Genetic Programming for Sensor Networks

Thomas Weise

University of Kassel, Wilhelmshöher Allee 73, 34121 Kassel, Germany
weise@vs.uni-kassel.de

Abstract. In this paper we present an approach to automated program code generation for sensor nodes and other small devices using Genetic Programming. We give a short introduction to Genetic Algorithms. Our new Distributed Genetic Programming Framework facilitates the development of sensor network applications. Genetic evolution of programs requires program testing. Therefore we use a simulation environment for distributed systems of sensor nodes. The simulation model takes into account characteristic features of sensor nodes, such as unreliable communication and resource constraints. Two application examples are presented that demonstrate the feasibility of our approach and its potential to create robust and adaptive code for sensor network applications.

1 Introduction

Today we experience a growing demand for distributed systems of small devices, like sensor nodes [1]. Such devices are restricted in resources like memory size, processing speed, and battery power. The communication among them is not reliable and the topology of their network is volatile. The program code created for sensor nodes should thus be robust and as efficient as possible. Our goal is to develop a methodology that supports the development of software for sensors and similar resource-constrained devices.

In this paper we describe the structure and functionality of a framework that allows the automated code generation for sensors by Genetic Programming. Using a behavioral description called fitness function, programs that are close-to-optimal solutions for problems can be found with our system.

The benefits of genetically engineering code are manifold. Fitness functions can be combined using multi-objective Genetic Algorithms [2]. That way, programs can be created that perform more than one task. For example, sensor networks can be instructed to vote for a "master node" via election *and* send all important data to that node, whereby the significance of the data is measured by one fitness function and the election algorithm is defined by another one.

Genetic Programming is able to create functional code, but it can also pursue nonfunctional optimizations concerning code size, execution speed and network traffic. Furthermore, Genetic Programming allows prioritizing such quality aspects. For some applications, code size might be such a vital aspect that it becomes even more important than correctness – a program might be more useful even if it produces correct data only in 95% of all cases than one that is always right but doubled in size. For some applications, reliability of transmission could be the most important aspect while code size is only of secondary interest.

Another interesting aspect of Genetic Programming is that it constitutes one suitable back end for a Model Driven Architecture [3]. Providing a model of what a program should do in form of a fitness function, the user obtains automatically created and even optimized source code.

Separating the model from the target platform in a way close to MDA, fitness functions, once defined, may be reused for a variety of hardware or software environments. Therefore, we believe that Genetic Programming has the potential to become an integral part of modern programming methodologies for ubiquitous computing environments.

The remainder of this article is structured as follows: In section 2, we give a quick overview on the principles of Genetic Programming. Section 3 presents our DGP Framework. First experimental results may be found in section 4. Some information about related work and an outlook on future work conclude the article.

The prototype software of the project can be found in the internet at http://sourceforge.net/projects/dgpf, governed by an open source license (LGPL).

2 The Genetic Programming Approach

For a long time, Genetic Algorithms have been used in science to derive solutions for any type of problems, from construction of wind turbines [4] to pattern-recognition systems [5]. The application of Genetic Algorithms with the goal to evolve computer programs is called Genetic Programming [6]. This section will give an overview on how Genetic Algorithms work in common.

In nature, a species adapts to an environment because the individuals that are the fittest in respect to that environment will have the best chance to reproduce, possibly creating even fitter offspring. Though this is a very rough simplification, it sums up the basic idea of genetic evolution. A more detailed explanation would be out of scope here.

As shown in Fig. 1, Genetic Algorithms start with an initial population of random solution candidates, called individuals. In our case, the individuals are small programs that can be executed on sensor nodes. As in nature, the population will be refined step by step in a cycle of computing the fitness of its individuals, selecting the best individuals and creating a new generation derived from these. If a reasonable good solution has evolved, the algorithm will stop. [7]

One of the strengths of the genetic approach is that the fitness of the individuals can be computed in parallel. Reproduction can be treated in the same manner. That way, many possible solutions of a problem can be tested at once.

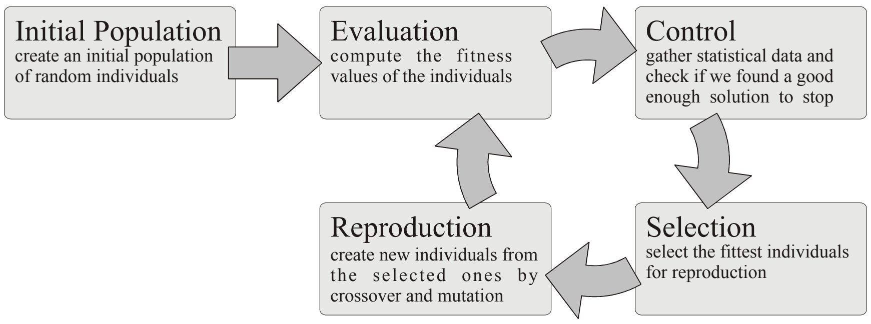

Initial Population
create an initial population of random individuals
Evaluation
compute the fitness values of the individuals
Control
gather statistical data and check if we found a good enough solution to stop
Selection
select the fittest individuals for reproduction
Reproduction
create new individuals from the selected ones by crossover and mutation

Fig. 1. Common cycle of Genetic Algorithms.

3 The Distributed Genetic Programming Framework

We have developed a Distributed Genetic Programming Framework (DGPF) based on Java. It provides a highly customizable, open-source infrastructure for the development of sensor software. The basic idea behind its architecture is to unleash the distribution ability of Genetic Algorithms mentioned in the previous section while preserving its versatility of being applied to many different problem domains. The overall structure of the DGPF is structured in multiple layers, as illustrated in Fig. 2.

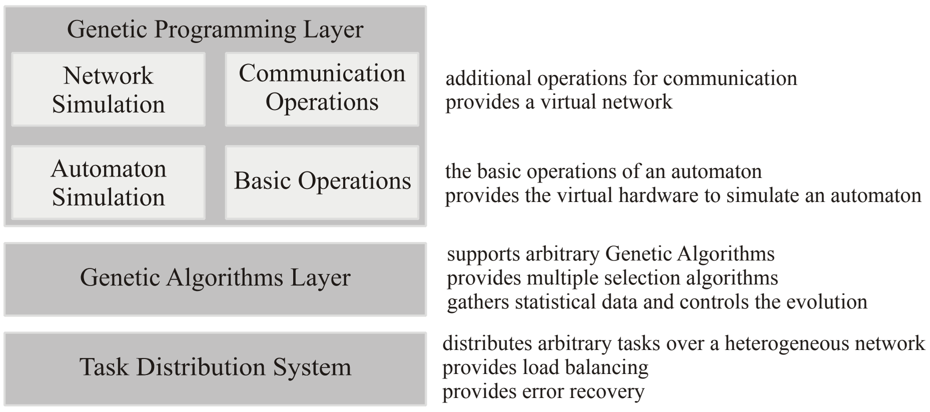

Genetic Programming Layer
Network Simulation
Communication Operations
additional operations for communication
provides a virtual network
Automaton Simulation
Basic Operations
the basic operations of an automaton
provides the virtual hardware to simulate an automaton
Genetic Algorithms Layer
supports arbitrary Genetic Algorithms
provides multiple selection algorithms
gathers statistical data and controls the evolution
Task Distribution System
distributes arbitrary tasks over a heterogeneous network
provides load balancing
provides error recovery

Fig. 2. The layered architecture of the Distribute Genetic Programming Framework.

3.1 The Task Distribution System

The base layer is formed by a task distribution system which allows one client system to let arbitrary tasks be processed by arbitrary many task servers. The integrated load balancing feature ensures that the load of tasks distributed to servers is proportional to their computational power. It also checks how many tasks one node can perform in parallel which may be more than one even for off-the-shelf PCs since their CPU may support, for example, Hyper-Threading [8]. Another service of that layer is error recovery which is especially useful when running experiments outside of a dedicated testbed. Even when all task-processing servers are rebooted simultaneously, the system keeps its stability (of course, it has to wait until all servers are available again).

3.2 The Genetic Algorithms Layer

Genetic Algorithms are a versatile tool for problem solving. We want to concentrate on Genetic Programming, especially for sensor networks. Our framework remains flexible by providing a general abstraction layer for Genetic Algorithms.

Using the Task Distribution System, the breeding of solutions for a problem will be distributed over a network analogously to the description in the previous chapter. Fig. 3 illustrates how the basic cycle of Figure 1 maps on our framework.

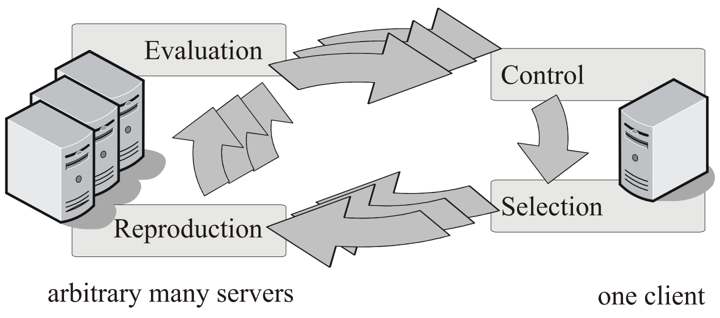


Fig. 3. Distribution of Genetic Programming

While reproduction and evaluation of an individual is located at a distant server, the new individual and its fitness are received at the client side.

To investigate a new problem domain using Genetic Algorithms, the user can introduce new types of individuals. For these individuals, he or she must provide

- creation, mutation and crossover routines
- a fitness function

In order to assist the user, a layer may be put on top of the Genetic Algorithm abstraction which delivers a template with creation/mutation/crossover routines but without a fitness function. This template could for example simulate a special hardware. This leaves the user the opportunity to specify different problems (using different fitness functions) to be solved by this hardware by using the intermediate layer.

The Genetic Algorithm layer also provides mechanisms to influence the parameters of the evolution while it is actually in progress. The mutation- and crossover-rate, the selection algorithm used, and many other parameters can be modified during the genetic process to maximize the performance and to improve the problem-space examination. Gathering statistical data, even a feedback loop can be created that enables dynamic tuning of the genetic evolution process. Optimal parameter adaptation strategies are currently subject of our research. This concept of central control was the main reason for not applying other distribution schemes, like the island model [9], [10]and the diffusion model [11].

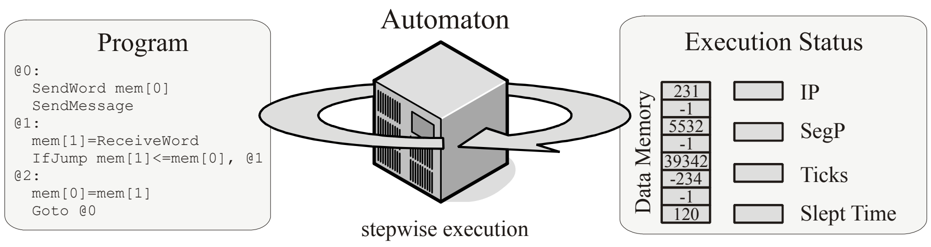
3.3 The Automaton Simulation Layer

Since we want to breed code that may run on sensor nodes, we need to simulate such distributed systems in order to evaluate the fitness of individuals. Our model for distributed systems is twofold; it divides into

- the separate automata, and
- the communication between them.

An automaton consists of a virtual hardware holding its execution status and a program running on that hardware. Unlike most other approaches in Genetic Programming which grow stateless functions, we have developed an automaton architecture with a fixed-sized memory. This allows us to breed programs that are stateful instruction sequences instead of expressions that must be called every time an input value changes to determine a new output.

An automaton is driven by a virtual CPU which executes one instruction of a program per tick. It also holds statistical information on the clock cycles performed and the time spent in sleep-mode. The step-by-step execution permits the specification of a run time limit which is needed since the programs grown by the DGPF are purposed to drive sensor nodes and hence do normally not terminate. They usually contain a main loop executing a sequence of actions infinitely. The simulation of an automaton may be paused at any time, allowing, for example, taking a look on its memory: This feature is essential when testing whether the value of a memory cell converges to a constant or changes infinitely, because one must compare the values at *t* and *t*+*n* for equality. In some experiments like the evolution of an automated unique ID creation system, this option has proved particularly useful.



```
@0:
  SendWord mem[0]
  SendMessage
@1:
  mem[1]=ReceiveWord
  IfJump mem[1]<=mem[0], @1
@2:
  mem[0]=mem[1]
  Goto @0
```

Fig. 4. The components of an automaton and its simulation.

In the Automaton Simulation Layer we also define the format of the instructions and expressions that can be executed on the virtual hardware. Using this format, we are able to specify mutation, crossover and individual creation routines for the Genetic Algorithm Layer, leaving only the fitness function subject to user implementation.

The predefined expression set contains constants, binary, and unary expressions - both logical and numerical - and a memory read instruction. Memory access can be done using either direct addressing to obtain a value stored at a specified address or by indirect addressing. For indirect access, the value stored at one address will be read and used as address of the target memory cell.

The basic instruction set of our automata consists of only three instructions: a conditional jump which has an expression and a target label as parameter, a memory write instruction which can write the value of an expression to a memory cell using either direct or indirect addressing, and a sleep instruction which puts the virtual CPU into sleep mode for a specified count of clock cycles.

The instruction set can be reduced to the one introduced by Teller [12], granting Turing completeness [13] in the usually applied informal sense.

3.4 The Network Simulation Layer

Resting on the Automaton Simulation Layer, the Network Simulation Layer extends the instruction and expression sets by communication primitives. Automata are now able to write data to their output buffer, send the contents of that buffer as a message, receive such a message in their input buffer and iterate over the data items in that input buffer. Monitoring that activity, the Network Simulation Layer also provides additional statistical data for each automaton, holding information on the number of messages sent, lost due to input buffer overflow, and successfully processed.

To model a distributed system, many automata are simulated in parallel for each grown program. For this set of automata, the following assumptions will hold:

1. All automata run at *approximately* the same speed, since they are based on the same virtual hardware. The execution speed might differ from automaton to automaton and cannot be regarded as constant either.
2. Hence, the system of automata runs asynchronously, like real sensor nodes running asynchronously even if they were switched on at the same time.
3. The automata will be started at different times.

The network simulator instance provides an absolute system time, keeps track on the transmissions currently underway, and also maintains global statistics. Fig. 5 shows how the automata of Fig. 4 will interact in a simulated network. Multiple automata need to be instantiated each time the fitness of an individual is evaluated; the network simulator can be reused. Since it is the most complex component in the simulation, this saves initialization time and memory capacity on the server nodes.

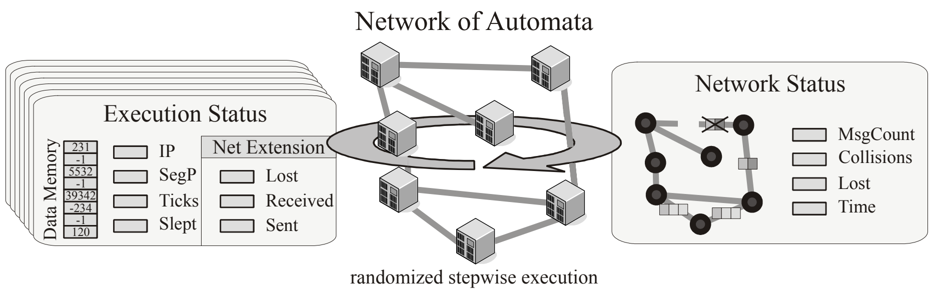


Fig. 5. The Network Simulator.

The network simulator evaluates systems that are connected wirelessly, and cannot a priori guarantee reliable communication [14]. It therefore has the following properties:

1. The links between the nodes are randomly created, yet it will be ensured that there are no network partitions.
2. Messages are simple sequences of memory words with no predefined structure.
3. Messages cannot be sent directly. Like radio broadcasts they will be received by any node in transmission distance. Finding out which message is of concern will be in the responsibility of each node.
4. Messages can get lost without special cause.
5. Transmissions may take a random time until they reach their target.
6. The collision of two transmissions underway leads to the loss of both messages.

Code working correctly in such an environment can also expected to be robust and adaptive in a real-world application. One example for such programs is shown in Fig. 6. We will refer to it again later.

```
@0:
  SendWord mem[0]
  SendMessage
@1:
  mem[1]=ReceiveWord
  IfJump mem[1]<=mem[0], @1
@2:
  mem[0]=mem[1]
  Goto @0
```

Fig. 6. A small example of code written in an assembler-like language. If each automaton were initialized with a random number in its first memory cell (*mem[0]*), the biggest one of these numbers would be known by all automata in the network after some time.

3.5 Evaluation of the Fitness function

As illustrated in Fig. 7, the fitness of an individual can be determined by running its code on all automata in a simulated network. The results of this simulation run will be compared to the results that the ideal behavior would yield. To obtain stable values, this process will be repeated multiple times.

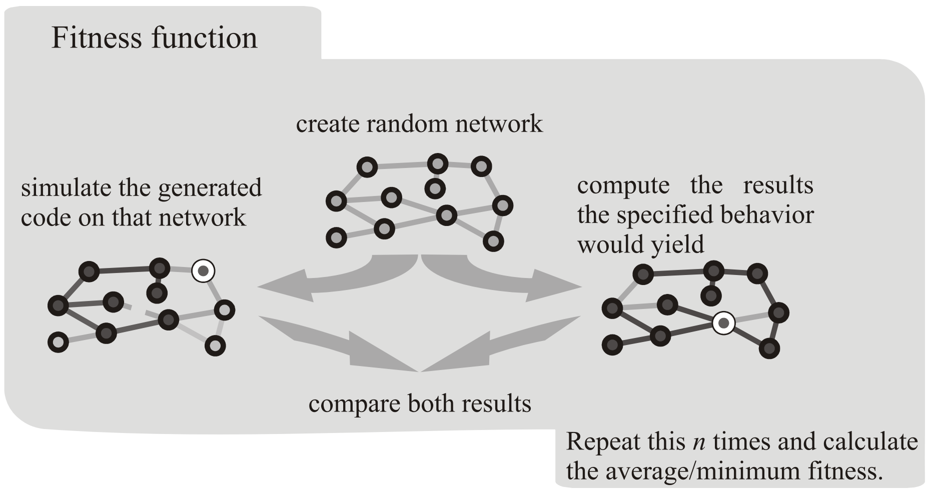

Fitness function
create random network
simulate the generated code on that network
compute the results the specified behavior would yield
compare both results
Repeat this *n* times and calculate the average/minimum fitness.

Fig. 7. Computation of the fitness function for sensor networks.

Starting with the problem introduced in Fig. 6, we will discuss fitness function in the following.
We want to solve a problem very similar to the well known election-problem. Each of our automata will be initialized knowing only a unique, random number. The DGPF should then grow a program allowing the nodes to find the maximum of these numbers.

The ideal program would lead to each automaton knowing the maximum number after its execution. A proper fitness function would thus be:

f(p) = the count of automatons knowing the maximum number.

This will work but also leads to a slow evolution because knowing the maximum number enforces a lot of steps to be taken by the program. An automaton must send its own unique number to its neighbors. Then, all received numbers must be compared to its number, storing them only if they are bigger. The sequence of these steps must be guessed by the evolution if there is no reward for intermediate results. Breeding speed can be increased by incorporating knowledge about interim results, like giving points for

- knowing the number of any other automaton
- knowing the number of any other automaton bigger as the own one

4 Experimental Results

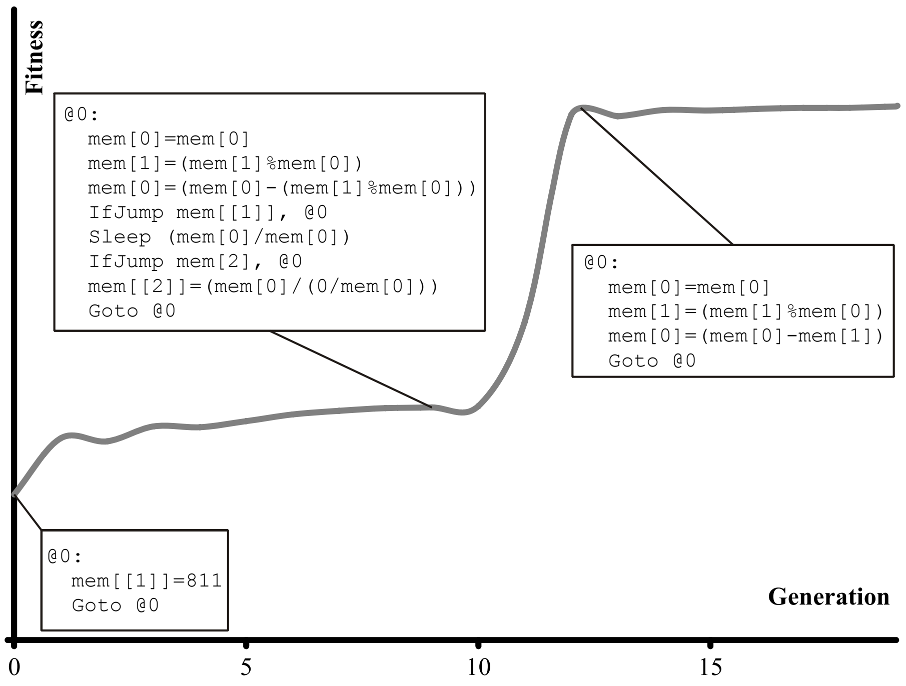
Our Genetic Programming framework has reached its second revision. The framework is a distributed system itself and unleashes the full parallelization potential of Genetic Programming. We will demonstrate the functionality of our approach using two examples. First we will look at the Euclidian Algorithm for determining the greatest common divisor (GCD) of two numbers. The second example will show the system's performance when growing solutions for election problem, (i.e. finding the maximum) mentioned in the previous chapter.

4.1 The GCD-Example

The first instance will only involve simulation on the Automaton Simulation Layer (see section 3.3). The objective is to breed an algorithm that finds the GCD of two numbers stored in the memory of an automaton. The fitness function for this case is straightforward. After executing some time the automaton will be stopped. The value it has now stored in its first memory cell will be compared with the GCD of two values initially stored in its memory. The fitness is inversely proportional to the difference of the real GCD to the computed value.

An evolutionary process was started, using only one PC for the computation with client and server running on it simultaneously. One of the test runs produced the fitness curve illustrated in Fig. 8: Starting with a bogus solution, problem space exploration derives a complex algorithm using lots of unnecessary instructions like putting the CPU into sleep mode for some time (`Sleep`) or indirect memory access (`mem[[n]]`). The complicated solution suggested a few cycles later, already producing correct results sometimes, is then refined to a more simple but not perfect version which is always right.[1]

[1] At this point it is useful to mention that the arithmetic operations division and modulo have been modified as proposed in [6]. A division (or modulo) by zero will not fail but yield the same result as a division (or modulo) by one.



```
@0:
  mem[0]=mem[0]
  mem[1]=(mem[1]%mem[0])
  mem[0]=(mem[0]-(mem[1]%mem[0]))
  IfJump mem[[1]], @0
  Sleep (mem[0]/mem[0])
  IfJump mem[2], @0
  mem[[2]]=(mem[0]/(0/mem[0]))
  Goto @0
```

```
@0:
  mem[0]=mem[0]
  mem[1]=(mem[1]%mem[0])
  mem[0]=(mem[0]-mem[1])
  Goto @0
```

```
@0:
  mem[[1]]=811
  Goto @0
```

Fig. 8. The evolutionary progress of the GCD example.

4.2 The Maximum Example

As specified in 3.5, each automaton will be initialized knowing one unique, random number in its first memory cell. After running a network simulation (using the Network Simulation Layer) for some time, each automaton should know the maximum of all these unique numbers. This strategy equals a simple election algorithm. The fitness function will first simulate the system of the automata and then compare the numbers the automata have stored in their first memory cell afterwards. The fitness function will be the sum of all fitness values that the single automata have achieved. Non-zero fitness values will be assigned to an automaton only if it has stored a valid number - that is one which was used for initializing another automaton. The fitness values assigned are proportional to the size of this number. By doing so, we allow also intermediate results to be incorporated, which has turned out to be a good strategy.

For this experiment we ran the evolution process using standard PCs connected via a 100 MBit/s LAN and measured the throughput (Fig. 9). The network connection is the bottleneck in the test, restricting the performance to only being something between doubled and tripled in comparison to only a single computer. To maximize throughput in systems of only a few computers we recommend running the client simultaneously with a server on one machine. This server will then be the fastest since it has no network latency. For networks of many computers, we expect the performance gain by this approach being nullified by the resulting slowdown of other processes on the system, including the load balancer module of the client.

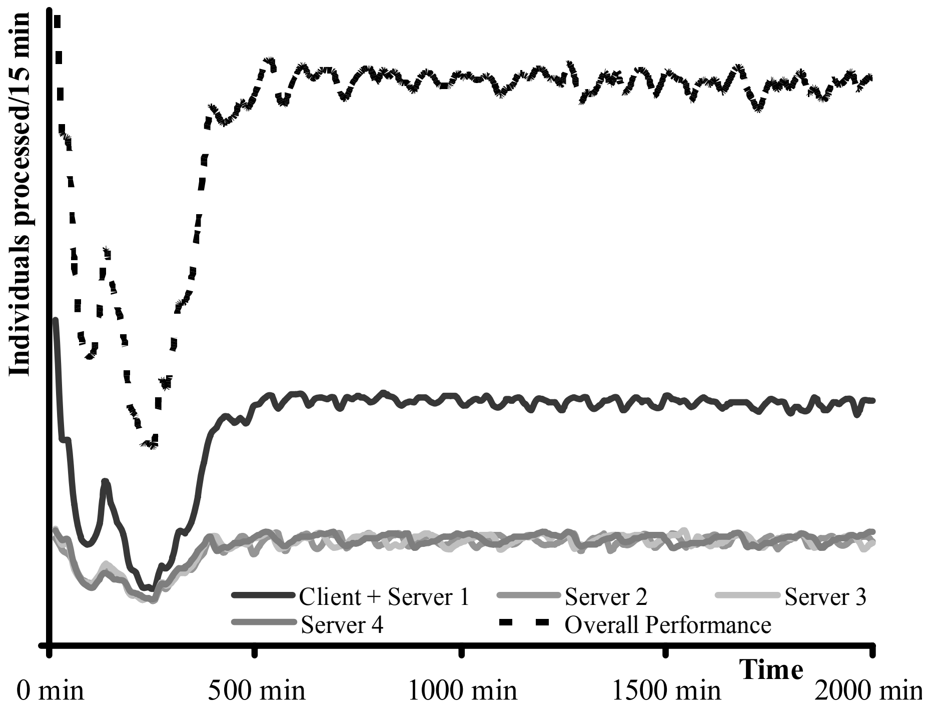


Fig. 9. The throughput of the maximum example.

As in the previous example, at first very long programs will be created, containing only some small segments of useful code. Since we set the runtime granted to a program for testing proportional to its size, this leads to a performance breakdown at the beginning. After better solutions have been found, these are fine-tuned during the rest of the time leading to almost constant throughput. At the end of the evolution, solutions like the code illustrated in Fig. 6 were found.

4.3 Experiment Summary

Our Genetic Programming system found the solution for the Euclidean Algorithm for calculating the GCD and a near-to optimal solution for determining the node with the maximum id, as shown in Fig. 6. To find an optimal automaton model, we have tested three different program code architectures. The current assembler-like language has replaced the previous high-level-language for the sake of faster interpretation and more efficient genetic operations. The first model was basically a non-Turing complete version of the current architecture.

We have also accumulated findings about mechanisms special to the evolution of distributed systems that will help us in further experiments. For example, some techniques have been developed to steer the evolution and increase the individual diversity, if the evolution gets stuck on a local optimum.

5 Related Work

The main advantages of our framework compared to other existing Genetic Programming or Genetic Algorithm libraries like GAUL [15], PMDGP [16], and JAGA [17] are the high scalability reached by the distribution of evaluation and reproduction (Fig. 3), the platform-independence allowing heterogeneous networks to collaborate, and a user-defined, reusable simulation environment. Most Genetic Programming frameworks which are based on interpreting use non-interruptible interpreters. This denies intermediate memory introspection, (see section 3.3) and was one of the reasons why we did not use ECJ [18], which is, in our opinion, the most versatile and efficient open source Genetic Programming framework currently available.

Another issue distinguishing our approach from many others is that we do not use genomes. Instead we apply mutation and crossover directly to program code, allowing statistic knowledge to influence individual altering. If a binary operation has been selected for mutation, an “addition” instance for example, it will more likely be changed to a “binary or” than to a “xor” operation, since “binary or” and “addition” are more similar than “addition” and “xor” in terms of the likeness of the produced results of the operations. This leads to more efficient genetic operations.

The built-in Network Simulator is structured simpler than pure simulation applications like SENSE [19], ns [23], ATEMU [22], J-Sim [20], and GloMoSim [21]. We do not put focus on simulating communications physically in a totally exact way, but reflect its characteristics by a stochastic model which is close enough to reality. Therefore, the performance is much higher than it would be when integrating one of these simulators.

6 Conclusions and Future Work

In this article, we have presented our Genetic Programming framework for sensor programming as well as first experimental results. The primary goal of our framework is to develop implementations of distributed programs running on sensor nodes, using Genetic Programming. The specific characteristics of sensor nodes like running autonomously, asynchronously, and without direct connection, has been explicitly modeled in our simulator. Our framework employs an assembler-like language which is the base for the evolution of the distributed programs.

Currently, we are pursuing three short and medium term goals:

1. We pay special attention to determining the scalability limits of the Task Distribution System used in our Genetic Programming Framework. Therefore we run tests on clusters and, hopefully, in grid environments.
2. As already mentioned in section 3.2, research on parameter adaptation strategies for Genetic Programming will be performed in spin-off projects. Studies show that the efficiency of an adaptation strategy is closely linked to the problem domain: methods that suit for numerical regression might fail completely for Genetic Programming and vice versa.
3. A library of fitness functions for several algorithms for sensor networks will be built and maintained. Multi-objective Genetic Algorithms, such as SPEA2 [25], will be investigated for their utility for super-positioning fitness functions in our System.

The ultimate goal of the project is to develop a methodology for automated sensor network software design employing Genetic Programming including a layered tool chain. The software designer will model the desired program by selecting predefined functional and non-functional attributes. Limits of the hardware, such as memory restricted to 100 words, can be composed with functional requirements, such as routing sensor data to a special node in the network. Using this model, a fitness function will be derived. With that fitness function, suitable programs can be grown which are automatically transformed to assembler- or high level language code and compiled to the target platform.

References

[1] Chee-Yee Chong; Kumar, S.P. Proc, "Sensor networks: Evolution, opportunities, and challenges", IEEE, August 2003

[2] J. David Schaffer, “Multiple Objective Optimization with Vector Evaluated Genetic Algorithms”, PhD thesis, Vanderbilt University, 1984

[3] Architecture Board ORMSC, “Model Driven Architecture (MDA)”, 2001, document number ormsc/2001-07-01, http://www.omg.org/mda/

[4] Benini, Ernesto and Andrea Toffolo. "Optimal design of horizontal-axis wind turbines using blade-element theory and evolutionary computation." Journal of Solar Energy Engineering, vol.124, no.4, p.357-363 (November 2002)

[5] Rizki, Mateen, Michael Zmuda and Louis Tamburino. "Evolving pattern recognition systems." IEEE Transactions on Evolutionary Computation, vol.6, no.6, p.594-609 (December 2002)

[6] Koza, John R., “Genetic Programming - On the Programming of Computers by Means of Natural Selection”, The MIT Press, Massachusetts Institute of Technology (1992)

[7] D. Whitley, A genetic algorithm tutorial, Tech. Rep. CS-93-103, Department of Computer Science, Colorado State University, Fort Collins, CO 8052, March 1993

[8] Hyper-Threading Technology , Intel, http://www.intel.com/technology/hyperthread/

[9] Spector, L., and J. Klein. 2005. "Trivial Geography in Genetic Programming" in Genetic Programming Theory and Practice III, edited by T. Yu, R.L. Riolo, and B. Worzel, pp. 109–124. Boston, MA: Kluwer Academic Publishers.

[10] W.N. Martin, J. Lienig and J. P. Cohoon (1997), "Island (migration) models: evolutionary algorithms based on punctuated equilibria", in T. Back, D.B. Fogel, Z. Michalewicz (eds.), Handbook of evolutionary Computation. IOP Publishing and Oxford University Press.

[11] C. C. Pettey (1997), "Diffusion (cellular) models", in T. Back, D.B. Fogel, Z. Michalewicz (eds.), Handbook of evolutionary Computation. IOP Publishing and Oxford University Press.

[12] Astro Teller, "Turing completeness in the language of genetic programming with indexed memory", Proceedings of the 1994 {IEEE} World Congress on Computational Intelligence Volume 1, IEEE Press, 1994

[13] Woodward, John R., "Evolving turing complete representations", In Congress on Evolutionary Computation, Cake talk at Birmingham 11th August 2003

[14] Farinaz Koushanfar, Miodrag Potkonjak, Alberto Sangiovanni-Vincentelli, "Fault Tolerance in Wireless Sensor Networks", http://www-cad.eecs.berkeley.edu/~farinaz/Papers/chapter-FT_04.pdf

[15] S. Adcock, "GAUL, the Genetic Algorithm Utility Library", 2004, http://gaul.sourceforge.net/

[16] Meulen, P.G.M. van der, H. Schipper, A.M. Bazen and S.H. Gerez, "PMDGP: A Distributed Object-Oriented Genetic Programming Environment", 7th Annual Conference of the Advanced School for Computing and Imaging, Heijen, The Netherlands, (2001)

[17] Paperin, Greg, "JAGA - Java API for Genetic Algorithms", 2004, http://www.sourceforge.org

[18] Luke, Sean, "ECJ: A Java-based evolutionary computation and genetic programming system", 2000, http://cs.gmu.edu/~eclab/projects/ecj/

[19] Chen G., J. Branch, M. J. Pflug, L. Zhu and B. Szymanski (2004). "SENSE: A Sensor Network Simulator". Advances in Pervasive Computing and Networking. B. Szymanksi and B. Yener, Springer: 249-267

[20] Hung-ying Tyan, "Design, Realization, and Evaluation of a Componen-Based Compositional Software Architecture for Network Simulation", Dissertation, 2002, The Ohio State University

[21] Xiang Zeng, Rajive Bagrodia, Mario Gerla, "GloMoSim: a library for parallel simulation of large-scale wireless networks", Proceedings of the twelfth workshop on Parallel and distributed simulation, 1998

[22] Jonathan Polley, Dionysys Blazakis, Jonathan McGee, Dan Rusk, John S. Baras: "ATEMU: A Fine-grained Sensor Network Simulator", Proceedings of First IEEE International Conference on Sensor and Ad Hoc Communication Networks (SECON'04), Santa Clara, CA, October 2004

[23] “The Network Simulator – ns”, http://nsnam.isi.edu/nsnam/index.php/Main_Page

[24] Lidia Yamamoto, Christian Tschudin, "Genetic Evolution of Protocol Implementations and Configurations", IFIP/IEEE International workshop on Self-Managed Systems and Services (SelfMan 2005), Nice, France

[25] Zitzler, Laumanns, Thiele, "SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization", Evolutionary Methods for Design, Optimization and Control, CIMNE, Barcelona, Spain 2002