- `void derived_asynch_compute_response (const DakotaIntArray &asv)`
  
  *portion of asynch_compute_response() specific to SingleModel (invokes an asynchronous map() on user-DefinedInterface).*

- `const DakotaArray< DakotaResponse > & derived_synchronize ()`
  
  *portion of synchronize() specific to SingleModel (invokes synch() on userDefinedInterface).*

- `const DakotaList< DakotaResponse > & derived_synchronize_nowait ()`
  
  *portion of synchronize_nowait() specific to SingleModel (invokes synch_nowait() on userDefinedInterface).*

- `DakotaString local_eval_synchronization ()`
  
  *return userDefinedInterface synchronization setting.*

- `const DakotaIntArray & synchronize_nowait_completions ()`
  
  *return completion id’s matching response list from synchronize_nowait (request forwarded to userDefined-Interface).*

- `short derived_master_overload () const`
  
  *flag which prevents overloading the master with a multiprocessor evaluation (request forwarded to user-DefinedInterface).*
6.75 SingleModel Class Reference

- void *derived_init_communicators* (const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)

  *portion of init_communicators() specific to SingleModel (request forwarded to userDefinedInterface).*

- void *free_communicators* ()

  *deallocate communicator partitions for the SingleModel (request forwarded to userDefinedInterface).*

- void *serve* ()

  *Service job requests received from the master. Completes when a termination message is received from stop_servers() (request forwarded to userDefinedInterface).*

- void *stop_servers* ()

  *executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (request forwarded to userDefinedInterface).*

- int *total_eval_counter* () const

  *return the total evaluation count for the SingleModel (request forwarded to userDefinedInterface).*

- int *new_eval_counter* () const

  *return the new evaluation count for the SingleModel (request forwarded to userDefinedInterface).*

**Private Attributes**

- DakotaInterface *userDefinedInterface*

  *the interface used for mapping variables to responses.*

**6.75.1 Detailed Description**

Derived model class which utilizes a single interface to map variables into responses.

The SingleModel class is the simplest of the derived model classes. It provides the capabilities the old DakotaModel class, prior to the development of layered and nested model extensions. The derived response computation and synchronization functions utilize a single interface to perform the function evaluations.

The documentation for this class was generated from the following files:

- SingleModel.H
- SingleModel.C
6.76 SNLLOptimizer Class Reference

Wrapper class for the OPT++ optimization library.

```cpp
#include "SNLLOptimizer.H"
```

Inheritance diagram for SNLLOptimizer::

```
          DakotIterator
             |
             v
  DakotaOptimizer
             |
             v
SNLLOptimizer
```

Public Methods

- **SNLLOptimizer (DakotaModel &model)**
  
  *constructor.*

- **~SNLLOptimizer ()**
  
  *destructor.*

- **void find_optimum ()**
  
  *Performs the iterations to determine the optimal solution.*

Static Private Methods

- **void init_fn (int n, ColumnVector &x)**
  
  *An initialization mechanism provided by OPT++ (not currently used).*

- **void nfl0_evaluator (int n, const ColumnVector &x, Real &f, int &result_mode)**
  
  *objective function evaluator function for OPT++ methods which require only function values.*

- **void fdnlf1_evaluator (int n, const ColumnVector &x, Real &f, int &result_mode)**
  
  *objective function evaluator function which provides function values to OPT++ methods for computing numerical gradients by finite differences.*

- **void nfl1_evaluator (int mode, int n, const ColumnVector &x, Real &f, ColumnVector &g, int &result_mode)**
  
  *objective function evaluator function which provides function values and gradients to OPT++ methods.*

- **void nfl2_evaluator (int mode, int n, const ColumnVector &x, Real &f, ColumnVector &g, SymmetricMatrix &h, int &result_mode)**
Objective function evaluator function which provides function values, gradients, and Hessians to OPT++ methods.

- void nlf2_evaluator_gn (int mode, int n, const ColumnVector &x, Real &fx, ColumnVector &grad, SymmetricMatrix &hess, int &result_mode)
  
  Objective function evaluator function which obtains values and gradients for least square terms and computes objective function value, gradient, and Hessian using the Gauss-Newton approximation.

- void constraint0_evaluator (int n, const ColumnVector &x, ColumnVector &f, int &result_mode)
  
  Constraint evaluator function for OPT++ methods which require only constraint values.

- void constraint1_evaluator (int mode, int n, const ColumnVector &x, ColumnVector &f, Matrix &g, int &result_mode)
  
  Constraint evaluator function which provides constraint values and gradients to OPT++ methods.

- void constraint2_evaluator (int mode, int n, const ColumnVector &x, ColumnVector &f, Matrix &g, OptppArray< SymmetricMatrix > &h, int &result_mode)
  
  Constraint evaluator function which provides constraint values, gradients, and Hessians to OPT++ methods.

Private Attributes

- DakotaString searchMethod
  
  Value-based line search, gradient-based line search, trust region, or trustPDS.

- SearchStrategy s
  
  Enum: LineSearch, TrustRegion, or TrustPDS.

- MeritFcn mfcn
  
  Enum: NormFmu, ArgyaezTapia, or VanShanno.

- short vendorNumericalGradFlag
  
  Flags numerical gradients via the internal OPT++ finite differencing routine.

- NLP0 * nlfObjective
  
  Objective NLF base class pointer.

- NLP0 * nlfConstraint
  
  Constraint NLF base class pointer.

- NLP * nlpConstraint
  
  Constraint NLP pointer.

- NLF0 * nlf0
  
  Pointer to objective NLF for nongradient optimizers.

- NLF1 * nlf1
  
  Pointer to objective NLF for (analytic) gradient optimizers.
- NLF1 * nlf1Con
  pointer to constraint NLF for (analytic) gradient optimizers.

- FDNLF1 * fdnlf1
  pointer to objective NLF for (finite diff) gradient optimizers.

- FDNLF1 * fdnlf1Con
  pointer to constraint NLF for (finite diff) gradient optimizers.

- NLF2 * nlf2
  pointer to objective NLF for full Newton optimizers.

- NLF2 * nlf2Con
  pointer to constraint NLF for full Newton optimizers.

- NLF1 * bcnlf1
  pointer to objective NLF for bound constrained (analytic) gradient optimizers.

- FDNLF1 * bcfdnlf1
  pointer to objective NLF for bound constrained (finite diff) gradient optimizers.

- NLF2 * bcnlf2
  pointer to objective NLF for bound constrained full Newton optimizers.

- OptimizeClass * theOptimizer
  optimizer base class pointer.

- OptPDS * optpds
  PDS optimizer pointer.

- OptCG * optcg
  CG optimizer pointer.

- OptNewton * optnewton
  Newton optimizer pointer.

- OptQNewton * optqnewton
  Quasi-Newton optimizer pointer.

- OptFDNewton * optfdnewton
  Finite Difference Newton optimizer pointer.

- OptBCNewton * optbcnewton
  Bound constrained Newton optimizer pointer.

- OptBCQNewton * optbcqnewton
  Bound constr Quasi-Newton optimizer pointer.

- OptBaNewton * optbanewton
  Barrier Newton optimizer pointer.
- OptBaQNewton * optbaqnewton
  
  Barrier Quasi-Newton optimizer pointer.

- OptBCEllipsoid * optbcellipsoid
  
  Bound constrained ellipsoid pointer.

- OptNIPS * optnips
  
  NIPS optimizer pointer.

- OptQNIPS * optqnips
  
  Quasi-Newton NIPS optimizer pointer.

- OptFDNIPS * optfdnips
  
  Finite Difference NIPS optimizer pointer.

### Static Private Attributes

- int staticSpeculativeFlag
  
  flags speculative gradient logic (for parallel load-balancing).

- int staticModeOverrideFlag
  
  flags OPT++ mode override (for combining value, gradient, and Hessian requests).

- DakotaRealVector staticConstraintValues
  
  vector of nonlinear constraints.

- int staticNumNonlinearEqConstraints
  
  number of nonlinear equality constraints.

- int staticNumNonlinearIneqConstraints
  
  number of nonlinear inequality constraints.

### 6.76.1 Detailed Description

Wrapper class for the OPT++ optimization library.

The SNLLOptimizer class provides a wrapper for OPT++, a C++ optimization library of nonlinear programming and pattern search techniques from the Computational Sciences and Mathematics Research (CSMR) department at Sandia’s Livermore CA site. It uses a function pointer approach for which passed functions must be either global functions or static member functions. Any attribute used within static member functions must be either local to that function or static as well. To isolate the effect of these static requirements from the rest of the iterator hierarchy, static copies are made of many non-static attributes inherited from above.

The user input mappings are as follows: max_iterations, max_function_evaluations, convergence_tolerance, max_step, gradient_tolerance, search_method, initial_radius, and search_scheme_size are set using OPT++’s setMaxIter(), setMaxFeval(), setFcnTol(),
setMaxStep(), setGradTol(), setSearchStrategy(), setInitialEllipse(), and setSSS() member functions, respectively; output verbosity is used to toggle OPT++’s debug mode using the setDebug() member function. Internal to OPT++, there are 3 search strategies, while the DAKOTA search method specification supports 4 (value_based_line_search, gradient_based_line_search, trust_region, or tr_pds). The difference stems from the “is_expensive” flag in OPT++. If the search strategy is LineSearch and “is_expensive” is turned on, then the value_based_line_search is used. Otherwise (the “is_expensive” default is off), the algorithm will use the gradient_based_line_search. Refer to [Meza, J.C., 1994] and to the OPT++ source in the Dakota/VendorOptimizers/opt++ directory for information on OPT++ class member functions.

6.76.2 Member Function Documentation

6.76.2.1 void SNLLOptimizer::nlf0_evaluator (int n, const ColumnVector & x, Real & f, int & result_mode) [static, private]

objective function evaluator function for OPT++ methods which require only function values.
For use when DAKOTA computes f and no gradients are available. There is currently no difference between this function & fdnlf1_evaluator.

6.76.2.2 void SNLLOptimizer::fdnlf1_evaluator (int n, const ColumnVector & x, Real & f, int & result_mode) [static, private]

objective function evaluator function which provides function values to OPT++ methods for computing numerical gradients by finite differences.
For use when DAKOTA computes f and opt++’s internal finite difference capability computes df/dX. In effect, fdnlf1 is like an nlf0.

6.76.2.3 void SNLLOptimizer::nlf1_evaluator (int mode, int n, const ColumnVector & x, Real & f, ColumnVector & g, int & result_mode) [static, private]

objective function evaluator function which provides function values and gradients to OPT++ methods.
For use when DAKOTA computes f and df/dX (regardless of gradientType). Vendor numerical gradient case is handled within fdnlf1_evaluator.

6.76.2.4 void SNLLOptimizer::nlf2_evaluator (int mode, int n, const ColumnVector & x, Real & f, ColumnVector & g, SymmetricMatrix & h, int & result_mode) [static, private]

objective function evaluator function which provides function values, gradients, and Hessians to OPT++ methods.
For use when DAKOTA receives f, df/dX, & d^2f/dx^2 from the ApplicationInterface (analytic only). Finite differencing does not make sense for a full Newton approach, since lack of analytic gradients & Hessian should dictate the use of quasi-newton or fd-newton. Thus, there is no fdnlf2_evaluator for use with full Newton approaches, since it is preferable to use quasi-newton or fd-newton with nlf1. Gauss-Newton does not fit this model; it uses nlf2_evaluator_gn instead of nlf2_evaluator.
6.76.2.5 void SNLLOptimizer::nlf2_evaluator_gn (int mode, int n, const ColumnVector & x, Real & fx, ColumnVector & grad, SymmetricMatrix & hess, int & result_mode) [static, private]

objective function evaluator function which obtains values and gradients for least square terms and computes objective function value, gradient, and Hessian using the Gauss-Newton approximation.

This nlf2_evaluator function is used for the Gauss-Newton method in order to exploit the special structure of the nonlinear least squares problem. Here, \( \text{fx} = \sum (T_i - T_{\text{bar},i})^2 \) and DakotaResponse is made up of residual functions and their gradients with numFunctions = numLeastSquaresTerms. The objective function and its gradient vector and Hessian matrix are computed directly from the residual functions and their derivatives (which are returned from the DakotaResponse object).

The documentation for this class was generated from the following files:

- SNLLOptimizer.H
- SNLLOptimizer.C

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
6.77  SortCompare Class Template Reference

#include <DakotaList.H>

Public Methods

- **SortCompare** (int(*func)(const T &, const T &))
  Constructor that defines the pointer to function.

- **bool operator()** (const T &p1, const T &p2) const
  The operator() must be defined. Calls the defined sortFunction.

Private Attributes

- int(* sortFunction )(const T &, const T &)
  Pointer to test function.

6.77.1  Detailed Description

template<class T> class SortCompare<T>

Internal Functor used in the sort algorithm to sort using a specified compare method. The class holds a pointer to the sort function.

The documentation for this class was generated from the following file:

- DakotaList.H
6.78 SurrBasedOptStrategy Class Reference

Strategy for provably-convergent surrogate-based optimization.

```c
#include <SurrBasedOptStrategy.H>
```

Inheritance diagram for SurrBasedOptStrategy:

```
DakotaStrategy

SurrBasedOptStrategy
```

### Public Methods

- **SurrBasedOptStrategy** (ProblemDescDB &problem_db)
  
  *constructor.*

- **~SurrBasedOptStrategy** ()
  
  *destructor.*

- **void run_strategy** ()
  
  *Performs the surrogate-based optimization strategy by optimizing local, global, or hierarchical surrogates over a series of trust regions.*

### Private Methods

- **Real compute_penalty_function** (const DakotaRealVector &fn_vals)
  
  *compute a penalty function from a set of function values.*

- **int hard_convergence_check** (const DakotaResponse &response_center_truth)
  
  *check for hard convergence (satisfaction of gradient FCD condition).*

- **int soft_convergence_check** (const DakotaRealVector &c_vars_center, const DakotaRealVector &c_vars_star, const DakotaResponse &response_center_truth, const DakotaResponse &response_center_approx, const DakotaResponse &response_star_truth, const DakotaResponse &response_star_approx)
  
  *check for soft convergence (diminishing returns).*

### Private Attributes

- **DakotaModel approximateModel**
  
  *the surrogate model (a LayeredModel object).*
- **DakotaIterator selectedIterator**
  
  _the optimizer used on approximateModel._

- **Real trustRegionSize**
  
  _size of the current trust region (dimensional +/- offset around center for each variable defines a hypercube)._

- **Real minTrustRegionSize**
  
  _a soft convergence control: stop SBO when the trust region size is reduced below the minTrustRegionSize._

- **Real convergenceTol**
  
  _the optimizer convergence tolerance; used in several SBO hard and soft convergence checks._

- **Real constraintTol**
  
  _a tolerance specifying the distance from a constraint boundary that is allowed before an active constraint is considered to be a violated constraint (only violated constraints are used in penalty function computations)._

- **Real penaltyParameter**
  
  _the penalization factor for violated constraints used in penalty function calculations; increases exponentially with iteration count._

- **Real gammaContract**
  
  _trust region contraction factor._

- **Real gammaExpand**
  
  _trust region expansion factor._

- **Real gammaNoChange**
  
  _factor for maintaining the current trust region size (normally 1.0)._ 

- **Real fcdGradientTerm**
  
  _gradient-related term from the fraction of Cauchy decrease (FCD) calculation._

- **int iterMax**
  
  _maximum number of SBO iterations._

- **int convergenceFlag**
  
  _code indicating satisfaction of hard or soft convergence conditions._

- **int numFns**
  
  _number of response functions._

- **int numVars**
  
  _number of active continuous variables._

- **int softConvCount**
  
  _number of consecutive candidate point rejections. If the count reaches softConvLimit, stop SBO._

- **int softConvLimit**
  
  _the limit on consecutive candidate point rejections. If exceeded by softConvCount, stop SBO._
6.78 SurrBasedOptStrategy Class Reference

- **short gradientFlag**
  flags the use of gradients throughout the SBO process.

- **short correctionFlag**
  flags the use of surrogate correction techniques at the center of each trust region.

- **short globalApproxFlag**
  flags the use of a global data fit surrogate (rsm, ann, mars, kriging).

- **short localApproxFlag**
  flags the use of a local data fit surrogate (Taylor series).

- **short hierarchApproxFlag**
  flags the use of a hierarchical surrogate.

- **short newCenterFlag**
  flags the acceptance of a candidate point and the existence of a new trust region center.

- **short daceCenterPtFlag**
  flags the availability of the center point in the DACE evaluations for global approximations (CCD, Box-Behnken).

- **size numObjFs**
  number of objective functions.

- **size numNonlinIneqConstr**
  number of nonlinear inequality constraints.

- **size numNonlinEqConstr**
  number of nonlinear equality constraints.

- **DakotaRealVector multiObjWts**
  vector of multiobjective weights.

- **DakotaRealVector nonlinIneqLowerBnds**
  vector of nonlinear inequality constraint lower bounds.

- **DakotaRealVector nonlinIneqUpperBnds**
  vector of nonlinear inequality constraint upper bounds.

- **DakotaRealVector nonlinEqTargets**
  vector of nonlinear equality constraint targets.

### 6.78.1 Detailed Description

Strategy for provably-convergent surrogate-based optimization.

This strategy uses a *LayeredModel* to perform optimization based on local, global, or hierarchical surrogates. It achieves provable convergence through the use of a sequence of trust regions and the application of surrogate corrections at the trust region centers.
6.78.2 Member Function Documentation

6.78.2.1 `void SurrBasedOptStrategy::run_strategy()` [virtual]

Performs the surrogate-based optimization strategy by optimizing local, global, or hierarchical surrogates over a series of trust regions.

Trust region-based strategy to perform surrogate-based optimization in subregions (trust regions) of the parameter space. The optimizer operates on approximations in lieu of the more expensive simulation-based response functions. The size of the trust region is varied according to the goodness of the agreement between the approximations and the true response functions.

Reimplemented from DakotaStrategy.

6.78.2.2 `Real SurrBasedOptStrategy::compute_penalty_function(const DakotaRealVector & fn_yals)` [private]

compute a penalty function from a set of function values.

The penalty function computation applies a penalty multiplier to any constraint violations and adds this to the objective function. This implementation supports multiple objectives, equality constraints, and 2-sided inequalities. A negative constraintTol can be used to provide a push-back into the feasible region.

6.78.2.3 `int SurrBasedOptStrategy::hard_convergence_check(const DakotaResponse & center_truth)` [private]

check for hard convergence (satisfaction of gradient FCD condition).

The hard convergence check computes a fraction of Cauchy decrease (FCD) condition using gradients. It is based on equations from Alexandrov (Structural Opt 1998). Basically, if the gradient of the penalty function at the trust region center is near zero, stop the SBO iterations. Some computed terms from this function are reused in `soft_convergence_check()`.

6.78.2.4 `int SurrBasedOptStrategy::soft_convergence_check(const DakotaRealVector & c_yars_center, const DakotaRealVector & c_yars_star, const DakotaResponse & response_center_truth, const DakotaResponse & response_center_approx, const DakotaResponse & response_star_truth, const DakotaResponse & response_star_approx)` [private]

check for soft convergence (diminishing returns).

Compute soft convergence metrics (quasi-FCD, trust region ratio, number of consecutive failures, min trust region size, etc.) and use them to assess whether the convergence rate has decreased to a point where the process should be terminated (diminishing returns). Some gradient-related terms are computed in `hard_convergence_check()`.

The documentation for this class was generated from the following files:

- SurrBasedOptStrategy.H
- SurrBasedOptStrategy.C
6.79  **SurrLayeredModel Class Reference**

Derived model class within the layered model branch for managing data fit surrogates (global and local).

```cpp
#include <SurrLayeredModel.H>
```

Inheritance diagram for SurrLayeredModel:

- DakotaModel
- LayeredModel
- SurrLayeredModel

### Public Methods

- **SurrLayeredModel** *(ProblemDescDB &problem_db)*
  - *constructor.*

- ~**SurrLayeredModel** *
  - *destructor.*

### Protected Methods

- void **derived_compute_response** (const DakotaIntArray &asv)
  - *portion of compute_response() specific to SurrLayeredModel.*

- void **derived_asynch_compute_response** (const DakotaIntArray &asv)
  - *portion of asynch_compute_response() specific to SurrLayeredModel.*

- const DakotaArray & **DakotaResponse** & **derived_synchronize**
  - *portion of synchronize() specific to SurrLayeredModel.*

- const DakotaList & **DakotaResponse** & **derived_synchronize_nowait**
  - *portion of synchronize_nowait() specific to SurrLayeredModel.*

- short **derived_master_overload** const
  - *flag which prevents overloading the master with a multiprocessor evaluation.*

- DakotaModel & **subordinate_model**
  - *returns actualModel to SurrBasedOptStrategy.*

- DakotaIterator & **subordinate_iterator**
return daceIterator to SurrBasedOptStrategy.

- int maximum_concurrency() const
  return the maximum concurrency available for actualModel computations during global approximation builds.

- void build_approximation()
  Builds the local/multipoint/global approximation using daceIterator/actualModel.

- const DakotaIntList & synchronize_nowait_completions()
  return completion id's matching response list from synchronize_nowait (request forwarded to approxInterface).

- void update_approximation(const DakotaRealVector &x_star, const DakotaResponse &response_star)
  Adds a point to a global approximation (request forwarded to approxInterface).

- int total_eval_counter() const
  return the total evaluation count for the SurrLayeredModel (request forwarded to approxInterface).

- int new_eval_counter() const
  return the new evaluation count for the SurrLayeredModel (request forwarded to approxInterface).

- void derived_init_communicators(const DakotaIntArray &message_lengths, const int &max_iterator_concurrency)
  portion of init_communicators() specific to SurrLayeredModel.

- void free_communicators()
  deallocate communicator partitions for the SurrLayeredModel (request forwarded to actualModel).

- void serve()
  Service job requests received from the master. Completes when a termination message is received from stop_servers() (request forwarded to actualModel).

- void stop_servers()
  Executed by the master to terminate all slave server operations on a particular model when iteration on that model is complete (request forwarded to actualModel).

Private Attributes

- DakotaInterface approxInterface
  manages the building and subsequent evaluation of the approximations (required for both global and local).

- DakotaString actualInterfacePointer
  pointer to the actual interface from the local approximation specification (required for local); used to build actualModel for local approximations.

- DakotaString daceMethodPointer
  pointer to the dace method from the global approximation specification; used in building daceIterator and actualModel for global approximations (optional for global since restart data may also be used).
6.79 SurrLayeredModel Class Reference

- **DakotaModel actualModel**
  
  *the truth model which provides evaluations for building the surrogate (optional for global since restart data may also be used, required for local).*

- **DakotaIterator dacelator**

  *selects parameter sets on which to evaluate actualModel in order to generate the necessary data for building global approximations (optional for global since restart data may also be used).*

6.79.1 Detailed Description

Derived model class within the layered model branch for managing data fit surrogates (global and local).

The SurrLayeredModel class manages global or local approximations (surrogates that involve data fits) that are used in place of an expensive model. The class contains an approxInterface (required for both global and local) which manages the approximate function evaluations, an actualModel (optional for global, required for local) which provides truth evaluations for building the surrogate, and a dacelator (optional for global, not used for local) which selects parameter sets on which to evaluate actualModel in order to generate the necessary data for building global approximations.

6.79.2 Member Function Documentation

6.79.2.1 `void SurrLayeredModel::derived_compute_response(const DakotaIntArray & asv)`

[inline, protected, virtual]

portion of `compute_response()` specific to SurrLayeredModel.

Build the approximation (if needed), evaluate the approximate response using approxInterface, and, if correction is active, correct the results.

Reimplemented from DakotaModel.

6.79.2.2 `void SurrLayeredModel::derived_asynch_compute_response(const DakotaIntArray & asv)`

[inline, protected, virtual]

portion of `asynch_compute_response()` specific to SurrLayeredModel.

Build the approximation (if needed) and evaluate the approximate response using approxInterface in a quasi-asynchronous approach (`ApproximationInterface::map()` performs the map synchronously and book-keeps the results for return in `derived_synchronize()` below).

Reimplemented from DakotaModel.

6.79.2.3 `const DakotaArray < DakotaResponse > & SurrLayeredModel::derived_synchronize()`

[inline, protected, virtual]

portion of `synchronize()` specific to SurrLayeredModel.

Retrieve quasi-asynchronous evaluations from approxInterface and, if correction is active, apply correction to each response in the array.
Reimplemented from DakotaModel.

6.79.2.4  `const DakotaList< DakotaResponse > & SurrLayeredModel::derived_synchronize_nowait ()` [inline, protected, virtual]

portion of `synchronize_nowait()` specific to SurrLayeredModel.
Retrieve quasi-asynchronous evaluations from approxInterface and, if correction is active, apply correction to each response in the list.
Reimplemented from DakotaModel.

6.79.2.5  `short SurrLayeredModel::derived_master_overload () const` [inline, protected, virtual]

flag which prevents overloading the master with a multiprocessor evaluation.
compute_response calls never overload the master since there is no parallelism in the use of approxInterface.
Reimplemented from DakotaModel.

6.79.2.6  `int SurrLayeredModel::maximum_concurrency () const` [protected, virtual]

return the maximum concurrency available for actualModel computations during global approximation builds.
Return the greater of the dace samples user-specification or the min_samples approximation requirement.
min_samples does not have to account for reuse_samples, since this will vary (assume 0).
Reimplemented from DakotaModel.

6.79.2.7  `void SurrLayeredModel::build_approximation ()` [protected, virtual]

Builds the local/multipoint/global approximation using daceIterator/actualModel.
Build either a global approximation using daceIterator or a local approximation using actualModel. Selection triggers on actualInterfacePointer (required specification for local approximation interfaces, not used in global specification).
Reimplemented from DakotaModel.

6.79.2.8  `void SurrLayeredModel::derived_init_communicators (const DakotaIntArray & message_lengths, const int & max_iterator_concurrency)` [inline, protected, virtual]

portion of `init_communicators()` specific to SurrLayeredModel.
asynchronous flags need to be initialized for the sub-models. In addition, max_iterator_concurrency is the outer level iterator concurrency, not the DACE concurrency that actualModel will see, and recomputing the message_lengths on the sub-model is probably not a bad idea either. Therefore, recompute everything on actualModel using init_communicators.
Reimplemented from DakotaModel.
6.79 SurrLayeredModel Class Reference

6.79.3 Member Data Documentation

6.79.3.1 DakotaString SurrLayeredModel::actualInterfacePointer [private]

pointer to the actual interface from the local approximation specification (required for local); used to build
actualModel for local approximations.

Specification is used only for local approximations, since the dace method pointer in the global approxi-
mation specification is responsible for identifying all actualModel components.

6.79.3.2 DakotaModel SurrLayeredModel::actualModel [private]

the truth model which provides evaluations for building the surrogate (optional for global since restart data
may also be used, required for local).

There are no restrictions on actualModel in the global case, so arbitrary nestings are possible. In the
local case, model_type must be set to "single" to avoid recursion on SurrLayeredModel, since there is no
additional method specification.

The documentation for this class was generated from the following files:

- SurrLayeredModel.H
- SurrLayeredModel.C
6.80 SurrogateDataPoint Class Reference

Simple container class encapsulating basic parameter and response data for defining a "truth" data point.

```cpp
#include <DakotaApproximation.H>
```

Public Methods

- `SurrogateDataPoint ()
  
  default constructor.

- `SurrogateDataPoint (const DakotaRealVector &x, const Real &f, const DakotaRealVector &grad_f)
  
  standard constructor.

- `SurrogateDataPoint (const SurrogateDataPoint &sdp)
  
  copy constructor.

- `~SurrogateDataPoint ()
  
  destructor.

- `int operator==(const SurrogateDataPoint &sdp) const
  
  equality operator.

- `SurrogateDataPoint & operator=(const SurrogateDataPoint &sdp)
  
  assignment operator.

Public Attributes

- `DakotaRealVector continuousVars
  
  continuous variables.

- `Real responseFn
  
  truth response function value.

- `DakotaRealVector responseGrad
  
  truth response function gradient.

6.80.1 Detailed Description

Simple container class encapsulating basic parameter and response data for defining a "truth" data point.

A list of these data points is contained in each DakotaApproximation instance (`DakotaApproximation::currentPoints`) and provides the data to build the approximation. Data is public to avoid maintaining set/get functions, but is still encapsulated within DakotaApproximation since
DakotaApproximation::currentPoints is protected (a similar model is used with Data class objects contained in ProblemDescDB and with ParallelismLevel objects contained in ParallelLibrary).

The documentation for this class was generated from the following file:

- DakotaApproximation.H
6.81 SysCallAnalysisCode Class Reference

Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls.

```cpp
#include <SysCallAnalysisCode.H>
```

Inheritance diagram for SysCallAnalysisCode::

```
AnalysisCode

SysCallAnalysisCode
```

### Public Methods

- **SysCallAnalysisCode** (const ProblemDescDB &problem_db)
  
  *constructor.*

- **~SysCallAnalysisCode** ()
  
  *destructor.*

- void **spawn_evaluation** (const short block_flag)
  
  *spawn a complete function evaluation.*

- void **spawn_input_filter** (const short block_flag)
  
  *spawn the input filter portion of a function evaluation.*

- void **spawn_analysis** (const int &analysis_id, const short block_flag)
  
  *spawn a single analysis as part of a function evaluation.*

- void **spawn_output_filter** (const short block_flag)
  
  *spawn the output filter portion of a function evaluation.*

- const DakotaString & **command_usage** () const
  
  *return commandUsage.*

### Private Attributes

- **DakotaString commandUsage**
  
  *optional command usage string for supporting nonstandard command syntax (supported only by SysCall analysis codes).*
6.81 SysCallAnalysisCode Class Reference

6.81.1 Detailed Description

Derived class in the AnalysisCode class hierarchy which spawns simulations using system calls.
SysCallAnalysisCode creates separate simulation processes using the C system() command. It utilizes
CommandShell to manage shell syntax and asynchronous invocations.

6.81.2 Member Function Documentation

6.81.2.1 void SysCallAnalysisCode::spawn_evaluation (const short block flag)

spawn a complete function evaluation.
Put the SysCallAnalysisCode to the shell using either the default syntax or specified commandUsage syn-
tax. This function is used when all portions of the function evaluation (i.e., all analysis drivers) are executed
on the local processor.

6.81.2.2 void SysCallAnalysisCode::spawn_input_filter (const short block flag)

spawn the input filter portion of a function evaluation.
Put the input filter to the shell. This function is used when multiple analysis drivers are spread between
processors. No need to check for a Null input filter, as this is checked externally. Use of nonblocking shells
is supported in this fn, although its use is currently prevented externally.

6.81.2.3 void SysCallAnalysisCode::spawn_analysis (const int & analysis id, const short block flag)

spawn a single analysis as part of a function evaluation.
Put a single analysis to the shell using the default syntax (no commandUsage support for analyses). This
function is used when multiple analysis drivers are spread between processors. Use of nonblocking shells
is supported in this fn, although its use is currently prevented externally.

6.81.2.4 void SysCallAnalysisCode::spawn_output_filter (const short block flag)

spawn the output filter portion of a function evaluation.
Put the output filter to the shell. This function is used when multiple analysis drivers are spread between
processors. No need to check for a Null output filter, as this is checked externally. Use of nonblocking
shells is supported in this fn, although its use is currently prevented externally.

The documentation for this class was generated from the following files:

- SysCallAnalysisCode.H
- SysCallAnalysisCode.C
6.82 SysCallApplicInterface Class Reference

Derived application interface class which spawns simulation codes using system calls.

```cpp
#include <SysCallApplicInterface.H>
```

Inheritance diagram for SysCallApplicInterface::

```
    DakotaInterface
       |
    ApplicationInterface
       |
   SysCallApplicInterface
```

Public Methods

- **SysCallApplicInterface** (const `ProblemDescDB` &`problem_db`, const `size_t` &`num_fns`)  
  *constructor.*

- **~SysCallApplicInterface** ()  
  *destructor.*

- **void derived_map** (const `DakotaVariables` &`vars`, const `DakotaIntArray` &`asv`, `DakotaResponse` &`response`, int `fn`, `eval_id`)  
  Called by `map()` and other functions to execute the simulation in synchronous mode. The portion of performing an evaluation that is specific to a derived class.

- **void derived_map_asynch** (const `ParamResponsePair` &`pair`)  
  Called by `map()` and other functions to execute the simulation in asynchronous mode. The portion of performing an asynchronous evaluation that is specific to a derived class.

- **void derived_sync** (`DakotaList<ParamResponsePair>` &`prp_list`)  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version waits for at least one completion.

- **void derived_sync_nowait** (`DakotaList<ParamResponsePair>` &`prp_list`)  
  For asynchronous function evaluations, this method is used to detect completion of jobs and process their results. It provides the processing code that is specific to derived classes. This version is nonblocking and will return without any completions if none are immediately available.

- **int derived_synchronous_local_analysis** (const `int` &`analysis_id`)  
  Execute a particular analysis (identified by `analysis_id`) synchronously on the local processor. Used for the derived class specifics within `ApplicationInterface::serve_analyses_sync()`.

---

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
Private Methods

- void spawn\_application (const short block\_flag)
  
  Spawn the application by managing the input filter, analysis drivers, and output filter. Called from derived\_map() & derived\_map\_async().

- void derived\_synch\_kernel (DakotaList<ParamResponsePair> &prp\_list)
  
  Convenience function for common code between derived\_synch() & derived\_synch\_nowait().

- short system\_call\_file\_test (const DakotaString &root\_file)
  
  detect completion of a function evaluation through existence of the necessary results file(s).

Private Attributes

- SysCallAnalysisCode sysCallSimulator
  
  SysCallAnalysisCode provides convenience functions for passing the input filter, the analysis drivers, and the output filter to a CommandShell in various combinations.

- DakotaIntList sysCallList
  
  list of function evaluation id’s for active asynchronous system call evaluations.

- DakotaIntList failIdList
  
  list of function evaluation id’s for tracking response file read failures.

- DakotaIntList failCountList
  
  list containing the number of response read failures for each function evaluation identified in failIdList.

6.82.1 Detailed Description

Derived application interface class which spawns simulation codes using system calls.

SysCallApplicInterface uses a SysCallAnalysisCode object for performing simulation invocations.

The documentation for this class was generated from the following files:

- SysCallApplicInterface.H
- SysCallApplicInterface.C
6.83 TaylorSurf Class Reference

Derived approximation class for 1st order Taylor series (local approximation).

```cpp
#include <TaylorSurf.H>
```

Inheritance diagram for TaylorSurf:

```
DakotaApproximation
  └── TaylorSurf
```

### Public Methods

- **TaylorSurf** (const ProblemDescDB &problem\_db)
  
  * constructor.

- **~TaylorSurf** ()
  
  * destructor.

### Protected Methods

- void **find\_coefficients** ()
  
  * calculate the data fit coefficients using the currentPoints list of SurrogateDataPoints.

- int **required\_samples** (int num\_vars)
  
  * return the minimum number of samples required to build the derived class approximation type in num\_vars dimensions.

- Real **get\_value** (const DakotaRealVector &x)
  
  * retrieve the approximate function value for a given parameter vector.

- const DakotaRealVector & **get\_gradient** (const DakotaRealVector &x)
  
  * retrieve the approximate function gradient for a given parameter vector.

### 6.83.1 Detailed Description

Derived approximation class for 1st order Taylor series (local approximation).

The TaylorSurf class provides a local approximation based on data from a single point in parameter space. It uses a first order Taylor series expansion:

\[
f(x) = f(x_0) + \nabla f(x_0) \cdot (x - x_0)
\]

The documentation for this class was generated from the following files:
- TaylorSurf.H
- TaylorSurf.C
### 6.84 VariablesUtil Class Reference

Utility class for the DakotaVariables and DakotaVarConstraints hierarchies which provides convenience functions for variable vectors and label arrays for combining design, uncertain, and state variable types and merging continuous and discrete variable domains.

```cpp
#include <VariablesUtil.H>
```

Inheritance diagram for VariablesUtil::

```
VariablesUtil
```

#### Public Methods

- **VariablesUtil ()**
  
  *constructor.*

- **~VariablesUtil ()**
  
  *destructor.*

#### Protected Methods

- void **update_merger** (const DakotaRealVector &c_array, const DakotaIntVector &d_array, DakotaRealVector &m_array)
  
  *combine a continuous array and a discrete array into a single continuous array through promotion of integers to reals (merged view).*

- void **update_all_continuous** (const DakotaRealVector &c1_array, const DakotaRealVector &c2_array, const DakotaRealVector &c3_array, DakotaRealVector &all_array) const
  
  *combine 3 continuous arrays (design, uncertain, state) into a single continuous array (all view).*

- void **update_all_discrete** (const DakotaIntVector &d1_array, const DakotaIntVector &d2_array, DakotaIntVector &all_array) const
  
  *combine 2 discrete arrays (design, state) into a single discrete array (all view).*

- void **update_labels** (const DakotaStringArray &l1_array, const DakotaStringArray &l2_array, DakotaStringArray &all_array)
  
  *combine 2 label arrays into a single label array (merged or all views).*

- void **update_labels** (const DakotaStringArray &l1_array, const DakotaStringArray &l2_array, const DakotaStringArray &l3_array, DakotaStringArray &all_array)
  
  *combine 3 label arrays (design, uncertain, state) into a single label array (all view).*
6.84 VariablesUtil Class Reference

6.84.1 Detailed Description

Utility class for the DakotaVariables and DakotaVarConstraints hierarchies which provides convenience functions for variable vectors and label arrays for combining design, uncertain, and state variable types and merging continuous and discrete variable domains.

Derived classes within the DakotaVariables and DakotaVarConstraints hierarchies use multiple inheritance to inherit these utilities.

The documentation for this class was generated from the following file:

- VariablesUtil.H
Chapter 7

DAKOTA File Documentation

7.1  keywordtable.C File Reference

file containing keywords for the strategy, method, variables, interface, and responses input specifications from dakota.input.spec.

Variables

- const struct KeywordHandler idrKeywordTable []

  Initialize the keyword table as a vector of KeywordHandler structures (KeywordHandler declared in idr-keyword.h). A null KeywordHandler structure signifies the end of the keyword table.

7.1.1  Detailed Description

file containing keywords for the strategy, method, variables, interface, and responses input specifications from dakota.input.spec.
7.2 main.C File Reference

file containing the main program for DAKOTA.

Functions

- void `manage_inputs` (int argc, char **argv, `ProblemDescDB` &problem_db, `CommandLineHandler` &cmd_handler)
  
  manage command line inputs, parse the input file, and populate the problem description database.

- void `manage_outputs` (ofstream &output_stream, ofstream &error_stream, const `ParallelLibrary` &parallel_lib)
  
  manage output streams if required.

- void `manage_restart` (`CommandLineHandler` &cmd_handler, `ParallelLibrary` &parallel_lib)
  
  manage the restart file(s).

- void `clean_up` (ofstream &output_stream, ofstream &error_stream, const `ParallelLibrary` &parallel_lib)
  
  close output streams.

- int `main` (int argc, char *argv[])
  
  The main DAKOTA program.

Variables

- `DakotaList < ParamResponsePair > data_pairs`

  list of all param/response pairs.

- `DakotaBoStream * write_restart`

  the restart binary output stream.

- ofstream `write_ostream`

  Rogue Wave requirement for binary streams.

- int `write_precision = 10`

  used in ostream data output fn's.

7.2.1 Detailed Description

file containing the main program for DAKOTA.
### 7.2 Function Documentation

#### 7.2.2.1 `void manage_inputs (int argc, char ** argv, ProblemDescDB & problem_db, CommandLineHandler & cmdline_handler)`

manage command line inputs, parse the input file, and populate the problem description database.

Manage command line inputs using the `CommandLineHandler` class and parse the input file using the Input Deck Reader (IDR) parsing system. IDR populates the `ProblemDescDB` object with the input file data.

#### 7.2.2.2 `void manage_outputs (ofstream & output ofstream, ofstream & error ofstream, const ParallelLibrary & parallel_lib)`

manage output streams if required.

If concurrent iterators are to be used, create and tag multiple output streams in order to prevent jumbled output. This uses the CommonIO facility from UTILIB.

#### 7.2.2.3 `void manage_restart (CommandLineHandler & cmdline_handler, ParallelLibrary & parallel_lib)`

manage the restart file(s).

Manage restart file(s) by processing any incoming evaluations from an old restart file and by setting up the binary output stream for new evaluations.

#### 7.2.2.4 `void clean_up (ofstream & output ofstream, ofstream & error ofstream, const ParallelLibrary & parallel_lib)`

close output streams.

Close streams associated with `manage_inputs`, `manage_outputs`, and `manage_restart`.

#### 7.2.2.5 `int main (int argc, char * argv[])`

The main DAKOTA program.

Manage command line inputs, input files, restart file(s), output streams, and top level parallel iterator communicators. Instantiate the `DakotaStrategy` and invoke its `runStrategy()` virtual function.
7.3 restart_util.C File Reference

file containing the DAKOTA restart utility main program.

Functions

- void `print_restart` (int argc, char **argv, DakotaString print_dest)
  
  `print a restart file`.

- void `print_restart_tabular` (int argc, char **argv, DakotaString print_dest)
  
  `print a restart file (tabular format)`.

- void `read_neutral` (int argc, char **argv)
  
  `read a restart file (neutral file format)`.

- void `repair_restart` (int argc, char **argv)
  
  `repair a restart file by removing corrupted evaluations`.

- void `concatenate_restart` (int argc, char **argv)
  
  `concatenate multiple restart files`.

- int `main` (int argc, char *argv[])
  
  `The main program for the DAKOTA restart utility`.

Variables

- DakotaBoStream * `write_restart`
  
  `the restart binary output stream`.

- ofstream `write_oiostream`
  
  `Rogue Wave requirement for binary streams`.

- int `write_precision` = 16
  
  `used in ostream output fns. DAKOTA's main.C sets this to 10, however "print" outputs parameters in full precision (16 digits for double)`.

7.3.1 Detailed Description

file containing the DAKOTA restart utility main program.

7.3.2 Function Documentation
7.3.2.1  void print_restart (int argc, char ** argv, DakotaString print_dest)

print a restart file.
**Usage:** "dakota_restart_util print dakota rst"
"dakota_restart_util to_neutral dakota rst dakota neu"

Prints all evals. in full precision to either stdout or a neutral file. The former is useful for ensuring that
duplicate detection is successful in a restarted run (e.g., starting a new method from the previous best), and
the latter is used for translating binary files between platforms.

7.3.2.2  void print_restart_tabular (int argc, char ** argv, DakotaString print_dest)

print a restart file (tabular format).
**Usage:** "dakota_restart_util to_pdb dakota rst dakota pdb"
"dakota_restart_util to_tabular dakota rst dakota txt"

Unrolls all data associated with a particular tag for all evaluations and then writes this data in a tabular
format (e.g., to a PDB database or MATLAB/TECPLLOT data file).

7.3.2.3  void read_neutral (int argc, char ** argv)

read a restart file (neutral file format).
**Usage:** "dakota_restart_util from_neutral dakota neu dakota rst"

Reads evaluations from a neutral file. This is used for translating binary files between platforms.

7.3.2.4  void repair_restart (int argc, char ** argv)

repair a restart file by removing corrupted evaluations.
**Usage:** "dakota_restart_util remove 0.0 dakota old rst dakota new rst"

Removes all evals for which one of the function values matches argv[2].

7.3.2.5  void concatenate_restart (int argc, char ** argv)

concatenate multiple restart files.
**Usage:** "dakota_restart_util cat dakota 1 rst ... dakota n rst dakota new rst"

Combines multiple restart files into a single restart database.

7.3.2.6  int main (int argc, char * argv[])

The main program for the DAKOTA restart utility.
Parse command line inputs and invoke the appropriate utility function (print_restart(),
print_restart_tabular(), read_neutral(), repair_restart(), or concatenate_restart()).
Chapter 8

Recommended Practices for DAKOTA Development

8.1 Introduction

Common code development practices can be extremely useful in multiple developer environments. Particular styles for code components lead to improved readability of the code and can provide important visual cues to other developers.

Much of this recommended practices document is borrowed from the CUBIT mesh generation project, which in turn borrows its recommended practices from other projects. As a result, C++ coding styles are fairly standard across a variety of Sandia software projects in the engineering and computational sciences.

8.2 Style Guidelines

Style guidelines involve the ability to discern at a glance the type and scope of a variable or function.

8.2.1 Class and variable styles

Class names should be composed of two or more descriptive words, with the first character of each word capitalized, e.g.:

    class ClassName;

Class member variables should be composed of two or more descriptive words, with the first character of the second and succeeding words capitalized, e.g.:

    double classMemberVariable;
Temporary (i.e. local) variables are lower case, with underscores separating words in a multiple word temporary variable, e.g.:

```c
int temporary_variable;
```

Constants (i.e. parameters) are upper case, with underscores separating words, e.g.:

```c
const double CONSTANT_VALUE;
```

### 8.2.2 Function styles

Function names are lower case, with underscores separating words, e.g.:

```c
int function_name();
```

There is no need to distinguish between member and non-member functions by style, as this distinction is usually clear by context. This style convention arose from the desire to have member functions which set and return the value of a private member variable, e.g.:

```c
int memberVariable;
void member_variable(int a) { // set
    memberVariable = a;
}
int member_variable() const { // get
    return memberVariable;
}
```

In cases where the data to be set or returned is more than a few bytes, it is highly desirable to employ const references to avoid unnecessary copying, e.g.:

```c
void continuous_variables(const DakotaRealVector& c_vars) { // set
    continuousVariables = c_vars;
}
const DakotaRealVector& continuous_variables() const { // get
    return continuousVariables;
}
```

Note that it is not necessary to always accept the returned data as a const reference. If it is desired to be able change this data, then accepting the result as a new variable will generate a copy, e.g.:

```c
const DakotaRealVector& c_vars = model.continuous_variables(); // reference to continuousVariables cannot be changed
DakotaRealVector c_vars = model.continuous_variables(); // local copy of continuousVariables can be changed
```

### 8.2.3 Miscellaneous

Appearance of typedefs to redefine or alias basic types is isolated to a few header files (data_types.h, template_defs.h), so that issues like program precision can be changed by changing a few lines of typedefs rather than many lines of code, e.g.:
typedef double Real;

xemacs is the preferred source code editor, as it has C++ modes for enhancing readability through color (turn on "Syntax highlighting" in the Options pull down menu). Other helpful features include "Paren highlighting" for matching parentheses (Options pull down menu) and the "New Frame" utility (File pull down menu) to have more than one window operating on the same set of files (note that this is still the same edit session, so all windows are synchronized with each other). Window width should be set to 80 internal columns, which can be accomplished by manual resizing, or preferably, using the following alias in your shell resource file (e.g., .cshrc):

```
alias xemacs "xemacs -g 81x63"
```

where an X-window width of 81 gives 80 columns internal to the window and the desired height of the window will vary depending on monitor size. This window width imposes a coding standard since you should avoid line wrapping by continuing anything over 80 columns onto the next line.

Indenting increments are 2 spaces per indent and comments are aligned with the code they describe, e.g.:

```c
void abort_handler(int code)
{
    int initialized = 0;
    MPI_Initialized(&initialized);
    if (initialized) {
        // comment aligned to block it describes
        int size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (size>1)
            MPI_Abort(MPI_COMM_WORLD, code);
        else
            exit(code);
    }
    else
        exit(code);
}
```

Also, the continuation of a long command is indented 2 spaces, e.g.:

```c
const DakotaString& iterator_scheduling
    = problem_db.get_string("strategy.iterator_scheduling");
```

and similar lines are aligned for readability, e.g.:

```c
cout << "Numerical gradients using " << finiteDiffStepSize*100. << "%" << finiteDiffType << " differences\nto be calculated by the " << methodSource << " finite difference routine." << endl;
```

Lastly, ifdef’s are not indented (to make use of syntax highlighting in xemacs).

### 8.3 File Naming Conventions

In addition to the style outlined above, the following file naming conventions have been established for the DAKOTA project.
File names for C++ classes should be identical to the class name defined by that file. Exceptions: in some cases it is convenient to maintain several closely related classes in a single file, in which case the file name may reflect the top level class (e.g., DakotaResponse.C/.H files contain DakotaResponse and DakotaResponseRep classes) or some generalization of the set of classes (e.g., DakotaBinStream.C/.H files contain the DakotaBiStream/DakotaBoStream classes for binary input and binary output).

The type of file is determined by one of the four file name extensions listed below:

- **.H** A class header file ends in the suffix .H. The header file provides the class declaration. This file does not contain code for implementing the methods, except for the case of inline functions. Inline functions are to be placed at the bottom of the file with the keyword inline preceding the function name.
- **.C** A class implementation file ends in the suffix .C. An implementation file contains the definitions of the members of the class.
- **.h** A header file ends in the suffix .h. The header file contains information usually associated with procedures. Defined constants, data structures and function prototypes are typical elements of this file.
- **.c** A procedure file ends in the suffix .c. The procedure file contains the actual procedures.

### 8.4 Class Documentation Conventions

Class documentation uses the doxygen tool available from [http://www.doxygen.org](http://www.doxygen.org) and employs the JAVA-doc comment style. Brief comments appear in header files next to the attribute or function declaration. Detailed descriptions for functions should appear alongside their implementations (i.e., in the .C files for non-inlined, or in the headers next to the function definition for inlined). Detailed comments for a class or a class attribute must go in the header file as this is the only option.

NOTE: Previous class documentation utilities (class2frame and class2html) used the "//-" comment style and comment blocks such as this:

```
//- Class: DakotaModel
//- Description: The model to be iterated. Contains DakotaVariables, DakotaInterface, and DakotaResponse objects.
//- Owner: Mike Eldred
```

These tools are no longer used, so remaining comment blocks of this type are informational only and will not appear in the documentation generated by doxygen.
Chapter 9

Instructions for Modifying DAKOTA’s Input Specification

9.1 Modify dakota.input.spec

The master input specification resides in $DAKOTA/src/dakota.input.spec. As part of the Input Deck Reader (IDR) build process, a soft link to this file is created in $DAKOTA/VendorPackages/idr. The master input specification can be modified with the addition of new constructs using the following logical relationships:

- {} for required individual specifications
- () for required group specifications
- [] for optional individual specifications
- [ ] for optional group specifications
- | for “or” conditionals

These constructs can be used to define a variety of dependency relationships in the input specification. It is recommended that you review the existing specification and have an understanding of the constructs in use before attempting to add new constructs.

Warning:
- Do not skip this step. Attempts to modify the keywordtable.C and ProblemDescDB.C files in $DAKOTA/src without reference to the results of the code generator are very error-prone. Moreover, the input specification provides a reference to the allowable inputs of a particular executable and should be kept in sync with the parser files (modifying the parser files independent of the input specification creates, at a minimum, undocumented features).
- Since the Input Deck Reader (IDR) parser allows abbreviation of keywords, you must avoid adding a keyword that could be misinterpreted as an abbreviation for a different keyword within the same keyword handler (the term "keyword handler" refers to the strategy kwhandler(), method kwhandler(), variables kwhandler(), interface kwhandler(), and responses kwhandler() member functions in the ProblemDescDB class). For example, adding the keyword "expansion" within the method specification would be a mistake if the keyword "expansion factor" already was being used in this specification.
Since IDR input is order-independent, the same keyword may be reused multiple times in the specification if and only if the specification blocks are mutually exclusive. For example, method selections (e.g., dot_frcg, dot_bfgs) can reuse the same method setting keywords (e.g., optimization_type) since the method selection blocks are all separated by logical "or"s. If dot_frcg and dot_bfgs were not exclusive and could be specified at the same time, then association of the optimization_type setting with a particular method would be ambiguous. This is the reason why repeated specifications which are non-exclusive must be made unique, typically with a prepended identifier (e.g., cdrv_initial_point, ddrv_initial_point).

### 9.2 Rebuild IDR

```bash
cd $DAKOTA/VendorPackages/idr
make clean
make
```


### 9.3 Update keywordtable.C in $DAKOTA/src

Do not directly replace the `keywordtable.C` in `$DAKOTA/src` using the one from `idr`, as there are important differences in the kwhandler bindings. Rather, update the `keywordtable.C` in `$DAKOTA/src` using the one from `idr` as a reference. Once this step is completed, it is a good idea to verify the match by diff'ing the 2 files. The only differences should be in comments, includes, and kwhandler declarations.

### 9.4 Update ProblemDescDB.C in $DAKOTA/src

Find the keyword handler functions (e.g., `variables_kwhandler()`) in `$DAKOTA/VendorPackages/idr/<canonical_build_directory>/idr-gen-code.C` and `$DAKOTA/src/ProblemDescDB.C` which correspond to your modifications to the input specification. The `idr-gen-code.C` file is the result of a code generator and contains skeleton constructs for extracting data from IDR. You will be copying over parts of this skeleton to `ProblemDescDB.C` and then adding code to populate attributes within Data class container objects.

#### 9.4.1 Replace keyword handler declarations and counter loop

Rather than trying to update these line by line, it is recommended to delete the entire block starting with the keyword declarations and ending at the bottom of the keyword counter loop. The declarations assign -1 to keywords and look like this:
9.4 Update ProblemDescDB.C in $DAKOTA/src

Int cdv_descrip = -1;
Int cdv_initial_point = -1;

They start after the line "Int cntr;". The keyword counter loop looks like this:

```
for ( cntr=data_len; cntr--; ) {
    if ( idr_find_id( &cdv_descrip, cntr,
                    "cdv_descrip", id_str, kw_str ) ) continue;
    ...
    if ( idr_find_id( &wuv_dist_upper_bounds, cntr,
                     "wuv_dist_upper_bounds", id_str, kw_str ) ) continue;
}
```

Once the old keyword declarations and keyword counter loop have been deleted, replace them with the corresponding blocks from idr-gen-code.C containing the updated keyword declarations and counter loop.

9.4.2 Update keyword handler logic blocks

For the newly added or modified input specifications, copy the appropriate skeleton constructs from idr-gen-code.C and paste them into the corresponding location in ProblemDescDB.C.

The next step is to add code to these skeletons to set data attributes within the Data class object used by the keyword handler. At the top of each keyword handler, a Data class object is instantiated, e.g.:

```
DataMethod data_method;
```

Each data class is a simple container class which contains the data from a single keyword handler invocation. Within each skeleton construct, you will extract data from the IDR data structures and then use this data to set the corresponding attribute within the Data class.

Integer, real, and string data are extracted using the idata, rdata, and cdata arrays provided by IDR. These arrays are indexed using a bracket operator with the keyword as an index.

Lists of integer and real data are extracted using the idr_table constructs provided by IDR. Unfortunately, IDR does not provide an idr_table for string data, so these extractions are more involved. Refer to existing <LISTof><STRING> extractions for use as a model.

**Example 1:** if you added the specification:

```
[method_setting = <REAL>]
```

you would copy over

```
if ( method_setting >= 0 ) {
}
```

from idr-gen-code.C into ProblemDescDB.C and then populate the if block with a call to set the corresponding attribute within the data_method object using data extracted using the rdata array:

```
if ( method_setting >= 0 ) {
    data_method.methodSetting = rdata[method_setting];
}
```
Use of a set member function within DataMethod is not needed since the data is public. The data is public since ProblemDescDB already provides sufficient encapsulation (ProblemDescDB::methodList, ProblemDescDB::variablesList, ProblemDescDB::interfaceList, and ProblemDescDB::responsesList are private attributes), and no other classes have direct access. A similar model is used with SurrogateDataPoint objects contained in DakotaApproximation (DakotaApproximation::currentPoints) and with Parallelism-Level objects contained in ParallelLibrary (ParallelLibrary::parallelismLevels). Allowing public access to the Data class attributes is essentially equivalent to declaring ProblemDescDB a friend, but with the important bonus of a significant reduction in the amount of code to maintain. That is, the Data classes can be streamlined (and the work in modifying the input specification can be reduced) by omitting set/get functions.

Example 2: if you added the specification

[method_setting = <LISTof><REAL>]

you would copy over

    if ( method_setting >= 0 ) {
        Int idr_table_len;
        Real** idr_table = idr_get_real_table( parsed_data, method_setting,
                                idr_table_len, 1, 1 );
    }
}

from idr-gen-code.C into ProblemDescDB.C and then populate it with a loop which extracts each entry of the table and populates the corresponding attribute within the dataMethod object. The idr_table_len attribute is used for the loop limit and to size the dataMethod object.

    if ( method_setting >= 0 ) {
        Int idr_table_len;
        Real** idr_table = idr_get_real_table( parsed_data, method_setting,
                                idr_table_len, 1, 1 );

        data_method.methodSetting.reshape(idr_table_len);
        for (int i = 0; i<idr_table_len; i++)
            data_method.methodSetting[i] = idr_table[0][i];
    }
}

Attention:
If no new data attributes have been added, but instead there are only new settings for existing attributes, then you’re done with the database augmentation at this point (you just need to add code to use these new settings in the places where the existing attributes are used).

9.4.3 Augment/update get_<data_type>() functions

The final update step for ProblemDescDB.C involves extending the database retrieval functions. These retrieval functions accept an identifier string and return a database attribute of a particular type, e.g. a DakotaRealVector:

    const DakotaRealVector& get drv(const DakotaStrings& entry_name);

The implementation of each of these functions has a simple series of if-else checks which return the appropriate attribute based on the identifier string. For example,
9.5 Update Corresponding Data Classes

In this step, we extend the Data class definitions to include the new attributes referenced in Update keyword handler logic blocks and Augment/update get_<data_type>() functions.

9.5.1 Update the Data class header file

Add a new attribute to the private data for each of the new specifications. Follow the style guide for class attributes (or mimic the existing code).

9.5.2 Update the .C file

Define defaults for the new attributes in the constructor initialization list (or in the case of DataMethod, in the body of the constructor for readability). Add the new attributes to the assign() function for use by the copy constructor and assignment operator. Add the new attributes to the write(PackBuffer&), read(UnPackBuffer&), and write(ostream&) functions, paying attention to using a consistent ordering.
In the case of a strategy specification change, there is no DataStrategy container class since only one strategy specification is allowed and there is no need for a list of data objects. Rather, in this case, it is necessary to add static attributes to ProblemDescDB.H, add their initialization to the top of ProblemDescDB.C, and add them to the ProblemDescDB::sendObjBuffer() and ProblemDescDB::receiveObjBuffer() buffer insertion/extraction functions (again, paying attention to using a consistent ordering).

9.6 Use get_<data_type>() Functions

At this point, the new specifications have been mapped through all of the database classes. The only remaining step is to retrieve the new data within the constructors of the classes that need it. This is done by invoking the get_<data_type>() function on the ProblemDescDB object using the identifier string you selected in Augment/update get_<data_type>() functions. For example, from DakotaModel.C:

```cpp
const DakotaStrings interface_type = problem_db.get_string("interface.type");
```

passes the "interface.type" identifier string to the ProblemDescDB::get_string() retrieval function, which returns the desired attribute from the active DataInterface object.

**Warning:**

Use of the get_<data_type>() functions is restricted to class constructors, since only in class constructors are the data list indices (i.e., methodIndex, interfaceIndex, variablesIndex, and responsesIndex) guaranteed to be set correctly. Outside of the constructors, the database list nodes will correspond to the last set operation, and may not return data from the desired list node.

9.7 Update the Documentation

Doxygen comments should be added to the Data class headers for the new attributes, and the reference manual sections describing the portions of dakota.input.spec that have been modified should be updated.
Index

~ANNSurf
  ANNNsurf, 46
~AllMergedVarConstraints
  AllMergedVarConstraints, 29
~AllMergedVariables
  AllMergedVariables, 32
~AllVarConstraints
  AllVarConstraints, 36
~AllVariables
  AllVariables, 39
~AnalysisCode
  AnalysisCode, 44
~ApplicationInterface
  ApplicationInterface, 48
~ApproximationInterface
  ApproximationInterface, 58
~BranchBndStrategy
  BranchBndStrategy, 62
~CONMINOptimizer
  CONMINOptimizer, 69
~CommandLineHandler
  CommandLineHandler, 64
~CommandShell
  CommandShell, 65
~ConcurrentStrategy
  ConcurrentStrategy, 67
~CtelRegexp
  CtelRegexp, 76
~DACEIterator
  DACEIterator, 78
~DOTOptimizer
  DOTOptimizer, 194
~DakotaApproximation
  DakotaApproximation, 85
~DakotaArray
  DakotaArray, 87
~DakotaBaseVector
  DakotaBaseVector, 90
~DakotaBiStream
  DakotaBiStream, 95
~DakotaBoStream
  DakotaBoStream, 96
~DakotaGraphics
  DakotaGraphics, 99
~DakotaInterface
  DakotaInterface, 104
~DakotaIterator
  DakotaIterator, 111
~DakotaList
  DakotaList, 113
~DakotaMatrix
  DakotaMatrix, 117
~DakotaModel
  DakotaModel, 127
~DakotaNonD
  DakotaNonD, 130
~DakotaOptimizer
  DakotaOptimizer, 133
~DakotaResponse
  DakotaResponse, 138
~DakotaResponseRep
  DakotaResponseRep, 142
~DakotaStrategy
  DakotaStrategy, 150
~DakotaString
  DakotaString, 152
~DakotaVarConstraints
  DakotaVarConstraints, 158
~DakotaVariables
  DakotaVariables, 164
~DakotaVector
  DakotaVector, 166
~DataInterface
  DataInterface, 170
~DataMethod
  DataMethod, 174
~DataResponses
  DataResponses, 181
~DataVariables
  DataVariables, 184
~DirectFnApplicInterface
  DirectFnApplicInterface, 190
~ForkAnalysisCode
  ForkAnalysisCode, 199
~ForkApplicInterface
  ForkApplicInterface, 201
~FundamentalVarConstraints
  FundamentalVarConstraints, 205
~FundamentalVariables
  FundamentalVariables, 208
<table>
<thead>
<tr>
<th>Class/Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllMergedVariables</td>
<td>33</td>
</tr>
<tr>
<td>AllVariables</td>
<td>40</td>
</tr>
<tr>
<td>DakotaVariables</td>
<td>161</td>
</tr>
<tr>
<td>FundamentalVariables</td>
<td>209</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>242</td>
</tr>
<tr>
<td>all_continuous_variables</td>
<td>33</td>
</tr>
<tr>
<td>all_continuous_variables</td>
<td>40</td>
</tr>
<tr>
<td>all_discrete_variables</td>
<td>161</td>
</tr>
<tr>
<td>all_discrete_variables</td>
<td>209</td>
</tr>
<tr>
<td>MergedVariables</td>
<td>242</td>
</tr>
<tr>
<td>all_responses</td>
<td>108</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>all_variables</td>
<td>41</td>
</tr>
<tr>
<td>allContinuousLabels</td>
<td>41</td>
</tr>
<tr>
<td>allContinuousLowerBnds</td>
<td>37</td>
</tr>
<tr>
<td>allContinuousUpperBnds</td>
<td>37</td>
</tr>
<tr>
<td>allContinuousVars</td>
<td>37</td>
</tr>
<tr>
<td>allDataFlag</td>
<td>41</td>
</tr>
<tr>
<td>allDiscreteLabels</td>
<td>41</td>
</tr>
<tr>
<td>allDiscreteLowerBnds</td>
<td>37</td>
</tr>
<tr>
<td>allDiscreteUpperBnds</td>
<td>37</td>
</tr>
<tr>
<td>allDiscreteVars</td>
<td>37</td>
</tr>
<tr>
<td>allMergedLabels</td>
<td>34</td>
</tr>
<tr>
<td>allMergedLabels</td>
<td>41</td>
</tr>
<tr>
<td>allMergedLowerBnds</td>
<td>30</td>
</tr>
<tr>
<td>allMergedUpperBnds</td>
<td>30</td>
</tr>
<tr>
<td>allMergedVarConstraints</td>
<td>34</td>
</tr>
<tr>
<td>allMergedVarConstraints</td>
<td>30</td>
</tr>
<tr>
<td>allMergedVarConstraints</td>
<td>30</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>69</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>AllMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
<tr>
<td>AllVarConstraints</td>
<td>36</td>
</tr>
<tr>
<td>allMergedVariables</td>
<td>35</td>
</tr>
<tr>
<td>allMergedVars</td>
<td>34</td>
</tr>
<tr>
<td>allocate_workspace</td>
<td>36</td>
</tr>
<tr>
<td>CONMINOptimizer</td>
<td>69</td>
</tr>
<tr>
<td>DOTOptimizer</td>
<td>194</td>
</tr>
<tr>
<td>NPSOLOptimizer</td>
<td>276</td>
</tr>
<tr>
<td>allResponses</td>
<td>110</td>
</tr>
<tr>
<td>DakotaIterator</td>
<td>108</td>
</tr>
</tbody>
</table>
discrete_lower_bounds, 36
discrete_upper_bounds, 36
numCDV, 37
numCSV, 37
numDDV, 37
numDSV, 37
numUV, 37
read, 37
write, 36
AllVarConstraints, 36
AllVariables, 36
AllVarConstraints, 38
AllVariables
～AllVariables, 39
acv, 40
adv, 40
all_continuous_variables, 40
all_discrete_variables, 40
allContinuousLabels, 41
allContinuousVars, 41
allDiscreteLabels, 41
allDiscreteVars, 41
AllVariables, 39, 42
continuous_variable_labels, 39, 40
continuous_variables, 39
copy_rep, 41
cv, 39
discrete_variable_labels, 40
discrete_variables, 39
dv, 39
emptyIntVector, 41
emptyRealVector, 41
inactive_continuous_variables, 40
inactive_discrete_variables, 40
numCDV, 41
numCSV, 41
numDDV, 41
numDSV, 41
numUV, 41
operator==, 42
read, 40, 41
read.annotated, 40
tv, 39
write, 40, 41
write.annotated, 40
AllVariables, 39
AllVariables, 42
allVariables
DakotaIter, 110
allVarsFlag
DataMethod, 179
NonDProbability, 268
NonDSampling, 272
ALPHAX
KrigApprox, 224
amvFlag
NonDAdvMeanValue, 258
analysis_communicator_rank
ParallelLibrary, 282
analysis_communicator_size
ParallelLibrary, 282
analysis_intra_communicator
ParallelLibrary, 282
analysis_master_flag
ParallelLibrary, 282
analysis_server_id
ParallelLibrary, 282
analysis_servers
ParallelLibrary, 282
AnalysisCode
～AnalysisCode, 44
AnalysisCode, 44
apreproFlag, 44
define_filenames, 43
fileNameKey, 45
fileSaveFlag, 44
fileTagFlag, 44
iFilterName, 44
input_filter_name, 43
modified_parameters_filename, 43
modified_results_filename, 43
modifiedParamsFileName, 44
modifiedResFileName, 45
numPrograms, 44
oFilterName, 44
output_filter_name, 43
parallelLib, 45
parametersFileName, 44
parametersFNameList, 45
program_names, 43
programNames, 44
quiet_flag, 43, 44
quietFlag, 44
read_results_file, 43
results_filename, 43
resultsFileName, 44
resultsFNameList, 45
verboseFlag, 44
write_parameters_file, 43
AnalysisCode, 43
analysisComm
ApplicationInterface, 51
analysisCommRank
ApplicationInterface, 50
ParallelLibrary, 286
analysisCommSize
ApplicationInterface, 50
ParallelLibrary, 286
analysisDrivers
ApplicationInterface, 51
DataInterface, 171
analysisIntraComm
ParallelMasterLibrary, 285
analysisMasterFlag
ParallelLibrary, 285
analysisMessagePass
ApplicationInterface, 50
analysisScheduling
ApplicationInterface, 52
DataInterface, 172
analysisServerId
ApplicationInterface, 50
ParallelLibrary, 286
analysisServers
DataInterface, 172
analysisUsage
DataInterface, 171
annObject
ANNSurf, 46
ANNSurf, 46
~ANNSurf, 46
annObject, 46
ANNSurf, 46
find_coefficients, 46
get_value, 46
required_samples, 46
append
DakotaList, 114
ApplicationInterface
~ApplicationInterface, 48
analysisComm, 51
analysisCommRank, 50
analysisCommSize, 50
analysisDrivers, 51
analysisMessagePass, 50
analysisScheduling, 52
analysisServerId, 50
ApplicationInterface
asvControl, 53
asynch_local_evaluation_concurrency, 48
asynchLocalAnalysisConcurrency, 50
asynchLocalAnalysisFlag, 50
asynchLocalEvalConcurrency, 52
beforeSynchDuplicateIds, 53
beforeSynchDuplicateIndices, 53
beforeSynchDuplicateResponses, 53
beforeSynchPRPList, 53
clear_bookkeeping, 49
continuation, 52
defaultASV, 53
derived_map, 49
derived_map_asynch, 49
derived_synch, 49
derived_synch_nowait, 49
derived_synchronous_local_analysis, 49
evalComm, 51
evalCommRank, 50
evalCommSize, 50
evalDedMasterFlag, 50
evalMessagePass, 50
evalScheduling, 52
evalServerId, 50
failAction, 53
failRecoveryFnVals, 53
failRetryLimit, 53
free_communicators, 48
get_source_pair, 52
headerFlag, 53
historyDuplicateIds, 53
historyDuplicateResponses, 53
init_communicators, 48
interface_synchronization, 48
interfaceSynchronization, 52
iteratorCommRank, 50
iteratorCommSize, 50
lenPRPMessage, 51
lenResponseMessage, 51
lenVarsASVMessage, 51
lenVarsMessage, 51
manage_failure, 48
multiProcAnalysisFlag, 51
numAnalysisDrivers, 51
numAnalysisServers, 51
numEvalServers, 52
parallelLib, 49
procsPerAnalysis, 52
runningList, 53
suppressOutput, 50
worldRank, 50
worldSize, 50
ApplicationInterface, 48
asynchronous_local_evaluations, 56
asynchronous_local_evaluations_nowait, 56
duplication_detect, 55
map, 54
self_schedule_analyses, 55
self_schedule_evaluations, 55
serve_analyses_synch, 55
serve_evaluations, 54
serve_evaluations_async, 57
serve_evaluations_peer, 57
serve_evaluations_synch, 57
static_schedule_evaluations, 56
stop_evaluation_servers, 55
synch, 54
synch_nowait, 54
synchronous_local_evaluations, 56
apply_correction
  DakotaModel, 120
LayeredModel, 232
approxBuilds
  LayeredModel, 234
approxCorrection
  DataInterface, 173
approxDaceMethodPtr
  DataInterface, 173
approxGradUsageFlag
  DataInterface, 173
approximateModel
  SurrBasedOptStrategy, 319
ApproximationInterface
  approximationInterface, 58
approxOffset, 59
approxScale, 59
beforeSyncResponseList, 59
build_global_approximation, 58
build_local_approximation, 58
graphicsFlag, 59
map, 58
minimum_samples, 58
minSamples, 59
sampleReuse, 59
synch, 58
synch_nowait, 59
update_approximation, 58
ApproximationInterface, 58
actualInterfacePointer, 60
daceMethodPointer, 59
functionSurfaces, 60
approxInterface
  SurrLayeredModel, 324
approxLowerBounds
  DakotaApproximation, 84
approxOffset
  approximationInterface, 59
approxRep
  DakotaApproximation, 84
approxSampleReuse
  DataInterface, 173
approxScale
  approximationInterface, 59
approxType
  DakotaApproximation, 83
DataInterface, 172
approxUpperBounds
  DakotaApproximation, 84
aPPSOptimizer
  SGOPTOptimizer, 307
apreproFlag
  AnalysisCode, 44
DakotaVariables, 162
apreproFormatFlag
  DataInterface, 171
argC
  BranchBndStrategy, 63
argList
  ForkAnalysisCode, 199
argument_list
  ForkAnalysisCode, 199
argV
  BranchBndStrategy, 63
array_
  DakotaBaseVector, 91
assign
  DataInterface, 173
DataMethod, 180
DataResponses, 183
DataVariables, 188
asv_mapping
  NestedModel, 249
asvControl
  ApplicationInterface, 53
asvList
  DakotaModel, 125
asynch
  compute_response
  DakotaModel, 120, 121
asynch_flag
  CommandShell, 65
DakotaModel, 123
asynch_local_evaluation_concurrency
  ApplicationInterface, 48
DakotaInterface, 102
asynchEvalFlag
  DakotaModel, 125
asynchFDFlag
  DakotaModel, 125
asynchFlag
  CommandShell, 65
DakotaIterator, 110
asynchLocalAnalysisConcurrency
  ApplicationInterface, 50
DataInterface, 171
asynchLocalAnalysisFlag
  ApplicationInterface, 50
asynchLocalEvalConcurrency
  ApplicationInterface, 52
DataInterface, 171
asynchronous_local_analyses
  ForkApplicInterface, 203
asynchronous_local_evaluations
  ApplicationInterface, 56
asynchronous_local_evaluations_nowait
  ApplicationInterface, 56
augment_bounds
NPSOLOptimizer, 276
auto_correction
DakotaModel, 120
LayeredModel, 232
autoCorrection
LayeredModel, 234

B
CONMINOptimizer, 74
KrigApprox, 228
badScalingFlag
LayeredModel, 233
BaseConstructor
BaseConstructor, 61
baseOptimizer
SGOPTOptimizer, 306
batchSize
DataMethod, 178
bcast
ParallelLibrary, 280
bcfdnlf1
SNLLOptimizer, 314
bcnlf1
SNLLOptimizer, 314
bcnlf2
SNLLOptimizer, 314
beforeSyncDuplicateIds
ApplicationInterface, 53
beforeSyncDuplicateIndices
ApplicationInterface, 53
beforeSyncDuplicateResponses
ApplicationInterface, 53
beforeSyncIdList
DakotaInterface, 103
beforeSyncPRPList
ApplicationInterface, 53
beforeSyncResponseList
ApproximationInterface, 59
bestObjectiveFn
DACEIterator, 79
ParamStudy, 293
bestResponses
DACEIterator, 79
DakotaOptimizer, 135
ParamStudy, 293
bestVariables
DACEIterator, 79
DakotaOptimizer, 135
ParamStudy, 293
bestViolations
DACEIterator, 79

ParamStudy, 293
betaApproxCenterGrads
LayeredModel, 233
betaApproxCenterVals
LayeredModel, 233
betaCenterPt
LayeredModel, 233
betaFcnRatio
LayeredModel, 233
betaGradOffset
LayeredModel, 234
betaGrads
LayeredModel, 233
betaHat
KrigApprox, 225
betaHessOffset
LayeredModel, 234
bigBoundSize
DakotaOptimizer, 134
boundsArraySize
NPSOLOptimizer, 276
BranchBndStrategy
~BranchBndStrategy, 62
argC, 63
argV, 63
BranchBndStrategy, 62
numIteratorServers, 62
numNodeSamples, 62
numRootSamples, 62
picoComm, 62
picoCommRank, 63
picoCommSize, 63
picoListOfIntegers, 63
picoLowerBnds, 63
picoUpperBnds, 63
run_strategy, 62
selectedIterator, 62
userDefinedModel, 62
BranchBndStrategy, 62
build
DakotaApproximation, 82
build_approximation
DakotaModel, 119
HierLayeredModel, 218
SurrLayeredModel, 326
build_global_approximation
ApproximationInterface, 58
DakotaInterface, 102
build_label
ProblemDescDB, 297
buildLabels
ProblemDescDB, 297
build_local_approximation
ApproximationInterface, 58
DakotaInterface, 102

C

CONMINOptimizer, 74
KrigApprox, 228
centered_loop
ParamStudy, 292
centeredParam
DataMethod, 177
centralPath
DataMethod, 176
chaosCoeffs
HermiteSurf, 217
chaosSamples
HermiteSurf, 217
check_error
NonDSampling, 272
check_input
ProblemDescDB, 295
check_status
ForkAnalysisCode, 200
check_submodel_compatibility
LayeredModel, 232
check_usage
CommandLineHandler, 64
cholCorrMatrix
NonDAdvMeanValue, 257
cLambda
NPSOLOptimizer, 276
clean_up
main.C, 341
clear_bookkeeping
ApplicationInterface, 49
clearErrors
CtelRegexp, 76
coeffArray
NonDPCE, 264
comm_usage
SysCallAnalysisCode, 330
CommandLineHandler
~CommandLineHandler, 64
check_usage, 64
CommandLineHandler, 64
read_restart_evals, 64
read_restart_stream, 64
write_restart_stream, 64
CommandLineHandler, 64
CommandLineShell
~CommandLineShell, 65
async_flag, 65
asyncFlag, 65
CommandShell, 65
operator<<, 65
quiet_flag, 65
quietFlag, 65
unixCommand, 65
CommandShell, 65
flush, 66
commandUsage
SysCallAnalysisCode, 330
compile
CtelRegexp, 76
completionList
DakotaInterface, 103
NestedModel, 250
compute_correction
DakotaModel, 120
LayeredModel, 234
compute_penalty_function
SurrBasedOptStrategy, 322
compute_response
DakotaModel, 120
compute_vector_steps
ParamStudy, 291
concatenate_restart
restartutil.C, 343
ConcurrentStrategy
~ConcurrentStrategy, 67
ConcurrentStrategy, 67
iteratorParameterSets, 67
iteratorServerId, 68
numIteratorJobs, 67
numIteratorServers, 67
run_strategy, 67
selectedIterator, 67
strategyDedicatedMasterFlag, 67
userDefinedModel, 67
ConcurrentStrategy, 67
conmin_theta_lower_bnds
KrigApprox, 224
conmin_theta_upper_bnds
KrigApprox, 224
conmin_theta_vars
KrigApprox, 224
conminDesVars
CONMINOptimizer, 71
conminInfo
CONMINOptimizer, 72
KrigApprox, 223
conminLowerBnds
CONMINOptimizer, 71
CONMINOptimizer, 69
~CONMINOptimizer, 69
A, 75
allocate_workspace, 69
B, 74
C, 74
conminDesVars, 71
DakotaModel, 124

currentVariables
DakotaModel, 124
cv
AllMergedVariables, 32
AllVariables, 39
DakotaModel, 121
DakotaVariables, 160
FundamentalVariables, 208
MergedVariables, 241
cyl_head
DirectFnApplicInterface, 191

DABFUN
CONMINOptimizer, 71
KrigApprox, 223
daceCenterPtFlag
SurrBasedOptStrategy, 321
DACEIterator, 78
~DACEIterator, 78
allDataFlag, 79
bestObjectiveFn, 79
bestResponses, 79
bestVariables, 79
bestViolations, 79
DACEIterator, 80
daceMethod, 79
iterator_response_results, 78
iterator_variable_results, 78
multiObjWeights, 79
nonlinearEqTargets, 80
nonlinearIneqLowerBnds, 79
nonlinearIneqUpperBnds, 80
numNonlinearEqConstraints, 79
numNonlinearIneqConstraints, 79
numObjectiveFunctions, 79
numSamples, 79
numSymbols, 79
print_iterator_results, 78
randomSeed, 79
resolve_samples_symbols, 80
run_iterator, 80
sampling_reset, 78
sampling_scheme, 78
update_best, 78
daceIterator
SurrLayeredModel, 325
daceMethod
DACEIterator, 79
DataMethod, 176
daceMethodPointer
ApproximationInterface, 59
SurrLayeredModel, 324
dakota_asynch_flag

SGOPTApplication, 304
DakotaApproximation
add_point, 84
add_point_rebuild, 83
approxLowerBounds, 84
approxRep, 84
approxType, 83
approxUpperBounds, 84
build, 82
currentPoints, 83
DakotaApproximation, 84, 85
draw_surface, 83
find_coefficients, 82
gradient, 82
gradientFlag, 83
gradVector, 83
num_variables, 83
numCurrentPoints, 83
numSamples, 83
numVars, 83
referenceCount, 84
required_samples, 82
set_bounds, 83
DakotaApproximation, 82
~DakotaApproximation, 85
DakotaApproximation, 84, 85
get_approx, 85
operator=, 85
DakotaArray
~DakotaArray, 87
DakotaArray, 87, 88
length, 88
operator(), 87, 88
operator=, 88
reshape, 88
DakotaArray, 87
copy_array, 89
DakotaArray, 88
data, 89
operator T *, 89
operator=, 89
testClass, 89
DakotaBaseVector
~DakotaBaseVector, 90
array_, 91
DakotaBaseVector, 90, 91
length, 91
npts_, 91
operator()_, 91
operator=, 90
DakotaBaseVector, 90
DakotaBaseVector, 91
data, 92

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch (©) 1997-2001
probDescDB, 109
referenceCount, 110
sampling_reset, 108
sampling_scheme, 108
staticModel, 110
user_defined_model, 108
userDefinedModel, 109
verboseOutput, 110
DakotaIterator, 107
~DakotaIterator, 111
DakotaIterator, 111, 112
get_iterator, 112
operator=, 112
populate_gradient_vars, 112
run_iterator, 112
DakotaList
~DakotaList, 113
DakotaList, 113
entries, 113
isEmpty, 114
operator=, 113
DakotaList, 113
append, 114
contains, 115
find, 115
get, 114
index, 115, 116
insert, 115
occurrencesOf, 116
operator[], 116
remove, 115
removeAt, 115
sort, 115
testClass, 114
DakotaMatrix
~DakotaMatrix, 117
DakotaMatrix, 117
num_columns, 117
num_rows, 117
print, 117
read, 117
reshape2D, 117
DakotaMatrix, 117
operator=, 118
testClass, 118
DakotaModel
activate_model_auto_graphics, 123
active_variables, 121
apply_correction, 120
asvList, 125
asynch_compute_response, 120, 121
asynch_flag, 123
asynchEvalFlag, 125
asynchFDFlag, 125
auto_correction, 120
build_approximation, 119
compute_correction, 120
compute_response, 120
continuous_lower_bounds, 122
continuous_upper_bounds, 122
continuous_variable_labels, 121
continuous_variables, 121
current_response, 123
current_variables, 123
currentResponse, 124
currentVariables, 124
cv, 121
DakotaModel, 126, 127
dbFnsList, 125
dbResponseList, 125
deltaList, 126
derived_asynch_compute_response, 123
derived_compute_response, 123
derived_init_communicators, 124
derived_master_overload, 120
derived_synchronize, 123
derived_synchronize_nowait, 124
discrete_lower_bounds, 122
discrete_upper_bounds, 122
discrete_variable_labels, 121
discrete_variables, 121
dv, 121
finiteDiffSS, 126
free_communicators, 120
gradient_concurrency, 123
gradient_method, 123
gradType, 126
idAnalytic, 126
inactive_continuous_variables, 121
inactive_discrete_variables, 122
initialMapList, 125
intervalType, 126
is_null, 123
linear_eq_constraint_coeffs, 122
linear_eq_constraint_targets, 122
linear_ineq_constraint_coeffs, 122
linear_ineq_constraint_lower_bounds, 122
linear_ineq_constraint_upper_bounds, 122
maximum_concurrency, 119
merged_integer_list, 122
message_lengths, 123
messageLengths, 125
methodSrc, 126
modelType, 123
modelAutoGraphicsFlag, 125
modelRep, 125
modelType, 125
new_eval_counter, 120

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch (C) 1997-2001
realWorkSpaceSize, 134
speculativeFlag, 135
staticMultiObjWeights, 136
staticNumContinuousVars, 135
staticNumNonlinearConstraints, 136
staticNumObjFns, 135
vendorNumericalGradFlag, 135
DakotaOptimizer, 133
DakotaOptimizer, 136
multi_objective_modify, 137
printIterator_results, 136
run_iterator, 136
DakotaResponse
~DakotaResponse, 138
activeSet vector, 138
copy, 139
copy_results, 140
DakotaResponse, 138, 140
DakotaResponseRep, 144
data size, 140
fnTags, 138, 139
function_gradients, 139
function_hessians, 139
function_values, 139
interface_id, 138
numFunctions, 138
operator=, 138
operator==, 138
overlay, 140
purge_inactive, 140
read, 139
read_annotated, 139
read_data, 140
reset, 140
responseRep, 140
write, 139
write_annotated, 139
write_data, 140
write_tabular, 139
DakotaResponse, 138
DakotaResponse, 140
dakotaResponseList
SGOPTApplication, 303
DakotaResponseRep
~DakotaResponseRep, 142
copy_results, 143
DakotaResponse, 144
DakotaResponseRep, 142, 144
data_size, 143
fnTags, 143
functionGradients, 143
functionHessians, 143
functionValues, 143
interfaceId, 143
overlay, 143
purge_inactive, 143
read_data, 143
referenceCount, 143
reset, 143
responseASV, 143
write_data, 143
DakotaResponseRep, 142
DakotaResponseRep, 144
read, 144–146
read_annotated, 145
write, 145, 146
write_annotated, 145
write_tabular, 145
DakotaStrategy
DakotaStrategy, 149, 150
graphicsFlag, 148
iterator_communicator, 147
iterator_communicator_size, 147
iteratorComm, 148
iteratorCommRank, 148
iteratorCommSize, 148
parallelLib, 148
probDescDB, 148
referenceCount, 149
run_strategy, 147
strategyName, 148
strategyRep, 149
tabularDataFile, 148
tabularDataFlag, 148
worldRank, 148
worldSize, 148
DakotaStrategy, 147
~DakotaStrategy, 150
DakotaStrategy, 149, 150
get_strategy, 150
initialize_graphics, 150
operator=, 150
probDescDB, 151
run_iterator, 150
DakotaString
~DakotaString, 152
DakotaString, 152
isNull, 152
operator=, 152
toLower, 152
toUpper, 152
DakotaString, 152
contains, 153
data, 153
lower, 153
operator const char *, 153
testClass, 153
upper, 153
DakotaVarConstraints
  continuous_lower_bounds, 155
dakotaVarConstraints, 155
  continuous_upper_bounds, 155
DakotaVarConstraints, 158
discrete_lower_bounds, 155
dakotaVarConstraints, 155
  discrete_upper_bounds, 156
  discreteFlag, 157
linear_eq_constraint_coeffs, 156
linearEqConstraintCoeffs, 157
linearEqConstraintTargets, 157
linearIneqConstraintCoeffs, 157
linearIneqConstraintLowerBounds, 156
linearIneqConstraintUpperBounds, 156
linearEqConstraintCoeffs, 157
linearEqConstraintTargets, 157
linearIneqConstraintCoeffs, 157
linearIneqConstraintLowerBounds, 156
linearIneqConstraintUpperBounds, 156
read, 156
referenceCount, 157
varConstraintsRep, 157
variablesType, 157
write, 156
DakotaVarConstraints, 155
  dakotaVarConstraints, 158
  get_var_constraints, 159
  manage_linear_constraints, 159
  operator=, 159
DakotaVariables, 160
  dakotaVariables, 163, 164
  get_variables, 165
  operator=, 164
DakotaVector
  dakotaVector, 166
  operator T *, 168
  operator=, 167
  print, 167
  print_annotated, 167
  print_aprepro, 167
  print_partial, 167
  print_partial_aprepro, 167
  read, 166, 167
  read_annotated, 167
  read_partial, 166, 167
DakotaVector, 166
  copy_array, 169
  dakotaVector, 168
  operator=, 168
  testClass, 168
data
  dakotaArray, 89
  dakotaBaseVector, 92
  dakotaString, 153
data_pairs
    main.C, 340
data_size
    dakotaResponse, 140
    dakotaResponseRep, 143
DataInterface
  dataInterface, 170
  activeSetVectorFlag, 172
  actualInterfacePtr, 172
  actualInterfaceResponsesPtr, 172
  analysisDrivers, 171
  analysisScheduling, 172
  analysisServers, 172
  analysisUsage, 171
  approxCorrection, 173
  approxDaceMethodPtr, 173
  approxGradUsageFlag, 173
  approxSampleReuse, 173
approxType, 172
apreproFormatFlag, 171
assign, 173
asyncLocalAnalysisConcurrency, 171
dlInterface, 170
evalScheduling, 172
evalServers, 172
failAction, 172
fileSaveFlag, 171
fileTagFlag, 171
highFidelityInterfacePtr, 172
dlInterface, 170
inputFilter, 170
interfaceSynchronization, 171
interfaceType, 170
krigingCorrelations, 173
lowFidelityInterfacePtr, 172
operator=, 170
operator==, 170
outputFilter, 170
parametersFile, 171
procsPerAnalysis, 171
read, 170
recoveryFnVals, 172
resultsFile, 171
retryLimit, 172
write, 170
xmlHostNames, 171
xmlProcsPerHost, 171
DataInterface, 170
DataMethod
  ~DataMethod, 174
  allVarsFlag, 179
  assign, 180
  batchSize, 178
  centeringParam, 177
centralPath, 176
constraintTolerance, 175
crFactor, 177
crossoverRate, 177
crossoverType, 178
daceMethod, 176
dataMethod, 174
deltasPerVariable, 180
expandAfterSuccess, 178
expansionFlag, 178
expansionOrder, 179
expansionTerms, 179
exploratoryMoves, 178
finalPoint, 179
functionPrecision, 176
gradientTolerance, 176
idMethod, 174
initDelta, 177
initialRadius, 177
interfacePointer, 174
linearEqConstraintCoeffs, 176
linearEqTargets, 176
linearIneqConstraintCoeffs, 176
linearIneqLowerBnds, 176
linearIneqUpperBnds, 176
lineSearchTolerance, 176
listOfPoints, 180
maxCPUTime, 177
maxFunctionEvaluations, 175
maxIterations, 175
maxStep, 176
meritFn, 176
methodName, 174
methodOutput, 175
minMaxType, 176
modelType, 175
mutationDimRate, 177
mutationMinScale, 177
mutationPopRate, 177
mutationRange, 178
mutationScale, 177
mutationType, 178
newSolnsGenerated, 177
nonAdaptiveFlag, 178
numberRetained, 178
numPartitions, 178
numSamples, 179
numSteps, 180
numSymbols, 179
operator=, 174
operator==, 174
optiInterfaceResponsesPointer, 175
paramStudyType, 179
patternBasis, 179
percentDelta, 180
populationSize, 177
primaryCoeffs, 175
probabilityLevels, 179
randomizeOrderFlag, 178
randomSeed, 179
read, 174
reliabilityMethod, 179
replacementType, 178
responseLevels, 179
responsesPointer, 175
responseThresholds, 179
sampleType, 179
searchMethod, 176
searchSchemeSize, 177

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
secondaryCoeffs, 175
selectionPressure, 178
solnAccuracy, 177
speculativeFlag, 175
stepLength, 179
stepLenToBoundary, 177
stepVector, 179
subMethodPointer, 175
threshDelta, 177
totalPatternSize, 178
variablesPointer, 174
verifyLevel, 176
write, 174

DataMethod, 174

DataResponses
~DataResponses, 181
assign, 183
DataResponses, 181
fdStepSize, 182
gradientType, 182
hessianType, 182
idAnalytic, 182
idNumerical, 182
idResponses, 182
intervalType, 182
methodSource, 182
multiObjectiveWeights, 182
nonlinearEqTargets, 182
nonlinearIneqLowerBnds, 182
nonlinearIneqUpperBnds, 182
numLeastSquaresTerms, 181
numNonlinearEqConstraints, 181
numNonlinearIneqConstraints, 181
numObjectiveFunctions, 181
numResponseFunctions, 182
operator=, 181
operator==, 181
read, 181
write, 181

DataResponses, 181

DataVariables
~DataVariables, 184
assign, 188
continuousDesignLabels, 186
continuousDesignLowerBnds, 185
continuousDesignUpperBnds, 185
continuousDesignVars, 185
continuousStateLabels, 188
continuousStateLowerBnds, 188
continuousStateVars, 188
DataVariables, 184
design, 184
operator=, 181
operator==, 181
read, 184
state, 184
uncertain, 184
uncertainCorrelations, 187
uncertainDistLowerBnds, 187
uncertainDistUpperBnds, 188
uncertainLabels, 188
uncertainVars, 187
uniformUncDistLowerBnds, 186
uniformUncDistUpperBnds, 187
weibullUncAlphas, 187
weibullUncBetas, 187
weibullUncDistLowerBnds, 187
weibullUncDistUpperBnds, 187
write, 184

DataVariables, 184
dbFnsList
  DakotaModel, 125
dbResponseList
  DakotaModel, 125
debugFlag
  DakotaInterface, 103
debugOutput
  DakotaIterator, 110
defaultASV
  ApplicationInterface, 53
define_filenames
  AnalysisCode, 43
dELFUN
  CONMINOptimizer, 71
  KrigApprox, 223
deltaList
  DakotaModel, 126
deltasPerVariable
  DataMethod, 180
  ParamStudy, 292
derived_asynch_compute_response
  DakotaModel, 123
  HierLayeredModel, 220
  NestedModel, 251
  SingleModel, 310
  SurrLayeredModel, 325
derived_compute_response
  DakotaModel, 123
  HierLayeredModel, 220
  NestedModel, 251
  SingleModel, 310
  SurrLayeredModel, 325
derived_init_communicators
  DakotaModel, 124
  HierLayeredModel, 219
  NestedModel, 251
  SingleModel, 311
  SurrLayeredModel, 326
derived_map
  ApplicationInterface, 49
  DirectFnApplicInterface, 190
  ForkApplicInterface, 201
  SysCallApplicInterface, 332
derived_map_ac
  DirectFnApplicInterface, 191
derived_map_asynch
  ApplicationInterface, 49
  DirectFnApplicInterface, 190
  ForkApplicInterface, 201
  SysCallApplicInterface, 332
derived_map_if
  DirectFnApplicInterface, 191
derived_map_of
  DirectFnApplicInterface, 191
derived_master_overload
  DakotaModel, 120
  HierLayeredModel, 219
  NestedModel, 248
  SingleModel, 310
  SurrLayeredModel, 326
derived_synch
  ApplicationInterface, 49
  DirectFnApplicInterface, 190
  ForkApplicInterface, 201
  SysCallApplicInterface, 332
derived_synch_kernel
  ForkApplicInterface, 202
  SysCallApplicInterface, 333
derived_synch_nowait
  ApplicationInterface, 49
  DirectFnApplicInterface, 190
  ForkApplicInterface, 201
  SysCallApplicInterface, 332
derived_synch_synchronize
  DakotaModel, 123
  HierLayeredModel, 220
  NestedModel, 251
  SingleModel, 310
  SurrLayeredModel, 325
derived_synch_synchronize_nowait
  DakotaModel, 124
  HierLayeredModel, 220
  NestedModel, 251
  SingleModel, 310
  SurrLayeredModel, 326
derived_synchronous_local_analysis
  ApplicationInterface, 49
  DirectFnApplicInterface, 190
  ForkApplicInterface, 201
  SysCallApplicInterface, 332
design
  DataVariables, 184
designModel
  NonDOptStrategy, 262
DF
  CONMINOptimizer, 74
  KrigApprox, 228
DirectFnApplicInterface
  ~DirectFnApplicInterface, 190
cyl_head, 191
derived_map, 190
derived_map_ac, 191
derived_map_asynch, 190
derived_map_if, 191
derived_map_of, 191
derived_synch, 190
derived_synch_nowait, 190
derived_synchronous_local_analysis, 190
printControl, 196
realCnlpArray, 196
draw_surface  
DakotaApproximation, 83
dummyFlag  
ParallelLibrary, 283
duplication_detect  
ApplicationInterface, 55
dv  
AllMergedVariables, 32
AllVariables, 39
DakotaModel, 121
DakotaVariables, 160
FundamentalVariables, 208
MergedVariables, 241
emptyIntVector  
AllMergedVarConstraints, 30
AllMergedVariables, 34
AllVariables, 41
FundamentalVarConstraints, 206
FundamentalVariables, 211
MergedVarConstraints, 239
MergedVariables, 243
emptyRealVector  
AllMergedVariables, 34
AllVariables, 41
FundamentalVarConstraints, 206
FundamentalVariables, 211
emptyStringArray  
AllMergedVariables, 34
FundamentalVariables, 211
MergedVariables, 243
enroll  
GetLongOpt, 214
enroll_done  
GetLongOpt, 213
entries  
DakotaList, 113
ePSAOptimizer  
SGOPTOptimizer, 307
erInverse  
NonDAdvMeanValue, 256
ErrorTable  
msg, 198
rc, 198
ErrorTable, 198
eval_analysis_inter_communicator  
ParallelLibrary, 282
eval_analysis_inter_communicators  
ParallelLibrary, 282
eval_analysis_message_pass  
ParallelLibrary, 282
eval_analysis_split_flag  
ParallelLibrary, 282
ParallelLibrary, 282
eval_id  
ParamResponsePair, 289
evalAnalysisInterComm  
ParallelLibrary, 285
evalAnalysisInterCommss  
ParallelLibrary, 285
evalAnalysisMessagePass  
ParallelLibrary, 285
evalAnalysisSplitFlag  
ParallelLibrary, 285
evalComm  
ApplicationInterface, 51
evalCommRank  
ApplicationInterface, 50
ParallelLibrary, 285
evalCommSize  
ApplicationInterface, 50
ParallelLibrary, 285
evalDedicatedMasterFlag  
ParallelLibrary, 285
evalDedMasterFlag  
ApplicationInterface, 50
evalId  
ParamResponsePair, 290
evalIdList  
ForkApplicInterFace, 202
HierLayeredModel, 219
evalIntraComm  
ParallelLibrary, 284
evalMasterFlag  
ParallelLibrary, 284
evalMessagePass  
ApplicationInterface, 50
evalScheduling  
ApplicationInterface, 52
DataInterface, 172
evalServerId  
ApplicationInterface, 50
ParallelLibrary, 285
evalServers  
DataInterface, 172
evaluation_communicator_rank  
ParallelLibrary, 282
evaluation_communicator_size  
ParallelLibrary, 282
evaluation_dedicated_master_flag  
ParallelLibrary, 282
evaluation_intra_communicator  
ParallelLibrary, 281
evaluation_master_flag  
ParallelLibrary, 281
evaluation_server_id  
ParallelLibrary, 282
evaluation_servers
   ParallelLibrary, 282
expandAfterSuccess
   DataMethod, 178
expansionFlag
   DataMethod, 178
expansionOrder
   DataMethod, 179
expansionTerms
   DataMethod, 179
exploratoryMoves
   DataMethod, 178
   SGOPTOptimizer, 306

f of x array
   KrigingSurf, 230
failAction
   ApplicationInterface, 53
   DataInterface, 172
failCountList
   SysCallApplicInterface, 333
failIdList
   SysCallApplicInterface, 333
failRecoveryFnVals
   ApplicationInterface, 53
failRetryLimit
   ApplicationInterface, 53
fcdGradientTerm
   SurrBasedOptStrategy, 320
fd gradients
   DakotaModel, 128
FDCH
   CONMINOptimizer, 70
   KrigApprox, 223
FDCHM
   CONMINOptimizer, 70
   KrigApprox, 223
fdnlf1
   SNLLOptimizer, 314
fdnlf1 evaluator
   SNLLOptimizer, 316
fdnlf1Con
   SNLLOptimizer, 314
fdStepSize
   DataResponses, 182
fileNameKey
   AnalysisCode, 45
fileSaveFlag
   AnalysisCode, 44
   DataInterface, 171
fileTagFlag
   AnalysisCode, 44
   DataInterface, 171
finalPoint
   DataMethod, 179
   ParamStudy, 292
finalStatistics
   DakotaNonD, 132
find
   DakotaList, 115
find coefficients
   ANNSurf, 46
   DakotaApproximation, 82
   HermiteSurf, 216
   KrigingSurf, 230
   MARRSSurf, 236
   RespSurf, 301
   TaylorSurf, 334
find optimum
   CONMINOptimizer, 69
   DakotaOptimizer, 133
   DOTOptimizer, 194
   NPSOLOptimizer, 275
   SGOPTOptimizer, 308
   SNLLOptimizer, 312
find optimum on model
   NPSOLOptimizer, 275
find optimum on user functions
   NPSOLOptimizer, 275
finiteDiffSS
   DakotaModel, 126
finiteDiffStepSize
   DakotaIterator, 109
finiteDiffType
   DakotaIterator, 109
flags
   MARRSSurf, 236
flush
   CommandShell, 66
fn_tags
   DakotaResponse, 138, 139
fn EvalCntr
   NPSOLOptimizer, 277
fn EvalId
   DakotaInterface, 103
fn Grads
   DirectFnApplicInterface, 192
fn Hessians
   DirectFnApplicInterface, 192
fnTags
   DakotaResponseRep, 143
fnVals
   DirectFnApplicInterface, 192
fork application
   ForkApplicInterface, 202
fork program
   ForkAnalysisCode, 199
ForkAnalysisCode
discreteStateVars, 210
dv, 208
emptyIntVector, 211
emptyRealVector, 211
emptyStringArray, 211
FundamentalVariables, 208, 211
inactive_continuous_variables, 209
inactive_discrete_variables, 209
nonDFlag, 210
operator==, 211
read, 209, 210
read_annotated, 209
tv, 208
uncertainLabels, 210
uncertainVars, 210
write, 209, 210
write_annotated, 209
FundamentalVariables, 208
FundamentalVariables, 211
fValueVector
  KrigApprox, 226
G1
  CONMINOptimizer, 74
    KrigApprox, 228
G2
  CONMINOptimizer, 74
    KrigApprox, 228
gammaContract
  SurrBasedOptStrategy, 320
gammaExpand
  SurrBasedOptStrategy, 320
gammaNoChange
  SurrBasedOptStrategy, 320
get
  DakotaList, 114
get_approx
  DakotaApproximation, 85
get_chaos
  HermiteSurf, 216
get_db
  ProblemDescDB, 295
get_db_list_nodes
  ProblemDescDB, 295
get_dia
  ProblemDescDB, 296
get_dil
  ProblemDescDB, 296
get_dilv
  ProblemDescDB, 296
get_dila
  ProblemDescDB, 296
get_dilv
  ProblemDescDB, 296
get_drm
  ProblemDescDB, 296
get_drv
  ProblemDescDB, 296
get_dlsa
  ProblemDescDB, 296
get_dsl
  ProblemDescDB, 296
get_gradient
  DakotaApproximation, 82
    RespSurf, 301
    TaylorSurf, 334
get_int
  ProblemDescDB, 296
get_interface
  DakotaInterface, 105
get_iterator
  DakotaIterator, 112
get_model
  DakotaModel, 128
get_num_chaos
  HermiteSurf, 216
get_real
  ProblemDescDB, 296
get_short
  ProblemDescDB, 296
get_sizet
  ProblemDescDB, 296
get_source_pair
  ApplicationInterface, 52
get_strategy
  DakotaStrategy, 150
get_string
  ProblemDescDB, 296
get_value
  ANNSurf, 46
    DakotaApproximation, 82
    HermiteSurf, 216
    KrigingSurf, 230
    MARSSurf, 236
    RespSurf, 301
    TaylorSurf, 334
get_var_constraints
  DakotaVarConstraints, 159
get_variables
  DakotaVariables, 165
GetLongOpt
  ~GetLongOpt, 212
    basename, 213
    enroll_done, 213
    GetLongOpt, 213
    last, 213
    optmarker, 213
    pname, 213
    setcell, 213
    table, 213

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
usage, 212
ustring, 213
GetLongOpt, 212
enroll, 214
GetLongOpt, 213
parse, 214
retrieve, 214
usage, 214
getRe
CtlRegexp, 76
getStatus
CtlRegexp, 76
getStatusMsg
CtlRegexp, 76
globalApproxFlag
SurrBasedOptStrategy, 321
gradFlag
DirectFnApplicInterface, 192
gradient_concurrency
DakotaModel, 123
gradient_method
DakotaModel, 123
gradientFlag
DakotaApproximation, 83
SurrBasedOptStrategy, 321
gradientTolerance
DataMethod, 176
gradientType
DakotaIterator, 109
DataResponses, 182
gradType
DakotaModel, 126
gradVector
DakotaApproximation, 83
graphics2D
DakotaGraphics, 99
graphicsCntr
DakotaGraphics, 99
graphicsFlag
ApproximationInterface, 59
DakotaStrategy, 148
hard_convergence_check
SurrBasedOptStrategy, 322
headerFlag
ApplicationInterface, 53
HermiteSurf
~HermiteSurf, 216
chaosCoeffs, 217
chaosSamples, 217
find_coefficients, 216
get_chaos, 216
get_num_chaos, 216
get_value, 216
HermiteSurf, 216
highestOrder, 217
index, 217
numChaos, 217
required_samples, 216
HermiteSurf, 216
hessFlag
DirectFnApplicInterface, 192
hessianType
DakotaIterator, 109
DataResponses, 182
hierarchApproxFlag
SurrBasedOptStrategy, 321
HierLayeredModel
~HierLayeredModel, 218
build_approximation, 218
derived_init_communicators, 219
derived_master_overload, 219
evalIdList, 219
free_communicators, 219
HierLayeredModel, 218
highFidelityModel, 219
highFidResponse, 219
local_eval_synchronization, 218
lowFidelityInterface, 219
new_eval_counter, 219
serve, 219
stop_servers, 219
subordinate_model, 218
synchronize_nowait_completions, 219
total_eval_counter, 219
HierLayeredModel, 218
derived_async_computes_response, 220
derived_compute_response, 220
derived_synchronize, 220
derived_synchronize_nowait, 220
highestOrder
HermiteSurf, 217
NonDPCE, 265
highFidelityInterfacePtr
DataInterface, 172
highFidelityModel
HierLayeredModel, 219
highFidResponse
HierLayeredModel, 219
histogramFilenames
DakotaNonD, 131
histogramUncDistLowerBnds
DataVariables, 187
histogramUncDistUpperBnds
DataVariables, 187
histogramUncFilenames
DataVariables, 187
historyDuplicateIds
INFOG

  KrigApprox, 225
informResult
  NPSOLOptimizer, 276
init\_analysis\_communicators
  ParallelLibrary, 286
init\_communicators
  ApplicationInterface, 48
DakotaInterface, 102
DakotaModel, 128
init\_evaluation\_communicators
  ParallelLibrary, 286
init\_fn
  SNLLOptimizer, 312
init\_iterator\_communicators
  ParallelLibrary, 286
initDelta
  DataMethod, 177
initialize\_graphics
  DakotaStrategy, 150
initialMapList
  DakotaModel, 125
initialPoint
  NPSOLOptimizer, 277
ParamStudy, 292
initialRadius
  DataMethod, 177
input\_filter\_name
  AnalysisCode, 43
inputFilter
  DataInterface, 170
insert
  DakotaList, 115
intCntlParmArray
  DOTOptimizer, 196
interface\_id
  DakotaResponse, 138
ParamResponsePair, 289
interface\_kwhandler
  ProblemDescDB, 297
interface\_synchronization
  ApplicationInterface, 48
DakotaInterface, 102
interface\_type
  DakotaInterface, 102
interface\_Id
  DakotaResponseRep, 143
interface\_Index
  ProblemDescDB, 299
interface\_List
  ProblemDescDB, 299
interface\_Pointer
  DataMethod, 174
NestedModel, 250

APPLICATION INTERFACE, 53
history\_Duplicate\_Responses
APPLICATION INTERFACE, 53

IC
  CONMINOptimizer, 75
  KrigApprox, 229
ICNDIR
  KrigApprox, 225
id\_Analytic
  DakotaModel, 126
  DataResponses, 182
id\_Interface
  DataInterface, 170
id\_Method
  DataMethod, 174
id\_Numerical
  DataResponses, 182
id\_Responses
  DataResponses, 182
id\_Keyword\_Table
  keyword\_table.C, 339
id\_Variables
  Data\_Variables, 185
i\_Filter\_Name
  Analysis\_Code, 44
  Direct\_Fn\_App\_Interface, 192
i\_Flag
  Krig\_Approx, 229
IGOTO
  Krig\_Approx, 225
imp\_Factor
  Non\_DA\_Adv\_Mean\_Value, 257
inactive\_continuous\_variables
  All\_Merged\_Variables, 33
  All\_Variables, 40
  Dakota\_Model, 121
  Dakota\_Variables, 161
  Fundamental\_Variables, 209
  Merged\_Variables, 242
inactive\_discrete\_variables
  All\_Merged\_Variables, 33
  All\_Variables, 40
  Dakota\_Model, 122
  Dakota\_Variables, 161
  Fundamental\_Variables, 209
  Merged\_Variables, 242
in\_Buf
  Dakota\_Bi\_Stream, 94
index
  Dakota\_List, 115, 116
  Hermite\_Surf, 217
info
  Resp\_Surf, 302
interfaceRep
  DakotaInterface, 104
interfaceResponse
  NestedModel, 250
interfaceSynchronization
  ApplicationInterface, 52
  DataInterface, 171
interfaceType
  DakotaInterface, 103
  DataInterface, 170
intervalType
  DakotaModel, 126
  DataResponses, 182
intProblem
  SGOPTOptimizer, 307
intWorkSpace
  DakotaOptimizer, 134
intWorkSpaceSize
  DakotaOptimizer, 134
invcorrelMatrix
  KrigApprox, 226
iPivotVector
  KrigApprox, 226
IPRINT
  CONMINOptimizer, 70
  KrigApprox, 226
irecv_ea
  ParallelLibrary, 280
irecv_ie
  ParallelLibrary, 280
irecv_si
  ParallelLibrary, 280
isNull
  DakotaIterate, 108
  DakotaModel, 123
ISC
  CONMINOptimizer, 75
  KrigApprox, 228
isEmpty
  DakotaList, 114
isend_ea
  ParallelLibrary, 280
isend_ie
  ParallelLibrary, 280
isend_si
  ParallelLibrary, 280
isNull
  DakotaString, 152
ITER
  KrigApprox, 225
iteratorCommunicator
  DakotaStrategy, 147
  ParallelLibrary, 281
iteratorDedicatedMasterFlag
  DakotaInterface, 102
  ParallelLibrary, 281
iteratorEvalInterCommunicator
  ParallelLibrary, 281
iteratorEvalInterCommunicators
  ParallelLibrary, 282
iteratorEvalMessagePass
  ParallelLibrary, 281
iteratorEvalSplitFlag
  ParallelLibrary, 281
iteratorIntraCommunicator
  ParallelLibrary, 281
iteratorMasterFlag
  ParallelLibrary, 281
iteratorResponseResults
  DACEIterator, 78
  DakotaIterator, 107
  DakotaNonD, 130
  DakotaOptimizer, 133
  ParamStudy, 291
iteratorServerId
  ParallelLibrary, 281
iteratorServers
  ParallelLibrary, 281
iteratorVariableResults
  DACEIterator, 78
  DakotaIterator, 107
  DakotaOptimizer, 133
  ParamStudy, 291
iteratorComm
  DakotaStrategy, 148
iteratorCommRank
  ApplicationInterface, 50
  DakotaStrategy, 148
  ParallelLibrary, 284
iteratorCommSize
  ApplicationInterface, 50
  DakotaStrategy, 148
  ParallelLibrary, 284
iteratorDedicatedMasterFlag
  ParallelLibrary, 284
iteratorDedMasterFlag
  DakotaInterface, 103
iteratorEvalInterComm
  ParallelLibrary, 285
iteratorEvalInterComms
  ParallelLibrary, 285
iteratorEvalMessagePass
  ParallelLibrary, 284
iteratorEvalSplitFlag
ParallelLibrary, 284
iteratorIntraComm
ParallelLibrary, 284
iteratorMasterFlag
ParallelLibrary, 284
iteratorParameterSets
ConcurrentStrategy, 67
iteratorRep
DakotaIterator, 110
iteratorServerId
ConcurrentStrategy, 68
ParallelLibrary, 284
iterMax
SurroundingOptStrategy, 320
ITMAX
CONMINOptimizer, 70
KrigApprox, 223
ITRM
KrigApprox, 225
iworkVector
KrigApprox, 226
jacUToX
NonDAdvMeanValue, 261
jacXToU
NonDAdvMeanValue, 260
jacXToZ
NonDAdvMeanValue, 260
jacZToX
NonDAdvMeanValue, 260
keywordTable.C, 339
idrKeywordTable, 339
KrigApprox
~KrigApprox, 222
ABOBJ1, 224
ALPHAX, 224
betaHat, 225
comm_theta_lower_bnds, 224
comm_theta_upper_bnds, 224
comm_theta_vars, 224
commInfo, 223
commSingleArray, 222
constraintVector, 226
correlationMatrix, 226
CTL, 223
CTLMIN, 223
CTMIN, 223
DABFUN, 223
DELFUN, 223
FDCH, 223
FDCHM, 223
fRinvVector, 226
fValueVector, 226
ICNDIR, 225
IGOTO, 225
INFOG, 225
invcorrelMatrix, 226
iPivotVector, 226
IPRINT, 223
ITER, 225
ITMAX, 223
ITRM, 225
iworkVector, 226
KrigApprox, 222
LINOBI, 224
maxLikelihoodEst, 225
ModelBuild, 222
NAC, 225
NACMXI, 224
NDFG, 222
NSCAL, 224
NSIDE, 224
numcon, 222
numNewPts, 225
numSampQuad, 225
PHI, 224
rhsTermsVector, 226
rXhatVector, 226
THETA, 224
thetaLoBndVector, 225
thetaUpBndVector, 226
thetaVector, 225
workVector, 226
workVectorQuad, 226
xMatrix, 225
xNewVector, 225
yfbRinvVector, 226
yfbVector, 226
yNewVector, 225
yValueVector, 225
KrigApprox, 222
A, 228
B, 228
C, 228
CT, 227
DF, 228
G1, 228
G2, 228
IC, 229
iFlag, 229
ISC, 228
ModelApply, 227
MS1, 228
N1, 227
N2, 227
N3, 227
N4, 227
N5, 227
S, 227
SCAL, 228
krigingCorrelations
DataInterface, 173
KrigingSurf
~KrigingSurf, 230
correlationVector, 231
f_of_x_array, 230
find_coefficients, 230
get_value, 230
KrigingSurf, 230
krigObject, 230
required_samples, 230
runConminFlag, 231
x_matrix, 230
KrigingSurf, 230
krigObject
KrigingSurf, 230

last
GetLongOpt, 213
LayeredModel
~LayeredModel, 232
apply_correction, 232
auto_correction, 232
badScalingFlag, 233
betaApproxCenterGrads, 233
betaApproxCenterVals, 233
betaCenterPt, 233
betaFcnRatio, 233
betaGradOffset, 234
betaGrads, 233
betaHessOffset, 234
check_submodel_compatibility, 232
correctedResponseArray, 232
correctedResponseList, 232
correctionComputed, 233
correctionType, 233
LayeredModel, 232
offsetValues, 233
rawCVarsList, 233
scaleFactors, 233
LayeredModel, 232
approxBuilds, 234
autoCorrection, 234
compute_correction, 234
leadDimA
RespSurf, 302
leadDimB
RespSurf, 302
length
DakotaArray, 88
DakotaBaseVector, 91
lenPRPairMessage
ApplicationInterface, 51
lenResponseMessage
ApplicationInterface, 51
lenVarsASVMessage
ApplicationInterface, 51
lenVarsMessage
ApplicationInterface, 51
lhsSampler
NonDPCE, 265
NonDProbability, 268
lin_approx_constraint_eval
NonDAdvMeanValue, 259
lin_approx_objective_eval
NonDAdvMeanValue, 258
linConGenerator
SGPTOptimizer, 306
linConstraintMatrixF77
NPSLOptimizer, 276
linConstraintMatrixSize
NPSLOptimizer, 276
linearEq_constraint_coeffs
DakotaModel, 122
DakotaVarConstraints, 156
linear_eq_constraint_targets
DakotaModel, 122
DakotaVarConstraints, 156
linear_ineq_constraint_coeffs
DakotaModel, 122
DakotaVarConstraints, 156
linear_ineq_constraint_lower_bounds
DakotaModel, 122
DakotaVarConstraints, 156
linear_ineq_constraint_upper_bounds
DakotaModel, 122
DakotaVarConstraints, 156
linearEqConstraintCoeffs
DakotaOptimizer, 135
DakotaVarConstraints, 157
DataMethod, 176
linearEqConstraintTargets
DakotaVarConstraints, 157
linearEqTargets
DakotaOptimizer, 135
DataMethod, 176
linearIneqConstraintCoeffs
DakotaOptimizer, 135
DakotaVarConstraints, 157
DataMethod, 176
linearIneqConstraintLowerBnds
DakotaVarConstraints, 157
linearIneqConstraintUpperBnds
DakotaVarConstraints, 157
linearIneqLowerBnds
DakotaOptimizer, 135
DataMethod, 176
linearIneqUpperBnds
   DakotaOptimizer, 135
   DataMethod, 176
lineSearchTolerance
   DataMethod, 176
LINOBJ
   KrigApprox, 224
listOfPoints
   DataMethod, 180
   ParamStudy, 292
localEvalSynchronization
   DakotaModel, 127
   HierLayeredModel, 218
   SingleModel, 310
localApproxFlag
   SurrBasedOptStrategy, 321
localConstraintArraySize
   DakotaOptimizer, 135
   localConstraintValues
   CONMINOptimizer, 72
   DOTOptimizer, 196
localSearchProb
   MultilevelOptStrategy, 246
lognormalDistLowerBnds
   DakotaNonD, 131
lognormalDistUpperBnds
   DakotaNonD, 131
lognormalErrFacts
   DakotaNonD, 131
lognormalMeans
   DakotaNonD, 131
lognormalStdDevs
   DakotaNonD, 131
lognormalUncDistLowerBnds
   DataVariables, 186
lognormalUncDistUpperBnds
   DataVariables, 186
lognormalUncErrFacts
   DataVariables, 186
lognormalUncMeans
   DataVariables, 186
lognormalUncStdDevs
   DataVariables, 186
loguniformDistLowerBnds
   DakotaNonD, 131
loguniformDistUpperBnds
   DakotaNonD, 131
loguniformUncDistLowerBnds
   DataVariables, 187
loguniformUncDistUpperBnds
   DataVariables, 187
lower
   DakotaString, 153
   lowerBounds
   NPSOLOptimizer, 277
   lowFidelityInterface
   HierLayeredModel, 219
   lowFidelityInterfacePtr
   DataInterface, 172
   lwork
   RespSurf, 302
main
   main.C, 341
   restart_util.C, 343
   main.C, 340
   clean_up, 341
data_pairs, 340
   main, 341
   manage_inputs, 341
   manage_outputs, 341
   manage_restart, 341
   write_precision, 340
   write restart, 340
   manage_asv
   DakotaModel, 129
   manage_failure
   ApplicationInterface, 48
   manage_inputs
   main.C, 341
   manage_linear_constraints
   DakotaVarConstraints, 159
   manage_outputs
   main.C, 341
   manage_restart
   main.C, 341
map
   ApplicationInterface, 54
   ApproximationInterface, 58
   DakotaInterface, 101
marsObject
   MARSSurf, 236
   MARSSurf, 236
   ~MARSSurf, 236
   find_coefficients, 236
   flags, 236
   get_value, 236
   marsObject, 236
   MARSSurf, 236
   required_samples, 236
match
   CtelRegexp, 76
matrixS
   RespSurf, 302
   matrixTerms
DakotaIterator, 108

methodIndex
  ProblemDescDB, 299

methodList
  MultilevelOptStrategy, 245
  ProblemDescDB, 299

methodName
  DakotaIterator, 109
  DataMethod, 174

methodOutput
  DataMethod, 175

methodSource
  DakotaIterator, 109
  DataResponses, 182

methodSrc
  DakotaModel, 126

mfcn
  SNLLOptimizer, 313

minimum_samples
  ApproximationInterface, 58
  DakotaInterface, 102

minMaxType
  DataMethod, 176

minSamples
  ApproximationInterface, 59

minTrustRegionSize
  SurrBasedOptStrategy, 320

mixedGradAnalyticIds
  DakotaIterator, 110

mixedGradNumericalIds
  DakotaIterator, 110

model_type
  DakotaModel, 123

ModelApply
  KrigApprox, 227

modelAutoGraphicsFlag
  DakotaModel, 125

ModelBuild
  KrigApprox, 222

modelRep
  DakotaModel, 125

modelType
  DakotaModel, 125
  DataMethod, 175

modified_parameters_filename
  AnalysisCode, 43

modified_results_filename
  AnalysisCode, 43

modifiedParamsFileName
  AnalysisCode, 44

modifiedResFileName
  AnalysisCode, 45

mostProbPointU
  NonDAdvMeanValue, 257

mostProbPointX
  NonDAdvMeanValue, 257

mpirunFlag
  ParallelLibrary, 283

MS1
  CONMINOptimizer, 74
  KrigApprox, 228

msg
  ErrorTable, 198

multi_objective_modify
  DakotaOptimizer, 137

multi_objective_weights
  DakotaIterator, 108
  DakotaOptimizer, 133

multi_proc_eval_flag
  DakotaInterface, 102

multiDimLoop
  ParamStudy, 292

MultilevelOptStrategy
  MultilevelOptStrategy, 245

localSearchProb, 246

methodList, 245

MultilevelOptStrategy, 245

multiLevelType, 245

numIterators, 246

progressMetric, 246

progressThreshold, 246

run_strategy, 245

selectedIterators, 246

userDefinedModels, 246

MultilevelOptStrategy, 245

run_coupled, 246

run_uncoupled, 246

run_uncoupled_adaptive, 247

multiLevelType
  MultilevelOptStrategy, 245

multiObjectiveWeights
  DataResponses, 182

multiObjModifyPtr
  SOOPTApplication, 304

multiObjWeights
  DACEIterator, 79
  DakotaOptimizer, 136
  ParamStudy, 293

multiObjWts
  SurrBasedOptStrategy, 321

multiProcAnalysisFlag
  ApplicationInterface, 51

multiProcEvalFlag
  DakotaFlag, 103

mutationDimRate
  DataMethod, 177

mutationMinScale
  DataMethod, 177
mutationPopRate
DataMethod, 177
mutationRange
DataMethod, 178
mutationScale
DataMethod, 177
mutationType
DataMethod, 178

N1
CONMINOptimizer, 73
KrigApprox, 227
N2
CONMINOptimizer, 73
KrigApprox, 227
N3
CONMINOptimizer, 73
KrigApprox, 227
N4
CONMINOptimizer, 73
KrigApprox, 227
N5
CONMINOptimizer, 73
KrigApprox, 227
NAC
KrigApprox, 225
NACMX1
KrigApprox, 224
nestedFlag
ParamStudy, 292
NestedModel
~NestedModel, 248
asv_mapping, 249
completionList, 250
constrCoefFs, 250
derived_master_overload, 248
freeCommunicators, 249
interfacePointer, 250
interfaceResponse, 250
NestedModel, 248
next_eval_counter, 249
numInterEqConstr, 250
numInterIneqConstr, 250
numInterObjFns, 250
numSubIteratorEqConstr, 249
numSubIteratorIneqConstr, 249
objCoefFs, 250
optionalInterface, 250
responseArray, 250
responseList, 250
serve, 249
stop_servers, 249
subIterator, 249
subordinate_model, 248
total_eval_counter, 249
NestedModel, 248
derived_async_compute_response, 251
derived_compute_response, 251
derived_initCommunicators, 251
derived_synchronize, 251
derived_synchronize_nowait, 251
response_mapping, 252
subModel, 252
synchronize_nowait_completions, 251
new_eval_counter
DakotaInterface, 102
DakotaModel, 120
HierLayeredModel, 219
NestedModel, 249
SingleModel, 311
SurrLayeredModel, 324
ewCenterFlag
SurrBasedOptStrategy, 321
newFnEvalId
DakotaInterface, 103
newSolnsGenerated
DataMethod, 177
next_eval
SGOPTApplication, 304
NFDG
CONMINOptimizer, 70
KrigApprox, 222
nlf0
SNLLOptimizer, 313
nlf0_evaluator
SNLLOptimizer, 316
nlf1
SNLLOptimizer, 313
nlf1_evaluator
SNLLOptimizer, 316
nlf1Con
SNLLOptimizer, 314
nlf2
SNLLOptimizer, 314
nlf2_evaluator
SNLLOptimizer, 316
nlf2_evaluator_gn
SNLLOptimizer, 316
nlf2Con
SNLLOptimizer, 314
nlfConstraint
SNLLOptimizer, 313
nlfObjective
SNLLOptimizer, 313
nlpConstraint
SNLLOptimizer, 313
NoDBBaseConstructor
NoDBBaseConstructor, 254
num_variables
  DakotaApproximation, 83
  DataVariables, 185
numActiveVars
  NonDPCE, 265
  NonDProbability, 268
  NonDSampling, 272
numAnalysisDrivers
  ApplicationInterface, 51
numAnalysisServers
  ApplicationInterface, 51
  ParallelLibrary, 285
numDerivativeIterations
  NPSOLOptimizer, 276
numRetained
  DataMethod, 178
numCDV
  AllVarConstraints, 37
  AllVariables, 41
numChaos
  HermiteSurf, 217
  NonDPCE, 265
numColsA
  RespSurf, 301
numCon
  KrigApprox, 222
numConstraints
  DakotaOptimizer, 135
numContinuousDesVars
  DataVariables, 185
numContinuousStateVars
  DataVariables, 185
numContinuousVars
  DakotaIterator, 109
numCSV
  AllVarConstraints, 37
  AllVariables, 41
numCurrentPoints
  DakotaApproximation, 83
numDDV
  AllVarConstraints, 37
  AllVariables, 41
numDesignVars
  NonDProbability, 268
  NonDSampling, 272
numDiscreteDesVars
  DataVariables, 185
numDiscreteStateVars
  DataVariables, 185
numDiscreteVars
  DakotaIterator, 109
numDSV
  AllVarConstraints, 37
  AllVariables, 41
numEvalServers
  ApplicationInterface, 52
  ParallelLibrary, 285
numFns
  DakotaModel, 124
  DirectFnApplicationInterface, 192
  SurrBasedOptStrategy, 320
numFunctions
  DakotaOptimizer, 109
numGradVars
  DakotaModel, 124
  DirectFnApplicationInterface, 192
numHistogramUncVars
  DataVariables, 185
numHistogramVars
  DakotaNonD, 132
numInterfEqConstr
  NestedModel, 250
numInterfIneqConstr
  NestedModel, 250
numInterfObjFns
  NestedModel, 250
numIteratorJobs
  ConcurrentStrategy, 67
numIterators
  MultiLevelOptStrategy, 246
numIteratorServers
  BranchBndStrategy, 62
  ConcurrentStrategy, 67
  ParallelLibrary, 284
numLeastSquaresTerms
  DataResponses, 181
numLinearConstraints
  DakotaOptimizer, 135
numLinearEqConstraints
  DakotaOptimizer, 135
  DakotaVarConstraints, 157
numLinearIneqConstraints
  DakotaOptimizer, 135
  DakotaVarConstraints, 157
numLognormalUncVars
  DataVariables, 185
numLognormalVars
  DakotaNonD, 132
numLoguniformUncVars
  DataVariables, 185
numLoguniformVars
  DakotaNonD, 132
numMapsList
  DakotaModel, 126
numNewPts
  KrigApprox, 225
numNodeSamples
  BranchBndStrategy, 62
numNonlinCons
  SGOPTApplication, 304
numNonlinearConstraints
  DakotaOptimizer, 134
numNonlinearEqConstraints
  DACEIterator, 79
  DakotaOptimizer, 134
  DataResponses, 181
  ParamStudy, 293
numNonlinearIneqConstraints
  DACEIterator, 79
  DakotaOptimizer, 134
  DataResponses, 181
  ParamStudy, 293
numNonlinEqConstr
  SurrBasedOptStrategy, 321
numNonlinIneqConstr
  SurrBasedOptStrategy, 321
numNormalUncVars
  DataVariables, 185
numNormalVars
  DakotaNonD, 131
numObjectiveFunctions
  DACEIterator, 79
  DakotaOptimizer, 134
  DataResponses, 181
  ParamStudy, 293
numObjFns
  SGOPTApplication, 304
  SurrBasedOptStrategy, 321
numObservations
  NonDPCE, 265
  NonDPProbability, 268
  NonDSampling, 272
numPartitions
  DataMethod, 178
numPrograms
  AnalysisCode, 44
numResponseFunctions
  DakotaNonD, 132
  DataResponses, 182
numRHS
  RespSurf, 302
numRootSamples
  BranchBndStrategy, 62
numRowsA
  RespSurf, 301
numSamples
  DACEIterator, 79
  DakotaApproximation, 83
  DataMethod, 179
numSampQuad
  KrigApprox, 225
numStateVars
  NonDPProbability, 268
  NonDSampling, 272
numSteps
  DataMethod, 180
  ParamStudy, 292
numSubIteratorEqConstr
  NestedModel, 249
numSubIteratorIneqConstr
  NestedModel, 249
numSymbols
  DACEIterator, 79
  DataMethod, 179
numUncertainVars
  DakotaNonD, 132
numUniformUncVars
  DataVariables, 185
numUniformVars
  DakotaNonD, 132
numUV
  AllVarConstraints, 37
  AllVariables, 41
numVars
  DakotaApproximation, 83
  DakotaIterator, 109
  DirectFnApplicInterface, 192
  SurrBasedOptStrategy, 320
numWeibullUncVars
  DataVariables, 185
numWeibullVars
  DakotaNonD, 132
numX
  NonDPCE, 265
objCoeffs
  NestedModel, 250
objective
  NPSOLOptimizer, 276
occurrencesOf
  DakotaList, 116
offsetValues
  LayeredModel, 233
oFilterName
  AnalysisCode, 44
  DirectFnApplicInterface, 192
operator const char *
  DakotaString, 153
operator T *
  DakotaArray, 89
  DakotaVector, 168
operator()
  DakotaArray, 87, 88
  DakotaBaseVector, 91
  FunctionCompare, 204
  SortCompare, 318
ParallelLibray, 279

analysis_communicator_rank, 282
analysis_communicator_size, 282
analysis_intra_communicator, 282
analysis_master_flag, 282
analysis_server_id, 282
analysis_servers, 282
analysisCommRank, 286
analysisCommSize, 286
analysisIntraComm, 285
analysisMasterFlag, 285
analysisServerld, 286
bcast, 280
dummyFlag, 283
eval_analysis_inter_communicator, 282
eval_analysis_inter_communicators, 282
eval_analysis_message_pass, 282
eval_analysis_split_flag, 282
evalAnalysisInterComm, 285
evalAnalysisInterComms, 285
evalAnalysisMessagePass, 285
evalAnalysisSplitFlag, 285
evalCommRank, 285
evalCommSize, 285
evalDedicatedMasterFlag, 285
evalIntraComm, 284
evalMasterFlag, 284
evalServerld, 285
evaluation_communicator_rank, 282
evaluation_communicator_size, 282
evaluation_dedicated_master_flag, 282
evaluation_intra_communicator, 281
evaluation_master_flag, 281
evaluation_server_id, 282
evaluation_servers, 282
free_analysis_communicators, 279
free_evaluation_communicators, 279
free_iterator_communicators, 279
irecv_ia, 280
irecv_je, 280
irecv_si, 280
isend_ia, 280
isend_je, 280
isend_si, 279
iterator_communicator_rank, 281
iterator_communicator_size, 281
iterator_dedicated_master_flag, 281
iterator_eval_inter_communicator, 281
iterator_eval_inter_communicators, 282
iterator_eval_message_pass, 281
iterator_eval_split_flag, 281
iterator_intra_communicator, 281
iterator_master_flag, 281
iterator_server_id, 281
iterator_servers, 281
iteratorCommRank, 284
iteratorCommSize, 284
iteratorDedicatedMasterFlag, 284
iteratorEvalInterComm, 285
iteratorEvalInterComms, 285
iteratorEvalMessagePass, 284
iteratorEvalSplitFlag, 284
iteratorIntraComm, 284
iteratorMasterFlag, 284
iteratorServerld, 284
mpirunFlag, 283
numAnalysisServers, 285
numEvalServers, 285
numIteratorServers, 284
parallelism_levels, 280
parallelismLevels, 283
ParallelLibray, 279
print_configuration, 279
procsPerAnalysis, 285
procsPerEval, 285
procsPerIterator, 284
recv_ia, 280
recv_je, 280
recv_si, 279
send_ia, 280
send_je, 280
send_si, 279
split_communicator_dedicated_master, 283
split_communicator_peer_partition, 283
startClock, 283
startCPUtime, 283
startMPItime, 283
startWCTime, 283
strategy_dedicated_master_flag, 281
strategy_iterator_inter_communicator, 281
strategy_iterator_inter_communicators, 281
strategy_iterator_message_pass, 281
strategy_iterator_split_flag, 281
strategyDedicatedMasterFlag, 284
stratIteratorInterComm, 284
stratIteratorInterComms, 284
stratIteratorMessagePass, 284
stratIteratorSplitFlag, 284
waitall, 280
world_rank, 280
world_size, 280
worldRank, 283
worldSize, 283
ParallelLibray, 279
init_analysis_communicators, 286
init_evaluation_communicators, 286
init_iterator_communicators, 286
resolve_inputs, 287
parametersFile
  DataInterface, 171
parametersFileName
  AnalysisCode, 44
parametersFNameList
  AnalysisCode, 45
ParamResponsePair
  ~ParamResponsePair, 288
active set vector, 289
eval_id, 289
interface_id, 289
operator=, 288
operator==, 289
ParamResponsePair, 288, 290
prp parameters, 289
prp response, 289
prPairParameters, 289
prPairResponse, 289
read, 288, 289
read_annotated, 288
write, 288, 289
write_annotated, 288
write_tabular, 288
ParamResponsePair, 288
eval_id, 290
ParamResponsePair, 290
paramSamples
  NonDPCE, 265
  NonDProbability, 268
  NonDSampling, 272
ParamStudy
  ~ParamStudy, 291
bestObjectiveFn, 293
bestResponses, 293
bestVariables, 293
bestViolations, 293
centered_loop, 292
compute_vector_steps, 291
deltasPerVariable, 292
finalPoint, 292
initialPoint, 292
iterator_response_results, 291
iterator_variable_results, 291
listOfPoints, 292
multidim_loop, 292
multiObjWeights, 293
nestedFlag, 292
nonlinearEqTargets, 293
nonlinearIneqLowerBnds, 293
nonlinearIneqUpperBnds, 293
numNonlinearEqConstraints, 293
numNonlinearIneqConstraints, 293
numObjectiveFunctions, 293
numSteps, 292
ParamStudy, 291
percentDelta, 293
print_iterator_results, 291
psCounter, 293
pStudyType, 292
recurse, 292
recurseFlag, 292
sample, 291
stepLength, 292
stepVector, 292
update_best, 292
variablePartitions, 293
vector_loop, 291
vectorOfVars, 293
ParamStudy, 291
run_iterator, 294
paramStudyType
  DataMethod, 179
parse
  GetLongOpt, 214
patternBasis
  DataMethod, 179
patternSearchOptimizer
  SGOPTOptimizer, 307
penaltyParameter
  SurrBasedOptStrategy, 320
percentDelta
  DataMethod, 180
ParamStudy, 293
petraCorrMatrix
  NonDAvgMeanValue, 257
petraProbLevels
  NonDAvgMeanValue, 258
petraRespLevels
  NonDAvgMeanValue, 258
pGAIIntOptimizer
  SGOPTOptimizer, 307
pGARealOptimizer
  SGOPTOptimizer, 307
PHI
  KrigApprox, 224
picoComm
  BranchBndStrategy, 62
PicoCommRank
  BranchBndStrategy, 63
picoCommSize
  BranchBndStrategy, 63
picoListOfIntegers
  BranchBndStrategy, 63
picoLowerBnds
  BranchBndStrategy, 63
picoUpperBnds
  BranchBndStrategy, 63
pname
GetLongOpt, 213
populate_gradient_vars
DakotaIterator, 112
populationSize
DataMethod, 177
primaryCoeffs
DataMethod, 175
print
DakotaMatrix, 117
DakotaVector, 167
print_annotated
DakotaVector, 167
print_aprepro
DakotaVector, 167
print_configuration
ParallelLibrary, 279
print_iterator_results
DACEIterator, 78
DakotaIterator, 108
DakotaOptimizer, 136
NonDAдвMeanValue, 255
NonDPCE, 264
NonDProbability, 267
NonDSampling, 271
ParamStudy, 291
print_partial
DakotaVector, 167
print_partial_aprepro
DakotaVector, 167
print_restart
restart_util.C, 342
print_restart_tabular
restart_util.C, 343
printControl
CONMINOptimizer, 72
DOOmpimizer, 196
prob_desc_db
DakotaModel, 123
DakotaStrategy, 151
probabilityLevels
DataMethod, 179
probabilityLevelTargets
NonDAдвMeanValue, 257
probDescDB
DakotaIterator, 109
DakotaModel, 125
DakotaStrategy, 148
ProblemDescDB
~ProblemDescDB, 295
build_label, 297
build_labels, 297
check_input, 295
get_db_interface_node, 295
get_db_list_nodes, 295
get_dia, 296
get_dim, 296
get_dlv, 296
get_dla, 296
get_dlm, 296
get_dlr, 296
get_dmac, 296
get_dsl, 296
get_int, 296
get_real, 296
get_short, 296
get_sizet, 296
get_string, 296
interface_kwhandler, 297
interfaceIndex, 299
interfaceList, 299
method_kwhandler, 297
methodIndex, 299
methodList, 299
parallelLibrary, 296
parallelLib, 297
ProblemDescDB, 295
receive_db_buffer, 296
responses_kwhandler, 297
responsesIndex, 299
responsesList, 299
send_db_buffer, 295
set_db_interface_node, 295
set_db_list_nodes, 295
set_db_responses_node, 295
set_other_list_nodes, 297
strategy_kwhandler, 297
strategyBandBNumSamplesNode, 298
strategyBandBNumSamplesRoot, 298
strategyConcurrentNumJobs, 299
strategyConcurrentParameterSets, 299
strategyGraphicsFlag, 297
strategyIndex, 299
strategyIteratorScheduling, 298
strategyIteratorServers, 298
strategyMethodPointer, 298
strategyMultilevelGlobalMethodPointer, 298
strategyMultilevelLocalMethodPointer, 298
strategyMultilevelLSP, 298
strategyMultilevelMethodList, 298
strategyMultilevelProgThresh, 298
strategyMultilevelType, 298
strategySBOMaxIterations, 299
strategySBOTRC, 299
strategySBOTRecordExpand, 299
strategySBOTInitSize, 299
strategyTabularDataFile, 298
strategyTabularDataFlag, 298
strategyType, 297
variables_kwhandler, 297
variablesIndex, 299
variablesList, 299
ProblemDescDB, 295
set_db_model_type, 300
probLevels
NonDAAdvMeanValue, 258
probMoreThanThreshold
DakotaNonD, 132
processIdList
ForkApplicationInterface, 202
procsPerAnalysis
ApplicationInterface, 52
DataInterface, 171
ParallelLibrary, 285
procsPerEval
ParallelLibrary, 285
procsPerIterator
ParallelLibrary, 284
programNames
AnalysisCode, 43
programNames
AnalysisCode, 44
progressMetric
MultilevelOptStrategy, 246
progressThreshold
MultilevelOptStrategy, 246
prpParameters
ParamResponsePair, 289
prpResponse
ParamResponsePair, 289
prPairParameters
ParamResponsePair, 289
prPairResponse
ParamResponsePair, 289
psCounter
ParamStudy, 293
pStudyType
ParamStudy, 292
purge inactive
DakotaResponse, 140
DakotaResponseRep, 143
quadCoeffs
RespSurf, 302
quantify uncertainty
DakotaNonD, 130
NonDAAdvMeanValue, 255
NonDPCE, 264
NonDProbability, 269
NonDSampling, 273
quiet_flag
AnalysisCode, 43, 44
CommandShell, 65
quietFlag
AnalysisCode, 44
CommandShell, 65
r
CtelRegexp, 77
randomizeOrderFlag
DataMethod, 178
randomSeed
DACEIterator, 79
DataMethod, 179
NonDPCE, 265
NonDProbability, 268
NonDSampling, 272
rank
RespSurf, 302
ranVarMeans
NonDAAdvMeanValue, 257
ranVarSigmas
NonDAAdvMeanValue, 258
ranVarType
NonDAAdvMeanValue, 258
rawCV arsList
LayeredModel, 233
rawResponseArray
DakotaInterface, 105
rawResponseList
DakotaInterface, 105
rc
ErrorTable, 198
rcond
RespSurf, 302
read
AllMergedVarConstraints, 30
AllMergedVariables, 33, 34
AllVarConstraints, 37
AllVariables, 40, 41
DakotaMatrix, 117
DakotaResponse, 139
DakotaResponseRep, 144–146
DakotaVarConstraints, 156
DakotaVariables, 161, 162
DakotaVector, 166, 167
DataInterface, 170
DataMethod, 174
DataResponses, 181
DataVariables, 184
FundamentalVarConstraints, 206
FundamentalVariables, 209, 210
MergedVarConstraints, 239
MergedVariables, 242, 243
ParamResponsePair, 288, 289
read_annotated
  AllMergedVariables, 33
  AllVariables, 40
  DakotaResponse, 139
  DakotaResponseRep, 145
  DakotaVariables, 162
  DakotaVector, 167
  FundamentalVariables, 209
  MergedVariables, 242
  ParamResponsePair, 288

read_data
  DakotaResponse, 140
  DakotaResponseRep, 143

read_neutral
  restart_util.C, 343

read_partial
  DakotaVector, 166, 167

read_restart_values
  CommandLineHandler, 64

read_restart_stream
  CommandLineHandler, 64

read_results_file
  AnalysisCode, 43

realCntlParmArray
  DOTOptimizer, 196

realProblem
  SGOPTOptimizer, 307

realWorkSpace
  DakotaOptimizer, 134

realWorkSpaceSize
  DakotaOptimizer, 134

receive_db_buffer
  ProblemDescDB, 296

recoveryFnVals
  DataInterface, 172

recur
  ParamStudy, 292

recurFlag
  ParamStudy, 292

recv_ea
  ParallelLibrary, 280

recv_je
  ParallelLibrary, 280

recv_si
  ParallelLibrary, 279

referenceCount
  DakotaApproximation, 84
  DakotaInterface, 104
  DakotaIterator, 110
  DakotaModel, 125
  DakotaResponseRep, 143
  DakotaStrategy, 149
  DakotaVarConstraints, 157
  DakotaVariables, 163

reliabilityMethod
  DataMethod, 179
  NonDAdvMeanValue, 257

remove
  DakotaList, 115

removeAt
  DakotaList, 115

repair_restart
  restart_util.C, 343

replacementType
  DataMethod, 178

required_samples
  ANNSurf, 46
  DakotaApproximation, 82
  HermiteSurf, 216
  KrigingSurf, 230
  MARSurf, 236
  RespSurf, 301
  TaylorSurf, 334

reset
  DakotaResponse, 140
  DakotaResponseRep, 143

reshape
  DakotaArray, 88
  DakotaBaseVector, 92

reshape_2d
  DakotaMatrix, 117

resolve_inputs
  ParallelLibrary, 287

resolve_samples_symbols
  DACEIterator, 80

respLevelCount
  NonDAdvMeanValue, 258

response_mapping
  NestedModel, 252

responseArray
  DakotaModel, 126
  NestedModel, 250

responseASV
  DakotaResponseRep, 143

responseFn
  SurrogateDataPoint, 328

responseFnSamples
  NonDPCE, 265
  NonDProbability, 268
  NonDSampling, 272

responseGrad
  SurrogateDataPoint, 328

responseLevels
  DataMethod, 179

responseLevelTargets
  NonDAdvMeanValue, 257

responseList
  DakotaModel, 126
sampleReuse
  ApproximationInterface, 59
sampleType
  DataMethod, 179
  NonDPCE, 265
  NonDProbability, 268
  NonDSampling, 272
samplingReset
  DACEIterator, 78
  DakotaIterator, 108
  NonDProbability, 270
  NonDSampling, 272
samplingScheme
  DACEIterator, 78
  DakotaIterator, 108
  NonDProbability, 267
  NonDSampling, 271
SCAL
  CONMINOptimizer, 74
  KrigApprox, 228
scaleFactors
  LayeredModel, 233
searchVal
  FunctionCompare, 204
searchMethod
  DataMethod, 176
  SNLLOptimizer, 313
searchSchemeSize
  DataMethod, 177
secondaryCoeffs
  DataMethod, 175
selectedIterator
  BranchBndStrategy, 62
  ConcurrentStrategy, 67
  SingleMethodStrategy, 309
  SurrBasedOptStrategy, 320
selectedIterators
  MultiLevelOptStrategy, 246
selectionPressure
  DataMethod, 178
selfScheduleAnalyses
  ApplicationInterface, 55
selfScheduleEvaluations
  ApplicationInterface, 55
sendDbBuffer
  ProblemDescDB, 295
sendEa
  ParallelLibrary, 280
sendJe
  ParallelLibrary, 280
sendSi
  ParallelLibrary, 279
serve
  DakotaModel, 120
HierLayeredModel, 219
NestedModel, 249
SingleModel, 311
surfLayeredModel, 324
serve_analyses Asynch
  ForkApplicInterface, 203
serve_analyses Sync
  ApplicationInterface, 55
serve_evaluations
  ApplicationInterface, 57
serve_evaluationsPeer
  ApplicationInterface, 57
serve_evaluationsSync
  ApplicationInterface, 57
setBounds
  DakotaApproximation, 83
setDbInterfaceNode
  ProblemDescDB, 295
setDbListNodes
  ProblemDescDB, 295
setDbModelType
  ProblemDescDB, 300
setDbResponsesNode
  ProblemDescDB, 295
setLocalData
  DirectFnApplicInterface, 191
setMethodOptions
  SGOPTOptimizer, 308
setOtherListNodes
  ProblemDescDB, 297
setCell
  GetLongOpt, 213
setUpType
  NPSOLOptimizer, 277
SGOPTApplication, 303
~SGOPTApplication, 303
activeSetVector, 303
copy, 303
dakota_asynch_flag, 304
dakotaCompletionList, 304
dakotaModelAsynchFlag, 303
dakotaResponseList, 303
DoEval, 304
multiobjModifyPtr, 304
nextEval, 304
numNonlinCons, 304
numObjFns, 304
SGOPTApplication, 303
synchronize, 304
userDefinedModel, 303
sgoptApplication
SGOPTOptimizer, 308
SGOPTOptimizer, 306
~SGOPTOptimizer, 308
aPPSOptimizer, 307
baseOptimizer, 306
discreteAppFlag, 306
ePSAOptimizer, 307
exploratoryMoves, 306
findoptimum, 308
intProblem, 307
linConGenerator, 306
patternSearchOptimizer, 307
pGAIntOptimizer, 307
pGARealOptimizer, 307
realProblem, 307
setmethodoptions, 308
goptApplication, 308
SGOPTOptimizer, 308
SMCrealOptimizer, 307
SWOptimizer, 307
show data 3d
DakotaGraphics, 99
SingleMethodStrategy
~SingleMethodStrategy, 309
run_strategy, 309
selectedIterator, 309
SingleMethodStrategy, 309
userDefinedModel, 309
SingleMethodStrategy, 309
SingleModel
~SingleModel, 310
derived_asynchCompute_response, 310
derived_compute_response, 310
derived_init_communicators, 311
derived_master_overload, 310
derived_synchronize, 310
derived_synchronize_nowait, 310
free_communicators, 311
local_eval_synchronization, 310
new_eval_counter, 311
serve, 311
SingleModel, 310
stop_servers, 311
synchronize_nowait_completions, 310
total_eval_counter, 311
userDefinedInterface, 311
SingleModel, 310
SMCrealOptimizer
SGOPTOptimizer, 307
SNLLOptimizer, 312
~SNLLOptimizer, 312
bcdfnlf1, 314
bcnlf1, 314
bcnlf2, 314
constraint0_evaluator, 313
constraint1_evaluator, 313
constraint2_evaluator, 313
fdnlf1, 314
fdnlf1_evaluator, 316
fdnlf1Con, 314
find_optimum, 312
init_fn, 312
mfcn, 313
nlf0, 313
nlf0_evaluator, 316
nlf1, 313
nlf1_evaluator, 316
nlf1Con, 314
nlf2, 314
nlf2_evaluator, 316
nlf2Con, 314
nlfConstraint, 313
nlfObjective, 313
nlpConstraint, 313
optbanewton, 314
optbaqnewton, 315
optbcellipsoid, 315
optbnewton, 314
optbcqnewton, 314
optcg, 314
optdfnewton, 314
optfqups, 315
optnewton, 314
optnips, 315
optpd, 314
optqnewton, 314
optqnips, 315
s, 313
searchMethod, 313
SNLLOptimizer, 312
staticConstraintValues, 315
staticModeOverrideFlag, 315
staticNumNonlinearEqConstraints, 315
staticNumNonlinearIneqConstraints, 315
staticSpeculativeFlag, 315
theOptimizer, 314
vendorNumericalGradFlag, 313
soft_convergence_check
SurrBasedOptStrategy, 322
softConvCount
SurrBasedOptStrategy, 320
softConvLimit
SurrBasedOptStrategy, 320
solnAccuracy
DataMethod, 177
sort
DakotaList, 115

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
SortCompare
   operator(), 318
SortCompare, 318
sortFunction(), 318
SortCompare, 318
sortFunction
   SortCompare, 318
spawn
   SysCallAnalysisCode, 331
spawn application
   SysCallApplicationInterface, 333
spawn evaluation
   SysCallAnalysisCode, 331
spawn_input_filter
   SysCallAnalysisCode, 331
spawn_output_filter
   SysCallAnalysisCode, 331
speculativeFlag
   DakotaOptimizer, 135
DataMethod, 175

split
   CtelRegexp, 76
split_communicator_dedicated_master
   ParallelLibrary, 283
split_communicator_peer_partition
   ParallelLibrary, 283
startClock
   ParallelLibrary, 283
startCPUTime
   ParallelLibrary, 283
startMPTTime
   ParallelLibrary, 283
startWCTime
   ParallelLibrary, 283
state
   DataVariables, 184
   static_schedule_evaluations
   ApplicationInterface, 56
staticConstraintValues
   SNLLOptimizer, 315
staticFnGrads
   NonDAdvMeanValue, 257
   NonDAdvMeanValue, 257
staticFnVals
   NonDAdvMeanValue, 257
staticGlobalGradsU
   NonDAdvMeanValue, 257
staticGlobalGradsX
   NonDAdvMeanValue, 257
staticMaxFnEvals
   NPSOLOptimizer, 277
staticModel
   DakotaIterator, 110
staticModelOverrideFlag
   SNLLOptimizer, 315
staticMultiObjWeights
   DakotaOptimizer, 136
staticNumContinuousVars
   DakotaOptimizer, 135
staticNumFuncs
   NonDAdvMeanValue, 258
staticNumNonlinearConstraints
   DakotaOptimizer, 136
staticNumNonlinearEqConstraints
   SNLLOptimizer, 315
staticNumNonlinearIneqConstraints
   SNLLOptimizer, 315
staticNumObjFns
   DakotaOptimizer, 135
staticNumUncVars
   NonDAdvMeanValue, 258
staticSpeculativeFlag
   SNLLOptimizer, 315
staticVendorNumericalGradFlag
   NPSOLOptimizer, 277
statsFlag
   NonDPCE, 265
   NonDProbability, 268
   NonDSampling, 272
status
   CtelRegexp, 77
statusMsg
   CtelRegexp, 77
stdDevStats
   DakotaNonD, 132
stdResponse
   NonDAdvMeanValue, 257
stepLength
   DataMethod, 179
   ParamStudy, 292
stepLenToBoundary
   DataMethod, 177
stepVector
   DataMethod, 179
   ParamStudy, 292
stop_evaluation_servers
   ApplicationInterface, 55
   DakotaInterface, 101
stop_servers
   DakotaModel, 120
   HierLayeredModel, 219
   NestedModel, 249
   SingleModel, 311
   SurrLayeredModel, 324
strategy_dedicated_master_flag
   ParallelLibrary, 281
strategy_iterator_inter_communicator
   ParallelLibrary, 281
strategy_iterator_inter_communicators
numNonlinEqConstr, 321
numNonlinIneqConstr, 321
numObjFns, 321
numV ars, 320
penaltyParameter, 320
selectedIterator, 320
softConvCount, 320
softConvLimit, 320
SurrBasedOptStrategy, 319
trustRegionSize, 320
SurrBasedOptStrategy, 319
compute_penalty_function, 322
hard_convergence_check, 322
run_strategy, 322
soft_convergence_check, 322
SurrLayeredModel
~SurrLayeredModel, 323
approXInterface, 324
daceIterator, 325
daceMethodPointer, 324
free_communicators, 324
new_eval_counter, 324
serve, 324
stop_servers, 324
subordinate_iterator, 323
subordinate_model, 323
SurrLayeredModel, 323
synchronize_nowait_completions, 324
total_eval_counter, 324
update_approximation, 324
SurrLayeredModel, 323
actualInterfacePointer, 327
actualModel, 327
build_approximation, 326
derived_async_compute_response, 325
derived_compute_response, 325
derived_init_communicators, 326
derived_master_overload, 326
derived_synchronize, 325
derived_synchronize_nowait, 326
maximum_concurrency, 326
SurrogateDataPoint
~SurrogateDataPoint, 328
continuousVars, 328
operator=, 328
operator==, 328
responseFn, 328
responseGrad, 328
SurrogateDataPoint, 328
SurrogateDataPoint, 328
sWOptimizer
SGOPTOptimizer, 307
synchrony
ApplicationInterface, 54
ApproximationInterface, 58
DakotaInterface, 101
synchronize
ApplicationInterface, 54
ApproximationInterface, 59
DakotaInterface, 101
synchronize_nowait
DakotaInterface, 102
synchronize
DakotaModel, 121
SGOPTApplication, 304
synchronize_nowait
DakotaModel, 128
synchronize_nowait
DakotaModel, 121
synchronize_nowait_completions
DakotaModel, 120
HierLayeredModel, 219
NestedModel, 251
SingleModel, 310
SurrLayeredModel, 324
synchronous_local_analyses
ForkApplicationInterface, 203
synchronous_local_evaluations
ApplicationInterface, 56
SysCallAnalysisCode
~SysCallAnalysisCode, 330
command_usage, 330
commandUsage, 330
SysCallAnalysisCode, 330
SysCallAnalysisCode, 330
spawn_analysis, 331
spawn_evaluation, 331
spawn_input_filter, 331
spawn_output_filter, 331
SysCallApplicationInterface
~SysCallApplicationInterface, 332
derived_map, 332
derived_map_async, 332
derived_sync, 332
derived_sync_kernel, 333
derived_sync_nowait, 332
derived_synchronous_local_analyses, 332
failCountList, 333
failIdList, 333
SysCallApplicationInterface, 333
SysCallApplicInterface, 333
sysCallList, 333
sysCallList, 333
system_call_file_test, 333
SysCallApplicationInterface, 332
sysCallList
SysCallApplicationInterface, 333
sysCallSimulator

Generated on Mon Apr 1 11:51:37 2002 for DAKOTA by Doxygen written by Dimitri van Heesch © 1997-2001
uniformDistLowerBnds
  DakotaNonD, 131
uniformDistUpperBnds
  DakotaNonD, 131
uniformUncDistLowerBnds
  DataVariables, 186
uniformUncDistUpperBnds
  DataVariables, 187
unixCommand
  CommandShell, 65
update_all_continuous
  VariablesUtil, 336
update_all_discrete
  VariablesUtil, 336
update_approximation
  ApproximationInterface, 58
  DakotaInterface, 102
  DakotaModel, 119
  SurrLayeredModel, 324
update_best
  DACEIterator, 78
  ParamStudy, 292
update_labels
  VariablesUtil, 336
update_merged
  VariablesUtil, 336
update_response
  DakotaModel, 128
upper
  DakotaString, 153
upperBounds
  NPSOLOptimizer, 277
upperFactorHessianF77
  NPSOLOptimizer, 276
usage
  GetLongOpt, 212, 214
user_defined_model
  DakotaIterator, 108
userConstraintEval
  NPSOLOptimizer, 277
userDefinedInterface
  SingleModel, 311
userDefinedModel
  BranchBndStrategy, 62
  ConcurrentStrategy, 67
  DakotaIterator, 109
  SGOPTApplication, 303
  SingleMethodStrategy, 309
userDefinedModels
  MultilevelOptStrategy, 246
userDefinedVarConstraints
  DakotaModel, 124
userObjectiveEval
  NPSOLOptimizer, 277
unsqueeze
  GetLongOpt, 213
varConstraintsRep
  DakotaVarConstraints, 157
variablePartitions
  ParamStudy, 293
variables_kwhandler
  ProblemDescDB, 297
variables_type
  DakotaVariables, 162
variablesIndex
  ProblemDescDB, 299
variablesList
  ProblemDescDB, 299
variablesPointer
  DataMethod, 174
variablesRep
  DakotaVariables, 163
variablesType
  DakotaVarConstraints, 157
  DakotaVariables, 162
VariablesUtil
  ~VariablesUtil, 336
  update_all_continuous, 336
  update_all_discrete, 336
  update_labels, 336
  update_merged, 336
  VariablesUtil, 336
veloperseFlag
  AnalysisCode, 44
  DakotaIterator, 109
verboseOutput
  DakotaOptimizer, 135
SNLLOptimizer, 313
verifylevel
  DataMethod, 176
waitall
  ParallelLibrary, 280
weibullAlphas