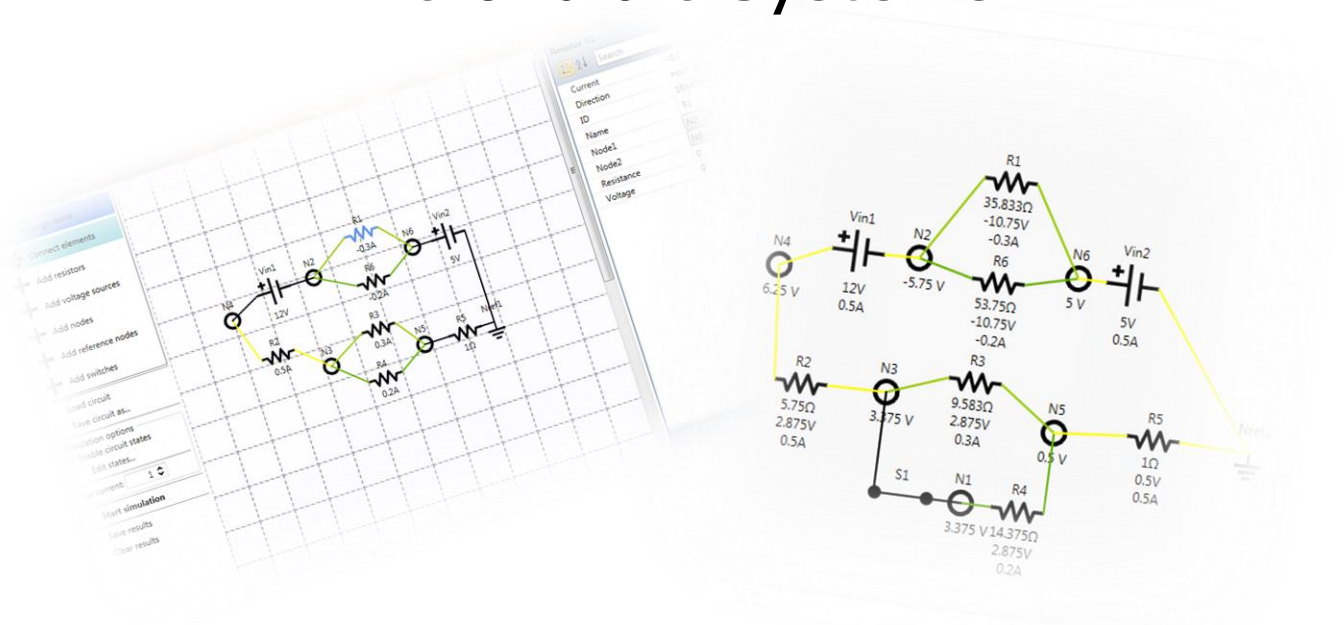
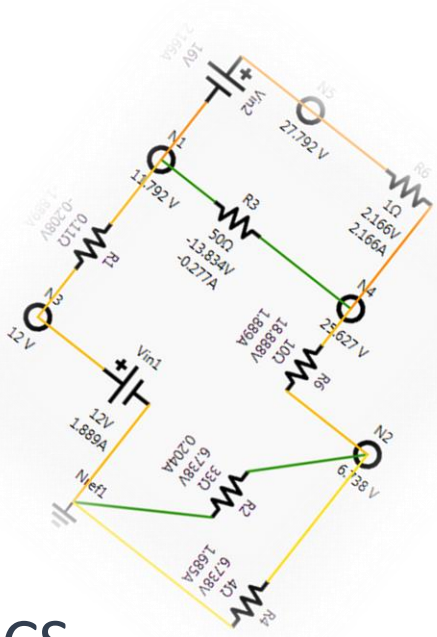
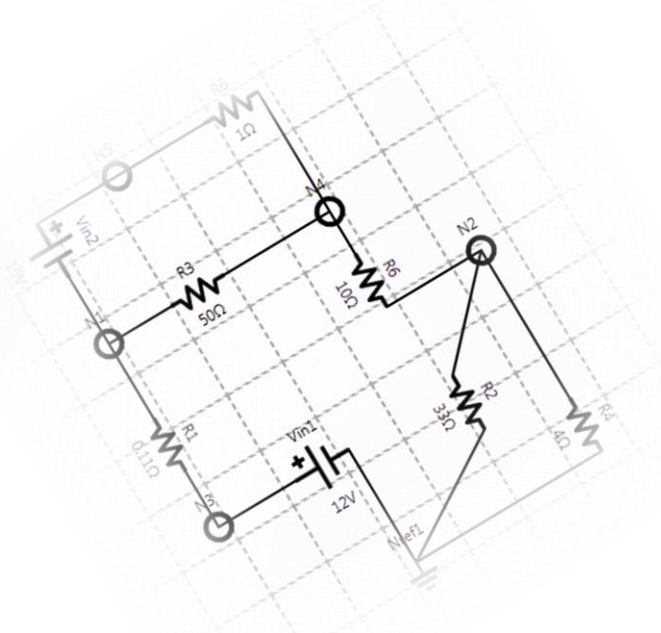
Electronic Circuit Simulation for Microfluidic Systems







ECS

By Mordechai Mor Sofer, Oron Feinerman

Advisor: Dr. Elishai Ezra-Tsur

Table of Contents

Acknowledgements

This project was a great and rare opportunity for us to learn about a subject previously unknown to us. We enjoyed exploring the field of microfluidics, and applying the knowledge we gained to a practical use. We feel that we have achieved something great with this project, but only thanks to everyone who has supported us along the way.

This project marks the end of our studies at the Jerusalem College of Technology (JCT) and our completion of B.Sc. in Software Engineering. We began studying for this degree in high school through the new academic studies program from Kiryat Herzog yeshiva high school, for Computer Science and Software Engineering, in cooperation with JCT, at the Machon Lev campus. We are members of the first class to take part in this program.

We give thanks to all of the people who made this long journey possible, and to God for enabling us to meet all the challenges and keep on living each day.

We would also like to give special thanks to:

- **Dr. Elishai Ezra-Tzur**, our project advisor, for the patience he had for us, the professional assistance he gave us, all the necessary information he gave us to read, and more than anything the opportunity he gave us to learn about the field of microfluidics.
- **Dr. Aryeh Teitelbaum**, Faculty Director of the Computer Science Department at JCT (Machon Lev campus), for recommending this project to us and for all the patience and help he has given to us and all of the students in the Computer Science Department at JCT.
- **Dr. Dan Bouhnik**, for his hard work in the mediation of the new academic studies program between JCT and Kiryat Herzog yeshiva high school, and for patience and help he has given to us and the rest of our class, as high school students and as college students.
- **Rabbi Moshe Zvi Vexler**, head of Kiryat Herzog yeshiva high school, who created the new program with JCT, for all the patience he had for us, and for all his help and support for 8 long years, to this very day.
- **Kiryat Herzog teachers and management**, for all the support in the hard times, and for the moments we almost gave up (as individuals and as a group) they encouraged us to continue doing what other people thought to be impossible, which we have achieved today.
- And finally, to **our families**, for supporting us at all times until now and beyond, and for all the help and emotional support they gave us.

Introduction to the Project

Our project is about microfluidic systems. Microfluidics deals with the behavior, precise control and manipulation of fluids. Usually we are talking about small fluids, typically sub millimeter scale. Microfluidics is a multidisciplinary field at the intersection of engineering, physics, chemistry, biochemistry, nanotechnology and biotechnology. In other words microfluidics is a field that combines a lot of different aspects and subjects. The field emerged in the early 1980s and is used in inkjet printers, DNA chips, lab-on-a-chip, micro-propulsion and micro-thermal technologies.

This field has great importance in the context of technological systems that work with fluids, particularly in the Biology field. For example, BioMEMS (**Bio**medical or **bio**logical **m**icro**e**lectro**m**echanical **s**ystems) - the science and technology of operating at the microscale for biological and biomedical applications - has biological applications which include genomics, proteomics, molecular diagnostics, point-of-care diagnostics, tissue engineering, and implantable microdevices.

As we will describe below our projects purpose is to describe microfluidic systems using electronic circuits. There is an analogy between microfluidic systems and electronic circuits, every channel is like a resistor, every pressure source is like a voltage source and the fluids current is like the electric current which flows in the circuit.

The circuit is an interconnection of electronic components (such as resistors, batteries, capacitors, switches) which are connected with conductors, in a way that allows an electrical current to flow through them.

A schematic illustration of circuit.

In order for the current to flow in the circuit, two conditions must be met:

1. The circuit needs to be connected to an electrical energy source. The electrical energy source will supply voltage or current, which can be constant or dependent on another factor.
2. The circuit must have at least one closed path which passes through an electrical energy source.

The problem description

The design and development of microfluidics system is very expensive, because engineers and scientists need to engineer, produce, examine and test the components, then rinse and repeat. Therefore simulation of microfluidics systems is necessary in order to design a better system initially, so it will have better chances of working in practice, instead of spending most of the time and budget on production and testing again and again until the desired results are reached through trial and error.

The problem with the simulation of microfluidic systems is that it is not a trivial task at all – an analytical or numeric solution to this kind of system is very complicated, so a simulation of that sort will require a significant amount of computational resources, thus increasing required time for design, equipment costs and energy costs.

The popular solutions of today

There are few popular solutions to this problem today:

The first solution is to perform a simulation on smaller and simpler microfluidic systems, thus reducing the amount of computational resources required to run the simulation routines. The second solution is simple – just build supercomputers that can perform the simulation very quickly.

The third solution is use of distributed computing; however this solution is used almost exclusively for microfluidics research projects that are for the greater good. Volunteers from the public can contribute computational resources towards the simulation, by use of a program that runs part of the calculations that are needed for the simulation (for example µFluids@Home which runs on BOINC, a popular distributed computing platform). As such, the simulation will be a lot quicker.

The problem with the solutions of today

All the common solutions have problems, and do not completely solve the problem of microfluidic simulation. The first solution, of course, is insufficient, since there is a need for simulation of bigger systems. The second solution does not fit most budgets, due to the high building costs and the high energy costs of a supercomputer. The third solution is not suitable for commercial researches, still requires a lot of time in order to perform the simulation (each volunteer is solving a calculation for 20 hours), and also depends on the public interest (but at least it's free).

Our solution description

We can solve the problem by solving another analogous problem which is electronic circuit simulation. There are well-known and well-optimized simulation algorithms for electronic circuits, such as Nodal analysis, its improved version: Modified nodal analysis, and Mesh current analysis. These algorithms require far less computational resources than any known dedicated microfluidic simulation algorithm.

We have built an application for the design and simulation of electronic circuits that represent microfluidic systems, with additional features specifically targeted at this purpose. The program allows building equivalent electronic circuits, and performing simulation of the circuit. It also allows defining different states of the switches in the circuit and simulating each of these states, a feature that is particularly useful for microfluidic applications. The circuit can be built by selecting components from the tool box and placing them in the design area, and connecting them with a couple of clicks. The properties of the components can be edited, and in the simulation window the details of each one of the components in the circuit will be displayed. The connections in the circuit are displayed in different colors which indicate the level of the current flow in that conductor. The states of the switches can be defined and saved to a file, as well as the circuit itself.

Our application was built in four stages:

1) We created a data model that represents a circuit which allows tracking the various values in the circuit, and implemented the basic version of the simulation algorithm and any additional required data structures.
2) We built a graphical user interface (GUI) to visualize the simulation of the circuit and provide a comfortable way to build the circuit. Our goal was to make the interface user friendly, as is the goal of every good user interface designer, and we have worked hard to achieve that goal.
3) We added the support for switches. Although it may sound like an insignificant change because it is only a small addition to the existing program, however this is a very important feature for research purposes. In addition to that, supporting switches in the simulation proved to be more challenging that we had initially predicted.
4) We implemented a large update to the simulation algorithm which enables providing only the target current for one or more resistors and then finding the required resistance values. This includes a

new algorithm which chooses “optimal” (non-negative) resistance values where multiple solutions exist.

The validity and the preferability of our solution

As we mentioned before, there is an analogy between microfluidic systems to electronic circuits. Simulation of an electronic circuit requires much less computational resources and can be executed a lot faster on any computer, even a home computer. In that way we can lower the cost of microfluidic systems by solving the analogous problem, and there will be no need for a large financial investment in a supercomputer. An accurate analytic solution of a parallel problem will result in an equivalent solution but a faster one.

Summary

The simulation of microfluidic systems has a problem of complexity of calculation. We propose a solution to this problem by simulation of an electronic circuit which is analogous to the microfluidic system. Our application will run the simulation quickly and provide a friendly yet advanced GUI (Graphical User Interface). Our program can provide a cheap, viable solution to the simulation problem of microfluidic systems.

Detailed Background

Microfluidics is a multidisciplinary field which involves physics, biology, chemistry, programming and more. That is the research of behavior of fluids, usually we talking about a very small scale, and it gives us a way to control experiments in the microscale.

The importance of switches in microfluidic research

The switches enable researching different states of the circuit, so instead of creating many circuits to research something it is possible to do it with a single circuit with switches. Let's say we have microfluidic circuit with three types of fluid and we want to see how they react when we fuse them and to see how they work in different combinations. Instead of creating each circuit separately we can use the same circuit, and programmatically activate/deactivate the switches.

The effect of switches on the cost, time and work of microfluidic research

When we are using a circuit with switches we are using one circuit instead of few different circuits to research the different states, which results in a significantly lower development cost. The ability to model a single circuit with switches also saves time and work for developing the system. The ability to model and simulate all the states at once in our application also helps reduce the time and work required, especially due to the use of a significantly faster electronic circuit simulation algorithm, rather than a traditional microfluidic simulation which requires much more time to compute.

Implementation of switches in our application

While standard circuit simulation algorithms do not support switches, it is possible to add support for them in the steps before the simulation. This process is explained fully in "Analyze and Update" but in a nutshell, it merges the two nodes connected to the switch if the switch is closed. Each state is applied to the circuit, simulated, and then displayed in a tab in the results window. This method of integrating switches into the application is only in effect when needed and does not require special handling during the simulation itself.

Common Algorithms for Simulation of an Electrical Circuit

The most common used in applications for the purpose of simulating an electrical circuit (and finding the voltages and currents in the circuit) are Nodal Analysis and Mesh Analysis. Both of these algorithms rely on Ohm's Law and one or more of Kirchhoff's Circuit Laws - which we will briefly recount before explaining the algorithms themselves.

Basic laws of electronics

Ohm's law

Ohm's law states that the current which flows through a conductor between two points is directly proportional to the voltage across those two points. The constant ratio between the voltage and the current is the resistance, which is independent of the current. The following mathematical equation describes this relation:

$$I = \frac{V}{R}$$

Where I is the current through the conductor in units of amperes, V is the voltage measured across the conductor in units of volts, and R is the resistance of the conductor in units of ohms.

Kirchhoff's circuit laws

Kirchhoff's circuit laws are two equalities that deal with the current and the potential difference (voltage) in an electrical circuit. The first of these, **Kirchhoff's current law (KCL)**, can be defined in the following two equivalent ways:

- At any node (junction) in an electrical circuit, the sum of the currents flowing into that node is equal to the sum of the currents flowing out of that node.
- The algebraic sum of currents in a network of conductors meeting at a point is zero.

Since currents are a signed quantity, with the sign reflecting the direction of the current, the law can be defined algebraically as follows:

$$\sum_{k=1}^{n} I_k = 0$$

Where n is the total number of paths through which current flows towards or away from the node, and I_k is the current flowing through the kth path.

The second of Kirchhoff's circuit laws, **Kirchhoff's voltage law (KVL)**, is defined as follows:

- The directed sum of all of the electrical potential differences (voltages) around any closed network (a loop in the circuit) is zero.

Similarly to KCL, the algebraic definition is:

$$\sum_{k=1}^{n} V_k = 0$$

Where n is the number of voltages measured around said loop, and V_k is the kth voltage measured.

Nodal analysis

Nodal analysis is a method of determining the voltage between nodes (junctions) in an electrical circuit using Kirchhoff's current law combined with Ohm's law. Nodal analysis is possible when all components in the circuit can give current as a function of voltage. For example, a resistor must have a defined conductance in order to give current as a function of voltage: $I_c = V_c \cdot G$, where G is the conductance ($G = \frac{1}{R}$) of the resistor.

The generalized method of nodal analysis:

- Find all of the nodes in the circuit
- Select a reference node, a node which will be the ground reference. The choice does not affect the results.
- Assign a variable for each node whose voltage is unknown.
- For each unknown voltage, form an equation based on Kirchhoff's current law. Basically, add together all currents leaving from the node and mark the sum equal to zero. Current can be represented as voltage divided by resistance, as per Ohm's law.
- If there are voltage sources between two unknown voltages, join the two nodes as a supernode. The currents of the two nodes are combined in a single equation, and a new equation for the voltages is formed.
- Solve the system of simultaneous equations for each unknown voltage.

Nodal analysis can be implemented in software by assigning an index to each node, building a system of linear equations (n equations with n variables, where n is the number of nodes with unknown voltages) in the form $Ax = b$, and then solving it with matrix inversion (using decomposition algorithms):

- The x vector has a length of n and contains our unknown variables: The voltages of the nodes.
- A is an n-by-n matrix and represents the interactions between the passive elements in the circuit as KCL equations. Each row is the KCL equation of the node with the same index as the row. The values contained in this matrix are formed from the conductances of the resistors in the circuit, and the voltages of voltage sources. When the matrix A is multiplied by the x vector the KCL equations are formed with current represented as voltage over resistance, as per Ohm's law.
- The b vector also has a length of n and contains the total current applied to each node by independent current sources, which have a fixed current. Since this is the right-hand side of the equations, the values here are in reverse polarity.
- The resulting system of equations can be solved using a software library for linear algebra (e.g. LAPACK).

Because it uses a compact system of equations, many circuit simulation programs (e.g. SPICE) use nodal analysis as a basis. When elements do not have admittance representations, a more general extension of nodal analysis, modified nodal analysis, can be used.

Modified nodal analysis

Modified nodal analysis (MNA) is an extension of nodal analysis which not only determines the circuit's node voltages (as in classical nodal analysis), but also some branch currents. Modified nodal analysis was developed as a formalism to mitigate the difficulty of representing voltage-defined components in nodal analysis (e.g. voltage-controlled voltage sources).

Just like the standard nodal analysis algorithm, modified nodal analysis can be used to build a system of linear equations, $Ax = b$, however the number of equations in the system is now $n + m$, where n is the number of nodes with an unknown voltage, and m is the number of voltage sources:

- The x vector (which holds the unknown variables):

- Its size is now $n + m$. As in standard nodal analysis, the top n elements hold the node voltages.
- The bottom m elements hold the unknown currents that flow through each voltage source in the circuit.

- The A matrix:
  - Its size is now $n + m$ by $n + m$. The upper left n-by-n submatrix G is almost the same as the A matrix from standard nodal analysis. However, the additional m columns are used to factor in the current from each voltage source, instead of adding the voltages to G (since the voltage may be dependant).
  - The bottom m lines form branch constitutive equations (or BCEs, the voltage-current characteristics of the voltage sources).
- The b vector:
  - Its size is now $n + m$. As in standard nodal analysis, the top n elements hold the total currents applied to each node by independent current sources, which have a fixed current. Since this is the right-hand side of the equations, the values here are in reverse polarity.
  - The bottom m elements hold the voltages of each independent voltage source.

We have chosen to use modified nodal analysis for this application due to its robustness and the ability to calculate the current of each voltage source (branch currents). We will describe our customized implementation of MNA in "The Simulation Algorithms".

Mesh current analysis

Mesh current analysis is an alternative to nodal analysis which is used to find the current loops in the circuit. Mesh current analysis is very similar to nodal analysis, but uses Kirchhoff's voltage law instead of Kirchhoff's current law.

First, find the current flow direction in each part of the circuit. Then, find and define the current loops in the circuit. Now apply Kirchhoff's voltage law on every loop and create a matrix for the system of equations obtained from KVL. Similarly to MNA, the solution can be represented in the form $Ax = b$ where the A matrix is of size nxn while n is the number of the different current loops in the circuit.

The A matrix is built from resistances. Each row is a mesh based on the loop with the same number, and each column is multiplied by the corresponding loop current. In the row k, for each element z in the loop k, its resistance is added to the cell in the (k, k) position. For each element y in loop k that is shared with another loop (j): If the current that goes through y from loop j is in the same direction as the current from the loop k, then the value will be added to (k, j). If the currents flow in opposite directions, the value will be subtracted from (k, j).

Vectors x and b are of size n. Vector x contains the current in each loop (our result), and vector b sums the voltages of voltage sources that are in each loop. (The values are in reverse polarity since this is the right-hand size of the equation. The direction of the voltage source in relation to the loop direction is taken into account: If there is a voltage rise the value is positive, if there is a drop it is negative.)

The Simulation Algorithms

The Simulator class has two functions: ModifiedNodalAnalysis and AnalyzeAndUpdate. The AnalyzeAndUpdate function prepares the circuit for simulation and sends it to the ModifiedNodalAnalysis function, which performs the simulation itself and is where we will begin from.

MNA Algorithm Mathematical Theory

Our algorithm differs from standard Modified Nodal Analysis only with the treatment of resistors with known current but unknown resistance, and the solution of the system of equations when there are multiple possible solutions.

Let's say there are n nodes in the circuit (excluding reference nodes) and m voltage sources. As we explained earlier the Modified Nodal Analysis algorithm builds a system of linear equations in the form $Ax = b$, where A is a matrix which is $n + m$ by $n + m$, and x and b are vectors with a length of $n + m$.

The first n rows of A and the first n values in both vectors actually represent a system of equations where each row k is in the form:

$$\sum_{r \in CR(k)} \frac{V(k) - V\big(O(r,k)\big)}{R(r)} + \sum_{v \in CV(k)} DI(v,k) = \sum_{i \in CI(k)} -DI(i,k)$$

In the first expression:

- $CR(k)$ is the group of resistors with known resistance connected to the node with the index k,
- $R(r)$ is the resistance of the resistor r,
- $V(k)$ is the voltage at the node k,
- $O(r,k)$ is the index of the node other than k which is connected to the resistor r. (Note: the O function is the same as the OtherNode extension method shown later)
- $V\big(O(r,k)\big)$ is the voltage at the above node.

The actual meaning of the expression for a specific r is the difference between the voltages at each side of the resistor r (which gives the voltage on r itself), divided by r's resistance value. This is exactly Ohm's law, so the result will be the current flowing through r. Therefore, if we sum up the results of this for every r in $CR(k)$, the first expression is the total current which flows through resistors connected to the node k.

Note that the polarity of the current matters: If the current is flowing out of the node then the result of the equation $V(k) - V\big(O(r,k)\big)$ will be positive, but if the current flows into the node it will be negative. This will be important later.

This expression is obtained by multiplying the top left n-by-n submatrix of A by the upper n values in the vector x. The latter contains the voltages at each node (which are unknown). The former contains conductances of resistors connected to each node: For each resistor, let's say it is connected to nodes a and b. Its conductance will be added to cells (a,a) and (b,b), and subtracted from (a,b) and (b,a). Since each column is multiplied by the corresponding node voltage, the result will be equal to the first expression.

Note: If, for example b is a reference node, it is enough to only add the conductance of the resistor to (a,a) since $V\big(O(r,a)\big)$ will always be equal to zero.

In the second expression:

- $CV(k)$ is the group of voltage sources connected to
- $DI(v,k)$ is the directional current which flows between the k node and the v voltage source. If k is on the positive side of v this value will be positive, otherwise it will be negative.

Note that as with the first expression, the value is positive when current flows out of the node, and negative when current flows into the node.

This expression is obtained by multiplying the top right n-by-m submatrix of A by the lower m values in x. The latter contains the currents on each voltage source (which are unknown). The former contains values of (1,-1,0) depending on whether each node is connected to the positive side or negative side of each voltage source, or not connected at all. Since each column is multiplied by the corresponding voltage source's current, the result will be equal to the second expression.

The third expression is the same as the second one except that the polarity is reversed, and the components here are the current sources (in our case, resistors with known current but unknown resistance – see below). The polarity must be reversed since this expression is on the other side of the = sign (see below). This expression is obtained directly from the upper n values in the vector b.

How can we treat resistors with known I but unknown R as current sources? The A matrix requires $\frac{1}{R}$ to be known but we don't have that. So in order to complete the equation, since Ohm's law says that $\frac{V}{R} = I$, we used the (known) I value instead, on the other side of the equation (vector b). Since current sources are handled exactly like this in MNA, we can use the same methods. The only thing to note is that the polarity of the resistor matters, which means that the current entered by the user must be relative to the left to right direction (or top to bottom if in vertical orientation).

The entire equation is based on Kirchhoff's current law: The sum of currents flowing into a node is equal to the sum of the currents flowing out of the node. $\sum Iin = \sum Iout$ is the same as $\sum Iin - \sum Iout = 0$, therefore, as long as we move values between the two sides of the equation correctly and use correct polarity, the equation remains a true statement. Modified Nodal Analysis takes advantage of this and puts the values into a matrix by grouping the variables in a way that can be automated easily.

The last m rows in A are the branch constitutive equations, which represent the voltage-current characteristics of the voltage sources. However, in our case the voltage sources have fixed voltages, so for each voltage source s, the row equation will be:

$$VP(s) - VN(s) = V(s)$$

For the voltage source s, $VP(s)$ is the voltage at the node connected to the positive side of the voltage source, $VN(s)$ is the voltage at the node connected to the negative side of the voltage source, and $V(s)$ is the nominal voltage of the voltage source itself. This equation is much simpler – it is trivial: The voltage of s is the voltage difference across s.

Some of you may be asking: Can't we just set $VP(s)$ to $V(s)$ and $VN(s)$ to 0 ahead of time and reduce the number of unknowns that way? The answer is no. Voltages are relative to the reference point (reference node), therefore the equation remains true even with other values. A real life example is in two AA batteries in series: For one of them $VP(1) = 1.5V$ and $VN(1) = 0V$, for the other one $VP(2) = 3V$ and $VN(2) = 1.5V$. In both cases, $V(s) = 1.5V$.

These extra equations are necessary because we need $n + m$ equations in order to solve for $n + m$ unknowns (the current on each voltage source is also unknown). These equations are not implied by any of the previous ones, so all of the rows of A are linearly independent (which makes the system solvable) - unless there are multiple resistors with unknown resistance, in which case our result selection algorithm is used.

The Pseudo-Inverse Result Selection Algorithm

When there are multiple resistors with unknown resistance but known current, there may be an infinite number of solutions to $Ax = b$. We can find all of those solutions by using Moore-Penrose pseudoinversion to get A^+ and then calculating values of x according to this formula:

$$x = A^+b + [I - A^+A]w$$

Where w is an arbitrary vector. Solutions exist if and only if $AA^+b = b$, and if so then A^+ will be a unique solution if $[I - A^+A]$ is a zero matrix. Using arbitrary values of w any of the solutions can be reached.

The problem is that just choosing an arbitrary solution will not work because sometimes the resulting resistance values will be negative, due to impossible node voltages which indicate a voltage rise instead of a drop across a resistor. Therefore we need an algorithm that can choose a value of w which normalizes the resulting node voltages.

As well as wanting to find only non-negative resistance values, it would also be nice to use multiples of the same resistance value wherever possible. Our result selection algorithm solves both of these problems.

The algorithm's basic concept is to find the total voltage drop across the circuit, determine which resistors have a fixed voltage drop (resistance), and to manipulate the result so there will be a multiple of an equal voltage drop across all other resistors (the ones with unknown resistance). The algorithm works like this:

1) Compute A^+ using Moore-Penrose pseudoinverse, confirm that a solution exists ($AA^+b = b$)
2) Compute $s = A^+b$.
3) Compute $F = [I - A^+A]$. If F is a zero matrix, return $x = A^+b$.
4) The F matrix contains factors which can be multiplied by the w vector and added to s in order to get a possible result. If two of the rows in F are linearly-dependent, that means that the two nodes with the same indexes as those rows have a fixed voltage drop between them, as different values of w will not change this dependency in the result.

To find these linearly-dependent pairs of rows in F along with the constant voltage drop, for every column in $\mathrm{Ker}(F)$ with at least two non-zero values, find the indexes of the first two non-zero values i_1 and i_2, and store them along with $s_{i1} - s_{i2}$.

5) Find the node at which to start the algorithm's operation: There is always at least one node h which its voltage cannot be changed with different values in w, as it has a corresponding zero row in F, so its voltage is always s_h. This node must always exist – it is probably on the positive side of a voltage source since a voltage source has a fixed voltage here. (If it is not found, return $x = A^+b$)
6) Create vector d which contains desired voltages for nodes, initialize $d_h = s_h$. Initialize total voltage drop $t = s_h$, and initialize non-zero row node counter i to 0 (Note: i does not count h!). Create vector e which counts number of times to add the final value of m (which will be t/i) to each value of d.
7) Perform breadth-first search (BFS) of circuit (excluding reference nodes), starting from h. For each node $n1$ visited by the search: For each node $n2$ directly connected via a component to $n1$, preform the following steps:
    a. Increment d_{n2} by d_{n1}
    b. Increment e_{n2} by e_{n1}
    c. If there is a linear dependency in the rows of F corresponding to these two nodes, increment both t and d_{n2} by $s_{n2} - s_{n1}$. Otherwise, decrement e_{n2} by 1
    d. Increment i by 1
8) Divide t by i to get the desired voltage drop divisor m
9) The final w vector will be equal to $d - s + me$. Return $x = s + Fw$

The voltage drop divisor is equal to the total voltage drop, minus the fixed voltage drops, divided by the number of non-fixed voltage drops. This gives us a common denominator that we can use to represent any unknown resistors (effectively normalizing the resistance values), while also correcting for any fixed voltages (linear dependencies) in order to prevent invalid results (negative resistance values).

This algorithm has been tested with several complex circuits and it has succeeded in preventing invalid (negative) resistor values in all cases. As for the normalizing of values, as this was not the main focus of the algorithm, it may not give the best result but the resistor values will generally have a common denominator.

The Implementation

The simulation is implemented in a single, non-recursive function, with some supporting preparation code extracted to a separate method (see "Analyze and Update"). Any BFS is implemented with a queue in order to avoid recursion which can be especially slow in the .NET CLR.

In the result selection algorithm, the names of the variables differ from the names used in the explanation above. The corresponding names will be noted where necessary.

```
private static Vector<double> ModifiedNodalAnalysis([NotNull] SimulationCircuit circuit)
{
	if (circuit == null) throw new ArgumentNullException(nameof(circuit));
```

Initialize the A matrix and the b vector:

```
	var a = Matrix<double>.Build.Dense(circuit.NodeCount + circuit.SourceCount,
	                                   circuit.NodeCount + circuit.SourceCount);
	var b = Vector<double>.Build.Dense(circuit.NodeCount + circuit.SourceCount);

	Log.Information("Starting simulation");
```

Save nodes to a lookup table by index (used later on)

```
	var ndict = new Dictionary<int, INode>();
```

Start a BFS on the circuit:

```
	var q = new Queue<INode>();
	q.Enqueue(circuit.Head);
	while (q.Count > 0)
	{
		var n = q.Dequeue();
```

Update lookup table

```
		ndict.Add(n.SimulationIndex, n);
		Log.Information("Visiting node {0}", n.ToString());
```

Check for issues

```
		if (n.SimulationIndex >= circuit.NodeCount)
			throw new SimulationException("Invalid index for node {0}" + n, n);
```

Iterate over components connected to the node

```
		foreach (var c in n.Components)
		{
```

Get node on other side of component

```
			var o = c.OtherNode(n).OrEquivalent();
```

Check for issues

```
                if (o == null)
                {
                        Log.Warning("Component {0} is detached!", c.ToString());
                        continue;
                }
                if (o.SimulationIndex >= circuit.NodeCount)
                        throw new SimulationException("Invalid index for node " + o, o);
```

Add the node to the queue if it hasn't been visited yet

```
                if (!o.Mark && !o.IsReferenceNode)
                {
                        q.Enqueue(o);
                        o.Mark = true;
                }
```

Update A and b based on component type:

```
                if (c is IResistor)
                {
                        var r = (IResistor)c;
                        Log.Information("Visiting resistor {0} connected to node {1}",
r.ToString(), n.ToString());
```

If the resistance is known:

```
                        if (r.Resistance > 0 && !c.Mark)
                        {
                                Log.Information("Resistor {0} has known resistance, adding
to matrix A", r.ToString());
```

Always add conductance to the cell (n, n)

```
                                a[n.SimulationIndex, n.SimulationIndex] += r.Conductance;
```

If o is not a reference node, add the conductance to (o, o) and subtract it from (o, n) and (n, o):

```
                                if (!o.IsReferenceNode)
                                {
                                        a[o.SimulationIndex, o.SimulationIndex] +=
r.Conductance;
                                        a[o.SimulationIndex, n.SimulationIndex] -=
r.Conductance;
                                        a[n.SimulationIndex, o.SimulationIndex] -=
r.Conductance;
                                }
                        }
```

If the resistance is unknown:

```
                        else if (r.Current != 0 && r.Resistance <= 0)
                        {
                                Log.Information("Resistor {0} has known current, adding to
vector B", r.ToString());
```

Add current to b, reverse polarity:

```
                                b[n.SimulationIndex] += c.Node1.OrEquivalent() == n ? -
r.Current : r.Current;
                        }
                }
                else if (c is IVoltageSource)
                {
```

```
                        var v = (IVoltageSource)c;
                        Log.Information("Visiting voltage source {0} connected to node
{1}", v.ToString(), n.ToString());
```

Check for issues

```
                        if (v.SimulationIndex >= circuit.SourceCount)
                                throw new SimulationException("Invalid index for voltage
source " + v, v);
```

Add 1 or -1 to (n, v) and (v, n):

```
                        a[circuit.NodeCount + v.SimulationIndex, n.SimulationIndex] =
a[n.SimulationIndex, circuit.NodeCount + v.SimulationIndex] = Equals(v.Node1?.OrEquivalent(),
n) ? 1 : -1;
```

Add voltage to b if it hasn't been added yet

```
                        if (!v.Mark) b[circuit.NodeCount + v.SimulationIndex] =
v.Voltage;
                    }
                    else if (c is ISwitch && !c.Mark) { Log.Warning("Switch was linked: {0}"
+ c); }
                    c.Mark = true;
                }
                n.Mark = true;
        }
```

Print A and b to the log

```
        Log.Information("The matrix:");
        for (var i = 0; i < a.RowCount; i++) Log.Information("{0}", a.Row(i));
        Log.Information("The vector: {0}", b);
```

Solve the linear equation system according to the result selection algorithm

Note: Since the values used are floating-point, precision issues must be handled. Any value below 10^{-13} is considered to be equal to 0!

Compute A^+ ("s")

```
        var ap = a.PseudoInverse();
```

Check if there actually is a solution ($AA^+b = b$)

```
        if (!b.AlmostEqual(a * ap * b, 1e-13)) throw new SimulationException("No solution
found!");
```

Compute A^+b

```
        var apb = ap * b;
```

Compute $F = [I - A^+A]$

```
        var identity = Matrix<double>.Build.DenseIdentity(a.RowCount, a.ColumnCount);
        var f = identity - ap * a;
```

Create vector x

```
        Vector<double> x;
```

If $F = 0$, then there is a single solution: $x = A^+b$

```
        if (!f.Exists(v => Math.Abs(v) > 1e-13)) { x = apb; }
```

```
        else
        {
```

Clean up values of F (due to precision issue)

```
            for (var i = 0; i < f.RowCount; ++i)
            for (var j = 0; j < f.ColumnCount; ++j) f[i, j] = Math.Round(f[i, j], 13);
```

Get linearly-dependent rows of F and the fixed voltage drop.

This LINQ expression gets pairs of non-zero values in the columns of $Ker(F)$ and saves the required values to a collection of Tuples:

```
            var dep = (from col in f.Kernel()
                       select col.EnumerateIndexed(Zeros.AllowSkip).ToList()
                       into ld
                       where ld.Count >= 2
                       select new Tuple<int, int, double>(ld[0].Item1, ld[1].Item1,
                                   apb[ld[0].Item1] - apb[ld[1].Item1])).ToList();
```

Initialize “d” and “e” (the desired w vector and the multiples of the voltage drop divisor “m”)

```
            var desiredw = Vector<double>.Build.Dense(apb.Count);
            var desiredwDiff = Vector<double>.Build.Dense(apb.Count);
```

Initialize “i”. This is the number of nodes that have a non-zero row in f

```
            var div = 0;
```

Find the starting node “h”.

desiredw is currently zero, so this works to find a zero row in F:

```
            var si = f.EnumerateRowsIndexed().FirstOrDefault(r =>
r.Item2.AlmostEqual(desiredw, 1e-13)).Item1;
            if (si >= circuit.NodeCount)
            {
                   Log.Error("Insufficient data to select optimal result! Falling back to
result vector {0}", apb);
                   x = apb;
                   return x;
            }
            var fn = ndict[si];
```

Initialize “t” to the fixed voltage at fn

```
            var diff = apb[fn.SimulationIndex];
```

And add it to d

```
            desiredw[fn.SimulationIndex] = apb[fn.SimulationIndex];
```

All nodes were marked previously, so treat marked as unmarked and vice versa to save time

```
            fn.Mark = false;
```

Reuse the old queue, which is empty by now

```
            q.Enqueue(fn);
            while (q.Count > 0)
            {
                   var n = q.Dequeue();
```

Traverse through the circuit like in MNA

```
                    foreach (var c in n.Components)
                    {
```

Get the node on the other side

```
                        var o = c.OtherNode(n).OrEquivalent();
```

Don't visit reference nodes and don't visit the same node twice

```
                        if (o.IsReferenceNode || !o.Mark) continue;
```

Copy values from previous node

```
                        desiredw[o.SimulationIndex] += desiredw[n.SimulationIndex];
                        desiredwDiff[o.SimulationIndex] +=
desiredwDiff[n.SimulationIndex];
```

Check if there was a linear dependency between the nodes

```
                        var depInf = dep.FirstOrDefault(d => d.Item1 == n.SimulationIndex
&& d.Item2 == o.SimulationIndex);
                        var depInf2 = dep.FirstOrDefault(d => d.Item2 ==
n.SimulationIndex && d.Item1 == o.SimulationIndex);
                        if (depInf != null)
                        {
```

In this case, the dependency was listed as n,o so Item3 contains $s_n - s_o$ which can be negated to get $s_o - s_n$

```
                            diff -= depInf.Item3;
                            desiredw[o.SimulationIndex] -= depInf.Item3;
                        }
                        else if (depInf2 != null)
                        {
```

In this case, the dependency was listed as o,n so Item3 contains $s_o - s_n$

```
                            diff += depInf2.Item3;
                            desiredw[o.SimulationIndex] += depInf2.Item3;
                        }
                        else
                        {
```

In this case there was no linear dependency. Decrement the multiplier

```
                            desiredwDiff[o.SimulationIndex] -= 1;
                        }
                        q.Enqueue(o);
```

Increment non-zero row counter. This line is down here to avoid counting fn

```
                        ++div;
                        o.Mark = false;
                    }
                }
```

Update $m = t/i$

```
                diff = diff / div;
```

Copy last values corresponding to voltage sources. Not required, but just a precaution

```
            for (var i = 0; i < circuit.SourceCount; ++i)
                desiredw[circuit.NodeCount + i] = apb[circuit.NodeCount + i];
```

Compute final w vector, $w = d - s + me$

```
            desiredw = desiredw - apb + desiredwDiff * diff;
```

Compute the result, $x = s + Fw$

```
            x = apb + f * desiredw;
        }
```

Once again, handle floating-point precision issues

```
        for (var i = 0; i < x.Count; ++i) x[i] = Math.Round(x[i], 13);
        Log.Information("The result vector: {0}", x);
```

Finished! Return the x vector

```
        return x;
}
```

The Models

A quick look at the models used for the simulation: (please read “ECS project” for an in-depth explanation of how the models are used)

- ICircuitObject – contains the essential properties needed for the simulation. Mark, GUID, Name, Simulation Index.
- IComponent – derives from ICircuitObject, contains direct references to two nodes. Node1 is on the left and Node2 is on the right. If the component has polarity, Node1 is positive.
- INode – derives from ICircuitObject, contains properties required for nodes:
    - List of the components that are connected to this node. This list is only used in the simulation process, and is populated in the “Analyze and Update” function.
    - Voltage on this node, in volts
    - A flag that specifies if this node is a reference node
    - EquivalentNode – useful for merging nodes and used when handling switches (see following chapter, “Analyze and Update”)
- IResisrors – derives from IComponent, contains properties required for resistors:
    - Resistance value, in ohms
    - Conductance getter, returns 1/resistance
    - Voltage on this resistor, in volts
    - Current, in amperes
- ISwitch - derives from IComponent, just contains a flag that specifies if this switch is open or closed
- IVoltageSource – derives from IComponent, contains properties for fixed voltage sources: Voltage in volts and current in amperes.

Analyze and Update

The second function, AnalyzeAndUpdate, gets collections of nodes and components, prepares the circuit, passes the circuit to the ModifedNodalAnalysis function, and updates the circuit with the results. We split this function from the ModifiedNodalAnalysis function to make the simulation code more flexible and readable. It can be easier to understand for other people.

This function handles switches. If a switch is open we ignore it continue to the next node, but if the switch is closed then the function will "merge" the nodes it is connected to. The "child" nodes are merged with the "parent" node (arbitrarily chosen), and any nested children are converted to direct children (when there are a few switches in a row).

To merge the nodes, we first reset all the values to default, and then we create a Dictionary of nodes to sets of nodes - a reverse lookup table: For each parent node, provide the list of children. The purpose of the lookup table is to keep track of which nodes must be updated when a parent becomes a child.

Next, the function iterates over each switch in the circuit. If the switch is open it is ignored. If the switch is close, the node on the left will be the parent node. In case this node is already the child of another node, the OrEquivalent extension method is used, which returns the parent of the node or the node itself if the node has no parent. Next, repoint the right node and all of its children to the new parent node both with the EquivalentNode property and in the lookup table, and remove the right node from the lookup table.

The function will give an index to every node and voltage source, and will clear all the unneeded information and sets all the default values for simulation-specific fields.

The code:

```
public static void AnalyzeAndUpdate([NotNull] IEnumerable<INode> nodes,
```

```
                                    [NotNull] IEnumerable<IComponent> components)
{
        var nodesList = nodes.ToList();
        var componentsList = components.ToList();
```

Switch handling begins here. First, reset values to default:

```
        nodesList.ForEach(n => n.EquivalentNode = null);
```

Create the lookup table:

```
        var equiv = new Dictionary<INode, HashSet<INode>>();
```

For each switch:

```
        foreach (var sw in componentsList.OfType<ISwitch>())
        {
                if (!sw.IsClosed) continue;
```

Choose parent node using first node and OrEquivalent:

```
                var p = sw.Node1.OrEquivalent();
```

Update sw.Node2:

```
                sw.Node2.EquivalentNode = p;
```

Add p to the lookup table:

```
                if (!equiv.ContainsKey(p)) equiv.Add(p, new HashSet<INode>());
                equiv[p].Add(sw.Node2);
```

If sw.Node2 is a parent of other nodes:

```
                if (equiv.ContainsKey(sw.Node2))
                {
```

Repoint all children of sw.Node2 to p:

```
                        foreach (var n in equiv[sw.Node2])
                        {
                                n.EquivalentNode = p;
                                equiv[p].Add(n);
                        }
```

Remove sw.Node2 from lookup table (no longer a parent):

```
                        equiv.Remove(sw.Node2);
                }
        }
```

Index assignment begins here.

```
        int vsId = 0, nId = 0;
        INode h = null, refNode = null;
```

Prepare the nodes:

```
        foreach (var n in nodesList)
        {
```

Clear previous links

```
                n.Components.Clear();
```

Clear mark

```
            n.Mark = false;
```

Set default simulation index

```
            n.SimulationIndex = -1;
```

Skip redundant nodes

```
            if (n.EquivalentNode != null) continue;
```

Set simulation index if not a reference node

```
            if (!n.IsReferenceNode) n.SimulationIndex = nId++;
```

Get first non-reference node, which will be the first node to be visited in the simulation:

```
            if (!n.IsReferenceNode && h == null) h = n;
```

Make sure we have at least one reference node:

```
            else if (n.IsReferenceNode && refNode == null) refNode = n;
        }
        if (h == null) throw new InvalidOperationException("Missing non-reference node!");
        if (refNode == null) throw new InvalidOperationException("Missing reference node!");
```

Create reverse links to components:

Add component to nodes on each side, and at the same time assign indexes to voltage sources

```
        foreach (var c in componentsList.Where(i => !(i is ISwitch)))
        {
```

Clear mark

```
            c.Mark = false;
```

Create relevant links

```
            c.Node1.OrEquivalent()?.Components?.Add(c);
            c.Node2.OrEquivalent()?.Components?.Add(c);
```

Assign an index

```
            if (c is IVoltageSource) c.SimulationIndex = vsId++;
        }
```

Circuit is ready for simulation!

```
        var circuit = new SimulationCircuit(h, nId, vsId);
```

Perform the actual simulation here:

```
        var result = ModifiedNodalAnalysis(circuit);
```

Begin updating circuit with the results from the simulation:

```
        Log.Information("Updating circuit elements");
```

Input voltages at nodes from result vector:

```
        foreach (var n in nodesList)
        {
                if (n.SimulationIndex <= -1) continue;
                n.Voltage = result[n.SimulationIndex];
                Log.Information("Voltage at node {0}: {1}", n.ToString(), n.Voltage);
        }
```

Update child nodes

```
        foreach (var e in equiv)
```

Copy parent node's voltage to each child node

```
        foreach (var n in e.Value)
        {
                n.Voltage = e.Key.Voltage;
                Log.Information("Voltage at node {0} is the same as at node {1}: {2}",
                                e.Key.ToString(), n.ToString(), n.Voltage);
        }
```

Update component information

```
        foreach (var c in componentsList)
        if (c is IVoltageSource)
        {
```

Input computed current

```
                var v = (IVoltageSource)c;
```

Result is in opposite direction due to the reverse direction of current inside the voltage source. Reverse it.

```
                v.Current = -result[circuit.NodeCount + v.SimulationIndex];
                Log.Information("Current at voltage source {0}: {1}", v.ToString(), v.Current);
        }
        else if (c is IResistor)
        {
```

Input computed voltage and current or resistance, depending which was known at the start

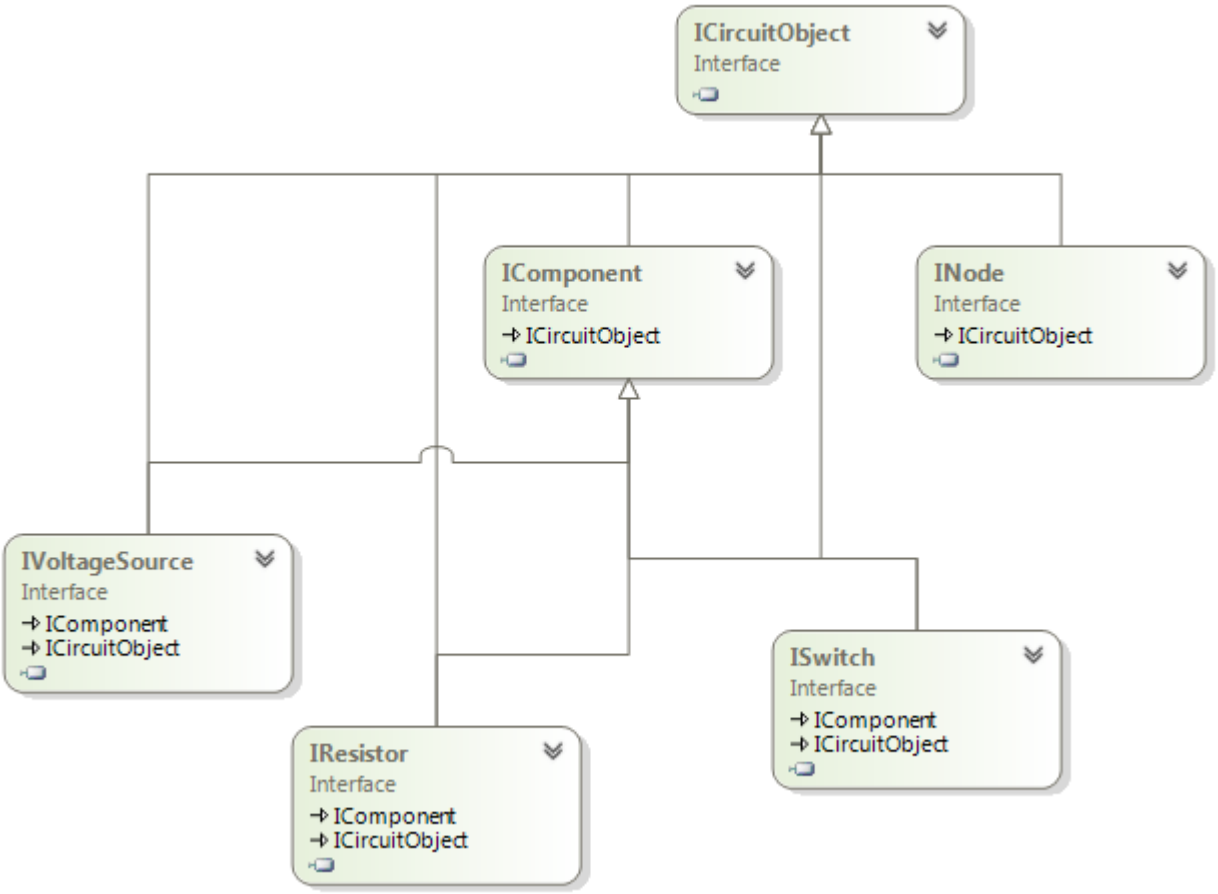
```
                var r = (IResistor)c;
                r.Voltage = (r.Node1?.Voltage ?? 0) - (r.Node2?.Voltage ?? 0);
                if (r.Resistance > 0) r.Current = r.Voltage / r.Resistance;
                else
                {
                        r.Resistance = r.Voltage / r.Current;
```

If the resulting resistance is negative, something went wrong with the values

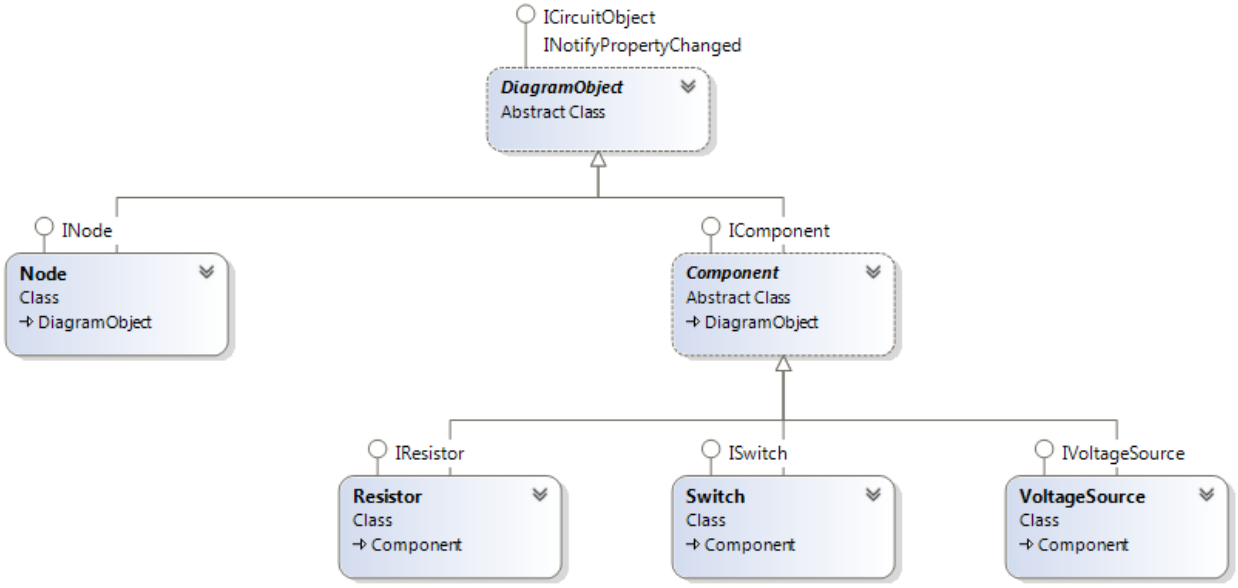
```
                        if (r.Resistance < 0) throw new SimulationException("Invalid current at
resistor! " + r, r);
                }
        }
}
```

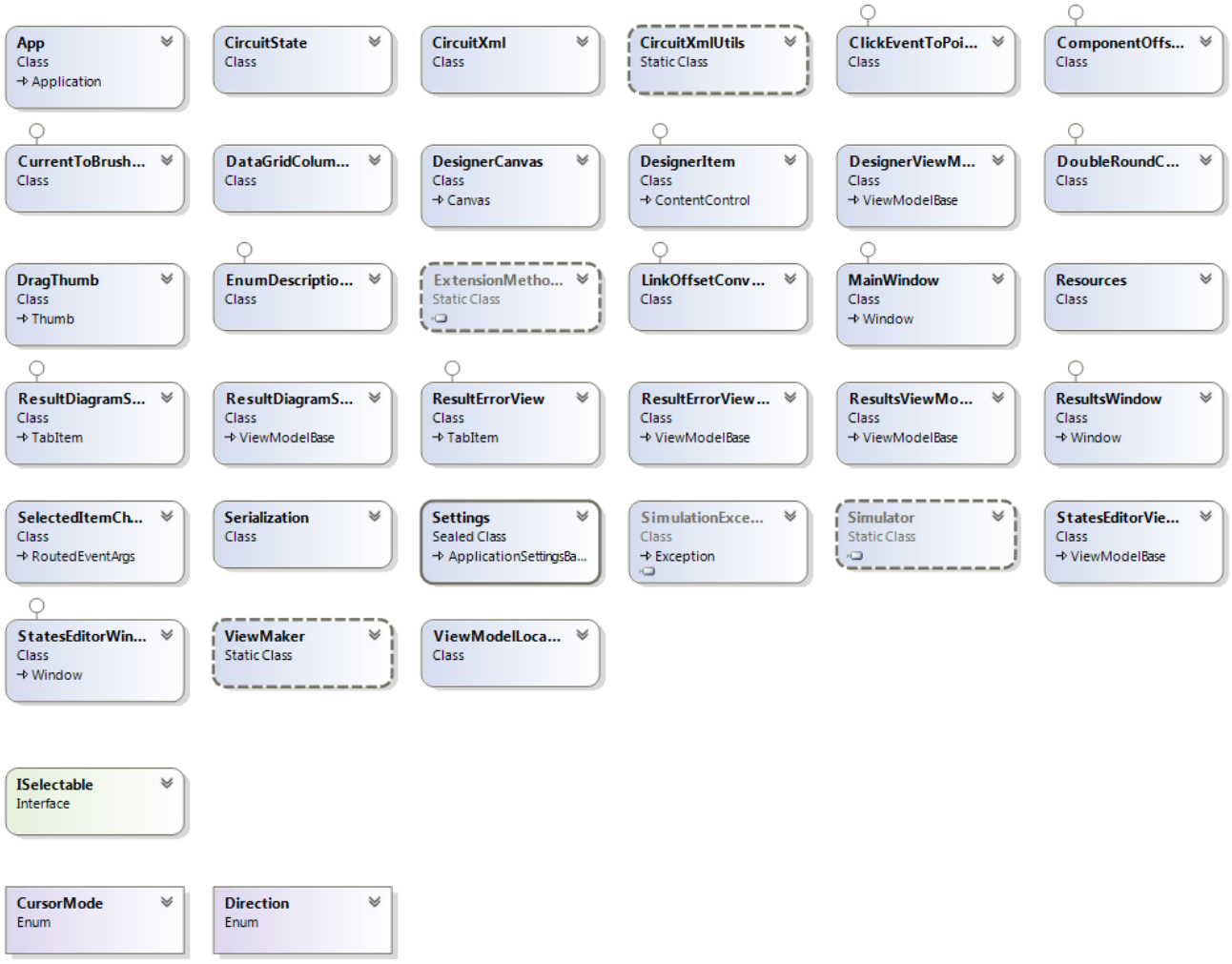
UML Class Diagram

ECS.Core project:



ECS project:




App
Class
Application
CircuitState
Class
CircuitXml
Class
CircuitXmlUtils
Static Class
ClickEventToPoi...
Class
ComponentOffs...
Class
CurrentToBrush...
Class
DataGridColum...
Class
DesignerCanvas
Class
Canvas
DesignerItem
Class
ContentControl
DesignerViewM...
Class
ViewModelBase
DoubleRoundC...
Class
DragThumb
Class
Thumb
EnumDescriptio...
Class
ExtensionMetho...
Static Class
LinkOffsetConv...
Class
MainWindow
Class
Window
Resources
Class
ResultDiagramS...
Class
TabItem
ResultDiagramS...
Class
ViewModelBase
ResultErrorView
Class
TabItem
ResultErrorView...
Class
ViewModelBase
ResultsViewMo...
Class
ViewModelBase
ResultsWindow
Class
Window
SelectedItemCh...
Class
RoutedEventArgs
Serialization
Class
Settings
Sealed Class
ApplicationSettingsBa...
SimulationExce...
Class
Exception
Simulator
Static Class
StatesEditorVie...
Class
ViewModelBase
StatesEditorWin...
Class
Window
ViewMaker
Static Class
ViewModelLoca...
Class
ISelectable
Interface
CursorMode
Enum
Direction
Enum

ECS project

Our model does not follow one of the more common and simpler patterns. Instead, we have defined the model interfaces in ECS.Core (in the ECS.Core.Model namespace) and implemented those interfaces in the ECS project (in the ECS.Model namespace). We chose to do this for a few reasons:

- The first reason was to make the code easier to reuse in the future, to avoid more refactoring if/when we would need to reuse a lot of the code in a different application, or with a different front-end and/or UI.
- The second reason was to avoid Multiple Inheritance: On the User Interface side each component derives from an abstract class which contains common functionality for the designer (DiagramObject), and using an interface instead would have required to duplicate several lines of code (some of which have changed during development), so we decided to expose a more flexible API for simulation.
- Another alternative we had was to use a separate model on the User Interface side. While this would have simplified things a lot in some places, it would have required conversion methods between the two models - which can be long and repetitive - and can require many more changes in the event of a minor change to one of the models, in some cases.

The ECS.Core.Model namespace also contains XML-related classes, which provide methods of converting circuits to XML. These methods were implemented using XmlSerializer, which is part of the .NET Framework, and can serialize or deserialize objects with minimal coding effort. The model implementations contain XML Attributes throughout, in order to control whether properties should be serialized to XML or not, and the representation of certain data elements in the XML.

Model Implementation

Let's begin with the implementations of the model interfaces, and the other classes in the ECS.Core.Model namespace:

- CircuitState, which represents a single state of the switches in the circuit:
  - The SwitchStates property is a dictionary of all the switches and their states (open or closed, represented by a Boolean value).
  - String to hold the name.
  - In addition, two important static functions are implemented:
  - Serialize: Converts a list of CircuitStates to the CSV (Comma Separated Values) format. The states are effectively stored as a table, where each row is a state and each column is the state of a specific switch. The columns are designated by the ID of the switch.
    The code:

```
public static string Serialize(IEnumerable<CircuitState> states)

{
        var statesL = states.ToList();
        var usedIds = statesL.SelectMany(s => s.SwitchStates.Keys).Distinct().ToList();
        var result = "[name]," + string.Join(",", usedIds) + "\n";
        var lines = statesL.Select(s => s.Name + "," + string.Join(",", usedIds.Select(i =>
        !s.SwitchStates.ContainsKey(i)
        ? "*"
        : s.SwitchStates[i]
        ? "1"
        : "0")));
        result += string.Join("\n", lines);
```

```
        return result;
}
```

- Deserialize: The reverse of Serialize. Parses the CSV and returns them as a list of CircuitStates.

```
public static IEnumerable<CircuitState> Deserialize(string csv, List<Guid> usedIds = null)
{
    var lines = csv.Split('\n');
    if (usedIds == null) usedIds = lines[0].Split(',').Skip(1).Select(Guid.Parse).ToList();
    return from line in lines.Skip(1)
        let st = line.Split(',')
        select new CircuitState
        {
              Name = st[0],
              SwitchStates = usedIds
              .Select((i, n) => st[n + 1] == "*" ? null : new { Id = i, State = st[n + 1] ==
        "1" })
              .Where(g => g != null)
              .ToDictionary(g => g.Id, g => g.State)
         };
}
```

- Component: Simple implementation of the IComponent interface. Derives from DiagramObject. Adds the following new properties:
  - Direction: An enumeration value that determines whether the component should be displayed vertically or horizontally.
  - Node1Id and Node2Id: Helper properties for the XML serialization process. (see the explanation of the XML serialization code)
- Node: Implementation of INode, derives from DiagramObject. Hides properties not used in the UI side – INode.Components and INode.EquivalentNode.
- Resistor: Implementation of IResistor, derives from Component.
- Switch: Implementation of ISwitch, derives from Component.
- VoltageSource: Implementation of IVoltageSource, derives from Component.
- DiagramObject: The basis of all objects displayable is the designer. Implements ICircuitObject, and INotifyPropertyChanged so it can automatically update values displayed in the GUI. It adds the following new functionalities:
  - X and Y properties: The coordinates at which it will be displayed in the designer.
  - Overrides for GetHashCode and Equals: To enable direct comparisons between two DiagramObjects by value (as opposed to by reference).
- And now, the classes within ECS.Core.Model.Xml:
- The first class is CircuitXml. This class represents the structure of the resulting XML file, and it contains all the data which must be saved. There is a list for Nodes, Resistors, VoltageSources, and Switches. The constructor initializes these four lists.
- The second class is CircuitXmlUtils, and it has three functions:
  - The first function ToDiagram, an extension method which prepares a freshly deserialized circuit for use in the application. In short, it updates the Node1 and Node2 properties of each component with the actual Node objects, based on the Node1Id and Node2Id properties which were present in the XML data. The reason this is necessary is that Node1 and Node2 cannot be saved as objects in the XML since this would cause their data to be duplicated across all components that are connected to them, so instead they are saved separately, and their IDs are used to reference them from a component.

```
public static IEnumerable<DiagramObject> ToDiagram(this CircuitXml cx)
{
    var nodes = cx.Nodes.Where(n => n != null).ToDictionary(n => n.Id);
    foreach (var n in nodes.Values) yield return n;
    Node ln;
    foreach (var r in cx.Resistors.Where(r => r != null))
    {
        if (r.Node1Id != null && nodes.TryGetValue(r.Node1Id.Value, out ln)) r.Node1 = ln;
        if (r.Node2Id != null && nodes.TryGetValue(r.Node2Id.Value, out ln)) r.Node2 = ln;
        yield return r;
    }
    foreach (var v in cx.VoltageSources)
    {
        if (v.Node1Id != null && nodes.TryGetValue(v.Node1Id.Value, out ln)) v.Node1 = ln;
        if (v.Node2Id != null && nodes.TryGetValue(v.Node2Id.Value, out ln)) v.Node2 = ln;
        yield return v;
    }
    foreach (var s in cx.Switches)
    {
        if (s.Node1Id != null && nodes.TryGetValue(s.Node1Id.Value, out ln)) s.Node1 = ln;
        if (s.Node2Id != null && nodes.TryGetValue(s.Node2Id.Value, out ln)) s.Node2 = ln;
        yield return s;
    }
}
```

- The second function is ToCircuitXml, which simply creates a CircuitXml object from a circuit (which was represented as a collection of DiagramObjects). To do this, the lists of items are populated by type.

```
public static CircuitXml ToCircuitXml(IEnumerable<DiagramObject> diagramObjects)

{
    var cx = new CircuitXml();
    cx.Nodes.AddRange(diagramObjects.OfType<Node>());
    cx.Resistors.AddRange(diagramObjects.OfType<Resistor>());
    cx.VoltageSources.AddRange(diagramObjects.OfType<VoltageSource>());
    cx.Switches.AddRange(diagramObjects.OfType<Switch>());
    return cx;
}
```

- The last function is MaxDefaultId, which finds the last name automatically assigned to a certain kind of element in the circuit by this application. This extension method is used when loading a circuit, to determine what the default names of the next added elements should be. (See the explanation of the toolbox area in the main window)

```
public static int MaxDefaultId<T>([NotNull] this IEnumerable<T> collection, [NotNull] string
prefix = "")
where T : ICircuitObject
{
    Try
    {
        return collection.Max(n =>
        {
                var m = Regex.Match(n.Name, @"$" + Regex.Escape(prefix) + @"([1-9][0-9]*)");
                return m.Success ? int.Parse(m.Groups[0].Value) : 0;
        });
    }
    // If the collection has no elements:
    catch (InvalidOperationException) { return 0; }
}
```

- The third and final class is Serialization, which handles the actual XML serialization. This class has the following functions:
    - The Serialize function has two overloads. The first one outputs the XML directly to a Stream, and the second outputs to a string using a StringWriter.

```
public void Serialize([NotNull] CircuitXml cx, [NotNull] Stream s)
{
	if (cx == null) throw new ArgumentNullException(nameof(cx));
	if (s == null) throw new ArgumentNullException(nameof(s));
	_ser.Serialize(s, cx);
}

public string Serialize([NotNull] CircuitXml cx)
{
	if (cx == null) throw new ArgumentNullException(nameof(cx));
	string r;
	using (var sw = new StringWriter())
	{
		_ser.Serialize(sw, cx);
		r = sw.ToString();
	}
	return r;
}
```

    - The Deserialize function also has two overloads, and similarly to the Serialize function, the first one reads XML from a Stream and the second one reads from a string using a StringReader.

```
public CircuitXml Deserialize([NotNull] Stream s)
{
	if (s == null) throw new ArgumentNullException(nameof(s));
	return (CircuitXml)_ser.Deserialize(s);
}

public CircuitXml Deserialize([NotNull] string s)
{
    if (s == null) throw new ArgumentNullException(nameof(s));
    CircuitXml cx;
    using (var sr = new StringReader(s)) { cx = (CircuitXml)_ser.Deserialize(sr); }
    return cx;
}
```

WPF User Interface: XAML, ViewModels, Layouts, Converters and Behaviors

- Our user interface is build using the Windows Presentation Foundation (WPF), a powerful framework for building complex GUIs on Windows with data bindings, lookless controls, animations, comprehensive styling and templating, vector graphics and more. It also is well suited for use with the Model-View-ViewModel design pattern, which is the one we have used here.
- We have used the MVVM Light Toolkit by GalaSoft, an open-source library released under the permissive MIT license, which includes the System.Windows.Interactivity library from Microsoft (from the Blend 4 SDK, which is redistributable). Together these libraries simplify the use of Commands in MVVM, and add support for Behaviors in WPF. We have also used the Extended WPF Toolkit Community Edition by Xceed, also an open-source library released under the Ms-PL, which adds several extra controls for WPF.
- Small portions of the code here are based on the work of others (released under the public domain or a permissive open-source license).

ECS.Layout

- This namespace contains the designer controls, and contains four classes:
- The first and most important class is DesignerCanvas, which inherits from Canvas. This is the designer control itself. This class contains many functions, properties and fields:
    - The field _items contains the elements currently displayed in the designer. It maps each DiagramObject to a DesignerItem which contains it (see description of the DesignerItem class).
    - Two DependencyProperties: ItemsSourceProperty, which contains a reference to the collection of elements in the designer, and SelectedItemProperty, which contains the element that is currently selected. They have corresponding PropertyChangedCallbacks: OnItemsSourceChanged and OnSelectedItemChanged, as well as getters and setters.
    - The callback function OnItemsSourceChanged handles a replacement of the collection of elements. It removes all the elements of the old collection and adds all of the new ones. It also handles registration and unregistration of the ObservableCollectionChanged callback, if the collection implements INotifyCollectionChanged.

```
private static void OnItemsSourceChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
{
    var c = d as DesignerCanvas;
    if (c == null) return;

    var l = e.OldValue as IEnumerable<DiagramObject>;
    if (l != null) foreach (var i in l) c.RemoveItem(i);

    var ol = l as INotifyCollectionChanged;
    if (ol != null) ol.CollectionChanged -= c.ObservableCollectionChanged;

    var n = e.NewValue as IEnumerable<DiagramObject>;
    if (n != null) foreach (var i in n) c.AddItem(i);

    var on = n as INotifyCollectionChanged;
    if (on != null) on.CollectionChanged += c.ObservableCollectionChanged;
}
```

    - The callback function ObservableCollectionChanged is called when there is a change in the collection of elements. If it is a Reset action then all old values will be removed, and all values in the collection will be added again. If it is a different action then all OldItems are removed and all NewItems are added.

```
private void ObservableCollectionChanged(object sender, NotifyCollectionChangedEventArgs
args)
{
    if (args.Action == NotifyCollectionChangedAction.Reset)
    {
        foreach (var p in _items)
        {
               if (p.Key != null && p.Value != null)
               {
                      BindingOperations.ClearBinding(p.Value, LeftProperty);
                      BindingOperations.ClearBinding(p.Value, TopProperty);
               }
               Children.Remove(p.Value);
        }
        _items.Clear();
        foreach (var i in (IEnumerable)sender) AddItem(i as DiagramObject);
        }
        else
        {
            if (args.OldItems != null) foreach (var i in args.OldItems) RemoveItem(i as
        DiagramObject);
            if (args.NewItems != null) foreach (var i in args.NewItems) AddItem(i as
        DiagramObject);
        }
}
```

- The function AddItem handles adding a new element to the designer. It gets the correct DataTemplate from the resources, loads the content of the element, adds it to _items, sets Bindings for its position in the designer (X and Y properties to Canvas.Left and Canvas.Top) and finally displays it in the designer.

```
private void AddItem(DiagramObject o)
{
    if (o == null){
        Log.Warning("Null object detected in designer!");
        return;
    }
    var i = new DesignerItem { DataContext = o };
    if (Resources != null){
        var dt = (DataTemplate)Resources[new DataTemplateKey(o.GetType())];
        i.Content = dt.LoadContent();
    }
    _items.Add(o, i);
    var dobj = o;
    if (dobj != null)
    {
        BindingOperations.SetBinding(i, LeftProperty,
               new Binding(nameof(DiagramObject.X)){
               Source = dobj,
               Mode = BindingMode.TwoWay
               });
        BindingOperations.SetBinding(i, TopProperty,
        new Binding(nameof(DiagramObject.Y)){
               Source = dobj,
               Mode = BindingMode.TwoWay
               });
    }
    Children.Add(i);
}
```

- The function RemoveItem handles removing an element from the designer. It removes it from _items, clears the Bindings of its position and removes it from the designer.

```
private void RemoveItem(DiagramObject o)
{
    if (o == null || !_items.ContainsKey(o)) return;

    var i = _items[o];
    _items.Remove(o);
    var dobj = o;
    if (dobj != null)
    {
        BindingOperations.ClearBinding(i, LeftProperty);
        BindingOperations.ClearBinding(i, TopProperty);
    }
    Children.Remove(i);
}
```

- A RoutedEvent named SelectedItemChangedEvent. This event is registered as Direct (without Bubbling or Tunneling), and is raised when the selected element in the circuit has changed. There are also add/remove handle functions for it, a function to raise the event (RaiseSelectedItemChangedEvent), and an extension of RoutedEventArgs for it, named SelectedItemChangedEventArgs, which contains the new value and the previous value.
- The function OnSelectedItemChanged updates the IsSelected property of elements (which highlights the element, see description of the ISelectable interface) whenever a different one is selected.

```
private static void OnSelectedItemChanged(DependencyObject d
DependencyPropertyChangedEventArgs e)
{
       if (!(d is DesignerCanvas)) return;
       var dc = (DesignerCanvas)d;
       DesignerItem i;
       if (e.OldValue is DiagramObject && dc._items.TryGetValue((DiagramObject)e.OldValue,
out i))
       i.IsSelected = false;
       if (e.NewValue is DiagramObject && dc._items.TryGetValue((DiagramObject)e.NewValue,
out i))
       i.IsSelected = true;
       dc.RaiseSelectedItemChangedEvent(e.OldValue, e.NewValue);
}
```

- An override of MeasureOverride which changes the behavior of the DesignerCanvas when measuring the required space for it in the UI, given a size constraint. To do this, it takes position of each element, measures the size of the element, and adds these values. Then, it finds the maximum X and Y values across all elements, effectively finding the furthest point right and the furthest point down at which an element is occupying the space, which will give us the size needed to contain all of the elements.

```
protected override Size MeasureOverride(Size constraint)
{
       var size = new Size();

       foreach (UIElement element in InternalChildren)
       {
              var left = GetLeft(element);
              var top = GetTop(element);
              left = double.IsNaN(left) ? 0 : left;
              top = double.IsNaN(top) ? 0 : top;

              // Measure desired size for each child
              element.Measure(constraint);
```

```
            var desiredSize = element.DesiredSize;
            if (double.IsNaN(desiredSize.Width) || double.IsNaN(desiredSize.Height))
        continue;
            size.Width = Math.Max(size.Width, left + desiredSize.Width);
            size.Height = Math.Max(size.Height, top + desiredSize.Height);
        }
        // Add margins
        size.Width += 10;
        size.Height += 10;
        return size;
}
```

    - The override of OnPreviewMouseDown updates SelectedItem whenever the user clicks on an element inside the designer.
- The class DragThumb derives from Thumb, and enables dragging the elements inside the designer.
    - The callback function DragThumb_DragDelta is registered in the constructor. It updates the position according to the distance it was dragged by the users and limits the movement so the element won't leave the designer area.

```
private void DragThumb_DragDelta(object sender, DragDeltaEventArgs e)
{
        var designerItem = DataContext as DesignerItem;
        var designer = VisualTreeHelper.GetParent(designerItem) as DesignerCanvas;
        if (designer == null || designer.SelectedItem != designerItem.DataContext) return;

        // Get current position of item
        var left = Canvas.GetLeft(designerItem);
        var top = Canvas.GetTop(designerItem);
        left = double.IsNaN(left) ? 0 : left;
        top = double.IsNaN(top) ? 0 : top;

        // Prevent out-of-bounds at top/left
        var deltaHorizontal = Math.Max(-left, e.HorizontalChange);
        var deltaVertical = Math.Max(-top, e.VerticalChange);

        // Prevent out-of-bounds at bottom/right, and make sure the item will stay visible
        deltaHorizontal = Math.Min(deltaHorizontal, designer.Width - left - 24);
        deltaVertical = Math.Min(deltaVertical, designer.Height - top - 24);

        // Update position
        Canvas.SetLeft(designerItem, left + deltaHorizontal);
        Canvas.SetTop(designerItem, top + deltaVertical);

        designer.InvalidateMeasure();
        e.Handled = true;
}
```

- The class DesignerItem inherits from ContentControl and ISelectable. It is a skeleton of an item shown on the designer, and its appearance is controlled by Templates and Styles. Every DiagramObject in the designer is displayed within a DesignerItem using a matching DataTemplate, thus separating the WPF-specific code from the serializable data.
    - The callback function DesignerItem_Loaded loads the templated parts of the DesignerItem.

```
private void DesignerItem_Loaded(object sender, RoutedEventArgs e)
{
    var contentPresenter = Template?.FindName("PART_ContentPresenter", this) as
ContentPresenter;
    if (contentPresenter == null) return;
    var contentVisual = VisualTreeHelper.GetChild(contentPresenter, 0) as UIElement;
```

```
    if (contentVisual == null) return;
    var thumb = Template.FindName("PART_DragThumb", this) as DragThumb;
    if (thumb == null) return;
    var template = GetDragThumbTemplate(contentVisual);
    if (template != null) thumb.Template = template;
}
```

- The interface ISelectable, as the name suggests, enables selecting an object. It has only one Boolean property named IsSelected. In the templates, this property is used to control whether an element is highlighted.

The DesignerItem and DragThumb Styles

The default Styles of DesignerItem and DragThump are defined in App.xaml. Default values are set for the MinWidth, MinHeight and SnapsToDevicePixels properties:

```
<Style TargetType="{x:Type s:DesignerItem}">
       <Setter Property="MinWidth" Value="10" />
       <Setter Property="MinHeight" Value="10" />
       <Setter Property="SnapsToDevicePixels" Value="True" />
```

Next the default template is set:

```
<Setter Property="Template">
       <Setter.Value>
              <ControlTemplate TargetType="{x:Type s:DesignerItem}">
```

And the required parts are added:

```
<Grid DataContext="{Binding RelativeSource={RelativeSource TemplatedParent}}">
       <!-- PART_ContentPresenter -->
              <ContentPresenter x:Name="PART_ContentPresenter"
                                HorizontalAlignment="Stretch"
                                VerticalAlignment="Stretch"
                                Content="{TemplateBinding ContentControl.Content}"
                                Margin="{TemplateBinding ContentControl.Padding}" />
       <!-- PART_DragThumb -->
       <s:DragThumb x:Name="PART_DragThumb"
                    Cursor="SizeAll" />
</Grid>
```

The DragThumb style is quite simple, it only contains a template with a transparent rectangle which is the area that can be dragged:

```
<Style TargetType="{x:Type s:DragThumb}">
       <Setter Property="Template">
              <Setter.Value>
                     <ControlTemplate TargetType="{x:Type s:DragThumb}">
                            <Rectangle Fill="Transparent" />
                     </ControlTemplate>
              </Setter.Value>
       </Setter>
</Style>
```

The Element Templates

After explaining the Logical structure of our designer layout, as well as the high-level Visual structure of element containers (DesignerItems), we shall elaborate on how the elements are actually styled and templated - the Visual structure. The ElementTemplates XAML file contains a ResourceDictionary with everything needed to use the DesignerCanvas.

First, here are some common styles and templates. The following style colors a shape in black, but changes the color to blue if it is selected:

```
<Style TargetType="Shape" x:Key="SelectFillStyle">
	<Setter Property="Fill" Value="Black" />
	<Setter Property="Stroke" Value="Black" />
	<Style.Triggers>
		<DataTrigger Binding="{Binding IsSelected,
					RelativeSource={RelativeSource FindAncestor,
							AncestorType={x:Type s:DesignerItem}}}"
				Value="True">
			<Setter Property="Fill" Value="CornflowerBlue" />
			<Setter Property="Stroke" Value="CornflowerBlue" />
		</DataTrigger>
	</Style.Triggers>
</Style>
```

The following style (LinkStyle) sets the color, thickness and shape of a Line (a conductor in the circuit):

```
<Style TargetType="Line" x:Key="LinkStyle">
	<Setter Property="Stroke"
		Value="{Binding Current, Converter={StaticResource CurrentToColorConv},
			FallbackValue=Black}" />
	<Setter Property="StrokeThickness" Value="2" />
	<Setter Property="StrokeStartLineCap" Value="Round" />
	<Setter Property="StrokeEndLineCap" Value="Round" />
</Style>
```

The color of the conductor changes based on the current flowing through it, using the converter CurrentToColorConv, a CurrentToBrushConverter defined in App.xaml:

```
<converters:CurrentToBrushConverter x:Key="CurrentToColorConv" MinColor="Green"
				MidColor="Yellow" MaxColor="Red" NullColor="Black"
				MinValue="0" MaxValue="1" />
```

CurrentToBrushConverter is an IValueConverter which generates a Brush based the current and predefined colors and values. The color will change from MinColor to MidColor to MaxColor, between the values MinValue and MaxValue. If the value is 0, NullColor is used.

The reason that we added MidColor is to make the color gradient make more sense visually. If we would have generated colors between red and green then the color at 50% would be brown (RGB: 127,127,0) while we want it to be yellow.

```
public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
{
	var v = Math.Abs(value as double? ?? 0.0);
	if (double.IsNaN(v)) v = 0.0;
	if (v < 1e-13) return new SolidColorBrush(NullColor);
	v = Math.Max(v, MinValue);
	v = Math.Min(v, MaxValue);
	var factor = 2 * (v - MinValue) / (MaxValue - MinValue);
	return new SolidColorBrush(new Color
		{
			A = 255,
			R = GenColorValue(MinColor.R, MidColor.R, MaxColor.R, factor),
			G = GenColorValue(MinColor.G, MidColor.G, MaxColor.G, factor),
			B = GenColorValue(MinColor.B, MidColor.B, MaxColor.B, factor)
		});
}
```

After checking all the values, the Convert function converts the value to a factor which represents its linear position between MinValue and MaxValue, where 0 is MinValue, 1 is MidValue, and 2 is MaxValue. Then it uses GenColorValue to apply this factor to each channel in the min/mid/max colors.

The GenColorValue function applies the factor by multiplication. If the factor is greater than 1, then the result will be between mid and max. Otherwise, it will be between min and mid:

```
private byte GenColorValue(byte min, byte mid, byte max, double factor)
{
	return factor > 1.0 ? (byte)(mid + (max - mid) * (factor - 1.0))
			    : (byte)(min + (mid - min) * factor);
}
```

Next is a template which is the basis for all Components in the circuit (not including nodes). The reason it's needed to define the height and width is to update the size of the DragThumb. The component name is shown above the component icon, and the electrical properties are shown below:

```
<ControlTemplate TargetType="ContentControl" x:Key="CircuitComponentTemplate">
	<Grid x:Name="Outer" ClipToBounds="False">
		<Canvas x:Name="Host"
			Height="{Binding Content.Height, ElementName=ComponentIcon}"
			Width="{Binding Content.Width, ElementName=ComponentIcon}">
			<TextBlock Canvas.Bottom="{Binding Content.Height,
							ElementName=ComponentIcon}"
				MinWidth="{Binding Content.Width, ElementName=ComponentIcon}"
				TextAlignment="Center" Text="{Binding Name}" />
			<ContentControl x:Name="ComponentIcon" Content="{TemplateBinding
								ContentControl.Content}" />
			<ContentControl Canvas.Top="{Binding Content.Height,
							ElementName=ComponentIcon}"
					MinWidth="{Binding Content.Width,
							ElementName=ComponentIcon}"
					Content="{TemplateBinding Tag}" />
		</Canvas>
```

The following part of the template draws a line from the positive end of a component to a node. The coordinates are bound using MultiBindings and converters:

```
<Line x:Name="Line1" X1="0" Y1="{Binding Content.Height, ElementName=ComponentIcon,
				Converter={StaticResource LinkOffsetConv},
				ConverterParameter={StaticResource LinkOffsetH}}">
	<Line.X2>
		<MultiBinding Converter="{StaticResource ComponentOffsetConverter}"
				ConverterParameter="{StaticResource NodeOffset}" Mode="OneWay">
			<Binding Path="X" />
			<Binding Path="Node1.X" />
		</MultiBinding>
	</Line.X2>
	<Line.Y2>
		<MultiBinding Converter="{StaticResource ComponentOffsetConverter}"
				ConverterParameter="{StaticResource NodeOffset}" Mode="OneWay">
			<Binding Path="Y" />
			<Binding Path="Node1.Y" />
		</MultiBinding>
	</Line.Y2>
```

The ComponentOffsetConverter is used to calculate where the line ends, or the position of a node relative to a component. It simply returns the difference between the coordinates of the node and the component, plus a predefined offset (to make sure that the line ends in the center of the node as opposed to the top left corner):

```
public class ComponentOffsetConverter : IMultiValueConverter
{
	public object Convert(object[] values, Type targetType, object parameter,
				CultureInfo culture)
	{
		var src = values?[0] as double? ?? 0;
		var dest = values?[1] as double? ?? 0;
		var offset = parameter as double? ?? 0;
		return dest - src + offset;
	}
```

The LinkOffsetConverter is used to calculate the Y coordinate of where the line starts, which is the middle point of the height of the component. It returns half of the height of the component, plus a predefined offset (which is -1, to account for zero-based coordinates):

```
public class LinkOffsetConverter : IValueConverter
{
	public object Convert(object value, Type targetType, object parameter,
			CultureInfo culture)
	{
		return (value as double?) / 2.0 + (parameter as double?) ?? 0;
	}

	public object ConvertBack(object value, Type targetType, object parameter,
			CultureInfo culture)
	{
		return (value as double?) * 2.0 - (parameter as double?) ?? 0;
	}
}
```

A style based on LinkStyle (shown earlier) is applied to the line, which also hides it when the component is not linked to a node:

```
<Line.Style>
	<Style TargetType="Line" BasedOn="{StaticResource LinkStyle}">
		<Style.Triggers>
			<DataTrigger Binding="{Binding Node1}" Value="{x:Null}">
				<Setter Property="Visibility" Value="Collapsed" />
			</DataTrigger>
		</Style.Triggers>
	</Style>
</Line.Style>
```

The second line code is almost the same as the first, except for the X coordinate at which to start drawing the line, which must be set to the width of the component in order to get the right side of it:

```
<Line x:Name="Line2" X1="{Binding Content.Width, ElementName=ComponentIcon}"
	Y1="{Binding Content.Height, ElementName=ComponentIcon,
	Converter={StaticResource LinkOffsetConv}, ConverterParameter={StaticResource
	LinkOffsetH}}">
	<Line.X2>
		<MultiBinding
			Converter="{StaticResource ComponentOffsetConverter}"
			ConverterParameter="{StaticResource NodeOffset}" Mode="OneWay">
			<Binding Path="X" />
			<Binding Path="Node2.X" />
		</MultiBinding>
	</Line.X2>
	<Line.Y2>
		<MultiBinding
			Converter="{StaticResource ComponentOffsetConverter}"
			ConverterParameter="{StaticResource NodeOffset}" Mode="OneWay">
```

```
                <Binding Path="Y" />
                <Binding Path="Node2.Y" />
            </MultiBinding>
        </Line.Y2>
        <Line.Style>
            <Style TargetType="Line" BasedOn="{StaticResource LinkStyle}">
                <Style.Triggers>
                    <DataTrigger Binding="{Binding Node2}" Value="{x:Null}">
                        <Setter Property="Visibility" Value="Collapsed" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </Line.Style>
</Line>
```

This trigger updates the orientation of the component, which by default is horizontal and the above code assumes that. When the orientation is changed to vertical, the component layout is rotated by 90 degrees, and the starting points of the lines have the coordinates swapped: (Note that the predefined offset for the LinkOffsetConverter is now 1)

```
<DataTrigger Binding="{Binding Direction}" Value="Vertical">
        <Setter Property="LayoutTransform" TargetName="Host">
            <Setter.Value>
                <RotateTransform Angle="90" />
            </Setter.Value>
        </Setter>
        <Setter Property="X1" TargetName="Line1"
                Value="{Binding Content.Height, ElementName=ComponentIcon,
                        Converter={StaticResource LinkOffsetConv},
                        ConverterParameter={StaticResource LinkOffsetV}}" />
        <Setter Property="Y1" TargetName="Line1" Value="0" />
        <Setter Property="X1" TargetName="Line2"
            Value="{Binding Content.Height, ElementName=ComponentIcon,
                    Converter={StaticResource LinkOffsetConv},
                    ConverterParameter={StaticResource LinkOffsetV}}" />
        <Setter Property="Y1" TargetName="Line2" Value="{Binding Content.Width,
                                                        ElementName=ComponentIcon}" />
</DataTrigger>
```

The following style (AlignFix) sets the alignment properties of a TextBlock. It is used for the tags of each element in the circuit, and the minimum width is set to the full width of the element in order to center the text correctly:

```
<Style x:Key="AlignFix" TargetType="TextBlock">
        <Setter Property="TextAlignment" Value="Center" />
        <Setter Property="HorizontalAlignment" Value="Left" />
        <Setter Property="MinWidth"
            Value="{Binding MinWidth, RelativeSource={RelativeSource FindAncestor,
        AncestorType={x:Type ContentControl}}}" />
</Style>
```

The node template

The name of the node is shown above it:

```
<TextBlock Canvas.Bottom="26" MinWidth="24" TextAlignment="Center" Text="{Binding Name}" />
```

The node itself is an ellipse. The stroke color is black, but when it is selected it is set to blue:

```
<Ellipse Height="24" Width="24" StrokeThickness="4" Fill="Transparent" x:Name="Ellipse">
        <Ellipse.Style>
            <Style TargetType="Ellipse">
                <Setter Property="Stroke" Value="Black" />
```

```
                <Style.Triggers>
                    <DataTrigger Binding="{Binding IsSelected,
                RelativeSource={RelativeSource FindAncestor,
            AncestorType={x:Type s:DesignerItem}}}" Value="True">
                        <Setter Property="Stroke" Value="CornflowerBlue" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </Ellipse.Style>
</Ellipse>
```

If the node is a reference node, the nodes appearance is completely changed. It is drawn with vector graphics with a few transforms:

```
<Path x:Name="Icon"
        StrokeThickness="1"
        Stroke="Black"
        StrokeLineJoin="Miter"
        StrokeStartLineCap="Flat"
        StrokeEndLineCap="Flat"
        Data="{StaticResource RefNodeIc}">
        <Path.RenderTransform>
            <TransformGroup>
                <TranslateTransform X="-4.43708" Y="-15.242819" />
                <MatrixTransform Matrix="-1 0 0 1 376 -342.3622" />
                <ScaleTransform ScaleX="2" ScaleY="2.823529" />
                <TranslateTransform Y="12" />
            </TransformGroup>
        </Path.RenderTransform>
</Path>
```

Once again, the color changes to blue when it is selected:

```
<ControlTemplate.Triggers>
        <DataTrigger Binding="{Binding IsSelected, RelativeSource={RelativeSource
            FindAncestor, AncestorType={x:Type s:DesignerItem}}}"
        Value="True">
            <Setter TargetName="Icon" Property="Stroke" Value="CornflowerBlue" />
        </DataTrigger>
</ControlTemplate.Triggers>
```

Below the node the voltage is displayed. If the voltage is zero, or the node is a reference node, it will be hidden:

```
<TextBlock Canvas.Top="26" MinWidth="24" TextAlignment="Center">
        <TextBlock.Style>
            <Style TargetType="TextBlock">
                <Style.Triggers>
                    <DataTrigger Binding="{Binding IsReferenceNode}" Value="True">
                        <Setter Property="Visibility" Value="Collapsed" />
                    </DataTrigger>
                    <DataTrigger Binding="{Binding Voltage}" Value="0">
                        <Setter Property="Visibility" Value="Collapsed" />
                    </DataTrigger>
                </Style.Triggers>
            </Style>
        </TextBlock.Style>
        <TextBlock Text="{Binding Voltage, Converter={StaticResource DoubleRoundConvertor}}"/>
        V
</TextBlock>
```

The DoubleRoundConverter rounds values to three digits after the decimal point:

```
public class DoubleRoundConvertor : IValueConverter
{
	public object Convert(object value, Type targetType, object parameter,
				CultureInfo culture)
	{
		var v = value as double? ?? 0.0;
		if (double.IsNaN(v)) v = 0.0;
		return v.ToString("0.###");
	}
}
```

The resistor template

The resistor icon is a vector graphic. Since we have already defined a common template for components, we can use it here and just put the icon in the content and the electrical properties in the Tag property.

The icon:

```
<DataTemplate DataType="{x:Type model:Resistor}">
	<ContentControl Template="{StaticResource CircuitComponentTemplate}">
		<Path Height="24" Width="48" StrokeThickness="0"
			Data="{StaticResource ResistorIc}"
			Style="{StaticResource SelectFillStyle}" />
		<ContentControl.Tag>
```

The electrical properties are centered and rounded, and will be hidden if they are zero:

```
<StackPanel>
	<TextBlock>
		<TextBlock.Style>
			<Style BasedOn="{StaticResource AlignFix}" TargetType="TextBlock">
				<Style.Triggers>
					<DataTrigger Binding="{Binding Resistance}" Value="0">
						<Setter Property="Visibility" Value="Collapsed" />
					</DataTrigger>
				</Style.Triggers>
			</Style>
		</TextBlock.Style>
		<TextBlock Text="{Binding Resistance,
				Converter={StaticResource DoubleRoundConvertor}}" />Ω
	</TextBlock>
```

The current and voltage are shown in exactly the same way.

The voltage source template

Much like the resistor, most of the work was already done with the general component template. The only differences are that the icon requires a couple of transforms, and no resistance is displayed (only voltage and current).

The new icon:

```
<DataTemplate DataType="{x:Type model:VoltageSource}">
	<ContentControl Template="{StaticResource CircuitComponentTemplate}">
		<Canvas Background="Transparent" Width="48" Height="48">
			<Path Data="{StaticResource VoltageSourceIc}" Canvas.Top="-1"
				Style="{StaticResource SelectFillStyle}">
				<Path.RenderTransform>
					<TransformGroup>
						<ScaleTransform ScaleX="-0.104348"
								ScaleY="0.104348" />
						<TranslateTransform X="48" />
					</TransformGroup>
				</Path.RenderTransform>
```

```
                </Path>
            </Canvas>
```

The switch template

The switch is a little more complicated. It is drawn as two ellipses and a line. The line's end point is updated when the switch is opened or closed, using an animation.

The line is drawn as follows:

```
<DataTemplate DataType="{x:Type model:Switch}">
    <ContentControl Template="{StaticResource CircuitComponentTemplate}">
        <Canvas Background="Transparent" Width="48" Height="24"
                ClipToBounds="False">
            <Ellipse Height="12" Width="12" Canvas.Top="5" Canvas.Left="-5"
                StrokeThickness="0"
                x:Name="LEllipse" Style="{StaticResource SelectFillStyle}" />
            <Ellipse Height="12" Width="12" Canvas.Top="5" Canvas.Left="42"
                StrokeThickness="0"
                x:Name="REllipse" Style="{StaticResource SelectFillStyle}" />
            <Line X1="0" Y1="11">
                <Line.Style>
```

The default end point of the line is set, and the color is changed to blue when the component is selected:

```
<Style TargetType="Line" BasedOn="{StaticResource LinkStyle}">
    <Setter Property="X2" Value="40" />
    <Setter Property="Y2" Value="-12" />
    <Style.Triggers>
        <DataTrigger Binding="{Binding IsSelected,
                                RelativeSource={RelativeSource FindAncestor,
                                        AncestorType={x:Type s:DesignerItem}}}"
                Value="True">
            <Setter Property="Stroke" Value="CornflowerBlue" />
        </DataTrigger>
```

When the switch is opened or closed, both coordinates of the line's end are animated simultaneously, for a duration of 0.5 seconds:

```
<DataTrigger Binding="{Binding IsClosed}" Value="True">
    <DataTrigger.EnterActions>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetProperty="Y2" From="-12"
                                 To="11" Duration="0:0:0.5" />
                <DoubleAnimation Storyboard.TargetProperty="X2" From="40"
                                 To="47" Duration="0:0:0.5" />
            </Storyboard>
        </BeginStoryboard>
    </DataTrigger.EnterActions>
    <DataTrigger.ExitActions>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetProperty="Y2" From="11"
                                 To="-12" Duration="0:0:0.5" />
                <DoubleAnimation Storyboard.TargetProperty="X2" From="47"
                                 To="40" Duration="0:0:0.5" />
            </Storyboard>
        </BeginStoryboard>
    </DataTrigger.ExitActions>
</DataTrigger>
```

The Main Window: View and ViewModel

Since our UI is designed around the MVVM pattern, we will explain both the View and the ViewModel in parallel, while grouping together relevant parts of them.

The main window is divided into three parts:

- The designer canvas where the user creates the circuit
- The toolbox on the left side, where all the tools needed to build a circuit ("cursor modes") as well as functions such as load and save can be found
- The properties panel on the right side, where the properties of a selected circuit element can be edited

The ViewModel initializes a few values on startup:

```
public DesignerViewModel()
{
	_ser = new Serialization();
	DiagramObjects = new ObservableCollection<DiagramObject>();
	CursorMode = CursorMode.ArrangeItems;
	AreaHeight = 1000;
	AreaWidth = 1000;
}
```

The designer canvas

The designer is placed inside a border, which provides a background. The background is made up of 50x50 tiled rectangles, separated by dark gray dashed lines:

```
<Border.Background>
	<VisualBrush TileMode="Tile" Viewport="0,0,50,50" ViewportUnits="Absolute"
		     Viewbox="0,0,50,50" ViewboxUnits="Absolute">
		<VisualBrush.Visual>
			<Rectangle Stroke="DarkGray" StrokeThickness="1" Height="50"
				   Width="50" StrokeDashArray="5 3" />
		</VisualBrush.Visual>
	</VisualBrush>
</Border.Background>
```

Below is the creation of the DesignerCanvas. The background is almost transparent so it will register mouse clicks. The MouseLeftButtonUp and SelectedItemChanged events are bound to the ViewModel with the help of the System.Windows.Interactivity and MvvmLight libraries. The element templates are also loaded into a resource dictionary:

```
<layout:DesignerCanvas x:Name="EditBox" Background="#01FFFFFF"
		       ItemsSource="{Binding DiagramObjects}"
		       Height="{Binding AreaHeight}" Width="{Binding AreaWidth}"
		       VerticalAlignment="Top" HorizontalAlignment="Left"
		       SelectedItem="{Binding SelectedObject, Mode=TwoWay}"
		       Focusable="True">
	<i:Interaction.Triggers>
		<i:EventTrigger EventName="MouseLeftButtonUp">
			<command:EventToCommand PassEventArgsToCommand="True"
			 Command="{Binding ClickCommand}"
			 EventArgsConverter="{StaticResource ClickEventToPointConv}"
			 EventArgsConverterParameter="{Binding ElementName=EditBox}" />
		</i:EventTrigger>
		<i:EventTrigger EventName="SelectedItemChanged">
			<command:EventToCommand PassEventArgsToCommand="True"
				Command="{Binding SelectedItemChangedCommand}" />
		</i:EventTrigger>
```

```
            </i:Interaction.Triggers>
            <layout:DesignerCanvas.Resources>
                  <ResourceDictionary Source="ElementTemplates.xaml" />
            </layout:DesignerCanvas.Resources>
</layout:DesignerCanvas>
```

Now we will follow the aforementioned events to their handlers in the ViewModel. First, for the MouseLeftButtonUp event, the EventArgs are used by a converter to find the coordinates of the click, relative to the DesignerCanvas:

```
public class ClickEventToPointConverter : IEventArgsConverter
{
      public object Convert(object value, object parameter)
      {
            return (value as MouseButtonEventArgs)?.GetPosition(parameter as IInputElement);
      }
}
```

The MouseLeftButtonUp event is handled by the ClickCommand, which calls the OnClick function. This function behaves differently depending on what cursor mode the user has selected, but only adding an element to the circuit and generating a name for it are handled here:

```
public ICommand ClickCommand => new RelayCommand<Point>(OnClick);

private void OnClick(Point e)
{
      switch (CursorMode)
      {
            case CursorMode.AddResistor:
                  DiagramObjects.Add(new Resistor { Name = @"R" + ++_nextResistorId,
                                                    X = e.X, Y = e.Y });
            break;
            case CursorMode.AddVoltageSource:
                  DiagramObjects.Add(new VoltageSource { Name = @"Vin" + ++_nextVSourceId,
                                                         X = e.X, Y = e.Y });
            break;
            case CursorMode.AddNode:
                  DiagramObjects.Add(new Node { Name = @"N" + ++_nextNodeId,
                                                X = e.X, Y = e.Y });
            break;
            case CursorMode.AddRefNode:
                  DiagramObjects.Add(new Node
                  {
                        Name = @"Nref" + ++_nextRefNodeId,
                        X = e.X,
                        Y = e.Y,
                        IsReferenceNode = true
                  });
            break;
            case CursorMode.AddSwitch:
                  DiagramObjects.Add(new Switch
                  {
                        Name = @"S" + ++_nextSwitchId,
                        X = e.X,
                        Y = e.Y
                  });
            break;
      }
}
```

The SelectedItemChanged event is similarly handled by a function with the same name. This function connects nodes and components, therefore if the cursor mode is not the connection mode the function will return immediately. Once the connection has been made successfully, the selected item is cleared.

The function tries to get the information of the node and the component that were last clicked on. Since the order in which they were clicked on doesn't matter, the function assumes a specific order and tries again if either value is null. If one of the values is still equal to null, the function will not proceed.

Next the function checks which side of the component the user clicked on. If the user has clicked on the left half then the node will be connected to the left side of the component (as Node1) and if the user has clicked on the right half then the node will be connected to the right side of the component (as Node2).

In order to correctly and reliably determine where the user has clicked relative to the component, two important considerations were made. First, the user may have clicked on the component before the node, so the relative position at which the user clicked on a component must be saved to a local variable. Second, the component may have been rotated to a vertical position, so the comparison will be made on the Y axis instead of the X axis in this case.

The function code:

```
public ICommand SelectedItemChangedCommand =>
            new RelayCommand<SelectedItemChangedEventArgs>(SelectedItemChanged);

private void SelectedItemChanged(SelectedItemChangedEventArgs obj)
{
       if (CursorMode != CursorMode.ConnectToNode) return;
       var n = obj.NewValue as Node;
       var c = obj.OldValue as Component;
       if (n == null && c == null)
       {
              n = obj.OldValue as Node;
              c = obj.NewValue as Component;
              // If we just selected the component, save the mouse coordinates
              if (c != null)
              {
                     var di = (Application.Current.MainWindow as MainWindow).EditBox
                                   .Children.OfType<DesignerItem>()
                                   .FirstOrDefault(d => Equals(d.DataContext, c));
                     if (di == null) return;
                     _componentClickPos = Mouse.GetPosition(di);
              }
       }
       // By now n and c should contain correct objects
       if (n == null || c == null) return;
       // Use X value if component is horizontal, Y if it is vertical
       var compPos = c.Direction == Direction.Horizontal
                                  ? _componentClickPos.X : _componentClickPos.Y;
       if (compPos > 26 && compPos < 48) c.Node2 = n;
       else if (compPos >= 0 && compPos < 22) c.Node1 = n;
       SelectedObject = null;
}
```

The window also has a single key binding for deleting an element in the circuit using the Delete key. It is handled similarly to the events above:

```
<Window.InputBindings>
       <KeyBinding Key="Delete" Command="{Binding DeleteCommand}" />
</Window.InputBindings>

public ICommand DeleteCommand => new RelayCommand(Delete);
```

```
private void Delete()
{
	if (SelectedObject != null) DiagramObjects.Remove(SelectedObject);
}
```

The toolbox area

The toolbox itself is a ListBox. The selected item is bound to the cursor mode in the ViewModel, and the ListBox is populated from the values of the CursorMode enum. The highlight colors are manually overridden:

```
<ListBox SelectedItem="{Binding CursorMode}"
         ItemsSource="{Binding Source={StaticResource CursorModes}}">
	<ListBox.ItemContainerStyle>
		<Style TargetType="ListBoxItem">
			<Style.Resources>
				<SolidColorBrush x:Key="{x:StaticSystemColors.HighlightBrushKey}"
			Color="CornflowerBlue" />
				 <SolidColorBrush x:Key="{x:Static
				SystemColors.InactiveSelectionHighlightBrushKey}"
				Color="CornflowerBlue" />
			</Style.Resources>
			<Setter Property="Margin" Value="2" />
			<Style.Triggers>
				<Trigger Property="IsMouseOver" Value="True">
					<Setter Property="Background" Value="PowderBlue" />
				</Trigger>
			</Style.Triggers>
		</Style>
	</ListBox.ItemContainerStyle>
</ListBox>
```

The template for a cursor mode contains an icon vector graphic, which is different depending on which mode it represents:

```
<Style TargetType="Path">
	<Setter Property="Data" Value="{StaticResource AddIc}" />
	<Style.Triggers>
		<DataTrigger Binding="{Binding}" Value="{x:Static vm:CursorMode.ArrangeItems}">
			<Setter Property="Data" Value="{StaticResource ArrangeIc}" />
		</DataTrigger>
		<DataTrigger Binding="{Binding}" Value="{x:Static
vm:CursorMode.ConnectToNode}">
			<Setter Property="Data" Value="{StaticResource ConnectIc}" />
		</DataTrigger>
	</Style.Triggers>
</Style>
```

It also contains a short description:

```
<TextBlock Margin="10,0,0,0" VerticalAlignment="Center"
           Text="{Binding Converter={StaticResource EnumDescriptionConv}}" />
```

The EnumDescriptionConverter gets the values of an enum and checks if they have the Description attribute, if it is present the value of the attribute is returned:

```
public class EnumDescriptionConverter : IValueConverter
{
	public object Convert(object value, Type targetType, object parameter,
			CultureInfo culture)
	{
		var myEnum = value as Enum;
```

```
            var description = GetEnumDescription(myEnum);
            return description;
        }

        private static string GetEnumDescription(Enum enumObj)
        {
            if (enumObj == null) return null;

            var fieldInfo = enumObj.GetType().GetField(enumObj.ToString());
            var attribArray = fieldInfo?.GetCustomAttributes(false);

            if (attribArray == null || attribArray.Length == 0) return
            enumObj.ToString();

            var attrib = attribArray[0] as DescriptionAttribute;
            return attrib?.Description ?? enumObj.ToString();
        }
}
```

Populating the ListBox with all the possible cursor modes is performed by an ObjectDataProvider, which is used to provide WPF with the return value of a CLR function, in this case the equivalent function call in C# would be System.Enum.GetValues(typeof(CursorMode)):

```
<ObjectDataProvider x:Key="CursorModes" ObjectType="{x:Type system:Enum}"
                    MethodName="GetValues">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="vm:CursorMode" />
        </ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
```

The rest of the functionality on this side of the window is arranged like a menu. The first buttons are for loading and saving circuits (as XML):

```
<MenuItem Header="Load circuit" HorizontalAlignment="Stretch">
        <i:Interaction.Triggers>
            <i:EventTrigger EventName="Click">
                <command:EventToCommand Command="{Binding LoadCommand}" />
            </i:EventTrigger>
        </i:Interaction.Triggers>
</MenuItem>
<MenuItem Header="Save circuit as..." HorizontalAlignment="Stretch">
        <i:Interaction.Triggers>
            <i:EventTrigger EventName="Click">
                <command:EventToCommand Command="{Binding SaveCommand}" />
            </i:EventTrigger>
        </i:Interaction.Triggers>
</MenuItem>
```

The click events are once again handled in the ViewModel. *The following two functions use all three of the methods defined in CircuitXmlUtils, which were described earlier.*

The function to load a circuit shows an OpenFileDialog, opens the selected file, deserializes it, clears the designer and displays the new circuit. It also attempts to determine the numbers of the last auto-generated element names so it can name the next added component accordingly. The cursor mode is also reset for convenience:

```
private void Load()
{
        if (_ofd == null) _ofd = new OpenFileDialog { Filter = "XML file|*.xml" };
        var dr = _ofd.ShowDialog(Application.Current.MainWindow);
        if (dr != true) return;
```

```
        // Load circuit XML
        CircuitXml cx;
        using (var fs = File.OpenRead(_ofd.FileName)) { cx = _ser.Deserialize(fs); }
        DiagramObjects.Clear();
        foreach (var dobj in cx.ToDiagram()) DiagramObjects.Add(dobj);

        // Update next automatic names
        _nextResistorId = DiagramObjects.OfType<Resistor>().MaxDefaultId("R");
        _nextVSourceId = DiagramObjects.OfType<VoltageSource>().MaxDefaultId("Vin");
        _nextNodeId = DiagramObjects.OfType<Node>()
                                    .Where(n => !n.IsReferenceNode).MaxDefaultId("N");
        _nextRefNodeId = DiagramObjects.OfType<Node>()
                                       .Where(n => n.IsReferenceNode).MaxDefaultId("Nref");

        CursorMode = CursorMode.ArrangeItems;
}
```

The function to save a circuit is similar but simpler. It shows a SaveFileDialog, creates the selected file, and serializes the circuit:

```
private void Save()
{
        if (_sfd == null) _sfd = new SaveFileDialog { Filter = "XML file|*.xml" };
        var dr = _sfd.ShowDialog(Application.Current.MainWindow);
        if (dr != true) return;

        var cx = CircuitXmlUtils.ToCircuitXml(DiagramObjects);
        using (var fs = File.Create(_sfd.FileName)) { _ser.Serialize(cx, fs); }
}
```

Next is the simulation options group box. Circuit states can be enabled with a checkbox and the states editor window can be opened. The max current can be adjusted here, which changes the color gradient for the lines in the circuit (see CurrentToBrushConverter above):

```
<GroupBox Header="Simulation options">
        <StackPanel>
              <CheckBox Content="Enable circuit states" IsChecked="{Binding AreStatesEnabled}"
                        Margin="0,2" />
                      <MenuItem Header="Edit states..." HorizontalAlignment="Stretch">
                            <i:Interaction.Triggers>
                                  <i:EventTrigger EventName="Click">
                                        <command:EventToCommand Command="{Binding
                                                                    StatesEditorCommand}" />
                                  </i:EventTrigger>
                            </i:Interaction.Triggers>
                      </MenuItem>
                      <Separator Margin="0,2" />
                      <StackPanel Orientation="Horizontal" Margin="0,2">
                            <TextBlock VerticalAlignment="Center" Text="Max current: " />
                            <xctk:DoubleUpDown VerticalAlignment="Center" Width="60"
                                               DefaultValue="1.000" Increment="0.001"
                                               Minimum="0" Value="{Binding MaxValue,
                                               Source={StaticResource CurrentToColorConv}}"/>
               </StackPanel>
        </StackPanel>
</GroupBox>
```

The following function in the ViewModel opens the circuit state editor:

```
private void OpenStatesEditor()
{
        var sev = ViewMaker.CreateStatesEditor(SimulationStates, Switches);
```

```
        sev.Owner = Application.Current.MainWindow;
        sev.ShowDialog();
}
```

And finally, there is a button to begin the simulation:

```
<MenuItem Header="Start simulation" HorizontalAlignment="Stretch" FontWeight="Bold">
        <i:Interaction.Triggers>
                <i:EventTrigger EventName="Click">
                        <command:EventToCommand Command="{Binding SimulateCommand}" />
                </i:EventTrigger>
        </i:Interaction.Triggers>
</MenuItem>
```

First, the simulation function checks if the circuit has at least 2 nodes in it, if there are less than 2 the user is notified that the simulation cannot continue. Next it checks if there are any reference nodes, if there aren't any, the user is asked if the function should automatically choose one for them. By default the function chooses the node on the negative side of the first voltage source.

Now the function creates a results window and creates a copy of the circuit, to avoid changing the data in the designer. The XML serializer is used to create an actual copy of the circuit, in order to avoid having to write a long and difficult to maintain copy function. If circuit states are enabled, a copy is created for each state. Finally, each circuit copy is sent to the UpdateSimulation function, and the results are added to the results window.

```
private void Simulate()
{
        if (DiagramObjects.Count(d => d is Node) < 2)
        {
                MessageBox.Show(Application.Current.MainWindow, "Not enough nodes in the
circuit! At least 2 nodes are required for simulation.", "Simulation precondition",
                                MessageBoxButton.OK, MessageBoxImage.Error);
                return;
        }
        if (DiagramObjects.All(d => (d as Node)?.IsReferenceNode != true))
        {
                var mbr = MessageBox.Show(Application.Current.MainWindow, "Missing a reference
node! Would you like an existing node to be automatically chosen as a reference node?",
                                          "Simulation warning", MessageBoxButton.YesNo,
                                          MessageBoxImage.Warning);
                if (mbr != MessageBoxResult.Yes) return;
                var vs = DiagramObjects.OfType<VoltageSource>()
                                       .FirstOrDefault(v => v.Node2 != null);
                if (vs != null) vs.Node2.IsReferenceNode = true;
        }

        var diags = new ObservableCollection<TabItem>();
        var r = ViewMaker.CreateResults(diags);
        r.Owner = Application.Current.MainWindow;
        var xml = _ser.Serialize(CircuitXmlUtils.ToCircuitXml(DiagramObjects));
        if (AreStatesEnabled)
        {
                foreach (var state in SimulationStates)
                {
                        var cir = _ser.Deserialize(xml).ToDiagram().ToList();
                        // Apply state
                        foreach (var switchState in state.SwitchStates)
                                cir.OfType<Switch>()
                                   .FirstOrDefault(sw => sw.Id == switchState.Key)
                                   .IsClosed = switchState.Value;
                        // Update simulation
                        var s = UpdateSimulation(cir);
```

```
                        if (s == null)
                                diags.Add(ViewMaker.CreateResultDiagramSnapshot(cir,state.Name));
                        else diags.Add(ViewMaker.CreateResultError(s));
                }
        }
        else
        {
                var cir = _ser.Deserialize(xml).ToDiagram().ToList();
                var s = UpdateSimulation(cir);
                if (s == null)
                        diags.Add(ViewMaker.CreateResultDiagramSnapshot(cir, "default"));
                else diags.Add(ViewMaker.CreateResultError(s));
        }
        r.ShowDialog();
}
```

The UpdateSimulation method simply calls AnalyzeAndUpdate on a given circuit and handles exceptions:

```
private string UpdateSimulation(List<DiagramObject> cir)
{
        try
        {
                Simulator.AnalyzeAndUpdate(cir.OfType<Node>(), cir.OfType<IComponent>());
                return null;
        }
        catch (Exception ex) { return "Simulation error: " + ex; }
}
```

The properties panel

The properties panel allows the user to view and edit all properties of elements in the circuit. The properties shown are specified manually:

```
<xctk:PropertyGrid Grid.Row="0" Grid.Column="2" AutoGenerateProperties="False"
                   SelectedObject="{Binding SelectedObject}"
                   Width="200" Margin="2">
        <xctk:PropertyGrid.PropertyDefinitions>
                <xctk:PropertyDefinition TargetProperties="Name" DisplayName="Name" />
                <xctk:PropertyDefinition TargetProperties="Id" DisplayName="ID" />
                <xctk:PropertyDefinition TargetProperties="Resistance"
                                         DisplayName="Resistance" />
                <xctk:PropertyDefinition TargetProperties="Voltage" DisplayName="Voltage" />
                <xctk:PropertyDefinition TargetProperties="Current" DisplayName="Current" />
                <xctk:PropertyDefinition TargetProperties="IsClosed" DisplayName="IsClosed" />
                <xctk:PropertyDefinition TargetProperties="Node1" DisplayName="Node1" />
                <xctk:PropertyDefinition TargetProperties="Node2" DisplayName="Node2" />
                <xctk:PropertyDefinition TargetProperties="Direction"
                                         DisplayName="Direction" />
                <xctk:PropertyDefinition TargetProperties="IsReferenceNode"
                                         DisplayName="IsReferenceNode" />
        </xctk:PropertyGrid.PropertyDefinitions>
```

When the value of a property is a node, a ComboBox is used to allow the user to select a node by name:

```
<xctk:EditorTemplateDefinition.EditingTemplate>
        <DataTemplate>
                <ComboBox ItemsSource="{Binding DataContext.Nodes, Source={x:Reference View}}"
                          SelectedItem="{Binding Value}" DisplayMemberPath="Name" />
        </DataTemplate>
</xctk:EditorTemplateDefinition.EditingTemplate>
```

The Results Window

The results window uses the same element templates as the designer is the main window to display the circuit. The simulation of each state is shown in a separate tab. A single tab is defined as follows:

```
<layout:DesignerCanvas ItemsSource="{Binding Diagram}">
	<layout:DesignerCanvas.Resources>
		<ResourceDictionary Source="ElementTemplates.xaml" />
	</layout:DesignerCanvas.Resources>
	<i:Interaction.Triggers>
		<i:EventTrigger EventName="Loaded">
			<command:EventToCommand Command="{Binding LoadedCommand}" />
		</i:EventTrigger>
	</i:Interaction.Triggers>
</layout:DesignerCanvas>
```

The ViewModel for a tab simply gets a circuit and a state name. The constructor:

```
public ResultDiagramSnapshotViewModel(IEnumerable<DiagramObject> cir, string name)
{
	Diagram = new ObservableCollection<DiagramObject>();
	Name = name;
	_circuit = cir;
}
```

When the tab is loaded the diagram is populated:

```
private void Loaded()
{
	if (_circuit != null) foreach (var o in _circuit) Diagram.Add(o);
	_circuit = null;
}
```

The tabs are displayed in the window like this:

```
<Grid>
	<Grid.RowDefinitions>
		<RowDefinition Height="*" />
		<RowDefinition Height="Auto" />
	</Grid.RowDefinitions>
	<TabControl ItemsSource="{Binding Tabs}" />
	<StackPanel Grid.Row="1" Orientation="Horizontal" Margin="2">
		<TextBlock VerticalAlignment="Center" Text="Max current: " />
		<xctk:DoubleUpDown VerticalAlignment="Center" Width="60" DefaultValue="1.000"
				Increment="0.001" Minimum="0"
				Value="{Binding MaxValue,
					Source={StaticResource CurrentToColorConv}}" />
	</StackPanel>
</Grid>
```

Note that the max current option is also available in this window.

The Circuit State Editor Window

In the upper row of this window, there are 2 buttons - Load and Save:

```
<Button Content="Load states" HorizontalAlignment="Left">
	<i:Interaction.Triggers>
		<i:EventTrigger EventName="Click">
			<command:EventToCommand Command="{Binding LoadCommand}" />
		</i:EventTrigger>
	</i:Interaction.Triggers>
</Button>
<Button Grid.Row="0" Content="Save states as..." HorizontalAlignment="Right">
	<i:Interaction.Triggers>
		<i:EventTrigger EventName="Click">
			<command:EventToCommand Command="{Binding SaveCommand}" />
		</i:EventTrigger>
	</i:Interaction.Triggers>
</Button>
```

The Load and Save functions deserialize and serialize states from CSV files (see CircuitState description):

```
private void Load()
{
	if (_ofd == null) _ofd = new OpenFileDialog { Filter = "CSV file|*.csv" };
	var dr = _ofd.ShowDialog(Application.Current.Windows.OfType<StatesEditorWindow>()
                                                        .FirstOrDefault());
	if (dr != true) return;
	States.Clear();
	foreach (var cs in CircuitState.Deserialize(File.ReadAllText(_ofd.FileName)))
		States.Add(cs);
}

private void Save()
{
	if (_sfd == null) _sfd = new SaveFileDialog { Filter = "CSV file|*.csv" };
	var dr = _sfd.ShowDialog(Application.Current.Windows.OfType<StatesEditorWindow>()
                                                        .FirstOrDefault());
	if (dr != true) return;
	File.WriteAllText(_sfd.FileName, CircuitState.Serialize(States));
}
```

The states themselves can be added or edited in a DataGrid:

```
<DataGrid Grid.Row="1" ItemsSource="{Binding States}" AutoGenerateColumns="False"
          CanUserAddRows="True" CanUserDeleteRows="True"
          behaviors:DataGridColumnsBehavior.BindableColumns="{Binding Columns}">
	<i:Interaction.Triggers>
		<i:EventTrigger EventName="InitializingNewItem">
			<command:EventToCommand Command="{Binding InitNewItemCommand}"
                                        PassEventArgsToCommand="True" />
		</i:EventTrigger>
	</i:Interaction.Triggers>
</DataGrid>
```

The ViewModel is initialized with the collection of states and the list of switches. It creates a column in the DataGrid for each switch:

```
public StatesEditorViewModel(ObservableCollection<CircuitState> states,
                             IEnumerable<Switch> switches)
{
	_switches = switches.ToList();
	States = states;
	Columns = new ObservableCollection<DataGridColumn>
```

```
        {
                new DataGridTextColumn
                {
                        Header = "[State Name]",
                        Binding = new Binding("Name")
                }
        };
        foreach (var sw in _switches)
        Columns.Add(new DataGridCheckBoxColumn
        {
                Header = sw.Name,
                Binding = new Binding($"SwitchStates[{sw.Id}]")
        });
}
```

Every time when a new state is added to the DataGrid the function InitNewItem is called, and the new state is initialized manually:

```
private void InitNewItem(InitializingNewItemEventArgs obj)
{
        _switches.ForEach(sw => ((CircuitState)obj.NewItem).SwitchStates.Add(sw.Id, false));
}
```

In order to enable adding columns to a DataGrid dynamically, the DataGridColumnsBehavior is used. After registering the DependencyProperties, the function BindableColumnsPropertyChanged handles adding and removing columns based on the following changes to the observable collection of columns:

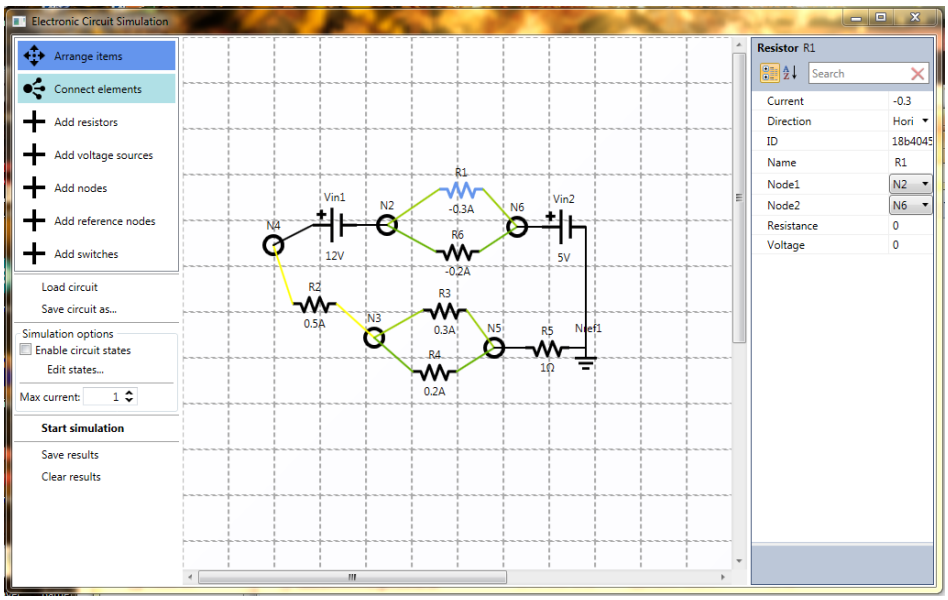
- If it is reset then the function will clear all the columns and add the new ones
- If an item was added then the function will add the new column
- If an item was moved then the function will update the order of the columns accordingly
- If an item was replaced then the function will replace the column

```
columns.CollectionChanged += (sender, e2) =>
{
        var ne = e2;
        switch (ne.Action)
        {
                case NotifyCollectionChangedAction.Reset:
                        dataGrid.Columns.Clear();
                        foreach (DataGridColumn column in ne.NewItems)
                                dataGrid.Columns.Add(column);
                        break;
                case NotifyCollectionChangedAction.Add:
                        foreach (DataGridColumn column in ne.NewItems)
                                dataGrid.Columns.Add(column);
                        break;
                case NotifyCollectionChangedAction.Move:
                        dataGrid.Columns.Move(ne.OldStartingIndex, ne.NewStartingIndex);
                        break;
                case NotifyCollectionChangedAction.Remove:
                        foreach (DataGridColumn column in ne.OldItems)
                                dataGrid.Columns.Remove(column);
                        break;
                case NotifyCollectionChangedAction.Replace:
                        dataGrid.Columns[ne.NewStartingIndex]= ne.NewItems[0] as DataGridColumn;
                        break;
        }
};
```

Finally, at the bottom of the window there is a close button. It uses the IsCancel property to close the window easily:

```
<Button Grid.Row="2" Content="Close" HorizontalAlignment="Center" IsCancel="True" />
```

User Guide



On the left side of the window there is a toolbox in which you can choose all the components which are needed to create a circuit.

The "Arrange items" is the option to just select and move around the objects in the designer.

The "Connect elements" option enables connecting between a node and a component. Click on one of them, then click on the other to connect them. The side of the component that will be connected depends on which side of the component you clicked.

All the other options are for adding components to the circuit as you like.

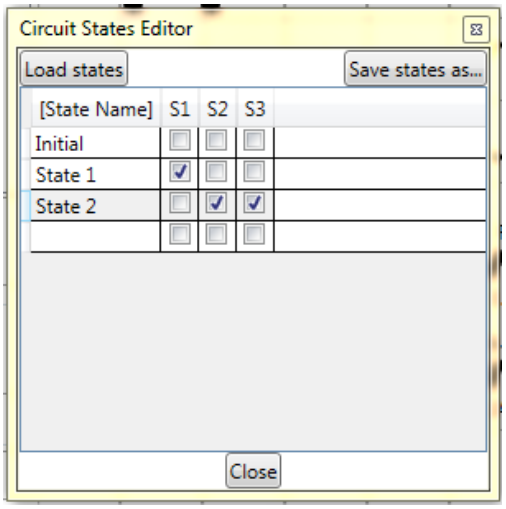
Below the toolbox there are the options to load and save a circuit as XML files.

The max current option is the value of the current which will be displayed in red in the simulation, the default is 1 ampere. Changing this value will change the colors used to represent different currents.

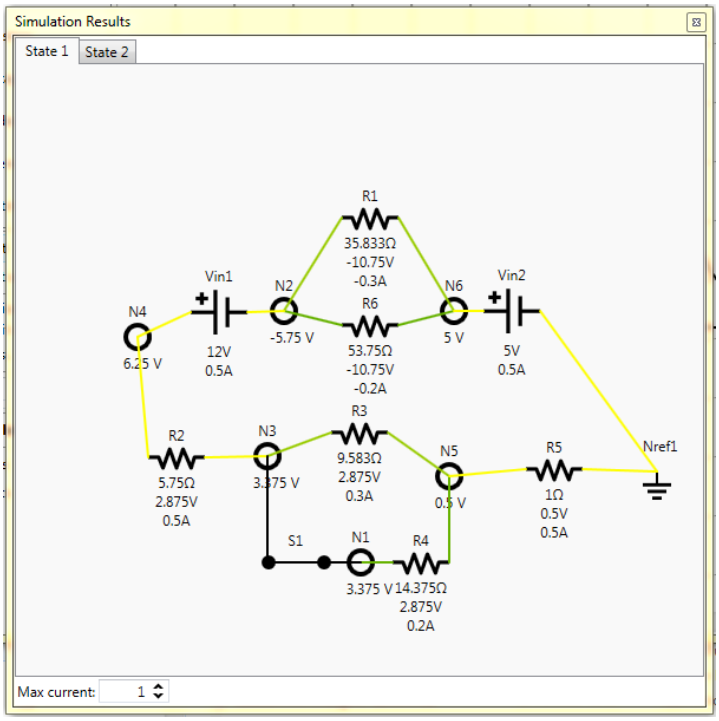
On the right side of the window are the properties and the values of the component you have selected (which is highlighted in blue). You can edit all the important values there.

Warning: Remember that the current direction is very significant, and the default direction is from left to right or from top to bottom. In other words, in order to define a current in the opposite direction (right to left or bottom to top) for a resistor you need to enter a negative value. If you have mistakenly entered the wrong value the program will display an error in the simulation that invalid values were detected.

In the simulation options group box there is the option to enable circuit states. The "Edit states" button open a little window in which you can create states for the circuit. The first column is the state name, the other columns are the switches and their state – open or closed. Enter the information for a new state in the blank row and press Enter when done, so a new blank row appears. There are save and load buttons here as well, for exporting/importing the states as CSV format data.



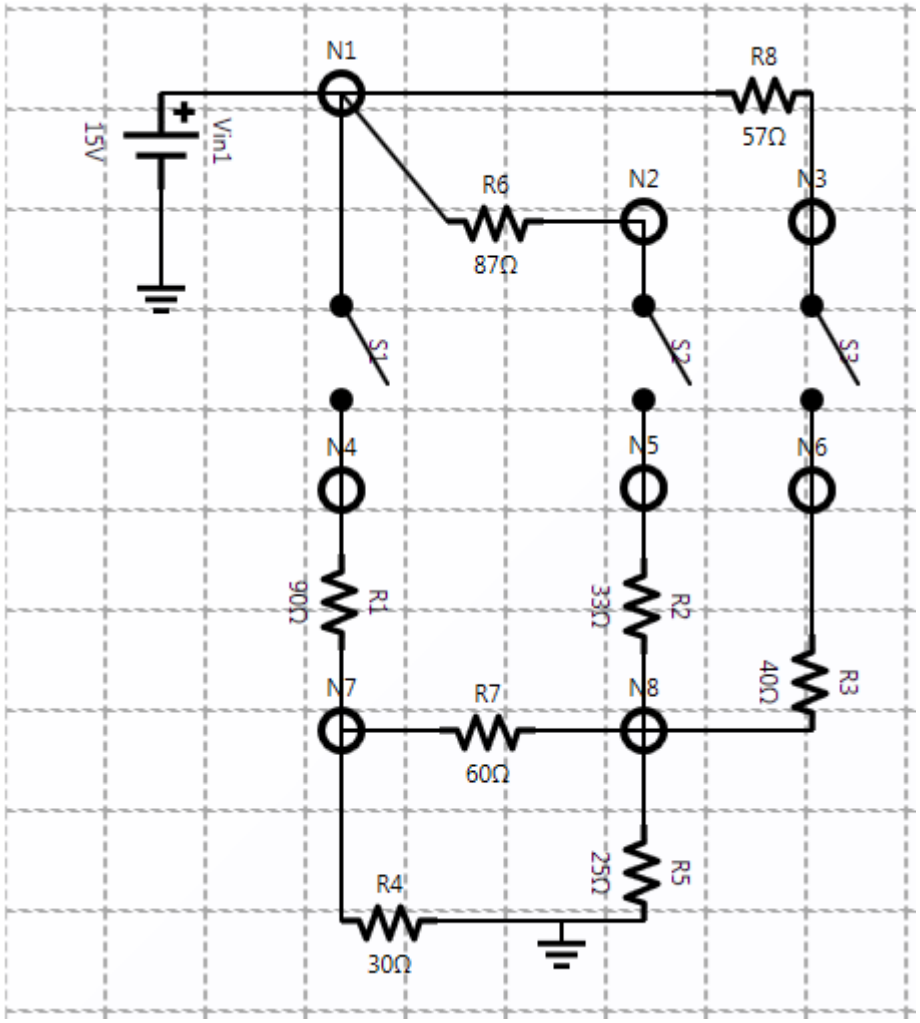
The simulation results are displayed in tabs, one for each state. If an error has occurred it will be displayed instead of the circuit in the tab. The components in the resulting circuit diagram can be moved, but these changes will not apply to the original circuit.



Simulation Example

Note: All MNA values are rounded up to 4 decimal places, and the results are displayed with Max Current = 0.3.

Consider the following circuit:



XML representation:

```
<?xml version="1.0"?>
<CircuitXml xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Nodes>
    <Node X="65.90333333333308" Y="166" Id="beeab9af-766e-4caf-997c-31ab767f7b5b" Name=""
Voltage="0" IsReferenceNode="true" />
    <Node X="155.90333333333331" Y="80" Id="43f59906-86f3-4372-a5c5-c282818d4115" Name="N1"
Voltage="0" IsReferenceNode="false" />
    <Node X="306.90333333333331" Y="144" Id="f3adbbe3-ef22-4d35-bbbb-31a0269604df" Name="N2"
Voltage="0" IsReferenceNode="false" />
    <Node X="390.90333333333331" Y="144" Id="e1a1c731-3c0d-4664-943a-8d24c45ac0c4" Name="N3"
Voltage="0" IsReferenceNode="false" />
    <Node X="155.90333333333331" Y="278" Id="bd2b6247-aa01-4b52-8723-61f1c822c664" Name="N4"
Voltage="0" IsReferenceNode="false" />
    <Node X="306.90333333333331" Y="277" Id="4f81f151-c4f8-4bdf-ba7e-7c62287cd59b" Name="N5"
Voltage="0" IsReferenceNode="false" />
    <Node X="390.90333333333331" Y="278" Id="c4fc78c1-d17a-4213-80e7-5172bba4189b" Name="N6"
Voltage="0" IsReferenceNode="false" />
```

```
    <Node X="155.90333333333331" Y="398" Id="8c189e1d-653f-4a79-926d-cf15e5d4ea3b" Name="N7"
Voltage="0" IsReferenceNode="false" />
    <Node X="306.90333333333331" Y="398" Id="c48b4bcb-a466-4338-9e7c-96498a0c78be" Name="N8"
Voltage="0" IsReferenceNode="false" />
    <Node X="265.90333333333331" Y="493" Id="c90e05be-15ec-4e61-908d-207f97d88edd" Name=""
Voltage="0" IsReferenceNode="true" />
  </Nodes>
  <Resistors>
    <Resistor X="154.90333333333331" Y="322" Id="b3f06812-4096-4b9f-8d96-b0eb7099da04"
Name="R1" Direction="Vertical" Resistance="90" Voltage="0" Current="0">
      <Node1Id>bd2b6247-aa01-4b52-8723-61f1c822c664</Node1Id>
      <Node2Id>8c189e1d-653f-4a79-926d-cf15e5d4ea3b</Node2Id>
    </Resistor>
    <Resistor X="305.90333333333331" Y="324" Id="f273635d-7c4b-4542-b21b-7ce90fea3924"
Name="R2" Direction="Vertical" Resistance="33" Voltage="0" Current="0">
      <Node1Id>4f81f151-c4f8-4bdf-ba7e-7c62287cd59b</Node1Id>
      <Node2Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node2Id>
    </Resistor>
    <Resistor X="389.90333333333331" Y="362" Id="9c2992fb-5460-459f-9640-92c063327f52"
Name="R3" Direction="Vertical" Resistance="40" Voltage="0" Current="0">
      <Node1Id>c4fc78c1-d17a-4213-80e7-5172bba4189b</Node1Id>
      <Node2Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node2Id>
    </Resistor>
    <Resistor X="167.90333333333331" Y="494" Id="632301ab-bfee-4221-9e7b-cb7ed6214e43"
Name="R4" Direction="Horizontal" Resistance="30" Voltage="0" Current="0">
      <Node1Id>8c189e1d-653f-4a79-926d-cf15e5d4ea3b</Node1Id>
      <Node2Id>c90e05be-15ec-4e61-908d-207f97d88edd</Node2Id>
    </Resistor>
    <Resistor X="305.90333333333331" Y="457" Id="c3073369-4fb5-446a-9121-5af5c87e24e8"
Name="R5" Direction="Vertical" Resistance="25" Voltage="0" Current="0">
      <Node1Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node1Id>
      <Node2Id>c90e05be-15ec-4e61-908d-207f97d88edd</Node2Id>
    </Resistor>
    <Resistor X="220.90333333333331" Y="145" Id="7c343178-87ef-4606-abfe-80e14aaba8b0"
Name="R6" Direction="Horizontal" Resistance="87" Voltage="0" Current="0">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>f3adbbe3-ef22-4d35-bbbb-31a0269604df</Node2Id>
    </Resistor>
    <Resistor X="216.90333333333331" Y="399" Id="c55837f9-1036-4065-a628-16510fb1b29e"
Name="R7" Direction="Horizontal" Resistance="60" Voltage="0" Current="0">
      <Node1Id>8c189e1d-653f-4a79-926d-cf15e5d4ea3b</Node1Id>
      <Node2Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node2Id>
    </Resistor>
    <Resistor X="354.90333333333331" Y="81" Id="45d13a44-40ea-4f10-b731-08c411faf0f1"
Name="R8" Direction="Horizontal" Resistance="57" Voltage="0" Current="0">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>e1a1c731-3c0d-4664-943a-8d24c45ac0c4</Node2Id>
    </Resistor>
  </Resistors>
  <VoltageSources>
    <VoltageSource X="52.90333333333308" Y="92" Id="f10fd17d-912d-438a-bd73-1fd59ed94636"
Name="Vin1" Direction="Vertical" Voltage="15" Current="0">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>beeab9af-766e-4caf-997c-31ab767f7b5b</Node2Id>
    </VoltageSource>
  </VoltageSources>
  <Switches>
    <Switch X="154.90333333333331" Y="197" Id="36bc6496-5ebe-47e6-9cfc-56bc5bd0e970"
Name="S1" Direction="Vertical" IsClosed="false">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>bd2b6247-aa01-4b52-8723-61f1c822c664</Node2Id>
    </Switch>
```

```
    <Switch X="305.90333333333331" Y="197" Id="99824b8f-fba5-4096-b2fb-1b4d6e1d6c8e"
Name="S2" Direction="Vertical" IsClosed="false">
      <Node1Id>f3adbbe3-ef22-4d35-bbbb-31a0269604df</Node1Id>
      <Node2Id>4f81f151-c4f8-4bdf-ba7e-7c62287cd59b</Node2Id>
    </Switch>
    <Switch X="389.90333333333331" Y="197" Id="cc6fc249-8375-4794-bef3-70ec6e6e5b3b"
Name="S3" Direction="Vertical" IsClosed="false">
      <Node1Id>e1a1c731-3c0d-4664-943a-8d24c45ac0c4</Node1Id>
      <Node2Id>c4fc78c1-d17a-4213-80e7-5172bba4189b</Node2Id>
    </Switch>
  </Switches>
</CircuitXml>
```

And the following states:

[State Name]	S1	S2	S3
First	☑	☐	☑
Second	☑	☑	☐

CSV representation:

```
[name],36bc6496-5ebe-47e6-9cfc-56bc5bd0e970,99824b8f-fba5-4096-b2fb-1b4d6e1d6c8e,cc6fc249-
8375-4794-bef3-70ec6e6e5b3b
First,1,0,1
Second,1,1,0
```

First state

Due to the closed switches S1 and S3, N4.EquivalentNode = N1 and N6.EquivalentNode = N3. Therefore the following nodes will be given simulation indexes in order: N1, N2, N3, N5, N7, N8. Since there are 6 distinct non-reference nodes and 1 voltage source, the A matrix will be 7x7.

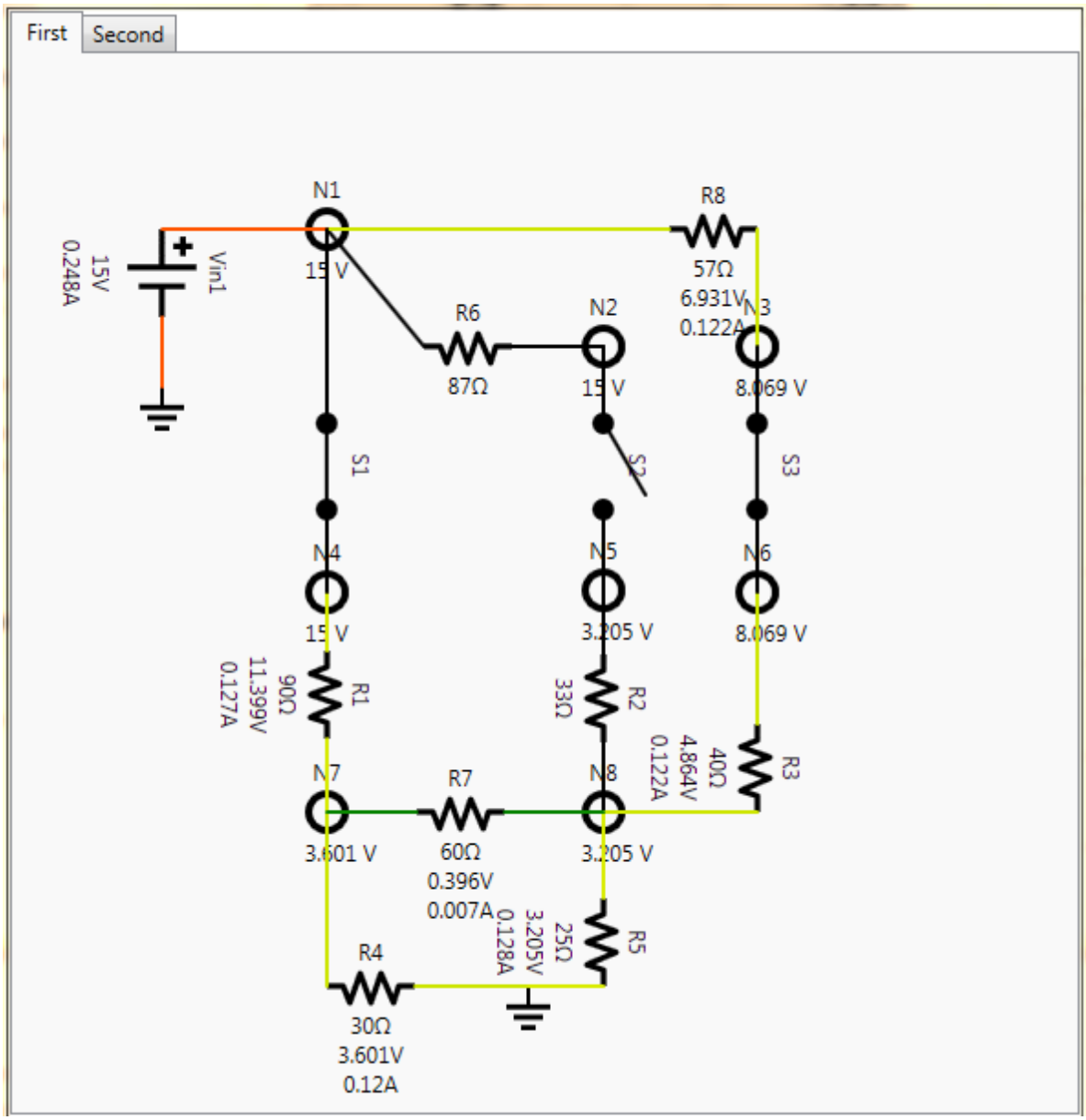
The values computed by MNA:

$$A = \begin{matrix} 0.0401 & -0.0115 & -0.0175 & 0 & -0.0111 & 0 & 1 \\ -0.0115 & 0.0115 & 0 & 0 & 0 & 0 & 0 \\ -0.0175 & 0 & 0.0425 & 0 & 0 & -0.025 & 0 \\ 0 & 0 & 0 & 0.0303 & 0 & -0.0303 & 0 \\ -0.0111 & 0 & 0 & 0 & 0.0611 & -0.0167 & 0 \\ 0 & 0 & -0.025 & -0.0303 & -0.0167 & 0.112 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}, \quad b = \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 15 \end{matrix}$$

Since there is a single solution, x is immediately returned:

$$x = \begin{matrix} 15 \\ 15 \\ 8.069 \\ 3.2051 \\ 3.6014 \\ 3.2051 \\ -0.2482 \end{matrix}$$

The circuit values are updated and the result is displayed in a tab:



Second state

Due to the closed switches S1 and S3, N4.EquivalentNode = N1 and N5.EquivalentNode = N2. Therefore the following nodes will be given simulation indexes in order: N1, N2, N3, N6, N7, N8. Since there are 6 distinct non-reference nodes and 1 voltage source, the A matrix will be 7x7.

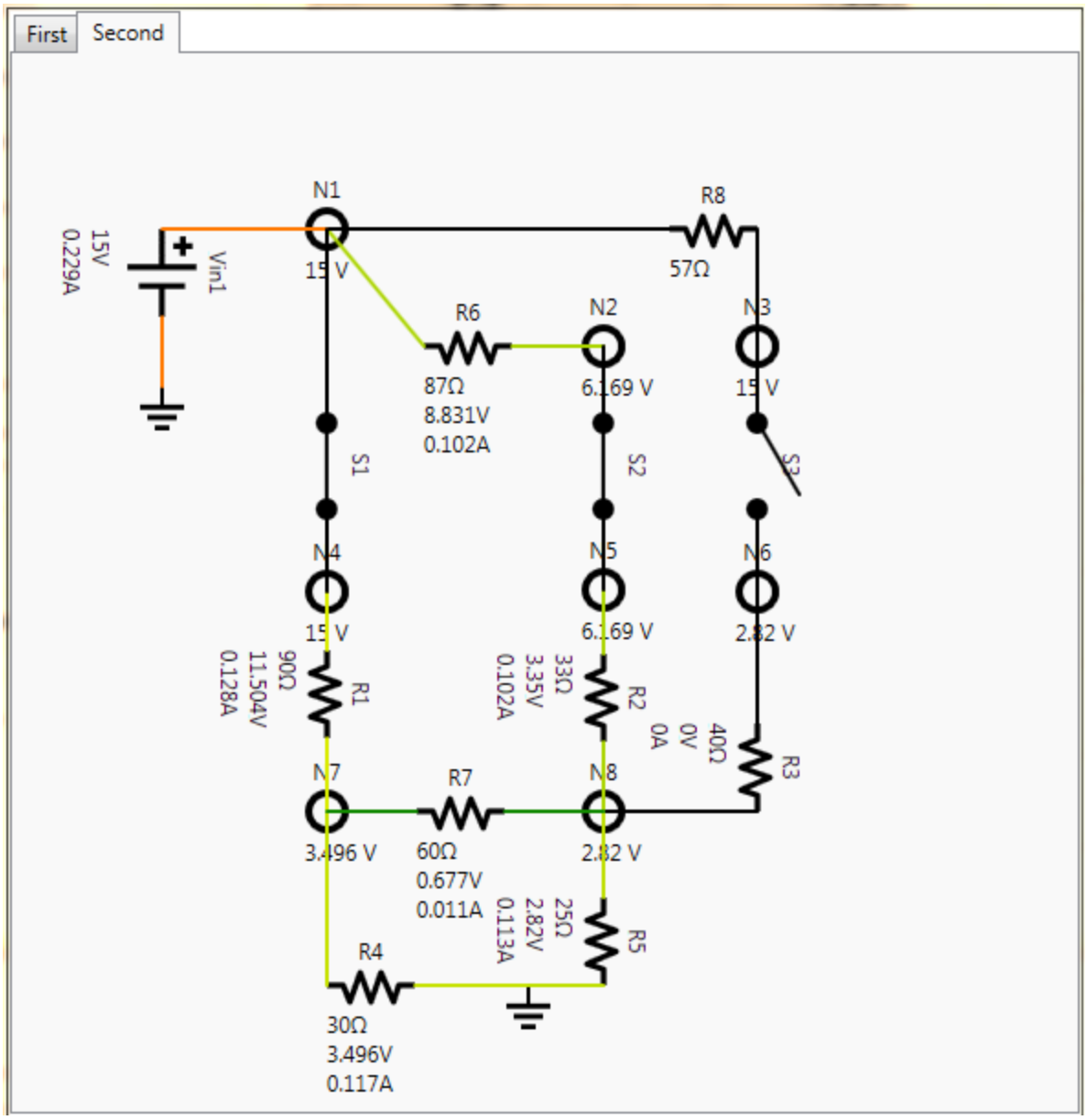
The values computed by MNA:

$$A = \begin{matrix} 0.0401 & -0.0115 & -0.0175 & 0 & -0.0111 & 0 & 1 \\ -0.0115 & 0.0418 & 0 & 0 & 0 & -0.0303 & 0 \\ -0.0175 & 0 & 0.0175 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.025 & 0 & -0.025 & 0 \\ -0.0111 & 0 & 0 & 0 & 0.0611 & -0.0167 & 0 \\ 0 & -0.0303 & 0 & -0.025 & -0.0167 & 0.112 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}, \quad b = \begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 15 \end{matrix}$$

Since there is a single solution, x is immediately returned:

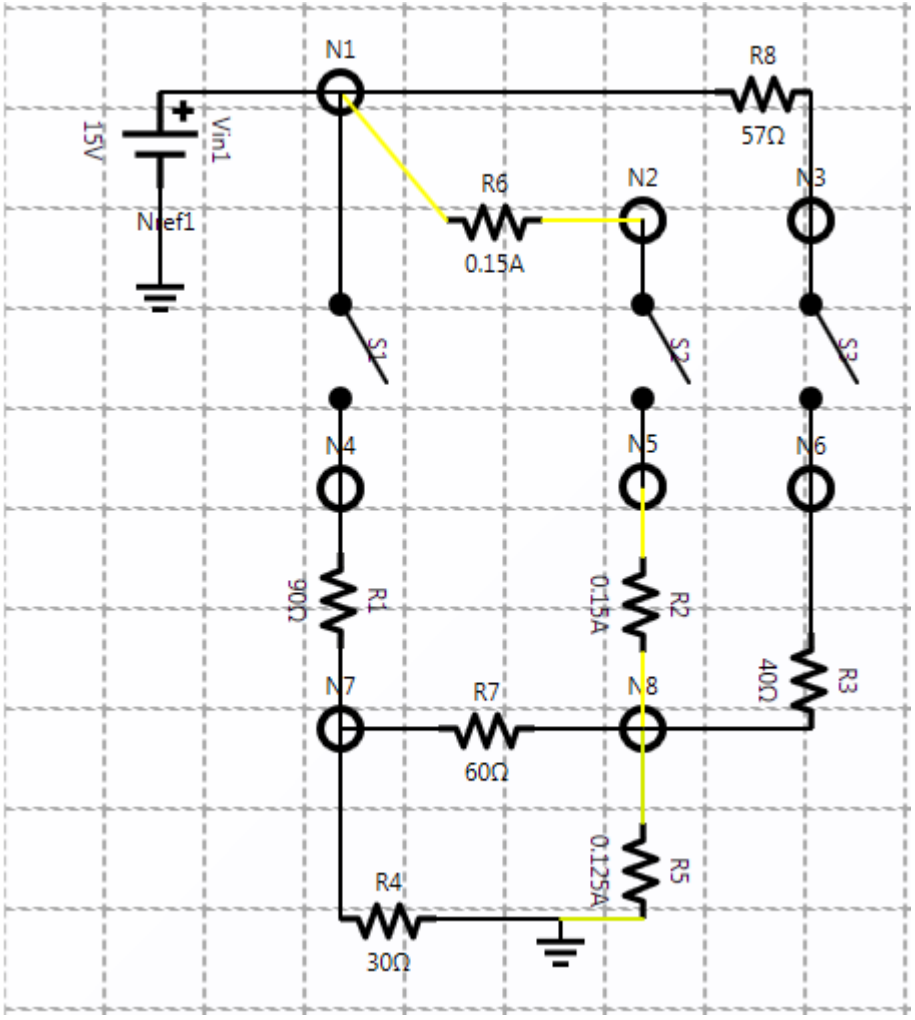
$$x = \begin{matrix} 15 \\ 6.1692 \\ 15 \\ 2.8195 \\ 3.4962 \\ 2.8195 \\ -0.2293 \end{matrix}$$

The circuit values are updated and the result is displayed in a tab:



Multiple solutions

Let's take the same circuit but change some of its resistors to known current and unknown resistance.



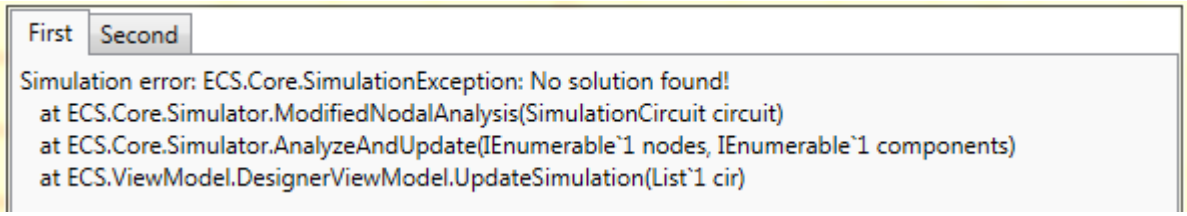
New XML representation:

```
<?xml version="1.0"?>
<CircuitXml xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Nodes>
    <Node X="65.90333333333308" Y="166" Id="beeab9af-766e-4caf-997c-31ab767f7b5b"
Name="Nref1" Voltage="0" IsReferenceNode="true" />
    <Node X="155.90333333333331" Y="80" Id="43f59906-86f3-4372-a5c5-c282818d4115" Name="N1"
Voltage="0" IsReferenceNode="false" />
    <Node X="306.90333333333331" Y="144" Id="f3adbbe3-ef22-4d35-bbbb-31a0269604df" Name="N2"
Voltage="0" IsReferenceNode="false" />
    <Node X="390.90333333333331" Y="144" Id="e1a1c731-3c0d-4664-943a-8d24c45ac0c4" Name="N3"
Voltage="0" IsReferenceNode="false" />
    <Node X="155.90333333333331" Y="278" Id="bd2b6247-aa01-4b52-8723-61f1c822c664" Name="N4"
Voltage="0" IsReferenceNode="false" />
    <Node X="306.90333333333331" Y="277" Id="4f81f151-c4f8-4bdf-ba7e-7c62287cd59b" Name="N5"
Voltage="0" IsReferenceNode="false" />
    <Node X="390.90333333333331" Y="278" Id="c4fc78c1-d17a-4213-80e7-5172bba4189b" Name="N6"
Voltage="0" IsReferenceNode="false" />
    <Node X="155.90333333333331" Y="398" Id="8c189e1d-653f-4a79-926d-cf15e5d4ea3b" Name="N7"
Voltage="0" IsReferenceNode="false" />
    <Node X="306.90333333333331" Y="398" Id="c48b4bcb-a466-4338-9e7c-96498a0c78be" Name="N8"
Voltage="0" IsReferenceNode="false" />
```

```
    <Node X="265.90333333333331" Y="493" Id="c90e05be-15ec-4e61-908d-207f97d88edd" Name=""
Voltage="0" IsReferenceNode="true" />
  </Nodes>
  <Resistors>
    <Resistor X="154.90333333333331" Y="322" Id="b3f06812-4096-4b9f-8d96-b0eb7099da04"
Name="R1" Direction="Vertical" Resistance="90" Voltage="0" Current="0">
      <Node1Id>bd2b6247-aa01-4b52-8723-61f1c822c664</Node1Id>
      <Node2Id>8c189e1d-653f-4a79-926d-cf15e5d4ea3b</Node2Id>
    </Resistor>
    <Resistor X="305.90333333333331" Y="324" Id="f273635d-7c4b-4542-b21b-7ce90fea3924"
Name="R2" Direction="Vertical" Resistance="0" Voltage="0" Current="0.15">
      <Node1Id>4f81f151-c4f8-4bdf-ba7e-7c62287cd59b</Node1Id>
      <Node2Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node2Id>
    </Resistor>
    <Resistor X="389.90333333333331" Y="362" Id="9c2992fb-5460-459f-9640-92c063327f52"
Name="R3" Direction="Vertical" Resistance="40" Voltage="0" Current="0">
      <Node1Id>c4fc78c1-d17a-4213-80e7-5172bba4189b</Node1Id>
      <Node2Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node2Id>
    </Resistor>
    <Resistor X="167.90333333333331" Y="494" Id="632301ab-bfee-4221-9e7b-cb7ed6214e43"
Name="R4" Direction="Horizontal" Resistance="30" Voltage="0" Current="0">
      <Node1Id>8c189e1d-653f-4a79-926d-cf15e5d4ea3b</Node1Id>
      <Node2Id>c90e05be-15ec-4e61-908d-207f97d88edd</Node2Id>
    </Resistor>
    <Resistor X="305.90333333333331" Y="457" Id="c3073369-4fb5-446a-9121-5af5c87e24e8"
Name="R5" Direction="Vertical" Resistance="0" Voltage="0" Current="0.125">
      <Node1Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node1Id>
      <Node2Id>c90e05be-15ec-4e61-908d-207f97d88edd</Node2Id>
    </Resistor>
    <Resistor X="220.90333333333331" Y="145" Id="7c343178-87ef-4606-abfe-80e14aaba8b0"
Name="R6" Direction="Horizontal" Resistance="0" Voltage="0" Current="0.15">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>f3adbbe3-ef22-4d35-bbbb-31a0269604df</Node2Id>
    </Resistor>
    <Resistor X="216.90333333333331" Y="399" Id="c55837f9-1036-4065-a628-16510fb1b29e"
Name="R7" Direction="Horizontal" Resistance="60" Voltage="0" Current="0">
      <Node1Id>8c189e1d-653f-4a79-926d-cf15e5d4ea3b</Node1Id>
      <Node2Id>c48b4bcb-a466-4338-9e7c-96498a0c78be</Node2Id>
    </Resistor>
    <Resistor X="354.90333333333331" Y="81" Id="45d13a44-40ea-4f10-b731-08c411faf0f1"
Name="R8" Direction="Horizontal" Resistance="57" Voltage="0" Current="0">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>e1a1c731-3c0d-4664-943a-8d24c45ac0c4</Node2Id>
    </Resistor>
  </Resistors>
  <VoltageSources>
    <VoltageSource X="52.903333333333308" Y="92" Id="f10fd17d-912d-438a-bd73-1fd59ed94636"
Name="Vin1" Direction="Vertical" Voltage="15" Current="0">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>beeab9af-766e-4caf-997c-31ab767f7b5b</Node2Id>
    </VoltageSource>
  </VoltageSources>
  <Switches>
    <Switch X="154.90333333333331" Y="197" Id="36bc6496-5ebe-47e6-9cfc-56bc5bd0e970"
Name="S1" Direction="Vertical" IsClosed="false">
      <Node1Id>43f59906-86f3-4372-a5c5-c282818d4115</Node1Id>
      <Node2Id>bd2b6247-aa01-4b52-8723-61f1c822c664</Node2Id>
    </Switch>
    <Switch X="305.90333333333331" Y="197" Id="99824b8f-fba5-4096-b2fb-1b4d6e1d6c8e"
Name="S2" Direction="Vertical" IsClosed="false">
      <Node1Id>f3adbbe3-ef22-4d35-bbbb-31a0269604df</Node1Id>
      <Node2Id>4f81f151-c4f8-4bdf-ba7e-7c62287cd59b</Node2Id>
```

```
    </Switch>
    <Switch X="389.90333333333331" Y="197" Id="cc6fc249-8375-4794-bef3-70ec6e6e5b3b"
Name="S3" Direction="Vertical" IsClosed="false">
      <Node1Id>e1a1c731-3c0d-4664-943a-8d24c45ac0c4</Node1Id>
      <Node2Id>c4fc78c1-d17a-4213-80e7-5172bba4189b</Node2Id>
    </Switch>
  </Switches>
</CircuitXml>
```

Note that this change was designed for the second state only. Since we have set a current for R6, and S2 is open in the first state, no solution is found and the simulation of the first state will result in an error:



Let's take a look at the second state. As before, the following nodes will be given simulation indexes in order: N1, N2, N3, N6, N7, N8. Once again, since there are 6 distinct non-reference nodes and 1 voltage source, the A matrix will be 7x7.

The values computed by MNA:

$$A = \begin{matrix} 0.0287 & 0 & -0.0175 & 0 & -0.0111 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.0175 & 0 & 0.0175 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.025 & 0 & -0.025 & 0 \\ -0.0111 & 0 & 0 & 0 & 0.0611 & -0.0167 & 0 \\ 0 & 0 & 0 & -0.025 & -0.0167 & 0.0417 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}, \quad b = \begin{matrix} -0.15 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.025 \\ 15 \end{matrix}$$

In this case, there are multiple solutions to $Ax = b$, so the result selection algorithm will be used. The following values are computed first:

$$s = A^+b = \begin{matrix} 15 \\ 0 \\ 15 \\ 5.8125 \\ 4.3125 \\ 5.8125 \\ -0.26875 \end{matrix}, \quad F = I - A^+A = \begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Note that s would have been the single solution if there were only zeros in F. Also, there are no linear-dependent rows in F in this case.

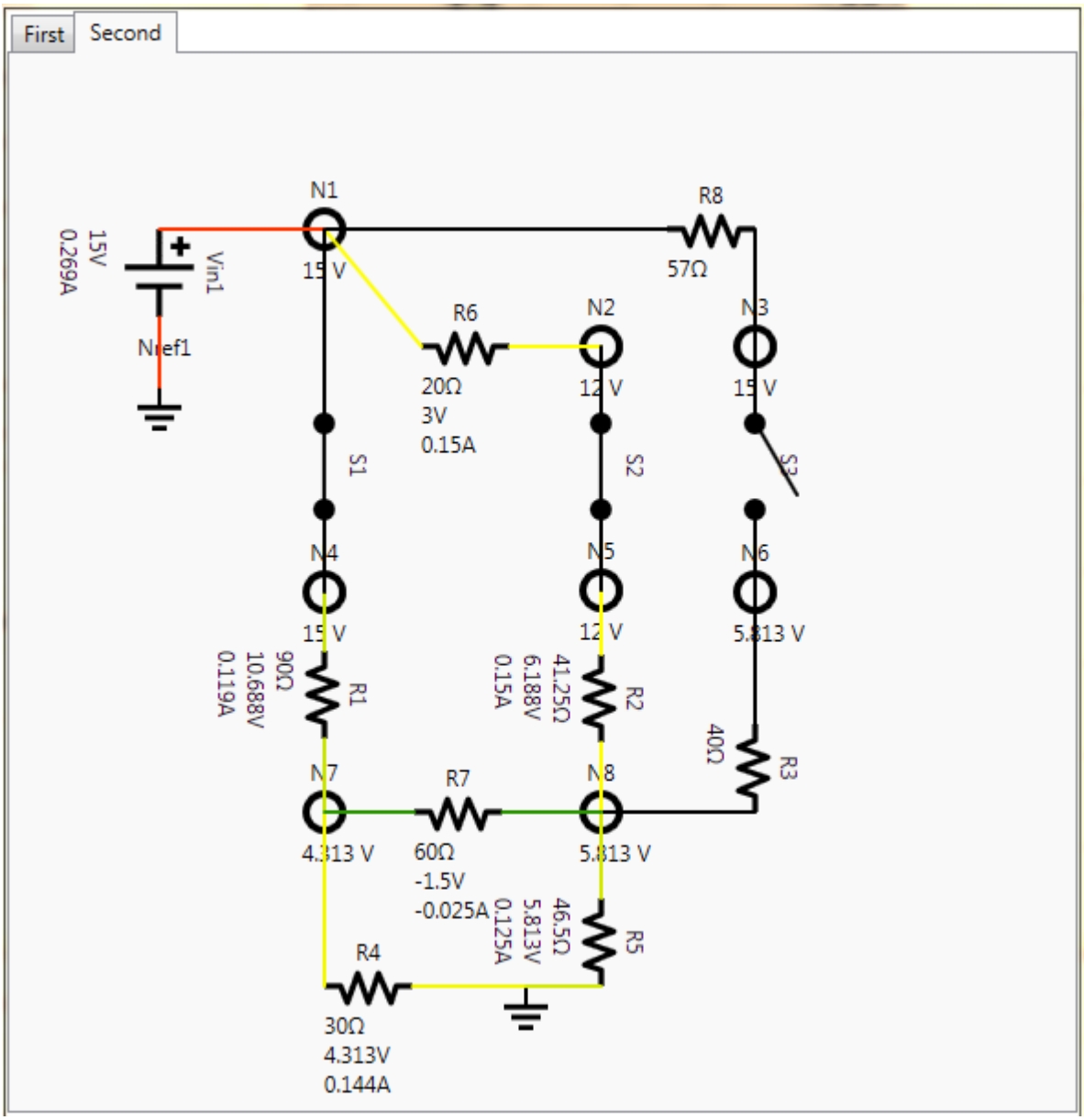
The result selection algorithm chooses to start from N1. It traverses in the order N1, N2, N7, N3, N8, N6 and computes the following values

$$d = \begin{matrix} 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ 15 \\ -0.26875 \end{matrix}, \quad e = \begin{matrix} 0 \\ -1 \\ -1 \\ -3 \\ -1 \\ -2 \\ 0 \end{matrix}, \quad m = \frac{15}{5}, \quad w = d - s + me = \begin{matrix} 0 \\ 12 \\ -3 \\ 0.1875 \\ 7.6875 \\ 3.1875 \\ 0 \end{matrix}$$

The new result is computed:

$$x = s + Fw = \begin{matrix} 15 \\ 12 \\ 15 \\ 5.8125 \\ 4.3125 \\ 5.8125 \\ -0.26875 \end{matrix}$$

Finally, the circuit values are updated and the result is displayed in a tab:



Want to try it yourself? This example is included with the source code in the files ExampleCircuit.xml, ExampleCircuit2.xml (known current version) and ExampleCircuit_States.csv.