

# Toward the Evolution of Analog Computers for Control of Data Networks

Nathan Ainslie  
Ryan Baula  
Nathan Deckard  
Luke Fowler  
Richard Li  
Mark Meiss  
Ye Myint

Computer Science Department  
Indiana University  
Bloomington, Indiana 47405

## *Abstract*

*The acceptance of analog computers as practical hardware solutions has been hindered by their often unintuitive design methods, especially for those used to digital design. We describe our experiments aimed at furthering the understanding of extended analog computers (EACs) and their potential applications, especially in Internet traffic management. We stepped around the design problem by automating the design process with genetic algorithms. This GA was used to evolve an EAC that computes exclusive or (XOR) in the same manner as Jonathan Mills' explicitly designed solution. We then attempted the evolution of an EAC to compute probabilities for the Random Early Dropout (RED) algorithm for network traffic control. This effort was less successful, probably due to a lack of expressiveness in our GA representation. Finally, we speculate as to possible application of extended analog computers in network traffic control. In particular, we discuss how one might use an analog computer to detect anomalies in Internet traffic and use this information to focus the search of existing digital systems.*

## 1: Evolving an EAC to Compute XOR (Part 1)

Because of the highly experimental and novel aspects of this research, we actually created two independent genetic simulations in our attempt to evolve the exclusive or function. If both systems were able to evolve a solution that worked as well as the known solution, this would be evidence that we owed this success to our genome and the evolution process rather than our particular implementation. We present both of these implementations (*evo-eac* and *ga*) in turn and discuss their performance on the XOR problem.

### 1.1: The *evo-eac* System

The *evo-eac* system is based on the premise that there are three steps towards evolving any EAC:

1. We must specify the genome that represents the EAC and how two genomes may combine to produce an offspring.
2. We must determine a fitness function to evaluate the genomes
3. We must construct an overseer function to determine which genomes get to mate with which other genomes, how many times they get to mate, and so forth.

This basic approach to genetic algorithms lies at the heart of *evo-eac*; thus far, it has appeared to be fairly successful.

### 1.1.1: The Genome

Our genome for the *evo-eac* system is partitioned into four *chromosomes*: the sheet inputs, the sheet outputs, the Lukasiewicz Logic Arrays (LLAs), and the sheet spacing. The sheet inputs and outputs indicate the spots on the conductive sheet at which current is either applied or measured. The LLAs have both a location and a piecewise linear function associated with them. The spacing is a floating point value which represents the distance between the potential contact points on the conductive sheet.

These chromosomes are in turn divided into *codons*. A single codon is associated with each connection point on the conductive sheet and represent a single input, output, or LLA.

Both the sheet inputs and output are represented as arrays of nine bits. The locations are numbered from 0 through 8 as shown below. Bit  $i$  in the input array is set if there is an input at the corresponding location on the sheet, and likewise for the output array.

```

0 1 2
3 4 5
6 7 8

```

For example, a sheet with inputs at locations 0 and 8 and an output at location 2 would have the following chromosomes for its inputs and outputs:

```

InputCodons = { 1, 0, 0, 0, 0, 0, 0, 0, 1 }
OutputCodons = { 0, 0, 1, 0, 0, 0, 0, 0, 0 }

```

The codon for the LLA associated with a particular sheet location is represented by an ordered triplet. Because we associate an LLA with each sheet location, an LLA is active only if there is an output at the same location. We use the same indexing for the LLAs as we do for the sheet locations.

The ordered triplets contain the values of the LLA at each of the control points (0.0, 0.5, 1.0). To avoid using real numbers unnecessarily, we encode a value of 0.0 as 0x00, 0.5 as 0x08, and 1.0 as 0x10. Thus, the “notch” Lukasiewicz function used in the canonical

solution of XOR would be encoded as (0x00, 0x10, 0x00). The identity function  $y = x$  would be encoded as (0x00, 0x08, 0x10).

The spacing value is represented with a single floating point value that specifies the horizontal and vertical distance between the connection points on the conductive sheet. This is expressed in abstract unit that serve to affect the scale of the calculations in our simulation.

### 1.1.2: Reproduction

The makeup of a new genome generated from two parent genomes is affected by two major forces: *crossover* and *mutation*. For the three array-valued chromosomes, crossover causes the offspring to copy codons directly from one parents until some randomly generated crossover point is reached, after which the rest of the codons in the chromosome are copied from the other parent. There is also a small but finite chance that any codon may be affected by a mutation, which causes a random value to be copied into the offspring.

For the spacing chromosome, crossover is achieved by generating a random value between 0.0 and 1.0. One parent's spacing is multiplied by this value, and the other parent's spacing is multiplied by 1.0 minus this value. These two adjusted values are then summed to get the spacing for the offspring. In this way we calculate a randomly weighted average of the two values. In the case of a mutation, we set the spacing of the offspring to a random value.

### 1.1.3: Evaluating Fitness

We compute the fitness of a genome by running a simulation of the EAC it represents. We bind input values to the input wires in sequential order and do the same for outputs. In other words, the input with the lowest-numbered index is the “first” input.

We have tried a variety of methods for calculating the fitness from this simulation. One method is to average the simulation errors for the four possible XOR input combinations (lower scores are more fit). However, this sometimes results in a good solution for three of the input patterns and a poor solution for the fourth input pattern; the good solutions push the average down and make an otherwise unfit organism appear fit. This pitfall can be avoided by using the total error as a fitness value rather than the average error. The current version of *evo-eac* uses the sum of errors rather than the average error as a fitness value (lower scores are still more fit).

### 1.1.4: The Overseer

The task of the overseer is to manage the selective breeding process over many generations. Our overseer currently takes potential parents from the most-fit 10% of a generation. Using a uniform distribution, it randomly selects pairs of genomes to mate

with each other to populate the next generation. The simulation can either run for a specified number of generations or halt when some error threshold has been reached.

### 1.1.5: Results

As shown in Figure 1, the *evo-eac* program has yielded very promising results.

```
Spacing: 1.128379
Sheet Inputs: |0|0|0|0|1|1|0|1|0|0
Sheet Outputs: |0|0|0|1|1|0|0|0|0|0
LLA's:
|0|0|0|16|8|8|0|16|8|0|16|0|8|8|8|16|16|8|16|8|16|8|8|8|16|0|0
Error: 0.000001

Test1 (0 XOR 0): f(0.000000) = 0.000000, Error: 0.000000
Test2 (1 XOR 0): f(0.500000) = 1.000000, Error: 0.000000
Test3 (0 XOR 1): f(0.500000) = 1.000000, Error: 0.000000
Test4 (1 XOR 1): f(1.000000) = 0.000001, Error: 0.000001
```

## 1.2: Figure 1: An excellent XOR candidate, evolved in 6084 generations

We developed the *ga* system in an attempt to confirm the promising results we had gotten when using *evo-eac*. The basic design philosophy is the same, but some details of the implementation and the genetic algorithm differ.

### 1.2.1: The Genome

In the genome for *ga*, the locations of the inputs and outputs are both stored in the same array so that no individual can have an input and an output at the same location. This array is indexed by location in the same way as the one from *evo-eac*; each value is one of *CXN\_NONE*, *CXN\_INPUT*, or *CXN\_OUTPUT*.

The LLAs are stored in much the same manner in *ga* as in *evo-eac*, except that we store the values for the control points as the floating point numbers 0.0, 0.5, and 1.0.

The spacing chromosome behaves in exactly the same way as it does in *ga*.

We also introduce the notion of a *species*. Every individual in our GA simulator is a member of some species. Every member of a species has the same number of inputs, number of outputs, number of connection points on the conductive sheet, and sheet resistivity.

### 1.2.2: Reproduction

The offspring of two individual receives the either connection array from one of its parents. Each LLA function is randomly selected from one of the parents. The spacing is also copied directly from one of the parents.

Each of the three chromosomes (connection, LLA, and spacing) has an independent probability of mutating during a generation of the GA simulation.

### 1.2.3: Evaluating Fitness

The same basic simulation technique is used in *ga* as in *evo-eac*. It would be good to have another simulation model with greater level of detail so that we can verify that the success of both systems is not a feature of the simulation model itself.

### 1.2.4: The Overseer

The overseer plays much the same role in *ga*, but it uses a different method of deciding which individuals get to reproduce. We always maintain a population of  $n$  individuals. When breeding the next generation, we first sort the existing individuals in order of decreasing fitness. The  $k^{\text{th}}$  individual in the new generation is the result of breeding two individuals randomly selected from the top  $k$  individuals in the current generation. This method of selection seems to give us rapid convergence toward a solution while also maintaining diversity in the gene pool.

### 1.2.5: Results

Our results in evolving an XOR EAC with *ga* are much the same as with *evo-eac*. The following is the result of breeding 1,000 individuals for 10 generations:

```
Gen #0: Best individual -- index = 423, sse = 0.0888
Gen #1: Best individual -- index = 10, sse = 0.2148
Gen #2: Best individual -- index = 188, sse = 0.0888
Gen #3: Best individual -- index = 0, sse = 0.0888
Gen #4: Best individual -- index = 2, sse = 0.0888
Gen #5: Best individual -- index = 0, sse = 0.0888
Gen #6: Best individual -- index = 3, sse = 0.0888
Gen #7: Best individual -- index = 831, sse = 0.0065
Gen #8: Best individual -- index = 0, sse = 0.0065
Gen #9: Best individual -- index = 0, sse = 0.0065
```

3 x 3 sheet, spacing = 0.811

```

      IN
  ___ ___ ___
  ___ ___ ___
  ___ ___ ___

      OUT
  ___ .* ___
  ___ ... ___
  ___ *. * ___

  IN
  ___ ___ ___
  ___ ___ ___
  ___ ___ ___
```

```

Test point 1: In = ( 1.0 1.0 ), Out = ( 0.0 ), Eval = ( 0.0 ), Err = ( 0.00 )
Test point 2: In = ( 1.0 0.0 ), Out = ( 1.0 ), Eval = ( 1.0 ), Err = ( 0.00 )
Test point 3: In = ( 0.0 1.0 ), Out = ( 1.0 ), Eval = ( 1.0 ), Err = ( 0.00 )
Test point 4: In = ( 0.0 0.0 ), Out = ( 0.0 ), Eval = ( 0.0 ), Err = ( 0.00 )

```

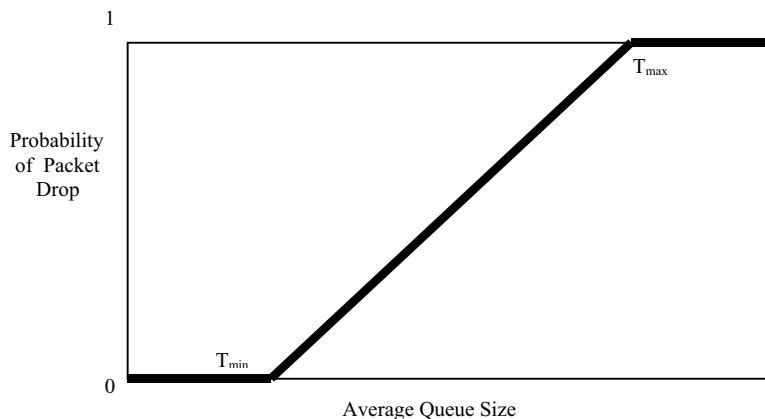
## 2: Evolving an EAC to Compute RED

After developing our genetic algorithm simulations and successfully reproducing the XOR function, we turned our sights to the Random Early Dropout (RED) algorithm.

### 2.1: Introduction to RED

TCP networks often face congestion, in which case routers must drop some or all packets in order to keep network performance at an acceptable level. One means of selecting packets to drop is the RED algorithm described by Floyd and Jacobson. RED first estimates the average size of a packet queue over a period of time. Associated with this average is a set of thresholds that specify the probability that any given packet will be dropped. Under some minimum threshold, the RED algorithm drops no packets. Above this minimum threshold, packets are dropped with a probability that increases linearly as a function of the average queue size. After a maximum threshold, all packets are dropped.

RED drops packets randomly according to the algorithm's probabilistic weights. It thus avoids the global synchronization issues that arise in other congestion avoidance algorithms when many nodes decrease their windows simultaneously. This global synchronization problem can cause large variations in queue size: at times a queue may be completely full, whereas at other times the same queue has excess capacity. Random selection of the packets to drop ensures that the queue will never become full.

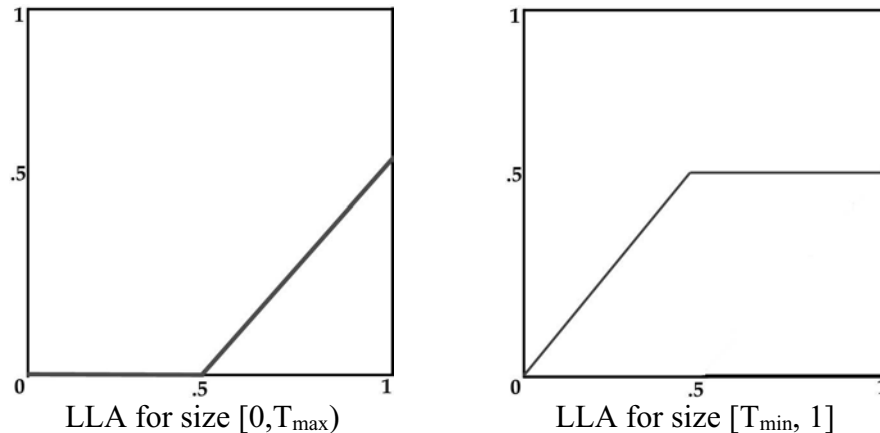


### 2.2: Using EACs to Solve RED

The relative simplicity of the RED algorithm suggests that genetic algorithms may be able to evolve an EAC to solve the RED function. Our tentative solution uses an EAC with a single conductive sheet to solve two segments of the RED function. The EAC has

a single input: the average queue size, which is normalized to a floating-point value between 0.0 (an empty queue) and 1.0 (a full queue). We evolved one EAC to solve RED for all queue sizes  $[0, T_{\max})$ , and another to solve RED for all queue sizes  $[T_{\min}, 1]$ .

These EACs had a single output, to which we connected a single LLA.



We evolved both of these EACs using our EAC evolution and simulation system. To test the quality of our evolved solutions, we generated random test points in the range of acceptable inputs. These test points were run through the simulation software to produce an error rate. The total sum of squares error for the sheet that calculates RED for queue sizes  $[0, T_{\max})$  was as low as 0.09 over a set of 21 test points. The error for queue sizes  $[T_{\min}, 1]$  was as low as 0.07 over a set of 21 test points.

These single-sheet EACs for computing RED could be a good companion to a digital system such as a router. When the average queue size is below the  $T_{\max}$  threshold, the router could send this queue size to the  $[0, T_{\max})$  EAC. When the average queue size is above  $T_{\min}$ , the  $[T_{\min}, 1]$  EAC could be used. These EACs are able to compute RED more quickly than any digital device, informing the router with virtually no delay with what probability it should drop packets.

### 2.3: Sample Results for RED

Output for size  $[T_{\min}, 1]$ :

```
Best individual : index = 0, sse = 0.0708
min_thresh = .103975, max_thresh = .735772;
Point: 0.200599, Expected: 0.152935, voltage: 0.116286, results: 0.232572
Point: 0.358134, Expected: 0.402279, voltage: 0.207608, results: 0.415216
Point: 0.403223, Expected: 0.473646, voltage: 0.233746, results: 0.467492
Point: 0.116299, Expected: 0.019506, voltage: 0.067418, results: 0.134836
Point: 0.277924, Expected: 0.275324, voltage: 0.161111, results: 0.322222
Point: 0.860278, Expected: 1.000000, voltage: 0.498699, results: 0.997397
Point: 0.874260, Expected: 1.000000, voltage: 0.506803, results: 1.000000
Point: 0.142212, Expected: 0.060521, voltage: 0.082440, results: 0.164879
Point: 0.398537, Expected: 0.466228, voltage: 0.231030, results: 0.462059
Point: 0.905926, Expected: 1.000000, voltage: 0.525160, results: 1.000000
Point: 0.156096, Expected: 0.082497, voltage: 0.090488, results: 0.180977
```

```

Point: 0.972238, Expected: 1.000000, voltage: 0.563601, results: 1.000000
Point: 0.854550, Expected: 1.000000, voltage: 0.495378, results: 0.990756
Point: 0.555077, Expected: 0.713998, voltage: 0.321775, results: 0.643550
Point: 0.135905, Expected: 0.050539, voltage: 0.078784, results: 0.157567
Point: 0.550948, Expected: 0.707463, voltage: 0.319381, results: 0.638763
Point: 0.361276, Expected: 0.407252, voltage: 0.209430, results: 0.418859
Point: 0.377712, Expected: 0.433267, voltage: 0.218957, results: 0.437915
Point: 0.839267, Expected: 1.000000, voltage: 0.486518, results: 0.973037
Point: 0.405381, Expected: 0.477061, voltage: 0.234997, results: 0.469994
Point: 0.206432, Expected: 0.162168, voltage: 0.119668, results: 0.239335

```

3 x 3 sheet, spacing = 1.488

```

OUT  IN
. **  ___
...  ___
* ..  ___

___  ___  ___
___  ___  ___
___  ___  ___

___  ___  ___
___  ___  ___
___  ___  ___

```

### Output for size $[0, T_{\max})$ :

```

Best individual : index = 0, sse = 0.0907
min_thresh = .103975, max_thresh = .735772;
Point: 0.276258, Expected: 0.272687, voltage: 0.170274, results: 0.340548
Point: 0.464122, Expected: 0.570036, voltage: 0.286066, results: 0.572132
Point: 0.220279, Expected: 0.184085, voltage: 0.135771, results: 0.271542
Point: 0.362924, Expected: 0.409862, voltage: 0.223692, results: 0.447384
Point: 0.665952, Expected: 0.889489, voltage: 0.410465, results: 0.820931
Point: 0.333037, Expected: 0.362556, voltage: 0.205270, results: 0.410541
Point: 0.599058, Expected: 0.783611, voltage: 0.369235, results: 0.738470
Point: 0.037283, Expected: 0.000000, voltage: 0.022980, results: 0.045959
Point: 0.362236, Expected: 0.408772, voltage: 0.223267, results: 0.446535
Point: 0.136836, Expected: 0.052011, voltage: 0.084340, results: 0.168680
Point: 0.508732, Expected: 0.640643, voltage: 0.313561, results: 0.627123
Point: 0.648127, Expected: 0.861276, voltage: 0.399479, results: 0.798958
Point: 0.253916, Expected: 0.237324, voltage: 0.156503, results: 0.313007
Point: 0.131304, Expected: 0.043255, voltage: 0.080930, results: 0.161860
Point: 0.266959, Expected: 0.257969, voltage: 0.164543, results: 0.329085
Point: 0.629163, Expected: 0.831261, voltage: 0.387791, results: 0.775581
Point: 0.189902, Expected: 0.136003, voltage: 0.117048, results: 0.234095
Point: 0.576994, Expected: 0.748688, voltage: 0.355635, results: 0.711271
Point: 0.338627, Expected: 0.371404, voltage: 0.208716, results: 0.417432
Point: 0.557515, Expected: 0.717857, voltage: 0.343630, results: 0.687259
Point: 0.055979, Expected: 0.000000, voltage: 0.034503, results: 0.069006

```

3 x 3 sheet, spacing = 1.016

```
— — —
— — —
— — —
IN  OUT
—  .** —
—  ... —
—  *.. —

— — —
— — —
— — —
```

### 3: Network Traffic Control

Having experienced success with the evolution of the XOR and RED function, we went on to consider the possible roles of analog computers in data network management.

#### 3.1: Background on Network Management

Quite a variety of different problems fall under the umbrella term of *network management*. Network management can mean distributing network traffic equitably over several interfaces (load balancing), gathering statistics on network traffic and performance (capacity planning), reacting to emergency situations on the network by blocking or rerouting certain types of traffic (traffic filtering), or adjusting the rate of flow of particular streams of data in the network (traffic shaping).

In each of these problem domains we are faced with a situation that makes the use of either analog or digital computers quite a challenge. It is now becoming common for long-distance networks to use OC-192 fiber connections that transfer data at 10.7 gigabits per second—a speed fast enough to transmit two CDs full of data every second. Even local area networks may run at gigabit speeds, which is enough bandwidth to transmit almost 100 DVDs every hour. At the time of this writing, even the fastest processors used in modern workstations run at speeds slower than 3.0 GHz. The network routers that handle these high rates of traffic cannot use conventional processors to transmit their data; they must use specially designed circuits and dedicated hardware to achieve line-rate performance.

With the network running so quickly, network management is a difficult challenge. The computers we would like to use to manage data networks—i.e., off-the-shelf, commodity hardware—simply cannot keep up with the full flow of network traffic. A high-end workstation is hard-put just to put a gigabit per second of data onto a network connection, let alone to analyze it and react to particular features. Using conventional PC hardware to analyze and control the full flow of network traffic moving at OC-192 data rates is not possible in the foreseeable future.

This need for speed would suggest that network management is a possible area in which to make use of the extreme speeds offered by analog computers. Unfortunately, many of

the decisions made in network management involve types of data that are difficult to represent in forms suitable for analog computation. There are many cases in which similar numeric values do not have similar meanings. How does one encode a TCP port number, a network prefix, or an autonomous system number for an analog computer?

We therefore suggest seeking a hybrid approach in which analog computers are used to perform portions of network management tasks in which speed is more critical than accuracy, and the analog results are then used to support or focus the tasks of digital systems. We thus use the analog computer to boost the effective operating speed of the digital system without seeking either a purely analog or purely digital solution.

### 3.2: Network Anomaly Detection

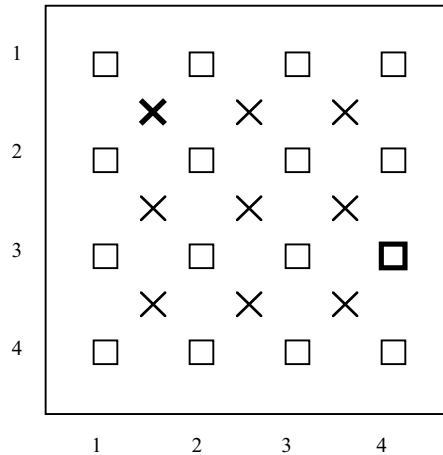
A great deal of network management revolves around the detection of unusual conditions in network traffic. This sort of detection is somewhat challenging because *all* network traffic is anomalous when examined at fine enough a time scale. Systems must therefore gather background data over a span of time and consider whether current rates of various types of traffic are within the ranges predicted by this background data.

All of this involves the analysis of *flow data* from network routers. Flow data contains information about the individual conversations taking place over the network: who is talking to whom, what sort of protocol they are using, how fast they are talking, and so forth. Neither routers nor network management systems are capable of handling the flow information for very fast network interfaces, so routers generally inform management systems about only a random sample of the flows they are managing. This random sample is assumed to be representative of the traffic as a whole.

We suggest that instead of using completely random sampling, routers can provide more finely-grained flow data for interfaces on which some sort of anomalous condition is already suspected. This suspicion of anomaly can be provided by an analog computer that works using inputs that are easily conditioned to the analog domain.

In our hypothetical system, there are two sorts of EACs in each router. The *global* EAC examines all of the traffic on all interfaces and tries to determine whether any anomaly currently exists on the router as a whole. The *interface* EACs examine the traffic on each interface and try to determine whether a particular interface is experiencing an anomaly.

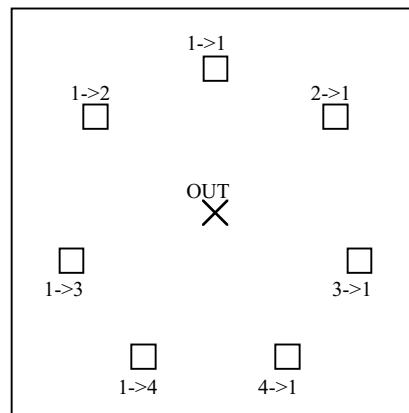
The conductive sheet of the global EAC has a two-dimensional grid of inputs; the voltage applied at a particular input is proportional to the amount of network traffic passing from the corresponding interface on the *y*-axis to the corresponding interface on the *x*-axis. These inputs are represented as squares on the diagram below. A output is located in the middle of every block of four inputs; this output will have an LLA associated with it.



The boldface square represents the output of the router for traffic passing from interface 3 to interface 4 on the router, normalized by the maximum capacity supported. The boldface output would be affected by every input, but most directly by the traffic from interfaces 1 to 1, 1 to 2, 2 to 1, and 2 to 2.

We can then characterize different patterns in the aggregate outputs of the LLAs in much the same way that we do for the analog retina EAC. Different patterns of traffic will produce different “fingerprints” in the outputs of the LLAs, and this information can then be used to control the sampling of flow information for export to network management systems.

The interface EAC for a particular interface would have one input for each interface-to-interface pathway within the router with which that interface is involved. Each input would be placed equidistantly in a ring around the center of the sheet, where there would be a single output connected to an LLA. This configuration is shown in the diagram below.



We do not attempt to “fingerprint” the output in this case; we just attempt to configure a system in which individual levels of traffic over the expected values trigger a corresponding rise in the output of the entire circuit. This also produces a crude sort of

anomaly detection that could be used to focus the behavior of the flow exporter in the router.

#### 4: Conclusion

The field of analog computing is still in its infancy because of the long-standing problems in analog computer design. Our research indicates that genetic algorithms may be an effective means of bypassing the need for explicit system design; with GAs, we are able to build analog computers that we could never hope to design in the same amount of time.

Our results also indicate that a more expressive genome will be necessary to develop EACs for some tasks. The single-sheet EAC is simply not complex enough to solve some problems, and so the genetic algorithm must have a means of connecting multiple sheets together in order to solve these more difficult tasks. We are still trying to develop an effective way of combining sheets in our GA system, which will involve writing a more sophisticated simulator for evaluating our genomes.

We have hope that the network anomaly detection circuits that we describe in the previous section can be built in the near future. They rely only on inputs that we know to be available at some point inside a router. If by no other means, we can guess at current traffic levels simply by analyzing the voltages on individual traces on the circuit boards for each interface.

We believe that hybrid analog-digital designs like the one we describe in this paper represent the best chance of improving the performance of high-bandwidth data analysis systems in the near future. This will advance the state of the art in both the analog and digital domains and help to bring ever more complex problems within our grasp.

#### References:

- [1] Braden, B., et al. *RFC 2309: Recommendations on Queue Management and Congestion Avoidance in the Internet*. IETF, 1998.
- [2] "Cisco IOS NetFlow." From <http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml>.
- [3] Forristal, Jeff. "Fireproofing Against DoS Attacks." From *Network Computing*, December 10, 2001.
- [4] Floyd, S. and Van Jacobson. "Random Early Detection Gateways for Congestion Avoidance." In *IEEE/ACM Transactions on Networking*. 1993.
- [5] Heitkoetter, Joerg and David Beasley, eds. "The Hitch-Hiker's Guide to Evolutionary Computation." (FAQ for *comp.ai.genetic*.)

- [6] Mills, J. "Kirchhoff-Lukasiewicz Machines." From <http://www.cs.indiana.edu/~jwills/ANALOG.NOTEBOOK/klm/klm.html>.
- [7] Mills, J. "Lukasiewicz' Insect: The Role of Continuous-Valued Logic in a Mobile Robot's Sensors, Control, and Locomotion." In *Proceedings of the 23th International Symposium on Multiple-Valued Logic*, IEEE Computer Society, 1993.
- [8] Mills, J. and C. A. Daffinger. "CMOS VLSI Lukasiewicz Logic Arrays." In *Proceedings of Application Specific Array Procesors*, Princeton, N.J., 1990.
- [9] Obitko, Marek. *Introduction to Genetic Algorithms*. Web site at <http://cs.felk.cvut.cz/~xobitko/ga/>.