MASTER THESIS

HARDWARE EVOLUTION

Thesis author:
Magnus Ekman

Examiner:
Peter Nordin
Department of Physical Resource Theory
Chalmers University of Technology

Tutor:
Tryggve Mathiesen, TietoEnator Embedded Tech AB
ACKNOWLEDGEMENT

Firstly, I would like to thank my tutor at TietoEnator Embedded Tech AB, Tryggve Mathiesen.

Secondly, I would like to thank my examiner at Chalmers University of Technology, Peter Nordin.

Thirdly, I would like to thank Andreas Magnusson for attending my presentation of the report.

Finally, I would like to thank Markus Bengtsson and David Olsson for reviewing this report.

Gothenburg, 2000-10-19.

Magnus Ekman
ABSTRACT

This master thesis deals with some of the possibilities and problems using FPGAs (Field Programmable Gate Array) for evolutionary computation. One reason why one wants to use an FPGA when evolving hardware is the gain in time. Since the FPGA works in real time (intrinsic evolution) instead of simulating the design, the test phase should go much faster. However, depending on the process that is used to perform the evolution, the set up time for a test often takes so much time that there is no actual gain in time. Another reason why an FPGA is suitable for hardware evolution is that there is no need to build a model of the environment. This grants that the construction will work in implementation, which is not always the case after simulation.

With the use of hardware description languages, the line between hardware evolution and software evolution is not quite clear. Software algorithms may be synthesised and implemented in silicon, and hardware devices may be simulated in software. This master thesis works with both software and hardware evolution, but the target device is always an FPGA. Two programs have been written; one that evolves VHDL-code (Very High Speed Integrated Circuit Hardware Description Language) that describes a hardware configuration, and one that evolves assembler code for a processor that can be synthesised in an FPGA. These programs have been used to evolve a configuration for the FPGA, which can control a small LEGO car. The thesis also suggests a general test structure that can be implemented in an FPGA. This test structure can evolve hardware at a high abstraction level, to be tested in real time, without placing and routing the construction.
CONTENTS

1 INTRODUCTION

1.1 BACKGROUND
1.1.1 Configuration strings
1.1.2 Hardware description language
1.1.3 Other work
1.2 Objectives and limitations

2 VHDLGP

2.1 Structure of VHDLGP
2.1.1 Fitness function and system environment
2.1.2 Construction library
2.1.3 VHDL simulator
2.2 Evolved system
2.2.1 Genetic operators
2.3 Settings for the evolution

3 SOFTGP

3.1 Structure of SoftGP
3.1.1 Fitness functions and system environment
3.1.2 Connection FIFOs
3.2 Evolved code
3.2.1 Genetic operators
3.3 Settings for the evolution
3.4 FPGA board and PC-interface
3.4.1 PC-interface

4 DEMO PRODUCT

4.1 Hardware platform

5 TEST RESULTS

5.1 Program behaviour
5.1.1 VHDLGP
5.1.2 SOFTGP
5.2 Recombination properties
5.3 A small test problem
5.4 Evolution of Demo product
5.4.1 IR-detector
5.4.2 Motor controller
<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 POSSIBLE IMPROVEMENTS</td>
<td>30</td>
</tr>
<tr>
<td>7 CONCLUSION</td>
<td>32</td>
</tr>
<tr>
<td>8 REFERENCES</td>
<td>33</td>
</tr>
<tr>
<td>APPENDIX A – REQUIREMENT SPECIFICATIONS</td>
<td>34</td>
</tr>
<tr>
<td>REQUIREMENTS FOR VHDLGP</td>
<td>34</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>34</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>34</td>
</tr>
<tr>
<td>EVOLVED SYSTEM</td>
<td>34</td>
</tr>
<tr>
<td>GENETIC OPERATORS</td>
<td>35</td>
</tr>
<tr>
<td>REQUIREMENTS FOR SOFTGP</td>
<td>36</td>
</tr>
<tr>
<td>STRUCTURE OF SOFTGP</td>
<td>36</td>
</tr>
<tr>
<td>CONNECTION FIFOs</td>
<td>36</td>
</tr>
<tr>
<td>EVOLVED CODE</td>
<td>36</td>
</tr>
<tr>
<td>GENETIC OPERATORS</td>
<td>37</td>
</tr>
<tr>
<td>EVOLUTION SETTINGS</td>
<td>37</td>
</tr>
<tr>
<td>REQUIREMENTS FOR HARDWARE</td>
<td>39</td>
</tr>
<tr>
<td>HARDWARE PLATFORM</td>
<td>39</td>
</tr>
<tr>
<td>DEMO PRODUCT</td>
<td>39</td>
</tr>
<tr>
<td>APPENDIX B – USER’S GUIDES</td>
<td>40</td>
</tr>
<tr>
<td>USING VHDLGP</td>
<td>40</td>
</tr>
<tr>
<td>NEW SYSTEM / EDIT SYSTEM DIALOGUE BOX</td>
<td>40</td>
</tr>
<tr>
<td>MAIN VIEW</td>
<td>41</td>
</tr>
<tr>
<td>INDIVIDUAL INFORMATION</td>
<td>43</td>
</tr>
<tr>
<td>MENUS</td>
<td>43</td>
</tr>
<tr>
<td>USING SOFTGP</td>
<td>45</td>
</tr>
<tr>
<td>NEW SYSTEM / EDIT SYSTEM DIALOGUE BOX</td>
<td>45</td>
</tr>
<tr>
<td>MAIN VIEW</td>
<td>46</td>
</tr>
<tr>
<td>INDIVIDUAL INFORMATION</td>
<td>48</td>
</tr>
<tr>
<td>MENUS</td>
<td>48</td>
</tr>
<tr>
<td>APPENDIX C – FILE FORMATS</td>
<td>50</td>
</tr>
<tr>
<td>FILE FORMATS FOR VHDLGP</td>
<td>50</td>
</tr>
<tr>
<td>FILE FORMAT FOR SOFTGP</td>
<td>51</td>
</tr>
</tbody>
</table>
APPENDIX D – PROGRAM DOCUMENTATION

DOCUMENTATION FOR VHDLGP
DOCUMENT STRUCTURE 52
INDIVIDUAL 52
VIEWS 52
CLASSES 53

DOCUMENTATION OF SOFTGP
DOCUMENT STRUCTURE 61
INDIVIDUAL 61
VIEWS 61
CLASSES 61
1 INTRODUCTION

The field of evolutionary computation is steadily growing, and hardware evolution is one part of it. The rapid growth in size of FPGAs (Field Programmable Gate Arrays) has contributed with interesting possibilities for hardware evolution, especially the possibility to perform intrinsic hardware evolution instead of just simulations of the designs.

1.1 Background

There are various methods available to evolve hardware. Work has been done both in the digital and the analogue field of electronics. The analogue work has mostly been simulated, typically with PSpice, but analogue FPGAs now exist and this may present new possibilities for intrinsic evolution of analogue circuits. The work on digital hardware evolution has been performed both intrinsically in FPGAs and in simulation. The two main ways to evolve hardware configurations for an FPGA are to manipulate the configuration strings and to evolve code in a hardware description language. Other work that does not include FPGAs has also been done.

1.1.1 Configuration strings

An FPGA consists of a huge amount of CLBs (configurable logic blocks). These can be configured to work in different ways and they can also be connected to each other to make more complex logic blocks. A configuration bit string that is loaded into the FPGA on startup, controls the ways in which these blocks are configured and interconnected. By modifying this bit string it is possible to alter the function of the FPGA. A genetic algorithm seems like an obvious choice to do this.

Adrian Thompson is one of the pioneers in this field [1]. He wanted to explore if evolution could make use of physical properties of the silicon chip that an engineer would not use. An example of this is the experiment where he evolved a circuit that can distinguish between a signal with a frequency of 1kHz and one with a frequency of 10 kHz that was evolved in an FPGA. To see if it was possible to evolve a design completely different than a design designed in a conventional way, he did not supply a clock signal to the circuit. The result was interesting. The evolved constructions were much smaller than constructions developed in a traditional way. The reason for this was that he did not configure the FPGA as a synchronous construction as it is usually configured. Instead he made an asynchronous design, which is very dependent on gate propagation delays, and analogue properties of the digital gates gave the construction its properties. However, there were some problems with the robustness of the constructions. The evolved constructions worked in some chips but not in others, even though they were of the same type. The temperature range in which the constructions could be used was also limited. By modifying the fitness function to also consider temperature and robustness, he successfully evolved more robust constructions [2].

The FPGA that Adrian Thompson used is an FPGA where the configuration strings are public, giving him full control of what he was doing, and letting him introduce some constraints that he wanted. One example is that he only let neighbouring cells connect to each other. One major problem with manipulation the configuration strings is that for most FPGAs the configuration strings are not public. This means that it is hard to know how to perform the evolution. It is important that a child created by two
parents in the population gets some properties from both of the parents. If one does not know how to mix two parents in a proper way the result can be completely random. Another problem is that the newer, bigger FPGAs are more sensitive to configurations that they are not constructed for. An invalid configuration string can easily destroy a chip. Delon Levi has presented a way to solve this problem [3]. He works at Xilinx, one of the major FPGA suppliers, and therefore has inside information about the configuration strings. With his program GeneticFPGA it is possible to create valid configuration strings without knowing the architecture of the chip.

Another problem with evolution of the configuration strings is that this method does not scale very well to more complex problems [4]. To do this, the method needs to work at a higher abstraction level.

1.1.2 Hardware description language

The other major way to evolve hardware is to work at a higher abstraction level, such as a hardware description language. Typical languages that are used are VHDL and Verilog. These languages are hardware independent, and there is therefore no problem to create a valid hardware configuration. The only problem is that the source code first needs to be compiled, and after that the netlist from the compiler needs to be placed and routed before the configuration can be downloaded to the chip. These two steps may take a while, especially if the construction is big. The result of this is often that it is faster to simulate the design than testing it intrinsically.

Montana, Popp, Iyer and Vidaver [4] have done hardware evolution with the help of a hardware description language. They have tried to evolve hardware sorting algorithms and an edge-detecting algorithm to be used in image processing. Their approach was to perform standard GP on a parse tree, which could be translated to VHDL-code. They have tried various ways to speed up the learning process. The most successful one seems to be to have most of the primitives used in the GP already downloaded on the FPGA and do the testing on the same computer running the GP-system, but with the help of these primitives on the FPGA. Primitives that are computationally heavy then takes less time to evaluate than to simulate, in spite of the overhead with downloading data to the FPGA.

1.1.3 Other work

Another experiment done by Adrian Thompson was to evolve a robot controller, which could control a robot with two motors without hitting the walls of the room, with the help of two sonars. He wanted to evolve unconventional designs in this experiment too, but this time he let a clock signal exist as a resource that the evolution could use if it needed to. This experiment was performed on customized hardware, constructed for the experiment giving the construction a predefined structure. The robot had two motors, each driving one wheel. The construction was actually split into one controller for each wheel with the two sonars as inputs. To let the evolution make an unconventional design he did not force the machine to be synchronous. Instead he made it possible for the construction to be asynchronous, synchronous or a mix between them. The controllers were based on a Boolean function implemented in RAM. There were feedback paths from the outputs to the inputs through registers that could be bypassed with the help of multiplexers. This structure makes it possible to construct both synchronous and asynchronous state machines, as well as ordinary combinational nets. The controlling signals to the multiplexers were also stored in registers and the configuration of the device could be represented by a bit stream. The
experiment was successful and resulted in a mixed circuit with both synchronous and asynchronous properties that worked well. [1]

1.2 Objectives and limitations

The purpose of this master thesis is to investigate how genetic algorithms and genetic programming can be used to evolve electronic constructions based on an FPGA. The constructions may use a processor, described in VHDL-code, which can be synthesised in an FPGA. The GP-systems that will be built shall be general problem solvers, and not specialised for a certain problem. The evolution shall be performed at a rather high abstraction level. Most of the parameters that control the evolution shall be adjustable. Further, a demo product that is based on a construction evolved by the systems shall be built.

Since the evolution shall be at a high abstraction level, evolution of configuration strings for the FPGA will not be implemented. The methods that will be implemented are evolution of VHDL-code and evolution of programs for one or several VHDL-processors. These systems are called VHDLGP and SoftGP. The fitness function will be kept outside the GP-systems to be able to solve a variety of problems.

The demo product shall be a LEGO car, which is controlled by a construction evolved by the GP-systems. The car shall have a sensor that can detect obstacles in front of the car. The evolved construction shall drive the car without hitting these obstacles.

The ways of hardware evolution used in these experiments shall be evaluated and possible improvements and further work on these methods shall be discussed.
2 VHDLGP

VHDLGP evolves a construction in VHDL code. The evolution is done on a PC and the VHDL code is then simulated. The reason that no intrinsic evolution is performed is that tests of the available compiling tools and place- and route tools showed that this would take more time than simulating the design. When a suitable solution for a problem is evolved it can be placed and routed, and then downloaded to an FPGA.

2.1 Structure of VHDLGP

The structure of VHDLGP can be found below.

![VHDLGP Structure Diagram]

The user configures the settings for the evolution in VHDLGP. The user also specifies a system environment and stimuli that is the interface to the rest of the world. The system environment also contains the fitness function. Keeping the system environment and fitness function outside the program makes VHDLGP more general, enabling it to solve a wide range of problems.

VHDLGP evolves individuals and creates an entity (an independent unit in VHDL) that contains both the evolved individual and the system environment, including the fitness function. The VHDL simulator tool then simulates this and the fitness score is sent back to VHDLGP. The evolved individuals can be stored in a functional unit library.

2.1.1 Fitness function and system environment

The system environment and fitness function are modelled in VHDL-code since this is probably how it will be represented in the implementation once the evolution is finished. The world outside the FPGA is modelled as stimuli to the FPGA pins. It might be hard to write a good fitness function in VHDL, but it is possible to include a processor that will be synthesised in the FPGA. This makes it possible to create the fitness function in software instead. The format of the file with the system environment, fitness function and the file with the stimuli can be found in Appendix C.

2.1.2 Construction library

The constructions that are evolved are saved in the construction library. The constructions are described as entities and architectures in VHDL.
2.1.3 VHDL simulator

The VHDL simulator ModelSim PE or ModelSim XE is used to test the individuals. It runs the evolved system together with the system environment. The included fitness function computes a score and the VHDL simulator returns this to VHDLGP.

2.2 Evolved system

The approach used to evolve VHDL-code does not deal with parse trees, but a custom structure of hardware. This structure consists of several state machines of various sizes that co-operate to solve the problem. The reason to have several machines instead of one big complicated one is that no appropriate way to mix two machines and still keeping the properties of both were found. The structure presented below seemed like a promising choice.

![Diagram of the evolved system]

It is built from several state machines that all are synchronous. They will only be of the type Moore or pure Moore machines. However, the combinatorial net on the output makes Mealy structures possible. It is also possible that some of the machines are missing. Every machine, as well as the input to the system, is assigned 16 bits on the bus to write to. If the state machine does not have so many outputs the bus pin will be assigned a logical zero. The inputs to the machines are also connected to the bus but they are not assigned to predetermined pins. Depending on the settings they can be connected to any pins on the bus or just bits located close to the output pins of the machine. The combinatorial net on the output can be connected to any of the bits on the bus. The output from the structure can be any of the bits on the bus or any of the outputs from the combinatorial net.
With this structure each machine will get its own duty to fulfil, i.e. to present good outputs on its 16 bits of the bus. The machine can do this with the help of outputs from other machines, inputs to the structure and different state transitions.

### 2.2.1 Genetic operators

The genetic operators that are used are crossovers, mutations and gene duplications. The crossovers in this structure mean that state machines are exchanged between two individuals and the combinatorial net is exchanged between two individuals. Each state machine is assigned its bits on the bus and this is not altered. This is necessary to keep some properties of the parents. The combinatorial net on the output is either selected from one of the parents or it is a mixture of the two combinatorial nets from the parents. The mixture of the combinatorial net is done by taking some rows from the truth table of the net from the first individual and some rows from the second individual. A picture of the crossover is shown below.
There are five different mutations. They are
- Change in a state-transition.
- Introduction a new state-transition.
- Deletion a state-transition.
- Change of output in a state.
- Change of combinatorial net on the output.

The only gene duplication duplicates a state and lets half of the transitions to the old state be redirected to this new state.

### 2.3 Settings for the evolution

The evolution process is depending on several different configurable factors. The factors that the system lets the user configure are described below.

- Genetic algorithms - There are two different genetic algorithms to choose among. The first one is the steady state algorithm “30 monkeys in a bus”. The algorithm keeps a population or, mathematically speaking, a set of individuals. A subset of four individuals are randomly selected and sorted according to their fitness values. The two worst individuals are removed from the set. The other two are combined with the help of genetic operators to generate two new individuals that are inserted into the set. The main advantage of this algorithm is that the random selection of individuals to test, makes it possible to keep bad individuals and thereby still have diversity in the population. However it is also possible that it does not improve so fast as an algorithm that chooses the best individual in the entire population would do.
The other algorithm is a traditional genetic algorithm [5] that tests all the individuals and then selects parents randomly but with higher probability if the individual has good fitness. These parents create a completely new population. This algorithm does not use elitism [5], i.e. keep good individuals to the next generation.

Evaluate all individuals to determine their fitness.

Copy individuals according to their fitness into a mating pool (higher fitness = more copies of an individual)

Randomly select parents from the mating pool and generate offspring until the new generation is full.

The other configurable factors are:
- Mutation frequency
- Gene duplication frequency
- Population size
- Individual size - Maximum number of state machines and if the number of state machines is fixed or dynamic.
- Connection Range - Information about whether the state machines shall be able to connect the inputs to all the other machines or mostly to neighbouring machines.
3 SOFTGP

SOFTGP evolves assembler code for one or more VHDL-processors that can be synthesised in an FPGA. SOFTGP is divided into one program for a VHDL-processor and one programme for a PC. The VHDL-processor serves as a gateway that can download and upload information between the PC and the other VHDL-processors. The PC runs the GP-system and also serves as a configuration and diagnosis tool. SOFTGP is an ordinary GP-system for software except that it works with a multi-processor system. It is also adapted to work with small processors with a small memory.

3.1 Structure of SoftGP

The structure of SoftGP is found below.

The user configures the system to be evolved with the PC-programme. The things to decide are the number of VHDL-processors and how many ports every VHDL-processor will have. The user also specifies a system environment that is the interface to the rest of the world. This system environment can include functional units developed by VHDLGP. The system environment also contains fitness functions or interfaces to fitness functions outside the FPGA. Keeping the system environment and fitness function outside the program makes SoftGP more general, enabling it to solve a wide range of problems.

The program creates a VHDL file from this that can be compiled, placed and routed, and then downloaded to the FPGA where different programs can be tested. The user also specifies the settings for the evolution. These settings can be altered during the evolution process.

It is possible to load and save the evolved programs or entire populations. This makes it possible to add or remove a VHDL-processor or alter the system environment without having to start the evolution from scratch.

3.1.1 Fitness functions and System Environment

The fitness functions and system environment are the interface to, and often a part of, the problem to be solved. The fitness functions and system environment are described as an entity in VHDL. This entity can contain VHDL-processors as well as constructions evolved by VHDLGP to make it as flexible as possible. Since the GP-system evolves programs for several processors and connections between them,
each processor needs a fitness function. The format of the file with the system environment and fitness functions can be found in Appendix C.

3.1.2 Connection fifos

Every VHDL-processor will be connected to every other VHDL-processor with two fifos, one for output and one for input. This makes it possible for the processors to communicate with each other and co-operate.

3.2 Evolved code

When this project started there was no compiler available for the VHDL-processors that are used. Hence, the evolution does not work with parse trees. Another reason for this is that the program memory is very limited (256 instructions). High level languages tend to produce longer programs than assembler language so that is another reason to keep the evolution at the assembler level. One disadvantage with genetic programming at the assembler level is that it might take a long time to evolve some useful constructs that would be terminal symbols in a high level language. Examples of this are a while-loop and if-statements. However, the time of invoking a compiler is saved which speeds up the testing process and may compensate for the speed loss. The approach in this project is somewhere in the middle. The GP-system works with program constructs that are often used when writing a program, but the program will still be seen as a linear sequence of assembler instructions. The program constructs will serve as the smallest parts that will be exchanged when two individuals are mixed. An example of a short program and description of the program constructs are found below.

```
START:
LOAD s9, s8
SUB s8, 74
OR s8, 117
AND s0, s8
CALL NZ, SUBROUTINE0
For(s8 = 0; s8 < 80; s8++)
RR s8
OR sa, 246
CALL C, SUBROUTINE2
AND s8, 72
OR s5, sa
End For
If(s3 < s0)
SL0 s7
OR s3, s3
AND s8, 87
SUBCY s3, 191
SLX s3
Else
CALL Z, SUBROUTINE0
CALL C, SUBROUTINE1
XOR s3, 95
SRX s0
CALL NC, SUBROUTINE2
End If
```
JMP START

SUBROUTINE0:
ADDCY s9, 134
SUBCY sa, 114
ADD sb, s8
SRX s9
AND sa, s9
RETURN

- **Sequence** - A sequence is a various number of instructions after each other.
- **If-statement** - The if statement chooses among two alternatives. The elseif construct is not supported but it is possible that the system combines several if-statements to form an elseif construct. The first or the second part of the if statement may be empty.
- **Iteration** - The system can do two kinds of iterations, for-loops and while-loops. Nested loops are permitted but just to a limited depth.
- **Subroutine** - It is possible to create subroutines. Since the stack is limited, recursive calls are not allowed.

### 3.2.1 Genetic operators

The genetic operators that are used are crossovers, mutations and gene duplications. There are two kinds of crossovers. One works at the system level (all processors together) and tries to find the best combination of the existing programs. Every processor has its own population and even if one program is very good it can be misjudged if the programs in the other processors are really bad and impossible to co-operate with. By looking upon one program from each processor together as one individual a crossover at the system level is to change places of some of the programs without altering them (see figure).

![Diagram of crossovers between processors](image)

**A crossover between individual 1 and individual 6 on the system level**

The other crossover works at the program level where a single program is thought of as an individual. With this crossover program constructs are exchanged between two programs. A construct will never be split into two halves. One important question is what one should do with the subroutines. When two
programs are mixed the created program will probably have calls to subroutines from both of the parents. The easiest way to solve this would simply be to copy the subroutines from both of the parents. However, this would mean that the number of subroutines would grow every generation. This is not possible due to the limited program memory of the used processor. To avoid this, some of the subroutines are deleted and the calls to them are redirected to other subroutines. This could cause the evolution to be unstable but it is hardly more different than exchanging a program construct in the middle of the program

A crossover on the program level.

The mutations that are used are:
- An instruction is inserted/removed.
- A construct is inserted/removed.
- An instruction is changed to another instruction.
- An argument to an instruction is altered.
- The number of iterations in a for-loop is altered.
- The iteration condition in a while-loop is altered.
- Removing a piece of code and inserting this into the subroutine creates a subroutine. A call to the subroutine is inserted where the code was located.

The gene duplications that are used are:
- A subroutine is duplicated and altered and half of the calls are redirected to the new subroutine.
- A construct is duplicated and altered.

3.3 Settings for the evolution

The evolution process is depending on several different configurable factors that can be altered from the PC-program. The factors that the system will let the user configure are described below.

- Program/system - The GP-system keeps one population of programs for every processor. It both tries to improve the programs and tries to combine programs in different processors to let them co-
operate at the system level. It shall be possible to choose how much the individual programs shall evolve and how much the programmes shall be tested together with other programs.

- Genetic algorithms – The same algorithms that are used in VHDLGP are used in SoftGP. The only modification is that they sometimes work at the program level and sometimes at the system level.
- Mutation frequency
- Gene duplication frequency
- Population size
- Individual size - The size of the initial individuals. Information about whether the size is fixed or dynamic.

3.4 FPGA board and PC-interface

The board that is used to run the FPGA part of SoftGP is a Xilinx Prototype Platform for Virtex Series FPGAs equipped with a Xilinx XCV100PQ240. It is connected to the printer port of a PC. The structure of the VHDL-entity that is created from SoftGP when the user has chosen two processors is shown below.
In the figure above, the address decoding and read/write signals have been left out. Input/output registers for the processors as well as the address decoding for them have also been left out.

The gateway downloads a program from the PC and writes it into the program memory of one of the processors. Then it gives a reset signal and gets a signal from the system environment when it is done. After this, the gateway can read the fitness for each of the processors from the system environment.

Flow of the gateway program

- Read byte from PC
- Byte = 0xFF
- Read program from PC and write to the processor indicated by the byte read.
- Give reset signal to processors and system environment.
- Wait for done signal from system environment.
- Read byte from PC
- Byte = 0xFF
- Read fitness from the processor indicated by the byte read and write to PC.
3.4.1 PC-interface

The PC-interface connects to the parallel port of a PC. It uses a custom protocol with a write buffer and a read buffer for both the PC and the VHDL-processor. There are also two status registers that can be read to get information about whether the device on the other side has read from, or written to, the interface.
4 DEMO PRODUCT

One idea with this project was to build a demo product that uses the GP-systems. A LEGO car equipped with electric motors and IR-sensors to be able to sense its environment has been built. The duty for the GP-systems was to evolve a controlling unit for this LEGO car.

4.1 Hardware platform

The demo product was built on a pre-manufactured motor controlling board equipped with an FPGA, motor drivers, an A/D-converter and an RS232-interface. To let the car sense its environment an IR-sensor was added. The board needs to be connected to a PC for the learning part but can be run autonomously once the controller is evolved.

There are three motor drivers on the board but just two of them are connected to a motor. Some of the unused pins of the FPGA are connected to an IR-diode and an IR-transistor with the help of some surrounding components. A schematic picture is shown below.

![Diagram of the pre-manufactured motor controlling board]

When the system is learning, the FPGA is configured like the FPGA on the prototype board described in the paragraph “FPGA board and PC-interface”. The system environment pulses the IR-diode with a frequency of 10 kHz.

The construction was mounted on a LEGO car with two motors and four wheels. The two motors were connected to the front wheels and by running one of the motors the car could turn sideways. The IR sensor was mounted in the front of the car and the photo resistor pointing backwards. The front of the car looked a little like a forklift truck but with wheels on the fork that stopped the car from getting stuck if it bumped into the walls. The size of the car is 20x8x10 cm (length, width, height).
Pictures of the LEGO car. Note the fork with wheels and IR-sensor on the upper picture.
5 TEST RESULTS

The tests of the implemented systems were done in four stages. The first stage tested that the programs behaved like they should, the second tested properties of the mutations and crossovers, the third one applied the systems to a small problem to see if they worked and the fourth test aimed at evolving the demo product.

5.1 Program behaviour

The first stage of the tests was to verify that the programs did what they should i.e. produced correct individuals and that the genetic operators worked. Many of these tests were done during implementation by stepping through the code and inspecting the individuals created. In addition to these tests the programs have also been run for 40 hours without stopping to confirm that their operation is stable and that they do not have any memory leaks.

5.1.1 VHDLGP

To check if VHDLGP produced correct individuals, these were compiled by a VHDL-compiler. This is done during normal operation of VHDLGP and no special test had to be carried out.

The genetic operators were tested by inspecting the individuals. The program was a little bit altered to make this test. Instead of applying both mutations and crossovers to generate new individuals, only one mutation or crossover was applied each time. This made it easy to conclude that the genetic operators worked.

To check that the genetic algorithms worked, the program was applied to a very easy problem. The net should output a numeric constant. The fitness function was the difference between this constant and the output of the net. When the program was running, the fitness of the population got better and better which showed that the worst individuals were deleted and that better ones were created.

5.1.2 SoftGP

To check if SoftGP produced correct individuals, these were disassembled. Since it was possible to disassemble the individuals, they represented valid programs. The disassembled code was then inspected to see if it consisted of program constructs like it should.

The tests of the genetic operators and the genetic algorithms were done in the same way as the tests of the genetic operators and the genetic algorithms in VHDLGP.

To test the gateway program, a test program for one of the processors on the board was created. The only thing the test program did was to toggle one of the outputs. This output was connected to a LED. This program was downloaded through the gateway and since the LED started flashing the download routines worked.
5.2 Recombination properties

The purpose of the tests of the recombination properties was to see if the way the evolution is performed is a promising way. It is important that the offspring has properties of both or one of its parents. If it does not have any properties of its parents, a randomly created individual would work as well as the offspring and the recombination part would be useless. This would mean that random tests in the fitness landscape are performed instead of gradually moving to a more fit position in the fitness landscape.

The test of the recombination properties was made in the same way for both VHDLGP and SoftGP. A number of different stimuli were applied to the inputs and the outputs were monitored. For the test to be successful, the output of the offspring should remind of the output of the parents, either as a mixture of the parents or as the output of one parent on some stimuli and the other parent on some other stimuli. Of course the offspring should be allowed to present some other completely different output values but if all of the output values were completely different, one could hardly say that the offspring had properties of its parent. The test showed that the recombination properties worked. Most of the times when two individuals were mixed, the offspring had properties of both of the parents and the evolution should work well.

5.3 A small test problem

One test problem that the GP-systems were applied to, was the problem to evolve a multiplier. The evolved system was specified to have two inputs and one output. Some different numbers were applied to the inputs, and the output was read after a number of clock cycles. The fitness function was the sum of the absolute value of the difference between the output and the product of the inputs.

To award individuals that really used the input values, the different output values were compared to each other. If the output values were identical, the score was set to the worst fitness value. This was not done in the beginning and the result was that individuals with a constant value somewhere close to the average of the tested products were scored with a rather good fitness value, even though they did not do anything at all.

VHDLGP almost solved the problem but it did not evolve a general multiplier. The construction evolved was probably more like a lookup table. This is not very surprising since VHDLGP does not have any information about how binary coded numbers and binary arithmetic work. It is probably a too big step to take to evolve a multiplier from scratch. A more natural way that could work would be to try to evolve a state machine that worked as an adder first, and then try to evolve a multiplier using this state machine.

SoftGP worked better. This is not very surprising, since the processor has instructions for both addition and subtraction. A multiplier is easily constructed using a while loop with an addition and a subtraction. A simple solution is shown below:

```
input1->s0
input2->s1
LOAD s2, 0
while(s1!=0)
    ADD s2, s0
    SUB s1, 1
end while
s2 -> output
```
The solution created by SoftGP was more complex with several instructions that did not do anything. Adding the size of the program to the fitness function would probably make the program smaller. This has not been done but would be a natural improvement to do.

Another test that was done was to evolve a construction that could tell if a 10 kHz signal was present or not. This test is rather similar to the one that Adrian Thompson has done except that his construction could distinguish between two signals of different frequencies. The reason that the test done here just checked for one frequency was that the evolved construction would be used in the demo product.

5.4 Evolution of demo product

The purpose of this test was to try to evolve something interesting that can get people’s attention. Another purpose was to try to evolve something that interacted with the real environment, since this can generate funny and unexpected solutions, that often make use of some property of the environment which a human designer would not think of.

The goal for the system was to evolve a construction that could drive the car, letting it move as much as possible without hitting the walls. The evolution was divided into two stages. The first was to evolve a unit that could detect the 10 kHz signal from the IR-transistor. The unit should present a logic high level on the output if a 10 kHz signal was present and a logic low level otherwise. The second stage was to evolve the motor controller.

5.4.1 IR-detector

The evolution of the IR-detector was not done with the help of the IR-transistor. Instead, a system environment was written that presented a 10 kHz signal with random intervals. This made it possible to evolve the IR-detector with VHDLGP. The reason that the 10 kHz signal was present in random intervals was to avoid that the construction could learn when it should output a high signal without using the input signal. This experiment was a little similar to the experiment done by Adrian Thompson but he had two signals with different frequencies that the construction should be able to distinguish from each other. Another difference was the fitness function. In Thompson’s experiment the fitness function integrated the output signal with an analogue construction. In this experiment the output signal was read by a VHDL-processor at a number of times. The time between readings, was not always the same. The fitness function gave the worst fitness to individuals that did not change the output at all. This was not done in the beginning resulting in that individuals that had a steady output got a better score and therefore eliminated individuals that had been more probable to give a good solution after some evolution.

The evolution was successful. A construction was evolved that made no mistakes at all, but it shall be noticed that the only test was to present a 10 kHz signal or no signal at all. No tests were done with signals of other frequencies but there is no reason to expect any major problems with evolving a construction that can distinguish between different frequencies. One thing should be noticed. The construction did not give a steady one and a steady zero on the output. The output toggled all the time but was adjusted so that the microprocessor that read the signal always got the correct value. The microprocessor caught the data on the positive edge of the clock, and at that time there was always valid
data on the output. This was not a problem in this project but it could probably have been avoided with a different fitness function. Below is a graph of the learning process. The upper line is the average fitness value and the lower line is the best fitness value. The construction was evolved with the steady state algorithm with a population of 16 individuals. The fitness function was adjusted after about 200 replacements, which explains the rise in average fitness after two thirds of the diagram. After about 300 replacements the evolution was stopped since the constructions were as good as possible.

The upper curve is the average fitness and the lower curve is the best fitness.

5.4.2 Motor controller

SoftGP evolved the motor controller. It would have been possible to evolve it with VHDLGP if the car and the environment had been simulated, but it would have been too extensive work to model the environment and properties of the motors in an accurate way. Instead the LEGO car was used to test programs evolved by SoftGP. First the fitness function was not very complicated. It basically gave a good fitness value for each motor that was running and a bad fitness value if the IR-detector detected a signal. The idea was that the car should keep both the motors running but turn when it was about to hit a wall, but this was a little bit too optimistic. An unexpected solution was evolved. The driver circuits to the motors normally just had two choices. Either the motor was running or not. However, if the program turned on and off the motor very fast, it could control the speed of it. This is a common method to regulate the speed of a motor. The program ran one of the motors on full speed while it pulsed the other. The result was that the car drove in a circle without hitting the walls. The fitness function was designed to avoid that the car could drive in a circle by awarding the individuals that drove both the motors. However, since it just checked if the motors were going on discrete occasions, the car managed to pulse it with a frequency that made the fitness function think that both motors were running all the time even though the car was driving in a circle. This is one of those unexpected solutions that a human designer would not think of. The evolved construction did not use the IR-sensor at all. To do this a different approach had to be taken.

The evolved constructions did not show any intelligent behaviour. Every program started the motor with one speed and kept this speed all the time. A solution that used the sensor or showed more intelligent behaviour was not awarded enough so the fitness function had to be modified. The new fitness function
gave good score if the car alternated the motors. It also gave a really good score if the car turned on or off a motor because of a signal from the IR-sensor. A diagram of the learning process is shown below. 160 replacements were required to pass this stage.

![Fitness progress](image.png)

*The upper curve is the average fitness and the lower curve is the best fitness.*

After this stage a great deal of the population reacted instantly to the IR-sensor. Some of them turned, some of them stopped and some of them drove straight ahead when they got an IR-signal. One thing that can be noticed is that most of them turned to the left. The reason for this is probably that in the initial population no individual used the left motor. The resulting individuals had started to use it but not as much as they used the right motor. To come around this problem, the fitness function was changed to one that awarded the individual if it used the left motor more than the right one, but this did not solve the problem. The population was too uniform and did not try to use the left motor more and therefore could not be awarded. Another approach had to be taken. The mutation frequency was increased significantly. This should result in that some individuals started to behave differently compared to the rest of the population. Hopefully, one of them would test to use the left motor more and, with the new fitness function, be able to spread this property to the rest of the population. This did not work completely. A problem was that the processor that executed the fitness function operated at the same frequency as the processor running the program. This made it possible for the program to make the fitness function believe that both motors were running or that just one was running, in spite of the fact that one or both of them were pulsing. A better solution would be to let the fitness function have a higher sample frequency. Another solution would be to have an analogue integrator connected to an A/D-converter.

The programs evolved by the GP-system avoided the walls, but their behaviour did not seem to be very intelligent. They were all going in the same direction with one motor running faster than the other was. They did react on signals from the IR-sensor, however, so they were better than the first try. The curves of the evolution showed that the fitness value became better and better so the GP-system worked. With a more complicated fitness function more interesting behaviour would probably evolve. The two graphs on the next page show the evolution using two different fitness functions. There is also a picture of the LEGO car in action.
6 POSSIBLE IMPROVEMENTS

One major problem with evolution is the time it takes. Time is therefore crucial when testing an individual. There are ways of speeding up the tests in this project. SoftGP would be much faster if the GP-system was running in the FPGA instead of on a PC since the uploading and downloading time is a major part of the test if the test itself is very short. This was not possible in this project due to the limitation of the processor used and the tools for this processor. The processor used is still under construction though, meaning that this should soon be possible.

VHDLGP would be faster if the constructions could be tested in an FPGA instead of simulating the design. However, this is not possible today since the place and route process takes so long time. Another solution would be to create a simulator in the program that only can run the type of constructions that are created by VHDLGP. This would probably be faster than invoking a VHDL-simulator that works at a lower abstraction level. However, this would mean that it would not be possible to change the fitness function as easily, and the program would be less general. One could create some kind of script language to be able to supply the fitness function and stimuli. A specialized simulator could improve the speed but there is a major advantage of using a VHDL-simulator to simulate the design instead of building a custom simulator. The stimuli are often generated by other parts of the construction, which already are described in VHDL. It is convenient to be able to use these in the test of the evolved construction, which is possible in VHDLGP.

A better way of speeding up the tests done by VHDLGP would be to build a structure of reconfigurable state machines in hardware. This could be done by keeping the state transition tables in RAM [1] and connect the inputs to the bus with the help of multiplexers. The combinatorial net on the output could also be coded in a RAM and the outputs from the entire structure could be connected by means of multiplexers. The downloading of the configuration would be fast and the test could be done in real time.

In some problems it is known that some functional units will most certainly help to solve a problem. An example is that an FFT (Fast Fourier Transform) probably will help in sound or image processing. There is no problem to include a predefined unit in a structure like this. All the bits on the bus do not have to be assigned to state machines. One of the state machines can be exchanged for a preconstructed unit. The inputs to the unit can either be assigned to the inputs if it is known that it will work best in this way or the evolution can decide which pins on the bus they should be connected to. There is a risk that these constraints force the GP-system to evolve in a certain way but this is always the case when evolution is performed at a high abstraction level. On the other hand, evolution at a high abstraction level might scale better to bigger problems.

Another thing that could give the GP-system fewer constraints is to let the state machines be Mealy machines. However, the RAM needed would increase exponentially. Each state would need as many output values as there are input combinations. One could compromise and just let some of the machines be Mealy machines. Another way to keep the memory down but still have more flexibility is to let each state have just a few output values that are equally distributed over the input values. A possible block scheme for a test circuit is shown on the next page.
The FPGAs on the market today contain block-RAMs and are quite big. This means that this testing structure probably can be synthesized in an FPGA. In this way there would be no need for a complicated hardware construction and the evolution can take place in the FPGA without any place and routing during the test phase. When a fit solution is found this solution can be placed and routed and downloaded to an FPGA that is smaller than the one used for the evolution, since the final solution is not as general as the test structure. Developing and implementing this test structure seems like a natural way of continuing the work done in this project.
7 CONCLUSION

The goals described in the introduction have been met. Two systems have been implemented and work satisfactorily. The LEGO car is controlled by a construction evolved by the two GP-systems and avoids the walls. The experiments show that it is possible to evolve electronic constructions based on an FPGA, but for this project the FPGA is just the chosen target implementation. It does not speed up the evolution. Even though an FPGA is reconfigurable and therefore seems like a perfect platform to use for the testing part in hardware evolution, this is not always the case. Adrian Thompson [1] and Delon Levi [3] have shown that it is possible to create constructions in this way. However, if one wants to perform the evolution at a higher abstraction level, like in this project, it is doubtful if an FPGA really improves the tests. The experiments done by Montana, Popp, Iyer and Vidaver [4] with EvolvaWare show that there are ways to speed up the testing part some, by using the FPGA but their methods would hardly work with the structure used in this project.

It seems tempting to build a test machine, like Thompson did in the robot controller experiment, since it will speed up the testing part significantly. The FPGA will be used to test the construction, but without the place and route stage, despite the fact that the evolution is performed at a rather high abstraction level. The only disadvantage is that the FPGA used in the testing phase might have to be larger than the target FPGA because of the size of the general test structure. However, this is not unusual in the field of genetic programming. The computer evolving a program is often much more powerful than the target platform. One interesting thing, which can be noticed, is that it is not the reconfigurability of the FPGA that makes it suitable for hardware evolution if it is done this way. The FPGA is only used to implement a general test structure, which could be done with discrete integrated circuits, as well as with an ASIC.
8 REFERENCES

[1] Explorations in Design Space: Unconventional electronics design through artificial evolution, Adrian Thompson

[2] Evolution of Robustness in an Electronics Design
Adrian Thompson, Paul Layzell

Delon Levi


[5] Evolutionary design by computers
Peter J. Bentley

Model Technology

Model Technology
APPENDIX A – REQUIREMENT SPECIFICATIONS

Requirements for VHDLGP

Introduction

VHDLGP shall evolve functional units in VHDL code. The evolution shall be run on a PC and the VHDL code shall be simulated. When a suitable solution for a problem is evolved, it shall be possible to place and route the solution, and then download it to an FPGA. VHDLGP shall only be able to develop synchronous modules. The user shall be able to configure the settings for the evolution in VHDLGP. The user shall also be able to also specify a system environment and stimuli that is the interface to the rest of the world. The system environment shall also contain the fitness function.

Structure

The structure of VHDLGP shall be as shown below.

```
x = System environment, Fitness function
VHDLGP
VHDL + VHDL simulator
Fitness value
VHDL
Func. unit library
```

VHDLGP shall evolve individuals and create an entity that contains both the evolved individual and the system environment, including the fitness function. The VHDL simulator tool shall then simulate this and the fitness score shall be sent back to VHDLGP. It shall be possible to store evolved individuals in a functional unit library.

Evolved system

The evolved system shall have the structure shown below.

```
Input  State machine  State machine  State machine  Comb. net  Output
```

The system evolved shall have the structure shown above. The state machines shall be of the type Moore or pure Moore machines. Every machine, as well as the input to the system, shall be assigned to 16 bits
on the bus that the machine can write to. The inputs to the machines shall be connected to bits on the bus by evolution. The combinatorial net shall be connected to bits on the bus by evolution.

**Genetic operators**
The following genetic operators shall be implemented:

**Cross-overs**
*State machines are exchanged between two individuals.*
*The combinatorial net, or parts of it, is exchanged between two individuals.*

**Mutations**
*Change in a state-transition.*
*Introduce a new state-transition.*
*Delete a state-transition.*
*Change of output in a state.*
*Change the combinatorial net on the output.*

**Gene duplications**
*Duplicate a state machine and replace another machine.*

**Evolution settings**
The following configurable settings shall be implemented:

**Genetic algorithm**
*There shall be two different genetic algorithms to choose among. One steady state algorithm and one classical genetic algorithm.*

**Mutation frequency**
*The frequency in which mutations happen.*

**Gene duplication frequency**
*The frequency in which gene duplications happen.*

**Population size**
*The number of individuals in the population.*

**Individual size**
*The size of the initial individuals. Maximum size of a state machine. Maximum number of state machines.*

**Connection Range**
*Information about if the state machines shall be able to connect to all the other machines or if they mostly will connect to neighbouring machines.*
Requirements for SoftGP

SOFTGP shall evolve assembler code for one or more VHDL-processors. SOFTGP shall be divided into one program for a processor and one programme for a PC. The processor shall serve as a gateway that can download and upload information between the PC and the other processors. The PC shall run the GP-system and also serve as a configuration and diagnosis tool.

Structure of SoftGP

The structure of SoftGP shall be as shown below.

The user shall be able to configure the system to be evolved with the PC-programme. The things to decide are the number of processors and how many ports every processor will have. The user shall also specify a system environment that is the interface to the rest of the world. This system environment can include functional units developed by VHDLG. The system environment shall also contain fitness functions or interfaces to fitness functions outside the FPGA.

The program shall create a VHDL file from this that can be compiled, placed and routed, and then downloaded to the FPGA. It shall be possible to load and save the evolved programs or entire populations so that it will be possible to alter the system without having to start the evolution from scratch.

Connection fifos

Every processor shall be connected to every other processor with two fifos. One for output and one for input.

Evolved code

The evolution shall produce machine code. However, the GP-system shall work at a higher abstraction level. It shall work with program constructs that are exchanged between the individuals. The following constructs shall be implemented:

Sequence

A sequence of instructions.
Selection
If statement. This will only choose among two alternatives. The elseif construct will not be supported but it is possible that the system combines several if-statements to form an elseif construct. The first or the second part of the if statement may be empty.

Iteration
For loop. A loop where the number of times it will be run is decided from start.
While loop. A loop where the number of times it will be run is calculated on the fly.

Subroutine
It will be possible to create subroutines. Since the stack is limited, recursive calls will not be allowed.

Genetic operators
The following genetic operators shall be implemented:

Cross-overs
Parts of the programme is replaced with a part from another individual. Program constructs will be replaced. A construct will never be split into two halves.

Mutations
An instruction is inserted/removed.
A construct is inserted/removed.
An instruction is changed to another instruction.
An argument to an instruction is altered.
The number of iterations in a for loop is altered.
The iteration condition in a while loop is altered.
Removing a piece of code and inserting this into the subroutine creates a subroutine. A call to the subroutine is inserted where the code was located.

Gene duplications
A subroutine is duplicated and altered and half of the calls are redirected to the new subroutine.
A construct is duplicated and altered.

Evolution settings
The following configurable settings shall be implemented:

Program/system
The GP-system shall keep one population of programs for every processor. It shall both try to improve the programs but also try to combine programs in different processors to let them cooperate. It shall be possible to choose how much the individual programs shall evolve and how much the programmes shall be tested together with other programs.
Genetic algorithm
There shall be two different genetic algorithms to choose among. One steady state algorithm and one classical genetic algorithm.

Mutation frequency
The frequency in which mutations happen.

Gene duplication frequency
The frequency in which gene duplications happen.

Population size
The number of individuals in the population.

Individual size
The size of the initial individuals.
Requirements for hardware

The GP-systems shall run on a hardware platform and evolve a construction for a demo product.

Hardware platform

The FPGA-board that the GP-system shall use to test individuals is the Xilinx prototype board, equipped with a Virtex XCV100PQ240 FPGA. It shall also have a connector to be able to communicate with SoftGP on the PC. The parallel port on the PC shall be used.

Demo product

The demo product shall be based on a prefabricated I/O-board equipped with a SpartanII XC2S100 FPGA and motor driver circuits. It shall also be connected to an IR-diode and an IR-transistor with necessary amplifying components.
APPENDIX B – USER’S GUIDES

Using VHDLGP

The three main parts of the program are the dialogue box “New system / Edit system”, the main window of the program and the dialogue box “Individual information”. These three parts and the commands in the menus will be described below.

New system / Edit system dialogue box

This dialogue box is used to create a new evolutionary system of state machines. It can also be used to alter an old system. In this dialogue box the user specifies the system environment and fitness function and the inputs and outputs that shall be connected to the evolutionary system.

System environment and fitness function

This field displays the chosen file containing the system environment and the fitness function. This file should only contain the entity containing the system environment and the fitness function. The “Browse….” button opens a dialogue box where the user can choose a new system environment file.

Inputs/outputs

The drop-down list shows all ports in the loaded entity. The user specifies which of these inputs and output that shall be connected to the evolutionary system. The rest of the inputs and outputs will be left open. They will be the interface to the rest of the world where stimuli are applied and results are read. An input/output is connected to the evolutionary system when the “Add” button is pressed and is disconnected when the “Delete” button is pressed.

Input/output list

This list shows the inputs/outputs that will be connected to the evolutionary system.
Main view

In the left part of the main view the user can alter the settings for the evolution that can be altered during run-time. In the right part of the main view there are information about the progress of the evolutionary system.

**Algorithm**

The algorithm chosen for the evolution.

**Connections to other state machines**

With this slider the user can specify how much the state machines will connect to the neighbouring state machines vs. how much they connect to all of the other state machines.
Type of state machines
The type of state machines – Moore or pure Moore.

Mutation frequency
The frequency with which mutations happen.

Duplication frequency
The frequency with which duplications happen.

Number of state machines
The number of state machines.

Number of states
The number of states and information about if the number will be fixed or dynamic. If dynamic is chosen the size field will be used as an upper limit for the number of states.

Population size
Number of individuals in the system.

Generation
Number of generations tested.

Status
Information about if the evolution is running or not.

Fitness progress
A graph describing the progress of the evolution. The progress for the selected processor is shown in this graph.

Reset graph
Resets the graph. Typically used when the fitness function is altered.

Individual progress
Information about the individuals. The user can double-click on an individual to get more detailed information.

Retest individuals
This check box only effects the steady-state algorithm. If this box is checked, the individuals will be re-tested every time. This check-box should be checked when a fitness function that does not always give the same value for a certain individual is used, for example a fitness function that depends on the initial position of a physical test object or some other fitness function depending on random factors.
Run evolution
This button runs the evolution.

Stop evolution
This button stops the evolution.

Individual information

![Individual information dialog box](image)

This dialogue box shows information about an individual. It is possible to select one of the state machines in the left-most list. The states and the corresponding outputs will then be shown in the list in the middle. When one state is selected the corresponding state-transitions will be shown in the right-most list.

Menus
Most of the alternatives in the menus are self-explanatory so the descriptions are rather short. Some menus and alternatives are left out since they do not differ from standard windows programs.
File menu

- **New System** creates a new system, lets the user specify a name and location.
- **Open System…** opens an existing system.
- **Save System** saves the system.
- **Save System As…** lets the user specify a new name and location for the system.
- **Load Individual** loads an and replaces the selected individual in “Individual progress” in the main window.
- **Save Individual As…** saves the individual selected in “Individual progress” in the main window.
- **Print** Prints some information about the evolutionary system.

Edit menu

- **Edit system** invokes the dialogue box “New system / Edit system”.

"New System" creates a new system, lets the user specify a name and location.

"Open System…" opens an existing system.

"Save System" saves the system.

"Save System As…" lets the user specify a new name and location for the system.

"Load Individual" loads an and replaces the selected individual in “Individual progress” in the main window.

"Save Individual As…" saves the individual selected in “Individual progress” in the main window.

"Print" Prints some information about the evolutionary system.

"Edit system" invokes the dialogue box “New system / Edit system”.
Using SoftGP

The three main parts of the program are the dialogue box “New system / Edit system”, the main window of the program and the dialogue box “Individual information”. These three parts and the commands in the menus will be described below.

New system / Edit system dialogue box

This dialogue box is used to create a completely new evolutionary system of processors. It can also be used to alter an old system. In this dialogue box the user specifies the system environment and fitness functions, the number of processors and how everything is connected.

System environment and fitness functions

This field displays the chosen file containing the system environment and the fitness functions. This file should only contain the entity containing the system environment and all the fitness functions. The “Browse….” button opens a dialogue box where the user can choose a new system environment file.

Processor

The drop-down list shows the latest added or altered processor. The drop-down list “input/output” below shows the inputs and outputs on the active processor. If the selected input or output is connected to a system environment signal, this signal is shown in the drop-down list “System environment input/output”. If it is not connected this field will be left blank. The drop-down list “Fitness function” shows the signal in the system environment that is to be considered as fitness function for the active unit.

It is possible to add and delete processors using the buttons “Add” and “Delete”. When adding a processor a dialogue box pops up that prompts for a name of the processor and it is possible to add inputs and outputs.
When an input or output is selected in the drop-down list "input/output" it is possible to connect or disconnect it from the system environment using the buttons "Connect" and "Disconnect".

Connection properties
This list shows the inputs/outputs from the processors and the signals that they are connected to in the system environment.

Build system
This button creates a VHDL-file. It also closes the dialogue box.

Main view

In the left part of the main view the user can alter the settings for the evolution.

Algorithm
The algorithm chosen for the evolution.

Type of evolution
With this slider the user can specify how much of the evolution that will be performed at the program level and how much that will be performed at the system level.
Mutation frequency
The frequency with which mutations happen.

Duplication frequency
The frequency with which duplications happen.

Assembler program size
The size of the assembler program.

Population size
Number of programs in each processor.

Generation
Number of generations tested.

Status
Information about if the evolution is running or not.

Fitness progress
A graph describing the progress of the evolution. The progress for the selected processor is shown in this graph.

Reset graph
Resets the graph. Typically used when the fitness function is altered.

Individual progress
Information about the individuals. The user can select a processor in the left list and the individuals of this processor will be shown in the right list. The user can double-click on an individual to get more detailed information.

Retest individuals
This check box only effects the steady-state algorithm. If this box is checked, the individuals will be re-tested every time. This check-box should be checked when a fitness function that does not always give the same value for a certain individual is used, for example a fitness function that depends on the initial position of a physical test object or some other fitness function depending on random factors.

Run evolution
This button runs the evolution.

Stop evolution
This button stops the evolution.
This dialogue box shows the source code and some other information about an individual.

Menus

Most of the alternatives in the menus are self-explanatory so the descriptions are rather short. Some menus and alternatives are left out since they do not differ from standard windows programs.
“New System” creates a new system, lets the user specify a name and location and opens the dialogue box “New system / Edit system”.

“Open System…” opens an existing system.

“Save System” saves the system.

“Save System As…” lets the user specify a new name and location for the system.

“Load Population” loads a population for the processor selected in “Individual progress” in the main window.

“Save Population As…” saves a population for the processor selected in “Individual progress” in the main window.

“Load Individual” loads an individual for the processor selected in “Individual progress” in the main window and replaces the selected individual in “Individual progress” in the main window.

“Save Individual As…” saves the individual selected in “Individual progress” in the main window.

“Print” Prints some information about the evolutionary system.

**Edit menu**

```
Edit > View > Help
Edit System
```

“Edit system” invokes the dialogue box “New system / Edit system”.
APPENDIX C – FILE FORMATS

File formats for VHDLGP

The entity that is to be used as system environment and fitness function for VHDLGP shall have the following structure:

```vhdl
entity system is
    port (
        clk, reset : in STD_LOGIC;
        miscreset : out STD_LOGIC;
        random : in STD_LOGIC_VECTOR(7 downto 0);
        input1 : in STD_LOGIC_VECTOR(7 downto 0);
        input2 : in STD_LOGIC_VECTOR(7 downto 0);
        output1 : out STD_LOGIC_VECTOR(7 downto 0);
        fitnessout : out STD_LOGIC_VECTOR(7 downto 0)
    );
end system;
```

The name of the entity does not matter as long as it does not interfere with other names. The name system can be used to be safe.

The first row is ignored and must be clk and reset.
The second row is ignored and must be misc_reset.
The third row is ignored and must be random. VHDLGP will use this port to supply a random number to the fitness function if it needs one.
The names on the rest of the rows will be read but not the type. The type must be STD_LOGIC_VECTOR(7 downto 0). It is not permitted to write more than one name on each row.

The file that will be used as stimuli file must follow the guidelines for running ModelSim XE/PE in batch mode. See reference [6] and [7]. The last row must be “exa xxxxxx”, where xxxxx is to be replaced with the name of the port that outputs the fitness value. An example of a stimuli file is shown below.

```vhdl
force clk 1 0, 0 5 -repeat 10
force reset 1 0, 0 7
force input1 01000000 0
force input2 00000100 0
run 200
exa fitnessout
```

The two files must be located in a project directory for ModelSim. The system environment architecture must be compiled before running VHDLGP.
File format for SoftGP

The entity that is to be used as system environment and fitness function for SoftGP shall have the following structure:

```vhdl
entity system is
  port (  
    clk, reset     : in STD_LOGIC;
    done, miscreset : out STD_LOGIC;
    input1         : in STD_LOGIC_VECTOR(7 downto 0);
    input2         : in STD_LOGIC_VECTOR(7 downto 0);
    output1        : out STD_LOGIC_VECTOR(7 downto 0);
    fitnessout     : out STD_LOGIC_VECTOR(7 downto 0)
  );
end system;
```

The name of the entity does not matter as long as it does not interfere with other names. The name system can be used to be safe.

The first row is ignored and must be clk and reset

The second row is ignored and must be done and misc_reset. The done signal shall give a logic high when the fitness value is calculated.

The names on the rest of the rows will be read but not the type. The type must be STD_LOGIC_VECTOR(7 downto 0). It is not permitted to write more than one name on each row.
APPENDIX D – PROGRAM DOCUMENTATION

Documentation for VHDLGP

Document structure

The document in VHDLGP contains an evolutionary system. The evolutionary system is built from the population of individuals, the system environment including fitness function provided by the user and the settings for the evolution. The structure is shown below.

Individual

The individuals that are evolved by the program are built from several state machines which are all connected to a bus. Each state machine is assigned 16 bits on the bus where it will present its output. The input to the state machines can be any of the bits from the bus or just bits from neighbouring machines depending on the settings. The inputs to the combinatorial net can be any of the bits on the bus. The input to the individual is also assigned bits on the bus. The output from the individual can be any of the bits on the bus or outputs from the combinatorial net.

Views

There are three views of the program. One view is used to specify system environment and number of inputs and outputs. The second one contains controls to alter the settings of the evolution and also provides information about the evolution progress. The third view shows information about an individual.
Classes

This section contains a description of the classes that are relevant to the function of the program. Most classes have functions to access the values of private variables. These functions and other functions and variables with obvious behaviour are not described here. Classes, functions and variables that are automatically generated by the developing environment are not described either if their behaviour does not differ from the standard behaviour.

class CChangePopulationSize : public CDialog

Dialogue box used to change the population size.

Variables

int m_OldSize;
The class that opens the dialogue box uses this variable to provide an init value to the size field.

class CIndividual : public CObject

Represents an individual.

Variables

DWORD m_DatabusLength;
The length of the data bus.
DWORD m_NumberOfStateMachines;
The number of state machines that the individual contains.
DWORD m_Fitness;
The last calculated fitness value for the individual.
CWordArray m_OutputNet;
The truth table for the outputnet.
CWordArray m_OutputNetInputs;
Information about how the combinatorial outputnet is connected
CWordArray m_OutputNetOutputs;
Information about how the outputs are connected
CTypedPtrList<CObList, CStateMachine*> m_StateMachineList;
List of pointers to the state machines that the individual contains.
POSITION m_StateMachineListPos;
Variable used to traverse the state machine list.

Functions

CIndividual();
Constructor used when loading an object from disk. Does not create anything more than the object itself.
CIndividual(CSys* sysp);
Constructor used when creating a population. I creates state machines and output nets and connections according to the settings that can be accessed through sysp.
void MixAndReplace(CIndividual* OldInd1, CIndividual* OldInd2);
Replaces the current state machines and output net with a mixture of the two individuals OldInd1 and OldInd2.

CStateMachine* GetFirstStateMachine();
Returns the first state machine.

CStateMachine* GetNextStateMachine();
Returns the next state machine in the list.

CStateMachine* GetStateMachine(int mnr);
Gets the state machine which is placed in the position indicated by mnr.

CString ToBinary(int nr, int bits);
Returns a string of ones and zeros that represents the number nr. The number of bits that are created depends on the variable bits.

CString GetOutputCode();
Returns a string with VHDL-code that describes the combinatorial net on the output.

class CIndividualInfo : public CDialog
Dialogue box that serves as the third view of the program. This dialogue box shows information about the individual.

Variables

int m_IndNumber;
The class that opens the dialogue box uses this variable to provide an init value to the individual number field, since the individual number is not stored in the individual.

CIndividual* m_IndividualPoint;
Pointer to the individual to show information about.

Functions

BOOL CIndividualInfo::OnInitDialog();
Adds the state machines to the box showing the state machines and inits other fields.

void CIndividualInfo::OnSelchangeMachinelist();
Clears the box showing the state transitions and adds the states for the current state machine to the box showing the states. Also write some other information about the current state machine in other fields.

void CIndividualInfo::OnSelchangeStatelist();
Adds the state transitions from the currently selected state to the box showing state transitions.

class CNewEditSys : public CDialog
Dialogue box used to edit a system.

Variables

CSys* m_SystemPoint;
Pointer to the evolutionary system.

CPort* m_SelectedPort;
Pointer to the port that is currently selected in the box that shows the different ports.
Functions

void CNewEditSys::OnBrowse();
Opens a file browser and lets the user select a VHDL-file that is supposed to be used as system environment and fitness function. Reads this file and updates the different boxes that show information about the system environment. Calls the function CreatePorts.

void CNewEditSys::OnSelchangeInOutBox();
Enables/Disables the add and delete buttons depending on if the selected port is connected or not.

void CNewEditSys::OnAdd();
Connects the port and enables/disables the add and delete buttons.

void CNewEditSys::OnDelete();
Disconnects the port and enables/disables the add and delete buttons.

BOOL CNewEditSys::OnInitDialog();
Calls the update functions to get init values to the different boxes. Puts the system environment path in the system environment field.

void CreatePorts(CString str);
Finds inputs and outputs in the string and creates ports to each of them.

void UpdateInOutBox();
Adds all ports to the box showing the ports.

void UpdateInOutList();
Adds connected ports to the list showing the connected ports.

class CPort : public CObject

Represents a port in the system environment.

Variables

CString m_Name;
The name of the port.

bool m_IsInput;
Tells if the port is an input or output.

bool m_Connected;
Tells if the port is connected to the evolved individual or if it is a port to the rest of the world.

class CSettings : public CObject

Contains the settings for the evolution.

Variables

DWORD m_ConnectionRange;
A number between 1 and 100 that tells how far away on the bus the inputs of a state machine can connect. 1 means that it can only connect to the neighbours while 100 means that it can connect to all signals.

DWORD m_NrOfStates;
The maximum number of states a state machine is allowed to have.
DWORD m_NrOfMachines;
The maximum number of machines an individual is allowed to have.
bool m_Algorithm;
Which algorithm that is to be used
DWORD m_MutationFreq;
Mutation frequency
DWORD m_DuplicationFreq;
Duplication frequency
DWORD m_PopulationSize;
Size of the population
bool m_PureMoore;
True if the state machines shall be pure Moore machines.
bool m_FixedMachine;
True if the number of machines shall be fixed.
bool m_FixedState;
True if the number of states shall be fixed.
bool m_ReTest;
True if the individuals shall be retested every time.

class CStateMachine : public CObject
Represents a state machine.

Variables
DWORD m_NumberOfInputs;
The number of inputs to the state machine.
DWORD m_NumberOfOutputs;
The number of outputs from the state machine.
DWORD m_NumberOfStates;
The number of states of the state machine.
DWORD m_StateMachineNumber;
The state machine number.
bool m_PureMoore;
True if it is a pure Moore machine.
CWordArray m_OutputVector;
The output values for the states
CWordArray m_InputConnection;
Information about to which bits on the bus the inputs are connected to.
CAArray<CAArray<DWORD, DWORD&>, CAArray<DWORD, DWORD&>>& m_StateTransitions;
Information about the state transitions.

Functions
CStateMachine();
Constructor used when loading an object from disk. Just constructs the object.
CStateMachine(CSettings* sp, DWORD number, CSys* sysp);
Constructor used when creating a new state machine. Creates states, transitions and connection information.
CStateMachine(CStateMachine& constr);
Copy constructor used to make a copy of a state machine.
CString GetCode();
Returns a string of VHDL-code that describes the state machine.
CString ToBinary(int nr, int bits);
Returns a string of ones and zeros that represent the number nr as binary code with number of bits depending on the variable bits.

class CSys : public CObject

Represents the evolutionary system with population, environment and settings.

Variables

DWORD m_Generation;
Generation counter.
CSysEnv* m_SysEnvPoint;
Pointer to the system environment.
CSettings m_Settings;
The settings of the evolution
CTypedPtrList<CObList, CIndividual*> m_IndividualList;
List of pointers to the individuals. This is the population.
POSITION m_IndividualListPos;
Used to traverse the individual list.
DWORD m_NumberOfInputs;
Number of inputs to the system.
DWORD m_NumberOfOutputs;
Number of outputs from the system.
bool m_NotTested;
Tells if the individuals are tested or not.
CByteArray m_BestFitness;
Keeps history of the evolution.
CWordArray m_AverageFitness;
Keeps history of the evolution.

Functions

int ToInteger(CString str);
Converts a string of ones and zeros to an integer.
void Evolve(CString path);
Tests the individuals if they are not tested. Calls the appropriate algorithm.
void MonkeyAlgorithm();
Runs one generation of the algorithm “30 monkeys in a bus”.
void ClassicAlgorithm();
  Runs one generation of the classic algorithm.
void TestIndividual(CIndividual* ip);
  Creates a VHDL-file from an individual, calls the simulator and reads the fitness value.
void AddIndividual(CIndividual* ip){m_IndividualList.AddTail(ip);};
  Adds an individual to the system.
CIndividual* GetFirstIndividual();
  Returns the first individual.
CIndividual* GetNextIndividual();
  Returns the next individual.
void SetIndividual(CIndividual* indp);
  Sets the individual pointed to by indp to the current individual.
void DeletePopulation();
  Deletes the population.
CIndividual* GetIndividual(int indnr);
  Gets the individual with the number indnr.
void RemoveLastIndividual(){delete m_IndividualList.RemoveTail();};
  Removes and deletes the last individual from the list.

class CSysEnv : public CObject

  Represents the system environment and fitness function.

Variables

CTypedPtrList<CObList, CPort*> m_PortPointList;
  List of pointers to the ports that describes inputs and outputs for the entity that the system environment represents.
POSITION m_PortPointListPos;
  Variable used to traverse the port pointer list.
CString m_VHDLCode;
  The VHDL-code that describes the entity
CString m_VHDLPath;
  The path to the code.

Functions

void AddPort(CPort *pp){m_PortPointList.AddTail(pp);};
  Adds a port to the port pointer list.
void DeletePorts();
  Deletes all ports.
CPort* GetFirstPort();
  Gets the first port.
CPort* GetNextPort();
  Gets the next port.
CPort* GetPort(CString str);
Gets the port with the name str.
int GetPortNr(CPort* pp);
Gets the number of the port pp.
CPort* GetPortAddress(int nr);
Gets a pointer to the port with the number nr.

**class CVHDLGPD**

**Doc : public CDocument**
The document used in the program.

**Variables**
CSys* m_SystemPoint;
Pointer to the evolutionary system.

**Functions**
void CVHDLGPD::OnEditSystem();
Opens the dialogue box CNewEditSys. Creates a population if a population does not exist.

**class CVHDLGP**

**View : public CFormView**
The main view of the program.

**Variables**
bool m_RunEvol;
True if the evolution is supposed to be running.
CIndividual* m_SelectedIndividual;
Pointer to the selected individual.
UINT m_Timer;
ID for the timer to be able to kill it.

**Functions**
void DrawDiagram();
Draws a diagram over the evolution.
void UpdateControls();
Reads the settings from the system and updates all controls on the view.
void UpdateIndividualBox();
Adds all the individuals to the box that shows the individuals.
void CVHDLGPView::OnChangesize();
Opens the dialogue box CChangePopulationSize. Adds or deletes individuals according to the user’s choice.
void CVHDLGPView::OnDblclkIndprogress();
Opens the dialogue box CIndividualInfo.
void CVHDLGPView::OnInitialUpdate();
Initialises some controls and calls UpdateControls().
void CVHDLGPView::OnKillfocusDupfreqfield();
Reads the duplication frequency.
void CVHDLGPView::OnKillfocusMutfreqfield();
Reads the mutation frequency.
void CVHDLGPView::OnKillfocusStatefield();
Reads the number of states.
void CVHDLGPView::OnKillfocusStatemachfield();
Reads the number of state machines.
void CVHDLGPView::OnMoore();
Alter the settings regarding the type of machines.
void CVHDLGPView::OnPuremoore();
Alter the settings regarding the type of machines.
void CVHDLGPView::OnReleasedcaptureConnections(NMHDR* pNMHDR, LRESULT* pResult);
Alter the settings regarding the connections.
void CVHDLGPView::OnRunevol();
Sets m_RunEvol and starts a timer.
void CVHDLGPView::OnSelchangeAlgorithm();
Alter the settings regarding the algorithm.
void CVHDLGPView::OnSelchangeIndprogress();
Read which individual that is selected and write to m_SelectedIndividual.
void CVHDLGPView::OnStatedyn();
Alter the settings regarding the number of states.
void CVHDLGPView::OnStatefixed();
Alter the settings regarding the number of states.
void CVHDLGPView::OnStatemachdyn();
Alter the settings regarding the number of state machines.
void CVHDLGPView::OnStatemachfixed();
Alter the settings regarding the number of state machines.
void CVHDLGPView::OnStopevol();
Kill the timer and clear m_RunEvol.
void CVHDLGPView::OnTimer(UINT nIDEvent);
Kills the timer, runs the right algorithm, calls UpdateIndividualBox() and starts a new timer.
void CVHDLGPView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
Calls UpdateControls() and UpdateIndividualBox().
Documentation of SoftGP

Document structure

The document in SoftGP contains an evolutionary system. The evolutionary system is built from the system environment including fitness function provided by the user, the settings for the evolution and one or more processors, each containing a population of programs. The structure is shown below.

Individual

The individuals that are evolved are built from several program constructs, which are sequenced after each other.

Views

There are three views of the program. One view is used to specify system environment and number of processors and their inputs and outputs. The second one contains controls to alter the settings of the evolution and also provides information of the evolution progress. The third view shows information about an individual.

Classes

This section contains a description of the classes that are relevant to the function of the program. Most classes have functions to access the values of private variables. These functions and other functions and variables with obvious behaviour are not described here. Classes, functions and variables that are automatically generated by the developing environment are not described either if their behaviour does not differ from the standard behaviour.
class CAddPort : public CDialog
Dialogue box used to add a port to a processor.

Variables

bool m_Input;
True if the radio button “input” is checked

Functions

BOOL OnInitDialog()
Initialises the radio button according to the value of m_Input;
void OnInput()
Called when the user clicks on the radio button. Alters the value of m_Input.
void OnOutput()
Called when the user clicks on the radio button. Alters the value of m_Input.

class CAddProcessor : public CDialog
Dialogue box used to add a processor.

Variables

CUnit* m_UnitPointer;
Pointer that shall point to a newly created object of the class CUnit.
int m_NumberOfOutputs;
The number of outputs created for the processor.
bool m_InputSelected;
Indicates if the selected port is an input or output.

Functions

CAddProcessor(CWnd* pParent /*=NULL*/) : CDialog(CAddProcessor::IDD, pParent)
Constructor. Initialises m_NumberOfOutputs.
void OnAddport()
Opens a dialogue box of the class CAddPort and adds a port to the processor according to the information from the dialogue box.
void OnDeleteport()
Deletes the selected port.
void UpdateInputList(CString str)
Clears the list box showing the input ports and adds all the input ports to the list box.
void UpdateOutputList(CString str)
Clears the list box showing the output ports and adds all the output ports to the list box.
void OnSelchangeInputs()
Sets the variable m_InputSelected and enables the “delete” button.
void OnSelchangeOutputs()
Clears the variable m_InputSelected and enables the “delete” button.
BOOL OnInitDialog()
Disables the “delete” button.
Void OnKillfocusUnitname()
Updates the name of the processor.

class CChangePopulationSize : public CDialog
Dialogue box used to change the size of the population.

Variables
int m_OldSize;
Variable used to transfer the old size to the dialogue box.

class CCommunicate
Class that handles the communication with the FPGA-board.

Functions
CCommunicate()  
Constructor that sets the direction of the port.
~CCommunicate()  
Destructor that restores the direction of the port.
void SendByte(BYTE data)  
Sends a byte to the FPGA-board
BYTE CCommunicate::RecieveByte()  
Receives a byte from the FPGA-board.

class CIndividual : public CObject
Class used to represent an individual.

Variables
DWORD m_Fitness;
Keeps the fitness value of the individual.
DWORD m_Connections;
Not used.
CTypedPtrList<CObList, CPgmConstruct*> m_MainPointList;
List of pointers to CPgmConstructs that are to be used as the main flow in the program.
CTypedPtrList<CObList, CPgmConstruct*> m_SubroutinePointList;
List of pointers to CPgmConstructs that are to be used as subroutines in the program.
DWORD m_NumberOfSubroutines;
The number of subroutines.
POSITION m_MainPointListPos;
Variable used to traverse the list m_MainPointList;
POSITION m_SubroutinePointListPos;
Variable used to traverse the list m_SubroutinePointList;
POSITION m_MainPointListPos2;  
Variable used to traverse the list m_MainPointList;  
POSITION m_SubroutinePointListPos2;  
Variable used to traverse the list m_SubroutinePointList;  
CUnit* m_UnitPoint;  
Pointer to the processor that owns the individual.

**Functions**

void DeleteProgram();  
Clears the list m_MainPointList and m_SubroutinePointList.

doMutate(CSettings* sp);  
Applies a mutation to the individual.

void MixAndReplace(CIndividual* OldInd1, CIndividual* OldInd2);  
Deletes the current contents of the individual and replaces this with a mix of the two individuals supplied as arguments.

CWordArray* GetMachineCode();  
Generates and returns the machine code of the individual.

CString GetSourceCode();  
Generates and returns the source code of the individual.

int GetSize();  
Calculates and returns the size of the individual.

CPgmConstruct* GetFirstMainPgmConstruct();  
Gets the first CPgmConstruct in the list m_MainPointList.

CPgmConstruct* GetNextMainPgmConstruct();  
Gets the next CPgmConstruct in the list m_MainPointList.

CPgmConstruct* GetFirstSubroutinePgmConstruct();  
Gets the first CPgmConstruct in the list m_SubroutinePointList.

CPgmConstruct* GetNextSubroutinePgmConstruct();  
Gets the next CPgmConstruct in the list m_SubroutinePointList.

CPgmConstruct* GetFirstMainPgmConstruct2();  
Gets the first CPgmConstruct in the list m_MainPointList.

CPgmConstruct* GetNextMainPgmConstruct2();  
Gets the next CPgmConstruct in the list m_MainPointList.

CPgmConstruct* GetFirstSubroutinePgmConstruct2();  
Gets the first CPgmConstruct in the list m_SubroutinePointList.

CPgmConstruct* GetNextSubroutinePgmConstruct2();  
Gets the next CPgmConstruct in the list m_SubroutinePointList.

class CIndividualInfo : public CDialog  
Dialogue box used to show information about an individual.

**Variables**

int m_IndNumber;
The number of the individual supplied by the caller.
CIndividual* m_IndividualPoint;
Pointer to the individual supplied by the caller.

Functions

BOOL OnInitDialog()
Initialises all the fields in the box with information from the individual.

class CNewEditSys : public CDialog
Dialogue box used to create a new system or edit an old one.

Variables

CUunit* m_SelectedUnit;
Unit selected in the list
CPort* m_SelectedPort;
Port selected in the list
CPort* m_SelectedEnvPort;
Environment port selected in the list
CSys* m_SystemPoint;
Used to supply the system to the dialogue box.

Functions

void OnBrowse();
Invokes a file requester and loads the system environment that is chosen in the requester.
void OnAdd();
Creates a new processor and invokes the dialogue box CAddUnit.
void OnSelchangeUnitname();
Updates the input/output list to the inputs/outputs from the selected processor.
void OnSelchangeFitnessfunc();
Changes the fitness function selection for the current processor.
void OnSelchangeInoutname();
If the selected input/output is connected, the connected input/output is shown in the list below.
m_SelectedPort is changed.
void OnSelchangeEnvinoutname();
m_SelectedEnvPort is changed.
void OnConnect();
Connects the two selected ports.
void OnDisconnect();
Disconnects the two selected ports.
BOOL OnInitDialog();
If the system is old, the dialogue box is initialised with data from the system.
void OnDelete();
Deletes a processor;
void OnOK();
Opens a file requester and creates a VHDL-entity and architecture that is written to the selected file.

void UpdateUnitTest(CString str);
Removes all elements from the combo box showing the processor and adds the processors to the box.
void UpdateInOutList();
Removes all elements from the combo box showing the inputs/outputs and adds the inputs/outputs to the box.
void UpdateEnvInOutList();
Removes all elements from the combo box showing the environment inputs/outputs and adds the environment inputs/outputs to the box.
void UpdateFitnessList();
Removes all elements from the combo box showing the selected fitness port and adds the environment inputs/outputs to the box.
void UpdateConnectionList();
Removes all elements from the list box showing the connections between inputs and outputs and adds the connected inputs and outputs to the list.

CString ToBinary(int nr);
Returns a string with ones and zeros representing the integer nr as a binary number.

void CreatePorts(CString str, CUnit* up, bool AddToUnit);
Parses the VHDL-entity supplied in the CString str and adds the ports to the system.

class CPgmConstruct : public CObject
Represents a program construct such as a sequence, loop or if-statement.

Variables
CIndividual* m_IndividualPoint;
Points to the individual that owns the contract.
WORD m_FirstSize;
Number of instructions in the first piece of code
WORD m_SecondSize;
Number of instructions in the second piece of code
WORD m_ForLoopNumber;
Number of times to loop ir it is a for loop
WORD m_RegisterOne;
The first register if the construct is conditional
WORD m_RegisterTwo;
The first register if the construct is conditional
DWORD m_Condition;
type of condition > < = != (C, NC, Z, NZ)
DWORD m_Length;
Length of construct including the init and jump instructions
DWORD m_ConstructType;
Type of construct (if, for, while)
CWordArray m_FirstCode;
The code included in the first part of the construct
CWordArray m_SecondCode;
The code included in the second part (after else)
bool m_LoopOk;
Tells if it is ok to include second loop (nested)
bool m_CallOk;
Tells if it is ok to call a subroutine (to avoid recurion)
CTypedPtrList<COBList, CPgmConstruct*> m_ConstructPointList;
List of pointers to PgmConstructs that are to be inlined in the present construct.
WORD m_NumberOfConstructs;
Number of inlined constructs.
POSITION m_ConstructPointListPos;
Variable used to traverse the list m_ConstructPointList;
POSITION m_ConstructPointListPos2;
Variable used to traverse the list m_ConstructPointList;
WORD m_Level;
Tells which level the construct is. Used to indent the code.

Functions

int GetDeltaLength(WORD instruction);
returns the size of the instruction (0, 1 or 2)
int GetDeltaLength2(WORD instruction);
returns the size of the instruction including size of an inlined construct
CString Disassemble(WORD instruction);
Disassembles an instruction.
CPgmConstruct* GetFirstConstruct();
Gets the first CPgmConstruct in the list m_ConstructPointList.
CPgmConstruct* GetNextConstruct();
Gets the next CPgmConstruct in the list m_ConstructPointList.
CPgmConstruct* GetFirstConstruct2();
Gets the first CPgmConstruct in the list m_ConstructPointList.
CPgmConstruct* GetNextConstruct2();
Gets the next CPgmConstruct in the list m_ConstructPointList.
WORD GetRandomRegister();
Returns a random register or port.
WORD GetRandomInstruction();
Returns a random instruction.
void RemoveConstruct(int constrnr);
Removes a construct from the construct list and adjusts references to the rest of the constructs.
CWordArray* GetMachineCode(BYTE startaddr);
Creates and returns the machine code for the construct.
BYTE GetLength();
Returns the length of the construct.
CString GetSourceCode();
Creates and returns the source code for the construct.

void Mutate();
Applies a mutation to the construct.

class CPort : public CObject

Represents a port.

Variables

CString m_Name;
Name of the port.
bool m_IsInput;
True if the port is input.
bool m_IsFitness;
True if the port is a fitness port.
bool m_Connected;
True if the port is connected.
CPort* m_PortPoint;
Points to an output if the port is an input and connected, undefined if not connected or if it is an output
int m_Size;
Size of the port.

class CSettings : public CObject

Contains the settings for the evolution.

Variables

bool m_Algorithm;
Which algorithm that is to be used.
DWORD m_EvolutionType;
Program vs. System.
DWORD m_MutationFreq;
Mutation frequency.
DWORD m_DuplicationFreq;
Duplication frequency.
bool m_FixedSize;
True if the size is fixed.
DWORD m_ProgramSize;
Size of program.
DWORD m_PopulationSize;
Size of population
bool m_UploadCont;
Not used.
bool m_ReTest;
True if the individuals shall be retested every time.

class CSoftGPDoc : public CDocument

The document class for the program.

Variables

CCommunicate m_Communicate;
Object of type CCommunicate to be able to communicate with the FPGA-board.
CSys* m_SystemPoint;
Pointer to the evolutionary system.

class CSoftGPView : public CFormView

The main view of the program.

Variables

CUnit* m_SelectedUnit;
Points to the selected processor.
CIndividual* m_SelectedIndividual;
Points to the selected individual.
bool m_RunEvol;
True if the evolution shall be running.
UINT m_Timer;
Timer used to trigger the evolution.

Functions

void OnKillfocusProgramsizefield();
Updates the program size in the settings.
void OnKillfocusMutfreqfield();
Updates the mutation frequency in the settings.
void OnKillfocusDupfreqfield();
Updates the duplication frequency in the settings.
void OnSelchangeAlgorithm();
Updates the algorithm in the settings.
void OnReleasedcaptureEvolutiontype(NMHDR* pNMHDR, LRESULT* pResult);
Updates the evolution type in the settings.
void OnChangesize();
Invokes the dialogue box CChangePopulationSize. Deletes individuals if the new size is less than the old size. Creates individuals if the new size is greater than the old size.
void OnSelchangeUnitlist();
Lets m_SelectedUnit point to the selected processor.
void OnRunevol();
Changes m_RunEvol and starts a timer that trigs the evolution.
void OnStopevol();
Changes m_RunEvol and deletes the timer.
void OnDblclkIndividualbox();
Invokes the dialogue box CIndividualInfo.
void OnSelchangeIndividualbox();
Lets m_SelectedIndividual point to the selected individual.
void OnTimer(UINT nIDEvent);
Stops the timer, calls CSys::Evolve and starts a new timer.
void OnRetest();
Changes the retest setting in the settings.
void OnResetgraph();
Changes the setting that tells if the population has been tested.
void UpdateControls();
Updates the controls of the view with data from the settings.
void UpdateUnitBox();
Removes all items from the list box that shows the processors and adds all the processors to the list box.
void UpdateIndividualBox();
Removes all items from the list box that shows the individuals and adds all the individuals to the list box.
void DrawDiagram();
Draws a diagram of the evolution progress.

class CSys : public CObject

Represents the evolutionary system.

Variables

bool m_NotTested;
True if the population has not been tested.
DWORD m NrOfProcessors;
The number of processors.
POSITION m_UnitPointListPos;
Variable used to traverse the list m_UnitPointList.
CTypedPtrList<CObList, CUnit*> m_UnitPointList;
List of pointers to processors.
CSysEnv* m_SysEnvPoint;
Pointer to system environment.
CSettings m_Settings;
Settings for the evolution
DWORD m_Generation;
Tells how many generations that have passed

Functions

void Evolve(CCommunicate *cp);
Tests the population if it is not tested and calls the appropriate algorithm.
void TestIndividual(int nr, CCommunicate *cp);
Tests the individual with the number nr.
CUnit* GetUnitNr(int nr);
Gets the processor with the number nr.

void MonkeyAlgorithmProg(CCommunicate* cp);
Runs the steady state algorithm at the program level.

void ClassicAlgorithmProg(CCommunicate *cp);
Runs the classic algorithm at the program level.

void MonkeyAlgorithmSys(CCommunicate* cp);
Runs the steady state algorithm at the system level.

void ClassicAlgorithmSys(CCommunicate *cp);
Runs the classic algorithm at the system level.

CUnit* GetFirstUnit();
Gets the first processor in the list m_UnitPointList.

CUnit* getNextUnit();
Gets the next processor in the list m_UnitPointList.

CUnit* GetUnit(CString str);
Gets the processor with the name str in the list m_UnitPointList.

void AddUnit(CUnit *up);
Adds a processor to the list m_UnitPointList.

void DeleteUnit(CUnit *up);
Deletes a unit from the list m_UnitPointList.

class CSysEnv : public CObject

Represents the system environment.

Variables

CTypedPtrList<CObList, CPort*> m_PortPointList;
List of pointers to the ports that belong to the environment.

POSITION m_PortPointListPos;
Variable used to traverse the list m_PortPointList.

CString m_VHDLCode;
VHDL-code that describes the entity

CString m_VHDLPath;
Path to the code file.

Functions

void AddPort(CPort *pp);
Adds a port to the list m_PortPointList.

void DeletePorts();
Deletes all ports from the list m_PortPointList.

CPort* GetFirstPort();
Gets the first port in m_PortPointList.

CPort* GetNextPort();
Gets the next port in m_PortPointList.
CPort* GetPort(CString str);
Gets the port with the name str from m_PortPointList.

int GetPortNr(CPort* pp);
Gets the port number of the port pp.

CPort* GetPortAddress(int nr);
Gets the address to the port with the number nr.

class CUnit : public CObject

Represents a processor.

Variables

WORD m_NumberOfInputs;
The number of inputs.

WORD m_NumberOfOutputs;
The number of outputs.

bool m_IsNew;
Tells if the processor is new and a population of programs should be created.

POSITION m_PortPointListPos;
Variable used to traverse the list m_PortPointList.

POSITION m_IndividualPointListPos;
Variable used to traverse the list m_IndividualPointList.

CString m_Name;
The name of the processor.

bool m_IsProcessor;
Not used. Must be true.

CTypedPtrList<CObList, CPort*> m_PortPointList;
List of pointers to the ports that belong to the processor.

CTypedPtrList<CObList, CIndividual*> m_IndividualPointList;
List of pointers to individuals (this list is a population)

CString m_VHDLPath;
Not used.

CString m_VHDLCode;
Not used.

CPort* m_Fitness;
The port in the system environment that is to be used as fitness for the processor.

CSys* m_System;
The system that the processor belongs to.

CByteArray m_BestFitness;
History of fitness values.

CWordArray m_AverageFitness;
History of fitness values.
### Functions

```c
void RemoveLastIndividual;
Removes and deletes the last individual in the list.

void DeleteIndividuals();
Removes and deletes all the individuals.

void SerializeIndividuals(CArchive& ar);
Serializes the individuals only.

CIndividual* GetIndividual(int indnr);
Gets the individual with the number indnr.

void AddPort(CPort *pp);
Adds the port pp to the list.

void DeletePort(CPort *pp);
Removes and deletes the port pp from the list.

CPort* GetFirstPort();
Gets the first port in the list.

CPort* GetNextPort();
Gets the next port in the list.

CPort* GetPort(CString str);
Gets the port with the name str.

CIndividual* GetFirstIndividual();
Gets the first individual.

CIndividual* GetNextIndividual();
Gets the next individual.

void AddIndividual(CIndividual* ip){m_IndividualPointList.AddTail(ip);};
Adds an individual to the list.

void AddBestFitness(BYTE fit){m_BestFitness.Add(fit);};
Add an element to the array m_BestFitness.

void AddAverageFitness(WORD fit){m_AverageFitness.Add(fit);};
Add an element to the array m_AverageFitness.
```