OverlapRemoval Design

Contents

[Overview 3](#_Toc300931340)

[External Synopsis 3](#_Toc300931341)

[Classes 4](#_Toc300931342)

[Public 4](#_Toc300931343)

[BorderInfo 4](#_Toc300931344)

[ConstraintGenerator 4](#_Toc300931345)

[OverlapRemovalCluster 4](#_Toc300931346)

[OverlapRemovalNode 4](#_Toc300931347)

[OverlapRemovalGlobalConfiguration 4](#_Toc300931348)

[OverlapRemovalParameters 5](#_Toc300931349)

[RectangularClusterBoundary 5](#_Toc300931350)

[Internal 5](#_Toc300931351)

[OverlapRemovalEvent 5](#_Toc300931352)

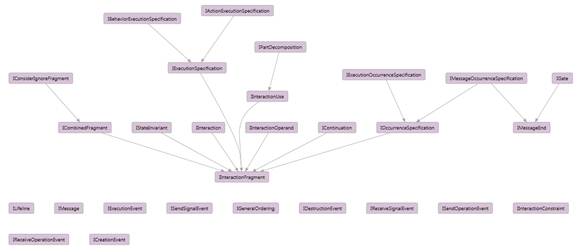
[OverlapRemovalScanLine 5](#_Toc300931353)

[Details of Selected Implementation Areas 5](#_Toc300931354)

[Clusters 5](#_Toc300931355)

[ClusterHierarchies 6](#_Toc300931356)

[Defer To Vertical 7](#_Toc300931357)

[ Generating Constraints 7](#_Toc300931358)

[Iterative recursion 8](#_Toc300931359)

[Testing 8](#_Toc300931360)

[Read this section in the ProjectionSolver document 8](#_Toc300931361)

[OverlapRemoval-specific considerations 8](#_Toc300931362)

[Existing datafile parameters 8](#_Toc300931363)

[Where to set breakpoints 9](#_Toc300931364)

[Merging with Progression 9](#_Toc300931365)

[Unimplemented areas 9](#_Toc300931366)

[#if NOTNEEDED\_FXCOP 9](#_Toc300931367)

[Performance (@@PERF) 9](#_Toc300931368)

[Possible Additional or Modifications to Existing Features (@@DCR) 10](#_Toc300931369)

[ClusterContainmentOnly 10](#_Toc300931370)

[Precalculate Cluster Sizes 10](#_Toc300931371)

[Incremental Generation 10](#_Toc300931372)

[Add weights to the Defer-To-Vertical calculation 11](#_Toc300931373)

[Additional verifications 11](#_Toc300931374)

[Enforce that clusters have only one parent 11](#_Toc300931375)

[Enforce that a node is in only one cluster per hierarchy 11](#_Toc300931376)

# Overview

The Microsoft.MSAGL.OverlapRemoval namespace comprises a family of classes and structures that generate non-overlap constraints to provide separation between variables, as well as to ensure that variables are appropriately placed within one or more clusters if specified. Our implementation is based upon “Fast Node Overlap Removal” by Dwyer et al. and follows the paper very closely.

OverlapRemoval has a dependency upon Microsoft.MSAGL.ProjectionSolver, which exists together with its design document at a sibling level of the enlistment. Additionally, the testing (both TestConstraints.exe and the MSTest-based Unit Testing) is tightly coupled with the ProjectionSolver, so it is highly recommended to read that document before this one; a number of sections of this document merely present OverlapRemoval-specific aspects and specializations of the common infrastructure.

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

The main class for OverlapRemoval is a ConstraintGenerator which is tightly coupled with an instance of the ProjectionSolver. At a high level, the caller obtains and solves a set of separation constraints using the following steps:

* Instantiate a ProjectionSolver.Solver and call its AddVariables method for each desired variable.
* Optionally, generate an initial set of constraints manually (e.g. user-defined or simple hierarchy) and load them into the ProjectionSolver.Solver. These will be evaluated along with the constraints generated by OverlapRemoval. There is no way to add a constraint directly to the ConstraintGenerator.
* For each axis (that is, horizontal and vertical direction):
  + Instantiate an OverlapRemoval.ConstraintGenerator and iterate the ProjectionSolver.Solver's variables (or your own structure into which you've stored the Variables returned from Solver.AddVariable), passing each to the ConstraintGenerator's AddNode method, with the perpendicular (vertical) coordinates as well (for event generation and least-movement-direction determination). This returns the OverlapRemovalNode.
  + Optionally, call AddCluster and pass the returned Cluster when calling AddNode.
  + Pass the ProjectionSolver.Solver and optionally a ProjectionSolver.Parameters object to OverlapRemoval.ConstraintGenerator.Generate to cause the overlap-removal constraints to be generated into it.
  + Call OverlapRemoval.ConstraintGenerator.Solve(), optionally passing the same ProjectionSolver.Parameters object used for Generate(). This returns the Solution object from ProjectionSolver.Solver.Solve() which provides information about the result. The updated position is in the OverlapRemovalNode returned from AddNode, from which the caller can update its own variables as appropriate.

# Classes

## Public

### BorderInfo

This structure specifies information for one of the four borders of a rectangular cluster. It carries the InnerMargin, FixedPosition (which is only as “fixed” as the weight is high relative to other borders and nodes).

InnerMargin is named so to clarify that it is inside the cluster; this distinction is necessary because WPF uses “margin” to refer to space outside a node, whereas we use “padding” for that space.

### ConstraintGenerator

This is the driving class of OverlapRemoval; one instance is created for each axis (horizontal or vertical) in which overlaps are to be removed). ConstraintGenerator is fairly simple: its primary methods are to add nodes, add clusters, generate constraints, and act as a wrapper around ProjectionSolver.Solve.

### OverlapRemovalCluster

A Cluster inherits from a Node and provides a container to remove overlaps with nodes that are not its children (with some additional complexities for nested nodes and multiple Cluster Hierarchies). See the discussion of Cluster and ClusterHierarchy for more detailed information.

Cluster processing extends the existing constraint-generation mechanism by processing a Cluster as a normal Node (which it inherits from), adding a pair of fake variables (corresponding to its borders along the primary axis) to the Solver for it. This allows us to generate Constraints between Nodes and Clusters at the same level (i.e. within the same Cluster). Clusters may contain Nodes which are other Clusters; these form a tree starting at the root of a ClusterHierarchy. See the description of ClusterHierarchies for more information.

Additional detailed information is commented in OverlapRemovalCluster.cs, especially in the large meta-comment at the top of the class.

### OverlapRemovalNode

This corresponds to a node on the layout, which contains parallel and perpendicular positions and a ProjectionSolver variable from which it updates its parallel position to the final solved position.

### OverlapRemovalGlobalConfiguration

Provides default constants for weights and border widths.

### OverlapRemovalParameters

This contains members that control execution of the OverlapRemoval components (currently only regarding whether to resolve overlaps horizontally or vertically), as well as carrying through a ProjectionSolver Parameters object for the actual solving of the generated constraints.

### RectangularClusterBoundary

This is not used by core OverlapRemoval; instead it is a utility class for IncrementalLayout which contains information about a rectangular cluster boundary, if any, for use in overlap avoidance. When the OverlapRemovalCluster is created, the individual fields of the RectangularClusterBoundary are passed to the OverlapRemovalCluster constructor.

## Internal

### OverlapRemovalEvent

This is carried in the OverlapRemovalScanLine and indicates the position at which a node (or a cluster, which is derived from a node) opening or closing border is encountered.

### OverlapRemovalScanLine

This maintains the sweep line through the nodes and an ordered tree of the currently open nodes to enable traversal to the neighbors to either side.

# Details of Selected Implementation Areas

The implementation closely follows the pseudocode in “Fast Node Overlap Removal”, for the generate\_Cno and get\_left\_nbours procedures (get\_right\_nbours is the same as get\_left\_nbours but is to the other side). This flow is well-commented in the code and will not be discussed further here.

Following are some areas that are not in the paper.

## Clusters

A Cluster inherits from a Node and provides a container to remove overlaps with nodes that are not its children (with some additional complexities for nested nodes and multiple Cluster Hierarchies). More fully, the Cluster is usually a dynamically-sized Node which obtains its size from the solved (overlap-removed) positions of its child nodes, and then functions as a Node for purposes of overlap removal at the level of its siblings.

Cluster processing extends the existing constraint-generation mechanism by processing a Cluster as a normal Node (which it inherits from), adding a pair of fake variables (corresponding to its borders along the primary axis) to the Solver for it. This allows us to generate Constraints between Nodes and Clusters at the same level (i.e. within the same Cluster). Clusters may contain Nodes which are other Clusters; these form a tree starting at the root of a ClusterHierarchy. See the description of ClusterHierarchies for more information on this topic.

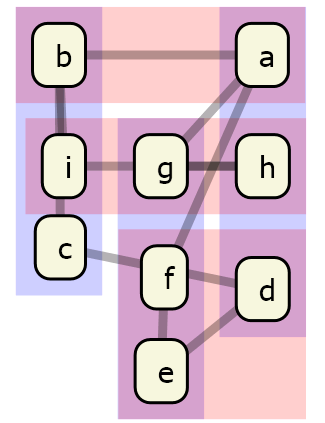
A node may be in multiple Clusters, but in only one Cluster per ClusterHierarchy.

Additional detailed information is provided in comments in OverlapRemovalCluster.cs, especially in the large meta-comment at the top of the class.

## ClusterHierarchies

A ClusterHierarchy is simply a Cluster (there is no ClusterHierarchy class). A ConstraintGenerator contains always at least one hierarchy, the DefaultClusterHierarchy, which is created in the ConstraintGenerator constructor. All nodes live in a ClusterHierarchy; AddNode and AddCluster pass the appropriate cluster (DefaultClusterHierarchy, a ClusterHierarchy created by AddCluster with a null parent Cluster, or a child cluster of one of these clusters). Thus there is no concept of a single "Root Cluster"; rather, each cluster in clusterHierarchies is the root of a separate hierarchy.

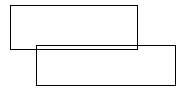
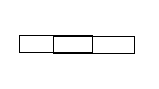
By allowing multiple cluster hierarchies, we can control which nodes and clusters are allowed to overlap, because overlaps are only enforced on a per-ClusterHierarchy basis; non-overlap constraints are not generated for clusters (or their children) in different ClusterHierarchies. For more implementation details of this see the header comments in OverlapRemovalCluster.cs. Here is a prettier picture of the “Layer Cake”:



(This was generated from the old FastIncrementalLayoutExample program on LayerCake.xml; currently VS doesn’t actually use this in the UI).

## Defer To Vertical

For two nodes positioned such that the vertical overlap is much smaller than the horizontal, it can result in much less overall movement to allow the horizontal pass to ignore the overlap and let the vertical pass remove it. There are two ways of determining this: absolute amount of overlap, or the proportion of horizontal overlap size relative to the combined horizontal node sizes vs. the vertical-axis overlap relative to the combined vertical node sizes.

For the left-hand pair of nodes above, the vertical resolution is clearly desired. In the middle pair, the vertical overlap is smaller than the horizontal overlap, but when normalizing it as a proportion of the node sizes in the axis of overlap, the horizontal overlap is smaller (about 1/3 the combined horizontal node sizes, vs. about equal to the combined vertical node sizes). Thus if the OverlapRemovalParameters.ConsiderProportionalOverlap property is set, then the overlap resolution of the second pair of nodes will be deferred to the vertical pass. The reason for this heuristic is that Visual Studio layout often has wide nodes to accommodate labels, and this tends to give a better balance of horizontal and vertical overlap when all nodes are wide (or tall, as in the right-hand pair of nodes above).

Below is a more realistic illustration that uses constraints on each edge to require the edges to point down, as well as non-overlap constraints. Since we want it to be spread out more horizontally to show the layering, the “proportional” heuristic will make it tend to resolve wide node overlap horizontally rather than deferring to vertical.

## cid:image003.jpg@01CC590F.050E5A70 Generating Constraints

To generate constraints, create a ConstraintGenerator and ProjectionSolver for each axis (horizontal and vertical), processing the horizontal before the vertical. As described above, “processing” means:

* Add constraints directly to the Solver, if desired
* Add nodes and clusters to the ConstraintGenerator, usually recording the returned object for use in app positional updates after solving
* Call ConstraintGenerator.Generate, which adds constraints to the solver
* Call ConstraintGenerator.Solve()

Between the horizontal and vertical passes it is important to update BorderInfos with the solved horizontal positions so the ConstraintGenerator has the updated values for these (as positionP and sizeP, where the “P” is “Perpendicular”).

### Iterative recursion

The FastNodeOverlapRemoval paper’s pseudocode uses recursion. In the case of very deeply nested clusters this may exceed stack allowance. Therefore, a typical Stack<ClusterItem> approach is used, with the current Cluster’s child Clusters being pushed onto the Stack<> and then the top one selected for deeper “recursive” exploration of its child Clusters.

There are actually two full “recursive” passes, first to call GenerateWorker and then to call SqueezeNonFixedBorderPositions. Both are run out of Generate, passing a delegate that does the actual work to ProcessClusterHierarchy, which does the “recursion”.

Deep cluster hierarchies are not expected and therefore no special GC-sparing recycling of stack items is done.

# Testing

Much of this is common to ProjectionSolver and OverlapRemoval, and they use the same primary test application, TestConstraints.exe.

## Read this section in the ProjectionSolver document

For the most part this is the same as in the Testing section of the ProjectionSolver design document, with a few OverlapRemoval-specific items as noted below.

## OverlapRemoval-specific considerations

### Existing datafile parameters

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

* Clusters\_Vars500\_ConstraintsMax3\_PosMax1M\_WeightMax10K\_Margins100\_ClustersDefault\_FixedBordersRandom.txt

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

* The initial Clusters just identifies the main focus of the testfile run
* 500 variables
* ConstraintsMax is ignored because OverlapRemoval generates the constraints itself (this was mistakenly added to the filename)
* Random initial node positions up to 1 million
* Random variable weights up to 10,000
* Random margins (padding inside the cluster) up to 100
* Randomly generate the default amount of clusters, which is currently one cluster for each 10 variables.
* Create fixed cluster borders at random

### Where to set breakpoints

The only really tricky part of the code is in cluster processing (each node is in a cluster, even if it’s the DefaultClusterHierarchy), so most breakpoints will be in OverlapRemovalCluster.cs, probably in either GenerateWorker or GenerateFromEvents. It’s best to just walk through one of the programmatic tests in OverlapRemovalTests.cs and get a feel for the code before making any changes or trying to diagnose a newly-discovered bug.

# 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\Geometry\OverlapRemoval
* GraphLayout\MSAGLTests\Constraints\OverlapRemoval\* (or just the entire directory)
* GraphLayout\MSAGLTests\Infrastructure\Constraints
* GraphLayout\ MSAGLTests\Resources\Constraints\OverlapRemoval\Data

# Unimplemented areas

## #if NOTNEEDED\_FXCOP

OverlapRemovalNode.cs has a section of method overrides that are #ifdef’d out by this; they are recommended by FxCop, but as the code comments indicate, the reason for the FxCop warning does not apply here since reference equality is adequate, and having them entails a perf hit due to ==/!= becoming a non-inlined function call in some cases. Assuming this meets requirements for FxCop, this section can be removed.

## Performance (@@PERF)

There are a few sections of code marked @@PERF in comments, indicating some local optimizations that may be possible for performance. Each of these should be profiled before and after to determine if the change is worthwhile.

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

### ClusterContainmentOnly

This was just an idea to do a solving pass that only calculated child node positions relative to cluster boundary positions; in other words it would not remove overlaps between nodes in a cluster (or between clusters). This would allow the user to drag nodes without bumping other nodes, but would preserve cluster containment.

### Precalculate Cluster Sizes

We process clusters depth-first, going to the lowest level and generating constraints there before generating them in the parents. This lets us know how large the child clusters will be based upon initial node positions, but we do not know how close this will be to the final node positions after its internal nodes (or child clusters) have been moved to their solved positions. Knowing these final positions could give better “Defer to vertical” evaluation for overlaps involving child clusters.

Fixing this would probably require a per-cluster Solver to solve the cluster’s positions; probably just a null Solver being passed to GenerateWorker would indicate this is to be done. This would apply in the first, highly-overlapped call of a possibly iterative call sequence; on subsequent calls, when there is expected to be relatively little overlap, then we want to use a single global solver to find the global solution, which may involve moving enough blocks around that we can shrink the cluster rather than keep it at a fixed size.

If this were to be implemented we would want to test situations with high horizontal overlap to see if this makes a difference; the needs-improvement case is that it generates a wider/flatter graph than it should. Also, if we do this, we'll need to handle App/User-generated constraints that cross clusters; although the transitivity across the cluster border "fake nodes" would handle simple separation, the required gap may be larger than we would generate by default. Currently, FastIncrementalOverlap handles most of this through MakeFeasible() (and the ClusterContainmentOnly pass if that is implemented).

#### Incremental Generation

If we implement “Incremental Solving” in ProjectionSolver then we can compare the OverlapRemovalNode’s solver Variable.Pos/Size (as of last solution) to the current OverlapRemovalNode.Pos/Size to know what needs to be moved. In fact we may be able to do this without “Incremental Solving” by using Solver.UpdateVariables.

This will also require no longer nulling the Variable in Node.UpdateFromVariable and adding an mapping from OverlapRemovalNode to its parent clusters, so the operation can bubble up (the caller will have to call a new ConstraintGenerator.SolveIncremental() method so we know to do this, or perhaps we can determine from the passed solver that this is an incremental solution). E.g. a node is moved within the cluster; we solve the cluster and if its size hasn't changed we do nothing, else we bubble up one Cluster level and repeat. This would also benefit from precalculating cluster sizes, which would provide some of this bubble-up implementation itself.

### Add weights to the Defer-To-Vertical calculation

Currently the deferral from horizontal to vertical resolution considers only the overlap size and possibly the node size, but does not consider node weights. In some cases, adding this may lead to less movement of high-weight nodes.

### Additional verifications

Following are some additional verifications that could be added to the code (along with appropriate unit tests) to guard against programmatic error by the caller.

#### Enforce that clusters have only one parent

A Cluster can be in only one ClusterHierarchy, which is a subset of the requirement that it can have only one parent. We could add a cluster.parentHierarchy to enforce this.

#### Enforce that a node is in only one cluster per hierarchy

We could keep a node->hierarchyParentList hash and use cluster.parentCluster to traverse to the hierarchy root to verify that the node is in one cluster per hierarchy.