Listing 1: Experiment

```

//
// ExpiFramework
// Excperiment.cs
//
// Copyright © 2022 Nikolai Tiunin. All rights reserved.
//

using NecProblemFramework;
using PAASolveFramework;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Numerics;

namespace Expi
{
    class Experiment
    {
        string nec;
        string package;
        string solver;
        double theta;
        double phi;
        string name;
        string compare;
        bool draw;
        NecUtilities.In.Parser parser = new NecUtilities.In.Parser();
        NecProblemBuilder problemBuilder = new NecProblemBuilder();
        ProblemWritter problemWritter = new ProblemWritter();
        Dictionary<string, object> parameters;

        string[] PreferredOutputOrder = new string[] {
            "Nec", "Theta", "Phi", "p", "q", "force", "z", "de_status", "
                de_record_iteration", "de_time", "grad_time"
        };

        public class PackageResult
        {
            public string Path;
            public OptimizationResult Result;

            public PackageResult(string Path, OptimizationResult Result)
            {
                this.Path = Path;
                this.Result = Result;
            }
        }

        public class PackageResolver
        {
```

```
52              string directory;
53              string name;
54              NecUtilities.In.Model model;
55              Problem problem;
56              ProblemWritter problemWritter = new ProblemWritter();
57              public PackageResolver(string directory, NecUtilities.In.Model
                     model, Problem problem)
58              {
59                  this.directory = directory;
60                  this.model = model;
61                  this.problem = problem;
62              }
63
64              public PackageResult Resolve(string package, string solver,
                    Dictionary<string, object> parameters)
65              {
66                  string solvedNec = null;
67                  var result = UsePackage(package, solver, parameters, ref
                        solvedNec);
68                  var writter = new OptimizationResultWritter();
69                  writter.Write(result, $"{directory}\\solution.txt");
70                  WriteSolution(result.GetComplexSolution(), solvedNec);
71                  var packageResult = new PackageResult(solvedNec, result);
72                  return packageResult;
73              }
74
75              private OptimizationResult UsePackage(string package, string
                    solver, Dictionary<string, object> parameters, ref string
                    solvedNec)
76              {
77                  switch (package.ToLower())
78                  {
79                      case "gams":
80                          return UseGamsPackage(solver, parameters, ref
                                solvedNec);
81                      default:
82                          return UseCustomPackage(package, solver,
                                parameters, ref solvedNec);
83                  }
84              }
85
86              public void WriteSolution(Complex[] solution, string fileName)
87              {
88                  if (solution == null)
89                  {
90                      return;
91                  }
92                  var clone = model.Copy();
93                  var sources = clone.Sources;
94                  for (var i = 0; i < sources.Length; i++)
95                  {
96                      sources[i].Value = solution[i];
97                  }
98                  File.WriteAllText(fileName, clone.ToString());
```

```
99              }
100
101             private OptimizationResult UseGamsPackage(string solver,
                    Dictionary<string, object> parameters, ref string
                    solvedNec)
102             {
103                 solvedNec = $"{directory}\\solved_by_gms.nec";
104                 var reslim = 1000;
105                 if (parameters.ContainsKey("reslim"))
106                 {
107                     reslim = (int)parameters["reslim"];
108                 }
109
110                 var input = $"{directory}\\{name}.gms";
111                 var gamsWritter = new GAMSWritter();
112                 gamsWritter.Write(problem, input, problem.analyticSolution
                        , reslim);
113                 var gams = new GAMSSolver(solver);
114                 var result = gams.Solve(input);
115                 return result;
116             }
117
118             private void WriteGams()
119             {
120                 var input = $"{directory}\\problem.gms";
121                 var gamsWritter = new GAMSWritter();
122                 var reslim = 1000;
123                 gamsWritter.Write(problem, input, problem.analyticSolution
                        , reslim);
124             }
125
126             private OptimizationResult UseCustomPackage(string packageName
                    , string solver, Dictionary<string, object> parameters,
                    ref string solvedNec)
127             {
128                 solvedNec = $"{directory}\\solved_by_{packageName}_{solver
                        }.nec";
129                 var package = new CustomSolver(packageName, solver,
                        parameters);
130                 var problemFile = $"{directory}\\problem.dat";
131                 problemWritter.WriteBin(problem, problemFile);
132                 WriteGams();
133                 var result = package.Solve(problemFile);
134                 return result;
135             }
136         }
137
138         public Experiment(string nec, string package, string solver,
                double theta, double phi, string compare, bool draw,
                Dictionary<string, object> parameters)
139         {
140             this.nec = nec;
141             this.package = package;
142             this.solver = solver;
```

```
143             this.theta = theta;
144             this.phi = phi;
145             this.name = nec.Replace(".nec", "");
146             this.compare = compare;
147             this.parameters = parameters;
148             this.draw = draw;
149         }
150 
151         public void Solve(bool rerunNec)
152         {
153             var paaFolder = $"{name}_results";
154             Console.WriteLine(paaFolder);
155             var model = parser.Parse(nec);
156             var frequency = model.Frequency.frequency;
157             var frequesncyFolder = $"{paaFolder}\\{frequency}MHz";
158             var directionFolder = $"{frequesncyFolder}\\{theta}-{phi}";
159             Directory.CreateDirectory(directionFolder);
160             Console.WriteLine(directionFolder);
161             var analyticNec = $"{directionFolder}\\analytic.nec";
162             Console.WriteLine("Run nec");
163             var problem = problemBuilder.Build(model, theta, phi,
                    frequesncyFolder, rerunNec);
164             Utils.Symmetrize(problem);
165             ExtendingAnalyze(model, problem, $"{frequesncyFolder}\\report.
                    txt");
166             Console.WriteLine("Problem files ready");
167             WriteMatrix(problem.A, $"{directionFolder}\\A.txt", 10);
168             for (var i = 0; i < problem.B.Length; i++)
169             {
170                 WriteMatrix(problem.B[i], $"{directionFolder}\\B{i + 1}.
                        txt", 15);
171             }
172             var resolver = new PackageResolver(directionFolder, model,
                    problem);
173             resolver.WriteSolution(problem.analyticSolution, analyticNec);
174             var result = resolver.Resolve(package, solver, parameters);
175             if (draw)
176             {
177                 MakeDiagrams(result.Path, analyticNec, compare,
                        directionFolder, rerunNec, frequency);
178             }
179             CollectResult(result.Result);
180         }
181 
182         private void CollectResult(OptimizationResult result)
183         {
184             var filename = "batch_results.txt";
185             var text = " ";
186             var pars = ComposeOutputLine(result);
187             if (File.Exists(filename) == false)
188             {
189                 foreach(var par in pars)
190                 {
191                     text += $"\t{par.Key}";
```

```
192                 }
193                 text += "\n ";
194             }
195             foreach (var par in pars)
196             {
197                 text += $"\t{HumanReadableValue(par)}";
198             }
199             text += "\n";
200             File.AppendAllText(filename, text);
201         }
202 
203         private string HumanReadableValue(KeyValuePair<string, object>
                pair)
204         {
205             if (pair.Value is string)
206             {
207                 return $"\"{pair.Value}\"";
208             }
209             if (pair.Value is double)
210             {
211                 return $"{pair.Value}".Replace(",", ".");
212             }
213             if (pair.Value is int)
214             {
215                 if (pair.Key == "de_status")
216                 {
217                     switch (pair.Value)
218                     {
219                         case 0:
220                             return "Normal Completion";
221                         case 1:
222                             return "Iterations Limit";
223                         case 2:
224                             return "Record estimation";
225                         default:
226                             return "Unknown";
227                     }
228                 }
229             }
230             if (pair.Value is double[])
231             {
232                 var array = (double[])pair.Value;
233                 var text = "";
234                 foreach (var value in array)
235                 {
236                     text += $"{value} ".Replace(",", ".");
237                 }
238                 return text;
239             }
240             return $"{pair.Value}";
241         }
242 
243         private KeyValuePair<string, object>[] ComposeOutputLine(
                OptimizationResult result)
```

```
244         {
245             var dictionary = new Dictionary<string, object>();
246             foreach (var pair in parameters)
247             {
248                 dictionary[pair.Key] = pair.Value;
249             }
250             foreach(var pair in result.output)
251             {
252                 dictionary[pair.Key] = pair.Value;
253             }
254             dictionary["z"] = -result.z;
255             dictionary["Nec"] = nec;
256             dictionary["Theta"] = theta;
257             dictionary["Phi"] = phi;
258             var pairs = dictionary.ToArray();
259             Array.Sort(pairs, (lhs, rhs) => {
260                 var res = GetPriority(lhs.Key) - GetPriority(rhs.Key);
261                 if (res == 0)
262                 {
263                     return lhs.Key.CompareTo(rhs.Key);
264                 }
265                 return res;
266             });
267             return pairs;
268         }
269 
270         private int GetPriority(string key)
271         {
272             var index = Array.IndexOf(PreferredOutputOrder, key);
273             if (index < 0)
274             {
275                 return PreferredOutputOrder.Length;
276             }
277             return index;
278         }
279 
280         private void ExtendingAnalyze(NecUtilities.In.Model model, Problem
                problem, string reportFileName)
281         {
282             var file = new StreamWriter(reportFileName);
283             var segs = 0;
284             foreach (var wire in model.Wires)
285             {
286                 segs += wire.seg;
287             }
288             if (segs > 11000)
289             {
290                 file.WriteLine($"                    : {segs}.");
291             }
292             else
293             {
294                 file.WriteLine($"                     : {segs}.");
295             }
296             Utils.Symmetrize(problem);
```

```
297             var Bsum = problem.Bsum;
298             double[][] grid = new double[Bsum.Height / 2][];
299             for (var i = 0; i < Bsum.Height / 2; i++)
300             {
301                 grid[i] = new double[Bsum.Width / 2];
302                 for (var j = 0; j < Bsum.Width / 2; j++)
303                 {
304                     grid[i][j] = Bsum[i, j];
305                 }
306             }
307             var eigenValues = Utils.EigenValues(Bsum, 50);
308             if (eigenValues.Has((value) => value < 0))
309             {
310                 eigenValues = Utils.EigenValues(Bsum, 100);
311             }
312             if (eigenValues.Has((value) => value < 0))
313             {
314                 eigenValues = Utils.EigenValues(Bsum, 1000);
315             }
316             if (eigenValues.Has((value) => value < 0))
317             {
318                 file.WriteLine("    $B_{\\Sigma}$                    .
                                                          .");
319                 file.Close();
320                 return;
321             }
322             var lamdaMin = eigenValues.First;
323             var lamdaMax = eigenValues.Last;
324 
325             file.WriteLine("    $B_{\\Sigma}$
                                                  :\\\\ " +
326                 "$\\lambda_{max} = " + $"{lamdaMax}".Replace(",", ".") + "
                        $, \\\\" +
327                 "$\\lambda_{min} = " + $"{lamdaMin}".Replace(",", ".") + "
                        $.\\\\");
328 
329             var cond = lamdaMax / lamdaMin;
330 
331             file.WriteLine("                $B_{\\Sigma}     " + $"{cond}".
                    Replace(",", ".") + "$.");
332             var radius = Math.Sqrt(problem.A.Width / 2 / lamdaMin);
333             file.WriteLine("                    $r = " + $"{radius}".Replace
                    (",", ".") + "$.");
334             file.Close();
335             problem.Radius = radius;
336         }
337 
338         private void WriteMatrix(Matrix M, string fileName, int itemWidth)
339         {
340             var file = new StreamWriter(fileName);
341             for (var i = 0; i < M.Height; i++)
342             {
343                 for (var j = 0; j < M.Width; j++)
344                 {
```

```
                    file.Write($"{MatrixFormat(M[i, j], itemWidth)} ");
                }
                file.WriteLine();
            }
            file.Close();
        }

        private string MatrixFormat(double value, int width)
        {
            var radix = Math.Pow(10, width);
            var rounded = ((long)(value * radix)) / (1.0 * radix);
            var str = $"{rounded}";
            if (str.Length > width)
            {
                return $"{str.Substring(0, width - 3)}...";
            }
            while (str.Length < width)
            {
                str += " ";
            }
            return str;
        }

        private void WriteSolution(NecUtilities.In.Model model, Complex[]
            solution, string fileName)
        {
            if (solution == null) {
                return;
            }
            var clone = model.Copy();
            var sources = clone.Sources;
            for (var i = 0; i < sources.Length; i++)
            {
                sources[i].Value = solution[i];
            }
            File.WriteAllText(fileName, clone.ToString());
        }

        private OptimizationResult SolveWithGams(Problem problem, string
            folder, int reslim = 1000)
        {
            var input = $"{folder}\\{name}.gms";
            var gamsWritter = new GAMSWritter();
            gamsWritter.Write(problem, input, problem.analyticSolution,
                reslim);
            var solver = new GAMSSolver(this.solver);
            var writter = new OptimizationResultWritter();
            var result = solver.Solve(input);
            writter.Write(result, $"{folder}\\solution.txt");
            return result;
        }

        private void MakeDiagrams(string solvedNec, string analyticNec,
            string compareNec, string folder, bool rerunNec, double
```

```
            frequency)
395         {
396             var inputsList = new List<PatternInput>();
397             if (compareNec != null && compareNec.Length > 0)
398             {
399                 inputsList.Add(PatternInput("Single", "#909090", "2px",
                        compareNec, frequency));
400             }
401             var verticalPlaneTool = new VerticalPlanSVGTool();
402             var horisontalPlaneTool = new HorisontalPlanSVGTool();
403             inputsList.Add(PatternInput("Analytic", "#606060", "3px",
                    analyticNec, frequency));
404             inputsList.Add(PatternInput("PAA", "#000000", "3px", solvedNec
                    , frequency));
405             var inputs = inputsList.ToArray();
406             var verticalPatternsInput = new VerticalPlanPatternsInput(phi,
                     -90, 90, 1, "", inputs);
407             var horisontalPatternsInput = new HorisontalPlanPatternsInput(
                    theta, 0, 359, 1, "", inputs);
408             verticalPlaneTool.VerticalPlane(
409                 $"vertical_plane.svg",
410                 verticalPatternsInput,
411                 $"{folder}\\vertical_plane.svg"
412             );
413             horisontalPlaneTool.HorisontalPlane(
414                 $"horisontal_plane.svg",
415                 horisontalPatternsInput,
416                 $"{folder}\\horisontal_plane.svg"
417             );
418         }
419
420         private PatternInput PatternInput(string caption, string color,
                string width, string nec, double frequency)
421         {
422             var input = nec;
423             PrepareForDiagram(input, input, frequency);
424             var output = nec.Replace(".nec", ".out");
425             var pattern = Utils.BeamPattern(input, output, true);
426             return new PatternInput(pattern, caption, color, width);
427         }
428
429         private void PrepareForDiagram(string input, string output, double
                 frequency)
430         {
431             var model = new NecUtilities.In.Parser().Parse(input);
432             var rp = model.RadiationPattern;
433             rp.Phi0 = 0;
434             rp.PhiNumber = 360;
435             rp.PhiInc = 1;
436             rp.ThetaNumber = 91;
437             rp.Theta0 = 0;
438             rp.ThetaInc = 1;
439             model.Frequency.frequency = frequency;
440             var file = new StreamWriter(output);
```

```
441             file.Write(model.ToString());
442             file.Close();
443         }
444     }
445 }
```

Listing 2: IDE

```
1
2 //
3 // ExpiIDE
4 // FileInspectorPresenter.cs
5 //
6 // Copyright © 2022 Nikolai Tiunin. All rights reserved.
7 //
8
9 using ExpiIDE.Core;
10 using Presentation.Modules.EXP;
11 using Presentation.Modules.IDE;
12 using Presentation.Modules.NEC;
13 using Presentation.Modules.SVG;
14 using Presentation.Modules.TXT;
15 using System;
16 using System.Collections.Generic;
17 using System.Drawing;
18 using System.IO;
19 using System.Linq;
20 using System.Windows.Forms;
21
22 namespace Presentation.Modules.FileInspector
23 {
24
25     public class FileHierarchyItem
26     {
27         public string name;
28         public string path;
29
30         public bool IsExists
31         {
32             get
33             {
34                 return File.Exists(path);
35             }
36         }
37
38         public FileHierarchyItem(string name, string path)
39         {
40             this.path = path;
41             this.name = name;
42         }
43
44         public static FileHierarchyItem Item(string path)
45         {
```

```
46              if (Directory.Exists(path))
47              {
48                  return FolderItem(path);
49              }
50              return ContentItem(path);
51          }
52
53          public static FolderItem FolderItem(string path)
54          {
55              var name = new FileInfo(path).Name;
56              return new FolderItem(name, path);
57          }
58
59          public static ContentItem ContentItem(string path)
60          {
61              var name = new FileInfo(path).Name;
62              var ext = name.Split('.').Last();
63              var type = FileItemType.unknown;
64              switch (ext)
65              {
66                  case "nec":
67                      type = FileItemType.nec;
68                      break;
69                  case "svg":
70                      type = FileItemType.svg;
71                      break;
72                  case "exp":
73                      type = FileItemType.exp;
74                      break;
75                  case "txt":
76                      type = FileItemType.txt;
77                      break;
78              }
79              return new ContentItem(name, path, type);
80          }
81
82          public FileHierarchyItem Find(string keyPath)
83          {
84              if (name == keyPath)
85              {
86                  return this;
87              }
88              if (keyPath.Length == 0)
89              {
90                  return null;
91              }
92              var slashIndex = keyPath.IndexOf('\\');
93              var title = keyPath;
94              var remaining = "";
95              if (slashIndex > 0)
96              {
97                  title = keyPath.Substring(0, slashIndex);
98                  remaining = keyPath.Remove(0, slashIndex + 1);
99              }
```

```
100             if (title != name)
101             {
102                 return null;
103             }
104             if (title == name && remaining.Length == 0)
105             {
106                 return this;
107             }
108             if (this is FolderItem)
109             {
110                 var folder = (FolderItem)this;
111                 foreach (var item in folder.items)
112                 {
113                     var subItem = item.Value.Find(remaining);
114                     if (subItem != null)
115                     {
116                         return subItem;
117                     }
118                 }
119             }
120             return null;
121         }
122     }
123     public class FolderItem: FileHierarchyItem
124     {
125         public Dictionary<string, FileHierarchyItem> items;
126         public bool isLoaded = false;
127         public bool isExpanded = false;
128 
129         public bool IsExists
130         {
131             get
132             {
133                 return Directory.Exists(path);
134             }
135         }
136 
137 
138         public FolderItem(string name, string path): base(name, path)
139         {
140             items = new Dictionary<string, FileHierarchyItem>();
141         }
142 
143         public bool Load()
144         {
145             var isChanged = false;
146             var checkList = new Dictionary<string, FileHierarchyItem>();
147             foreach (var item in items)
148             {
149                 checkList[item.Key] = item.Value;
150             }
151             var files = Directory.GetFiles(path);
152             var directories = Directory.GetDirectories(path);
153             var content = new List<string>();
```

```
154             content.AddRange(files);
155             content.AddRange(directories);
156             foreach(var path in content)
157             {
158                 FileHierarchyItem item;
159                 if (checkList.ContainsKey(path) == false)
160                 {
161                     item = Item(path);
162                     items[path] = item;
163                     isChanged = true;
164                 }
165                 else {
166                     item = checkList[path];
167                     checkList.Remove(item.path);
168                 }
169 
170                 if (item is FolderItem)
171                 {
172                     isChanged |= ((FolderItem)item).Load();
173                 }
174             }
175             if (checkList.Count > 0)
176             {
177                 foreach (var item in checkList)
178                 {
179                     items.Remove(item.Key);
180                 }
181                 isChanged = true;
182             }
183             return isChanged;
184         }
185 
186         public void Toggle()
187         {
188             isExpanded = !isExpanded;
189 
190         }
191     }
192 
193     public class ContentItem : FileHierarchyItem
194     {
195         public FileItemType type;
196 
197         public ContentItem(string name, string path, FileItemType type) :
                base(name, path)
198         {
199             this.type = type;
200         }
201     }
202 
203 
204     public partial class FileInspectorPresenter
205     {
206         IDEModuleOutput output;
```

```
207         public FileInspectorView view;
208         EXPModule expModule;
209         SVGModule svgModule;
210         NECModule necModule;
211         TXTModule txtModule;
212         FolderItem root;
213         System.Timers.Timer timer;
214         Dictionary<string, FileItem> fileItems = new Dictionary<string,
                FileItem>();
215         FileItem currentFileItem = null;
216         private int timerTicks = 0;
217 
218         public FileInspectorPresenter(
219             string path,
220             EXPModule expModule,
221             SVGModule svgModule,
222             NECModule necModule,
223             TXTModule txtModule,
224             IDEModuleOutput output)
225         {
226             this.expModule = expModule;
227             this.svgModule = svgModule;
228             this.necModule = necModule;
229             this.txtModule = txtModule;
230             var directory = path;
231             if (Directory.Exists(path) == false)
232             {
233                 var info = new FileInfo(path);
234                 directory = info.Directory.FullName;
235             }
236             root = FileHierarchyItem.FolderItem(directory);
237             root.isExpanded = true;
238             this.output = output;
239             StartTimer();
240         }
241 
242         ~FileInspectorPresenter()
243         {
244             StopTimer();
245         }
246 
247         public void DidSelect(string path)
248         {
249             var item = root.Find(path);
250             if (item == null || item is ContentItem == false)
251             {
252                 return;
253             }
254             var contentItem = (ContentItem)item;
255             var fileItem = FindOrCreateFileItem(contentItem);
256 
257             switch (contentItem.type)
258             {
259                 case FileItemType.exp:
```

```
                    currentFileItem = fileItem;
                    expModule.Input.Open(fileItem);
                    view.previewView.Show(expModule.View);
                    break;
                case FileItemType.nec:
                    currentFileItem = fileItem;
                    necModule.Input.Open(item.path);
                    view.previewView.Show(necModule.View);
                    break;
                case FileItemType.svg:
                    currentFileItem = null;
                    svgModule.Input.Open(item.path);
                    view.previewView.Show(svgModule.View);
                    break;
                case FileItemType.txt:
                    currentFileItem = fileItem;
                    txtModule.Input.Open(fileItem);
                    view.previewView.Show(txtModule.View);
                    break;
                default:
                    currentFileItem = null;
                    view.previewView.ShowPlaceholder();
                    break;
            }
            output.DidUpdate(fileItem);
        }

        public void UpdateContent()
        {
            if (currentFileItem == null)
            {
                view.previewView.ShowPlaceholder();
                return;
            }
            switch (currentFileItem.type)
            {
                case FileItemType.exp:
                    expModule.Input.Update();
                    break;
                case FileItemType.txt:
                    txtModule.Input.Update();
                    break;
                case FileItemType.nec:
                    break;
                case FileItemType.svg:
                    break;
                default:
                    view.previewView.ShowPlaceholder();
                    break;
            }
        }

        public void DidRequestOptions(string path, Point location)
        {
```

```
314             var item = root.Find(path);
315 
316             if (item == null)
317             {
318                 return;
319             }
320 
321             var del = new ToolStripMenuItem("    ", null, (o, e) => {
322                 Delete(item.path);
323             });
324             if (item is FileItem)
325             {
326                 Show(new ToolStripItem[] { del }, location);
327             } else
328             {
329                 var folder = item.path;
330                 var createFolder = new ToolStripMenuItem("   ", null, (o,
                        e) => {
331                     view.ShowCreateFileView("folder", (name) => {
332                         CreateFolder($"{folder}\\{name}");
333                     });
334                 });
335                 var createFile = new ToolStripMenuItem("  ", null, (o, e)
                        => {
336                     view.ShowCreateFileView("file", (name) => {
337                         CreateFile($"{folder}\\{name}");
338                     });
339                 });
340                 var create = new ToolStripMenuItem("    ", null, new
                        ToolStripItem[] {
341                    createFolder, createFile
342                 });
343                 Show(new ToolStripItem[] {
344                     create, del
345                 }, location);
346             }
347         }
348 
349         public void Expand(string path)
350         {
351             var item = root.Find(path);
352             if (item is FolderItem == false)
353             {
354                 return;
355             }
356             var folder = (FolderItem)item;
357             folder.isExpanded = true;
358         }
359 
360         public void Collapse(string path)
361         {
362             var item = root.Find(path);
363             if (item is FolderItem == false)
364             {
```

```
365                 return;
366             }
367             var folder = (FolderItem)item;
368             folder.isExpanded = false;
369         }
370 
371         private void Show(ToolStripItem[] items, Point location)
372         {
373             view.hierarchyView.ShowContextMenu(items, location);
374         }
375 
376         private void CreateFile(string path)
377         {
378             File.WriteAllText(path, "");
379         }
380 
381         private void CreateFolder(string path)
382         {
383             Directory.CreateDirectory(path);
384         }
385 
386         private void Delete(string path)
387         {
388             if (Directory.Exists(path))
389             {
390                 Directory.Delete(path, true);
391             }
392             else if (File.Exists(path))
393             {
394                 File.Delete(path);
395             }
396         }
397 
398         private void StartTimer()
399         {
400             timerTicks = 0;
401             timer = new System.Timers.Timer(300);
402             timer.Elapsed += new System.Timers.ElapsedEventHandler(
                    TimerFired);
403             timer.AutoReset = true;
404             timer.Start();
405         }
406 
407         private void StopTimer()
408         {
409             if (timer != null)
410             {
411                 timer.Stop();
412             }
413         }
414 
415         private void TimerFired(object sender, EventArgs e)
416         {
417             timerTicks++;
```

```
418             if (timerTicks % 10 == 0)
419             {
420                 timerTicks = 0;
421                 UpdateUndoIfNeeded();
422             }
423             view.Invoke(new Action(() => {
424                 UpdateToolsIfNeeded();
425             }));
426             if (view == null)
427             {
428                 return;
429             }
430             if (root.Load() == false)
431             {
432                 return;
433             }
434             view.Invoke(new Action(() => {
435                 UpdateView();
436             }));
437         }
438 
439         private FileItem FindOrCreateFileItem(ContentItem contentItem)
440         {
441             if (fileItems.ContainsKey(contentItem.path))
442             {
443                 return fileItems[contentItem.path];
444             }
445             var item = new FileItem(contentItem.path, contentItem.type);
446             fileItems[contentItem.path] = item;
447             return item;
448         }
449     }
450 
451     partial class FileInspectorPresenter : FileInspectorModuleInput
452     {
453         public void Redo()
454         {
455             if (currentFileItem == null)
456             {
457                 return;
458             }
459             currentFileItem.Redo();
460             UpdateContent();
461         }
462 
463         public void Undo()
464         {
465             if (currentFileItem == null)
466             {
467                 return;
468             }
469             currentFileItem.Undo();
470             UpdateContent();
471         }
```

```
472         public void Save()
473         {
474             if (currentFileItem == null)
475             {
476                 return;
477             }
478             currentFileItem.Save();
479         }
480 
481         public void ToggleRun()
482         {
483             if (currentFileItem == null)
484             {
485                 return;
486             }
487             switch (currentFileItem.type)
488             {
489                 case FileItemType.exp:
490                     expModule.Input.ToggleRunning();
491                     break;
492                 default:
493                     break;
494             }
495         }
496 
497 
498         public void UpdateView()
499         {
500             var items = (root.IsExists) ?
501                 new FileHierarchyItem[] { root } :
502                 new FileHierarchyItem[] { };
503             view.hierarchyView.UpdateHierarchyItems(items);
504         }
505 
506         private void UpdateToolsIfNeeded()
507         {
508             output.DidUpdate(currentFileItem);
509         }
510 
511         private void UpdateUndoIfNeeded()
512         {
513             if (currentFileItem == null)
514             {
515                 return;
516             }
517             currentFileItem.UpdateHistory();
518         }
519 
520         public bool IsRunning(FileItem fileItem)
521         {
522             return expModule.Input.IsRunning(fileItem);
523         }
524     }
525 }
```

Listing 3: Solve

```
//
// ExpiFramework
// SolveParser.cs
//
// Copyright © 2022 Nikolai Tiunin. All rights reserved.
//

using Expi;
using Parsec;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Constro.Parser
{
    public class SolvePAA: ContextStatement
    {
        ContextArguments arguments;

        public SolvePAA(ContextArguments arguments)
        {
            this.arguments = arguments;
        }

        public void Execute(Context context)
        {
            var args = arguments.Value(context);
            var nec = (string)Arg("nec", "n", args);
            var theta = (double)Arg("theta", null, args);
            var phi = (double)Arg("phi", null, args);
            var package = (string)Arg("package", "p", args);
            var solver = (string)Arg("solver", "s", args);
            var compare = (string)Arg("compare", "c", args);
            DateTime start = DateTime.Now;
            Console.WriteLine($"Started at {start}");
            bool draw = true;
            if (args.ContainsKey("draw"))
            {
                draw = bool.Parse((string)args["draw"]);
                args.Remove("draw");
            }
            var exp = new Experiment(nec, package, solver, theta, phi,
                compare, draw, args);
            exp.Solve(false);
            DateTime end = DateTime.Now;
            Console.WriteLine($"Finished at {end}");
            double mils = (end - start).TotalMilliseconds;
```

```
50              Console.WriteLine($"Duration: {mils / 1000} s.");
51          }
52
53          object Arg(string l, string s, Dictionary<string, object> args)
54          {
55              if (args.ContainsKey(l))
56              {
57                  var a = args[l];
58                  args.Remove(l);
59                  return a;
60              }
61              if (s != null && args.ContainsKey(s))
62              {
63                  var a = args[s];
64                  args.Remove(s);
65                  return a;
66              }
67              throw new NoParameterException();
68          }
69      }
70
71      public class SolveParser : ConstroWordsParser
72      {
73          public SolveParser() : base(new Map<ContextArguments,
                ContextStatement, StringBuilder>(
74              ArgsAfter(Words("solve", "paa")),
75              a => new SolvePAA(a)
76          )) {}
77      }
78  }
```

Listing 4: Wire

```
1  //
2  // ExpiFramework
3  // WireParser.cs
4  //
5  // Copyright © 2022 Nikolai Tiunin. All rights reserved.
6  //
7
8  using Parsec;
9  using System.Collections.Generic;
10 using System.Numerics;
11 using System.Text;
12
13 namespace Constro.Parser
14 {
15
16     public class WireStatement : ContextStatement
17     {
18         public static string WiresKey = "${wires}";
19         ContextWire Wire;
20
```

```
21          public WireStatement(ContextWire Wire)
22          {
23              this.Wire = Wire;
24          }
25
26          public void Execute(Context context)
27          {
28              List<SegmentedWire> segmentedWires = (List<SegmentedWire>)
                    context.LocalValue(WiresKey);
29              if (segmentedWires == null)
30              {
31                  segmentedWires = new List<SegmentedWire>();
32              }
33              var wire = Wire.Value(context);
34              segmentedWires.Add(wire);
35              context.Set(segmentedWires, WiresKey);
36          }
37      }
38
39      public class WireStatementParser: CustomParser<ContextStatement,
            StringBuilder>
40      {
41          public WireStatementParser() : base(
42              new Map<ContextWire, ContextStatement, StringBuilder>(
43                  new WireParser(),
44                  wire => new WireStatement(wire)
45              )
46          ) { }
47      }
48      public class ContextJunction: ContextExpression<Junction>
49      {
50          public ContextJunction(string str = "->") : base(str) { }
51
52          public override Junction Value(Context context)
53          {
54              return new SimpleJunction();
55          }
56      }
57
58      public class ContextFedJunction : ContextJunction
59      {
60          ContextExpression<Complex> value;
61          FedJunction.Mesure mesure;
62          public ContextFedJunction(ContextExpression<Complex> value,
                FedJunction.Mesure mesure) : base($"~{value} {mesure}~")
63          {
64              this.value = value;
65              this.mesure = mesure;
66          }
67
68          public override Junction Value(Context context)
69          {
70              var complex = this.value.Value(context);
71
```

```
72              return new FedJunction(complex, mesure);
73          }
74      }
75
76      public class JunctionParser: CustomParser<ContextJunction,
            StringBuilder>
77      {
78          public JunctionParser() : base(
79              new Map<StringBuilder, ContextJunction, StringBuilder>(
80                  new StringParser("->").SkipLeadingWhitespaces,
81                  (str) => {
82                      return new ContextJunction();
83                  }
84              )
85          ) { }
86      }
87
88      public class FedJunctionParser : CustomParser<ContextFedJunction,
            StringBuilder>
89      {
90          static GenericParser<StringBuilder, StringBuilder> TildaParser =
                new StringParser("~").SkipLeadingWhitespaces;
91          static GenericParser<FedJunction.Mesure, StringBuilder>
                MesureParser = new Or<FedJunction.Mesure, StringBuilder>(
92              new Map<StringBuilder, FedJunction.Mesure, StringBuilder>(
93                  new SkipLeadingWhitespaces(new CharSetParser("vV").
                        StringParser),
94                  (str) => { return FedJunction.Mesure.Voltage; }
95              ),
96              new Map<StringBuilder, FedJunction.Mesure, StringBuilder>(
97                  new SkipLeadingWhitespaces(new CharSetParser("aA").
                        StringParser),
98                  (str) => { return FedJunction.Mesure.Current; }
99              )
100         );
101
102         static GenericParser<Pair<ContextComplex, FedJunction.Mesure>,
                StringBuilder> ValueParser = new Both<ContextComplex,
                FedJunction.Mesure, StringBuilder>(
103             new ContextComplexParser(),
104             MesureParser
105         );
106
107         static GenericParser<ContextFedJunction, StringBuilder>
                JunctionParser =
108             new Map<Pair<ContextComplex, FedJunction.Mesure>,
                    ContextFedJunction, StringBuilder>(
109                 ValueParser,
110                 (value) =>
111                 {
112                     return new ContextFedJunction(value.First, value.
                            Second);
113                 }
114             );
```

```
115
116         public FedJunctionParser() : base(
117             new Right<StringBuilder, ContextFedJunction, StringBuilder>(
118                 TildaParser,
119                 new Left<ContextFedJunction, StringBuilder, StringBuilder
                        >(
120                     JunctionParser,
121                     TildaParser
122                 )
123             )
124         )
125         { }
126     }
127
128     public class ContextWire: ContextExpression<SegmentedWire>
129     {
130         ContextPoint anchor;
131         Pair<ContextJunction, ContextPoint>[] segments;
132
133         public ContextWire(ContextPoint anchor, Pair<ContextJunction,
                ContextPoint>[] segments) : base($"{anchor} -> ...")
134         {
135             this.anchor = anchor;
136             this.segments = segments;
137         }
138
139         public override SegmentedWire Value(Context context)
140         {
141             var segments = new List<Segment>();
142             var segs = this.segments;
143             if (segs.Length == 0)
144             {
145                 return new SegmentedWire(segments.ToArray());
146             }
147             var t = MakeTransform(context);
148             var segment = MakeSegment(anchor, segs[0].First, segs[0].
                    Second, context, t);
149             segments.Add(segment);
150             for(var i = 0; i < segs.Length - 1; i++)
151             {
152                 var lhs = segs[i];
153                 var rhs = segs[i + 1];
154                 segment = MakeSegment(lhs.Second, rhs.First, rhs.Second,
                        context, t);
155                 segments.Add(segment);
156             }
157             return new SegmentedWire(segments.ToArray());
158         }
159
160         private Transform MakeTransform(Context context)
161         {
162             var initial = Transform.Identity();
163             if (context.Parent != null)
164             {
```

```
165                 initial = MakeTransform(context.Parent);
166             }
167             var localObj = context.LocalValue(TransformStatement.
                    TransformKey);
168             if (localObj == null)
169             {
170                 return initial;
171             }
172             var local = (Transform)localObj;
173             var t = initial * local;
174             return t;
175         }
176 
177         private Segment MakeSegment(ContextPoint left, ContextJunction
                junction, ContextPoint right, Context context, Transform t)
178         {
179             var lhs = t * left.Value(context);
180             var rhs = t * right.Value(context);
181             var junc = junction.Value(context);
182             return new Segment(lhs, junc, rhs);
183         }
184     }
185 
186     public class WireParser: CustomParser<ContextWire, StringBuilder>
187     {
188 
189         static GenericParser<ContextJunction, StringBuilder> JuctParser =
                new Or<ContextJunction, StringBuilder>(
190             new JunctionParser(),
191             new Map<ContextFedJunction, ContextJunction, StringBuilder>(
192                 new FedJunctionParser(), j => j
193             )
194         );
195         static GenericParser<Pair<ContextJunction, ContextPoint>,
                StringBuilder> SegmentParser = new Both<ContextJunction,
                ContextPoint, StringBuilder>(
196             JuctParser, new PointParser()
197         );
198 
199         static GenericParser<Pair<ContextJunction, ContextPoint>[],
                StringBuilder> SegmentsParser = new ManyOne<Pair<
                ContextJunction, ContextPoint>, StringBuilder>(
200             SegmentParser
201         );
202 
203         public WireParser() : base(
204             new Map<Pair<ContextPoint, Pair<ContextJunction, ContextPoint
                    >[]>, ContextWire, StringBuilder>(
205                 new Both<ContextPoint, Pair<ContextJunction, ContextPoint
                        >[], StringBuilder>(
206                     new PointParser(),
207                     SegmentsParser
208                 ), pair => new ContextWire(pair.First, pair.Second)
209             )
```

```
210         ) { }
211     }
212 }
```

Listing 5: Def

```
1  //
2  // ExpiFramework
3  // DefParser.cs
4  //
5  // Copyright © 2022 Nikolai Tiunin. All rights reserved.
6  //
7
8  using Parsec;
9  using System.Collections.Generic;
10 using System.Text;
11
12 namespace Constro.Parser
13 {
14     public class Def<A> : ContextStatement
15     {
16         protected string name;
17         ContextExpression<A> value;
18
19         public Def(string name, ContextExpression<A> value)
20         {
21             this.name = name;
22             this.value = value;
23         }
24
25         public virtual void Execute(Context context)
26         {
27             context.Set(this, name);
28         }
29
30         public A Unwrap(Context context)
31         {
32             return value.Value(context);
33         }
34     }
35
36     public class UnwrappingDef<A> : Def<A>
37     {
38
39         public UnwrappingDef(string name, ContextExpression<A> value):
               base(name, value)
40         {
41         }
42
43         public override void Execute(Context context)
44         {
45             context.Set(Unwrap(context), name);
46         }
```

```
47      }
48
49      public class ContextGroup: ContextExpression<Group>
50      {
51          public string Name;
52          public ContextStatement[] Content;
53
54          public ContextGroup(string name, ContextStatement[] content) :
                base(name)
55          {
56              this.Name = name;
57              this.Content = content;
58          }
59
60          public override Group Value(Context context)
61          {
62              foreach (var statement in Content)
63              {
64                  statement.Execute(context);
65              }
66              var list = (List<SegmentedWire>)context.Value(WireStatement.
                    WiresKey);
67              if (list == null)
68              {
69                  return new Group(Name, new SegmentedWire[] { });
70              }
71              return new Group(Name, list.ToArray());
72          }
73      }
74
75      public class DefParser<A>: ConstroWordsParser
76      {
77          protected static GenericParser<string, StringBuilder>
                IdentifierParser =
78              new Map<StringBuilder, string, StringBuilder>(
79                  IdAfter("def"),
80                  str =>
81                  {
82                      return str.ToString();
83                  }
84              );
85
86          protected static GenericParser<string, StringBuilder> AssignParser
                 = new Left<string, StringBuilder, StringBuilder>(
87              IdentifierParser,
88              Word("=")
89          );
90
91          public DefParser(GenericParser<ContextStatement, StringBuilder>
                parser): base(parser)
92          {
93
94          }
95      }
```

```
96
97      public class DefDoubleParser: DefParser<double>
98      {
99          static GenericParser<ContextStatement, StringBuilder> DoubleParser
                =
100             new Map<Pair<string, ContextExpression<double>>,
                    ContextStatement, StringBuilder>(
101                 new Both<string, ContextExpression<double>, StringBuilder
                        >(
102                     AssignParser,
103                     new ContextDoubleExpressionParser()
104                 ),
105                 pair => new UnwrappingDef<double>(pair.First, pair.Second)
106             );
107         public DefDoubleParser() : base(DoubleParser) {
108         }
109     }
110
111     public class DefPointParser: DefParser<Position>
112     {
113         static GenericParser<ContextStatement, StringBuilder> PointParser
                =
114             new Map<Pair<string, ContextPoint>, ContextStatement,
                    StringBuilder>(
115                 new Both<string, ContextPoint, StringBuilder>(
116                     AssignParser,
117                     new PointParser()
118                 ),
119                 pair => new UnwrappingDef<Position>(pair.First, pair.
                        Second)
120             );
121
122         public DefPointParser() : base(PointParser)
123         {
124         }
125     }
126
127     public class DefGroupParser: DefParser<Group>
128     {
129         public DefGroupParser(IdentifiersParser identifiersParser) : base(
130             Statements(identifiersParser, IdentifierParser, (i, s) =>
131             {
132                 identifiersParser.Register(i);
133                 return new Def<Group>(i, new ContextGroup(i, s));
134             })
135         )
136         { }
137
138         public override ResultOrError<ContextStatement, StringBuilder>
                Parse(StringBuilder input)
139         {
140             return base.Parse(input);
141         }
142     }
```

```
143 }
```