ProjectionSolver Design

Contents

[Overview 3](#_Toc300931379)

[External Synopsis 3](#_Toc300931380)

[Classes 4](#_Toc300931381)

[Public 4](#_Toc300931382)

[Constraint 4](#_Toc300931383)

[Parameters and AdvancedParameters 4](#_Toc300931384)

[Solution 4](#_Toc300931385)

[Solver 4](#_Toc300931386)

[Variable 4](#_Toc300931387)

[Internal implementation 5](#_Toc300931388)

[Block, BlockVector 5](#_Toc300931389)

[ConstraintVector 5](#_Toc300931390)

[DfDvNode 5](#_Toc300931391)

[GlobalConfiguration 5](#_Toc300931392)

[QPSC 5](#_Toc300931393)

[SimpleStopwatch 5](#_Toc300931394)

[SolverAlgorithm 5](#_Toc300931395)

[ViolationCache 5](#_Toc300931396)

[Wrappers for other internal modules 5](#_Toc300931397)

[ISolverShell 5](#_Toc300931398)

[SolverShell 6](#_Toc300931399)

[UniformOneDimensionalSolver.cs 6](#_Toc300931400)

[UniformSolverVar.cs 6](#_Toc300931401)

[Details of Selected Implementation Areas 6](#_Toc300931402)

[Flow of control: Project 6](#_Toc300931403)

[Equality Constraints 6](#_Toc300931404)

[Iterative recursion 6](#_Toc300931405)

[Constraint Cycles 6](#_Toc300931406)

[Flow of Control: QPSC 7](#_Toc300931407)

[Scaling 7](#_Toc300931408)

[Detailed walkthrough in Qpsc header comments 7](#_Toc300931409)

[Interaction between Solve() and Qpsc 7](#_Toc300931410)

[UpdateVariables 8](#_Toc300931411)

[Testing 8](#_Toc300931412)

[Unit Testing Architecture 8](#_Toc300931413)

[ITestConstraint – the testerInstance 8](#_Toc300931414)

[Overriding default values 9](#_Toc300931415)

[Datafiles 9](#_Toc300931416)

[Creating and Regenerating 10](#_Toc300931417)

[#define VERBOSE in Debug build 10](#_Toc300931418)

[TestConstraints.exe 11](#_Toc300931419)

[Useful Parameters for Test Creation and Debugging 11](#_Toc300931420)

[Existing datafile parameters: ProjectionSolver-specific 11](#_Toc300931421)

[Debugging Tips 11](#_Toc300931422)

[General 12](#_Toc300931423)

[Conditional Breakpoints 12](#_Toc300931424)

[Back up an earlier version’s binaries 12](#_Toc300931425)

[#define VERBOSE in MSAGL 12](#_Toc300931426)

[ProjectionSolver-specific 12](#_Toc300931427)

[Where to set Breakpoints 12](#_Toc300931428)

[SolverFoundation 12](#_Toc300931429)

[Merging with Progression 12](#_Toc300931430)

[Unimplemented areas 13](#_Toc300931431)

[Performance (@@PERF) 13](#_Toc300931432)

[Possible Additional or Modifications to Existing Features (@@DCR) 13](#_Toc300931433)

[Incremental Solving 13](#_Toc300931434)

[Replace VERBOSE with DevTrace and TraceSwitches 13](#_Toc300931435)

# Overview

The Microsoft.MSAGL.ProjectionSolver namespace comprises a family of classes and structures that implement the Quadratic Programming with Separation Constraints algorithm described in “*IPSep-CoLa: an incremental procedure for separation constraint layout of graphs*” by Tim Dwyer et al. (in the enlistment at GraphLayout\MSAGL\Papers\"IpSep\_Cola.pdf"). The QPSC implementation has been extended to include diagonal scaling as in “*Constrained Stress Majorization Using Diagonally Scaled Gradient Projection*” by Tim Dwyer and Kim Marriott (in the enlistment at GraphLayout\MSAGL\Papers\"*Constrained Stress Majorization Using Diagonally Scaled Gradient Projection.pdf*").

This document is intended to be a high-level summary, providing a road map that is fully fleshed out by the detailed descriptions in the code comments.

# External Synopsis

At a high level, the caller obtains a solution using the following steps:

* Create a ProjectionSolver instance. Inside MSAGL, this may be done indirectly by creating an instance of SolverShell.
* Add variables via AddVariable. The primary properties of a variable are its weight and its desired position, which is the position computed by the application for its own purposes. The Solver will obtain a solution which keeps variables as close to their desired positions as possible, with any movement necessary to satisfy the constraints being proportional to the weights of the variables.
* Add constraints via AddConstraint. These constraints consist of two variables and the required space between them, as well as an option for this space to be either the minimum separation required or the exact separation required.
* Add neighbor pairs, if desired, via AddNeighborPair. Neighbor pairs consist of two variables whose changes in positions are to be minimized relative to each other, and the weight of the relationship (which determines how “resistant” the variables are to movement influenced by other variables).
* Create a Parameters object that describes options for controlling the Solver’s operation. Some options are related to speed vs. accuracy tradeoffs, and others are intended to mitigate potentially nonconvergent scenarios. Parameters also contain an AdvancedParameters object which describes more “internal” options for fine-tuning performance in the event such adjustments are useful.
* Call Solve to obtain a Solution object. The solved positions are in Variable.ActualPos, and the Solution object contains information about the process such as the number of iterations required, any limits exceeded, and the number of constraints that could not be satisfied, if any (usually due to cycles).

# Classes

Following are the public and internal classes of the ProjectionSolver. Other than renaming these generally follow the definitions in IpSep\_Cola.

## Public

### Constraint

A Constraint contains the two Variables involved, the required gap, the current violation of the constraint’s separation (which may be negative or zero if satisfied, else positive), and some members to carry information during the constraint-traversal phase of Solver.Project(). In other words, a Constraint defines the required relative positions of its two Variables as one of the following:

* variable1 + gap <= variable2; these are satisfied with the minimum movement, i.e. to minimize the goal function (x-i)^2. Note that the gap can be negative, which means “variable1 must be less than or no more than abs(gap) greater than variable2”.
* variable1 + gap == variable2

This is different from a neighbor pair, where we try to keep the neighbors as close as possible while satisfying all constraints, i.e. to minimize a goal function of the form (x1-x2)^2.

### Parameters and AdvancedParameters

These control execution of the Solver, setting tolerances, epsilons, and limits on time or iterations. These do not usually need to be modified, especially AdvancedParameters.

### Solution

Returns additional information about the solution (the actual variable positions are returned in Variable.ActualPos), such as the number of iterations required and any execution limits that were exceeded.

### Solver

This is the main class to which Variables, Constraints, and Neighbors are added, and which contains the Solve() method that returns a Solution.

### Variable

This adds a variable (usually derived from a node on the graph). It is one side of a Constraint or of a Neighbor pair. On input it has its initial position (DesiredPos) and weight; after ProjectionSolver.Solve, it contains the result position (ActualPos).

## Internal implementation

### Block, BlockVector

A Block defines an active set of constraints; that is, a set of constraints that are transitively satisfied. A BlockVector contains that set and provides a Debug validation procedure.

### ConstraintVector

ConstraintVector maintains the list of all constraints, providing a central point for constraint-wide operations (such as maintaining the index of active/inactive constraints). It also serves as a convenient central location for various block-global operations related to constraints such as managing the DfDvNode recycling for “iterative recursion” and maintaining the number of unsatisfiable constraints detected.

### DfDvNode

This is a node on the DfDv “recursive iteration” queue and in the ConstraintVector’s DfDvRecycleStack.

### GlobalConfiguration

This defines global parameters for all ProjectionSolvers. Currently only one internal constant is defined.

### QPSC

Stands for Quadratic Programming of Separation Constraints, this drives the gradient-projection aspect of the ProjectionSolver. By default it is used only if neighbor pairs have been defined. It can be force by AdvancedParameters if neighbors have not been added; this is usually not useful but it may help with convergence if there is a wide range of variation in constraint weights.

### SimpleStopwatch

A simple implementation of the .NET Stopwatch class, supplied because Silverlight before 4.0 did not have a Stopwatch class. Unfortunately Silverlight 4.0 includes it only for the phone; it is hoped that Silverlight 5.0 will include it for all platforms and this class can be removed.

### SolverAlgorithm

Indicates which Solver algorithm was used to obtain the Solution.

### ViolationCache

Caches the top N violated constraints encountered on the enumeration of all inactive constraints. After the highest violation is adjusted, this cache is cleared of constraints involving the affected blocks. If any constraints remain, this can avoid a full enumeration of all active constraints on the next pass.

## Wrappers for other internal modules

### ISolverShell

Defines a simplified interface around the ProjectionSolver to make it easier for other modules (currently RectilinearEdgeRouting, specifically the Nudger, and LayeredLayout’s ConstrainedOrdering) to create a simple solver. Also provides an underlying map of app variables to solver variables so the app does not need to know about the solver variables.

### SolverShell

This is currently the only implementation of ISolverShell. In addition to providing the features mentioned under that topic, SolverShell also implements a loop which determines if any “fixed” (high-weight) variables have been moved (past a certain tolerance) by the solution, and if so, modifies constraints and re-executes the ProjectionSolver. This feature is used by RectilinearEdgeRouter’s Nudger.

### UniformOneDimensionalSolver.cs

This is a wrapper around SolverShell, for RectilinearEdgeRouter’s Nudger.

### UniformSolverVar.cs

This is a variable in a UniformOneDimensionalSolver.

# Details of Selected Implementation Areas

The code contains detailed comments, so this section presents a high-level overview that provides context to complement the low-level comments.

## Flow of control: Project

This is very close to the Ipsep\_Cola paper. Starting at Solve(), the flow is very well commented so will not be repeated here. However a few things are not discussed in the paper so are described here.

### Equality Constraints

Equality constraints are those for which the gap condition is x + gap == y rather than x + gap <= y. These constraints are solved first and are never split.

### Iterative recursion

The IpSep\_Cola paper’s pseudocode uses recursion. In the case of a long chain of dependencies, this can easily exceed stack allowance (see, for example, ProjectionSolverTests.StartAtZero10000). Therefore, a typical Stack<DfDvNode> approach is used, with the next Variables in the desired direction (left-to-right or right-to-left) being pushed onto the Stack<> and then the top one selected for deeper “recursive” exploration of its constraints.

To avoid exercising the GC unnecessarily, DfDvNodes that are popped from the queue are pushed onto the ConstraintVector.DfDvRecycleStack, up to a maximum count which is large enough for the vast majority of cases.

### Constraint Cycles

It is possible to add constraints that cannot be satisfied because they create a cycle, such as a < b and b < a, or a < b < c and a > c. In practice these can be much longer and more indirect, and are rarely encountered, usually as the result of an application error. Equality constraints can also generate cycles.

Cycles can only be “created” during Block.Expand. The only way to get a cycle is to add a constraint where both variables are already connected by an active tree, so therefore they must already be in the same block; therefore the cycle can't be created by MergeBlocks. If there is a forward non-equality constraint in the path, then that constraint will be deactivated and its variables moved, so there is no cycle. So the only condition for a cycle is that Expand finds no forward non-equality constraint.

Cycles are not actually allowed to be created; instead, ConstraintVector.NumberOfUnsatisfiableConstraints is incremented, the constraint’s IsUnsatisfiable property is set, and the constraint is subsequently ignored. Look for ProjectionSolverTests with “Cycle” or “Unsatisfiable” in the name.

## Flow of Control: QPSC

This implementation was initially done from the Ipsep paper and then modified with the Diagonal Scaling paper, with some modifications for efficiency and to fit into the existing implementation. Starting at Solve(), the flow outside of Qpsc is as for the non-Qpsc path; this is very well commented so will not be repeated here. A few details specific to Qpsc that are not directly as in the papers are described here.

### Scaling

Scaling is the only reason we use the QPSC approach, and we have found this to be of benefit only for neighbor pairs.

It was initially thought that scaling would help improve convergence performance related to widely divergent variable weights with constraints even in the absence of neighbor pairs, but this turns out not to be a significant factor without neighbor pairs present. As discussed above, neighbor pairs consist of two variables whose changes in positions are to be minimized relative to each other, and the weight of the relationship (which determines how “resistant” the variables are to movement influenced by other variables). For these cases QPSC can dramatically improve performance as well as ensuring convergence (see for example ProjectionSolverTests Test\_Qpsc\_\*\_Gap\_Heavy\_Neighbor for cases where it does not converge well without scaling).

### Detailed walkthrough in Qpsc header comments

The Qpsc class contains a large meta-comment section that describes the paper’s assumptions and reasoning in considerably more detail and with much finer granularity of pseudocode and matrix illustrations. The Qpsc implementation was initially done from the Ipsep paper and then modified with the Diagonal Scaling paper, with some modifications for efficiency and to fit into the existing implementation. These are all discussed in this section.

### Interaction between Solve() and Qpsc

solver.SolveQpsc drives the Qpsc process as follows:

* Initialization:
  + Qpsc qpsc = new Qpsc(numVariables);
  + foreach (variable in (foreach block))
    - qpsc.AddVariable(variable)
  + qpsc.VariablesComplete()
* Per iteration:
  + if (!qpsc.PreProject()) break;
  + solver.SplitBlocks()
  + solver.Project()
  + if (!qpsc.PostProject()) break;
* Done:
  + qpsc.ProjectComplete()

## UpdateVariables

This is called when one or more Variables have had their InitialPosition changed after adding them to the solver, or between calls to Solve(). Although the name is "UpdateVariables", that's just to encapsulate internals from the caller; this really is updating the blocks after the variables have already been updated one at a time by the caller.

This is not required to be called when constraints have been re-gapped; Solve() handles that automatically. This is what Rectilinear’s Nudger does, so it does not need to call UpdateVariables.

# Testing

The following are common to ProjectionSolver and OverlapRemoval, and they use the same primary test application, TestConstraints.exe.

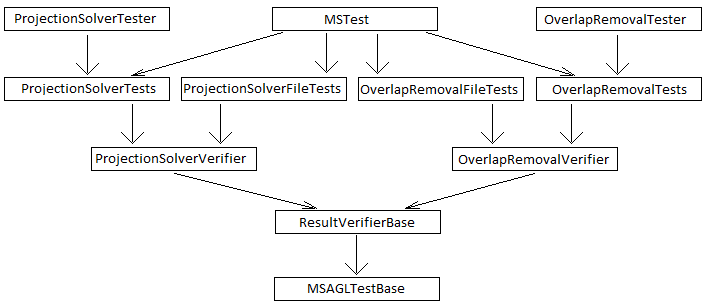
## Unit Testing Architecture

Testing is based upon the MSTest Unit-Testing architecture via the MsaglTestBase class. The core set of testing utilities for Rectilinear is ProjectionSolverVerifier which inherits from ResultVerifierBase (a common base for ProjectionSolver and OverlapRemoval), which in turn inherits from MsaglTestBase. Following that we split by whether we are doing programmatic testing (test methods containing code) via ProjectionSolverTests or file-based testing (test methods that load and execute a test data file) via ProjectionSolverFileTests, both of which inherit from ProjectionSolverVerifier. Additionally, TestConstraints.exe extends MsaglTestBase (for RedirectDefaultTraceListener; the MsaglTestBase here is separate from that in the ProjectionSolverVerifier inheritance chain).

### ITestConstraint – the testerInstance

TestConstraints uses the first argument, “proj” or “olap” or “sf”, to instantiate its testerInstance member; this class implements ITestConstraint which allows the rest of TestConstraints.exe to operate without knowing what particular class is being tested. It uses the testerInstance for a few common utility operations, and to Reflect on it to run one or more tests specified on the TestConstraints command line.

The testerInstance is one of the following top-level classes; it sits directly on top of the programmatic unit tests for that module, which is also what MSTest communicates with. The inheritance hierarchy is shown here as well as who is called by MSTest.



(SolverFoundationTester is not shown because it is a separate class.)

Thus the testerInstance is a thin layer over the (class)Tests which is what MSTest talks to, so both TestConstraints and MSTest can Reflect to invoke test methods.

Tests come in two varieties: programmatic and file-based. Programmatic tests are those which set up the variables and constraints and expected results in the method code, and execute the test. File-based tests have the variables, constraints, and expected results in the data file.

Programmatic tests are run by TestConstraints Reflecting on its testerInstance to find the named method and invoking it. Of course, a similar approach is used by the MSTest implementation.

File-based tests are run via TestConstraints with the -file parameter, which gathers all filespecs into a list and then calls testerInstance.ProcessFile. TestConstraints does not know about the \*FileTests classes; it talks directly to the files. For MSTest, file-based tests are run by ProjectionSolverFileTests and OverlapRemovalFileTests which contain methods that invoke specific files; these methods bottom out in the same ProjectionSolverVerifier or OverlapRemovalVerifier methods that are called by testerInstance.ProcessFile, as shown in the above diagram.

### Overriding default values

This is fairly straightforward. TestConstraints recognizes a number of parameters that allow defaults to be overridden. Look at the ShowUsage() output to see what you want and then trace into it; this will set a value in TestConstraints.exe, testerInstance, or a static ResultVerifierBase member.

### Datafiles

ProjectionSolver and OverlapRemoval use their own regex-based format for datafiles, located in the MSAGLTests\Resources\Constraints\ProjectionSolver\Data or MSAGLTests\Resources\Constraints\OverlapRemoval\Data directories. Files are parsed with regular expressions by RectFileReader. The regexes allow an extremely condensed format, which reduces the copy time and space requirement during Unit tests.

#### Creating and Regenerating

Initial creation of a datafile is done by running TestConstraints.exe with the -CreateFile argument. In many cases this is done as the result of a failed test from a run of TestConstraints with a given commandline containing -ToFailure, which continuously writes to the output file until a failure is encountered, if any. This datafile is then used to repro the problem and verify the fix. Once fixed, the test is checked in to guard against future regressions. For example:

* Overnight run:
  + TestConstraints.exe proj -CreateFile 100 3 ToFailure c:\temp\vars100\_constraints3.txt
* If there is any failure, the file will remain. If the run was interrupted, such as ctrl-c or machine restart, the file may be empty, which indicates no error. If an error is found, perform the following steps.
* Copy the failing file to the MsaglTest\Resources\Constraints\ProjectionSolver\Data directory as <newFileName>. “tf add” it from the commandline and reload the project, or add it in Visual Studio and then edit MSAGLTests.csproj and clean up the mess of specific filenames VS added; we have \*.txt already copied so we don’t need the individual file names.
* Create a new test in ProjectionSolverFileTests that is named exactly the same as the newFileName; the tests use the name of the test method as the filename to load.
* Verify the failure reproduces.
* Fix the failure and verify the test passes
* Check in

For OverlapRemoval, use “olap” for the first argument to TestConstraints and whatever OverlapRemoval-specific parameters are desired, and otherwise replace “ProjectionSolver” with “OverlapRemoval” in the above steps.

TestConstraints supports a -ReCreateFile argument which reads the file for its variable, constraint, and neighbor definitions and other parameters, then reruns it to regenerate results.

It is advisable to store off a recent build of TestConstraints in three flavors, Debug, Debug with VERBOSE enabled in the MSAGL project (or replace both of these with DebugDevTrace if VERBOSE is replaced by DEVTRACE), and Release, and then run the previous and current versions to compare and judge diffs. This is also useful for debugging.

## #define VERBOSE in Debug build

The DEBUG build configuration supports a VERBOSE preprocessor directive in the production MSAGL side, which calls Console.WriteLine with a lot of useful information for determining where various values are derived. For example, you can track a particular variable as it is added to blocks and/or its position changed. It is possible to selectively enable VERBOSE in some files, but for the most part, there are enough cross-file dependencies that this is not feasible, so one is left with either enabling the individual VERBOSE blocks (and possibly others that are required, and then others that are required by that…) or just turning it on for the full build. Unfortunately this often creates a huge file and windiff is not so good on files of that size. See below for a DCR that would replace this with the DebugDevTrace build which provides much better control.

## TestConstraints.exe

This is the test app for both ProjectionSolver and OverlapRemoval; which one is to be done, including the format of any datafile to be loaded or created, is determined by the first parameter, which must be a case-insensitive match for one of:

* Proj or ProjectionSolver
* Olap or OverlapRemoval
* SF or SolverFoundation (more on this below)

### Useful Parameters for Test Creation and Debugging

The settings that modify ProjectionSolver or OverlapRemoval Parameters or AdvancedParameters properties are good candidates for test creation. Also look at all CreateFile (ShowCreateFileUsage()) parameters; these were designed to support exercising a wide range of parameter values.

Looking over the filenames in the Data directory will give you an idea of what kinds of tests have been run (or at least what showed issues that had to be fixed).

#### Existing datafile parameters: ProjectionSolver-specific

The checked-in datafile tests encode their creation parameters in the filename; for example the ProjectionSolver file:

* Cycles\_Vars100\_ConstraintsMax10\_EqualityConstraints\_PosMax1M\_GapMax100K\_WeightMax10K\_Cycles10.txt

Translates left-to-right to a random generation (the seed is in the file) of a file with:

* The initial Cycles just identifies the main focus of the testfile run
* 100 variables
* Up to 10 constraints generated per variable (so the average is between 1 and 10, so on average there will be 5.5 constraints per variable)
* Randomly make some constraints equality rather than “less or equal” (these are identified in the file with an “=” before the gap)
* Random initial position up to 1 million
* Random gap up to 100,000
* Random variable weights up to 10,000
* Try to create 10 cycles (by reversing the direction of 10 of the generated constraints)

## Debugging Tips

Here are some tips for debugging failures in UnitTests and/or TestConstraints.

### General

#### Conditional Breakpoints

Be careful of floating-point rounding. Sometimes the display will round and you won’t get equivalence. If your breakpoint refuses to fire, you may need to use something like “variable.ActualPos > 22.4 && variable.ActualPos < 22.5”.

#### Back up an earlier version’s binaries

It can be very useful in detecting regressions to have backed up a build from before changes were made. You can shelve current changes, undo them, rebuild, store off, and then reapply the shelveset.

In particular it can be very useful to run two versions with VERBOSE enabled in MSAGL and then diff the output (piped to files). This can be quite large but can save a lot of effort.

#### #define VERBOSE in MSAGL

By defining VERBOSE in MSAGL, you will be able to step through and when arriving at a desired breakpoint, look at the console output. This is especially useful with two monitors.

### ProjectionSolver-specific

#### Where to set Breakpoints

The main Block methods, Bllock.Split and Solver.MergeBlocks, are good points. Try using a conditional on Block ID or some other characteristic. The main QPSC functions called during interaction with Solve() are also good if neighbor pairs are involved.

## SolverFoundation

The Microsoft SolverFoundation is a general-purpose solver. The ProjectionSolver’s algorithm is suitable for our particular requirements and is much faster than SolverFoundation as the number of variables and constraints increases. Because we should get the same results, TestConstraints has a third “first parameter”, “sf”, which runs the SolverFoundation over the created variables and constraints, or the input datafile (which is then automatically verified). A copy of the SolverFoundation .dll is checked in with TestConstraints.

# Merging with Progression

“Progression” is the name for the Dev11 Visual Modeling tool, which consumes RER (the old Tuvalu group is now in Progression). There are 4 directories that should remain in sync when merging to Progression:

* GraphLayout\MSAGL\Core\ProjectionSolver
* GraphLayout\MSAGLTests\Constraints\ProjectionSolver\* (or just the entire directory)
* GraphLayout\MSAGLTests\Infrastructure\Constraints
* GraphLayout\ MSAGLTests\Resources\Constraints\ProjectionSolver\Data

# Unimplemented areas

## Performance (@@PERF)

There are a couple of potential performance improvements marked by @@PERF in the code, with more detailed discussion. These areas could be profiled to evaluate how much time might be saved by implementing them. Briefly these are:

* Tweaks to MatrixVectorMultiply in QPSC
* Maintaining a Constraint count in each Block to improve accuracy of the ViolationCache cutoff.

## Possible Additional or Modifications to Existing Features (@@DCR)

### Incremental Solving

Currently all variables, constraints, and neighbor pairs must be added before the first Solve(); this membership cannot subsequently change. In other words, it is not possible to call Solve() with one set of variables and constraints, then add variables and new constraints, and re-Solve () taking advantage of the Block structure from the previous Solve().

This is different from UpdateVariable, which allows variable positions to be modified before another call to Solve(). It is also not the same thing as allowing constraints to be re-gapped (this capability is used by Rectilinear Edge Router’s Nudger). Also, it is different from the incremental Project() calls made by QPSC.

If it becomes desirable to implement this, the two initial necessary changes are marked in Solver.cs by @@DCR “Incremental Solving”. After implementing those, try to call Project() again. This should not be done when neighbor pairs are present, because the new members will not be present in the QPSC matrix. Also, this will need additional work to handle removal of variables or constraints.

## Replace VERBOSE with DevTrace and TraceSwitches

There is a VERBOSE preprocessor directive that when activated prints a lot of very useful information. However, as a preprocessor directive it requires a rebuild (or new project configuration) to enable it, and it is not possible to tell it to send out only a certain level of output; this has to be done by selectively enabling it at various locations. Additionally, there are too interdependencies between the files now for per-file VERBOSE enabling, because constructor parameters change and so on.

This needs to change to use the same DevTrace framework that Rectilinear Edge Routing does. This will enable dynamically switching the trace level and the level of detailed verification done in extended production-side “assert-like” code.