Exploring the Search Space of HW/SW Embedded Systems by Means of GP

Milos Minarik and Lukas Sekanina

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
iminarikm@fit.vutbr.cz, sekanina@fit.vutbr.cz

Abstract. This paper presents a new platform for development of small application-specific digital embedded architectures based on a data path controlled by a microprogram. Linear genetic programming is extended to evolve a program for the controller together with suitable hardware architecture. Experimental results show that the platform can automatically design general solutions as well as highly optimized specialized solutions to benchmark problems such as maximum, parity or iterative division.

1 Introduction

A general research problem, practically untouched by the genetic programming community, is whether it is possible to concurrently evolve hardware and software for a given task and whether the evolutionary system can discover general solutions to the problem (under some constraints) or highly optimized solutions to the same problem under other constraints. For example, in the case of the n-input parity problem, one evolved solution would be a general program sequentially performing the XOR operation over the intermediate result and the incoming bit (no constrains on the execution time are formulated, but the hardware size is constrained), while another solution would be an n-bit parity tree calculating the parity in parallel (no constrains on the hardware size are given, but the execution time is constrained).

This type of problems can be investigated using a platform [1] that we have developed for design and optimization of small HW/SW embedded systems, in which it is impossible to employ a general purpose processor because of its relatively high cost. The platform consists of an application specific data path controlled by a programmable logic controller which is programmed using the so-called microprograms. The overall architecture as well as the microprogram are highly optimized in order to minimize area, delay and power consumption. The designer has to determine the number of registers and their bit width, the number of ALUs, the set of functions supported by each ALU, interconnection options (allowed by, for example, multiplexers), instruction set etc.

Our framework allows the designer to describe the hardware part, create a program for the logic controller, generate external stimuli and collect and

analyze the outputs of the system [1]. As the framework is fully programmable and configurable, a suitable search algorithm can be utilized either to optimize or even automatically design not only the program for the controller but also the hardware architecture. The goal of research, which is reported in this paper, was to remove some constrains given on the hardware modules in the original version of the framework, and demonstrate that more general problems can be solved. The proposed solution is based on extending linear genetic programming (LGP) to concurrently evolve the program for the logic controller and the hardware architecture.

The proposed approach can be classified as a combination of genetic programming and evolvable hardware. We believe that our approach is new and unique; however, some common features can be identified with conventional hardware/software co–design based on evolutionary algorithms [2–4], co-evolution of programs and cellular MOVE processors [5], and genetic parallel programming (GPP) [6]. While GPP enables to automatically map a problem on parallel resources (multiple ALUs) in order to evolve efficient parallel programs, our method is more hardware oriented which allows for optimizing low-level properties of the underlying digital circuits.

In summary, the main contributions of this paper are as follows: (1) We propose an extension of our previous framework [1] in which a more general reconfigurable hardware architecture is supported. (2) We validate the proposed method of HW/SW concurrent evolution using 3 test problems. (3) We show that both general purpose solutions and application specific solutions can automatically be evolved on the proposed platform when suitable constrains are formulated.

The rest of the paper is organized as follows. Section 2 presents main features of the evolvable HW/SW platform. Section 3 is devoted to extending our platform by new features, particularly by relaxing some constraints on the organization of hardware modules. In Section 4, test problems are formulated, experimental setup is defined, and finally, obtained results are presented. Conclusions are given in Section 5.

2 Previous Work

In our previous work [1], we proposed a framework capable of concurrent evolution of HW and SW for application specific microprogrammed systems. This section will briefly summarize some basic terms so they could be used in the following sections. The framework is responsible for evolving the HW architecture and appropriate SW part as well as for providing the interconnections of the architecture with environment by providing inputs and consuming outputs.

2.1 Hardware

As can be seen in Fig. 1, the HW part is composed of a configurable datapath which is controlled by a microprogram. The components drawn in gray are responsible for instructions decoding and instruction pointer manipulation. This

part is fixed and does not udergo the evolution. The parts affected by the evolution are the registers and the modules. The registers are connected to modules by a set of multiplexers composed in such way that every register can be connected to every module. The connections from module outputs to the registers are realized by a set of decoders.

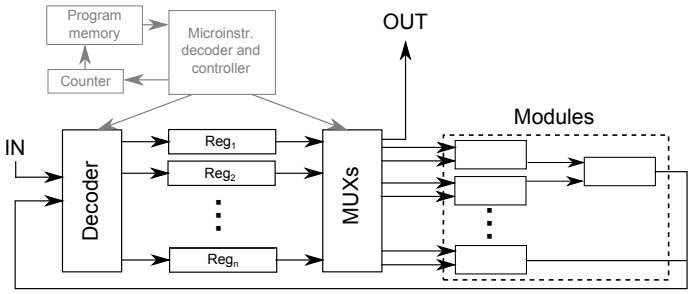

Fig. 1. HW architecture

Registers. The number of registers available in the architecture remains constant during the evolution. Their bit widths, however, can be changed by the genetic operators. Thus the width of the register can vary from 0 to a maximal value specified by the user. When the bit width of the register is set to zero, the register is considered unused, because it does not affect program execution.

Modules. Modules can be thought of as black boxes with inputs and outputs realizing an arbitrary function. Formally the module is defined as a 6–tuple

$$M =< n_i, n_o, a, p, d, f_o >, \tag{1}$$

where n_i is the number of module inputs, n_o is the number of outputs, a is the area used by the module, p is its power consumption, $d : \mathbf{D}^{n_i} \rightarrow \mathbf{D}$ is the function specifying the processing delay and $f_o : \mathbf{D}^{n_i} \times \mathbf{Q} \rightarrow \mathbf{D}^{n_o}$ is the output function. $\mathbf{D}$ denotes a user chosen data type and $\mathbf{Q}$ is the set of module's possible internal states. The whole HW part is described by the following components:

$\mathbf{i}$	the number of inputs
$\mathbf{o}$	the number of outputs
$\mathbf{R} = \{r_1, r_2, ...r_r\}$	a set of registers
$\mathbf{w} : \mathbf{R} \rightarrow \mathbb{N}$	a funcion setting widths of the registers
$\mathbf{A} = \{M_1, M_2, ...M_m\}$	a set of available modules
$\mathbf{u} : \mathbf{A} \rightarrow \{0, 1\}$	a function specifying module utilization

2.2 Software

Each program is composed of instructions $i_1, i_2, ..., i_s$, where s is the program size. An instruction can be composed of several microinstructions which get executed in order in which they are specified in the instruction. The internal representation of the microinstruction is depicted in Fig. 2. The header contains an operation code specifying the type of the microinstruction and a mask defining the module usage. Then a constant may be specified, which is used by certain types of microinstructions. Finally, there are input connections ($I_{k,l}$) and output connections ($O_{k,l}$) specifications for individual modules used by the microinstruction. More detailed information can be found in the aforementioned article [1].

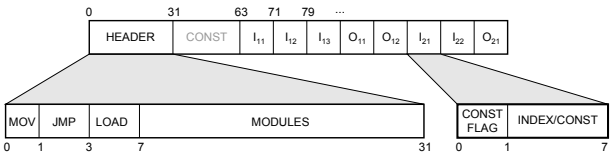

Fig. 2. Microinstruction format

3 Proposed Extensions

The architecture described in previous section had several limitations that didn't allow the evolution to create all useful compositions of modules. As the modules were arranged in parallel to each other, the only possible way of using the outputs of one module in another module was storing the results of one module in registers and passing them to another module in the following instruction. Although this method allowed the modules to use the outputs of previous modules, such connection was quite resource consuming because of the registers needed. Therefore, the need to pass the values between modules directly arose. Another issue we came across was the inability of the framework to process the inputs sequentially by one individual and in parallel by another individual. This issue has been addressed by introducing a new type of module that can be involved in instruction processing and allows the architecture to load virtually an arbitrary number of inputs inside one instruction.

3.1 Evolvable Hardware Topology Related Changes

There are several possible ways of implementing this functionality. At first, the possibility to create hardwired connection between modules was considered. However, such solution would limit the evolved architectures so that one fixed architecture would be used during the whole program execution. Therefore, the

decision to let the instructions choose the interconnections between modules was made. Such a solution allows the HW architecture to vary during the program execution to fulfill current needs.

Then it was crucial to choose a suitable encoding of variable topology. The proposed solution exploits the encoding used in Cartesian Genetic Programming (CGP).

Modules Order Encoding. Similarly to CGP, the modules are organized in one row and n_c columns, where $n_c = m$. As the order of modules is not defined, it is necessary to introduce the ordering into the chromosome, in our case by permutation μ of the set $\{1, 2, ..., m\}$.

There are many ways to encode such a permutation in the chromosome (e.g. [7]). Finally the encoding proposed in [8] has been chosen due to its properties regarding genetic operators usage and fast evaluation.

This encoding represents the permutation $i_1, i_2, ..., i_m$ of the set $\{1, 2, ..., m\}$ by an inversion sequence $a_1, a_2, ..., a_m$. In this sequence, each a_j denotes the number of integers in the permutation which precede j, but are greater than j. For example, if the original permutation is $3, 2, 4, 1$, the inversion sequence would be $3, 1, 0, 0$. In this case the value 3 at the 1st position in the inversion sequence means that there are 3 values in the original permutation that precede the value 1 and are greater than the value 1. After generating the inversion sequence the genetic operators can be applied on it in common manner and then the new prescription for module ordering can be easily generated from the inversion sequence. More details can be found in [8].

Instructions Related Changes. The changes of the HW part of the architecture require some additional changes in the SW part. The changes are in allowed ranges of inputs and outputs during their initial random generation or mutations. In the previous version of the framework, the inputs were allowed to be constants or register indices. To support the interconnections between individual modules the latter one has to be changed, in order to enable the connection of the input either to a specific register or to an output of a module which precedes the current module in the module order specified by the HW part. Therefore, the parameter $n_{in} \in \mathbb{N}$ specifying the connection is generated from the interval $< 0, n_{avail})$. The number of available connections n_{avail} for an input of module M_i can be computed as

$$n_{avail} = r + \sum_{j=1}^{i} n_{oj}, \tag{2}$$

where r is the registers count and n_{oj} is the number of outputs of module M_j. This change in generating of the connections imposes also the change in instruction execution. When the instruction is executed, used modules are evaluated one by one in order specified by the chromosome. During the module execution the available inputs are limited to registers and outputs of preceding modules. If

any of the preceding modules used by the instruction is disabled, the parameter specifying the input connection can be greater than the number of inputs actually available. This issue is addressed by performing the modulo operation using the actual number of available inputs. That means the instruction stays valid even after one of the modules used is disabled (e.g. by mutation). However, the connection will probably point to another module output or register. This way of instruction execution also ensures program validity when the order of modules is changed.

The last change imposed by the change of the framework is related to the outputs. In previous version of the framework, module outputs had to be connected to registers. Considering the possibility to connect the module output directly to another module input, there is no need for the output to be connected to a register. It is now possible for an output to be specified as 'no reg', which ensures the output value can be used by other modules, but will not be stored to a register.

3.2 Input Modules

As stated above, the previous version of the framework did not support the simultaneous evolution of individuals with sequential and parallel processing of the inputs. This was due to the fact that the number of inputs had to be constant. However, we came across some experiments, where this constraint imposed serious limitations. For example, when there was a possible solution, that could process four input values at once, and there were only two inputs available, the four values had to be loaded into registers before they could be processed. Increasing the number of inputs would not help in this case as e.g. a sequential solution using only two inputs would not be possible.

This issue has been addressed by introducing a new module type. According to the convention specified by formula 1, the input module is defined as

$$M_i =< 0, 1, 0, 0, d_{in}, f_o > . \tag{3}$$

As each input module represents one input, k–input system could be modeled by instantiation of k such modules.

The user can create several input modules and group them to form a specified number of input groups. Then the sequences of the input values have to be supplied for individual groups. After that the framework will set all the input modules to point to the first input value of a respective input group sequence and when the input module is executed, all modules of the same input group are updated to point to the next input value. When all the inputs from a given input group are already processed during the simulation and an input module from such an input group is executed, it returns the default value specified for this input group (e.g. 0).

3.3 Problem Encoding and Search Method

A candidate solution is represented in the chromosome as a string of integers. The first part of the chromosome is devoted to the program which is encoded in

the LGP-like style [9] as a sequence of instructions, each of which consisting of several microinstructions (Fig. 2). The second part of the chromosome defines the hardware—the usage and bit widths of the registers, the usage of the modules and the μ permutation. It is ensured that the program stays valid independently of the HW architecture changes.

The initial population is generated randomly. The fitness is represented by a vector of components containing functionality, area and speed. The NSGA-II algorithm [10] is utilized because it allows for non-dominated sorting of candidate solutions and a multiobjective optimization is naturally supported. Selection is performed by a tournament method with base 2. A two-point crossover operates at the level of (micro)instructions in the software part and at the level of modules in the hardware part. Mutation modifies the specification of registers, modules or the program (microinstruction type and parameters).

4 Experimental Results

Several experiments were carried out to evaluate the proposed method. It is important to keep in mind that comparison with other methods can serve only as a rough assessment of how fast the proposed method is, because the proposed method evolves HW and SW part simultaneously and has, therefore, to explore larger search space than the methods evolving just a program. All the experiments utilize a slightly modified version of NSGA-II algorithm. When the individuals are compared, first, their functionality fitness component is compared. If the value of this fitness component is the same for both the individuals, the NSGA-II is carried out on other fitness components. Therefore the selection prefers the individuals with the highest functionality.

4.1 Newton-Raphson Division

This experiment was chosen mainly to verify the ability of the new version of framework to evolve solutions for iterative problems. We solved this task with a modified version of CGP in [11].

Problem Description. Newton-Raphson iterative division is an algorithm that finds the quotient of numbers N and D ($0.5 \leq D \leq 1.0$), iteratively. The main principle of this algorithm lies in finding the reciprocal of the divisor D and then multiplying it by N to find the desired quotient. The iterative expression for finding the reciprocal is

$$X_{i+1} = X_i + X_i(1 - DX_i) = X_i(2 - DX_i) \tag{4}$$

This experiment was limited to finding the reciprocal of D such as in [11]. The parameters of LGP used for this experiment are listed in Table 1. The fitness function is defined as

$$f_o = \frac{1}{\sum_{i=1}^{s} \sum_{j=1}^{n_{it}} \frac{|Y_{ij} - T_{ij}|}{|Y_{ij-1} - T_{ij-1}|}}, \tag{5}$$

where s is the number of different target reciprocals (randomly generated), n_{it} is the number of iterations, Y is an output value and T is the expected value.

Table 1. LGP parameters used for Newton-Raphson division

Parameter	Value
Population size	20
Max. generation count	100,000
Crossover probability	0.05
Mutation probability	0.7
Max. logical time	15,000
Max. program length	15
Modules used	2xADD, 2xMUL
s	10
n_{it}	10

Results. After performing 200 independent runs the results were analyzed. A solution was found in 17.5 % of runs and the computational effort needed to find a solution with 99 % probability is 2.2×10^7. That is quite an interesting result, as the computational effort of CGP was of the same magnitude, despite that CGP search space was significantly smaller. After detailed analysis of all the solutions it was found that all of them had a similar structure. The Newton–Raphson expression was always found in an expanded form $X_{i+1} = X_i + X_i - X_i D X_i$ and there was no solution which would utilize the constant 2. This result is not surprising as CGP [11] produced the same one.

4.2 Finding the Maximum

This experiment was chosen to verify the ability of the framework to find various (sequential and parallel) different solutions during one run of LGP.

Problem Description. The main goal is to find an architecture calculating the maximum out of 8 input values. There are no further constraints on the number of inputs or processing time. After providing all 8 input values, all successive values will be zeroes and the zero flag of the input module will be set, so the architecture can take appropriate action. The evolution parameters are listed in Table 2. In this experiment a new module type was involved. The comparator module (CMP) has two inputs and two outputs and when executed, it sends the smaller value to the first output and the greater one to the second output. The functionality fitness component is defined as the number of correct outputs from 16 semi–randomly generated 8–tuples.

Table 2. Evolution parameters used for the Maximum experiment

Parameter	Value
Population size	50
Max. generation count	20,000
Crossover probability	0.05
Mutation probability	0.7
Max. logical time	500
Max. program length	10
Modules used	8xIN, 8xCMP

Results. After performing 3,000 independent runs the results were analyzed. Out of the total number of 3,000 runs over 62 % have successfully found a solution, with the computational effort of 1,026,114. The evolution was able to find various completely different solutions, including sequential and parallel solutions. We have sorted the results by their area fitness and speed fitness. The fitness values were scaled to range $< 0, 100 >$ for better readability. The nondominated solutions are depicted in Fig. 3.

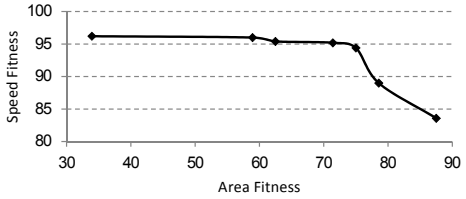

Fig. 3. The best fully–functional solutions of the maximum experiment

To show the progress of the evolution, two subsets of runs were selected. The first subset contained only the runs, which led to a minimal area solution (the rightmost ones in Fig. 3). Maximal values of the speed fitness and the area fitness were considered and the average value for each generation was computed. Fig. 4 shows that both speed fitness and area fitness grow rapidly during initial generations and then drop. This is implied by the fact that the individuals with higher functionality override other individuals even if they have higher speed and area fitness. When the functionality reaches a satisfactory level, the area fitness starts to grow, while the speed fitness still decreases. This corresponds to the expected trade-off between the area and speed.

The second subset is composed of the runs, that led to solutions with maximal speed. Fig. 5 shows that the speed fitness grows as expected, but after approximately 10,000 generations the area fitness also grows, so it appears there is no trade-off. After the investigation of the results we found out that the individuals tend to use more resources than needed at the beginning and the resources are optimized during later generations. The trade-off, however, still exists.

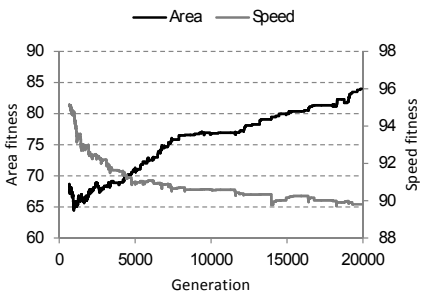

Fig. 4. Fitness progress for minimal area solutions

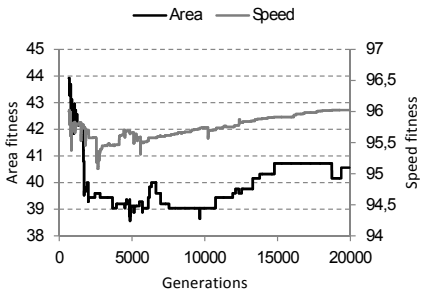

Fig. 5. Fitness progress for maximal speed solutions

Another interesting fact discovered during the analysis was that some solutions were general and could process an arbitrary number of values, whereas other solutions were limited to 8 input values. The general solutions were likely to appear among the solutions with smaller area, as the evolution had to develop a loop to process all the inputs, whereas large area solutions could process all the inputs by one instruction and then output the result.

4.3 Parity

This problem was chosen because it is one of typical problems solved using various evolutionary circuit design techniques. Another reason was to find out, whether some modifications speeding up the evaluation could be used.

Problem Description. In this experiment, the goal is to find an architecture, which computes parity of the binary inputs provided. The parameters used for the experiment were the same as in the previous case, but the comparator modules were substituted by XOR modules. The functionality fitness component in this case was the number of correct outputs.

Results. After performing 3,000 independent runs of LGP we evaluated the results and found out that the computational effort is 2,358,430. That was quite interesting as this value is more than twice as large as in the previous experiment, though the problem is quite similar and XOR modules have just one output, whereas the comparator has two outputs.

After some investigation we recognized that problem is significantly influenced by the definition of the fitness function. Because the XOR function gives just two possible results for each input combination (i.e. 0 or 1), even bad solutions can get quite high fitness. For example, when the output is zero all the time during the simulation, half of the input combinations are evaluated as correct. Therefore the right solution has to have a relatively high fitness value before it is considered as better than some of the bad solutions. Despite these problems

many solutions with various area/speed trade-off were found. The parameters of the best fully–functional solutions are depicted in Fig. 6.

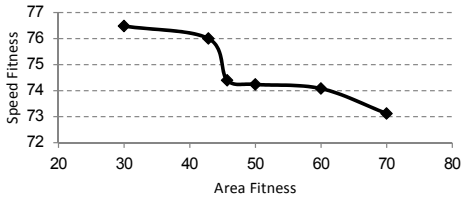

Fig. 6. The best fully–functional solutions of the parity experiment

Fig. 7 shows one of evolved solutions which is fast and resource consuming, but optimized for $n = 8$ and does not process any subsequent inputs. On the other hand, the solution depicted in Fig. 8 is slower and less expensive. It is also a general solution, as the inputs are loaded in a loop until there are no more inputs available. The computational effort can hardly be compared with other evolutionary techniques as they usually do not use the XOR module, but try to force the evolution to compose the solution from NAND and NOR modules. Such comparison will be one of our goals in the upcoming research.

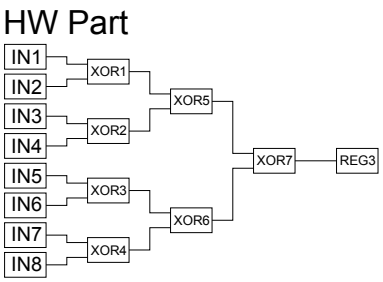

Fig. 7. Parallel solution for parity

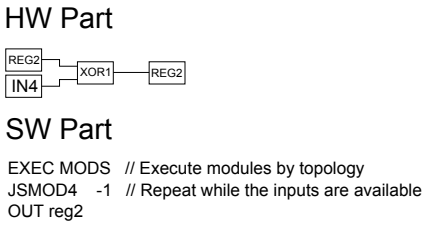

Fig. 8. Sequential solution for parity

5 Conclusions

In this paper, we extended our platform for development of small application-specific digital embedded architectures by supporting variable module interconnections and multiple input reading mechanisms. LGP was used to evolve a program for the controller together with a suitable organization of hardware modules. The proposed extension was evaluated by evolving a simple iterative

division algorithm. An important conclusion is that the platform can automatically synthesize multiple implementations, including a purely sequential solution and highly optimized parallel solutions, for a given specification

In our future research, we will deal with more complex iterative problems and their evolution on the proposed platform. However, accelerating the whole design process will be the first inevitable step.

Acknowledgments. This work was supported by the Czech science foundation project 14-04197S, Brno University of Technology project FIT-S-14-2297 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

References

1. Minarik, M., Sekanina, L.: Concurrent evolution of hardware and software for application-specific microprogrammed systems. In: International Conference on Evolvable Systems (ICES), IEEE Computational Intelligence (April 2013) 43–50
2. Dick, R.P., Jha, N.K.: Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems. IEEE Trans. on CAD of Integrated Circuits and Systems **17**(10) (1998) 920–935
3. Shang, L., Dick, R.P., Jha, N.K.: Slopes: Hardware-software cosynthesis of low-power real-time distributed embedded systems with dynamically reconfigurable fpgas. IEEE Trans. on CAD of Integrated Circuits and Systems **26**(3) (2007) 508–526
4. Deniziak, S., Gorski, A.: Hardware/software co-synthesis of distributed embedded systems using genetic programming. In Hornby, G., Sekanina, L., Haddow, P., eds.: Evolvable Systems: From Biology to Hardware. Volume 5216 of LNCS. Springer Berlin Heidelberg (2008) 83–93
5. Tempesti, G., Mudry, P.A., Zufferey, G.: Hardware/software coevolution of genome programs and cellular processors. In: First NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2006), IEEE Computer Society (2006) 129–136
6. Cheang, S.M., Leung, K.S., Lee, K.H.: Genetic parallel programming: design and implementation. Evol. Comput. **14**(2) (2006) 129–156
7. Goldberg, D.E., Lingle, R.: Alleles, loci, and the traveling salesman problem. In: Proc. of the International Conference on Genetic Algorithms and Their Applications, Pittsburgh, PA, Lawrence Erlbaum Associates, Publishers (1985) 154–159
8. Üçoluk, G.: Genetic algorithm solution of the tsp avoiding special crossover and mutation. Intelligent Automation & Soft Computing **8**(3) (2002) 265–272
9. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer Verlag, Berlin (2007)
10. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. on Evolutionary Computation **6**(2) (April 2002) 182–197
11. Minarik, M., Sekanina, L.: Evolution of iterative formulas using cartesian genetic programming. In Knig, A., Dengel, A., Hinkelmann, K., Kise, K., Howlett, R., Jain, L., eds.: Knowledge-Based and Intelligent Information and Engineering Systems. Volume 6881 of LNCS. Springer Berlin Heidelberg (2011) 11–20