SPARK Build Process and Problem Driver API Reference Manual

VisualSPARK 2.01

by Dimitri Curtil

Wed Nov 5 15:06:34 2003

Contents

1 SPARK Build Process and Problem Driver API 1

2 SPARK Build Process and Problem Driver API Namespace Documentation 3

2.1 SPARK Namespace Reference . . . 3

2.2 SPARK::DefaultComponentSettings Namespace Reference . . . 13

2.3 SPARK::DefaultGlobalSettings Namespace Reference . . . 16

2.4 SPARK::DefaultRuntimeControls Namespace Reference . . . 18

2.5 SPARK::Problem Namespace Reference . . . 20

2.6 SPARK::Problem::DynamicBuild Namespace Reference . . . 26

2.7 SPARK::Problem::StaticBuild Namespace Reference . . . 28

3 SPARK Build Process and Problem Driver API Class Documentation 31

3.1 SPARK::Problem::simulation_parameter< T > Class Template Reference . . . 31

3.2 SPARK::TComponent Class Reference . . . 34

3.3 SPARK::TComponentSettings Class Reference . . . 38

3.4 SPARK::TGlobalSettings Class Reference . . . 47

3.5 SPARK::TInverse Class Reference . . . 51

3.6 SPARK::TObject Class Reference . . . 54

3.7 SPARK::TPreferenceSettings Class Reference . . . 56

3.8 SPARK::TProblem Class Reference . . . 59

3.9 SPARK::TProblem::TState Class Reference . . . 69

3.10 SPARK::TRuntimeControls Class Reference . . . 72

3.11 SPARK::XAssertion Class Reference . . . 80

3.12 SPARK::XDimension Class Reference . . . 83

3.13 SPARK::XInitialization Class Reference . . . 85

3.14 SPARK::XIO Class Reference . . . 87

3.15 SPARK::XMemory Class Reference . . . 89

3.16 SPARK::XOutOfRange Class Reference . . . 91

3.17 SPARK::XStepper Class Reference . . . 93

3.18 SPARK::XTimeStep Class Reference . . . 95

4 SPARK Build Process and Problem Driver API File Documentation 97

4.1 component.h File Reference 97
4.2 ctrls.h File Reference 99
4.3 exceptions.h File Reference 100
4.4 exitcode.h File Reference 102
4.5 inverse.h File Reference 103
4.6 object.h File Reference 104
4.7 prefs.h File Reference 105
4.8 problem.h File Reference 106
4.9 sparkapi.h File Reference 108
4.10 sparkmacro.h File Reference 110
4.11 types.h File Reference 113

5 SPARK Build Process and Problem Driver API Example Documentation 115

5.1 multiproblem_example1.cpp 115
5.2 sparksolver.cpp 119

6 SPARK Build Process and Problem Driver API Page Documentation 125

6.1 Description of the SPARK Build Process 125
6.2 Generate the solution sequence for a SPARK problem 126
6.3 Build a SPARK problem statically 128
6.4 Build a SPARK problem dynamically 129
6.5 Automate the build process 131
6.6 Build a SPARK problem simulator with a customized driver 133
6.7 Implementation of the SPARK Driver Function 134
6.8 Header files to include in the C++ file implementing the driver function 135
6.9 Start the simulation session 136
6.10 Load a SPARK problem at runtime 137
6.11 Load a statically-built SPARK problem 139
6.12 Set up the runtime controls 141
6.13 Set up the preference settings 142
6.14 Solve the SPARK problem 143
6.15 Re-solve the SPARK problem with different runtime controls 145
6.16 End the simulation session 146
6.17 Catch the runtime exceptions 147
6.18 SPARK Problem Driver Application Programming Interface 148
6.19 Document Type Definition file 150
6.20 Makefile 154
6.21 Multi-problem command file 159

Chapter 1

SPARK Build Process and Problem Driver API

This document describes the overall build process to produce an executable SPARK simulator. It also explains step by step how to implement a driver function for a SPARK problem. For example, writing your own driver function will let you solve multiple problems sequentially as well as re-solve the same problem(s) with different runtime control parameters and/or preference settings (e.g., to perform sensitivity analysis or to recover from a failed simulation with a new set of solution settings).

- Description of the SPARK Build Process
- Implementation of the SPARK Driver Function
- SPARK Problem Driver Application Programming Interface

Chapter 2

SPARK Build Process and Problem Driver API Namespace Documentation

2.1 SPARK Namespace Reference

Definition of global functions and classes used in the SPARK simulation environment.

Classes

- class TRuntimeControls

  Wrapper class for all the runtime control information required to initialize a TProblem object in order to make a simulation run.

- class TGlobalSettings

  Class acts as repository of global control settings defined at the problem level.

- class TComponentSettings

  Class acts as repository of settings defined for each component.

- class TPreferenceSettings

  Wrapper class to store and manipulate information required to initialize the settings for the solution methods for each component.

- class TProblem

  Representation of a problem object in the SPARK solver.

- class TProblem::TState

  Interface class defining the methods used to save and restore the state of the problem using the TProblem::Save() and TProblem::Restore() methods.

- class TComponent

  Class that solves the set of DAE equations generated by setupcpp.

- class XAssertion

  Base class for all SPARK exceptions.

- class XDimension

  Indicates that a runtime error occured due to mismatched dimension.

- class XOutOfRange

  Indicates that a runtime error occured due to an out of range access operation on a container.

- class XMemory

  Indicates that a runtime error occured because memory could not be allocated.

- class XInitialization

  Indicates that a runtime error occured while initializing an object.

- class XIO

  Indicates that a runtime error occured while performing an IO operation.

- class XTimeStep

  Indicates that a runtime error occured while adapting the time step.

- class XStepper

  Inidicates that stepping to the next step failed.

- class TObject

  Class used to represent an instance of an inverse.

- class TInverse

  Class that defines the callbacks for an inverse.

Functions to manage the SPARK solving environment

- void Start (const char *sessionName, const char *runLogFilename, const char *errorLogFilename, const char *debugLogFilename, bool verbose=true) throw (SPARK::XInitialization)

  Starts the SPARK solving environment.

- void End ()

  Terminates the SPARK solving environment.

- void ExitWithError (SPARK::ExitCodes ec, const std::string &callingSub, const std::string &msg, const SPARK::TProblem *problem=0)

  Terminates program execution with exit code.

Utility functions

- char * GetFileName (unsigned argc, char *argv[ ], const char *extension) throw (SPARK::XInitialization)

  Returns the pointer to the first string in the argv[] that has the specified extension. If cannot find file with desired extension, returns 0.

- void Log (std::ostream &os, const char *strFileName, const char *strSenderName, const char *strMsg, const SPARK::TProblem *problem=0)

  Writes a message to the specified log file.

- void Log (std::ostream &os, const SPARK::TInverse *sender, unsigned line, const char *strMsg)

  Writes a message to the specified log file from a static callback file in a SPARK atomic class.

- void Log (std::ostream &os, const SPARK::TObject ∗sender, unsigned line, const char ∗strMsg)
  Writes a message to the specified log file from a non-static callback file in a SPARK atomic class.

Access methods

- const char ∗ GetProgramName ()
  Returns the name of the program as specified during the call to SPARK::Start().
- const char ∗ GetBaseName ()
  Returns the base name of the program name (i.e., the program name witout path and without any extension).
- const char ∗ GetVersion ()
  Returns the version of the solver library being used.
- std::ostream & GetRunLog ()
  Returns the output stream for the run log.
- std::ostream & GetErrorLog ()
  Returns the output stream for the error log.
- const char ∗ GetRunLogFilename ()
  Returns the name of the run log file as specified during the call to SPARK::Start().
- const char ∗ GetErrorLogFilename ()
  Returns the name of the error log file as specified during the call to SPARK::Start().
- const char ∗ GetDebugLogFilename ()
  Returns the name of the debug log file as specified during the call to SPARK::Start().

Enumerations

- enum ExitCodes {
  ExitCode_OK = 0,
  ExitCode_ERROR_IO = 100,
  ExitCode_ERROR_LEX_SCAN = 101,
  ExitCode_ERROR_URL = 102,
  ExitCode_ERROR_OUT_OF_MEMORY = 120,
  ExitCode_ERROR_NULL_POINTER = 121,
  ExitCode_ERROR_COMMAND_LINE = 130,
  ExitCode_ERROR_INVALID_RUN_CONTROLS = 131,
  ExitCode_ERROR_INVALID_PREFERENCES = 132,
  ExitCode_ERROR_INVALID_PROBLEM = 133,
  ExitCode_ERROR_EXIT_SPARK_FACTORY = 140,
  ExitCode_ERROR_RUNTIME_ERROR = 150,
  ExitCode_ERROR_INVALID_VARIABLE_NAME = 151,

ExitCode_ERROR_INVALID_FEATURE = 152,
ExitCode_ERROR_NUMERICAL = 160 }

Exit codes returned by the SPARK solver.

- enum CallbackTypes {
CallbackType_EVALUATE = 0,
CallbackType_PREDICT,
CallbackType_CONSTRUCT,
CallbackType_DESTRUCT,
CallbackType_PREPARE_STEP,
CallbackType_ROLLBACK,
CallbackType_COMMIT,
CallbackType_CHECK_INTEGRATION_STEP,
CallbackType_STATIC_CONSTRUCT,
CallbackType_STATIC_DESTRUCT,
CallbackType_STATIC_PREPARE_STEP,
CallbackType_STATIC_ROLLBACK,
CallbackType_STATIC_COMMIT,
CallbackType_STATIC_CHECK_INTEGRATION_STEP,
CALLBACKTYPES_L,
CallbackType_NONE = CALLBACKTYPES_L }

Enum for callback types.

- enum ProtoTypes {
ProtoType_MODIFIER = 0,
ProtoType_NON_MODIFIER,
ProtoType_PREDICATE,
ProtoType_STATIC_NON_MODIFIER,
ProtoType_STATIC_PREDICATE,
PROTOTYPES_L,
ProtoType_NONE = PROTOTYPES_L }

Enum for callback prototypes.

- enum ReturnTypes {
ReturnType_VALUE = 0,
ReturnType_RESIDUAL,
RETURNTYPES_L,
ReturnType_NONE = RETURNTYPES_L }

Enum for return types from modifier callbacks.

- enum VariableTypes {
VariableType_GLOBAL_TIME = 0,
VariableType_GLOBAL_TIME_STEP,
VariableType_PARAMETER,

VariableType_INPUT,
VariableType_UNKNOWN,
VARIABLETYPES_L,
VariableType_NONE = VARIABLETYPES_L }

Enum for the various variable types in the problem.

- enum RequestTypes {
RequestType_ABORT = 0,
RequestType_STOP,
RequestType_SET_STOP_TIME,
RequestType_REPORT,
RequestType_SNAPSHOT,
RequestType_SET_MEETING_POINT,
RequestType_CLEAR_MEETING_POINTS,
RequestType_RESTART,
RequestType_SET_TIME_STEP,
RequestType_SET_DYNAMIC_STEPPER,
REQUESTTYPES_L,
RequestType_NONE = REQUESTTYPES_L }

2.1.1 Detailed Description

Definition of global functions and classes used in the SPARK simulation environment.

2.1.2 Enumeration Type Documentation

2.1.2.1 enum SPARK::ExitCodes

Exit codes returned by the SPARK solver.

Enumeration values:

ExitCode_OK successful simulation
ExitCode_ERROR_IO error caused by writing to or reading from a file
ExitCode_ERROR_LEX_SCAN see exit(YY_EXIT_FAILURE) calls in lex_yy.c
ExitCode_ERROR_URL error caused by URL engine
ExitCode_ERROR_OUT_OF_MEMORY cannot allocate memory at run-time
ExitCode_ERROR_NULL_POINTER null pointer accessed somewhere
ExitCode_ERROR_COMMAND_LINE syntax problem
ExitCode_ERROR_INVALID_RUN_CONTROLS problem in the "∗.run" file
ExitCode_ERROR_INVALID_PREFERENCES problem in the "∗.prf" file
ExitCode_ERROR_INVALID_PROBLEM problem description is invalid
ExitCode_ERROR_EXIT_SPARK_FACTORY generated by runtime loader
ExitCode_ERROR_RUNTIME_ERROR generic runtime error ususally caused by container, adpators, iterators
ExitCode_ERROR_INVALID_VARIABLE_NAME accessing variable by invalid name
ExitCode_ERROR_INVALID_FEATURE trying to use a not yet supported feature
ExitCode_ERROR_NUMERICAL cannot converge after MaxIterations, singularity, division by zero...

2.1.2.2 enum SPARK::CallbackTypes

Enum for callback types.

Enumeration values:

CallbackType_EVALUATE Specifies an EVALUATE callback.

CallbackType_PREDICT Specifies a PREDICT callback.

CallbackType_CONSTRUCT Specifies a CONSTRUCT callback.

CallbackType_DESTRUCT Specifies a DESTRUCT callback.

CallbackType_PREPARE_STEP Specifies a PREPARE_STEP callback.

CallbackType_ROLLBACK Specifies a ROLLBACK callback.

CallbackType_COMMIT Specifies a COMMIT callback.

CallbackType_CHECK_INTEGRATION_STEP Specifies a CHECK_INTEGRATION_STEP callback for CLASSTYPE=INTEGRATOR only.

CallbackType_STATIC_CONSTRUCT Specifies a STATIC_CONSTRUCT callback.

CallbackType_STATIC_DESTRUCT Specifies a STATIC_DESTRUCT callback.

CallbackType_STATIC_PREPARE_STEP Specifies a STATIC_PREPARE_STEP callback.

CallbackType_STATIC_ROLLBACK Specifies a STATIC_ROLLBACk callback.

CallbackType_STATIC_COMMIT Specifies a STATIC_COMMIT callback.

CallbackType_STATIC_CHECK_INTEGRATION_STEP Specifies a STATIC_CHECK_INTEGRATION_STEP callback for CLASSTYPE=INTEGRATOR only.

CALLBACKTYPES_L Number of different callback types.

CallbackType_NONE Default value for non-initialized CallbackTypes variables.

2.1.2.3 enum SPARK::ProtoTypes

Enum for callback prototypes.

Enumeration values:

ProtoType_MODIFIER Prototype used by modifier callbacks (e.g., evaluate and predict).

ProtoType_NON_MODIFIER Prototype used by non-modifier callbacks (e.g., construct, destruct, ...).

ProtoType_PREDICATE Prototype used by predicate callbacks (e.g., check_integration_step, ...).

ProtoType_STATIC_NON_MODIFIER Prototype used by static non-modifier callbacks (e.g., static_construct, ...).

ProtoType_STATIC_PREDICATE Prototype used by static predicate callbacks (e.g., static_check_integration_step, ...).

PROTOTYPES_L Number of different prototype types.

ProtoType_NONE Default value for non-initialized ProtoTypes variables.

2.1.2.4 enum SPARK::ReturnTypes

Enum for return types from modifier callbacks.

Enumeration values:

ReturnType_VALUE Indicates that scalar value written to the target TVariable object is the variable value.

ReturnType_RESIDUAL Indicates that scalar value written to the target TVariable object is the residual value of the function matched with this variable.

RETURNTYPES_L Number of different return types.

ReturnType_NONE Default value for non-initialized ReturnTypes variables.

2.1.2.5 enum SPARK::VariableTypes

Enum for the various variable types in the problem.

Enumeration values:

VariableType_GLOBAL_TIME Refers to the LINK tagged with the GLOBAL_TIME keyword.

VariableType_GLOBAL_TIME_STEP Refers to the LINK tagged with the GLOBAL_TIME_TIME keyword.

VariableType_PARAMETER Refers to the LINKs specified with the PARAMETER keyword.

VariableType_INPUT Refers to the LINKs specified with the INPUT keyword.

VariableType_UNKNOWN Refers to the LINKs specified with none of the above keywords.

VARIABLETYPES_L Number of different variable types.

VariableType_NONE Default value for non-initialized VariableTypes variables.

2.1.2.6 enum SPARK::RequestTypes

Enum for the various atomic class requests

The values are specified by the following bit masks that can be combined using the bitwise OR (|) operator.

Enumeration values:

RequestType_ABORT Refers to a SPARK::TDispatcher::abort() request (see macro REQUEST__ABORT in spark.h).

RequestType_STOP Refers to a SPARK::TDispatcher::stop() request (see macro REQUEST__STOP in spark.h).

RequestType_SET_STOP_TIME Refers to a SPARK::TDispatcher::set_stop_time() request (see macro REQUEST__SET_-STOP_TIME in spark.h).

RequestType_REPORT Refers to a SPARK::TDispatcher::report() request (see macro REQUEST__REPORT in spark.h).

RequestType_SNAPSHOT Refers to a SPARK::TDispatcher::snapshot() request (see macro REQUEST__SNAPSHOT in spark.h).

RequestType_SET_MEETING_POINT Refers to a SPARK::TDispatcher::set_meeting_point() request (see macro REQUEST__SET_MEETING_POINT in spark.h).

RequestType_CLEAR_MEETING_POINTS Refers to a SPARK::TDispatcher::clear_meeting_points() request (see macro REQUEST__CLEAR_MEETING_POINTS in spark.h).

RequestType_RESTART Refers to a SPARK::TDispatcher::restart() request (see macro REQUEST__RESTART in spark.h).

RequestType_SET_TIME_STEP Refers to a SPARK::TDispatcher::set_time_step() request (see macro REQUEST__SET_-TIME_STEP in spark.h).

RequestType_SET_DYNAMIC_STEPPER Refers to a SPARK::TDispatcher::set_dynamic_stepper() request (see macro REQUEST__SET_DYNAMIC_STEPPER in spark.h).

REQUESTTYPES_L Number of different request types.

RequestType_NONE Default value for an invalid request type.

2.1.3 Function Documentation

2.1.3.1 void Start (const char ∗ *sessionName*, const char ∗ *runLogFilename*, const char ∗ *errorLogFilename*, const char ∗ *debugLogFilename*, bool *verbose* = true) throw (SPARK::XInitialization)

Starts the SPARK solving environment.

Parameters:

sessionName Name of the simulation session (used in various output files to identify run)

runLogFilename Name of the run log file where diagnostics for each problem is sent to

errorLogFilename Name of the log file where runtime errors will be notified

debugLogFilename Name of the log file where debug informatiion is written in `SPARK_DEBUG` mode

verbose If true then generates session diagnostic to run log files; if false session remains silent

Examples:

multiproblem_example1.cpp, and sparksolver.cpp.

2.1.3.2 void End ()

Terminates the SPARK solving environment.

Closes all log files. Performs garbage collection for all problems loaded at runtime.

Postcondition:

The SPARK::TProblem objects that have been constructed at runtime using the SPARK::Problem::DynamicLoad() function will be automatically destructed following the call to SPARK::End().

Warning:

Never explicitly destroy a TProblem object in your code!

Examples:

multiproblem_example1.cpp, and sparksolver.cpp.

2.1.3.3 void ExitWithError (SPARK::ExitCodes *ec*, const std::string & *callingSub*, const std::string & *msg*, const SPARK::TProblem ∗ *problem* = 0)

Terminates program execution with exit code.

Writes out to the run log output stream the error message contained in `ossErrMsg` if not NULL.

Parameters:

ec is the exit code of type SPARK::ExitCodes

callingSub contains the name of the caller / owner (e.g., problem, atomic class)

msg contains the description of the error message.

problem points to the problem object that is requesting to terminate the simulation (usually the currently active problem).

Postcondition:

The simulation is terminated properly with a call to the function SPARK::End(), therefore cleaning up log files and performing garbage collection.

Note:

If `ec == SPARK::ExitCode_OK` , then the function returns without doing anything.

Examples:

multiproblem_example1.cpp, and sparksolver.cpp.

2.1.3.4 char∗ GetFileName (unsigned *argc*, char ∗ *argv*[ ], const char ∗ *extension*) throw (SPARK::XInitialization)

Returns the pointer to the first string in the argv[] that has the specified extension. If cannot find file with desired extension, returns 0.

This function is more versatile than the SPARK::StaticBuild::ParseCommandLine() and SPARK::DynamicBuild::ParseCommandLine() functions as it lets you retrieve one file at a time for the specified extension.

Note:
The string comparison againt the specified extension is case insensitive.

Exceptions:
If the desired extension is invalid, throws the SPARK::XInitialization exception.

2.1.3.5 void Log (std::ostream & *os*, const char ∗ *strFileName*, const char ∗ *strSenderName*, const char ∗ *strMsg*, const SPARK::TProblem ∗ *problem* = 0)

Writes a message to the specified log file.

Parameters:
strFileName name of the atomic class file. Use preprocessor macro `__FILE__` to pass file name automatically.
strSenderName name of the sender object. E.g., pass C-string with inverse function name.
strMsg C-string message terminated by '\0' character to be displayed in error log file.
problem pointer to the target problem (if available)

2.1.3.6 void Log (std::ostream & *os*, const SPARK::TInverse ∗ *sender*, unsigned *line*, const char ∗ *strMsg*)

Writes a message to the specified log file from a static callback file in a SPARK atomic class.

Parameters:
sender pointer to the Inverse instance that sends the message
line indicates the line number in file where the message is sent from
strMsg C-string message terminated by '\n' character to be displayed in error log file.

2.1.3.7 void Log (std::ostream & *os*, const SPARK::TObject ∗ *sender*, unsigned *line*, const char ∗ *strMsg*)

Writes a message to the specified log file from a non-static callback file in a SPARK atomic class.

Parameters:
sender pointer to the TObject instance that sends the message
line indicates the line number in file where the message is sent from
strMsg C-string message terminated by '\n' character to be displayed in error log file.

2.1.3.8 const char∗ GetProgramName ()

Returns the name of the program as specified during the call to SPARK::Start().

Returns:
program name as `const char*`

2.1.3.9 const char* GetBaseName ()

Returns the base name of the program name (i.e., the program name witout path and without any extension).

Returns:
base name as `const char*`

2.1.3.10 const char* GetVersion ()

Returns the version of the solver library being used.

Returns:
full solver version as `const char*`

2.1.3.11 std::ostream& GetRunLog ()

Returns the output stream for the run log.

Returns:
output stream as `std::ostream&`

2.1.3.12 std::ostream& GetErrorLog ()

Returns the output stream for the error log.

Returns:
output stream as `std::ostream&`

2.1.3.13 const char* GetRunLogFilename ()

Returns the name of the run log file as specified during the call to SPARK::Start().

Returns:
run log file name as `const char*`

2.1.3.14 const char* GetErrorLogFilename ()

Returns the name of the error log file as specified during the call to SPARK::Start().

Returns:
error log file name as `const char*`

2.1.3.15 const char* GetDebugLogFilename ()

Returns the name of the debug log file as specified during the call to SPARK::Start().

Returns:
debug log file name as `const char*`

2.2 SPARK::DefaultComponentSettings Namespace Reference

Defines the default values for the component settings.

Component solution method

- const unsigned ComponentSolvingMethod = 0

  Newton-Raphson method.

- const unsigned MaxIterations = 50

  Default maximum number of iterations.

- const unsigned MinIterations = 1

  Default minimum number of iterations.

- const unsigned ScalingMethod = 0

  By default, no scaling is performed.

- const unsigned CheckBadNumericsFlag = 0

  Do not check for bad numerics by default [new feature added in SPARK 2.01].

Step control method

- const unsigned StepControlMethod = 1

  Default step control method is basic halving backtracking strategy [changed in SPARK 2.01].

- const double MaxRelaxationCoefficient = 1.0e+0

  Default maximum relaxation coefficient.

- const double MinRelaxationCoefficient = 1.0e-6

  Default minimum relaxatio coefficient.

Jacobian evaluation method

- const unsigned TrueJacobianEvalStep = 0

  Refresh Jacobian automatically.

- const double Epsilon = 0.0

  By default, use scaled perturbation.

- const double JacobianRefreshRatio = 0.5

  Jacobian refreshed if ResidualsNorm > JacobianRefreshRatio∗PrevIterationResidualsNorm by default.

Matrix solution method

- const unsigned MatrixSolvingMethod = 0
  Gaussian elimination method for dense matrix.
- const unsigned PivotingMethod = 1
  Partial pivoting by default.
- const unsigned RefinementMethod = 0
  No refinement iterations by default.

2.2.1 Detailed Description

Defines the default values for the component settings.

2.2.2 Variable Documentation

2.2.2.1 const unsigned SPARK::DefaultComponentSettings::ComponentSolvingMethod = 0

Newton-Raphson method.

2.2.2.2 const unsigned SPARK::DefaultComponentSettings::MaxIterations = 50

Default maximum number of iterations.

2.2.2.3 const unsigned SPARK::DefaultComponentSettings::MinIterations = 1

Default minimum number of iterations.

2.2.2.4 const unsigned SPARK::DefaultComponentSettings::ScalingMethod = 0

By default, no scaling is performed.

2.2.2.5 const unsigned SPARK::DefaultComponentSettings::CheckBadNumericsFlag = 0

Do not check for bad numerics by default [new feature added in SPARK 2.01].

2.2.2.6 const unsigned SPARK::DefaultComponentSettings::StepControlMethod = 1

Default step control method is basic halving backtracking strategy [changed in SPARK 2.01].

2.2.2.7 const double SPARK::DefaultComponentSettings::MaxRelaxationCoefficient = 1.0e+0

Default maximum relaxation coefficient.

2.2.2.8 const double SPARK::DefaultComponentSettings::MinRelaxationCoefficient = 1.0e-6

Default minimum relaxatio coefficient.

2.2.2.9 const unsigned SPARK::DefaultComponentSettings::TrueJacobianEvalStep = 0

Refresh Jacobian automatically.

2.2.2.10 const double SPARK::DefaultComponentSettings::Epsilon = 0.0

By default, use scaled perturbation.

2.2.2.11 const double SPARK::DefaultComponentSettings::JacobianRefreshRatio = 0.5

Jacobian refreshed if ResidualsNorm $>$ JacobianRefreshRatio$*$PrevIterationResidualsNorm by default.

2.2.2.12 const unsigned SPARK::DefaultComponentSettings::MatrixSolvingMethod = 0

Gaussian elimination method for dense matrix.

2.2.2.13 const unsigned SPARK::DefaultComponentSettings::PivotingMethod = 1

Partial pivoting by default.

2.2.2.14 const unsigned SPARK::DefaultComponentSettings::RefinementMethod = 0

No refinement iterations by default.

2.3 SPARK::DefaultGlobalSettings Namespace Reference

Defines the default values for the global settings.

Tolerance settings

- const double Tolerance = 1.0e-6

  Relative tolerance used throughout the simulation.

- const double MaxTolerance = 1.0e-3

  Looser relative tolerance used to recover if convergence cannot be achieved against Tolerance.

Safety factors

Safety factors are used to modify the convergence check depending on the context (i.e., check after prediction step or after an iteration) and on the type of the unknwon variable (i.e., a normal unknwon or a break unknown). A safety factor smaller than 1 makes the convergence check tougher to satisfy, with 0 making it impossible. A safety factory larger than 1 essentially corresponds to a relaxation of the convergence check. E.g, if `IterationSafetyFactor=10` then the convergence check at each iteration after the prediction will be made 10 times easier to satsify than the specified tolerence, because the convergence error must be samller than $<$ SafetyFactor$*$Tolerance. The default safety factors are equal to 1. therefore, they treat break and normal unknowns with the same weight in the convergence check. For certain applications, it might be useful to only perform a convergence check on the break variables. This is achieved by setting the NormalUnknownSafetyFactor entry to a number larger than 1.

- const double PredictionSafetyFactor = 0.01

  Safety factor used to check convergence after prediction [feature added in SPARK 2.0].

- const double IterationSafetyFactor = 0.9

  Safety factor used to check convergence after each iteration [feature added in SPARK 2.0].

- const double BreakUnknownSafetyFactor = 1.0

  Safety factor used to check convergence for a break unknown [feature added in SPARK 2.0].

- const double NormalUnknownSafetyFactor = 1.0

  Safety factor used to check convergence for a normal unknown (i.e., not a break variable) [feature added in SPARK 2.0].

2.3.1 Detailed Description

Defines the default values for the global settings.

2.3.2 Variable Documentation

2.3.2.1 const double SPARK::DefaultGlobalSettings::Tolerance = 1.0e-6

Relative tolerance used throughout the simulation.

2.3.2.2 const double SPARK::DefaultGlobalSettings::MaxTolerance = 1.0e-3

Looser relative tolerance used to recover if convergence cannot be achieved against Tolerance.

2.3.2.3 const double SPARK::DefaultGlobalSettings::PredictionSafetyFactor = 0.01

Safety factor used to check convergence after prediction [feature added in SPARK 2.0].

2.3.2.4 const double SPARK::DefaultGlobalSettings::IterationSafetyFactor = 0.9

Safety factor used to check convergence after each iteration [feature added in SPARK 2.0].

2.3.2.5 const double SPARK::DefaultGlobalSettings::BreakUnknownSafetyFactor = 1.0

Safety factor used to check convergence for a break unknown [feature added in SPARK 2.0].

2.3.2.6 const double SPARK::DefaultGlobalSettings::NormalUnknownSafetyFactor = 1.0

Safety factor used to check convergence for a normal unknown (i.e., not a break variable) [feature added in SPARK 2.0].

2.4 SPARK::DefaultRuntimeControls Namespace Reference

Default controls are defined in this namespace. The default controls are used if no entry is found in the *.run file for the control in question.

Variables

- const double InitialTime = 0.0

  Default initial time.

- const double FinalTime = 0.0

  Default final time.

- const double InitialTimeStep = 1.0

  Default initial time step value.

- const double MinTimeStep = 1.0e-6

  Default min time step value [feature added in SPARK 2.0].

- const double MaxTimeStep = 1.0e+6

  Default max time step value [feature added in SPARK 2.0].

- const double ReportCycle = InitialTimeStep

  Default report cycle is equal to initial time step.

- const double FirstReport = InitialTime

  Default first report is at inital time.

- const unsigned VariableTimeStep = 0

  Constant time step mode by default [feature added in SPARK 2.0].

- const unsigned ConsistentInitialCalculation = 1

  Consistent initalization calculation is required by default [feature added in SPARK 2.0].

- const unsigned DiagnosticLevel = 0

  Silent diagnostic mode by default.

2.4.1 Detailed Description

Default controls are defined in this namespace. The default controls are used if no entry is found in the *.run file for the control in question.

2.4.2 Variable Documentation

2.4.2.1 const double SPARK::DefaultRuntimeControls::InitialTime = 0.0

Default initial time.

2.4.2.2 const double SPARK::DefaultRuntimeControls::FinalTime = 0.0

Default final time.

2.4.2.3 const double SPARK::DefaultRuntimeControls::InitialTimeStep = 1.0

Default initial time step value.

2.4.2.4 const double SPARK::DefaultRuntimeControls::MinTimeStep = 1.0e-6

Default min time step value [feature added in SPARK 2.0].

2.4.2.5 const double SPARK::DefaultRuntimeControls::MaxTimeStep = 1.0e+6

Default max time step value [feature added in SPARK 2.0].

2.4.2.6 const double SPARK::DefaultRuntimeControls::ReportCycle = InitialTimeStep

Default report cycle is equal to initial time step.

2.4.2.7 const double SPARK::DefaultRuntimeControls::FirstReport = InitialTime

Default first report is at inital time.

2.4.2.8 const unsigned SPARK::DefaultRuntimeControls::VariableTimeStep = 0

Constant time step mode by default [feature added in SPARK 2.0].

2.4.2.9 const unsigned SPARK::DefaultRuntimeControls::ConsistentInitialCalculation = 1

Consistent initalization calculation is required by default [feature added in SPARK 2.0].

2.4.2.10 const unsigned SPARK::DefaultRuntimeControls::DiagnosticLevel = 0

Silent diagnostic mode by default.

2.5 SPARK::Problem Namespace Reference

Definition of the SPARK problem driver API library used to manage the SPARK::TProblem objects in the driver function.

Classes

- class simulation_parameter

    Class used to specify a simulation parameter.

Simulation methods

- typedef simulation_parameter< bool > TRestartFlag

    Parameter type describing the restart flag.

- typedef simulation_parameter< double > TStopTime

    Parameter type describing the stop time.

- typedef simulation_parameter< double > TTimeStep

    Parameter type describing the time step.

- SPARK::TProblem::SimulationFlags Simulate (SPARK::TProblem ∗instance, const SPARK::Problem::TRestartFlag &restartFlag, const SPARK::Problem::TStopTime &stopTime, const SPARK::Problem::TTimeStep &initialTimeStep=SPARK::Problem::TTimeStep())

    Simulates the `instance` *problem until the final time or the stopping time is reached, whichever occurs first.*

- SPARK::TProblem::SimulationFlags Step (SPARK::TProblem ∗instance, const SPARK::Problem::TTimeStep &timeStep=SPARK::Problem::TTimeStep())

    Computes the next step of the `instance` *problem and returns the simulation flag.*

- SPARK::TProblem::SimulationFlags StaticStep (SPARK::TProblem ∗instance)

    Computes one static step for the `instance` *problem and returns the simulation flag.*

Methods used to setup a problem for simulation

- bool RegisterStaticInstance (const char ∗pbName, SPARK::TProblem ∗instance)

    Registers statically-built problem by address with static repository.

- bool Unload (const char ∗pbName)

    Unloads the problem instance named "pbName" from repository and frees memory.

- void WriteInstances (std::ostream &os, const std::string &before)

    Writes out the list of loaded problem instances to the output stream `os` *with leading string* `before`.

- SPARK::TProblem ∗ Get (const char ∗pbName)

    Access method to retrieve a previously loaded problem through its unique name from the problem repository.

- void Initialize (SPARK::TProblem ∗instance, const SPARK::TRuntimeControls &controls)

    Initializes the problem with the specified runtime controls.

- void LoadPreferenceSettings (SPARK::TProblem *instance, const SPARK::TPreferenceSettings &preferences)

  Loads the preference settings into the problem.

- void Terminate (SPARK::TProblem *instance)

  Terminates the problem.

State management methods

- void Save (const SPARK::TProblem *instance, SPARK::TProblem::TState &state)

  Saves the state of the problem at the current time.

- void Restore (SPARK::TProblem *instance, const SPARK::TProblem::TState &state)

  Restores the problem to the specified state.

2.5.1 Detailed Description

Definition of the SPARK problem driver API library used to manage the SPARK::TProblem objects in the driver function.

2.5.2 Typedef Documentation

2.5.2.1 typedef simulation_parameter<bool> SPARK::Problem::TRestartFlag

Parameter type describing the restart flag.

2.5.2.2 typedef simulation_parameter<double> SPARK::Problem::TStopTime

Parameter type describing the stop time.

2.5.2.3 typedef simulation_parameter<double> SPARK::Problem::TTimeStep

Parameter type describing the time step.

2.5.3 Function Documentation

2.5.3.1 bool RegisterStaticInstance (const char * *pbName*, SPARK::TProblem * *instance*)

Registers statically-built problem by address with static repository.

Parameters:

pbName unique problem name

instance address of the SPARK::TProblem instance

Returns:

Returns true if operation was successful, false otherwise

Note:

Used in "problem.cpp" file for precompiled problem file.
The name of the problem instance must be unique for this simulation session, otherwise the SPARK solver will not be able to load this problem at runtime. This will result in the SPARK problem loader throwing a SPARK::XInitialization exception with the exit code SPARK::ExitCode_ERROR_INVALID_PROBLEM when calling the function SPARK::Start().

Warning:

Since this function is typically invoked in the "problem.cpp" file, it is called at startup before entering the main function. Therefore, no error message can be reported to a log file. SPARK::Start() will throw a SPARK::XInitialization exception if any such error occurred at startup.

2.5.3.2 bool Unload (const char * *pbName*)

Unloads the problem instance named "pbName" from repository and frees memory.

Returns:

Returns true if instance successfully unloaded; false otherwise

Parameters:

pbName name of the problem to unload

Note:

If there is no problem instance named after "pbName", the function will return false.
This function unloads both statically-built problems and problems loaded at runtime , i.e. problem instances loaded:

- either with a call to SPARK::Problem::StaticBuild::Load() or
- with a call to SPARK::Problem::DynamicBuild::Load().

Warning:

An unloaded problem can no longer be:

- used in the driver function or
- retrieved with a call to the function SPARK::Problem::Get().

2.5.3.3 void WriteInstances (std::ostream & *os*, const std::string & *before*)

Writes out the list of loaded problem instances to the output stream `os` with leading string `before`.

Parameters:

os Output stream for write operation

before Prefix string written before the names of the instances

2.5.3.4 SPARK::TProblem* Get (const char * *pbName*)

Access method to retrieve a previously loaded problem through its unique name from the problem repository.

Returns:

Address of the SPARK::TProblem instance describing the problem named "pbName"

Parameters:

pbName name of the problem whose address is to be returned

Note:
Returns NULL if problem name was not previouly loaded or if no problem with this name exists in global repository.

Examples:
multiproblem_example1.cpp, and sparksolver.cpp.

2.5.3.5 void Initialize (SPARK::TProblem ∗ *instance*, const SPARK::TRuntimeControls & *controls*)

Initializes the problem with the specified runtime controls.

Calls the SPARK::TProblem::Initialize() method.

Parameters:
instance Address of the SPARK::TProblem object

controls Runtime controls as specified in a ∗.run file

Examples:
multiproblem_example1.cpp.

2.5.3.6 void LoadPreferenceSettings (SPARK::TProblem ∗ *instance*, const SPARK::TPreferenceSettings & *preferences*)

Loads the preference settings into the problem.

Calls the SPARKK::TProblem::LoadPreferenceSettings() method.

Parameters:
instance Address of the SPARK::TProblem object

preferences Preference settings as specified in a ∗.prf file

Examples:
multiproblem_example1.cpp.

2.5.3.7 void Terminate (SPARK::TProblem ∗ *instance*)

Terminates the problem.

Calls the SPARK::TProblem::Terminate() method.

Postcondition:
The problem must be re-initialized with a call to SPARK::TProblem::Initialize() to allow for a new simulation.

Parameters:
instance Address of the SPARK::TProblem object

Examples:
multiproblem_example1.cpp.

2.5.3.8 void Save (const SPARK::TProblem * *instance*, SPARK::TProblem::TState & *state*)

Saves the state of the problem at the current time.

Calls the SPARK::TProblem::Save() method.

Postcondition:
The state object will contain the new state. If it contained another state, it will be overriden. The problem object remains unchanged.

Parameters:
instance Address of the SPARK::TProblem object
state SPARK::TProblem::TState object containing the stored state

2.5.3.9 void Restore (SPARK::TProblem * *instance*, const SPARK::TProblem::TState & *state*)

Restores the problem to the specified state.

Calls the SPARK::TProblem::Restore() method.

Precondition:
The state to restore must have been saved with a call to SPARK::Problem::Save().

Postcondition:
The problem is restored to the state, in particular the global time, the current values of all variables as well as their history.

Parameters:
instance Address of the SPARK::TProblem object
state SPARK::TProblem::TState object containing the state to restore

2.5.3.10 SPARK::TProblem::SimulationFlags Simulate (SPARK::TProblem * *instance*, const SPARK::Problem::TRestartFlag & *restartFlag*, const SPARK::Problem::TStopTime & *stopTime*, const SPARK::Problem::TTimeStep & *initialTimeStep* = SPARK::Problem::TTimeStep())

Simulates the `instance` problem until the final time or the stopping time is reached, whichever occurs first.

This function calls the TProblem:Simulate() method after having sent the appropriate requests for the specified simulation parameters. Uses the initial time step if specified. By default there is no initial time step parameter. Starts with a static step if the restart flag is set to true.

Precondition:
The problem must have been initialized.

Returns:
Simulation flag

Parameters:
instance Address of the SPARK::TProblem object
restartFlag Boolean flag. If true, forces the simulation to (re-)start with a static step to ensure consistent initialization.
stopTime Value of the desired stopping time as a double parameter. If not specified, the simulation ends when the final time specified in the runtime controls is reached.
initialTimeStep Value of the candidate initial time step as a double parameter.

Examples:
multiproblem_example1.cpp.

2.5.3.11 SPARK::TProblem::SimulationFlags Step (SPARK::TProblem * *instance*, const SPARK::Problem::TTimeStep & *timeStep* = SPARK::Problem::TTimeStep())

Computes the next step of the `instance` problem and returns the simulation flag.

This function essentially sends a stop() request to the instance problem and calls the TProblem::Simulate() method. This forces the finite-state machine to stop after the first step.

If the finite-state machine is set to perform a static step following a prior restart request, then the next step is a static step. Otherwise, it will perform the next dynamic step with the candidate time step if specified.

Note:

When calling this function make sure that the runtime controls do not request a final snapshot file as this would be generated each time the problem steps forward, therefore being very resource intensive. When stepping the problem as opposed to simulating it, prefer requesting a snapshot file using the snapshot request when needed.

Precondition:

The problem must have been initialized.

Parameters:

instance Address of the SPARK::TProblem object

timeStep Candidate time step

2.5.3.12 SPARK::TProblem::SimulationFlags StaticStep (SPARK::TProblem * *instance*)

Computes one static step for the `instance` problem and returns the simulation flag.

This function essentially sends a stop() request to the instance problem and calls the TProblem::Simulate() method. This forces the finite-state machine to stop after the first step.

Precondition:

The problem must have been initialized.

Parameters:

instance Address of the SPARK::TProblem object

2.6 SPARK::Problem::DynamicBuild Namespace Reference

Definition of the functions used to load a SPARK problem at runtime.

Functions

- void ParseCommandLine (unsigned argc, char **argv, char *&runFileName, char *&prfFileName, char *&xmlFileName) throw (SPARK::XInitialization)

  Parses up to argc command-line arguments in argv[] to produce the name of the files with extensions ".run", "*.prf" and "*.xml" as needed to run a problem loaded at runtime.*

- void ShowCommandLineUsage (std::ostream &os)

  Writes command-line usage of stand-alone SPARK engine for a problem loaded at runtime to output stream `os`.

- bool Load (const char *pbName, const char *xmlFileName)

  Loads the problem named "pbName" at runtime and sets name of problem instance after unique identifier "pbName".

2.6.1 Detailed Description

Definition of the functions used to load a SPARK problem at runtime.

These API functions are used to implement the driver function of the stand-alone SPARK engine.

2.6.2 Function Documentation

2.6.2.1 void ParseCommandLine (unsigned *argc*, char ** *argv*, char *& *runFileName*, char *& *prfFileName*, char *& *xmlFileName*) throw (SPARK::XInitialization)

Parses up to argc command-line arguments in argv[] to produce the name of the files with extensions "*.run", "*.prf" and "*.xml" as needed to run a problem loaded at runtime.

Note:

No memory is allocated for the 3 strings passed in the argument list. Upon returning from this function, the char*& are pointing to the entries in argv[] that contain the corresponding file names with the correct extensions.

Parameters:

argc first argument of the main() function declared in the "main.cpp" file

argv second argument of the main() function declared in the "main.cpp" file

runFileName points to argv[] with name of the file with extension "*.run"

prfFileName points to argv[] with name of the file with extension "*.prf"

xmlFileName points to argv[] with name of the file with extension "*.xml"

Exceptions:

SPARK::XInitialization Thrown if more than one *.run or *.prf or *.xml file is detected at the command-line.

Examples:

sparksolver.cpp.

2.6.2.2 void ShowCommandLineUsage (std::ostream & *os*)

Writes command-line usage of stand-alone SPARK engine for a problem loaded at runtime to output stream `os`.

Parameters:

os output stream where to write the command-line usage

2.6.2.3 bool Load (const char * *pbName*, const char * *xmlFileName*)

Loads the problem named "pbName" at runtime and sets name of problem instance after unique identifier "pbName".

Returns:

True if operation was successful, false otherwise.

Parameters:

pbName the unique name for this instance of the problem

xmlFileName name of the xml file with the problem description

Postcondition:

If the problem was loaded successfully, then the SPARK::TProblem instance is stored in the global problem repository with its name for later retrieval through a call to SPARK::Problem::Get().
Furthermore, if the problem was loaded successfully using the runtime loading mechanism, then the problem factory used to construct the problem is stored in the factory repository to support garbage collection upon the call to the function SPARK::End().

Examples:

multiproblem_example1.cpp, and sparksolver.cpp.

2.7 SPARK::Problem::StaticBuild Namespace Reference

Definition of the functions used to load a statically-built SPARK problem.

Functions

- void ParseCommandLine (unsigned argc, char **argv, char *&runFileName, char *&prfFileName) throw (SPARK::XInitialization)

  Parses up to argc command-line arguments in argv[] to produce the names of the files with extensions ".run" and "*.prf" as needed to run a statically-built problem.*

- void ShowCommandLineUsage (std::ostream &os)

  Writes command-line usage of stand-alone SPARK engine for a statically-built problem to output stream `os`.

- bool Load (const char *pbName)

  Loads the statically-built problem named "pbName".

2.7.1 Detailed Description

Definition of the functions used to load a statically-built SPARK problem.

These API functions are used to implement the driver function of the stand-alone SPARK engine.

2.7.2 Function Documentation

2.7.2.1 void ParseCommandLine (unsigned *argc*, char ** *argv*, char *& *runFileName*, char *& *prfFileName*) throw (SPARK::XInitialization)

Parses up to argc command-line arguments in argv[] to produce the names of the files with extensions "*.run" and "*.prf" as needed to run a statically-built problem.

This function is used to extract the names of the *.run and *.prf files at the command line as required by a statically-built simulator, i.e. a simulator that does not need the *.xml file to load the problem description as it is compiled and linked along the driver function.

Note:

No memory is allocated for the 2 strings passed in the argument list. Upon returning from this function, the char*& are pointing to the entries in argv[] that contain the corresponding file names with the correct extensions.

Parameters:

argc first argument of the main() function declared in the "main.cpp" file

argv second argument of the main() function declared in the "main.cpp" file

runFileName points to argv[] with name of the file with extension "*.run"

prfFileName points to argv[] with name of the file with extension "*.prf"

Exceptions:

SPARK::XInitialization Thrown if more than one *.run or *.prf file is detected at the command-line.

Examples:

sparksolver.cpp.

2.7.2.2 void ShowCommandLineUsage (std::ostream & *os*)

Writes command-line usage of stand-alone SPARK engine for a statically-built problem to output stream `os`.

Parameters:

os output stream where to write the command-line usage

2.7.2.3 bool Load (const char ∗ *pbName*)

Loads the statically-built problem named "pbName".

By statically-built problem, we refer to a problem whose definition is compiled from the corresponding "problem.cpp" file and linked along with the solver library and driver function.

Returns:

True if operation was successful, false otherwise.

Parameters:

pbName the unique name for this instance of the problem

Postcondition:

If the problem was loaded successfully, then the SPARK::TProblem instance is stored in the global problem repository with its name for later retrieval through a call to SPARK::Problem::Get().

Examples:

sparksolver.cpp.

Chapter 3

SPARK Build Process and Problem Driver API Class Documentation

3.1 SPARK::Problem::simulation_parameter< T > Class Template Reference

Class used to specify a simulation parameter.

```
#include <sparkapi.h>
```

Public Types

- typedef T value_type

  template type for the parameter

Public Member Functions

- simulation_parameter ()

  Default constructor: empty parameter.

- simulation_parameter (T value)

  Explicit constructor: stores value as parameter.

- simulation_parameter (const simulation_parameter &p)

  Copy constructor: deep copy of parameter value.

- bool empty () const

  Returns true if parameter is empty, false otherwise.

- value_type get () const

  Returns a copy of the parameter value.

- operator value_type () const

  Automatic conversion to parameter type.

3.1.1 Detailed Description

template<typename T> class SPARK::Problem::simulation_parameter< T >

Class used to specify a simulation parameter.

Wrapper class around a POD object of type T that keeps track of whether the parameter was initialized in the constructor. Used in the various simulation methods that accept parameters as arguments.

Examples:
multiproblem_example1.cpp.

3.1.2 Member Typedef Documentation

3.1.2.1 template<typename T> typedef T SPARK::Problem::simulation_parameter< T >::value_type

template type for the parameter

3.1.3 Constructor & Destructor Documentation

3.1.3.1 template<typename T> SPARK::Problem::simulation_parameter< T >::simulation_parameter () `[inline]`

Default constructor: empty parameter.

3.1.3.2 template<typename T> SPARK::Problem::simulation_parameter< T >::simulation_parameter (T *value*) `[inline, explicit]`

Explicit constructor: stores value as parameter.

3.1.3.3 template<typename T> SPARK::Problem::simulation_parameter< T >::simulation_parameter (const simulation_parameter< T > & *p*) `[inline]`

Copy constructor: deep copy of parameter value.

3.1.4 Member Function Documentation

3.1.4.1 template<typename T> bool SPARK::Problem::simulation_parameter< T >::empty () const `[inline]`

Returns true if parameter is empty, false otherwise.

3.1.4.2 template<typename T> value_type SPARK::Problem::simulation_parameter< T >::get () const `[inline]`

Returns a copy of the parameter value.

3.1.4.3 template<typename T> SPARK::Problem::simulation_parameter< T >::operator value_type () const `[inline]`

Automatic conversion to parameter type.

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

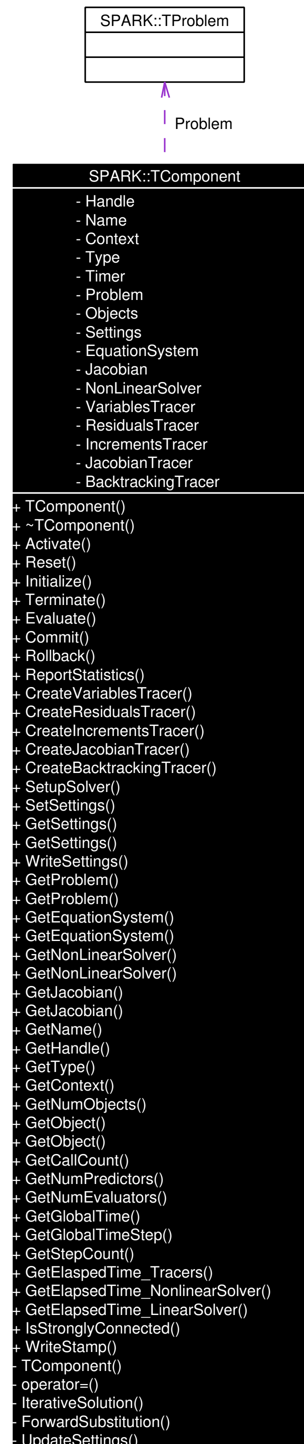
- sparkapi.h

3.2 SPARK::TComponent Class Reference

Class that solves the set of DAE equations generated by setupcpp.

`#include <component.h>`

Collaboration diagram for SPARK::TComponent:



Public Types

- enum ComponentTypes {
  ComponentType_WEAK = 0,
  ComponentType_STRONG = 1 }

  Type to indicate whether a component consists of a set of weakly- or strongly-connected equations.

Public Member Functions

- TComponent (unsigned handle, unsigned numNormalUnknowns, SPARK::TUnknown normalUnknowns[ ], unsigned numBreakUnknowns, SPARK::TUnknown breakUnknowns[ ], unsigned numObjects, SPARK::TObject ∗objects[ ]) throw (SPARK::XMemory)

  Constructor.

- ∼TComponent () throw ()

  Destructor.

- void Evaluate ()

  Solves the set of differential-algebraic equations.

- const char ∗ GetName () const

  Returns the name of the component as const char∗.

- unsigned GetHandle () const

  Returns the unique handle assigne to this component as an unsigned int.

- unsigned GetType () const

  Returns the component type as unsigned (see enum TComponent::TType).

- unsigned GetNumObjects () const

  Returns the number of objects comprising this component.

- SPARK::TObject ∗ GetObject (unsigned i)

  Returns a pointer to the TObject object evaluated in position i.

- const SPARK::TObject ∗ GetObject (unsigned i) const

  Returns a const pointer to the TObject object evaluated in position i.

- bool IsStronglyConnected () const

  Returns true if this component is a strongly-connected component.

3.2.1 Detailed Description

Class that solves the set of DAE equations generated by setupcpp.

A problem consists of a series of TComponent objects that are solved in a fixed order, from the first component (with index 0) to the last component. The order in which the components need to be solved reflects the topological dependencies derived by setupcpp.

3.2.2 Member Enumeration Documentation

3.2.2.1 enum SPARK::TComponent::ComponentTypes

Type to indicate whether a component consists of a set of weakly- or strongly-connected equations.

Enumeration values:

ComponentType_WEAK Indicates a weakly-connected component that is solved using forward substitution.

ComponentType_STRONG Indicates a strongly-connected component that requires iterative solution.

3.2.3 Constructor & Destructor Documentation

3.2.3.1 SPARK::TComponent::TComponent (unsigned *handle*, unsigned *numNormalUnknowns*, SPARK::TUnknown *normalUnknowns*[ ], unsigned *numBreakUnknowns*, SPARK::TUnknown *breakUnknowns*[ ], unsigned *numObjects*, SPARK::TObject ∗ *objects*[ ]) throw (SPARK::XMemory)

Constructor.

3.2.3.2 SPARK::TComponent::∼TComponent () throw ()

Destructor.

3.2.4 Member Function Documentation

3.2.4.1 void SPARK::TComponent::Evaluate ()

Solves the set of differential-algebraic equations.

Exceptions:

SPARK::XNoConvergence Thrown if convergence cannot be obtained.

SPARK::XBadNumerics Thrown if bad numerics detected.

3.2.4.2 const char∗ SPARK::TComponent::GetName () const `[inline]`

Returns the name of the component as const char∗.

3.2.4.3 unsigned SPARK::TComponent::GetHandle () const `[inline]`

Returns the unique handle assigne to this component as an unsigned int.

3.2.4.4 unsigned SPARK::TComponent::GetType () const `[inline]`

Returns the component type as unsigned (see enum TComponent::TType).

3.2.4.5 unsigned SPARK::TComponent::GetNumObjects () const `[inline]`

Returns the number of objects comprising this component.

3.2.4.6 SPARK::TObject* SPARK::TComponent::GetObject (unsigned *i*) [inline]

Returns a pointer to the TObject object evaluated in position i.

Note:
The first object evaluated in this component is stored in position 0.

3.2.4.7 const SPARK::TObject* SPARK::TComponent::GetObject (unsigned *i*) const [inline]

Returns a const pointer to the TObject object evaluated in position i.

Note:
The first object evaluated in this component is stored in position 0.

3.2.4.8 bool SPARK::TComponent::IsStronglyConnected () const [inline]

Returns true if this component is a strongly-connected component.

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

- component.h

3.3 SPARK::TComponentSettings Class Reference

Class acts as repository of settings defined for each component.

`#include <prefs.h>`

Public Member Functions

Structors

- TComponentSettings ()
  Initialize all settings to compile-time default values.
- TComponentSettings (const TComponentSettings &)
  Copy constructor.
- TComponentSettings & operator= (const TComponentSettings &)
  Assignement operator.
- ∼TComponentSettings () throw ()
  Trivial destructor.

Main operations

- void Reset ()
  Resets settings to default values.
- void Load (SPARK::TPrefList ∗prefList)
  Loads all settings from prefList.

Access methods for nonlinear solver

- unsigned GetComponentSolvingMethod () const
  Returns the code for the component solving method as `unsigned`.
- void SetComponentSolvingMethod (unsigned method)
  Sets the code for the component solving method to `method`.
- unsigned GetMaxIterations () const
  Returns the maximum number of iterations allowed in the nonlinear solver as `unsigned`.
- void SetMaxIterations (unsigned maxIterations)
  Sets the maximum number of iterations allowed in the nonlinear solver to `maxIterations`.
- unsigned GetMinIterations () const
  Returns the minimum number of iterations allowed in the nonlinear solver as `unsigned`.
- void SetMinIterations (unsigned minIterations)
  Sets the minimum number of iterations allowed in the nonlinear solver to `minIterations`.
- unsigned GetCheckBadNumericsFlag () const
  Returns the boolean flag (0=false | 1=true) indicating whether or not the solver will check for bad numerics at each iteration as `unsigned`.

- void SetCheckBadNumericsFlag (unsigned flag)
  Sets the flag indicating whether or not the solver will check for bad numerics at each iteration.

Access methods for jacobian evaluation method

- unsigned GetTrueJacobianEvalStep () const
  Returns the number of iterations until the Jacobian should be refreshed as `unsigned`.
- void SetTrueJacobianEvalStep (unsigned frequency)
  Sets the number of iterations until the jacobian should be refreshed to `frequency`.
- double GetJacobianRefreshRatio () const
  Returns the threshold value of the ratio of the residual norms over successive iterations that triggers a Jacobian refresh as `double`.
- void SetJacobianRefreshRatio (double refreshRatio)
  Sets the threshold value of the ratio of the residual norms over successive iterations that triggers a Jacobian refresh to `refreshRatio`.
- double GetEpsilon () const
  Returns the perturbation value used to estimate the partial derivatives with finite-differences as `double`.
- void SetEpsilon (double epsilon)
  Sets the perturbation value used to estimate the partial derivatives with finite-differences to `epsilon`.

Access methods for step control method

- unsigned GetStepControlMethod () const
  Returns the code for the step control method as `unsigned`.
- void SetStepControlMethod (unsigned method)
  Sets the code for the step control method to `method`.
- double GetMaxRelaxationCoefficient () const
  Returns the maximum relaxation coefficient used by the step control method as `double`.
- void SetMaxRelaxationCoefficient (double maxRelaxation)
  Sets the maximum relaxation coefficient used by the step control method to `maxRelaxation`.
- double GetMinRelaxationCoefficient () const
  Returns the minimum relaxation coefficient used by the step control method as `double`.
- void SetMinRelaxationCoefficient (double minRelaxation)
  Sets the minimum relaxation coefficient used by the step control method to `minRelaxation`.

Access methods for linear solver

- unsigned GetMatrixSolvingMethod () const
  Returns the code for the linear solution method as `unsigned`.
- void SetMatrixSolvingMethod (unsigned method)
  Sets the code for the linear solution method to `method`.
- unsigned GetScalingMethod () const
  Returns the code for the scaling method as `unsigned`.

- void SetScalingMethod (unsigned method)

    Sets the code for the scaling method to `method`.

- unsigned GetPivotingMethod () const

    Returns the code for the pivoting method used in conjunction with the Gaussian elimination method as `unsigned`.

- void SetPivotingMethod (unsigned method)

    Sets the code for the pivoting method used in conjunction with the Gaussian elimination method to `method`.

- unsigned GetRefinementMethod () const

    Returns the number of desired refinement iterations as `unsigned`.

- void SetRefinementMethod (unsigned method)

    Sets the number of desired refinement iterations to `method`.

Access methods for tracers

Note:

The tracer file names must be unique across all active components and problems being solved within the same process space.

- const char * GetVariablesTracerFilename () const

    Returns the name of the variables tracer file as `const char*` .

- void SetVariablesTracerFilename (const char *filename)

    Sets the name of the variables tracer file to `filename`.

- const char * GetIncrementsTracerFilename () const

    Returns the name of the increments tracer file as `const char*` .

- void SetIncrementsTracerFilename (const char *filename)

    Sets the name of the increments tracer file to `filename`.

- const char * GetResidualsTracerFilename () const

    Returns the name of the residuals tracer file as `const char*` .

- void SetResidualsTracerFilename (const char *filename)

    Sets the name of the residuals tracer file to `filename`.

- const char * GetJacobianTracerFilename () const

    Returns the name of the Jacobian tracer file as `const char*` .

- void SetJacobianTracerFilename (const char *filename)

    Sets the name of the Jacobian tracer file to `filename`.

I/O Operations

- void Write (std::ostream &os, const std::string &before) const

    Writes the list of component settings to `os`.

3.3.1 Detailed Description

Class acts as repository of settings defined for each component.

Behavior:

- compile-time default values are initialized with the hard-coded values.
- then at runtime, values are overloaded with the ones specified in the *.prf file in the section `ComponentSettings` for each component.

See namespace SPARK::DefaultComponentSettings for the list of hard-coded default settings.

Note:
By default, no tracer file names are specified.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 SPARK::TComponentSettings::TComponentSettings ()

Initialize all settings to compile-time default values.

3.3.2.2 SPARK::TComponentSettings::TComponentSettings (const TComponentSettings &)

Copy constructor.

3.3.2.3 SPARK::TComponentSettings::~TComponentSettings () throw () `[inline]`

Trivial destructor.

3.3.3 Member Function Documentation

3.3.3.1 TComponentSettings& SPARK::TComponentSettings::operator= (const TComponentSettings &)

Assignement operator.

3.3.3.2 void SPARK::TComponentSettings::Reset ()

Resets settings to default values.

3.3.3.3 void SPARK::TComponentSettings::Load (SPARK::TPrefList * *prefList*)

Loads all settings from prefList.

3.3.3.4 unsigned SPARK::TComponentSettings::GetComponentSolvingMethod () const `[inline]`

Returns the code for the component solving method as `unsigned`.

List of codes:

- Newton-Raphson method = 0

- Perturbed Newton-Raphson method = 1
- Fixed-point iteration (aka forward substitution) = 2
- Secant method (Newton method with Broyden's update formula) = 4

3.3.3.5 void SPARK::TComponentSettings::SetComponentSolvingMethod (unsigned *method*) [inline]

Sets the code for the component solving method to `method`.

3.3.3.6 unsigned SPARK::TComponentSettings::GetMaxIterations () const [inline]

Returns the maximum number of iterations allowed in the nonlinear solver as `unsigned`.

3.3.3.7 void SPARK::TComponentSettings::SetMaxIterations (unsigned *maxIterations*) [inline]

Sets the maximum number of iterations allowed in the nonlinear solver to `maxIterations`.

3.3.3.8 unsigned SPARK::TComponentSettings::GetMinIterations () const [inline]

Returns the minimum number of iterations allowed in the nonlinear solver as `unsigned`.

3.3.3.9 void SPARK::TComponentSettings::SetMinIterations (unsigned *minIterations*) [inline]

Sets the minimum number of iterations allowed in the nonlinear solver to `minIterations`.

3.3.3.10 unsigned SPARK::TComponentSettings::GetCheckBadNumericsFlag () const [inline]

Returns the boolean flag (0=false | 1=true) indicating whether or not the solver will check for bad numerics at each iteration as `unsigned`.

3.3.3.11 void SPARK::TComponentSettings::SetCheckBadNumericsFlag (unsigned *flag*) [inline]

Sets the flag indicating whether or not the solver will check for bad numerics at each iteration.

3.3.3.12 unsigned SPARK::TComponentSettings::GetTrueJacobianEvalStep () const [inline]

Returns the number of iterations until the Jacobian should be refreshed as `unsigned`.

Note:

If it is equal to 0 then it is automatically refreshed by the solver when the convergence process becomes to slow or diverging.

3.3.3.13 void SPARK::TComponentSettings::SetTrueJacobianEvalStep (unsigned *frequency*) [inline]

Sets the number of iterations until the jacobian should be refreshed to `frequency`.

3.3.3.14 double SPARK::TComponentSettings::GetJacobianRefreshRatio () const [inline]

Returns the threshold value of the ratio of the residual norms over successive iterations that triggers a Jacobian refresh as `double`.

3.3.3.15 void SPARK::TComponentSettings::SetJacobianRefreshRatio (double *refreshRatio*) [inline]

Sets the threshold value of the ratio of the residual norms over successive iterations that triggers a Jacobian refresh to `refreshRatio`.

3.3.3.16 double SPARK::TComponentSettings::GetEpsilon () const [inline]

Returns the perturbation value used to estimate the partial derivatives with finite-differences as `double`.

Note:
If it is equal to 0, then solver automatically computes a scaled perturbation value for each dependent variable

3.3.3.17 void SPARK::TComponentSettings::SetEpsilon (double *epsilon*) [inline]

Sets the perturbation value used to estimate the partial derivatives with finite-differences to `epsilon`.

3.3.3.18 unsigned SPARK::TComponentSettings::GetStepControlMethod () const [inline]

Returns the code for the step control method as `unsigned`.

List of codes:

- Fixed relaxation method = 0
- Basic backtracking method based on halving strategy = 1
- Line search backtracking method = 2
- Affine invariant backtracking strategy = 3

3.3.3.19 void SPARK::TComponentSettings::SetStepControlMethod (unsigned *method*) [inline]

Sets the code for the step control method to `method`.

3.3.3.20 double SPARK::TComponentSettings::GetMaxRelaxationCoefficient () const [inline]

Returns the maximum relaxation coefficient used by the step control method as `double`.

3.3.3.21 void SPARK::TComponentSettings::SetMaxRelaxationCoefficient (double *maxRelaxation*) [inline]

Sets the maximum relaxation coefficient used by the step control method to `maxRelaxation`.

3.3.3.22 double SPARK::TComponentSettings::GetMinRelaxationCoefficient () const [inline]

Returns the minimum relaxation coefficient used by the step control method as `double`.

3.3.3.23 void SPARK::TComponentSettings::SetMinRelaxationCoefficient (double *minRelaxation*) [inline]

Sets the minimum relaxation coefficient used by the step control method to `minRelaxation`.

3.3.3.24 unsigned SPARK::TComponentSettings::GetMatrixSolvingMethod () const [inline]

Returns the code for the linear solution method as `unsigned`.

List of codes:

- Dense Gaussian elimination = 0
- Dense singular value decomposition (SVD) = 1
- Dense LU decomposition = 2
- Dense Gauss-Jordan elimination = 3
- Sparse LU decomposition = 4 (implemented with the UMFPACK 4.0 library. See `http://www.cise.ufl.edu/research/sparse/umfpack/`)

3.3.3.25 void SPARK::TComponentSettings::SetMatrixSolvingMethod (unsigned *method*) [inline]

Sets the code for the linear solution method to `method`.

3.3.3.26 unsigned SPARK::TComponentSettings::GetScalingMethod () const [inline]

Returns the code for the scaling method as `unsigned`.

List of codes:

- No scaling = 0
- Affine invariant scaling (in both variable and residual spaces) = 1

3.3.3.27 void SPARK::TComponentSettings::SetScalingMethod (unsigned *method*) [inline]

Sets the code for the scaling method to `method`.

3.3.3.28 unsigned SPARK::TComponentSettings::GetPivotingMethod () const [inline]

Returns the code for the pivoting method used in conjunction with the Gaussian elimination method as `unsigned`.

List of codes:

- No pivoting method = 0
- Partial pivoting method = 1
- Total pivoting method = 2

Note:

The total pivoting method is implemented with the Gauss-Jordan method. See TComponentSettings::GetMatrixSolvingMethod().

3.3.3.29 void SPARK::TComponentSettings::SetPivotingMethod (unsigned *method*) [inline]

Sets the code for the pivoting method used in conjunction with the Gaussian elimination method to `method`.

3.3.3.30 unsigned SPARK::TComponentSettings::GetRefinementMethod () const [inline]

Returns the number of desired refinement iterations as `unsigned`.

Note:
Typically, 2/3 iterations is sufficient to improve the accuracy of the solution of the linear system.

3.3.3.31 void SPARK::TComponentSettings::SetRefinementMethod (unsigned *method*) [inline]

Sets the number of desired refinement iterations to `method`.

3.3.3.32 const char∗ SPARK::TComponentSettings::GetVariablesTracerFilename () const [inline]

Returns the name of the variables tracer file as `const char*` .

3.3.3.33 void SPARK::TComponentSettings::SetVariablesTracerFilename (const char ∗ *filename*) [inline]

Sets the name of the variables tracer file to `filename`.

3.3.3.34 const char∗ SPARK::TComponentSettings::GetIncrementsTracerFilename () const [inline]

Returns the name of the increments tracer file as `const char*` .

3.3.3.35 void SPARK::TComponentSettings::SetIncrementsTracerFilename (const char ∗ *filename*) [inline]

Sets the name of the increments tracer file to `filename`.

3.3.3.36 const char∗ SPARK::TComponentSettings::GetResidualsTracerFilename () const [inline]

Returns the name of the residuals tracer file as `const char*` .

3.3.3.37 void SPARK::TComponentSettings::SetResidualsTracerFilename (const char ∗ *filename*) [inline]

Sets the name of the residuals tracer file to `filename`.

3.3.3.38 const char∗ SPARK::TComponentSettings::GetJacobianTracerFilename () const [inline]

Returns the name of the Jacobian tracer file as `const char*` .

3.3.3.39 void SPARK::TComponentSettings::SetJacobianTracerFilename (const char ∗ *filename*) [inline]

Sets the name of the Jacobian tracer file to `filename`.

3.3.3.40 void SPARK::TComponentSettings::Write (std::ostream & *os*, const std::string & *before*) const

Writes the list of component settings to `os`.

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

- prefs.h

3.4 SPARK::TGlobalSettings Class Reference

Class acts as repository of global control settings defined at the problem level.

`#include <prefs.h>`

Public Member Functions

Structors

- TGlobalSettings ()
  Initialize all settings to compile-time default values.
- TGlobalSettings (const TGlobalSettings &)
  Copy constructor.
- ∼TGlobalSettings () throw ()
  Trivial destructor.

Main operations

- void Reset ()
  Resets settings to default values See namespace SPARK::DefaultSettings.
- void Load (SPARK::TPrefList ∗prefList)
  Performs shallow copy of prefList into member datum PrefList and loads hard-coded settings.

Access methods for hard-coded global settings

- double GetTolerance () const
  Returns the value of the relative tolerance specified with the key Tolerance in the GlobalSettings section of the ∗.prf file.
- void SetTolerance (double scalar)
  Sets relative tolerance to `scalar`.
- double GetMaxTolerance () const
  Returns the value of the max relative tolerance specified with the key MaxTolerance in the GlobalSettings section of the ∗.prf file.
- void SetMaxTolerance (double scalar)
  Sets the max relative tolerance to `scalar`.
- double GetPredictionSafetyFactor () const
  Returns the prediction safety factor (used in convergence test) as `double`.
- void SetPredictionSafetyFactor (double factor)
  Sets the prediction safety factor to `factor`.
- double GetIterationSafetyFactor () const
  Returns the iteration safety factor (used in convergence test) as `double`.
- void SetIterationSafetyFactor (double factor)
  Sets the iteration safety factor to `factor`.
- double GetBreakUnknownSafetyFactor () const

Returns the safety factor for a break unknown (used in convergence test) as `double`.

- void SetBreakUnknownSafetyFactor (double factor)
  Sets the safety factor for a break unknown to `factor`.
- double GetNormalUnknownSafetyFactor () const
  Returns the safety factor for a normal unknown (used in convergence test) as `double`.
- void SetNormalUnknownSafetyFactor (double factor)
  Sets the safety factor for a normal unknown to `factor`.

Access methods for customized global settings

Note:
The key can be specified either as a simple key with "Key" or a complex key with "Key1:Key2"

- const char * GetString (const std::string &key, const std::string &defaultString) const
  Returns as const char the entry for the key in the GlobalSettings section of the *.prf file.*
- double GetScalar (const std::string &key, double defaultScalar) const
  *Returns as double the entry for the key in the GlobalSettings section of the *.prf file.*
- unsigned GetCount (const std::string &key, unsigned defaultCount) const
  *Returns as unsigned the entry for the key in the GlobalSettings section of the *.prf file.*

I/O Operations

- void Write (std::ostream &os, const std::string &before) const

3.4.1 Detailed Description

Class acts as repository of global control settings defined at the problem level.

Behavior:

- compile-time default values are initialized with the hard-coded values
- then at runtime, values are overloaded with the values specified in the *.prf file in the section `GlobalSettings`

See namespace SPARK::DefaultGlobalSettings for a list of the default hard-coded values.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 SPARK::TGlobalSettings::TGlobalSettings ()

Initialize all settings to compile-time default values.

3.4.2.2 SPARK::TGlobalSettings::TGlobalSettings (const TGlobalSettings &)

Copy constructor.

3.4.2.3 SPARK::TGlobalSettings::∼TGlobalSettings () throw ()

Trivial destructor.

3.4.3 Member Function Documentation

3.4.3.1 void SPARK::TGlobalSettings::Reset ()

Resets settings to default values See namespace SPARK::DefaultSettings.

3.4.3.2 void SPARK::TGlobalSettings::Load (SPARK::TPrefList ∗ *prefList*)

Performs shallow copy of prefList into member datum PrefList and loads hard-coded settings.

3.4.3.3 double SPARK::TGlobalSettings::GetTolerance () const `[inline]`

Returns the value of the relative tolerance specified with the key Tolerance in the GlobalSettings section of the ∗.prf file.

3.4.3.4 void SPARK::TGlobalSettings::SetTolerance (double *scalar*) `[inline]`

Sets relative tolerance to `scalar`.

3.4.3.5 double SPARK::TGlobalSettings::GetMaxTolerance () const `[inline]`

Returns the value of the max relative tolerance specified with the key MaxTolerance in the GlobalSettings section of the ∗.prf file.

3.4.3.6 void SPARK::TGlobalSettings::SetMaxTolerance (double *scalar*) `[inline]`

Sets the max relative tolerance to `scalar`.

3.4.3.7 double SPARK::TGlobalSettings::GetPredictionSafetyFactor () const `[inline]`

Returns the prediction safety factor (used in convergence test) as `double`.

3.4.3.8 void SPARK::TGlobalSettings::SetPredictionSafetyFactor (double *factor*) `[inline]`

Sets the prediction safety factor to `factor`.

3.4.3.9 double SPARK::TGlobalSettings::GetIterationSafetyFactor () const `[inline]`

Returns the iteration safety factor (used in convergence test) as `double`.

3.4.3.10 void SPARK::TGlobalSettings::SetIterationSafetyFactor (double *factor*) `[inline]`

Sets the iteration safety factor to `factor`.

3.4.3.11 double SPARK::TGlobalSettings::GetBreakUnknownSafetyFactor () const [inline]

Returns the safety factor for a break unknown (used in convergence test) as `double`.

3.4.3.12 void SPARK::TGlobalSettings::SetBreakUnknownSafetyFactor (double *factor*) [inline]

Sets the safety factor for a break unknown to `factor`.

3.4.3.13 double SPARK::TGlobalSettings::GetNormalUnknownSafetyFactor () const [inline]

Returns the safety factor for a normal unknown (used in convergence test) as `double`.

3.4.3.14 void SPARK::TGlobalSettings::SetNormalUnknownSafetyFactor (double *factor*) [inline]

Sets the safety factor for a normal unknown to `factor`.

3.4.3.15 const char* SPARK::TGlobalSettings::GetString (const std::string & *key*, const std::string & *defaultString*) const

Returns as const char* the entry for the key in the GlobalSettings section of the *.prf file.

3.4.3.16 double SPARK::TGlobalSettings::GetScalar (const std::string & *key*, double *defaultScalar*) const

Returns as double the entry for the key in the GlobalSettings section of the *.prf file.

3.4.3.17 unsigned SPARK::TGlobalSettings::GetCount (const std::string & *key*, unsigned *defaultCount*) const

Returns as unsigned the entry for the key in the GlobalSettings section of the *.prf file.

3.4.3.18 void SPARK::TGlobalSettings::Write (std::ostream & *os*, const std::string & *before*) const

Writes the list of global settings to os

Note:
Only hard-coded global settings are printed since the other settings are not known a priori.

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

- prefs.h

3.5 SPARK::TInverse Class Reference

Class that defines the callbacks for an inverse.

`#include <inverse.h>`

Public Member Functions

Access methods

- unsigned GetHandle () const
  Returns unique inverse handle as specified in XML file.
- const char * GetName () const
  Returns name of inverse as specified in XML file as `const char*` .
- const char * GetAtomicClassName () const
  Returns class name as specified in XML file as `const char*` .
- const char * GetFileName () const
  Returns file name of atomic class where the inverse is declared as `const char*` .
- SPARK::AtomicTypes GetAtomicType () const
  Returns the type of the inverse as SPARK::AtomicTypes.
- const char * GetActiveCallbackName () const
  Returns the name of the currently active callback, Returns "" if none active!
- unsigned GetNumObjects () const
  Returns the number of TObject instances of this inverse.
- SPARK::TObject * GetObject (unsigned idx)
  Returns instance idx as a SPARK::TObject pointer.
- const SPARK::TProblem * GetProblem () const
  Returns pointer to SPARK::TProblem object this inverse belongs to.
- double GetTolerance () const
  Returns value of relative tolerance currently used by solver.

Data methods

- void * GetStaticData ()
  Returns static private data as `void*` .
- void SetStaticData (void *address)
  Sets the static private data to `address`.

3.5.1 Detailed Description

Class that defines the callbacks for an inverse.

Also contains all instances as SPARK::TObject objects of an inverse, as well as the void* pointer to the static data.

3.5.2 Member Function Documentation

3.5.2.1 unsigned SPARK::TInverse::GetHandle () const

Returns unique inverse handle as specified in XML file.

3.5.2.2 const char* SPARK::TInverse::GetName () const

Returns name of inverse as specified in XML file as `const char*` .

3.5.2.3 const char* SPARK::TInverse::GetAtomicClassName () const

Returns class name as specified in XML file as `const char*` .

3.5.2.4 const char* SPARK::TInverse::GetFileName () const

Returns file name of atomic class where the inverse is declared as `const char*` .

3.5.2.5 SPARK::AtomicTypes SPARK::TInverse::GetAtomicType () const

Returns the type of the inverse as SPARK::AtomicTypes.

3.5.2.6 const char* SPARK::TInverse::GetActiveCallbackName () const

Returns the name of the currently active callback, Returns "" if none active!

3.5.2.7 unsigned SPARK::TInverse::GetNumObjects () const

Returns the number of TObject instances of this inverse.

3.5.2.8 SPARK::TObject* SPARK::TInverse::GetObject (unsigned *idx*)

Returns instance idx as a SPARK::TObject pointer.

3.5.2.9 const SPARK::TProblem* SPARK::TInverse::GetProblem () const

Returns pointer to SPARK::TProblem object this inverse belongs to.

3.5.2.10 double SPARK::TInverse::GetTolerance () const

Returns value of relative tolerance currently used by solver.

3.5.2.11 void* SPARK::TInverse::GetStaticData () `[inline]`

Returns static private data as `void*` .

3.5.2.12 void SPARK::TInverse::SetStaticData (void * *address*) [inline]

Sets the static private data to address.

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

- inverse.h

3.6 SPARK::TObject Class Reference

Class used to represent an instance of an inverse.

```
#include <object.h>
```

Public Member Functions

Access methods

- unsigned GetHandle () const
  Returns unique handle as specified in XML file.
- unsigned GetInstanceHandle () const
  Returns unique inverse handle set by SPARK::TInverse object for each instance.
- const char * GetName () const
  Returns name of object as specified in XML file as `const char*` .
- const char * GetActiveCallbackName () const
  Returns the name of the currently active callback, Returns "" if none active!
- SPARK::TInverse * GetInverse ()
  Returns name of SPARK::TInverse this object is an instance of.
- const SPARK::TProblem * GetProblem () const
  Returns pointer to SPARK::TProblem object this object belongs to.
- const SPARK::TComponent * GetComponent () const
  Returns pointer to SPARK::TComponent object this object belongs to.
- double GetTolerance () const
  Returns value of relative tolerance curently used by solver.

Data methods

- void * GetData ()
  Returns private data associated with this instance as `void*` .
- void SetData (void *address)
  Sets the private data to `address`.

3.6.1 Detailed Description

Class used to represent an instance of an inverse.

Note:
Static callbacks do not require argument lists

3.6.2 Member Function Documentation

3.6.2.1 unsigned SPARK::TObject::GetHandle () const `[inline]`

Returns unique handle as specified in XML file.

3.6.2.2 unsigned SPARK::TObject::GetInstanceHandle () const [inline]

Returns unique inverse handle set by SPARK::TInverse object for each instance.

3.6.2.3 const char* SPARK::TObject::GetName () const

Returns name of object as specified in XML file as `const char*` .

3.6.2.4 const char* SPARK::TObject::GetActiveCallbackName () const

Returns the name of the currently active callback, Returns "" if none active!

3.6.2.5 SPARK::TInverse* SPARK::TObject::GetInverse ()

Returns name of SPARK::TInverse this object is an instance of.

3.6.2.6 const SPARK::TProblem* SPARK::TObject::GetProblem () const

Returns pointer to SPARK::TProblem object this object belongs to.

3.6.2.7 const SPARK::TComponent* SPARK::TObject::GetComponent () const

Returns pointer to SPARK::TComponent object this object belongs to.

3.6.2.8 double SPARK::TObject::GetTolerance () const

Returns value of relative tolerance curently used by solver.

3.6.2.9 void* SPARK::TObject::GetData () [inline]

Returns private data associated with this instance as `void*` .

3.6.2.10 void SPARK::TObject::SetData (void * *address*) [inline]

Sets the private data to `address`.

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

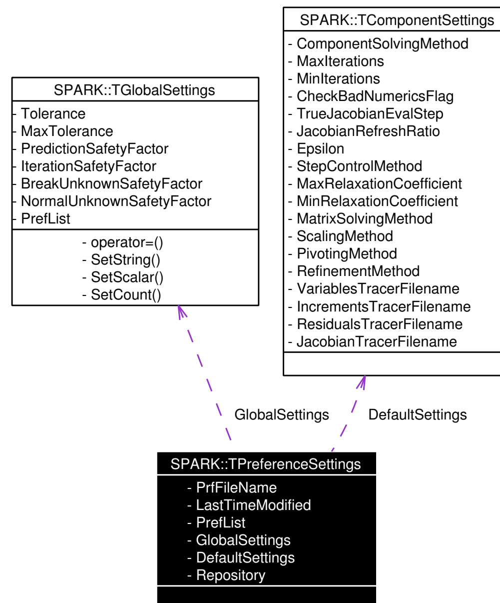
- object.h

3.7 SPARK::TPreferenceSettings Class Reference

Wrapper class to store and manipulate information required to initialize the settings for the solution methods for each component.

`#include <prefs.h>`

Collaboration diagram for SPARK::TPreferenceSettings:



Public Member Functions

Structors

- TPreferenceSettings (const char ∗prfFileName)
  Loads preference settings from ∗.prf file named "prfFileName".
- ∼TPreferenceSettings () throw ()
  Trivial destructor.

Access methods

- const char ∗ GetPrfFileName () const
  Returns the name of the ∗.prf file used to construct this object as `const char*` .
- TGlobalSettings & GetGlobalSettings ()
  Returns a reference to the TGlobalSettings object containing the global setting defined in the ∗.prf file.
- const TGlobalSettings & GetGlobalSettings () const
  Returns a const reference to the TGlobalSettings object containing the global setting defined in the ∗.prf file.
- TComponentSettings & GetComponentSettings (unsigned compHandle)
  Returns a reference to the TCopmonentSettings object containing the component setting for the component with handle `comphandle` *defined in the ∗.prf file.*
- const TComponentSettings & GetComponentSettings (unsigned compHandle) const

Returns a const reference to the TCopmonentSettings object containing the component setting for the component with handle `comphandle` *defined in the ∗.prf file.*

I/O Operations

- void Write (std::ostream &os, const std::string &before) const
  Writes the global settings and all component settings to `os`.

3.7.1 Detailed Description

Wrapper class to store and manipulate information required to initialize the settings for the solution methods for each component.

Examples:
multiproblem_example1.cpp, and sparksolver.cpp.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 SPARK::TPreferenceSettings::TPreferenceSettings (const char ∗ *prfFileName*)

Loads preference settings from ∗.prf file named "prfFileName".

3.7.2.2 SPARK::TPreferenceSettings::∼TPreferenceSettings () throw ()

Trivial destructor.

3.7.3 Member Function Documentation

3.7.3.1 const char∗ SPARK::TPreferenceSettings::GetPrfFileName () const [inline]

Returns the name of the ∗.prf file used to construct this object as `const char*` .

3.7.3.2 TGlobalSettings& SPARK::TPreferenceSettings::GetGlobalSettings ()

Returns a reference to the TGlobalSettings object containing the global setting defined in the ∗.prf file.

3.7.3.3 const TGlobalSettings& SPARK::TPreferenceSettings::GetGlobalSettings () const

Returns a const reference to the TGlobalSettings object containing the global setting defined in the ∗.prf file.

3.7.3.4 TComponentSettings& SPARK::TPreferenceSettings::GetComponentSettings (unsigned *compHandle*)

Returns a reference to the TCopmonentSettings object containing the component setting for the component with handle `comphandle` defined in the ∗.prf file.

3.7.3.5 const TComponentSettings& SPARK::TPreferenceSettings::GetComponentSettings (unsigned *compHandle*) const

Returns a const reference to the TCopmonentSettings object containing the component setting for the component with handle `comphandle` defined in the ∗.prf file.

3.7.3.6 void SPARK::TPreferenceSettings::Write (std::ostream & *os*, const std::string & *before*) const

Writes the global settings and all component settings to `os`.

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

- prefs.h

3.8 SPARK::TProblem Class Reference

Representation of a problem object in the SPARK solver.

```
#include <problem.h>
```

Public Types

- typedef std::bitset< DIAGNOSTICTYPES_L > TDiagnosticLevel

  Bitset used to specify the diagnostic level with the different diagnostic types.

- enum DiagnosticTypes {
  DiagnosticType_CONVERGENCE,
  DiagnosticType_INPUTS,
  DiagnosticType_REPORTS,
  DiagnosticType_PREFERENCES,
  DiagnosticType_STATISTICS,
  DiagnosticType_REQUESTS,
  DIAGNOSTICTYPES_L }

  Level of more or less detailed diagnostic to the run log output stream.

- enum SimulationFlags {
  SimulationFlag_OK = 0,
  SimulationFlag_BAD_NUMERICS,
  SimulationFlag_NO_CONVERGENCE,
  SimulationFlag_SINGULAR_SYSTEM,
  SimulationFlag_TIMESTEP_TOO_SMALL,
  SimulationFlag_FAILED_STEP,
  SimulationFlag_IDLE,
  SIMULATIONFLAGS_L }

  Simulation flags.

Public Member Functions

Structors

- TProblem (unsigned numInverses, SPARK::TInverse ∗inverses[ ], unsigned numVariables, SPARK::TVariable ∗variables[ ], unsigned numComponents, SPARK::TComponent ∗components[ ]) throw (SPARK::XMemory)

  Constructs a TProblem object from the structure specified by the variable arrays and the solution sequence described by the component array.

- ∼TProblem () throw ()

  Destructor that frees memory allocated for the various solvers.

Main methods invoked by driver function

- void Initialize (const SPARK::TRuntimeControls &controls) throw (SPARK::XMemory)

Performs run-time initialization.

- void LoadPreferenceSettings (const SPARK::TPreferenceSettings &preferences) throw (SPARK::XMemory)
  Loads preference settings (default and for each component).
- SimulationFlags Simulate () throw (SPARK::XInitialization)
  Computes solution from InitialTime to FinalTime.
- void Terminate ()
  Ends processing and writes statistics to run log file.

State management functions

These method let you save and restore the problem state in order to allow for a simulation restart from the saved state.

- void Save (SPARK::TProblem::TState &state) const throw (SPARK::XInitialization)
  Saves problem state at current time.
- void Restore (const SPARK::TProblem::TState &state)
  Restores problem state at the time specified in the state structure.

Access functions

- const char * GetName () const
  Returns the name of the problem as const char.*
- void SetName (const char *name)
  Sets the problem name from name.
- unsigned long GetStepCount () const
  Returns the number of simulation steps performed so far.
- SPARK::TGlobalSettings * GetGlobalSettings ()
  Returns pointer to global settings object.
- const SPARK::TGlobalSettings * GetGlobalSettings () const
  Returns pointer to const global settings object.

Access operations for global problem variables

- const SPARK::TVariable & GetGlobalTime () const
  Returns a const reference to the TVariable object that describes the global time link.
- const SPARK::TVariable & GetGlobalTimeStep () const
  Returns a const reference to the TVariable object that describes the global time step link.

Access operations for the problem variables

- SPARK::TVariable & GetVariable (unsigned handle) throw (SPARK::XAssertion)
- const SPARK::TVariable & GetVariable (unsigned handle) const throw (SPARK::XAssertion)
- SPARK::TVariable & GetVariable (const char *name) throw (SPARK::XAssertion)
- const SPARK::TVariable & GetVariable (const char *name) const throw (SPARK::XAssertion)

Access operations for the problem inverses

- SPARK::TInverse ∗ GetInverse (unsigned handle)
    Returns pointer to TInverse object by handle.
- SPARK::TInverse ∗ GetInverse (const char ∗name)
    Returns pointer to TInverse object by name.

Access operations for the problem objects

- SPARK::TObject ∗ GetObject (unsigned handle)
    Returns pointer to TObject object by handle.
- SPARK::TObject ∗ GetObject (const char ∗name)
    Returns pointer to TObject object by name.

Predicate methods

- bool IsInitialTime () const
    Returns true if global time is equal to initial time.
- bool IsFinalTime () const
    Returns true if global time is equal to final time.
- bool Starting () const
- bool IsStaticStep () const
    Returns true if current step is a static step.
- bool IsTimeStepVariable () const
    Returns true if key VariableTimeStep is set to 1 in runtime controls.
- bool IsReady () const
    Returns true if problem is ready for simulation. False otherwise.
- bool IsDiagnostic () const
    Returns true if any diagnostic is set.
- bool IsDiagnostic (DiagnosticTypes d) const
    Returns true if diagnostic d is set.

IO functions

- bool WriteStamp (std::ostream &os) const
    Writes current step stamp to output stream os.
- void ReportStatistics (std::ostream &os, const std::string &before) const
    Writes simulation statistics to output stream os.
- void GenerateSnapshot (const std::string &filename) const throw (SPARK::XInitialization)
    Generates the snapshot file named "filename".

3.8.1 Detailed Description

Representation of a problem object in the SPARK solver.

Class methods implement :

- read values from input files
- generate snapshot files and write reported values to output or trace files
- solve system of equations for the unknown problem variables at $Clock = t_{initial}$ where $t_{initial} = InitialTime$ as specified in the runtime controls
- solve system of equations for the unknown problem variables for $t_{initial} + dt <= t <= t_{final}$ where $dt = TimeStep$ and $t_{final} = FinalTime$ as specified in the runtime controls file
- save current values of problem variables as history

Note:

The TProblem class relies for all runtime information on a SPARK::TRuntimeControls object that is passed to the TProblem::Initialize() method.

- Invoke the TProblem::Initialize() method to bind the SPARK::TRuntimeControls object with a TProblem object.
- Before re-invoking the TProblem::Initialize() method, make sure that you have invoked the method TProblem::Terminate() first to reset the various managers.

Examples:

multiproblem_example1.cpp, and sparksolver.cpp.

3.8.2 Member Typedef Documentation

3.8.2.1 typedef std::bitset<DIAGNOSTICTYPES_L> SPARK::TProblem::TDiagnosticLevel

Bitset used to specify the diagnostic level with the different diagnostic types.

To make a run with convergence and preferences diagnostic, specifiy `DiagnosticLevel( 9 () )` in the runtime controls file whereby 9 = 1 (*DiagnosticType_CONVERGENCE*) + 8 (*DiagnosticType_PREFERENCES*)

3.8.3 Member Enumeration Documentation

3.8.3.1 enum SPARK::TProblem::DiagnosticTypes

Level of more or less detailed diagnostic to the run log output stream.

Enumeration values:

DiagnosticType_CONVERGENCE (=1) show convergence behavior at each iteration

DiagnosticType_INPUTS (=2) show where the variables get their input from

DiagnosticType_REPORTS (=4) show reported values at each report event

DiagnosticType_PREFERENCES (=8) show preference settings as loaded in LoadPreferenceSettings()

DiagnosticType_STATISTICS (=16) show run statistics at the end of simulation

DiagnosticType_REQUESTS (=32) show atomic class requests

DIAGNOSTICTYPES_L Number of diagnostic types.

3.8.3.2 enum SPARK::TProblem::SimulationFlags

Simulation flags.

Enumeration values:

SimulationFlag_OK Step was computed successfully.

SimulationFlag_BAD_NUMERICS Bad numerics detected.

SimulationFlag_NO_CONVERGENCE Could not obtain convergence while solving nonlinear system(s).

SimulationFlag_SINGULAR_SYSTEM Detected a singular or "badly-conditioned" linear system.

SimulationFlag_TIMESTEP_TOO_SMALL Time step selection is limited by MinTimeStep from runtime controls file.

SimulationFlag_FAILED_STEP Step was rejected too many times in response to user's requests.

SimulationFlag_IDLE Default step when starting the simulation.

SIMULATIONFLAGS_L Number of simulation flags.

3.8.4 Constructor & Destructor Documentation

3.8.4.1 SPARK::TProblem::TProblem (unsigned *numInverses*, SPARK::TInverse ∗ *inverses*[ ], unsigned *numVariables*, SPARK::TVariable ∗ *variables*[ ], unsigned *numComponents*, SPARK::TComponent ∗ *components*[ ]) throw (SPARK::XMemory)

Constructs a TProblem object from the structure specified by the variable arrays and the solution sequence described by the component array.

Parameters:

numInverses Number of inverses

inverses Array of pointers to TInverse objects

numVariables Number of problem variables

variables Array of pointers to TVariable objects

numComponents Number of components comprising the problem

components Array of pointers to TComponent objects

Exceptions:

Throws SPARK::XMemory if out of memory when building the problem.

3.8.4.2 SPARK::TProblem::∼TProblem () throw ()

Destructor that frees memory allocated for the various solvers.

Note:

Destructor does not destroy any data that was passed to the constructor to describe the problem topology. The calling program is responsible for destroying these data structures explicitly if needed.

3.8.5 Member Function Documentation

3.8.5.1 void SPARK::TProblem::Initialize (const SPARK::TRuntimeControls & *controls*) throw (SPARK::XMemory)

Performs run-time initialization.

Parameters:

controls runtime control parameters for the simulation

Postcondition:

Initializes clock manager
Allocates history and inner values
Prepares input and output managers
Loads initial values
Fires construct callbacks

Examples:

sparksolver.cpp.

3.8.5.2 void SPARK::TProblem::LoadPreferenceSettings (const SPARK::TPreferenceSettings & *preferences*) throw (SPARK::XMemory)

Loads preference settings (default and for each component).

Parameters:

preferences An object of class SPARK::TPreferenceSettings that contains data from the ∗.prf file

Postcondition:

The SPARK::TGlobalSettings object is loaded for this problem.
The SPARK::TComponentSettings object are loaded for each component.

Examples:

sparksolver.cpp.

3.8.5.3 SimulationFlags SPARK::TProblem::Simulate () throw (SPARK::XInitialization)

Computes solution from InitialTime to FinalTime.

Returns:

Simulation flag as enum SimulationFlags

Precondition:

Problem must have initialized with a prior call to TProblem::Initialize()

Postcondition:

Output and trace files are updated after each simulation step until end of simulation.
Initial snapshot file is generated after the first step if requested in the runtime controls file.
Final snapshot file is generated at the last step if requested in the runtime controls file.

Exceptions:

Throws a SPARK::XInitialization exception object if problem was not initialized or if it is in an inconsistent state.

Examples:

sparksolver.cpp.

3.8.5.4 void SPARK::TProblem::Terminate ()

Ends processing and writes statistics to run log file.

Postcondition:

Fires destruct callbacks
Deletes input and output managers
Resets internal counters to allow for a fresh simulation run with a new set of runtime controls specified with a call to TProblem::Initialize().

Examples:

sparksolver.cpp.

3.8.5.5 void SPARK::TProblem::Save (SPARK::TProblem::TState & *state*) const throw (SPARK::XInitialization)

Saves problem state at current time.

This method populates the state object with the state of the problem at the current time.

Parameters:

state Object containing the problem state being saved.

Exceptions:

Throws a SPARK::XInitialization exception object if problem could not be saved

3.8.5.6 void SPARK::TProblem::Restore (const SPARK::TProblem::TState & *state*)

Restores problem state at the time specified in the state structure.

The method initializes the variables with the trajectory values stored in the state object. For now there is no support for class data persistency.

Parameters:

state Object containing the saved problem state to be restored.

3.8.5.7 const char∗ SPARK::TProblem::GetName () const

Returns the name of the problem as const char∗.

3.8.5.8 void SPARK::TProblem::SetName (const char ∗ *name*)

Sets the problem name from name.

3.8.5.9 unsigned long SPARK::TProblem::GetStepCount () const

Returns the number of simulation steps performed so far.

3.8.5.10 SPARK::TGlobalSettings∗ SPARK::TProblem::GetGlobalSettings ()

Returns pointer to global settings object.

3.8.5.11 const SPARK::TGlobalSettings∗ SPARK::TProblem::GetGlobalSettings () const

Returns pointer to const global settings object.

3.8.5.12 const SPARK::TVariable& SPARK::TProblem::GetGlobalTime () const

Returns a const reference to the TVariable object that describes the global time link.

3.8.5.13 const SPARK::TVariable& SPARK::TProblem::GetGlobalTimeStep () const

Returns a const reference to the TVariable object that describes the global time step link.

3.8.5.14 SPARK::TVariable& SPARK::TProblem::GetVariable (unsigned *handle*) throw (SPARK::XAssertion)

Returns TVariable& object by handle

Parameters:

handle is the unsigned value that uniquely identifies this variable as specified in the XML file. See TVariable::Handle

Exceptions:

SPARK::XAssertion Could not find a variable for this handle

3.8.5.15 const SPARK::TVariable& SPARK::TProblem::GetVariable (unsigned *handle*) const throw (SPARK::XAssertion)

Returns const TVariable& object by handle

Parameters:

handle is the unsigned value that uniquely identifies this variable as specified in the XML file. See TVariable::Handle

Exceptions:

SPARK::XAssertion Could not find a variable for this handle

3.8.5.16 SPARK::TVariable& SPARK::TProblem::GetVariable (const char ∗ *name*) throw (SPARK::XAssertion)

Returns TVariable& object by name

Parameters:

name const char∗ that stands for the name of the variable as specified in the XML file. See TVariable::Name

Note:

The variable name is case-sensitive.

Exceptions:

SPARK::XAssertion Could not find a variable with this name.

3.8.5.17 const SPARK::TVariable& SPARK::TProblem::GetVariable (const char * *name*) const throw (SPARK::XAssertion)

Returns const TVariable& object by name

Parameters:

name const char* that stands for the name of the variable as specified in the XML file. See TVariable::Name

Note:

The variable name is case-sensitive.

Exceptions:

SPARK::XAssertion Could not find a variable with this name.

3.8.5.18 SPARK::TInverse* SPARK::TProblem::GetInverse (unsigned *handle*)

Returns pointer to TInverse object by handle.

3.8.5.19 SPARK::TInverse* SPARK::TProblem::GetInverse (const char * *name*)

Returns pointer to TInverse object by name.

3.8.5.20 SPARK::TObject* SPARK::TProblem::GetObject (unsigned *handle*)

Returns pointer to TObject object by handle.

3.8.5.21 SPARK::TObject* SPARK::TProblem::GetObject (const char * *name*)

Returns pointer to TObject object by name.

3.8.5.22 bool SPARK::TProblem::IsInitialTime () const

Returns true if global time is equal to initial time.

3.8.5.23 bool SPARK::TProblem::IsFinalTime () const

Returns true if global time is equal to final time.

3.8.5.24 bool SPARK::TProblem::Starting () const

Returns true if first step of simulation.

Note:

True for first step computed when calling TProblem::Simulate() This is different than checking for GetStepCount()==1 since the StepCount counter is incremented at every step, possibly for multiple calls to TProblem::Simulate(). StepCount is reset to 0 only when calling TProblem::Initialize().

3.8.5.25 bool SPARK::TProblem::IsStaticStep () const

Returns true if current step is a static step.

3.8.5.26 bool SPARK::TProblem::IsTimeStepVariable () const

Returns true if key VariableTimeStep is set to 1 in runtime controls.

3.8.5.27 bool SPARK::TProblem::IsReady () const

Returns true if problem is ready for simulation. False otherwise.

3.8.5.28 bool SPARK::TProblem::IsDiagnostic () const [inline]

Returns true if any diagnostic is set.

3.8.5.29 bool SPARK::TProblem::IsDiagnostic (DiagnosticTypes *d*) const [inline]

Returns true if diagnostic d is set.

3.8.5.30 bool SPARK::TProblem::WriteStamp (std::ostream & *os*) const

Writes current step stamp to output stream os.

3.8.5.31 void SPARK::TProblem::ReportStatistics (std::ostream & *os*, const std::string & *before*) const

Writes simulation statistics to output stream os.

3.8.5.32 void SPARK::TProblem::GenerateSnapshot (const std::string & *filename*) const throw (SPARK::XInitialization)

Generates the snapshot file named "filename".

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

- problem.h

3.9 SPARK::TProblem::TState Class Reference

Interface class defining the methods used to save and restore the state of the problem using the TProblem::Save() and TProblem::Restore() methods.

`#include <problem.h>`

Public Member Functions

Structors

- TState ()
  Default consrtuctor.
- TState (const TState &)
  Copy constructor: deep copy of values and class states.
- ∼TState () throw ()
  Destructor.

Access functions for trajectory

- const char * GetContext () const
  Returns the character string that dscribes the context at the time the problem state was saved.
- double GetTime (unsigned idx) const
  Returns the value of the global time variable for the past value `idx`*.*
- double GetTimeStep (unsigned idx) const
  Returns the value of the global time step variable for the past value `idx`*.*
- unsigned GetNumVariables () const
  Returns the number of variables stored in Trajectory.
- unsigned GetNumPastValues () const
  Returns the number of past values stored in Trajectory for each variable.
- value_container & GetPastValues (unsigned idx)
  Returns container with desired past values corresponding to index `idx`*.*
- const value_container & GetPastValues (unsigned idx) const
  Returns container with desired past values corresponding to index `idx`*.*

Main operations

- void Save (const std::string &context, const SPARK::TTrajectory &trajectory, const TTopology &topology)
  Loads the problem state at the current time (values and class data).
- void Restore (TTrajectory &trajectory, TTopology &topology) const
  Restores the problem state.

3.9.1 Detailed Description

Interface class defining the methods used to save and restore the state of the problem using the TProblem::Save() and TProblem::Restore() methods.

The class defines methods used to save the values of all variables at the current global time. Then an instance of this class can be used to restore the problem state by invoking TProblem::Restore() in order to restart a simulation.

Warning:

These methods do not try to store or load the private data associated with each inverse and/or object in the problem under study. This can potentially cause problems when restarting the simulation with saved solution values because the private data might be in a different state. A later version of SPARK will provide support for full serialization of the problem state, i.e., including class private data, through the addition of dedicated callbacks.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 SPARK::TProblem::TState::TState ()

Default consrtuctor.

3.9.2.2 SPARK::TProblem::TState::TState (const TState &)

Copy constructor: deep copy of values and class states.

3.9.2.3 SPARK::TProblem::TState::~TState () throw ()

Destructor.

3.9.3 Member Function Documentation

3.9.3.1 const char* SPARK::TProblem::TState::GetContext () const

Returns the character string that dscribes the context at the time the problem state was saved.

3.9.3.2 double SPARK::TProblem::TState::GetTime (unsigned *idx*) const

Returns the value of the global time variable for the past value `idx`.

Exceptions:

Throws SPARK::Xinitialization if operation failed

3.9.3.3 double SPARK::TProblem::TState::GetTimeStep (unsigned *idx*) const

Returns the value of the global time step variable for the past value `idx`.

Exceptions:

Throws SPARK::Xinitialization if operation failed

3.9.3.4 unsigned SPARK::TProblem::TState::GetNumVariables () const

Returns the number of variables stored in Trajectory.

Returns:

Number of values defining the trajectory, including the values of the global time and the global time step.

3.9.3.5 unsigned SPARK::TProblem::TState::GetNumPastValues () const

Returns the number of past values stored in Trajectory for each variable.

Returns:

Number of past values

3.9.3.6 value_container& SPARK::TProblem::TState::GetPastValues (unsigned *idx*)

Returns container with desired past values corresponding to index `idx`.

Returns:

Reference to container of values

3.9.3.7 const value_container& SPARK::TProblem::TState::GetPastValues (unsigned *idx*) const

Returns container with desired past values corresponding to index `idx`.

Returns:

Const reference to container of values

3.9.3.8 void SPARK::TProblem::TState::Save (const std::string & *context*, const SPARK::TTrajectory & *trajectory*, const TTopology & *topology*)

Loads the problem state at the current time (values and class data).

If necessary, frees previously allocated memory.

Parameters:

context Description of the problem context

trajectory Trajectory tracker for the current problem

topology Problem topology to save

3.9.3.9 void SPARK::TProblem::TState::Restore (TTrajectory & *trajectory*, TTopology & *topology*) const

Restores the problem state.

Parameters:

trajectory Trajectory tracker for the current problem

topology Problem topology to restore

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

- problem.h

3.10 SPARK::TRuntimeControls Class Reference

Wrapper class for all the runtime control information required to initialize a TProblem object in order to make a simulation run.

`#include <ctrls.h>`

Public Types

- typedef std::vector< std::string > TInputFiles

  Type used to describe the list of input files specified in the ∗.run file with the key `InputFiles`.

Public Member Functions

Structors

- TRuntimeControls (const char ∗name)

  Loads default controls.

- TRuntimeControls (const char ∗name, const char ∗runFileName) throw (SPARK::XInitialization)

  Loads controls from file "runFileName".

- ∼TRuntimeControls () throw ()

  Trivial destructor.

Access methods

- const char ∗ GetName () const

  Returns name of runtime controls as `const char*`.

- const char ∗ GetRunFileName () const

  Returns name of the ∗.run file as `const char*`.

- const char ∗ GetInitialSnapshotFileName () const

  Returns the name of the initial snapshot file as `const char*`.

- void SetInitialSnapshotFileName (const char ∗str)

  Sets the name of the initial snapshot file to `str`.

- const char ∗ GetFinalSnapshotFileName () const

  Returns the name of the final snapshot file as `const char*`.

- void SetFinalSnapshotFileName (const char ∗str)

  Sets the name of the final snapshot file to `str`.

- TInputFiles & GetInputFiles ()

  Returns a reference to the TInputFiles object that contains the list of the input files specified in the ∗.run file.

- const TInputFiles & GetInputFiles () const

  Returns a const reference to the TInputFiles object that contains the list of the input files specified in the ∗.run file.

- const char ∗ GetOutputFileName () const

  Returns the name of the output file as `const char*` *where the variables tagged with the* `REPORT` *keyword are reported.*

- void SetOutputFileName (const char *filename)
  Sets the name of the output file to `filename` *where the variables tagged with the* `REPORT` *keyword will be reported.*
- const char * GetInitialWallClock () const
  Returns the string describing the initial wall clock as `const char*`.
- void SetInitialWallClock (const char *str)
  Sets the string describing the initial wall clock to `str`.
- const char * GetTimeUnit () const
  Returns the string describing the time unit used in the physical model as `const char*`.
- void SetTimeUnit (const char *str)
  Sets the string describing the time unit used in the physical model to `str`.
- unsigned GetNumPastValues () const
  Returns the number of successive past values that the problem simulator keeps track of as `unsigned`.
- void SetNumPastValues (unsigned numPastValues)
  Sets the number of successive past values that the problem simulator keeps track of to `numPastValues`.
- unsigned GetDiagnosticLevel () const
  Returns the diagnostic level as `unsigned`.
- void SetDiagnosticLevel (unsigned level)
  Sets the diagnostic level to `level`.
- double GetInitialTime () const
  Returns the initial time value as `double`.
- void SetInitialTime (double initialTime)
  Sets the initial time value to `initialTime`.
- double GetFinalTime () const
  Returns the final time value as `double`.
- void SetFinalTime (double finalTime)
  Sets the final time value to `finalTime`.
- void SetInfiniteFinalTime ()
  Sets the final time value to infinity.
- double GetInitialTimeStep () const
  Returns the value of the intial time step as `double`.
- void SetInitialTimeStep (double initialTimeStep)
  Sets the value of the initial time step to `initialTimeStep`.
- double GetMinTimeStep () const
  Returns the value for the minimum time step allowed as `double`.
- void SetMinTimeStep (double minTimeStep)
  Sets the value for the minimum time step allowed to `minTimeStep`.
- double GetMaxTimeStep () const
  Returns the value for the maximum time step allowed as `double`.
- void SetMaxTimeStep (double maxTimeStep)

Sets the value for the maximum time step allowed to `maxTimeStep`.

- double GetFirstReport () const
  Returns the time for the first report to be generated as `double`.
- void SetFirstReport (double firstReport)
  Sets the time of the first report to `firstReport`.
- double GetReportCycle () const
  Returns the time step used to generate the reports as `double`.
- void SetReportCycle (double reportCycle)
  Sets the time step used to generate the reports to `reportCycle`.
- unsigned GetVariableTimeStep () const
  Returns the flag that controls whether variable time stepping operation is on or off as `unsigned`.
- void SetVariableTimeStep (unsigned flag)
  Sets the flag taht controls whether variable time stepping operation is on or off to `flag`.
- unsigned GetConsistentInitialCalculation () const
  Returns the flag that controls whether or not to perform an initial consistent calculation for the first step of the simulation as `unsigned`.
- void SetConsistentInitialCalculation (unsigned flag)

I/O Operations

- void Write (std::ostream &os, const std::string &before) const
  Writes the runtime controls to `os`.

Misc operation

- unsigned ValidateControls (std::ostream &os) const
  Checks specified controls and returns the number of invalid controls.
- void Reset ()
  Resets all controls to the default values.

3.10.1 Detailed Description

Wrapper class for all the runtime control information required to initialize a TProblem object in order to make a simulation run.

Examples:
multiproblem_example1.cpp, and sparksolver.cpp.

3.10.2 Member Typedef Documentation

3.10.2.1 typedef std::vector<std::string> SPARK::TRuntimeControls::TInputFiles

Type used to describe the list of input files specified in the *.run file with the key `InputFiles`.

3.10.3 Constructor & Destructor Documentation

3.10.3.1 SPARK::TRuntimeControls::TRuntimeControls (const char * *name*)

Loads default controls.

3.10.3.2 SPARK::TRuntimeControls::TRuntimeControls (const char * *name*, const char * *runFileName*) throw (SPARK::XInitialization)

Loads controls from file "runFileName".

Exceptions:

SPARK::XInitialization Thrown if the runtime controls could not be validated.

3.10.3.3 SPARK::TRuntimeControls::~TRuntimeControls () throw ()

Trivial destructor.

3.10.4 Member Function Documentation

3.10.4.1 const char* SPARK::TRuntimeControls::GetName () const [inline]

Returns name of runtime controls as `const char*`.

3.10.4.2 const char* SPARK::TRuntimeControls::GetRunFileName () const [inline]

Returns name of the *.run file as `const char*`.

3.10.4.3 const char* SPARK::TRuntimeControls::GetInitialSnapshotFileName () const [inline]

Returns the name of the initial snapshot file as `const char*`.

Note:

By default, no initial snapshot file will be generated.

3.10.4.4 void SPARK::TRuntimeControls::SetInitialSnapshotFileName (const char * *str*) [inline]

Sets the name of the initial snapshot file to `str`.

3.10.4.5 const char* SPARK::TRuntimeControls::GetFinalSnapshotFileName () const [inline]

Returns the name of the final snapshot file as `const char*`.

Note:

By default, no final snapshot file will be generated.

3.10.4.6 void SPARK::TRuntimeControls::SetFinalSnapshotFileName (const char * *str*) [inline]

Sets the name of the final snapshot file to `str`.

3.10.4.7 TInputFiles& SPARK::TRuntimeControls::GetInputFiles () [inline]

Returns a reference to the TInputFiles object that contains the list of the input files specified in the *.run file.

3.10.4.8 const TInputFiles& SPARK::TRuntimeControls::GetInputFiles () const [inline]

Returns a const reference to the TInputFiles object that contains the list of the input files specified in the *.run file.

3.10.4.9 const char* SPARK::TRuntimeControls::GetOutputFileName () const [inline]

Returns the name of the output file as `const char*` where the variables tagged with the `REPORT` keyword are reported.

3.10.4.10 void SPARK::TRuntimeControls::SetOutputFileName (const char * *filename*) [inline]

Sets the name of the output file to `filename` where the variables tagged with the `REPORT` keyword will be reported.

3.10.4.11 const char* SPARK::TRuntimeControls::GetInitialWallClock () const [inline]

Returns the string describing the initial wall clock as `const char*`.

The format of the initial wall clock string follows: "mm/dd/yyyy hh:mm:ss"

Note:
The initial wall clock string is parsed by the URL engine and is used for synchronization with various weather files.

3.10.4.12 void SPARK::TRuntimeControls::SetInitialWallClock (const char * *str*) [inline]

Sets the string describing the initial wall clock to `str`.

3.10.4.13 const char* SPARK::TRuntimeControls::GetTimeUnit () const [inline]

Returns the string describing the time unit used in the physical model as `const char*`.

- The time unit is used by the URL engine to initialize the weather file readers with the proper time unit used in the simulation model.

- Also, this time unit will be used to override the unit strings in the global time and global time step variables in the physical model for consistency reasons.

Note:
It should be a unit string reecognized by the URL engine.

3.10.4.14 void SPARK::TRuntimeControls::SetTimeUnit (const char * *str*) [inline]

Sets the string describing the time unit used in the physical model to `str`.

3.10.4.15 unsigned SPARK::TRuntimeControls::GetNumPastValues () const [inline]

Returns the number of successive past values that the problem simulator keeps track of as `unsigned`.

3.10.4.16 void SPARK::TRuntimeControls::SetNumPastValues (unsigned *numPastValues*) [inline]

Sets the number of successive past values that the problem simulator keeps track of to `numPastValues`.

3.10.4.17 unsigned SPARK::TRuntimeControls::GetDiagnosticLevel () const [inline]

Returns the diagnostic level as `unsigned`.

The possible values for the diagnostic level are defined in TProblem::DiagnosticTypes

Note:
The value 0 indicates that no diagnostic will be generated at runtime.

3.10.4.18 void SPARK::TRuntimeControls::SetDiagnosticLevel (unsigned *level*) [inline]

Sets the diagnostic level to `level`.

3.10.4.19 double SPARK::TRuntimeControls::GetInitialTime () const [inline]

Returns the initial time value as `double`.

3.10.4.20 void SPARK::TRuntimeControls::SetInitialTime (double *initialTime*) [inline]

Sets the initial time value to `initialTime`.

3.10.4.21 double SPARK::TRuntimeControls::GetFinalTime () const [inline]

Returns the final time value as `double`.

3.10.4.22 void SPARK::TRuntimeControls::SetFinalTime (double *finalTime*) [inline]

Sets the final time value to `finalTime`.

3.10.4.23 void SPARK::TRuntimeControls::SetInfiniteFinalTime ()

Sets the final time value to infinity.

The problem simulator will not stopped unless a stop request, abort request or a set stop time request is posted. This is equivalent to specifying `FinalTime( * ())` in the *.run file.

3.10.4.24 double SPARK::TRuntimeControls::GetInitialTimeStep () const [inline]

Returns the value of the intial time step as `double`.

Note:
If the value is zero, then the clock will not be advanced and only one step will be calculated.

3.10.4.25 void SPARK::TRuntimeControls::SetInitialTimeStep (double *initialTimeStep*) [inline]

Sets the value of the initial time step to `initialTimeStep`.

3.10.4.26 double SPARK::TRuntimeControls::GetMinTimeStep () const [inline]

Returns the value for the minimum time step allowed as `double`.

3.10.4.27 void SPARK::TRuntimeControls::SetMinTimeStep (double *minTimeStep*) [inline]

Sets the value for the minimum time step allowed to `minTimeStep`.

3.10.4.28 double SPARK::TRuntimeControls::GetMaxTimeStep () const [inline]

Returns the value for the maximum time step allowed as `double`.

3.10.4.29 void SPARK::TRuntimeControls::SetMaxTimeStep (double *maxTimeStep*) [inline]

Sets the value for the maximum time step allowed to `maxTimeStep`.

3.10.4.30 double SPARK::TRuntimeControls::GetFirstReport () const [inline]

Returns the time for the first report to be generated as `double`.

3.10.4.31 void SPARK::TRuntimeControls::SetFirstReport (double *firstReport*) [inline]

Sets the time of the first report to `firstReport`.

3.10.4.32 double SPARK::TRuntimeControls::GetReportCycle () const [inline]

Returns the time step used to generate the reports as `double`.

This report time step is independent from the simulation time step. However, if the simulation supports variable time stepping, then the simultation time step will be adapted to synchronize with the `i` desired reporting times $t_{report}[i]$: $t_{report}[i] = FirstReport + i * ReportCycle$

Note:

If $ReportCycle == 0$, then reports are generate at each simulation step.

3.10.4.33 void SPARK::TRuntimeControls::SetReportCycle (double *reportCycle*) [inline]

Sets the time step used to generate the reports to `reportCycle`.

3.10.4.34 unsigned SPARK::TRuntimeControls::GetVariableTimeStep () const [inline]

Returns the flag that controls whether variable time stepping operation is on or off as `unsigned`.

- If `VariableTimeStep==0` then the time step remains constant during the course of the simulation.

- If `VariableTimeStep==1` then the time step is adapted to satisfy the meeting points and the time step requests during the course of the simulation.

3.10.4.35 void SPARK::TRuntimeControls::SetVariableTimeStep (unsigned *flag*) `[inline]`

Sets the flag taht controls whether variable time stepping operation is on or off to `flag`.

3.10.4.36 unsigned SPARK::TRuntimeControls::GetConsistentInitialCalculation () const `[inline]`

Returns the flag that controls whether or not to perform an initial consistent calculation for the first step of the simulation as `unsigned`.

- If `ConsistentInitialCalculation == 0` then no initial calculation is performed.
- If `ConsistentInitialCalculation == 1` then an initial calculation is performed in order to ensure a consistent of initial values.

The initial consistent calculation consists in computing a static step to solve for the time-derivatives, the dynamic variables being set to their user-specified initial values.

Note:

It is highly recommended to perform an initial consistent calculation for dynamic problems to make sure that the integrator classes do rely on a consistent set of initial conditions. Otherwise, it is likely that the initial error will accumulate and is likely to produce an inaccurate dynamic solution.

3.10.4.37 void SPARK::TRuntimeControls::SetConsistentInitialCalculation (unsigned *flag*) `[inline]`

Sets the flag that controls whether or not to perform an initial consistent calculation for the first step of the simulation to `flag`

3.10.4.38 void SPARK::TRuntimeControls::Write (std::ostream & *os*, const std::string & *before*) const

Writes the runtime controls to `os`.

3.10.4.39 unsigned SPARK::TRuntimeControls::ValidateControls (std::ostream & *os*) const

Checks specified controls and returns the number of invalid controls.

Note:

This method is called in TRuntimeControls(const char* runFileName). If using the default constructor TRuntimeControls(), then you should call this method explicitly to make sure that the controls are valid.

3.10.4.40 void SPARK::TRuntimeControls::Reset ()

Resets all controls to the default values.

See namespace SPARK::DefaultRuntimeControls for the list of the default controls.

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

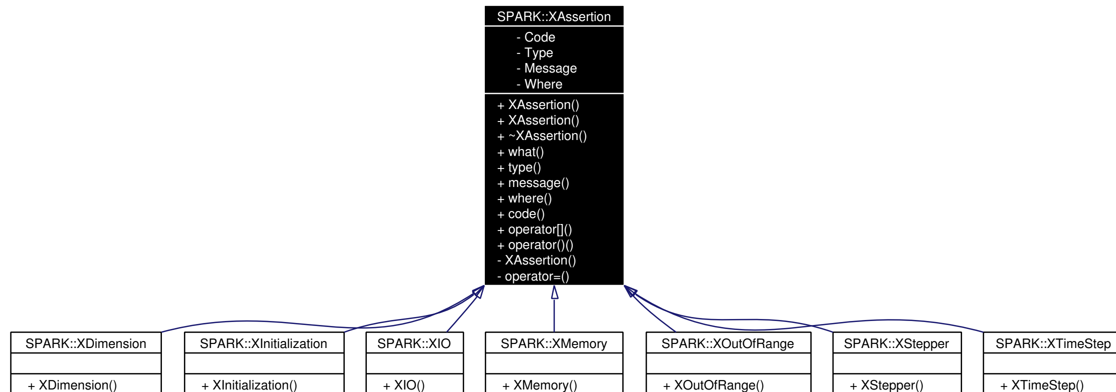
- ctrls.h

3.11 SPARK::XAssertion Class Reference

Base class for all SPARK exceptions.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XAssertion:



Public Types

- typedef enum ExitCodes TCode

    Locally rename ExitCodes as TCode.

Public Member Functions

- XAssertion (const std::string &type, const std::string &where, const std::string &message, TCode code)

    Constructor with message specified as std::string.

- XAssertion (const XAssertion &x)

    Copy constructor.

- virtual ∼XAssertion () throw ()

    Trivial destructor.

- virtual const char ∗ what () const throw ()

    Returns a description of the exception as a const char∗.

- const char ∗ type () const throw ()

    Returns type of the exception as const char∗.

- const char ∗ message () const throw ()

    Returns message associated with the exception as const char∗.

- const char ∗ where () const throw ()

    Returns where the exception was thrown from as const char∗.

- TCode code () const throw ()

Returns code of the exception as TCode.

- XAssertion & operator[ ] (const std::string &where)

  Appends sender's information to Where string following " <- ".

- XAssertion & operator() (const std::string &msg)

  Appends msg to Message string on a new line.

3.11.1 Detailed Description

Base class for all SPARK exceptions.

Examples:
multiproblem_example1.cpp, and sparksolver.cpp.

3.11.2 Member Typedef Documentation

3.11.2.1 typedef enum ExitCodes SPARK::XAssertion::TCode

Locally rename ExitCodes as TCode.

3.11.3 Constructor & Destructor Documentation

3.11.3.1 SPARK::XAssertion::XAssertion (const std::string & *type*, const std::string & *where*, const std::string & *message*, TCode *code*) `[inline]`

Constructor with message specified as std::string.

3.11.3.2 SPARK::XAssertion::XAssertion (const XAssertion & *x*) `[inline]`

Copy constructor.

3.11.3.3 virtual SPARK::XAssertion::∼XAssertion () throw () `[inline, virtual]`

Trivial destructor.

3.11.4 Member Function Documentation

3.11.4.1 virtual const char∗ SPARK::XAssertion::what () const throw () `[virtual]`

Returns a description of the exception as a const char∗.

3.11.4.2 const char∗ SPARK::XAssertion::type () const throw () `[inline]`

Returns type of the exception as const char∗.

3.11.4.3 const char* SPARK::XAssertion::message () const throw () [inline]

Returns message associated with the exception as const char*.

3.11.4.4 const char* SPARK::XAssertion::where () const throw () [inline]

Returns where the exception was thrown from as const char*.

3.11.4.5 TCode SPARK::XAssertion::code () const throw () [inline]

Returns code of the exception as TCode.

3.11.4.6]

XAssertion& SPARK::XAssertion::operator[ ] (const std::string & *where*)

Appends sender's information to Where string following " <- ".

3.11.4.7 XAssertion& SPARK::XAssertion::operator() (const std::string & *msg*)

Appends msg to Message string on a new line.

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

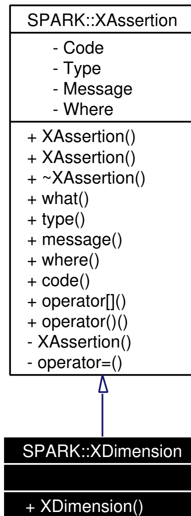
- exceptions.h

3.12 SPARK::XDimension Class Reference

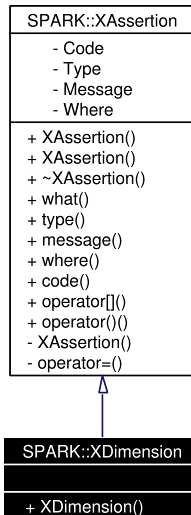
Indicates that a runtime error occured due to mismatched dimension.

```
#include <exceptions.h>
```

Inheritance diagram for SPARK::XDimension:



Collaboration diagram for SPARK::XDimension:



Public Member Functions

- XDimension (const std::string &where, const std::string &message)

  Constructor.

3.12.1 Detailed Description

Indicates that a runtime error occured due to mismatched dimension.

3.12.2 Constructor & Destructor Documentation

3.12.2.1 SPARK::XDimension::XDimension (const std::string & *where*, const std::string & *message*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

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

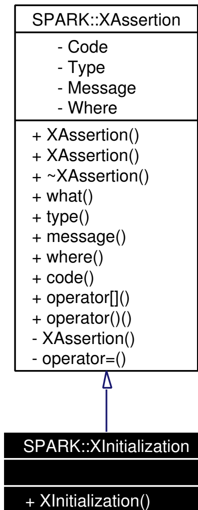
- exceptions.h

3.13 SPARK::XInitialization Class Reference

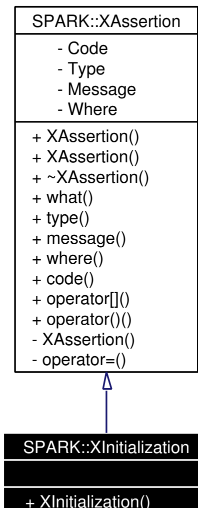
Indicates that a runtime error occured while initializing an object.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XInitialization:



Collaboration diagram for SPARK::XInitialization:



Public Member Functions

- XInitialization (const std::string &where, const std::string &message, TCode code)

  Constructor.

3.13.1 Detailed Description

Indicates that a runtime error occured while initializing an object.

3.13.2 Constructor & Destructor Documentation

3.13.2.1 SPARK::XInitialization::XInitialization (const std::string & *where*, const std::string & *message*, TCode *code*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

code ExitCodes used to qualify the type of initilazation exception

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

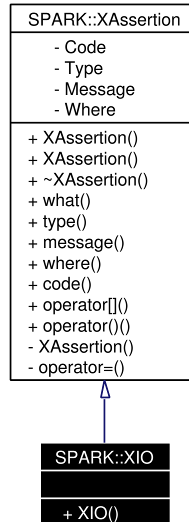
- exceptions.h

3.14 SPARK::XIO Class Reference

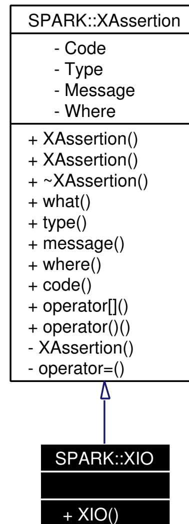
Indicates that a runtime error occured while performing an IO operation.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XIO:



Collaboration diagram for SPARK::XIO:



Public Member Functions

- XIO (const std::string &where, const std::string &message)

  Constructor.

3.14.1 Detailed Description

Indicates that a runtime error occured while performing an IO operation.

3.14.2 Constructor & Destructor Documentation

3.14.2.1 SPARK::XIO::XIO (const std::string & *where*, const std::string & *message*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

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

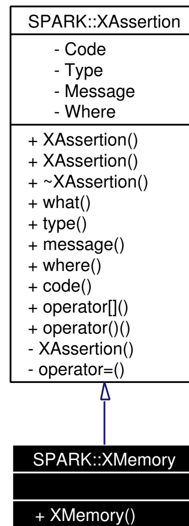
- exceptions.h

3.15 SPARK::XMemory Class Reference

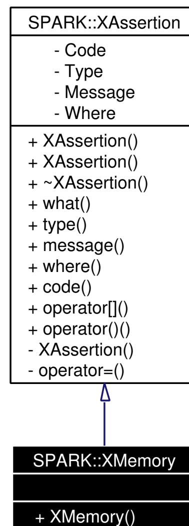
Indicates that a runtime error occured because memory could not be allocated.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XMemory:



Collaboration diagram for SPARK::XMemory:



Public Member Functions

- XMemory (const std::string &where, const std::string &message)

    Constructor.

3.15.1 Detailed Description

Indicates that a runtime error occured because memory could not be allocated.

3.15.2 Constructor & Destructor Documentation

3.15.2.1 SPARK::XMemory::XMemory (const std::string & *where*, const std::string & *message*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

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

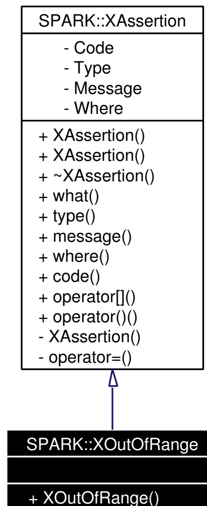
- exceptions.h

3.16 SPARK::XOutOfRange Class Reference

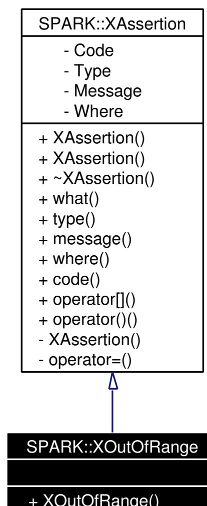
Indicates that a runtime error occured due to an out of range access operation on a container.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XOutOfRange:



Collaboration diagram for SPARK::XOutOfRange:



Public Member Functions

- XOutOfRange (const std::string &where, const std::string &message)

  Constructor.

3.16.1 Detailed Description

Indicates that a runtime error occured due to an out of range access operation on a container.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 SPARK::XOutOfRange::XOutOfRange (const std::string & *where*, const std::string & *message*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

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

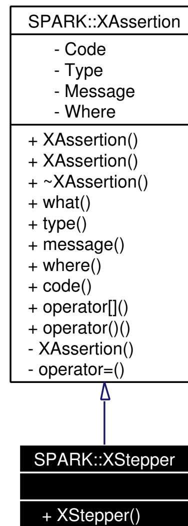
- exceptions.h

3.17 SPARK::XStepper Class Reference

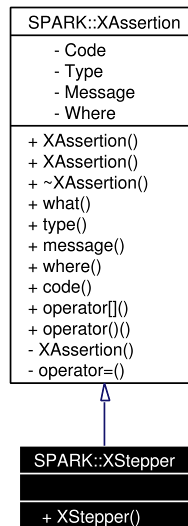
Inidicates that stepping to the next step failed.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XStepper:



Collaboration diagram for SPARK::XStepper:



Public Member Functions

- XStepper (const std::string &where, const std::string &message)

    Constructor.

3.17.1 Detailed Description

Inidicates that stepping to the next step failed.

3.17.2 Constructor & Destructor Documentation

3.17.2.1 SPARK::XStepper::XStepper (const std::string & *where*, const std::string & *message*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

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

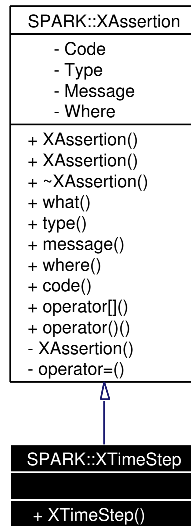
- exceptions.h

3.18 SPARK::XTimeStep Class Reference

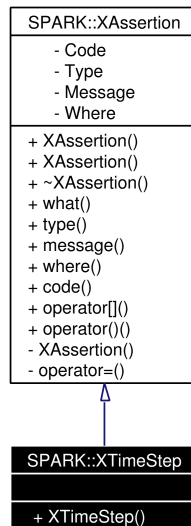
Indicates that a runtime error occured while adapting the time step.

`#include <exceptions.h>`

Inheritance diagram for SPARK::XTimeStep:



Collaboration diagram for SPARK::XTimeStep:



Public Member Functions

- XTimeStep (const std::string &where, const std::string &message)

  Constructor.

3.18.1 Detailed Description

Indicates that a runtime error occured while adapting the time step.

3.18.2 Constructor & Destructor Documentation

3.18.2.1 SPARK::XTimeStep::XTimeStep (const std::string & *where*, const std::string & *message*) `[inline]`

Constructor.

Parameters:

where string specified as `const std::string&` indicating where in the source code this exception is thrown from

message string specified as `const std::string&` documenting the reason for throwing this exception

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

- exceptions.h

Chapter 4

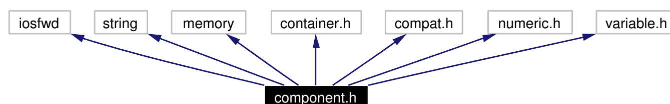
SPARK Build Process and Problem Driver API File Documentation

4.1 component.h File Reference

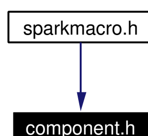
Declaration of the SPARK::TComponent class.

```
#include <iosfwd>
#include <string>
#include <memory>
#include "container.h"
#include "compat.h"
#include "numeric.h"
#include "variable.h"
```

Include dependency graph for component.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK

4.1.1 Detailed Description

Declaration of the SPARK::TComponent class.

Author:
Dimitri Curtil (LBNL/SRG)

Date:
September 9, 2002

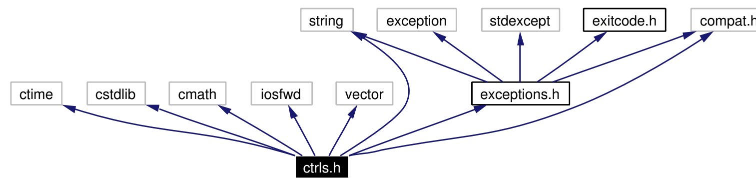
Attention:
PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

4.2 ctrls.h File Reference

Header file for the definition of the SPARK::TRuntimeControls class.

```
#include <ctime>
#include <cstdlib>
#include <cmath>
#include <iosfwd>
#include <vector>
#include <string>
#include "exceptions.h"
#include "compat.h"
```

Include dependency graph for ctrls.h:



Namespaces

- namespace SPARK::DefaultRuntimeControls
- namespace SPARK

4.2.1 Detailed Description

Header file for the definition of the SPARK::TRuntimeControls class.

The SPARK::TRuntimeControls class is a wrapper class around all run-time control data needed to run a SPARK problem. It also provides access and predicate methods for each runtime control.

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:

Dimitri Curtil (LBNL/SRG)

Date:

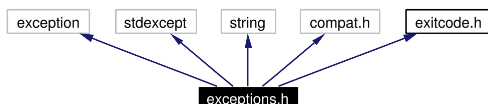
January 15, 2003

4.3 exceptions.h File Reference

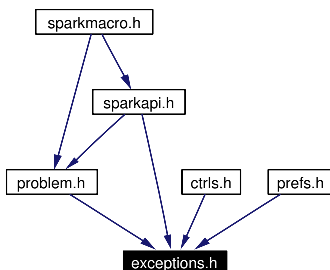
Declaration of the generic exception classes used in the SPARK solver.

```
#include <exception>
#include <stdexcept>
#include <string>
#include "compat.h"
#include "exitcode.h"
```

Include dependency graph for exceptions.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK

4.3.1 Detailed Description

Declaration of the generic exception classes used in the SPARK solver.

- class SPARK::XAssertion
- class SPARK::XDimension
- class SPARK::XOutOfRange
- class SPARK::XMemory
- class SPARK::XInitialization
- class SPARK::XIO
- class SPARK::XTimeStep
- class SPARK::XStepper

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:

Dimitri Curtil (LBNL/SRG)

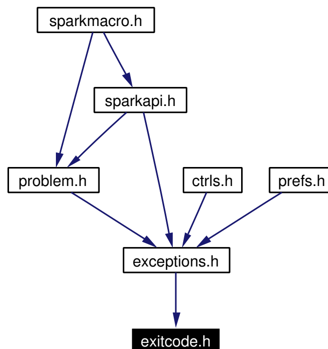
Date:

September 5, 2002

4.4 exitcode.h File Reference

Definition of the exit codes returned by the SPARK solver.

This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK

4.4.1 Detailed Description

Definition of the exit codes returned by the SPARK solver.

This header file defines the SPARK::ExitCodes enum that the is returned by the SPARK solver in case of abnormal execution. The main() driver function returns the value SPARK::ExitCode_OK if the simulation is successful.

Author:

Dimitri Curtil (LBNL/SRG)

Date:

June 28, 2002

Attention:

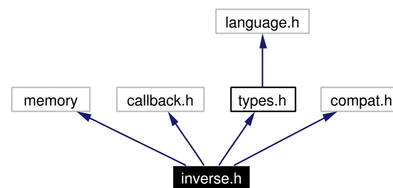
PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

4.5 inverse.h File Reference

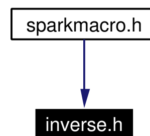
Declaration of the SPARK::TInverse class.

```
#include <memory>
#include "callback.h"
#include "types.h"
#include "compat.h"
```

Include dependency graph for inverse.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK

4.5.1 Detailed Description

Declaration of the SPARK::TInverse class.

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:

Dimitri Curtil (LBNL/SRG)

Date:

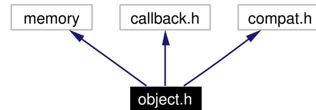
January 10, 2003

4.6 object.h File Reference

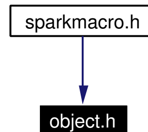
Declaration of the SPARK::TObject class.

```
#include <memory>
#include "callback.h"
#include "compat.h"
```

Include dependency graph for object.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK

4.6.1 Detailed Description

Declaration of the SPARK::TObject class.

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:

Dimitri Curtil (LBNL/SRG)

Date:

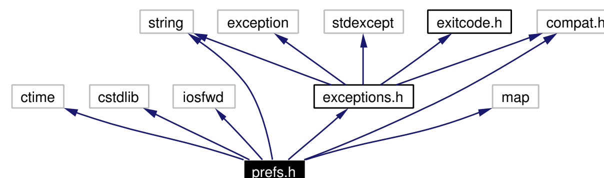
January 12, 2003

4.7 prefs.h File Reference

Header file for the definition of the classes SPARK::TGlobalSettings, SPARK::TComponentSettings and SPARK::TPreferenceSettings.

```
#include <ctime>
#include <cstdlib>
#include <iosfwd>
#include <string>
#include <map>
#include "exceptions.h"
#include "compat.h"
```

Include dependency graph for prefs.h:



Namespaces

- namespace SPARK
- namespace SPARK::DefaultGlobalSettings
- namespace SPARK::DefaultComponentSettings

4.7.1 Detailed Description

Header file for the definition of the classes SPARK::TGlobalSettings, SPARK::TComponentSettings and SPARK::TPreferenceSettings.

The SPARK::TPreferenceSettings class is a wrapper class that manages the settings for all components, including the default settings and the global settings.

The SPARK::TGlobalSettings class is a wrapper class that contains the global solution method settings defined at the problem level in the section `GlobalSettings` of the *.prf file.

The SPARK::TComponentSettings class is a wrapper class that contains the solution method settings defined for each component under the section `ComponentSettings` of the *.prf file.

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:

Dimitri Curtil (LBNL/SRG)

Date:

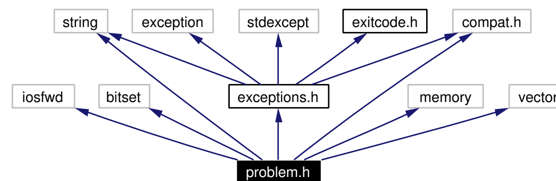
January 6, 2003

4.8 problem.h File Reference

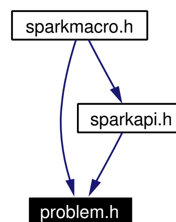
Declaration of the classes SPARK::TProblem and SPARK::TProblem::TState.

```
#include <iosfwd>
#include <bitset>
#include <string>
#include <memory>
#include <vector>
#include "compat.h"
#include "exceptions.h"
```

Include dependency graph for problem.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK
- namespace **SPARK::Requests**

Defines

- #define __PROBLEM_H__

4.8.1 Detailed Description

Declaration of the classes SPARK::TProblem and SPARK::TProblem::TState.

4.8.2 Define Documentation

4.8.2.1 #define __PROBLEM_H__

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:

Dimitri Curtil (LBNL/SRG)

Date:

April 20, 2001

4.9 sparkapi.h File Reference

Header file defining the functions required to manage the SPARK problems in a driver function.

`#include <iosfwd>`

`#include <string>`

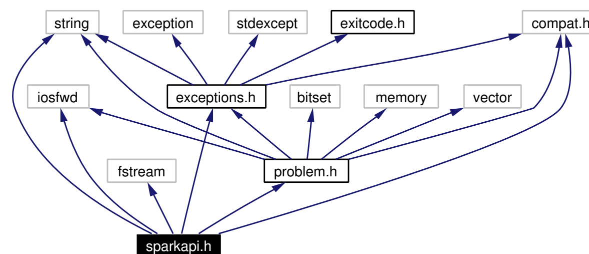
`#include <fstream>`

`#include "problem.h"`

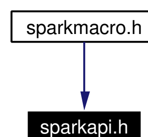
`#include "exceptions.h"`

`#include "compat.h"`

Include dependency graph for sparkapi.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK
- namespace SPARK::Problem::StaticBuild
- namespace SPARK::Problem
- namespace SPARK::Problem::DynamicBuild

4.9.1 Detailed Description

Header file defining the functions required to manage the SPARK problems in a driver function.

This file should be included in the C++ source file where the main driver function is implemented.

It defines functions to start, end and exit the simulation session :

- SPARK::Start()
- SPARK::End()
- SPARK::ExitWithError()

Utility functions :

- SPARK::Log()
- SPARK::GetFileName()

Access functions :

- SPARK::GetProgramName()
- SPARK::GetBaseName()
- SPARK::GetVersion()

- SPARK::GetRunLog()
- SPARK::GetRunLogFilename()

- SPARK::GetErrorLog()
- SPARK::GetErrorLogFilename()

SPARK problem API functions used to manage SPARK::TProblem objects :

- SPARK::Problem::StaticBuild::ParseCommandLine()
- SPARK::Problem::StaticBuild::ShowCommandLineUsage()
- SPARK::Problem::StaticBuild::Load()

- SPARK::Problem::DynamicBuild::ParseCommandLine()
- SPARK::Problem::DynamicBuild::ShowCommandLineUsage()
- SPARK::Problem::DynamicBuild::Load()

- SPARK::Problem::Get()
- SPARK::Problem::Unload()
- SPARK::Problem::Initialize()
- SPARK::Problem::LoadPreferenceSettings()
- SPARK::Problem::Terminate()
- SPARK::Problem::Save()
- SPARK::Problem::Restore()
- SPARK::Problem::Simulate()
- SPARK::Problem::Step()
- SPARK::Problem::StaticStep()

Author:

Dimitri Curtil (LBNL/SRG)

Date:

May 21, 2002 Extended API in Sept, 2003

Attention:

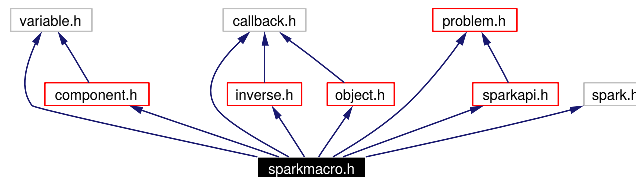
PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

4.10 sparkmacro.h File Reference

Header file defining the preprocessor macros required to build a static SPARK problem from the ∗.cpp file describing the problem topology.

```
#include "variable.h"
#include "callback.h"
#include "inverse.h"
#include "object.h"
#include "component.h"
#include "problem.h"
#include "sparkapi.h"
#include "spark.h"
```

Include dependency graph for sparkmacro.h:



Defines

- #define _QUOTE_(x) #x

  Causes the corresponding actual argument to be enclosed in double quotation marks.

- #define _NAMESPACE_(name) _##name

  Prefixes name with underscore to make it a valid qualifier for a namespace.

- #define START_ARRAY(size)

  Defines a non-empty array for elements of the pre-defined type element_type. See END_ARRAY.

- #define END_ARRAY };

  Defines the end of a non-empty array. See START_ARRAY.

- #define EMPTY_ARRAY

  Defines an empty array of pre-defined type element_type.

- #define START_PROBLEM(name)

  Starts the definition of a TProblem object named name. See END_PROBLEM.

- #define END_PROBLEM

  Ends the definition of a TProblem object. Also takes care of registering the compiled problem pointer with SPARK environment. See START_PROBLEM.

- #define START_INVERSES

  Start the declaration of list of inverse functions. See END_INVERSES.

- #define END_INVERSES };

  Terminates the declaration of the list of inverse functions. See START_INVERSES.

4.10.1 Detailed Description

Header file defining the preprocessor macros required to build a static SPARK problem from the *.cpp file describing the problem topology.

This file is included in the "problem.cpp" file by setupcpp. It defines the macros required to compile the problem. There is a one-to-one matching between the XML DTD used in the *.xml file and the preprocessor macros for each section. See the file "spark2.dtd" in the example section.

Author:
Dimitri Curtil (LBNL/SRG)

Date:
July 9, 2002

Attention:
PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

4.10.2 Define Documentation

4.10.2.1 #define _QUOTE_(x) #x

Causes the corresponding actual argument to be enclosed in double quotation marks.

4.10.2.2 #define _NAMESPACE_(name) _##name

Prefixes name with underscore to make it a valid qualifier for a namespace.

4.10.2.3 #define START_ARRAY(size)

Value:

```
const unsigned Size = size; \
        element_type Array[] = { \
```

Defines a non-empty array for elements of the pre-defined type element_type. See END_ARRAY.

4.10.2.4 #define END_ARRAY };

Defines the end of a non-empty array. See START_ARRAY.

4.10.2.5 #define EMPTY_ARRAY

Value:

```
const unsigned Size = 0; \
        element_type* Array = 0; \
```

Defines an empty array of pre-defined type element_type.

4.10.2.6 #define START_PROBLEM(name)

Value:

```
namespace Problem__##name  {  \
                const char* Name = _QUOTE_(name); \
```

Starts the definition of a TProblem object named name. See END_PROBLEM.

4.10.2.7 #define END_PROBLEM

Value:

```
SPARK::TProblem Element( \
                        Inverses::Size, Inverses::Array, \
                        Variables::Size, Variables::Array, \
                        Components::Size, Components::Array \
                ); \
                bool IsRegistered = SPARK::Problem::RegisterStaticInstance( Name, &Element ); \
        }; \
```

Ends the definition of a TProblem object. Also takes care of registering the compiled problem pointer with SPARK environment. See START_PROBLEM.

4.10.2.8 #define START_INVERSES

Value:

```
namespace Inverses { \
        typedef SPARK::TInverse*  element_type; \
```

Start the declaration of list of inverse functions. See END_INVERSES.

4.10.2.9 #define END_INVERSES };

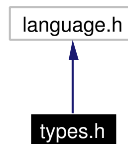
Terminates the declaration of the list of inverse functions. See START_INVERSES.

4.11 types.h File Reference

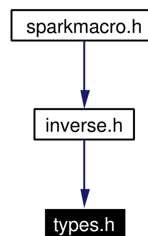
Declaration of the type enums describing the various entities in a SPARK problem.

```
#include "language.h"
```

Include dependency graph for types.h:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace SPARK

4.11.1 Detailed Description

Declaration of the type enums describing the various entities in a SPARK problem.

Declaration of :

- atomic class types in SPARK::AtomicTypes
- function types in SPARK::FunTypes
- callback types in SPARK::CallbackTypes
- callback return types in SPARK::ReturnTypes
- callback prototypes in SPARK::ProtoTypes
- variable types in SPARK::VariableTypes
- request types in SPARK::RequestTypes

Attention:

PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA . PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.

Author:
Dimitri Curtil (LBNL/SRG)

Date:
January 5, 2003

Chapter 5

SPARK Build Process and Problem Driver API Example Documentation

5.1 multiproblem_example1.cpp

The following code snippet is an example of a customized driver function that solves two problems respectively defined in the files spring.xml and room_fc.xml. These two simple problems are provided with the SPARK distribution and can be found in the `examples` subdirectory of the SPARK installation directory.

The code snippet shows the variable declarations and the sequence of operations required to solve the two SPARK problems within the same simulation session.

To build the simulator, you first have to build the problems room_fc.pr and spring.pr in their respective directories. Then you can use the multiproblem makefile Multi-problem command file to generate the customized simulator.

```
///////////////////////////////////////////////////////////////////////////////////////
// File nultiproblem_example1.cpp
//
// Definition of a SPARK driver function for multiple problems instantiated from the
// descriptions contained in the files :
// - spring.cpp
// - room_fc
// that you can find in the spark/examples/ subdirectory.
//
// This driver function is not used by the VisualSPARK distribution. Its purpose is to
// show how a multi-problem driver could be implemented.
//
// Make sure that all *.prf, *.run and *.xml files used in this driver are provided in the
// working directory.
//
// All problems are loaded at runtime. Prepare the dynamic libraries with the atomic classes
// required by each probem.
//
///////////////////////////////////////////////////////////////////////////////////////
//
// Author  Dimitri Curtil (LBNL/SRG)
// Date    March 30, 2003
//
///////////////////////////////////////////////////////////////////////////////////////
// PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
// PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA .
//   PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.
//
///////////////////////////////////////////////////////////////////////////////////////


// The debugger can't handle symbols more than 255 characters long.
```

```
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)


#ifdef MacVersion
#       include <console>
#endif

#include <iostream>
using std::cout;
using std::ofstream;
using std::endl;

#include <sstream>
using std::ostringstream;
using std::ends;

#include <string>
using std::string;


///////////////////////////////////////////////////////////////////////////////////////////
//
// Required header files
//
#include "sparkapi.h"    // for various API functions needed to manage the SPARK problem
#include "problem.h"     // for TProblem definition
#include "ctrls.h"       // for TRuntimeControls definition
#include "prefs.h"       // for TPreferenceSettings definition
#include "exitcode.h"    // for SPARK::TExitCode and SPARK::ExitWithError()
#include "exceptions.h"  // for SPARK exception classes
///////////////////////////////////////////////////////////////////////////////////////////



///////////////////////////////////////////////////////////////////////////////////////////
// Function name        : RunProblem
// Description      : Loads and solves problem for the specified instance name,
//                    and the *.run, *.prf and *.xml files.
// Return type          : void
// Argument         : const string& instanceName
// Argument         : const string& runFileName
// Argument         : const string& prfFileName
// Argument         : const string& xmlFileName
void RunProblem(
    const string& instanceName,
    const string& runFileName,
    const string& prfFileName,
    const string& xmlFileName )
{
    // We assume that the needed names are valid!
    // Construct TProblem object with:
    // - unique problem name
    // - name of the xml file with the problem description
    // Check whether problem has been successfully loaded or not
    if ( !SPARK::Problem::DynamicBuild::Load( instanceName.c_str(), xmlFileName.c_str() ) ) {
        ostringstream ErrMsg;

        ErrMsg << "Could not load the problem \"" << instanceName.c_str() << "\" " << ends;

        SPARK::ExitWithError(
            SPARK::ExitCode_ERROR_INVALID_PROBLEM,
            "Generic Problem Driver",
            ErrMsg.str()
        );
    }

    // Prepare run-time controls and preference settings
    ostringstream ControlsName;
```

```
    ControlsName << "Default SPARK driver for problem (" << instanceName.c_str() << ")" << ends;

    SPARK::TRuntimeControls RuntimeControls(
        ControlsName.str().c_str(),   // controls name
        runFileName.c_str()           // *.run file name
    );

    SPARK::TPreferenceSettings PreferenceSettings( prfFileName.c_str() );

    // Retrieve pointer to SPARK::TProblem instance
    SPARK::TProblem* P = SPARK::Problem::Get( instanceName.c_str() );

    // Setup problem for simulation
    SPARK::Problem::Initialize( P, RuntimeControls );
    SPARK::Problem::LoadPreferenceSettings( P, PreferenceSettings );

    // Perform simulation
    SPARK::TProblem::SimulationFlags SimulationFlag = SPARK::Problem::Simulate(
        P,
        SPARK::Problem::TRestartFlag(),
        SPARK::Problem::TStopTime(),
        SPARK::Problem::TTimeStep()
    );

    if ( SimulationFlag != SPARK::TProblem::SimulationFlag_OK ) {
        ostringstream ErrMsg;
        ErrMsg << "Cannot recover. Abort simulation." << ends;

        SPARK::ExitWithError(
            SPARK::ExitCode_ERROR_NUMERICAL,
            "Generic Problem Driver",
            ErrMsg.str(),
            P
        );
    }

    // Cleanup problem object and write statistics out.
    SPARK::Problem::Terminate( P );

}
///////////////////////////////////////////////////////////////////////////////////////////


///////////////////////////////////////////////////////////////////////////////////////////
//
// Entry point for the SPARK simulation program
//
int main(int argc, char *argv[ ])
{
    try {

        ///////////////////////////////////////////////////////////////////////////////
        // Start simulation environment and open global log files
        SPARK::Start(
            argv[0],        // session name
            "run.log",      // run log file name
            "error.log",    // error log filename
            "debug.log"     // debug log filename (only used with debug libsolver)
        );


        ///////////////////////////////////////////////////////////////////////////////
        // Load and solve problem 1 "spring"
        // We assume that the needed files are located in the current working directory
        RunProblem(
            "MySpring1",
            "spring.run",
            "spring.prf",
            "spring.xml"
        );
```

```
        //////////////////////////////////////////////////////////////////////////////
        // Re-load and re-solve problem 1 "spring"
        // We assume that the needed files are located in the current working directory
        RunProblem(
            "MySpring2",
            "spring.run",
            "spring.prf",
            "spring.xml"
        );


        //////////////////////////////////////////////////////////////////////////////
        // Load and solve problem 2 "room_fc"
        // We assume that the needed files are located in the current working directory
        RunProblem(
            "MyRoom_fc",
            "room_fc.run",
            "room_fc.prf",
            "room_fc.xml"
        );


        //////////////////////////////////////////////////////////////////////////////
        // Close global log files and perform garbage collection
        SPARK::End();

    } // try {}


    //////////////////////////////////////////////////////////////////////////////////////
    // Catch unprocessed exceptions
    catch (const SPARK::XAssertion& x1) { // SPARK specific exceptions
        SPARK::ExitWithError(
            x1.code(),
            __FILE__,
            x1.what()
        );
    }
    catch (const std::exception& x2) { // std exceptions
        SPARK::ExitWithError(
            SPARK::ExitCode_ERROR_RUNTIME_ERROR,
            __FILE__,
            x2.what()
        );
    }

    return SPARK::ExitCode_OK;
}
//////////////////////////////////////////////////////////////////////////////////////////
```

5.2 sparksolver.cpp

The following code snippet is an example of a driver function for a single problem. It is inspired from the sparksolver.cpp file that comes with the SPARK distribution and that is used by default when building a simulator unless a customized driver function is specified by the user.

The code snippet shows:

- The variable declarations; and
- The sequence of operations required to solve a single SPARK problem.

The sequence of operations required to load a problem changes depending on whether you are loading a statically-built problem or you are loading a problem at runtime from its XML description. The preprocessor macro `SPARK_STATIC_BUILD` controls which branch is executed at runtime. For example compiling the driver function with the preprocessor macro `SPARK_STATIC_BUILD` defined produces a SPARK simulator that will rely on the static build loading scheme.

```
//////////////////////////////////////////////////////////////////////////////////////////
// sparksolver.cpp
//
// Implementation of the default SPARK driver function. This is the entry point to the
// simulator program. It supports both statically-built problems and problems loaded at
// runtime through the definition of the preprocessor macro SPARK_STATIC_BUILD when
// compiling this file.
//
//////////////////////////////////////////////////////////////////////////////////////////
//
// Author  Dimitri Curtil (LBNL/SRG)
// Date    May 21, 2002
//
//////////////////////////////////////////////////////////////////////////////////////////
//
// VisualSPARK version 2.0
//
// Copyright (c) 1999-2003, The Regents of the University of California,
// through the Lawrence Berkeley National Laboratory (subject to receipt of
// any required approvals from the U.S. Department of Energy). All rights
// reserved.
//
// Portions of VisualSPARK version 2.0  were authored by Ayres Sowell
// Associates and are protected by copyright and international treaties.
//
// U.S. GOVERNMENT RIGHTS
// Portions of VisualSPARK version 2.0 (the Software) was developed under
// funding from the U.S. Department of Energy and the U.S. Government
// consequently retains certain rights as follows: the U.S. Government has
// been granted for itself and others acting on its behalf a paid-up,
// nonexclusive, irrevocable, worldwide license in the Software to reproduce,
// prepare derivative works, and perform publicly and display publicly.
// Beginning five (5) years after the date permission to assert copyright is
// obtained from the U.S. Department of Energy, and subject to any subsequent
// five (5) year renewals, the U.S. Government is granted for itself and
// others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide
// license in the Software to reproduce, prepare derivative works, distribute
// copies to the public, perform publicly and display publicly, and to permit
// others to do so.
//
// THIS SOFTWARE IS SUPPLIED "AS IS" WITHOUT WARRANTY OF ANY KIND.  LAWRENCE
// BERKELEY NATIONAL LABORATORY, ITS LICENSORS, THE UNITED STATES, THE UNITED
// STATES DEPARTMENT OF ENERGY, AYRES SOWELL ASSOCIATES, AND THEIR EMPLOYEES:
// (1) DISCLAIM ANY WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
// TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
// TITLE OR NON-INFRINGEMENT, (2) DO NOT ASSUME ANY LEGAL LIABILITY OR
// RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF THE SOFTWARE,
// (3) DO NOT REPRESENT THAT USE OF THE SOFTWARE WOULD NOT INFRINGE PRIVATELY
// OWNED RIGHTS, (4) DO NOT WARRANT THAT THE SOFTWARE WILL FUNCTION UNINTERRUPTED,
```

```
// THAT IT IS ERROR-FREE OR THAT ANY ERRORS WILL BE CORRECTED.
//
///////////////////////////////////////////////////////////////////////////////////////////
// PORTIONS COPYRIGHT (C) 2003 AYRES SOWELL ASSOCIATES, INC.
// PORTIONS COPYRIGHT (C) 2003 THE REGENTS OF THE UNIVERSITY OF CALIFORNIA .
//   PENDING APPROVAL BY THE US DEPARTMENT OF ENERGY. ALL RIGHTS RESERVED.
//
///////////////////////////////////////////////////////////////////////////////////////////



// The debugger can't handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// When symbols are longer than 255 characters, the warning is disabled.
#pragma warning(disable:4786)


#ifdef MacVersion
#       include <console>
#endif

#include <sstream>
using std::ostringstream;
using std::ends;

#include <string>
using std::string;


///////////////////////////////////////////////////////////////////////////////////////////
//
// Required header files
//
#include "sparkapi.h"    // for various API functions needed to manage the SPARK problem
#include "problem.h"     // for SPARK::TProblem definition
#include "ctrls.h"       // for SPARK::TRuntimeControls definition
#include "prefs.h"       // for SPARK::TPreferenceSettings definition
#include "exitcode.h"    // for SPARK::ExitCodes and SPARK::ExitWithError()
#include "exceptions.h"  // for SPARK exception classes
///////////////////////////////////////////////////////////////////////////////////////////

#include "spark.h"


///////////////////////////////////////////////////////////////////////////////////////////
//
// Entry point for the SPARK simulation program
//
int main(unsigned argc, char** argv)
{
// To support Mac platform
#ifdef MacVersion
    argc = ccommand(&argv);
#endif

    SPARK::TProblem* P = 0;
    char* RunFileName = 0;
    char* PrfFileName = 0;
    char* XmlFileName = 0;

    try {

        /////////////////////////////////////////////////////////////////////////////
        // Start simulation environment and open global log files
        /////////////////////////////////////////////////////////////////////////////
        SPARK::Start(
            argv[0],        // program name serves as session name
            "run.log",      // run log file name
            "error.log",    // error log file name
            "debug.log"     // debug log file name (only used with debug libsolver)
```

```
        );


        ///////////////////////////////////////////////////////////////////////////
        // Load the problem under study
        ///////////////////////////////////////////////////////////////////////////
        //
        // Note that if the driver is relying on a statically-built problem, then
        // there is no need to specify the *.xml file name at the command-line, hence
        // the two different calls controlled with the macro preprocessor
        // SPARK_STATIC_BUILD.
        //
        // WARNING: Never explicitly delete the problem object because the garbage
        //          collection is performed internally upon calling the function
        //          SPARK::End().
        //
        ///////////////////////////////////////////////////////////////////////////

#if defined(SPARK_STATIC_BUILD)
        ///////////////////////////////////////////////////////////////////////////
        // Process the command line arguments and retrieve :
        //    - the name of the "*.run" file
        //    - the name of the "*.prf" file
        //
        // Usage at the command line :
        //      ProgramName prefFile.prf runFile.run  <enter>
        // or
        //      ProgramName runFile.run  prefFile.prf <enter>
        // ...
        //
        ///////////////////////////////////////////////////////////////////////////
        // NOTE: This function will throw an exception if any of the required file names
        //       cannot be found in argv[]
        //
        SPARK::Problem::StaticBuild::ParseCommandLine(
            argc-1,       // number of entries in argv[]
            argv+1,       // we skip first entry which is full program name with path
            RunFileName,  // points to full name of "*.run" file in argv[]
            PrfFileName   // points to full name of "*.prf" file in argv[]
        );


        ///////////////////////////////////////////////////////////////////////////
        // Load the statically-built problem named after ProblemName
        //
        // NOTE: We use the session name by default as unique problem name.
        const string ProblemName( SPARK::GetBaseName() );

        if ( !SPARK::Problem::StaticBuild::Load(ProblemName.c_str()) ) {
            ostringstream ErrMsg;

            ErrMsg << "Could not load the statically-built problem \"" << ProblemName.c_str() << "\"." << ends;

            SPARK::ExitWithError(
                SPARK::ExitCode_ERROR_INVALID_PROBLEM,
                __FILE__,
                ErrMsg.str()
            );
        }

#else
        ///////////////////////////////////////////////////////////////////////////
        // Process the command line arguments and retrieve :
        //    - the name of the "*.run" file
        //    - the name of the "*.prf" file
        //    - the name of the "*.xml" file
        //
        // Usage at the command line :
        //      ProgramName prefFile.prf runFile.run  xmlFile.xml  <enter>
        // or
```

```
        //      ProgramName runFile.run  prefFile.prf xmlFile.xml  <enter>
        // or
        //      ProgramName prefFile.prf xmlFile.xml  runFile.run  <enter>
        // ...
        //
        ///////////////////////////////////////////////////////////////////////////
        // NOTE: This function will throw an exception if any of the required file names
        //       cannot be found in argv[]
        //
        SPARK::Problem::DynamicBuild::ParseCommandLine(
            argc-1,       // number of entries in argv[]
            argv+1,       // we skip first entry which is program name with path
            RunFileName,  // points to full name of "*.run" file in argv[]
            PrfFileName,  // points to full name of "*.prf" file in argv[]
            XmlFileName   // points to full name of "*.xml" file in argv[]
        );


        ///////////////////////////////////////////////////////////////////////////
        // Derive "unique" problem name from name of the XML file (without extension)
        string ProblemName;

        string Temp( XmlFileName ); // XmlFileName is a correct file name with XML extension
        string::size_type pos = Temp.find( "." );
        if ( pos != string::npos ) {
            ProblemName = Temp.substr(0, pos); // Truncate XML file name to form problem name
        }
        else {
            ProblemName = "My Problem"; // Default problem name
        }


        ///////////////////////////////////////////////////////////////////////////
        // Load the problem named ProblemName from the problem description defined in XML file
        //
        //      ProblemName :  unique problem name, we use the base name of the XML file by default!
        //      XmlFileName :  name of the XML file containing the problem description

        if ( !SPARK::Problem::DynamicBuild::Load( ProblemName.c_str(), XmlFileName ) ) {
            ostringstream ErrMsg;

            ErrMsg << "Could not load the problem \"" << ProblemName.c_str() << "\" at runtime from "
                   << "the XML description in file \"" << XmlFileName << "\"" << ends;

            SPARK::ExitWithError(
                SPARK::ExitCode_ERROR_INVALID_PROBLEM,
                __FILE__,
                ErrMsg.str()
            );
        }

#endif // defined(SPARK_STATIC_BUILD)


        ///////////////////////////////////////////////////////////////////////////
        // Get address fo SPARK::TProblem instance
        //
        P = SPARK::Problem::Get( ProblemName.c_str() );


        ///////////////////////////////////////////////////////////////////////////
        // Initialize problem with run-time controls
        //
        SPARK::TRuntimeControls RuntimeControls(
            "Default SPARK driver",  // name of controls set
            RunFileName              // *.run file name
        );
        P->Initialize( RuntimeControls );
```

```
        ///////////////////////////////////////////////////////////////////////////
        // Load preference settings
        //
        SPARK::TPreferenceSettings PreferenceSettings(  PrfFileName );
        P->LoadPreferenceSettings( PreferenceSettings );


        ///////////////////////////////////////////////////////////////////////////
        // Perform simulation
        //
        if ( P->Simulate() != SPARK::TProblem::SimulationFlag_OK ) {
            SPARK::ExitWithError(
                SPARK::ExitCode_ERROR_NUMERICAL,
                __FILE__,
                "Abort simulation.",
                P // Specifiy active problem to ensure proper diagnostic
            );
        }

        ///////////////////////////////////////////////////////////////////////////
        // Cleanup problem object and write statistics out.
        P->Terminate();


        ///////////////////////////////////////////////////////////////////////////
        // Close global log files and perform automatic garbage collection for all
        // problem instances
        ///////////////////////////////////////////////////////////////////////////
        SPARK::End();

    }


    /////////////////////////////////////////////////////////////////////////////////////
    // Catch unprocessed exceptions
    catch (const SPARK::XAssertion& x1) { // SPARK specific exceptions
        SPARK::ExitWithError(
            x1.code(),
            __FILE__,
            x1.message(),
            P // Specifiy active problem to ensure proper diagnostic
        );
    }
    catch (const std::exception& x2) { // std exceptions
        SPARK::ExitWithError(
            SPARK::ExitCode_ERROR_RUNTIME_ERROR,
            __FILE__,
            x2.what(),
            P // Specifiy active problem to ensure proper diagnostic
        );
    }

    return SPARK::ExitCode_OK;
}
/////////////////////////////////////////////////////////////////////////////////////////
```

Chapter 6

SPARK Build Process and Problem Driver API Page Documentation

6.1 Description of the SPARK Build Process

This section provides information on the SPARK build process, whereby the problem and class source files are converted to executable simulation programs.

- Generate the solution sequence for a SPARK problem
- Build a SPARK problem statically
- Build a SPARK problem dynamically
- Automate the build process
- Build a SPARK problem simulator with a customized driver

For advanced usage of SPARK, it is recommended to use SPARK at the command line instead of using the GUI front end. This gives you more control over the build operations, in particular the compilation and linkage steps.

6.2 Generate the solution sequence for a SPARK problem

Rather than solving a problem directly, SPARK builds a program that carries out the solution. This approach is taken in an effort to maximize solution efficiency. The overall process of generating the solution sequence for a SPARK problem expressed using the SPARK language is discussed in this section. The process of building a SPARK simulator for the generated solution sequence and of solving it is discussed in the following sections.

Several programs of the SPARK distribution are used in this process:

- the parser program, and
- the setupcpp program. The principal output of the build process are the problem.xml and problem.cpp files that describe the problem topology and the solution sequence to be used by the SPARK solver.

6.2.1 Parser step

As the problem.pr file is parsed, various declarations of atomic (∗.cc) classes and macro classes (∗.cm) are found. The specified class paths are searched for these classes. If a macro class is found, it is parsed, and if the macro refers to other macros, they are parsed. This continues recursively to any depth.

The result is to resolve all macros to atomic objects so that, at run time, SPARK can operate entirely on atomic classes. This exposes individual equations and variables to the graph theoretic algorithms for organizing an efficient solution sequence. The principal output of the parser is the setup file, problem.stp. This file is a different expression of the problem, with the principle difference being that the macros have been resolved.

The parser command line is:

```
parser  [options]  filename  <enter>
```

where filename gives the name of the file to be parsed, which can be either a problem, problem.pr or a class class.cm.

Parser options can include:

```
-p class_path
```

class_path is a list of directories separated by commas, e.g., c:\vspark200\hvactk, c:\vspark200\myClasses These directories are searched in the given order to find classes needed by the problem. If omitted, the environment variable `SPARK_CLASSPATH` will be used in the same manner.

```
-e error_level
```

Sets level at which parser will report errors. E.g., e1 will result in report of errors only. Higher numbers give increased verbosity: 1 = errors 2 = warnings 3 = information 4 = more information. The default is -e2.

```
-s setup_file_verbosity
```

Sets level of verbosity in the created setup file. Lowest is 1, with increasing verbosity up to 6. The default is -s2.

```
-l logFileName
```

If -l logFileName is specified, the diagnostic messages are sent to logFileName. If -l is given without logFileName, the messages are sent to the file parser.log.

```
-d debug_dump
```

Extensive debugging output will be generated.

6.2.2 Setup step

The next step is carried out by the setupcpp program. The main function of setupcpp is to perform the graph theoretic analysis of the problem, producing a compact, efficient solution procedure expressed either in C++ or in XML format. The C++ representation is emitted in the problem.cpp file whereas the XML representation is emitted in the problem.xml file. Depending on the problem build process selected at runtime either file will be needed by the simulator.

In addition to the problem.cpp and problem.xml files, setupcpp produces two other files. One is essential to final problem execution, namely problem.prf. This file serves to transmit needed information from the setup process to the solution process. Default solution settings can be specified in the default.prf file in the current working directory. If this file is not found, then setupcpp uses hard-coded default solution settings to populate the problem.prf file. The other produced file is problem.eqs. This file is a human readable expression of the solution sequence. It is not used by SPARK, but is sometimes helpful in debugging when numerical difficulties are encountered.

The setupcpp command line is:

```
setupcpp problem <enter>
```

The setupcpp program loads the setup file named problem.stp and performs the graph theoretic analysis to produce the problem.cpp and problem.xml files containing the solution sequence.

See Document Type Definition file for more information on the set of grammar rules used to generate the XML problem description.

6.2.3 Build step

The next step in the process is to build a SPARK simulator for the problem description obtained after the setup step. A simulator consists of a driver function (i.e., the main program or entry point) and of the solver fixed library, referred to as libsolver.

The default driver in the SPARK distribution solves a single problem using:

- the run controls specified in the ∗.run file, and
- the solution settings specified in the ∗.prf file.

There are two ways of loading the problem description for the driver function to solve the set of equations:

- Build a SPARK problem statically
- Build a SPARK problem dynamically

The static build approach consists in compiling and linking the problem.cpp file with the solver library and the driver function in order to produce a self-contained executable simulator. The atomic classes that appear in the solution sequence also need to be compiled and linked along. This approach is referred to as static build because the solution sequence is "constructed" during the compilation and linkage steps before runtime.

The dynamic build approach consists in loading the problem description at runtime from the problem.xml file. Then, the C++ funtions defined in the atomic classes appearing in the solution sequence are retrieved by solver using explicit linking of the dynamic libraries. This approach is referred to as dynamic build because the solution sequence is "constructed" dynamically at runtime.

6.3 Build a SPARK problem statically

The following diagram shows the static build process resulting in the executable simulator problem.exe. Red font is used to indicate systems program like the C++ compiler and linker. Blue font is used to indicate the SPARK programs and files involved in the build process. The bold arrows indicate file dependencies at runtime, whereas the normal arrows indicate build operations performed before runtime.

- The file libsolver.a is the static library for the fixed part of the SPARK solver.
- The problem.cpp and problem.prf files are generated by the setup step (See Setup step).

This mode of operation is accomplished by executing the SPARK makefile with the following flag at the command-line:

```
gmake SPARK_STATIC_BUILD=yes <some-target> <enter>
```

Note:

- On UNIX/Linux platforms, the resulting executable file is named problem without extension.
- On Windows platform, the static library for the fixed part of the solver is called libsolver.lib if you are using the Microsoft VC++ compiler in place of the mingw compiler.

6.3.1 Execute the simulator

At runtime, the only files that are needed to run a statically-built problem with the default driver function are:

- the executable simulator, problem.exe
- the preference file, problem.prf
- the run control file, problem.run.

To simulate a statically-built problem, type at the command line:

```
problem problem.run problem.prf <enter>
```

Once the simulator is built, the C++ compiler and the linker are no longer needed. All needed atomic classes and the solver library are statically linked along to produce the executable simulator, thus producing a self-contained simulator program that can be shared with and reused by different users on the same platform.

Note:

It is also possible to build dynamic SPARK problems that are built at runtime from the problem.xml file and the dynamic libraries containing the compiled atomic classes (See Build a SPARK problem dynamically).

6.4 Build a SPARK problem dynamically

The following diagram shows the operations involved in the dynamic build process where the problem description contained in the problem.xml file is loaded at runtime. The normal arrows indicate operations occurring before runtime whereas the bold arrows indicate the file dependencies at runtime when executing the simulator. Red font is used with system programs like the C++ compiler and linker. Blue font is used to indicate the SPARK programs and files used in the build process.

This is the default mode of operation accomplished when executing the SPARK makefile.

Note:

- The dynamic solver library is called libsolver.so on UNIX/Linux platforms.
- The dynamic libraries containing the compiled atomic classes have the extension ∗.so on the UNIX/Linux platforms.
- The executable simulator is called sparksolver on the UNIX/Linux platforms.

6.4.1 Locate the dynamic libraries with the compiled atomic classes

Along with the problem.xml file, the SPARK solver needs to be able to access at runtime the dynamic libraries class0.dll, class1.dll, ... where the various callbacks in each atomic class are compiled.

The paths to the dynamic libraries are specified in the problem.stp file generated by the parser program. They are relative paths with respect to the current working directory where the parser program is invoked.

At runtime, the dynamic libraries are located by the problem loader in solver by using the respective relative paths prefixed by either one of the following paths:

- the parser's working directory, or
- any directory that is listed in the environment variable `SPARK_LIBPATH`, or
- any directory that is listed in the environment variable `SPARK_CLASSPATH`.

6.4.1.1 The SPARK_LIBPATH environment variable

The environment variable `SPARK_LIBPATH` contains a series of paths delimited with commas. E.g.,

```
SPARK_LIBPATH="c:\users\spark_projects\spring,c:\users\spark_projects\room_fc"
```

If an entry in `SPARK_LIBPATH` indicates a relative path, then it is expanded to an absolute path by prefixing it with the current working directory of the simulator.

6.4.1.2 The SPARK_CLASSPATH environment variable

The environment variable `SPARK_CLASSPATH` contains a series of paths delimited with commas. E.g.,

```
SPARK_CLASSPATH="c:\users\spark_projects\spring,c:\users\spark_projects\room_fc"
```

If an entry in `SPARK_CLASSPATH` indicates a relative path, then it is expanded to an absolute path by prefixing it with the current directory where the parser program was called.

6.4.2 Execute the simulator

A dynamically-built problem is solved using the executable driver sparksolver.exe. The program sparksolver.exe is obtained by compiling the default driver function. It is relying on the dynamic solver library libsolver.dll which must be located in the system

path. The SPARK distribution installs the dynamic solver library in the bin/ subdirectory and adds the path this directory to the system path.

At runtime the following files are needed to make a simulation run:

- the problem description file, problem.xml
- the dynamic libraries containing the compiled atomic classes
- the preference file, problem.prf
- the run control file, problem.run.

The behavior of the default SPARK driver function assumes that only one problem is solved at each call of the simulator executable. The names of the needed files are passed to the simulator at the command-line in any order. Each file type is identified by parsing its extension. Remember that the names of the dynamic libraries containing the compiled atomic classes are obtained from the problem.xml file and located at runtime using the previously-described loading scheme (See Locate the dynamic libraries with the compiled atomic classes).

To simulate a dynamically-built problem with the default driver function, type at the command line:

```
sparksolver problem.run problem.prf problem.xml <enter>
```

Note:

It is also possible to build static SPARK problems that are essentially pre-compiled and do not need to be loaded at runtime (See Build a SPARK problem statically).

6.5 Automate the build process

The SPARK build process described above is rather complex. As has been noted, the SPARK objects used in the problem are not known until the problem.pr file has been parsed. Moreover, the particular inverse functions from these objects that will actually be used in the numerical solution process are not known until setupcpp has completed. Thus the files to be compiled and linked are not knowable beforehand. Finally, when you make a change to the problem.pr file or any used class files, the build process should remake only the affected files.

To simplify matters, the process of building a SPARK solver is automated using facilities of the GNU make program, gmake, and a makefile called makefile.prj, found in the SPARK `lib/` directory.

From the usage standpoint, gmake is straightforward. It will build an executable program from the specified problem.pr source file and then execute it with the command:

```
gmake run <enter>
```

This is the most common usage of gmake in the SPARK context. However, there are other command line targets to meet special needs.

```
run
```

To make a run using problem.inp as input and problem.run as run control file. If problem.run is missing, a default one is created.

```
(none)
```

After the first time, rebuilds makefile.inc and the solver executable as needed using the previous `SPARK_CLASSPATH`.

```
PROJ=xxxx  SPARK_CLASSPATH=../class
```

To create the file makefile.inc and the solver executable that has the same name as `xxxx`.

```
SPARK_STATIC_BUILD=yes
```

To make the solver executable problem.exe for a statically-built problem.

```
SPARK_DEBUG=yes
```

To make the solver executable that uses the DEBUG version of the solver library. In DEBUG mode, the SPARK solver generates a detailed log file called debug.log that describes the simulation process in great detail.

```
solver
```

To make the solver executable

```
stp
```

To make problem.stp file.

```
makefile.inc
```

To create makefile.inc file.

```
pkg
```

To create an export package in ../problem_pkg directory where problem is the name of the problem under study.

```
prf
```

To make problem.prf file.

```
clean
```

To delete all intermediate files.

```
cleanALL
```

Same as clean, but also deletes all run subdirectories.

```
help
```

To show the help information about the list of targets.

See Makefile for more information.

6.6 Build a SPARK problem simulator with a customized driver

The following diagram shows the operations involved in building a multi-problem simulator with a customized driver function implemented in the file mydriver.cpp. The resulting simulator mydriver.exe solves two different problems, described in the problem1.pr and problem2.pr files, respectively. To make matters more interesting in this example, the problem 1 is statically built whereas the problem 2 is dynamically built. Finally we note that the executable simulator relies on the dynamic solver library libsolver.dll at runtime.

The normal arrows indicate operations occurring before runtime whereas the bold arrows indicate the file dependencies at runtime when executing the simulator. Red font is used with system programs like the C++ compiler and linker. Blue font indicates the SPARK programs and files used in the build process.

Note:

- The dynamic solver library is called libsolver.so on UNIX/Linux platforms.
- The dynamic libraries containing the compiled atomic classes have the extension ∗.so on the UNIX/Linux platforms.
- The resulting executable simulator is called mydriver on the UNIX/Linux platforms.

6.6.1 Define a customized driver function

The overall build process essentially combines the static and dynamic build processes explained previously in Build a SPARK problem statically and Build a SPARK problem dynamically.

What's new concerns the process of explicitly specifying the driver function to be used by the executable simulator. First, the C++ source file mydriver.cpp where the driver function is implemented is compiled to produce the object file mydriver.o. Then the object file is linked along with the other object files for the atomic classes to generate the executable simulator mydriver.exe

To solve multiple problems in the same driver function, each problem topology must be described through a corresponding ∗.cpp file or ∗.xml file, depending on how the problem is to be built. There are no limitations as to how many different problem topologies (each defined in a different ∗.cpp file or ∗.xml file) can be used in the same driver function.

Of course, it is possible to instantiate multiple instances of the same problem topology (e.g., defined in the spring.xml file) as long as each instance is assigned a unique name in the process space (e.g., one instance could be named "MySpring1" and the other instance "MySpring2").

Also, for each problem instance being simulated, corresponding ∗.run and ∗.prf files must be specified.

See Implementation of the SPARK Driver Function for more infomation on how to implement the driver function for a SPARK simulator using the problem driver API.

6.6.2 Automate the build process

The multiproblem.makefile automates this process of building a SPARK simulator involving multiple problems and a customized driver file. For example, in order to build the SPARK simulator from the driver function implemented in the file mydriver.cpp and involving the two problems named problem1.pr and problem2.pr, defined respectively in the directories ../problem1 and ../problem2, type at the command line:

```
gmake -f multiproblem.makefile PROJ=mydriver SUBPROBLEMS="../problem1 ../problem2" <enter>
```

This will generate an executable simulator mydriver.exe as well as the needed dynamic libraries containing the compiled atomic classes. The makefile assumes that the problem files are located in directories with the same names.

See Multi-problem command file for a more detailed explanation of the multiproblem makefile.

6.6.3 Execute the simulator

The command line operation of the resulting simulator depends on the way main function and the customized driver function have been implemented in the source file mydriver.cpp.

6.7 Implementation of the SPARK Driver Function

This section provides information on how to implement the SPARK driver function. The code snippets are inspired from the default driver function implemented in the file sparksolver.cpp that comes with the SPARK distribution (See examples section). The default driver solves a single SPARK problem with the preference settings specified in the preference file (∗.prf), and for the run control parameters specified in the run controls file (∗.run).

See Description of the SPARK Build Process for more details on how to build a SPARK simulator.

- Header files to include in the C++ file implementing the driver function
- Start the simulation session
- Load a SPARK problem at runtime
- Load a statically-built SPARK problem
- Set up the runtime controls
- Set up the preference settings
- Solve the SPARK problem
- Re-solve the SPARK problem with different runtime controls
- End the simulation session
- Catch the runtime exceptions

6.8 Header files to include in the C++ file implementing the driver function

The following header files need to be included in the file where you implement the SPARK driver function:

- sparkapi.h for the various API driver functions needed to manage a SPARK problem;
- problem.h for the definition of the SPARK::TProblem class;
- ctrls.h for the definition of the SPARK::TRuntimeControls class;
- prefs.h for the definition of the SPARK::TGlobalSettings, SPARK::TComponentSettings and SPARK::TPreferenceSettings classes;
- exceptions.h for the various SPARK exceptions classes derived from the class SPARK::XAssertion;
- exitcode.h for the enum SPARK::ExitCodes.

If you need to interact with the SPARK::TInverse and SPARK::TObject instances comprised in each SPARK::TProblem instance, then you will also need to include the following header files:

- inverse.h for the definition of the SPARK::TInverse type;
- object.h for the definition of the SPARK::TObject type.

6.9 Start the simulation session

To set up the SPARK simulation session, the function SPARK::Start() must be invoked prior to any other SPARK method or function in the driver function.

This function identifies the simulation session with a name specified as a `const char*` string argument. It also opens the global run and error log files.

```
SPARK::Start(
       "My session", // session name
       "run.log",    // name of run log file
       "error.log",  // name of error log file
   "debug.log"   // name of the debug log file (used only if SPARK_DEBUG is defined)
);
```

Note:

The error log file is only created if warning(s) and/or error(s) are generated during the simulation run.

In order to make a simulation run, it is first required to load the problem under study. See Load a SPARK problem at runtime.

6.10 Load a SPARK problem at runtime

This section shows how to load a SPARK problem at runtime from the problem description defined in the ∗.xml file. See Build a SPARK problem dynamically.

The instance of the class SPARK::TProblem that internally describes the problem under study is actually allocated at runtime from the topology information contained in the ∗.xml file.

This loading scheme is also referred to as **dynamic build**. It is the default loading scheme used in the SPARK stand-alone driver. The API functions specific to this loading approach can be found in the namespace SPARK::Problem::DynamicBuild.

Note:

Another loading approach consists in compiling the C++ file that defines the data structures representing the topology of the problem and then linking the resulting object file(s) along with the driver's object file to produce the stand-alone executable. This loading approach is known as **static build**. Problems built this way are loaded in a different manner. See Load a statically-built SPARK problem.

6.10.1 Retrieve the required command-line arguments

The utility function SPARK::Problem::DynamicBuild::ParseCommandLine() can be used to process the command-line arguments in order to retrieve:

- the name of the ∗.run file,
- the name of the ∗.prf file, and
- the name of the ∗.xml file.

For a program called `sparksolver` (which would be called `sparksolver.exe` on a Windows platform), the SPARK usage at the command line with the default driver program is:

```
sparksolver prefFile.prf runFile.run xmlFile.xml  <enter>
```

The API function SPARK::Problem::DynamicBuild::ParseCommandLine() identifies each file name by parsing its extension and returns the name of each file matching the desired extension in the pointers to `char` passed as arguments.

```
char* RunFileName;
char* PrfFileName;
char* XmlFileName;

SPARK::Problem::DynamicBuild::ParseCommandLine(
       argc-1,         // number of entries in argv
       argv+1,         // we skip the first entry which is the full program name
       RunFileName,    // returns full name of *.run file
       PrfFileName,    // returns full name of *.prf file
       XmlFileName     // returns full name of *.xml file
);
```

If any of the three file names with the desired extensions cannot be detected in argv[], the function throws a SPARK::XInitialization exception.

If you write your own customized driver function, you could either specify these files at the command-line (like we do with the default driver function) or have these names hard-coded in the body of the function.

See the example driver function `void main()` implemented in the file multiproblem_example1.cpp in the Examples section.

6.10.2 Load the problem at runtime

A new SPARK problem object can be constructed using the API function SPARK::Problem::DynamicBuild::Load(). Each instantiated problem must also be assigned a unique, case-sensitive string identifier. Many instances can be constructed from the same description as long as they are each given a different and unique name.

If the API function fails to load the problem, then it returns false. It is good practice to test whether the loading operation failed or not before proceeding. In the next example, we force to exit the simulation if this is the case.

```
const string XmlFileName("problem.xml");             // name of the *.xml file with the problem description
const string ProblemName("My unique problem name");  // unique problem name

if ( !SPARK::Problem::DynamicBuild::Load(ProblemName.c_str(), XmlFileName.c_str()) ) {
       ostringstream ErrMsg;

   ErrMsg << "Could not load the problem \"" << ProblemName.c_str() << "\" at runtime from "
          << "the XML description in file \"" << XmlFileName << "\"" << ends;

   SPARK::ExitWithError(
      SPARK::ExitCode_ERROR_INVALID_PROBLEM,
      __FILE__,
      ErrMsg.str()
   );
}
```

One reason why the loading operation might fail is that the required dynamic libraries implementing the atomic classes used in the problem under study cannot be located at runtime. See Build a SPARK problem dynamically.

6.11 Load a statically-built SPARK problem

Another loading approach consists in compiling the C++ file that defines the data structures representing the topology of the problem and then linking the resulting object file(s) along with the driver's object file to produce a stand-alone executable. Problems built this way are referred to as **statically built** problems. See Build a SPARK problem statically.

A statically-built problem is loaded in a different manner than a problem loaded at runtime (See Load a SPARK problem at runtime). This section shows how to load a statically-built SPARK problem. The API functions specific to this loading approach can be found in the namespace SPARK::Problem::StaticBuild.

6.11.1 Retrieve the required command-line arguments

The utility function SPARK::Problem::StaticBuild::ParseCommandLine() can be used to process the command-line arguments in order to retrieve:

- the name of the *.run file, and
- the name of the *.prf file.

For a program called `simulator` (which would be called `simulator.exe` on a Windows platform), the SPARK usage at the command line with the default driver program is:

```
simulator prefFile.prf runFile.run  <enter>
```

The API function SPARK::Problem::StaticBuild::ParseCommandLine() identifies each file name by parsing its extension and returns the name of each file matching the desired extension in the pointers to `char` passed as arguments.

```
char* RunFileName;
char* PrfFileName;

SPARK::Problem::StaticBuild::ParseCommandLine(
       argc-1,         // number of entries in argv
       argv+1,         // we skip the first entry which is the full program name
       RunFileName,    // returns full name of *.run file
       PrfFileName     // returns full name of *.prf file
);
```

If any of the two file names with the desired extensions cannot be detected in argv[], the function throws a SPARK::XInitialization exception.

If you write your own customized driver function, you could either specify these files at the command-line (like we do with the default driver function) or have these names hard-coded in the body of the function.

6.11.2 Load the statically-built problem

A previously-built SPARK problem object is registered using the API function SPARK::Problem::StaticBuild::Load(). Each problem instance must be assigned a unique, case-sensitive string identifier in the problem.cpp file that is compiled and linked to the executable simulator. Note that loading a statically-built problem at runtime does not actually instantiate any data structures as the SPARK::TProblem instance is instantiated from the declarations contained in the compiled problem.cpp file. This is the main difference with the other loading approach called a dynamic build (See Load a SPARK problem at runtime).

To declare many instances of the same problem description you must provide a separate problem.cpp file that defines a SPARK::TProblem instance with a different and unique name. Each problem.cpp file must then be compiled and linked along with the other object files for the required atomic classes to produce the self-contained executable simulator program.

If the API function fails to load the problem, then it returns false. It is good practice to test whether the loading operation failed or not before proceeding. In the next example, we force to exit the simulation if this is the case.

```
// Unique problem name as specified in the compiled problem.cpp file
const string ProblemName("My unique problem name");

if ( !SPARK::Problem::StaticBuild::Load(ProblemName.c_str()) ) {
       ostringstream ErrMsg;

       ErrMsg << "Could not load the statically-built problem \"" << ProblemName.c_str() << "\"" << ends;

       SPARK::ExitWithError(
          SPARK::ExitCode_ERROR_INVALID_PROBLEM,
          __FILE__,
          ErrMsg.str()
       );
}
```

In the stand-alone driver function in SPARK, it is assumed that the name of the problem to load is derived from the name of the simulator program. In the sparksolver.cpp file that implements the default SPARK driver, the program name `argv[0]` is used to identify the simulation session. Thus, calling the API function SPARK::GetBaseName() returns the name of the program that is also the name of the previously-built problem. This contorted approach for identifying the unique name of problem to solve is necessary in order to be able to use the same driver function for all statically-built problems. Also, it implies that the name of the simulator program must be the name of the statically-built problem you want to solve.

6.12 Set up the runtime controls

To set up the runtime controls, an instance of the class SPARK::TRuntimeControls should be instantiated with the name of the *.run file passed to the constructor along with a name for the set of controls (used mostly for debugging purposes in case you use multiple SPARK::TRuntimeControls objects in the same run).

This class parses the *.run text file and generates an internal representation of the runtime control parameters that can then be queried by the other SPARK classes.

```
SPARK::TRuntimeControls RuntimeControls(
       "Default SPARK driver",  // name of controls set
       RunFileName              // *.run file name
);
```

6.13 Set up the preference settings

To set up the preference settings, an instance of the class SPARK::TPreferenceSettings should be instantiated with the name of the ∗.prf file passed to the constructor.

This object parses the ∗.prf text file and generates an internal representation of the preference settings that can then be queried by the other SPARK classes. It is also possible to create a SPARK::TPreferenceSettings object without a ∗.prf file thanks to the method interface provided in the SPARK::TPreferenceSettings, SPARK::TComponentSettings and SPARK::TGlobalSettings classes.

In particular, the class SPARK::TPreferenceSettings will produce:

- an instance of the class SPARK::TGlobalSettings, and
- for each component comprised in the problem under study, an instance of the class SPARK::TComponentSettings.

```
SPARK::TPreferenceSettings PreferenceSettings( PrfFileName );
```

6.14 Solve the SPARK problem

To solve a SPARK problem, you need to perform the following steps:

- Retrieve the pointer to the SPARK::TProblem instance that was previously loaded;
- Initialize the problem with the previously constructed SPARK::TRuntimeControls object;
- Load the previously contructed SPARK::TPreferenceSettings object;
- Simulate the problem from `InitialTime` to `FinalTime` (as specified in the *.run file); and
- Terminate the simulation run of the problem.

6.14.1 Retrieve the address of the SPARK::TProblem instance

The address of the SPARK::TProblem instance of a previously loaded problem (See Load a SPARK problem at runtime) is returned by the the API function SPARK::Problem::Get() by passing the unique name of the problem instance as `const char*`.

If no problem instance by this name can be found in the instance repository, then the function returns a NULL pointer. This will happen:

- if you failed to first load the problem (using either loading approach described in Load a SPARK problem at runtime and in Load a statically-built SPARK problem), or
- if the loading operation failed (i.e., the function SPARK::Problem::DynamicBuild::Load() or the function SPARK::Problem::StaticBuild::Load() returned false).

```
SPARK::TProblem* P = SPARK::Problem::Get( ProblemName.c_str() );
```

Warning:

Never explicitly delete the SPARK::TProblem object returned by SPARK::Problem::Get() with the C++ operator delete because this might produce memory violation problems and memory leaks. The garbage collection is performed internally upon calling the API function SPARK::End(). This is the only way that you can avoid memory leaks by ensuring that all memory allocated from the heap at runtime for this simulation session will be properly freed.

6.14.2 Initialize the problem

You initialize the problem with a previously constructed SPARK::TRuntimeControls object (See Set up the runtime controls) by invoking the SPARK::TProblem::Initialize() method:

```
P->Initialize( RuntimeControls );
```

This method prepares the simulator for the problem under study:

- it allocates various handlers for the specified runtime controls,
- it loads the initial values for all variables, and
- it fires the atomic class constructors (if any) to allocate the memory needed by each inverse and object.

Before re-initializing the same SPARK::TProblem object with new runtime controls, i.e., a new SPARK::TRuntimeControls object, you first have to call the SPARK::TProblem::Terminate() method to release the memory allocated by the previous call to SPARK::TProblem::Initialize().

Note:

The SPARK::TRuntimeControls object is copied internally in the SPARK::TProblem object, so that it can be destroyed (or go out of scope) as soon as the SPARK::TProblem::Initialize() method returns.

6.14.3 Load the preference settings in the problem

The preference settings described by an instance of the class SPARK::TPreferenceSettings (See Set up the preference settings) are loaded in the problem with the SPARK::TProblem::LoadPreferenceSettings() method:

```
P->LoadPreferenceSettings( PreferenceSettings );
```

It should be noted that you can call the SPARK::TProblem::LoadPreferenceSettings() method without having to re-initialize the SPARK::TProblem object. This feature can be useful to recover from non-converging simulations by loading a new set of "more robust" solution settings through a call to SPARK::TProblem::LoadPreferenceSettings() in the catch statement.

Warning:
The SPARK::TPreferenceSettings object is only partially copied internally in the SPARK::TProblem object. In particular, the global settings stored in the SPARK::TPreferenceSettings object through an instance of the class SPARK::TGlobalSettings might be accessed during the course of the simulation. Therefore, it is required that the SPARK::TPreferenceSettings object exists in memory until the simulation is over.

6.14.4 Simulate the problem

Starting from `InitialTime`, the solution is computed by advancing the global time until reaching `FinalTime` as specified in the SPARK::TRuntimeControls object. At each time step, the values for the known variables are read from the input files, and the computed values are reported to the output files.

It is good practice to test the value returned by the SPARK::TProblem::Simulate() method against SPARK::TProblem::SimulatorState_OK to detect possible runtime errors.

In the following code snippet, we exit the simulator if the simulation encountered any problem.

```
if ( P->Simulate() != SPARK::TProblem::SimulatorState_OK ) {
       SPARK::ExitWithError(
          SPARK::ExitCode_ERROR_NUMERICAL,
          __FILE__,
          "Abort simulation.",
          P // Specifiy active problem to ensure proper diagnostic
       );
}
```

6.14.5 Terminate the problem simulator

After a call to the SPARK::TProblem::Simulate() method, you should invoke the SPARK::TProblem::Terminate() method to:

- Call the atomic class destructors (if any) to free the memory allocated in the constructors;
- Write the simulation statistics to the global run log file (if such diagnostic is required);
- Free the control-specific memory allocated at runtime following the call to SPARK::TProblem::Initialize(); and
- Reset the internal counters and handlers to allow for another run with a new set of runtime controls (i.e., a new instance of the class SPARK::TRuntimeControls).

```
P->Terminate();
```

6.15 Re-solve the SPARK problem with different runtime controls

After solving a problem with the runtime controls specified in the SPARK::TRuntimeControls object passed to SPARK::TProblem::Initialize(), it is possible to re-use the SPARK::TProblem instance to solve the same problem with a different set of runtime controls specified in a new SPARK::TRuntimeControls object.

For example, assuming that the names for the new $*$.run and $*$.prf files are specified in the variables `RunFileName2` and `RuntimeControls2`, the following code snippet shows how to re-solve the same problem object pointed to by P:

```
SPARK::TRuntimeControls RuntimeControls2(
       "Runtime controls #2",   // name of controls set
       RunFileName2             // *.run file name
);
SPARK::TPreferenceSettings PreferenceSettings2( PrfFileName2 );

P->Initialize( RuntimeControls2 );
P->LoadPreferenceSettings( PreferenceSettings2 );

if ( P->Simulate() != SPARK::TProblem::SimulatorState_OK ) {
       SPARK::ExitWithError(
          SPARK::ExitCode_ERROR_NUMERICAL,
          __FILE__,
          "Abort simulation.",
          P // Specifiy active problem to ensure proper diagnostic
       );
}

P->Terminate();
```

Warning:

Before intializing the problem with the new SPARK::TRuntimeControls object, the SPARK::TProblem::Terminate() method must be invoked to ensure proper destruction of all control-specific handlers. If you fail to do this, you subject yourself to possible memory leaks.

6.16 End the simulation session

To terminate the SPARK simulation session, we need to call the SPARK::End() function. This will close the log files and perform proper destruction of the data stuctures allocated at runtime by the various solvers and problem factories.

```
SPARK::End();
```

Warning:

If you skip this step, then you subject yourself to potential memory leaks.

6.17 Catch the runtime exceptions

SPARK objects such as the SPARK::TProblem class and the various API functions rely on the C++ exception mechanism to deal with errors occuring at runtime. Therefore, it is recommended that you execute any SPARK instructions within a `try { ... }` statement and that you provide the required `catch () { ... }` statements to process the exceptions.

There are essentially two types of exceptions that will be thrown by the SPARK solver:

- the exception classes derived from the SPARK::XAssertion class, and
- the exception classes derived from the C++ built-in std::exception class.

Note:

All SPARK specific exceptions are children classes of the class SPARK::XAssertion, whereas the exceptions derived from the class std::exception usually come from the C++ standard library and the Standard Template Library (STL).

The next code snippet shows the typical framework used to catch all exceptions arising during the course of the simulation.

```
SPARK::TProblem* P = 0;

try {
 ...
}

catch (const SPARK::XAssertion& x1) { // SPARK specific exceptions
    SPARK::ExitWithError(
            x1.code(),
            __FILE__,
            x1.message(),
            P // Specifiy active problem to ensure proper diagnostic
    );
}
catch (const std::exception& x2) { // std exceptions
    SPARK::ExitWithError(
            SPARK::ExitCode_ERROR_RUNTIME_ERROR,
            __FILE__,
            x2.what(),
            P // Specifiy active problem to ensure proper diagnostic
    );
}

return SPARK::ExitCode_OK;
```

Note that the `catch` statements in the default driver exit the simulator by invoking the API function SPARK::ExitWithError().

6.18 SPARK Problem Driver Application Programming Interface

This document provides detailed information on the classes and functions used in the implementation of the driver function. We refer to this set of preprocessor macros, classes, types and functions as the SPARK Problem Driver Application Programming Interface (API).

6.18.1 List of classes

- the SPARK::TProblem class
- the SPARK::TComponent class
- the SPARK::TInverse class
- the SPARK::TObject class
- the SPARK::TRuntimeControls class
- the SPARK::TGlobalSettings class
- the SPARK::TComponentSettings class
- the SPARK::TPreferenceSettings class

6.18.2 List of typedefs and enums

- the enum SPARK::ExitCodes returned by the driver function

6.18.3 List of functions

Functions used to manage the simulation session:

- SPARK::Start()
- SPARK::End()
- SPARK::ExitWithError()

Utility functions :

- SPARK::Log()
- SPARK::GetFileName()

General access functions:

- SPARK::GetProgramName()
- SPARK::GetBaseName()
- SPARK::GetVersion()
- SPARK::GetRunLog()
- SPARK::GetRunLogFilename()

- SPARK::GetErrorLog()
- SPARK::GetErrorLogFilename()

Functions used to manage the SPARK::TProblem objects:

- SPARK::Problem::StaticBuild::ParseCommandLine()
- SPARK::Problem::StaticBuild::ShowCommandLineUsage()
- SPARK::Problem::StaticBuild::Load()

- SPARK::Problem::DynamicBuild::ParseCommandLine()
- SPARK::Problem::DynamicBuild::ShowCommandLineUsage()
- SPARK::Problem::DynamicBuild::Load()

- SPARK::Problem::Unload()
- SPARK::Problem::Get()
- SPARK::Problem::Initialize()
- SPARK::Problem::LoadPreferenceSettings()
- SPARK::Problem::Terminate()
- SPARK::Problem::Save()
- SPARK::Problem::Restore()
- SPARK::Problem::Simulate()
- SPARK::Problem::Step()
- SPARK::Problem::StaticStep()

6.19 Document Type Definition file

The following code snippet shows the Document Type Definition (DTD) file that describes the XML grammar used to represent the problem topology. The DTD file contains the set of rules that specify which tags can be used and what they can contain. It is used by the program setupcpp to generate the XML file corresponding to your SPARK problem.

```
<-- package (root element) -->
<!ELEMENT package ( functions, problem ) >
<!ATTLIST package
            name              CDATA   #REQUIRED
            filename          CDATA   #REQUIRED
            date              CDATA   #REQUIRED
            version           CDATA   #REQUIRED
            cwd               CDATA   #REQUIRED
>


<-- functions -->
<!ELEMENT functions ( function+ ) >
<!ATTLIST functions
            size              CDATA   #REQUIRED
>


<-- function  -->
<!ELEMENT function EMPTY >
<!ATTLIST function
            name              (#PCDATA)   #REQUIRED
            type              ( ProtoType_MODIFIER     |
                                ProtoType_NON_MODIFIER |
                                ProtoType_STATIC
                              )   #REQUIRED
            library           (#PCDATA)   #REQUIRED
>


<-- problem -->
<!ELEMENT problem ( inverses, variables, components ) >
<!ATTLIST problem
            name              CDATA   #REQUIRED
>


<-- inverses -->
<!ELEMENT inverses ( inverse+ ) >
<!ATTLIST inverses
            size              CDATA   #REQUIRED
>


<-- inverse -->
<!ELEMENT inverse ( callbacks ) >
<!ATTLIST inverse
            handle            ID      #REQUIRED
            name              CDATA   #REQUIRED
            classname         CDATA   #REQUIRED
            filename          CDATA   #REQUIRED
            type              ( AtomicType_DEFAULT    |
                                AtomicType_INTEGRATOR |
                                AtomicType_SINK
                              )   #REQUIRED
>


<-- callbacks -->
<!ELEMENT callbacks ( callback* ) >
<!ATTLIST callbacks
            size              CDATA   #REQUIRED
```

```
>


<-- callback -->
<!ELEMENT callback EMPTY >
<!ATTLIST callback
            type              ( CallbackType_EVALUATE              |
                                CallbackType_PREDICT               |
                                CallbackType_CONSTRUCT             |
                                CallbackType_DESTRUCT              |
                                CallbackType_PREPARE_STEP          |
                                CallbackType_CHECK_STEP            |
                                CallbackType_COMMIT                |
                                CallbackType_ROLLBACK              |
                                CallbackType_STATIC_CONSTRUCT      |
                                CallbackType_STATIC_DESTRUCT       |
                                CallbackType_STATIC_PREPARE_STEP   |
                                CallbackType_STATIC_CHECK_STEP     |
                                CallbackType_STATIC_COMMIT         |
                                CallbackType_STATIC_ROLLBACK
                              )   #REQUIRED
            function_name     CDATA   #REQUIRED
            return_type       ( ReturnType_VALUE        |
                                ReturnType_RESIDUAL
                              )   #IMPLIED
>


<-- variables -->
<!ELEMENT variables (variable+) >
<!ATTLIST variables
            size              CDATA   #REQUIRED
            num_past_values   CDATA   #REQUIRED
>


<-- variable -->
<!ELEMENT variable EMPTY >
<!ATTLIST variable
            handle            ID     #REQUIRED
            name              CDATA  #REQUIRED
            unit              CDATA  #REQUIRED
            type              ( VariableType_CLOCK     |
                                VariableType_DT        |
                                VariableType_PARAMETER |
                                VariableType_INPUT     |
                                VariableType_UNKNOWN
                              )   #REQUIRED
            init              CDATA      #REQUIRED
            min               CDATA      #REQUIRED
            max               CDATA      #REQUIRED
            atol              CDATA      #REQUIRED
            from_link_id      IDREF      #IMPLIED
            url_rd            CDATA      #IMPLIED
            url_wr            CDATA      #IMPLIED
>


<-- components -->
<!ELEMENT components ( component+ ) >
<!ATTLIST components
            size              CDATA   #REQURIED
>


<-- component -->
<!ELEMENT component ( normal_unknowns, break_unknowns, objects ) >
<!ATTLIST component
            handle            ID      #REQUIRED
>
```

```
<-- normal_unknowns -->
<!ELEMENT normal_unknowns ( unknown* ) >
<!ATTLIST normal_unknowns
            size              CDATA   #REQURIED
>


<-- break_unknowns -->
<!ELEMENT break_unknowns ( unknown* ) >
<!ATTLIST break_unknowns
            size              CDATA   #REQURIED
>


<-- unknown -->
<!ELEMENT unknown EMPTY >
<!ATTLIST unknown
            var_id       IDREF   #REQUIRED
>


<-- objects -->
<!ELEMENT objects (object+) >
<!ATTLIST objects
            size              CDATA   #REQURIED
>


<-- object -->
<!ELEMENT object ( targets, arglists ) >
<!ATTLIST object
            handle            ID      #REQUIRED
            name              CDATA   #REQUIRED
            inverse_id        IDREF   #REQUIRED
>


<-- targets -->
<!ELEMENT targets ( target* ) >
<!ATTLIST targets
            size              CDATA   #REQUIRED
>


<-- target -->
<!ELEMENT target EMPTY >
<!ATTLIST target
            var_id            IDREF   #REQUIRED
>


<-- arglists -->
<!ELEMENT arglists ( arglist* ) >
<!ATTLIST arglists
            size              CDATA   #REQUIRED
>


<-- arglist -->
<!ELEMENT arglist ( argument* ) >
<!ATTLIST arglist
            size              CDATA   #REQUIRED
            callback_type     ( CallbackType_CONSTRUCT     |
                                CallbackType_DESTRUCT      |
                                CallbackType_EVALUATE      |
                                CallbackType_PREDICT       |
                                CallbackType_PREPARE_STEP  |
                                CallbackType_CHECK_STEP    |
```

```
                                CallbackType_COMMIT        |
                                CallbackType_ROLLBACK
                              )   #REQUIRED
>


<-- argument -->
<!ELEMENT argument EMPTY >
<!ATTLIST argument
            var_id            IDREF   #REQUIRED
>
```

6.20 Makefile

This makefile makefile.prj facilitates the process of building a SPARK simulator with the default single-problem driver function.

Usage:

```
gmake [PROJ=xxxx]
      [SPARK_CLASSPATH=yyyy]
      [PARSER_LOG=zzzz]
      [clean or cleanALL or distclean or help or makefile.inc or pkg or prf or run or solver or stp]
      [SPARK_STATIC_BUILD=yes]
      [SPARK_DEBUG=yes]
<enter>
```

Where items in [] are optional.

Type

```
gmake help <enter>
```

to obtain the description of the various targets and environment variables.

```
ifndef SPARK_STATIC_BUILD
 SPARK_STATIC_BUILD := no
endif
## Usage:
##   gmake  [PROJ=xxxx] [SPARK_CLASSPATH=yyyy] [PARSER_LOG=zzzz] [clean or
##      cleanALL or distclean or help or makefile.inc or pkg or prf or
##      run or solver or stp] [SPARK_STATIC_BUILD=yes] [SPARK_DEBUG=yes]
##        Where items in [] are optional .
##
##      First time, type:
##              gmake PROJ=xxxx  SPARK_CLASSPATH=../class,...etc...
##        To create the file 'makefile.inc' and the solver executable
##          that has the same name as PROJ .
##
##      If 'PROJ=xxxx' is not specified, the default PROJ name is the current
##        directory name.
##      If 'SPARK_CLASSPATH=...' is not specified, the default for it is :
##        1) The value of the environment variable SPARK_CLASSPATH if that
##             environment variable exists.
##        or
##        2) .,../class,$(SPARK_DIR)/globalclass,$(SPARK_DIR)/hvactk/class
##
##      After the first time, type:
##              gmake
##        To rebuild the 'makefile.inc' and the solver executable as
##          needed using the previous SPARK_CLASSPATH.
##
## If you want a build with Debug information, either set the environment
##  variable SPARK_DEBUG=yes, or run gmake with "SPARK_DEBUG=yes" in command line. e.g.
##      gmake .... SPARK_DEBUG=yes
##
## If you want a build that does not load the problem at run time,
##  run gmake with "SPARK_STATIC_BUILD=yes" in command line. e.g.
##      gmake ..... SPARK_STATIC_BUILD=yes
##
##      Other targets are:
##              gmake clean     # To delete all intermediate files.
##              gmake cleanALL  # clean + delete all run subdirectories.
##              gmake distclean # cleanALL + delete .bak .set files.
##              gmake help      # To show this information.
##              gmake makefile.inc # To create makefile.inc file.
##              gmake pkg       # To make ../PROJ_pkg directory.
##              gmake proj_lib  # To make PROJ.a file that contains a library
##                                   of compiled atomic classes.
```

```
##              gmake prf       # To make PROJ.prf file.
##              gmake run       # To make a run using PROJ.inp as input and
##                                  PROJ.run as run control file.  If PROJ.run
##                                  is missing a default one is created.
##              gmake solver    # To make the solver executable i.e. PROJ .
##              gmake stp       # To make PROJ.stp file.
##
##      To change the parser log file name from 'parser.log' to xyz.log , run:
##              gmake PARSER_LOG=xyz.log  ...
##
##      Exit status values are put in file gmake.status :
##
##        1 : Parser error.  Caused by bad ....pr , ....cm , or ....cc file, or
##              bad SPARK_CLASSPATH .  Look at file parser.log .
##        2 : Setupcpp error.   Look at file setupcpp.log .
##        3 : Compiler error while compiling class.cc file(s).
##        4 : Compiler error while compiling M_PROJ.cpp file.
##        5 : Linker error while linking M_PROJ_.o and <CLASS...>.o files.
##        6 : Run error.  Usually 'cant converge'. Look at file run.log .
##
## makefile.inc looks like:
##      M_PROJ=   xxxx
##      M_DEPsCC= ../../globalclass/aaaa.cc  ../class/bbbb.cc
##      M_DEPsCM= ../../globalclass/mmmm.cm
##      SPARK_CLASSPATH= ".,$(SPARK_DIR)/globalclass"
##
##
##
##

include $(SPARK_DIR)/lib/make_compiler.inc

ifneq ("$(MAKECMDGOALS)","help")
 ifndef PROJ
  PROJ := $(notdir $(CURDIR))
 endif
 M_PROJ := $(PROJ)
 ifndef  SPARK_CLASSPATH
  include $(SPARK_DIR)/classpath.env
 endif
 ifndef  SPARK_CLASSPATH
  SPARK_CLASSPATH := .,../class,$(SPARK_DIR)/globalclass,$(SPARK_DIR)/hvactk/class
 endif
 ifeq ($(wildcard gmake.status),gmake.status)
  _TMP_COMMAND_ := $(shell rm -f gmake.status)
 endif
 include  makefile.inc
endif

PARSER_LOG := parser.log
PARSER     := $(SPARK_DIR)/bin/parser$(EXE)
PARSER_FLAGS := -e4 -s6 -p "$(SPARK_CLASSPATH)"
SETUP :=    $(SPARK_DIR)/bin/setupcpp$(EXE)
MKMAKINC := $(SPARK_DIR)/bin/mkmakinc$(EXE)

override CXXINCLUDES :=-I$(SPARK_DIR)/inc

DEBUGPREFIX :=
ifdef SPARK_DEBUG
 ifeq ($(SPARK_DEBUG),yes)
  DEBUGPREFIX := DEBUG/
 endif
endif

ifeq ($(SPARK_STATIC_BUILD),yes)
 override CXXDEFINES :=$(CXXDEFINES) -DSPARK_STATIC_BUILD
 LIBS := sparksolver_static.$(OBJ_EXT) *_static.$(LIB_EXT)
else
 LIBS := *.$(LIBDYN_EXT)
 ifdef WINDIR
```

```
  ifeq ($(CXX),vcpp.sh)
   LIBDYNLIBS := $(SPARK_DIR)/lib/$(DEBUGPREFIX)libsolver.$(LIB_EXT)
  else
   LIBDYNLIBS := $(SPARK_DIR)/bin/$(DEBUGPREFIX)*.$(LIBDYN_EXT)
  endif
 else
  LIBDYNLIBS :=
 endif
endif

ifdef WINDIR
 SYSLIBS := -lm
else
 SYSLIBS := -lm -ldl
endif

override LDXXLIBS := $(addprefix $(SPARK_DIR)/lib/$(DEBUGPREFIX),$(LIBS)) $(SYSLIBS)

DEPsO    := $(M_DEPsCC:%.cc=%.$(OBJ_EXT))
DEPsODYN := $(M_DEPsCC:%.cc=%.$(LIBDYN_EXT))

%.$(OBJ_EXT) :  %.cc
        -rm -f  $@
        $(CXX) -c -o $@ $(CXXFLAGS) $(CXXOPTIONS) $(CXXDEFINES) $(CXXINCLUDES) $< ;\
         if [ $$? != 0 ] ; then  echo 3 > gmake.status ; exit 3 ; fi

%.$(OBJ_EXT) :  %.cpp
        -rm -f  $@
        $(CXX) -c -o $@ $(CXXFLAGS) $(CXXOPTIONS) $(CXXDEFINES) $(CXXINCLUDES) $< ;\
         if [ $$? != 0 ] ; then  echo 3 > gmake.status ; exit 3 ; fi

%.$(LIBDYN_EXT) :  %.$(OBJ_EXT)
        -rm -f  $@
        $(LIBDYN_COMMAND) -o $@ $^ $(LIBDYNLIBS) ;\
         if [ $$? != 0 ] ; then  echo 3 > gmake.status ; exit 3 ; fi
ifeq ($(CXX),vcpp.sh)
        rm -f $*.lib $*.exp
endif

ifeq ($(SPARK_STATIC_BUILD),yes)
 TARGET := $(M_PROJ)$(EXE)
else
 TARGET := $(M_PROJ).xml
endif


.PHONY :  clean cleanALL distclean help pkg prf proj_lib run solver stp $(M_PROJ).out

solver :  makefile.inc
        $(MAKE) $(TARGET)

run :  makefile.inc
        $(MAKE) $(M_PROJ).out

ifeq ($(SPARK_STATIC_BUILD),yes)
 $(TARGET) :  $(M_PROJ)_.$(OBJ_EXT) $(DEPsO)
        -rm -f $@
        $(LDXX) -o $@ $(LDXXFLAGS) $(LDXXOPTIONS) $^ $(LDXXLIBS) ;\
         if [ $$? != 0 ] ; then  echo 5 > gmake.status ; exit 5 ; fi

 $(M_PROJ)_.$(OBJ_EXT) :  $(M_PROJ).cpp
        -rm -f $@
        $(CXX) -c -o $@ $(CXXFLAGS) $(CXXOPTIONS) $(CXXDEFINES) $(CXXINCLUDES) $< ;\
          if [ $$? != 0 ] ; then  echo 4 > gmake.status ; exit 4 ; fi
else
 $(TARGET) :  $(DEPsODYN)
endif

.PRECIOUS :  $(DEPsODYN)
proj_lib :  $(DEPsODYN)
```

```
##proj_lib :  $(M_PROJ).$(LIB_EXT)
##$(M_PROJ).$(LIB_EXT) :  $(DEPsODYN)
##      -rm -f $@
##      $(LIB_FROM_OBJs)

.PRECIOUS :  $(M_PROJ).prf $(M_PROJ).cpp $(M_PROJ).xml
prf :  $(M_PROJ).prf
$(M_PROJ).prf :  $(M_PROJ).cpp
$(M_PROJ).cpp :  $(M_PROJ).xml
$(M_PROJ).xml :  $(M_PROJ).stp  $(SETUP)
        -rm -f  $@  setupcpp.log
        if [ -r $(M_PROJ).eqs ] ; then \
         rm -f $(M_PROJ).eqs.tmp ; tail +6 $(M_PROJ).eqs > $(M_PROJ).eqs.tmp ;\
         rm -f $(M_PROJ).prf.tmp ; cp -p $(M_PROJ).prf $(M_PROJ).prf.tmp ; fi
        $(SETUP)  $(M_PROJ)   > setupcpp.log  2>&1 ;\
         if [ $$? != 0 ] ; then  echo 2 > gmake.status ; exit 2 ; fi
        if [ -r $(M_PROJ).eqs.tmp -a -r $(M_PROJ).eqs ] ; then \
         tail +6 $(M_PROJ).eqs > $(M_PROJ).eqsNew.tmp ; \
         if cmp -s  $(M_PROJ).eqs.tmp $(M_PROJ).eqsNew.tmp ; then \
          cp -p $(M_PROJ).prf.tmp $(M_PROJ).prf ;\
         fi ; rm -f $(M_PROJ).eqsNew.tmp ; fi
        rm -f  $(M_PROJ).eqs.tmp  $(M_PROJ).prf.tmp

stp :  $(M_PROJ).stp
$(M_PROJ).stp :  $(M_PROJ).pr  $(M_DEPsCM)  $(M_DEPsCC)  $(PARSER)
        -rm -f $(M_PROJ).stp $(M_PROJ).tmp $(PARSER_LOG)
        $(PARSER) $(PARSER_FLAGS)  $< 2>&1 | tee $(PARSER_LOG) ;\
          if [ $$? != 0 ] ; then  echo 1 > gmake.status ; exit 1 ; fi

$(M_PROJ).out :  $(TARGET) $(M_PROJ).prf $(M_PROJ).run $(M_PROJ).xml
        -rm -f $@ run.log
ifeq ($(SPARK_STATIC_BUILD),yes)
        ./$(TARGET) $(M_PROJ).prf $(M_PROJ).run > run.log ;\
          X=$$? ; if [ $$X != 0 ]; then  echo $$X > gmake.status; exit $$X ; fi
else
        $(SPARK_DIR)/bin/$(DEBUGPREFIX)sparksolver $(M_PROJ).prf $(M_PROJ).run $(M_PROJ).xml > run.log ;\
          X=$$? ; if [ $$X != 0 ]; then  echo $$X > gmake.status; exit $$X ; fi
endif

$(M_PROJ).run :
        @echo "Creating a default $(M_PROJ).run file"
        -rm -f  $@ ;  cp $(SPARK_DIR)/lib/default.run  $@
        if [ -r $(M_PROJ).inp ]; then  repref $@ InputFiles = "$(M_PROJ).inp" ;\
          else  echo "WARNING: Input file $(M_PROJ).inp does not exist." ;  fi
        repref $@  OutputFile = "$(M_PROJ).out"

makefile.inc :  $(M_PROJ).pr  $(M_DEPsCM)  $(PARSER)
        -rm -f  $(M_PROJ).stp $(M_PROJ).tmp $(PARSER_LOG)
        $(PARSER) $(PARSER_FLAGS) $< 2>&1 | tee $(PARSER_LOG) ;\
        if [ -r $(M_PROJ).stp ] ; then   \
         Stp=$(M_PROJ).stp              ;\
        elif [ -r $(M_PROJ).tmp ] ; then \
         Stp=$(M_PROJ).tmp              ;\
         echo 1 > gmake.status          ;\
        else                             \
         echo 1 > gmake.status ; exit 1 ;\
        fi                              ;\
        rm -f   makefile.inc            ;\
        $(MKMAKINC)  $$Stp  >  makefile.inc     ;\
        echo "SPARK_CLASSPATH=" $(SPARK_CLASSPATH)  >> makefile.inc

pkg :  makefile.inc
        -rm -rf $(M_PROJ)_pkg
        mkdir   $(M_PROJ)_pkg
        cp -p   $(M_DEPsCC)  $(M_DEPsCM)        $(M_PROJ)_pkg
        -cp -p  *.set                           $(M_PROJ)_pkg
        cp -p   $(M_PROJ).pr    $(M_PROJ)_pkg/$(M_PROJ)_pkg.pr
        cp -p   makefile.inc    $(M_PROJ)_pkg/makefile.inc.old
        if [ -r  $(M_PROJ).inp  ] ;   then      \
          cp -p  $(M_PROJ).inp  $(M_PROJ)_pkg/$(M_PROJ)_pkg.inp ;\
```

```
        fi
        -cp -p *.inp  $(M_PROJ)_pkg
        -cp -p *.prf  $(M_PROJ)_pkg
        -cp -p *.run  $(M_PROJ)_pkg
        if [ -d  ../$(M_PROJ)_pkg ] ; then  rm -rf ../$(M_PROJ)_pkg ; fi
        mv  $(M_PROJ)_pkg ..
clean :
        rm -f $(M_PROJ).stp  $(PARSER_LOG)  parser.log  gmake.status
        rm -f $(M_PROJ).cpp  $(M_PROJ).eqs  setupcpp.log
        rm -f $(TARGET)  $(M_PROJ)_.$(OBJ_EXT)
        rm -f $(M_PROJ).$(LIB_EXT)
        rm -f $(M_PROJ).out  $(M_PROJ).xml  solver.status
        rm -f $(DEPsO)
        rm -f $(DEPsODYN)
        rm -f *.factory.log
        rm -f debug.log error.log run.log .m.log
        -rm -f backtracking_$(M_PROJ)_*.log
        -rm -f *.template
        -rm -f *.tmp
        -rm -f \#*
        -rm -f *~
        -rm -f *\%
        -rm -f *.rsp
        rm -f makestp.log symbolics.log sparksym.log matho.err tmp.txt
        if [ -r makefile.inc ] ;  then  rm -f   makefile.inc.bak ; \
                mv makefile.inc makefile.inc.bak ;  fi
cleanALL :  clean
        if [ -r $(M_PROJ).prf ] ; then \
         rm -f   $(M_PROJ).prf.bak ; mv $(M_PROJ).prf $(M_PROJ).prf.bak ;  fi
        -rm -f default.prf
        -rm -f *.init
        -rm -f *.snap
        if [ -r *.set ] ;  then \
         for I in *.set ;  do  rm -rf `basename $$I .set` ;  done ; fi
        -rm -f *.trc
distclean :  cleanALL
        -rm -f $(M_PROJ).*.bak
        -rm -f makefile  makefile.inc.bak
        -rm -f  $(M_PROJ)_inp.set
help :
        @head -70 $(SPARK_DIR)/lib/makefile.prj
        @echo "SPARK_CLASSPATH=$(SPARK_CLASSPATH)"
        @echo "PROJ=$(PROJ)"
        @echo "SPARK_DEBUG=$(SPARK_DEBUG)"
        @echo "SPARK_STATIC_BUILD=$(SPARK_STATIC_BUILD)"
        @echo "(wildcard classpath.env)=$(wildcard classpath.env)"
        @echo "(wildcard ../classpath.env)=$(wildcard ../classpath.env)"
        @echo "(wildcard $(SPARK_DIR)/classpath.env)=$(SPARK_DIR)/classpath.env)"
        @echo "CXXFLAGS=$(CXXFLAGS)"
        @echo "CXXOPTIONS=$(CXXOPTIONS)"
        @echo "CXXDEFINES=$(CXXDEFINES)"
        @echo "CXXINCLUDES=$(CXXINCLUDES)"
        @echo "LDXXFLAGS=$(LDXXFLAGS)"
        @echo "LDXXOPTIONS=$(LDXXOPTIONS)"
        @echo "LDXXLIBS=$(LDXXLIBS)"
```

6.21 Multi-problem command file

The file build_multiproblem facilitates the process of building a SPARK simulator with a specialized driver function that makes use of possibly multiple problem descriptions.

Usage:

```
build_multiproblem <mainName> <project1dir> <project2dir> ... <enter>
```

<mainName> is the name without extension of the file where the customized driver function is implemented. The driver file <mainName>.cpp must be located in the current working directory. Also, <project1dir> refers to the path where the problem file called <project1dir>.pr can be found. Similarly, <project2dir> refers to the path where the problem file called <project2dir>.pr can be found.

Type

```
build_multiproblem nultiproblem_example1 ../room_fc ../spring  <enter>
```

to build the simulator for the driver nultiproblem_example1.cpp that uses the problems room_fc.pr and spring.pr, located in the directories ../room_fc and ../spring, respectively.

Warning:

It is the user's responsibility to make sure that the required files (∗.prf, ∗.run, ∗.inp) can be found in the directories indicated in the driver function. The ∗.map files for each problem must always be located in the current working directory, if needed. The ∗.xml files are copied by the command file to the current working directory from each problem directory.

Index

~TComponent
SPARK::TComponent, 36
~TComponentSettings
SPARK::TComponentSettings, 41
~TGlobalSettings
SPARK::TGlobalSettings, 48
~TPreferenceSettings
SPARK::TPreferenceSettings, 57
~TProblem
SPARK::TProblem, 63
~TRuntimeControls
SPARK::TRuntimeControls, 75
~TState
SPARK::TProblem::TState, 70
~XAssertion
SPARK::XAssertion, 81
NAMESPACE
sparkmacro.h, 111
QUOTE
sparkmacro.h, 111
__PROBLEM_H__
problem.h, 107

BreakUnknownSafetyFactor
SPARK::DefaultGlobalSettings, 17

CallbackType_CHECK_INTEGRATION_STEP
SPARK, 8
CallbackType_COMMIT
SPARK, 8
CallbackType_CONSTRUCT
SPARK, 8
CallbackType_DESTRUCT
SPARK, 8
CallbackType_EVALUATE
SPARK, 8
CallbackType_NONE
SPARK, 8
CallbackType_PREDICT
SPARK, 8
CallbackType_PREPARE_STEP
SPARK, 8
CallbackType_ROLLBACK
SPARK, 8
CallbackType_STATIC_CHECK_INTEGRATION_STEP
SPARK, 8
CallbackType_STATIC_COMMIT
SPARK, 8
CallbackType_STATIC_CONSTRUCT
SPARK, 8
CallbackType_STATIC_DESTRUCT
SPARK, 8
CallbackType_STATIC_PREPARE_STEP
SPARK, 8
CallbackType_STATIC_ROLLBACK
SPARK, 8
CallbackTypes
SPARK, 7
CALLBACKTYPES_L
SPARK, 8
CheckBadNumericsFlag
SPARK::DefaultComponentSettings, 14
code
SPARK::XAssertion, 82
component.h, 97
ComponentSolvingMethod
SPARK::DefaultComponentSettings, 14
ComponentType_STRONG
SPARK::TComponent, 36
ComponentType_WEAK
SPARK::TComponent, 36
ComponentTypes
SPARK::TComponent, 36
ConsistentInitialCalculation
SPARK::DefaultRuntimeControls, 19
ctrls.h, 99

DiagnosticLevel
SPARK::DefaultRuntimeControls, 19
DiagnosticType_CONVERGENCE
SPARK::TProblem, 62
DiagnosticType_INPUTS
SPARK::TProblem, 62
DiagnosticType_PREFERENCES
SPARK::TProblem, 62
DiagnosticType_REPORTS
SPARK::TProblem, 62
DiagnosticType_REQUESTS
SPARK::TProblem, 62
DiagnosticType_STATISTICS
SPARK::TProblem, 62
DiagnosticTypes
SPARK::TProblem, 62
DIAGNOSTICTYPES_L

SPARK::TProblem, 62

empty
SPARK::Problem::simulation_parameter, 32
EMPTY_ARRAY
sparkmacro.h, 111
End
SPARK, 10
END_ARRAY
sparkmacro.h, 111
END_INVERSES
sparkmacro.h, 112
END_PROBLEM
sparkmacro.h, 112
Epsilon
SPARK::DefaultComponentSettings, 15
Evaluate
SPARK::TComponent, 36
exceptions.h, 100
exitcode.h, 102
ExitCode_ERROR_COMMAND_LINE
SPARK, 7
ExitCode_ERROR_EXIT_SPARK_FACTORY
SPARK, 7
ExitCode_ERROR_INVALID_FEATURE
SPARK, 7
ExitCode_ERROR_INVALID_PREFERENCES
SPARK, 7
ExitCode_ERROR_INVALID_PROBLEM
SPARK, 7
ExitCode_ERROR_INVALID_RUN_CONTROLS
SPARK, 7
ExitCode_ERROR_INVALID_VARIABLE_NAME
SPARK, 7
ExitCode_ERROR_IO
SPARK, 7
ExitCode_ERROR_LEX_SCAN
SPARK, 7
ExitCode_ERROR_NULL_POINTER
SPARK, 7
ExitCode_ERROR_NUMERICAL
SPARK, 7
ExitCode_ERROR_OUT_OF_MEMORY
SPARK, 7
ExitCode_ERROR_RUNTIME_ERROR
SPARK, 7
ExitCode_ERROR_URL
SPARK, 7
ExitCode_OK
SPARK, 7
ExitCodes
SPARK, 7
ExitWithError
SPARK, 10

FinalTime
SPARK::DefaultRuntimeControls, 18
FirstReport
SPARK::DefaultRuntimeControls, 19

GenerateSnapshot
SPARK::TProblem, 68
Get
SPARK::Problem, 22
get
SPARK::Problem::simulation_parameter, 32
GetActiveCallbackName
SPARK::TInverse, 52
SPARK::TObject, 55
GetAtomicClassName
SPARK::TInverse, 52
GetAtomicType
SPARK::TInverse, 52
GetBaseName
SPARK, 11
GetBreakUnknownSafetyFactor
SPARK::TGlobalSettings, 49
GetCheckBadNumericsFlag
SPARK::TComponentSettings, 42
GetComponent
SPARK::TObject, 55
GetComponentSettings
SPARK::TPreferenceSettings, 57
GetComponentSolvingMethod
SPARK::TComponentSettings, 41
GetConsistentInitialCalculation
SPARK::TRuntimeControls, 79
GetContext
SPARK::TProblem::TState, 70
GetCount
SPARK::TGlobalSettings, 50
GetData
SPARK::TObject, 55
GetDebugLogFilename
SPARK, 12
GetDiagnosticLevel
SPARK::TRuntimeControls, 77
GetEpsilon
SPARK::TComponentSettings, 43
GetErrorLog
SPARK, 12
GetErrorLogFilename
SPARK, 12
GetFileName
SPARK, 10
SPARK::TInverse, 52
GetFinalSnapshotFileName
SPARK::TRuntimeControls, 75
GetFinalTime
SPARK::TRuntimeControls, 77
GetFirstReport
SPARK::TRuntimeControls, 78

GetGlobalSettings
SPARK::TPreferenceSettings, 57
SPARK::TProblem, 65
GetGlobalTime
SPARK::TProblem, 66
GetGlobalTimeStep
SPARK::TProblem, 66
GetHandle
SPARK::TComponent, 36
SPARK::TInverse, 52
SPARK::TObject, 54
GetIncrementsTracerFilename
SPARK::TComponentSettings, 45
GetInitialSnapshotFileName
SPARK::TRuntimeControls, 75
GetInitialTime
SPARK::TRuntimeControls, 77
GetInitialTimeStep
SPARK::TRuntimeControls, 77
GetInitialWallClock
SPARK::TRuntimeControls, 76
GetInputFiles
SPARK::TRuntimeControls, 76
GetInstanceHandle
SPARK::TObject, 54
GetInverse
SPARK::TObject, 55
SPARK::TProblem, 67
GetIterationSafetyFactor
SPARK::TGlobalSettings, 49
GetJacobianRefreshRatio
SPARK::TComponentSettings, 42
GetJacobianTracerFilename
SPARK::TComponentSettings, 45
GetMatrixSolvingMethod
SPARK::TComponentSettings, 43
GetMaxIterations
SPARK::TComponentSettings, 42
GetMaxRelaxationCoefficient
SPARK::TComponentSettings, 43
GetMaxTimeStep
SPARK::TRuntimeControls, 78
GetMaxTolerance
SPARK::TGlobalSettings, 49
GetMinIterations
SPARK::TComponentSettings, 42
GetMinRelaxationCoefficient
SPARK::TComponentSettings, 43
GetMinTimeStep
SPARK::TRuntimeControls, 78
GetName
SPARK::TComponent, 36
SPARK::TInverse, 52
SPARK::TObject, 55
SPARK::TProblem, 65
SPARK::TRuntimeControls, 75
GetNormalUnknownSafetyFactor
SPARK::TGlobalSettings, 50
GetNumObjects
SPARK::TComponent, 36
SPARK::TInverse, 52
GetNumPastValues
SPARK::TProblem::TState, 71
SPARK::TRuntimeControls, 76
GetNumVariables
SPARK::TProblem::TState, 70
GetObject
SPARK::TComponent, 36, 37
SPARK::TInverse, 52
SPARK::TProblem, 67
GetOutputFileName
SPARK::TRuntimeControls, 76
GetPastValues
SPARK::TProblem::TState, 71
GetPivotingMethod
SPARK::TComponentSettings, 44
GetPredictionSafetyFactor
SPARK::TGlobalSettings, 49
GetPrfFileName
SPARK::TPreferenceSettings, 57
GetProblem
SPARK::TInverse, 52
SPARK::TObject, 55
GetProgramName
SPARK, 11
GetRefinementMethod
SPARK::TComponentSettings, 44
GetReportCycle
SPARK::TRuntimeControls, 78
GetResidualsTracerFilename
SPARK::TComponentSettings, 45
GetRunFileName
SPARK::TRuntimeControls, 75
GetRunLog
SPARK, 12
GetRunLogFilename
SPARK, 12
GetScalar
SPARK::TGlobalSettings, 50
GetScalingMethod
SPARK::TComponentSettings, 44
GetStaticData
SPARK::TInverse, 52
GetStepControlMethod
SPARK::TComponentSettings, 43
GetStepCount
SPARK::TProblem, 65
GetString
SPARK::TGlobalSettings, 50
GetTime

SPARK::TProblem::TState, 70
GetTimeStep
SPARK::TProblem::TState, 70
GetTimeUnit
SPARK::TRuntimeControls, 76
GetTolerance
SPARK::TGlobalSettings, 49
SPARK::TInverse, 52
SPARK::TObject, 55
GetTrueJacobianEvalStep
SPARK::TComponentSettings, 42
GetType
SPARK::TComponent, 36
GetVariable
SPARK::TProblem, 66
GetVariablesTracerFilename
SPARK::TComponentSettings, 45
GetVariableTimeStep
SPARK::TRuntimeControls, 78
GetVersion
SPARK, 12

Initialize
SPARK::Problem, 23
SPARK::TProblem, 63
InitialTime
SPARK::DefaultRuntimeControls, 18
InitialTimeStep
SPARK::DefaultRuntimeControls, 19
inverse.h, 103
IsDiagnostic
SPARK::TProblem, 68
IsFinalTime
SPARK::TProblem, 67
IsInitialTime
SPARK::TProblem, 67
IsReady
SPARK::TProblem, 68
IsStaticStep
SPARK::TProblem, 67
IsStronglyConnected
SPARK::TComponent, 37
IsTimeStepVariable
SPARK::TProblem, 68
IterationSafetyFactor
SPARK::DefaultGlobalSettings, 17

JacobianRefreshRatio
SPARK::DefaultComponentSettings, 15

Load
SPARK::Problem::DynamicBuild, 27
SPARK::Problem::StaticBuild, 29
SPARK::TComponentSettings, 41
SPARK::TGlobalSettings, 49
LoadPreferenceSettings
SPARK::Problem, 23
SPARK::TProblem, 64
Log
SPARK, 11

MatrixSolvingMethod
SPARK::DefaultComponentSettings, 15
MaxIterations
SPARK::DefaultComponentSettings, 14
MaxRelaxationCoefficient
SPARK::DefaultComponentSettings, 14
MaxTimeStep
SPARK::DefaultRuntimeControls, 19
MaxTolerance
SPARK::DefaultGlobalSettings, 16
message
SPARK::XAssertion, 81
MinIterations
SPARK::DefaultComponentSettings, 14
MinRelaxationCoefficient
SPARK::DefaultComponentSettings, 14
MinTimeStep
SPARK::DefaultRuntimeControls, 19

NormalUnknownSafetyFactor
SPARK::DefaultGlobalSettings, 17

object.h, 104
operator value_type
SPARK::Problem::simulation_parameter, 32
operator()
SPARK::XAssertion, 82
operator=
SPARK::TComponentSettings, 41
operator[]
SPARK::XAssertion, 82

ParseCommandLine
SPARK::Problem::DynamicBuild, 26
SPARK::Problem::StaticBuild, 28
PivotingMethod
SPARK::DefaultComponentSettings, 15
PredictionSafetyFactor
SPARK::DefaultGlobalSettings, 16
prefs.h, 105
problem.h, 106
__PROBLEM_H__, 107
ProtoType_MODIFIER
SPARK, 8
ProtoType_NON_MODIFIER
SPARK, 8
ProtoType_NONE
SPARK, 8
ProtoType_PREDICATE
SPARK, 8
ProtoType_STATIC_NON_MODIFIER

SPARK, 8
ProtoType_STATIC_PREDICATE
SPARK, 8
ProtoTypes
SPARK, 8
PROTOTYPES_L
SPARK, 8

RefinementMethod
SPARK::DefaultComponentSettings, 15
RegisterStaticInstance
SPARK::Problem, 21
ReportCycle
SPARK::DefaultRuntimeControls, 19
ReportStatistics
SPARK::TProblem, 68
RequestType_ABORT
SPARK, 9
RequestType_CLEAR_MEETING_POINTS
SPARK, 9
RequestType_NONE
SPARK, 9
RequestType_REPORT
SPARK, 9
RequestType_RESTART
SPARK, 9
RequestType_SET_DYNAMIC_STEPPER
SPARK, 9
RequestType_SET_MEETING_POINT
SPARK, 9
RequestType_SET_STOP_TIME
SPARK, 9
RequestType_SET_TIME_STEP
SPARK, 9
RequestType_SNAPSHOT
SPARK, 9
RequestType_STOP
SPARK, 9
RequestTypes
SPARK, 9
REQUESTTYPES_L
SPARK, 9
Reset
SPARK::TComponentSettings, 41
SPARK::TGlobalSettings, 49
SPARK::TRuntimeControls, 79
Restore
SPARK::Problem, 24
SPARK::TProblem, 65
SPARK::TProblem::TState, 71
ReturnType_NONE
SPARK, 8
ReturnType_RESIDUAL
SPARK, 8
ReturnType_VALUE
SPARK, 8
ReturnTypes
SPARK, 8
RETURNTYPES_L
SPARK, 8

Save
SPARK::Problem, 23
SPARK::TProblem, 65
SPARK::TProblem::TState, 71
ScalingMethod
SPARK::DefaultComponentSettings, 14
SetBreakUnknownSafetyFactor
SPARK::TGlobalSettings, 50
SetCheckBadNumericsFlag
SPARK::TComponentSettings, 42
SetComponentSolvingMethod
SPARK::TComponentSettings, 42
SetConsistentInitialCalculation
SPARK::TRuntimeControls, 79
SetData
SPARK::TObject, 55
SetDiagnosticLevel
SPARK::TRuntimeControls, 77
SetEpsilon
SPARK::TComponentSettings, 43
SetFinalSnapshotFileName
SPARK::TRuntimeControls, 75
SetFinalTime
SPARK::TRuntimeControls, 77
SetFirstReport
SPARK::TRuntimeControls, 78
SetIncrementsTracerFilename
SPARK::TComponentSettings, 45
SetInfiniteFinalTime
SPARK::TRuntimeControls, 77
SetInitialSnapshotFileName
SPARK::TRuntimeControls, 75
SetInitialTime
SPARK::TRuntimeControls, 77
SetInitialTimeStep
SPARK::TRuntimeControls, 77
SetInitialWallClock
SPARK::TRuntimeControls, 76
SetIterationSafetyFactor
SPARK::TGlobalSettings, 49
SetJacobianRefreshRatio
SPARK::TComponentSettings, 42
SetJacobianTracerFilename
SPARK::TComponentSettings, 45
SetMatrixSolvingMethod
SPARK::TComponentSettings, 44
SetMaxIterations
SPARK::TComponentSettings, 42
SetMaxRelaxationCoefficient
SPARK::TComponentSettings, 43
SetMaxTimeStep

SPARK::TRuntimeControls, 78
SetMaxTolerance
SPARK::TGlobalSettings, 49
SetMinIterations
SPARK::TComponentSettings, 42
SetMinRelaxationCoefficient
SPARK::TComponentSettings, 43
SetMinTimeStep
SPARK::TRuntimeControls, 78
SetName
SPARK::TProblem, 65
SetNormalUnknownSafetyFactor
SPARK::TGlobalSettings, 50
SetNumPastValues
SPARK::TRuntimeControls, 77
SetOutputFileName
SPARK::TRuntimeControls, 76
SetPivotingMethod
SPARK::TComponentSettings, 44
SetPredictionSafetyFactor
SPARK::TGlobalSettings, 49
SetRefinementMethod
SPARK::TComponentSettings, 45
SetReportCycle
SPARK::TRuntimeControls, 78
SetResidualsTracerFilename
SPARK::TComponentSettings, 45
SetScalingMethod
SPARK::TComponentSettings, 44
SetStaticData
SPARK::TInverse, 52
SetStepControlMethod
SPARK::TComponentSettings, 43
SetTimeUnit
SPARK::TRuntimeControls, 76
SetTolerance
SPARK::TGlobalSettings, 49
SetTrueJacobianEvalStep
SPARK::TComponentSettings, 42
SetVariablesTracerFilename
SPARK::TComponentSettings, 45
SetVariableTimeStep
SPARK::TRuntimeControls, 79
ShowCommandLineUsage
SPARK::Problem::DynamicBuild, 26
SPARK::Problem::StaticBuild, 28
Simulate
SPARK::Problem, 24
SPARK::TProblem, 64
simulation_parameter
SPARK::Problem::simulation_parameter, 32
SimulationFlag_BAD_NUMERICS
SPARK::TProblem, 63
SimulationFlag_FAILED_STEP
SPARK::TProblem, 63
SimulationFlag_IDLE
SPARK::TProblem, 63
SimulationFlag_NO_CONVERGENCE
SPARK::TProblem, 63
SimulationFlag_OK
SPARK::TProblem, 63
SimulationFlag_SINGULAR_SYSTEM
SPARK::TProblem, 63
SimulationFlag_TIMESTEP_TOO_SMALL
SPARK::TProblem, 63
SimulationFlags
SPARK::TProblem, 62
SIMULATIONFLAGS_L
SPARK::TProblem, 63
SPARK, 3
CallbackType_CHECK_INTEGRATION_STEP, 8
CallbackType_COMMIT, 8
CallbackType_CONSTRUCT, 8
CallbackType_DESTRUCT, 8
CallbackType_EVALUATE, 8
CallbackType_NONE, 8
CallbackType_PREDICT, 8
CallbackType_PREPARE_STEP, 8
CallbackType_ROLLBACK, 8
CallbackType_STATIC_CHECK_INTEGRATION_-
STEP, 8
CallbackType_STATIC_COMMIT, 8
CallbackType_STATIC_CONSTRUCT, 8
CallbackType_STATIC_DESTRUCT, 8
CallbackType_STATIC_PREPARE_STEP, 8
CallbackType_STATIC_ROLLBACK, 8
CallbackTypes, 7
CALLBACKTYPES_L, 8
End, 10
ExitCode_ERROR_COMMAND_LINE, 7
ExitCode_ERROR_EXIT_SPARK_FACTORY, 7
ExitCode_ERROR_INVALID_FEATURE, 7
ExitCode_ERROR_INVALID_PREFERENCES, 7
ExitCode_ERROR_INVALID_PROBLEM, 7
ExitCode_ERROR_INVALID_RUN_CONTROLS, 7
ExitCode_ERROR_INVALID_VARIABLE_NAME, 7
ExitCode_ERROR_IO, 7
ExitCode_ERROR_LEX_SCAN, 7
ExitCode_ERROR_NULL_POINTER, 7
ExitCode_ERROR_NUMERICAL, 7
ExitCode_ERROR_OUT_OF_MEMORY, 7
ExitCode_ERROR_RUNTIME_ERROR, 7
ExitCode_ERROR_URL, 7
ExitCode_OK, 7
ExitCodes, 7
ExitWithError, 10
GetBaseName, 11
GetDebugLogFilename, 12
GetErrorLog, 12
GetErrorLogFilename, 12

GetFileName, 10
GetProgramName, 11
GetRunLog, 12
GetRunLogFilename, 12
GetVersion, 12
Log, 11
ProtoType_MODIFIER, 8
ProtoType_NON_MODIFIER, 8
ProtoType_NONE, 8
ProtoType_PREDICATE, 8
ProtoType_STATIC_NON_MODIFIER, 8
ProtoType_STATIC_PREDICATE, 8
ProtoTypes, 8
PROTOTYPES_L, 8
RequestType_ABORT, 9
RequestType_CLEAR_MEETING_POINTS, 9
RequestType_NONE, 9
RequestType_REPORT, 9
RequestType_RESTART, 9
RequestType_SET_DYNAMIC_STEPPER, 9
RequestType_SET_MEETING_POINT, 9
RequestType_SET_STOP_TIME, 9
RequestType_SET_TIME_STEP, 9
RequestType_SNAPSHOT, 9
RequestType_STOP, 9
RequestTypes, 9
REQUESTTYPES_L, 9
ReturnType_NONE, 8
ReturnType_RESIDUAL, 8
ReturnType_VALUE, 8
ReturnTypes, 8
RETURNTYPES_L, 8
Start, 9
VariableType_GLOBAL_TIME, 9
VariableType_GLOBAL_TIME_STEP, 9
VariableType_INPUT, 9
VariableType_NONE, 9
VariableType_PARAMETER, 9
VariableType_UNKNOWN, 9
VariableTypes, 8
VARIABLETYPES_L, 9
SPARK::DefaultComponentSettings, 13
SPARK::DefaultComponentSettings
CheckBadNumericsFlag, 14
ComponentSolvingMethod, 14
Epsilon, 15
JacobianRefreshRatio, 15
MatrixSolvingMethod, 15
MaxIterations, 14
MaxRelaxationCoefficient, 14
MinIterations, 14
MinRelaxationCoefficient, 14
PivotingMethod, 15
RefinementMethod, 15
ScalingMethod, 14
StepControlMethod, 14
TrueJacobianEvalStep, 14
SPARK::DefaultGlobalSettings, 16
SPARK::DefaultGlobalSettings
BreakUnknownSafetyFactor, 17
IterationSafetyFactor, 17
MaxTolerance, 16
NormalUnknownSafetyFactor, 17
PredictionSafetyFactor, 16
Tolerance, 16
SPARK::DefaultRuntimeControls, 18
SPARK::DefaultRuntimeControls
ConsistentInitialCalculation, 19
DiagnosticLevel, 19
FinalTime, 18
FirstReport, 19
InitialTime, 18
InitialTimeStep, 19
MaxTimeStep, 19
MinTimeStep, 19
ReportCycle, 19
VariableTimeStep, 19
SPARK::Problem, 20
Get, 22
Initialize, 23
LoadPreferenceSettings, 23
RegisterStaticInstance, 21
Restore, 24
Save, 23
Simulate, 24
StaticStep, 25
Step, 24
Terminate, 23
TRestartFlag, 21
TStopTime, 21
TTimeStep, 21
Unload, 22
WriteInstances, 22
SPARK::Problem::DynamicBuild, 26
SPARK::Problem::DynamicBuild
Load, 27
ParseCommandLine, 26
ShowCommandLineUsage, 26
SPARK::Problem::simulation_parameter, 31
empty, 32
get, 32
operator value_type, 32
simulation_parameter, 32
value_type, 32
SPARK::Problem::StaticBuild, 28
SPARK::Problem::StaticBuild
Load, 29
ParseCommandLine, 28
ShowCommandLineUsage, 28
SPARK::TComponent, 34

~TComponent, 36
ComponentType_STRONG, 36
ComponentType_WEAK, 36
ComponentTypes, 36
Evaluate, 36
GetHandle, 36
GetName, 36
GetNumObjects, 36
GetObject, 36, 37
GetType, 36
IsStronglyConnected, 37
TComponent, 36
SPARK::TComponentSettings, 38
SPARK::TComponentSettings
~TComponentSettings, 41
GetCheckBadNumericsFlag, 42
GetComponentSolvingMethod, 41
GetEpsilon, 43
GetIncrementsTracerFilename, 45
GetJacobianRefreshRatio, 42
GetJacobianTracerFilename, 45
GetMatrixSolvingMethod, 43
GetMaxIterations, 42
GetMaxRelaxationCoefficient, 43
GetMinIterations, 42
GetMinRelaxationCoefficient, 43
GetPivotingMethod, 44
GetRefinementMethod, 44
GetResidualsTracerFilename, 45
GetScalingMethod, 44
GetStepControlMethod, 43
GetTrueJacobianEvalStep, 42
GetVariablesTracerFilename, 45
Load, 41
operator=, 41
Reset, 41
SetCheckBadNumericsFlag, 42
SetComponentSolvingMethod, 42
SetEpsilon, 43
SetIncrementsTracerFilename, 45
SetJacobianRefreshRatio, 42
SetJacobianTracerFilename, 45
SetMatrixSolvingMethod, 44
SetMaxIterations, 42
SetMaxRelaxationCoefficient, 43
SetMinIterations, 42
SetMinRelaxationCoefficient, 43
SetPivotingMethod, 44
SetRefinementMethod, 45
SetResidualsTracerFilename, 45
SetScalingMethod, 44
SetStepControlMethod, 43
SetTrueJacobianEvalStep, 42
SetVariablesTracerFilename, 45
TComponentSettings, 41
Write, 45
SPARK::TGlobalSettings, 47
SPARK::TGlobalSettings
~TGlobalSettings, 48
GetBreakUnknownSafetyFactor, 49
GetCount, 50
GetIterationSafetyFactor, 49
GetMaxTolerance, 49
GetNormalUnknownSafetyFactor, 50
GetPredictionSafetyFactor, 49
GetScalar, 50
GetString, 50
GetTolerance, 49
Load, 49
Reset, 49
SetBreakUnknownSafetyFactor, 50
SetIterationSafetyFactor, 49
SetMaxTolerance, 49
SetNormalUnknownSafetyFactor, 50
SetPredictionSafetyFactor, 49
SetTolerance, 49
TGlobalSettings, 48
Write, 50
SPARK::TInverse, 51
GetActiveCallbackName, 52
GetAtomicClassName, 52
GetAtomicType, 52
GetFileName, 52
GetHandle, 52
GetName, 52
GetNumObjects, 52
GetObject, 52
GetProblem, 52
GetStaticData, 52
GetTolerance, 52
SetStaticData, 52
SPARK::TObject, 54
GetActiveCallbackName, 55
GetComponent, 55
GetData, 55
GetHandle, 54
GetInstanceHandle, 54
GetInverse, 55
GetName, 55
GetProblem, 55
GetTolerance, 55
SetData, 55
SPARK::TPreferenceSettings, 56
SPARK::TPreferenceSettings
~TPreferenceSettings, 57
GetComponentSettings, 57
GetGlobalSettings, 57
GetPrfFileName, 57
TPreferenceSettings, 57
Write, 57

SPARK::TProblem, 59
~TProblem, 63
DiagnosticType_CONVERGENCE, 62
DiagnosticType_INPUTS, 62
DiagnosticType_PREFERENCES, 62
DiagnosticType_REPORTS, 62
DiagnosticType_REQUESTS, 62
DiagnosticType_STATISTICS, 62
DiagnosticTypes, 62
DIAGNOSTICTYPES_L, 62
GenerateSnapshot, 68
GetGlobalSettings, 65
GetGlobalTime, 66
GetGlobalTimeStep, 66
GetInverse, 67
GetName, 65
GetObject, 67
GetStepCount, 65
GetVariable, 66
Initialize, 63
IsDiagnostic, 68
IsFinalTime, 67
IsInitialTime, 67
IsReady, 68
IsStaticStep, 67
IsTimeStepVariable, 68
LoadPreferenceSettings, 64
ReportStatistics, 68
Restore, 65
Save, 65
SetName, 65
Simulate, 64
SimulationFlag_BAD_NUMERICS, 63
SimulationFlag_FAILED_STEP, 63
SimulationFlag_IDLE, 63
SimulationFlag_NO_CONVERGENCE, 63
SimulationFlag_OK, 63
SimulationFlag_SINGULAR_SYSTEM, 63
SimulationFlag_TIMESTEP_TOO_SMALL, 63
SimulationFlags, 62
SIMULATIONFLAGS_L, 63
Starting, 67
TDiagnosticLevel, 62
Terminate, 64
TProblem, 63
WriteStamp, 68
SPARK::TProblem::TState, 69
~TState, 70
GetContext, 70
GetNumPastValues, 71
GetNumVariables, 70
GetPastValues, 71
GetTime, 70
GetTimeStep, 70
Restore, 71
Save, 71
TState, 70
SPARK::TRuntimeControls, 72
SPARK::TRuntimeControls
~TRuntimeControls, 75
GetConsistentInitialCalculation, 79
GetDiagnosticLevel, 77
GetFinalSnapshotFileName, 75
GetFinalTime, 77
GetFirstReport, 78
GetInitialSnapshotFileName, 75
GetInitialTime, 77
GetInitialTimeStep, 77
GetInitialWallClock, 76
GetInputFiles, 76
GetMaxTimeStep, 78
GetMinTimeStep, 78
GetName, 75
GetNumPastValues, 76
GetOutputFileName, 76
GetReportCycle, 78
GetRunFileName, 75
GetTimeUnit, 76
GetVariableTimeStep, 78
Reset, 79
SetConsistentInitialCalculation, 79
SetDiagnosticLevel, 77
SetFinalSnapshotFileName, 75
SetFinalTime, 77
SetFirstReport, 78
SetInfiniteFinalTime, 77
SetInitialSnapshotFileName, 75
SetInitialTime, 77
SetInitialTimeStep, 77
SetInitialWallClock, 76
SetMaxTimeStep, 78
SetMinTimeStep, 78
SetNumPastValues, 77
SetOutputFileName, 76
SetReportCycle, 78
SetTimeUnit, 76
SetVariableTimeStep, 79
TInputFiles, 74
TRuntimeControls, 75
ValidateControls, 79
Write, 79
SPARK::XAssertion, 80
~XAssertion, 81
code, 82
message, 81
operator(), 82
operator[], 82
TCode, 81
type, 81
what, 81

where, 82
XAssertion, 81
SPARK::XDimension, 83
XDimension, 84
SPARK::XInitialization, 85
XInitialization, 86
SPARK::XIO, 87
XIO, 88
SPARK::XMemory, 89
XMemory, 90
SPARK::XOutOfRange, 91
SPARK::XOutOfRange
XOutOfRange, 92
SPARK::XStepper, 93
XStepper, 94
SPARK::XTimeStep, 95
SPARK::XTimeStep
XTimeStep, 96
sparkapi.h, 108
sparkmacro.h, 110
NAMESPACE, 111
QUOTE, 111
EMPTY_ARRAY, 111
END_ARRAY, 111
END_INVERSES, 112
END_PROBLEM, 112
START_ARRAY, 111
START_INVERSES, 112
START_PROBLEM, 112
Start
SPARK, 9
START_ARRAY
sparkmacro.h, 111
START_INVERSES
sparkmacro.h, 112
START_PROBLEM
sparkmacro.h, 112
Starting
SPARK::TProblem, 67
StaticStep
SPARK::Problem, 25
Step
SPARK::Problem, 24
StepControlMethod
SPARK::DefaultComponentSettings, 14

TCode
SPARK::XAssertion, 81
TComponent
SPARK::TComponent, 36
TComponentSettings
SPARK::TComponentSettings, 41
TDiagnosticLevel
SPARK::TProblem, 62
Terminate
SPARK::Problem, 23
SPARK::TProblem, 64
TGlobalSettings
SPARK::TGlobalSettings, 48
TInputFiles
SPARK::TRuntimeControls, 74
Tolerance
SPARK::DefaultGlobalSettings, 16
TPreferenceSettings
SPARK::TPreferenceSettings, 57
TProblem
SPARK::TProblem, 63
TRestartFlag
SPARK::Problem, 21
TrueJacobianEvalStep
SPARK::DefaultComponentSettings, 14
TRuntimeControls
SPARK::TRuntimeControls, 75
TState
SPARK::TProblem::TState, 70
TStopTime
SPARK::Problem, 21
TTimeStep
SPARK::Problem, 21
type
SPARK::XAssertion, 81
types.h, 113

Unload
SPARK::Problem, 22

ValidateControls
SPARK::TRuntimeControls, 79
value_type
SPARK::Problem::simulation_parameter, 32
VariableTimeStep
SPARK::DefaultRuntimeControls, 19
VariableType_GLOBAL_TIME
SPARK, 9
VariableType_GLOBAL_TIME_STEP
SPARK, 9
VariableType_INPUT
SPARK, 9
VariableType_NONE
SPARK, 9
VariableType_PARAMETER
SPARK, 9
VariableType_UNKNOWN
SPARK, 9
VariableTypes
SPARK, 8
VARIABLETYPES_L
SPARK, 9

what
SPARK::XAssertion, 81
where

SPARK::XAssertion, 82
Write
SPARK::TComponentSettings, 45
SPARK::TGlobalSettings, 50
SPARK::TPreferenceSettings, 57
SPARK::TRuntimeControls, 79
WriteInstances
SPARK::Problem, 22
WriteStamp
SPARK::TProblem, 68

XAssertion
SPARK::XAssertion, 81
XDimension
SPARK::XDimension, 84
XInitialization
SPARK::XInitialization, 86
XIO
SPARK::XIO, 88
XMemory
SPARK::XMemory, 90
XOutOfRange
SPARK::XOutOfRange, 92
XStepper
SPARK::XStepper, 94
XTimeStep
SPARK::XTimeStep, 96