DE GRUYTER OPEN
RIGA TECHNICAL UNIVERSITY
Electrical, Control and Communication Engineering
ISSN 2255-9159 (online)
ISSN 2255-9140 (print)
July 2016, vol. 10, pp. 5–12
doi: 10.1515/ecce-2016-0001
https://www.degruyter.com/view/j/ecce

Automated Metabolic P System Placement in FPGA

Darius Kulakovskis (*Ph.D. student, Vilnius Gediminas Technical University*),
Dalius Navakauskas (*Professor, Vilnius Gediminas Technical University*)

***Abstract* – An original Very High Speed Integrated Circuit Hardware Description Language (VHDL) code generation tool that can be used to automate Metabolic P (MP) system implementation in hardware such as Field Programmable Gate Arrays (FPGA) is described. Unlike P systems, MP systems use a single membrane in their computations. Nevertheless, there are many biological processes that have been successfully modeled by MP systems in software. This is the first attempt to analyze MP system hardware implementations. Two different MP systems are investigated with the purpose of verifying the developed software: the model of glucose–insulin interactions in the Intravenous Glucose Tolerance Test (IVGTT), and the Non-Photochemical Quenching process. The implemented systems' calculation accuracy and hardware resource usage are examined. It is found that code generation tool works adequately; however, a final decision has to be done by the developer because sometimes several implementation architecture alternatives have to be considered. As an archetypical example serves the IVGTT MP systems' 21–23 bits FPGA implementation manifesting this in the Digital Signal Processor (DSP), slice, and 4-input LUT usage.**

***Keywords* – Biological system modeling; Chemical processes; Digital signal processors; Field programmable gate arrays; Fixed-point arithmetic.**

I. Introduction

Metabolic P (MP) systems evolved from and are based on membrane computing or P systems [1]. They were first proposed by Manca in 2005 [2]. Unlike P systems, MP systems use a single membrane in their computations. Any substances defined in a MP system are moved through this membrane or transform from one to another at a certain rate. These systems are inspired by a biological process known as metabolism that is crucial for the survival of living organisms. MP systems can model a variety of processes, periodic or not, and produce approximated substance amounts after each reaction step.

MP system is a discrete dynamical system which can be described by a construct known as MP graph [3]. It is a set of reactions, fluxes which regulate these reactions and determine their speed and other parameters or constants. The reaction representation way is very similar to that of chemical reactions. Therefore, it is easy to interpret for the scientists of different fields such as chemistry and biology. Because of this, required MP system reactions can be more easily derived from the reactions of corresponding chemical or biological processes.

Many processes have been modeled by MP systems, including Belousov-Zhabotinsky reaction [4], Lotka-Volterra dynamics, Susceptible-Infected-Recovered epidemic [5], the circadian rhythms, the mitotic cycles in early amphibian embryos [6], Pseudomonas quorum sensing model [7], the lac operon gene regulatory mechanism in glycolytic pathway [8]. Recently, Goldbeter's mitotic oscillator was modeled by MP system [9]. Of course, application of MP systems is not exclusive only to biological and chemical processes but can be used to model almost any kind of dynamical processes, starting from a simple sine wave function [10].

Currently, there are two major implementation ways of MP systems. Both of them are based on implementation in software. The first one started as a *Psim* [11]. It is a simulation tool developed for MP system modeling that allows describing a system by means of graphs and simulating the system dynamics based on metabolic algorithm. *Psim* is developed using Java programming language and features an input GUI, which is used to construct MP graphs. An improved version of the software called MetaPlab was introduced in [12]. It features a new plug-in based architecture which makes the software more versatile and able to perform multiple tasks.

Another implementation of MP systems is an open-source MpTheory Java Library. It was developed by V. Manca and L. Marchetti and is available for download [13]. The provided Java objects can be directly used to model selected MP systems, and the library can also be used within MATLAB, GNU Octave, Mathematica and R computing environments.

Although MP systems can be implemented in software using specialized MetaPlab software or relatively simple implementation of MP formulae in MATLAB [14], it is also possible to directly implement them in hardware. P systems are known to have been implemented in hardware [15], [16], but so far there are no known similar implementations of MP systems.

In this article, an original MP system Very High Speed Integrated Circuit Hardware Description Language (VHDL) code generation tool that can be used to automate MP system implementation in hardware such as Field Programmable Gate Arrays (FPGA) is described. Preliminary work was done in [17]. The developed tool not only generates the VHDL code but also selects the optimal word length for fixed-point arithmetic according to user-desired accuracy. Although the developed tool optimizes the binary word length with the assumption that the performance of FPGA scales linearly with the word length, it is shown that that is not always the case. This is demonstrated by two separate cases of automated MP system implementation using the developed tool. Results of the automated implementation are discussed in the experimental section of the article along with key hardware implementation parameters such as design frequency, the number of FPGA resources used, and the calculation accuracy of the implemented design. The Digital Signal Processor (DSP) cell is regarded as the main limited resource of FPGA chip. It is the main element that performs the calculations. DSPs are used in a variety of applications as either a part of FPGA or a standalone chip [18].

II. Automation of MP System Implementation

The MP system VHDL code generation tool written in Python interpreted programming language was developed to convert MP systems described by a text file in JavaScript Object Notation (JSON) format to VHDL architecture. The generated VHDL code can be implemented as a stand-alone component or incorporated into bigger VHDL programs that are implemented in FPGA. This enables easier implementation of MP systems in hardware and builds on key MP system aspect of accessibility and ease of description. The MP system VHDL code generation tool can be used not only by engineers who understand MP systems but also by people working in different fields who might find the functionality provided by MP systems useful but too time- and resource-consuming to develop independently. The developed tool allows achieving faster implementation and lowers the time needed to deploy a selected MP system or even a multitude or variation of them.

MP system VHDL code generation tool is composed of four main parts, as shown in Fig. 1.

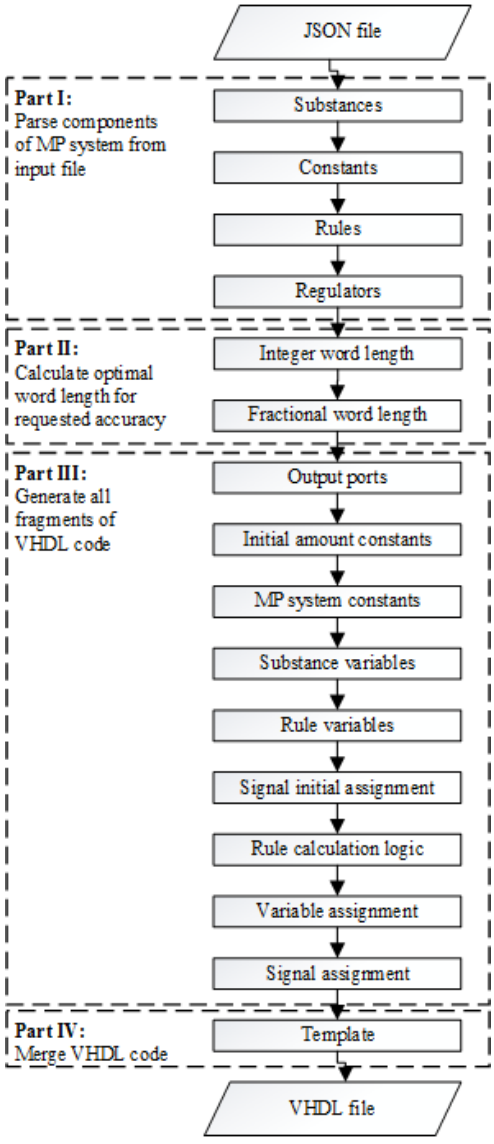

Fig. 1. MP system VHDL code generation tool.

The first part consists of an input file interpreter and a validator. The second part uses the input data and determines the binary word length needed to achieve the requested accuracy. The third part of the tool uses the determined word length and the input data to generate the key parts of VHDL code. Lastly, the fourth and final part merges the generated VHDL code fragments with a prepared template and produces a complete VHDL component.

As an input a file in JSON format is used to describe MP systems. The file includes entries for initial substance amounts, constants, rule logic, and regulator logic. Also, the developed tool accepts an input of a number of iterations the software should generate and the desired accuracy of the resulting fixed point calculations. The accuracy is used to determine the word length used in VHDL-fixed point library [19].

The first part of MP system VHDL code generation tool parses the input JSON format file. As the input file format is very similar to the Python dictionary data structure, the conversion of input data is straightforward. When the MP system data is loaded, a new class instance object is initialized according to provided substances, constants, rules, and regulators. The validation of the MP system defined in the input file is also performed along the way.

The second part of the MP system VHDL code generation tool determines the optimal word length that should be used to achieve the calculation accuracy requested by user. The word length is determined by an algorithm consisting of two parts, as shown in Fig. 2.

```
float_results = calculate_floating_point(...)
integer_bits, frac_bits = initial_values
# start integer part of the algorithm
# calculate results with initial values
fixed_results = calculate_fixed_point(...)
error = max(float_results - fixed_results)
# increment integer bits
integer_bits = integer_bits + 1
error_next = max(float_results -
fixed_results_next)
error_decreasing = True if error_next < error
if error_decreasing
    # increment until error is no longer decreasing
    while error < previous_error
        integer_bits = integer_bits + 1
        ... calculate error ...
else
    # decrement until error starts increasing
    while error == previous_error
        integer_bits = integer_bits - 1
        ... calculate error ...
# start fractional part of the algorithm
error = max(... with initial values ...)
bit_step = initial_bits/2
while bit_step != 1
    if error >= target_error:
        # increase until error is lower than target
        frac_bits = frac_bits + bit_step
    else
        # error is below target, try to optimize
        frac_bits = frac_bits - bit_step
    bit_step = bit_step/2
    all_results.append(error, fractional_bits)
# select closest result to target as final
final_frac_bits = closest_to_target(all_results)
```

Fig. 2. Pseudocode of algorithm to find the optimal word length.

Before starting the algorithm, MP system is modeled using floating-point arithmetic. The floating-point results are used as a reference point to calculate the error of fixed-point arithmetic. Then the first part of the algorithm tries to determine the required number of integer bits. As any decimal integer number can be represented in binary, there is a point where increasing the number of bits does not yield any improvement in the accuracy of the calculation. When the initial number of bits is selected, a system is modeled using that number of bits, and the error is calculated. Then the same operation is repeated using the integer word length greater by one. When the errors are compared, the algorithm determines whether the error is decreasing when the integer part of the word length is increasing. If it is true, this means that accuracy can still be improved by increasing the number of integer bits. If it is not true, the algorithm tries to lower the number of integer bits to optimize the system. When the optimal number of integer bits is found, the second part of the algorithm that tries to find the optimal number of fractional bits is started. Initially, a number of fractional bits that is a factor of two is selected, and then the binary search algorithm is used to get the number of fractional bits that yield the closest error value to the requested target.

The third part of the MP system VHDL code generation tool is the main VHDL code generation logic. It uses the input MP system data as well as determined the integer and fractional word lengths to generate the specific sections of VHDL code. The generation part consists of nine steps, as shown in Fig. 1.

Firstly, an output port must be defined for each substance participating in the provided MP system. The port direction is set to bidirectional (`inout`), and a signed fixed-point (`sfixed`) type is used. Then the constants used in this VHDL component are defined. They consist of substance initial amounts, taken from the substances part of JSON file, and other constants used in the MP system regulator expressions, defined separately in JSON file. Some variables must be defined for the behavioral process that calculates a step of MP system values on each clock cycle. Substance variables store the state of substance amounts after each calculation step (clock cycle). Rule variables store the calculation result for each MP system rule. When the VHDL process begins, firstly MP system substance signals are initialized by passing them the previously defined initial substance amount constants. Then the main calculation logic starts. For each MP system rule, a single line expression is generated that implements the regulator logic. Then, the calculated substance amounts for the current step are determined and assigned to substance variables. Finally, the calculated substances are passed to the signals that can be sent to the output.

The fourth and last part of MP system VHDL code generation tool uses a prepared template containing the rest of VHDL code, such as library declarations, utility port (step number, clock, start signal, etc.) declarations, rising clock edge detection, and the start and end markers of various VHDL code blocks. These code fragments are static for every implemented MP system and do not have to be generated. In addition, the template contains replacement fields that are substituted with generated VHDL code fragments by using the Python string formatting tools. The result of merging the template with generated code fragments is a complete VHDL component of a particular MP system provided as an input.

III. Study Description

A. Preliminaries of Original MP Grammar Representation by Data Structure

The input format of MP system VHDL code generation tool is a JSON data structure of original design. The JSON format was chosen because it is easy to read and convert in Python programming language, used by the generation tool. Also, it is a relatively simple format used by many applications, especially web pages that use JavaScript language. The structure of the JSON file closely mirrors the structure of MP systems represented by MP graphs. This feature makes it easier to convert MP systems from one format to another by automated tools or by manual human interaction.

The JSON file used in MP system VHDL code generation tool contains all required information to describe an MP system. As an example, an imaginary MP system described by the following MP graph can be represented in a JSON format used by the tool:

$$\begin{aligned} r_1 &: 2A \to 3B, \quad & \varphi_1 &= 5A + 8.5; \\ r_2 &: B \to A, \quad & \varphi_2 &= A + B - 1. \end{aligned} \tag{1}$$

The equation consists of two parts: left, and right. The left part of the equation represents the reactions *r*, and the left part represents regulators φ that determine the rate at which the reaction is happening. The MP system has two rule and regulator pairs (r_1 and φ_1, r_2 and φ_2), two substances (*A* and *B*), and three constants used in the regulators. This MP system represented by (1) can be rewritten in JSON format as is shown in Fig. 3.

```
{
  "substances": {
    "A": 1,
    "B": 2
  },
  "rules": [
    {
      "add": {"B": 3},
      "sub": {"A": 2},
    }, {
      "add": {"A": 1},
      "sub": {"B": 1},
    }
  ]
  "constants": {
    "c1": 5,
    "c2": 8.5,
    "c3": 1
  },
  "fluxes": [
    ["c1", "*", "A", "+", "c2"],
    ["A", "+", "B", "-", "c3"]
  ]
}
```

Fig. 3. An example MP system represented by JSON data structure.

When implementing the MP system in a new JSON format, it is split into four parts: substances, rules, constants, and regulators. Next, we will analyze them more thoroughly.

Substances are the main variables in MP systems. In this case, there are two substances: *A*, and *B*. These substances must have an initial amount that will be used at the start of MP system calculation. In JSON data structure, the keys of "substances" object represent the substances of MP system, and the objects' values represent the initial amount of corresponding substance.

Rules are the main MP system reactions that transform one or more substances into each other, introduce substances from the environment or expel them to the environment. In this case, there are two reactions. The first one transforms an amount of two of substance *A* into an amount of three of substance *B*. In other words, 2*A* is subtracted and 3*B* is added. This is represented in JSON data structure by a one-dimensional array that has a length of the number of rules in the MP system. It contains the two objects (each for one rule) with the numbers of the added and subtracted amount of each substance as key-value pairs.

Constants are the numerical values that do not change during the MP system iterative calculation process. Constants participate in the addition, subtraction or multiplication operations of MP system regulators. In JSON data structure, the constants are represented in a "constants" object by key-value pairs where the key is an arbitrary constant name and the value is the constant itself. This enables the separation of values from mathematical operations. In this case, there are three constants represented by two integer numbers and one fractional number. These are the only allowed types of constants as the MP system must be implemented in hardware, which has its limitations.

Regulators, sometimes also called fluxes, determine the rate at which the reactions are happening. Each MP system rule has its own regulator expression that contains mathematical operations using the previously defined constants and substances. In the JSON data structure, allowed mathematical operations are addition, subtraction, and multiplication. More complex operations could in principle be implemented, but hardware limitations must always be considered. For the purpose of this article, mentioned mathematical operations were sufficient. In JSON data structure, each regulator is represented by an array that contains each operand and mathematical operation of the regulator expression as a separate array element. The array must start and end with a substance or constant (valid mathematical operations cannot start or end with a sign), and elements must be separated by the addition ("+"), subtraction ("–"), or multiplication ("*") signs.

B. MP System Description – Case A

Two different Metabolic P systems were chosen and implemented in FPGA. This demonstrates the usage of the developed MP system VHDL code generation tool.

The first implemented system is a model of glucose–insulin interactions in the Intravenous Glucose Tolerance Test (IVGTT). It is an experimental medical procedure where a particular amount of glucose is injected intra-venously and the concentrations of glucose and insulin in blood are sampled at a frequent interval [20]. The IVGTT is used to better understand the interactions of glucose and insulin in human body. This test can help diagnose diabetes by observing the rate at which the glucose and insulin concentrations return to normal level.

One type of MP grammar of IVGTT model is used:

$$\begin{aligned} &r_1 : \varnothing \to G, \quad \varphi_1 = 0.6; \\ &r_2 : G \to \varnothing, \quad \varphi_2 = 0.12G + 1.6 \cdot 10^{-6} G^2 I; \\ &r_3 : \varnothing \to I, \quad \varphi_3 = 49.9 + 0.1G^3; \\ &r_4 : I \to \varnothing, \quad \varphi_4 = 0.84I. \end{aligned} \tag{2}$$

There are a total of four reactions in this model, each with its own flux regulator expression. Two substances participate in these reactions – glucose and insulin. Both of them are introduced to the system through a virtual membrane, which is represented in (2) by an empty set. The whole MP system structure must be considered and described when writing the JSON file for MP system VHDL code generation tool. JSON file for IVGTT system in (2) is shown in Fig. 4.

```
{
  "substances": {"G": 20, "I": 220},
  "rules": [
    {"add": {"G": 1}, "sub": {}},
    {"add": {}, "sub": {"G": 1}},
    {"add": {"I": 1}, "sub": {}},
    {"add": {}, "sub": {"I": 1}}
  ],
  "constants": {"f1c1": 0.6, "f2c1": 0.12, "f2c2":
0.0000016, "f3c1": 49.9, "f3c2": 0.1, "f4c1": 0.84},
  "fluxes": [
    ["f1c1"],
    ["f2c1", "*", "G", "+", "f2c2", "*", "G", "*",
"G", "*", "I"],
    ["f3c1", "+", "f3c2", "*", "G", "*", "G", "*",
"G"],
    ["f4c1", "*", "I"]
 ]
}
```

Fig. 4. MP grammar of IVGTT encoded in JSON data structure.

C. MP System Description – Case B

The second implemented MP system is the Non-Photochemical Quenching (NPQ) process. It is a photosynthetic phenomenon that determines how plants accommodate to various environmental light [21]. The NPQ process dissipates excess light, which can be absorbed by the plant in some environmental situations, using non-chemical ways (emitting heat). This process is very important for the survival of many plant species.

NPQ MP system has been iteratively improved many times and has numerous variations of MP grammar that have been tested is different cases [22]. For the research of automated implementation in FPGA, a single type of NPQ MP system was selected. The rules and regulators of the particular NPQ MP system are as follows [23]:

$$\begin{aligned} &r_1 : \mathrm{c} \to o + 12h + p, && \varphi_1 = \alpha_1 - \beta_1 c + \gamma_1 h - \eta_1 x + \upsilon_1 r + \rho_1 l; \\ &r_2 : c \to c + q^+, && \varphi_2 = -\alpha_2 - \beta_2 c - \gamma_2 h + \eta_2 x + \upsilon_2 l; \\ &r_3 : c \to c + f^+, && \varphi_3 = \alpha_3 - \beta_3 x + \gamma_3 l r^{-1}; \\ &r_4 : o \to c, && \varphi_4 = -\alpha_4 + \beta_4 o + \gamma_4 h - \eta_4 x + \upsilon_4 r + \rho_4 l; \\ &r_5 : h \to \varnothing, && \varphi_5 = \alpha_5 - \beta_5 c + \gamma_5 h - \eta_5 x + \upsilon_5 r + \rho_5 l; \\ &r_6 : p \to \varnothing, && \varphi_6 = \alpha_6 - \beta_6 c + \gamma_6 h - \eta_6 x + \upsilon_6 r + \rho_6 l; \\ &r_7 : x + 100v \to x + 100z, && \varphi_7 = -\alpha_7 + \beta_7 v; \\ &r_8 : y + h \to x, && \varphi_8 = \alpha_8 + \beta_8 y. \end{aligned} \tag{3}$$

There are many constants used in the NPQ MP system. They are as follows:

$$\begin{aligned}
&\varphi_1: \alpha_1 = 7.2389\cdot10^{-3}, \beta_1 = 0.238, \gamma_1 = 0.47722,\\
&\quad \eta_1 = 2.3954, \upsilon_1 = 6.3515\cdot10^{-5}, \rho_1 = 1.1321\cdot10^{-4};\\
&\varphi_2: \alpha_2 = 0.59811, \beta_2 = 2.5545, \gamma_2 = 32.921,\\
&\quad \eta_2 = 279.55, \upsilon_2 = 1.7732\cdot10^{-6};\\
&\varphi_3: \alpha_3 = 1.8208\cdot10^{3}, \beta_3 = 8.379\cdot10^{4}, \gamma_3 = 0.90737;\\
&\varphi_4: \alpha_4 = 0.1811, \beta_4 = 0.24184, \gamma_4 = 0.47722,\\
&\quad \eta_4 = 2.3954, \upsilon_4 = 6.3515\cdot10^{-5}, \rho_4 = 1.1321\cdot10^{-4};\\
&\varphi_5: \alpha_5 = 8.6799\cdot10^{-2}, \beta_5 = 2.8558, \gamma_5 = 5.7365,\\
&\quad \eta_5 = 28.755, \upsilon_5 = 7.6423\cdot10^{-4}, \rho_5 = 1.3585\cdot10^{-3};\\
&\varphi_6: \alpha_6 = 7.2389\cdot10^{-3}, \beta_6 = 0.238, \gamma_6 = 0.47722,\\
&\quad \eta_6 = 2.3954, \upsilon_6 = 6.3515\cdot10^{-5}, \rho_6 = 1.1321\cdot10^{-4};\\
&\varphi_7: \alpha_7 = 1.0734\cdot10^{-4}, \beta_7 = 3.3287\cdot10^{-5};\\
&\varphi_8: \alpha_8 = 5.2981\cdot10^{-7}, \beta_8 = 5.9746\cdot10^{-3}.
\end{aligned} \quad (3)$$

The MP grammar shown in (3) is much more complex than in the previous IVGTT MP system. Ten different substances participate in the reactions of this system. Each reaction has also a regulator expression that uses a list of constants derived from experimental results. In total, there are more than 30 different constants in this MP system. As a result, the JSON structure of this MP system, shown in Fig. 5, has much more elements than the structure of the previous MP system in Case A.

```
{
  "substances": {"C": 0.00397706,"H":
0.0006,"Q_cum": 0.01513153,"P": 3.74,"F_cum":
40,"V": 6.10763699,"Z": 0.02898116,"Y":
0.01170046493,"X": 0.00236685832,"O": 0.77989752},
  "rules": [
{"add": {"O": 1,"H": 12,"P": 1},"sub": {"C": 1}},
{"add": {"Q_cum": 1},"sub": {}},
                ...6 rules omitted...
  ],
  "constants": {
    "f1c1": 7.2389e-003, f1c2": -2.3800e-001,
              ...34 constants omitted...
  },
  "fluxes": [
    ["f1c1", "+", "f1c2", "*", "C", "+", "f1c3",
"*", "H", "+", "f1c4", "*", "X", "+", "f1c5", "+",
"f1c6"],
                ...7 fluxes omitted...
  ]
}
```

Fig. 5. MP grammar of NPQ encoded in JSON data structure (some parts omitted in order to save space).

D. Parameters for the Evaluation of MP Implementations

The MP systems described in Section IV are implemented automatically using the MP system VHDL code generation tool. Also, those MP systems were implemented in MATLAB software using a standard floating-point double-precision calculation. The implemented-in-hardware MP systems were evaluated using three main parameters: calculation accuracy (maximum error), execution speed (maximum frequency), and the amount of FPGA resources used.

To investigate the calculation accuracy, a reference point was needed. MATLAB software was chosen because it was already used in previous experiments [14]. The implementation in software can be used as a reference point as it uses the more precise floating-point arithmetic than the fixed-point FPGA implementation. Also, calculations in software do not have limitations such as the space and timing constraints. Maximum error across all available data points is calculated first. This means that it is ensured that the error will never exceed the desired one when generating VHDL code with fixed-point arithmetic. Even if a selected MP system has multiple outputs of which only one produces much greater error percentages, the total accuracy of the whole system will be dependent on the most inaccurate point. To determine the accuracy, the maximum error percentage is subtracted from a total of one hundred percent. When generating VHDL code, the number of steps generated also impacts the calculation of accuracy. This happens because the calculation of error is also only performed on the steps that will be used by FPGA to generate the output. If the number of steps is increased, there is a potential to encounter a point where the error will be higher than before.

For implementation in hardware, two different FPGA chips were chosen. Both FPGAs are from the Xilinx Virtex-4 family but have a different number of available resources. In Case A, for the smaller IVGTT MP system, a Xilinx Virtex-4 FPGA device with a model number of xc4vsx35 was used. This FPGA was sufficient for all investigated cases of the IVGTT MP system. For the second larger NPQ MP system in Case B, a bigger Xilinx FPGA device with a model number of xc4vsx55 was used. Both models of FPGA chips are functionally identical, except that the xc4vsx55 FPGA has a higher number of the components available. A bigger FPGA was needed for the NPQ MP system because of its complexity it did not fit inside the smaller FPGA model. The different number of components in selected FPGA models is shown in Table I.

TABLE I

NUMBER OF COMPONENTS IN SELECTED FPGA MODELS

FPGA component	**Total number of components in a chip**	
	xc4vsx35	**xc4vsx55**
Slices	15 360	24 576
4 input LUTs	30 720	49 152
DSP48s	192	512

For VHDL code implementation in FPGA, Xilinx ISE Design Suite software was used. The Xilinx software verifies the design and allows performing simulated VHDL design tests. The simulation is most of the time sufficient for obtaining the needed values of implementation performance such as used FPGA resources and design speed. Also, MP system calculation results can be obtained directly from the simulation. This speeds up the experimentation and allows easier result parsing.

The number of used FPGA resources as well as the design frequency is provided by Xilinx ISE Design Suite Synthesis Report. Calculation speed can be determined from the Timing Summary section of the report. It is measured using maximum design frequency value in MHz or minimum period value in ns. The number of used FPGA resources is provided in the Device Utilization Summary section of the Synthesis Report and is measured by providing the total number of both used cells and free cells.

IV. Experimental Results of FPGA Implementation

MP systems, described in Sections III.B and III.C, were implemented in FPGA using the VHDL code generated by the developed MP system VHDL code generation tool. The target error was set to be not more than 1 %. To achieve this accuracy, the optimal-word-length-finding algorithm was started. For the IVGTT MP system, 12 bits were needed for the integer part. After finding the optimal integer word length, a binary search algorithm was started to find the optimal number of bits for the fractional part. The process of finding the optimal number of bits is illustrated in Fig. 6. The algorithm selected 22 as the optimal number of bits for the fractional part. This means that in total, the word length of 34 bits plus the sign bit was used.

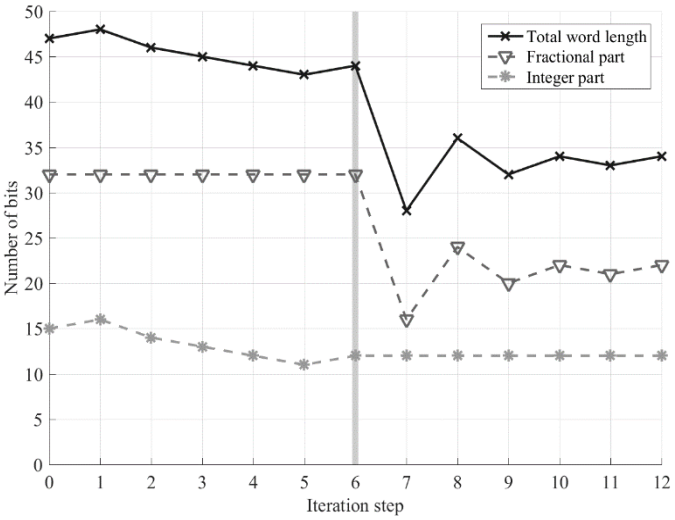

Fig. 6. Process of finding the number of bits for the IVGTT (Case A) implementation with the maximum error of 1 %. The gray bar emphasizes the change of optimization target – from the integer to fractional parts.

In the case of the NPQ MP system, which is more complex than the IVGTT, the word length required to achieve the same system accuracy was greater. When the target error was set to be not more than 1 %, the word-length-finding algorithm determined that the optimal number of bits for the integer part is 20 and for the fractional part is 25. This yields the total word length of 46 bits (one for the sign).

For both MP systems, the required number of steps to find the optimal word length was similar: 12 steps for the IVGTT, and 13 steps for the NPQ MP systems. The number of steps for determination of the integer and fractional parts is almost equal. Although this depends on the "guess" of the initial values, they were not specifically tailored to suit the selected MP systems. A starting point of a total of 48 bits (15 for the integer part, 32 for the fractional part, and 1 for the sign) was selected, which mirrors the capabilities of DSP48 cells of the selected FPGA.

It should be noted that the developed algorithm only searches for the minimum number of bits that achieve the requested accuracy. It does not, however, check whether the selected binary word length is effective when MP system is implemented in FPGA. It assumes that the use of x bits is always better than the use of $x + 1$ bits in terms of system performance, and always worse – in terms of system accuracy. However, as shown in Fig. 7, it is not always the case.

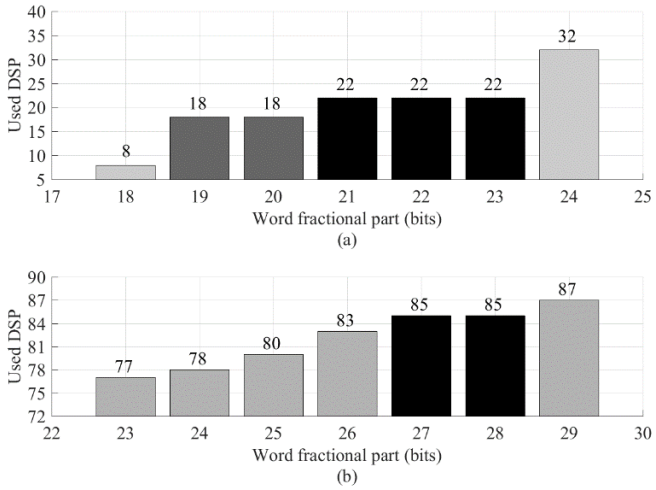

Fig. 7. Number of DSP cells needed to model IVGTT (a) and NPQ (b) MP systems with a particular number of bits for fractional part. The number of bits for the integer part of IVGTT system is always 12, while for NPQ system – 25.

For the IVGTT MP system, the automatically selected number of bits for the fractional part was 22 at the 1 % maximum error input value. When implemented in FPGA, this system was using 22 DSP cells. However, the same number of DSP cells was also used when the IVGTT MP system was implemented in FPGA using 21 or 23 bits for the fractional word length. Although the maximum error of the implementation with 21 fractional bits exceeds the targeted 1 %, the error of the implementation with 23 fractional bits is even better than the automatically selected one. The same phenomenon can be noticed at another place of Fig. 7. When the 19 or 20 bits fractional word length is used for the implementation of IVGTT MP system, the same number of 18 DSP cells is utilized.

As can be seen in Fig. 8a, when 19 bits are used for the fractional word length, the design speed is higher and less resources are used compared to the scenario when 20 bits are used. On the other hand, usage of 20 bits for the fractional part allows achieving a higher calculation accuracy. This outcome is very predictable and does not exhibit any special behavior: a longer word length in fixed-point arithmetic requires more FPGA resources but achieves a higher calculation accuracy.

A more interesting scenario can be seen in Fig. 8b. In three different cases, when using 21, 22, or 23 bits for the fractional part, the resulting IVGTT MP system implementations use the same number of 22 DSP cells. However, other parameters are not equal. The implementation with the highest number of bits (23) for the fractional part surprisingly uses the least amount of FPGA resources (slices and LUTs). In a very close second place is the implementation with 21 bits, and the automatically selected (closest to the 1 % error) implementation with 22 bits uses the most of FPGA resources. The implementation with the highest number of bits for the fractional part is also expectedly the most accurate. This leaves a single parameter where this implementation does not achieve the best results – execution speed. The implementation with 22 bits for the fractional part runs faster, even if not by a significant amount, than both the others. When 21 bits are used for the fractional part, there is no advantage in any implementation evaluation parameter.

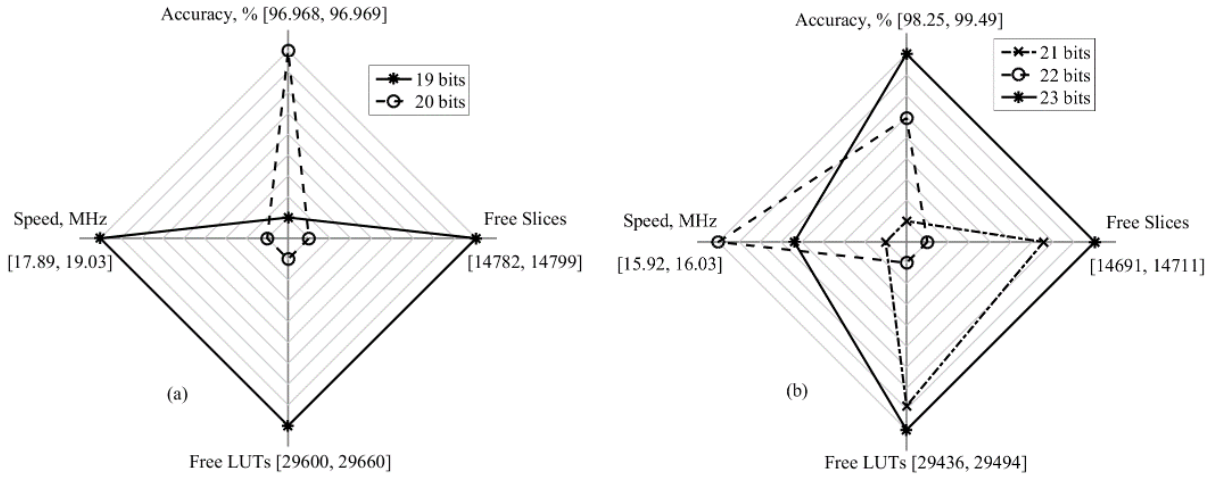

Fig. 8. Comparison of FPGA resources used and calculation accuracy when implementations of IVGTT MP system with different number of bits for the fractional part of binary word result in the same number of 18 (a) or 22 (b) DSP cells used.

This situation demonstrates that it is not always obvious how to choose the most appropriate parameters when implementing the MP systems in FPGA. Even though using a bigger word length for the fixed-point arithmetic can be expected to consume more FPGA resources, sometimes it can have a reverse effect. This may be influenced by the way the optimization is implemented in the VHDL code synthesizer provided with Xilinx ISE Design Suite software.

The case of NPQ MP system implementation confirms our findings – there also is a situation when the same number of DSP cells is utilized when using different word lengths, as shown in Fig. 7b. When using 27 or 28 bits for the fractional part, 85 DSP cells of the FPGA were used. Fig. 9 shows that, like in the case of Fig. 8a, using a higher number of bits achieves a greater accuracy but uses more resources.

In both cases, shown in Fig. 8a and Fig. 9, the difference in accuracy is not very large, maybe even insignificant, considering the fact that in principle MP systems are only approximations of real processes and, even if implemented with perfect accuracy, do not provide identical results to real data.

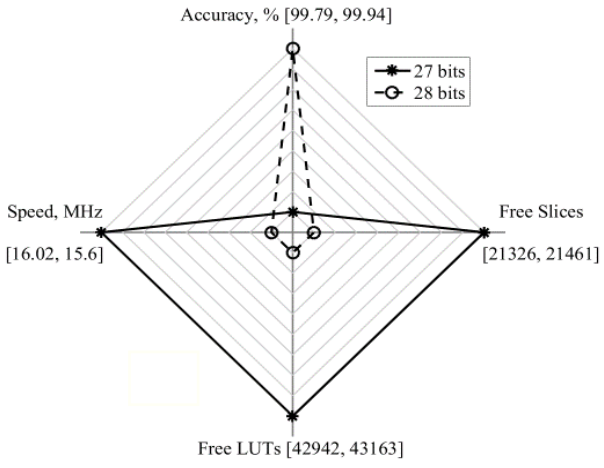

Fig. 9. Comparison of FPGA resources used and calculation accuracy when implementations of NPQ MP system with different number of bits for the fractional part of binary word result in the same number of 85 DSP cells used.

Considering this, in most cases it can be beneficial to select an implementation with slightly worse accuracy (0.001 % in Fig. 8b, and 0.15 % in Fig. 9) but more efficient resource usage. This can be especially important when hardware resources are limited or when implementing multiple systems.

According to the investigated automated MP system implementation cases, often it can be beneficial to not only determine a single optimal binary word length for use in fixed point calculations, but to further investigate the closest group of other options. The algorithm of MP system VHDL code generation tool is modified to take this into account. Firstly, it tries to select the word length that yields the results closest to the requested accuracy. Then the MP system is automatically implemented (synthesized) using Xilinx ISE Design Suite software. After that, FPGA performance parameters are obtained from the Design Report.

If DSP cells are considered to be the most valuable resource of an FPGA, the number of used DSP cells should be considered by the MP system VHDL code generation tool as the primary parameter. After implementing the MP system using the selected optimal word length and checking the number of DSP cells used, the algorithm can then select other closest possible word lengths and also implement them. For example, let's say that the optimal word length consists of x integer bits and y fractional bits. Then the closest other word lengths will use the same number of x integer bits, but two different numbers of fractional bits: $y - 1$ and $y + 1$.

If the MP systems using fixed-point arithmetic with the other selected closest word lengths are implemented and found to have utilized the same number of DSP cells, the MP system VHDL code generation tool can proceed to the comparison of other parameters. If the number of DSP cells used is not the same, the algorithm can keep using the previously selected optimal word length, considering the logic that DSP cells are the most important FPGA resource. Although the MP system VHDL code generation tool can find the word lengths with the same DSP cell usage, the selection of the best choice depends on the purpose of the system and the developers' choice.

V. Conclusion

1. A first valid automated MP system VHDL code generation tool for implementation in FPGA was developed:
   - the performance of the tool was demonstrated by automatically implementing the IVGTT and NPQ MP systems that reached an optimal word length in 12 and 13 steps correspondingly;
   - the automation of the tool is grounded on the use of Python programming language, and performs the search of the optimal binary word length (both the integer and fractional parts independently) for the fixed-point implementation;
   - a final decision to be done by the developer is supported by the tool that features the possibility of simulating the alternative implementations by VHDL synthesis software.
2. The experimental investigation of system implementations on FPGA revealed cases when bigger word lengths do not expectedly lead to a higher FPGA resource usage:
   - the cases with word length of fractional parts of IVGTT MP system (19 or 20 and 21 to 23 bits) and NPQ MP system (27 or 28 bits) manifest this in DSP cell usage;
   - the case of IVGTT system (word length of fractional part: 21 to 23 bits) manifests this in slice and LUT usage.

Future work includes investigation of different approaches to hardware implementation and further optimization.

Acknowledgment

This research was funded by a grant (No. MIP-083/2015) from the Research Council of Lithuania.

References

[1] G. Păun, "Computing with membranes," *J. of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000. https://doi.org/10.1006/jcss.1999.1693

[2] V. Manca, L. Bianco and F. Fontana, "Evolution and oscillation in P systems: applications to biological phenomena," *Membrane Computing*, pp. 63–84, 2005.

[3] V. Manca, "Fundamentals of Metabolic P Systems," *Handbook of Membrane Computing*, vol. 19, pp. 489–498, 2009.

[4] L. Bianco, F. Fontana and V. Manca, "P systems with reaction maps," *Int. J. of Found. of Comput. Sci.*, vol. 17, no. 1, pp. 27–48, 2006. https://doi.org/10.1142/S0129054106003681

[5] L. Bianco, F. Fontana, G. Franco and V. Manca, "P systems for biological dynamics," *Applicat. of Membrane Computing*, pp. 83–128, 2006.

[6] V. Manca and L. Bianco, "Biological networks in metabolic P systems," *Biosystems*, vol. 91, no. 3, pp. 489–498, 2008. https://doi.org/10.1016/j.biosystems.2006.11.009

[7] L. Bianco, D. Pescini, P. Siepmann, N. Krasnogor, F. J. Romero-Campero and M. Gheorghe, "Towards a P systems Pseudomonas quorum sensing model," *Memebrane Computing*, pp. 197–214, 2006. https://doi.org/10.1007/11963516_13

[8] A. Castellini, G. Franco and V. Manca, "Toward a representation of hybrid functional Petri nets by MP systems," *Natural Computing*, pp. 28–37, 2009. https://doi.org/10.1007/978-4-431-88981-6_3

[9] V. Manca and L. Marchetti, "Goldbeter's Mitotic Oscillator Entirely Modeled by MP Systems," *Membrane Computing*, pp. 273–284, 2010. https://doi.org/10.1007/978-3-642-18123-8_22

[10] V. Manca, and L. Marchetti, "Metabolic approximation of real periodical functions," *The J. of Logic and Algebraic Programming*, vol. 79, pp. 363–373, Aug. 2010. https://doi.org/10.1016/j.jlap.2010.03.005

[11] L. Bianco, V. Manca, L. Marchetti and M. Petterlini, "Psim: a simulator for biomolecular dynamics based on P systems," in *2007 IEEE Congr. on Evolutionary Computation*, CEC 2007, Singapore, 2007, pp. 883–887. https://doi.org/10.1109/cec.2007.4424563

[12] A. Castellini and V. Manca, "MetaPlab: a computational framework for metabolic P systems", *Membrane Computing*, pp. 157–168, 2009. https://doi.org/10.1007/978-3-540-95885-7_12

[13] L. Marchetti, *MpTheory Java Library* [Online]. Available: http://mptheory.scienze.univr.it/index.html. [Accessed: Feb. 23, 2016].

[14] D. Kulakovskis, "Application prospects of metabolic P systems," *Science – Future of Lithuania / Mokslas – Lietuvos Ateitis*, vol. 7, no. 3, pp. 285–290, 2015. https://doi.org/10.3846/mla.2015.784

[15] V. Nguyen, D. Kearney and G. Gioiosa, "An implementation of membrane computing using reconfigurable hardware," *Computing and Informatics*, vol. 27, no. 3, pp. 551–569, 2008.

[16] V. Nguyen, D. Kearney and G. Gioiosa, "A Region-Oriented Hardware Implementation for Membrane Computing Applications", *Membrane Computing*, pp. 385–409, 2010. https://doi.org/10.1007/978-3-642-11467-0_27

[17] D. Kulakovskis and D. Navakauskas, "Automation of metabolic P system implementation in FPGA: A case study", *in 2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, Riga, 2015, pp. 1–4. https://doi.org/10.1109/aieee.2015.7367289

[18] V. Skopis and I. Uteshevs, "Research in Adaptronic Automatic Control System and Biosensor System Modelling," *Elect., Control and Commun. Eng.*, vol. 8, no. 1, pp. 20–29, 2015. https://doi.org/10.1515/ecce-2015-0003

[19] D. W. Bishop. *VHDL-2008 Support Library* [Online]. Available: http://www.eda.org/fphdl/. [Accessed: Oct. 17, 2015].

[20] V. Manca, L. Marchetti and R. Pagliarini, "MP modeling of glucose-insulin interactions in the intravenous glucose tolerance test," *Int. J. of Natural Computing Research*, vol. 2, no. 3, pp. 13–24, 2011. https://doi.org/10.4018/jncr.2011070102

[21] V. Manca, R. Pagliarini and S. Zorzan, "A photosynthetic process modelled by a metabolic P system," *Natural Computing*, vol. 8, pp. 847–864, 2009. https://doi.org/10.1007/s11047-008-9104-x

[22] A. Castellini, G. Franco and R. Pagliarini, "Data analysis pipeline from laboratory to MP models," *Natural Computing*, vol. 10, pp. 55–76, 2011. https://doi.org/10.1007/s11047-010-9200-6

[23] A. Castellini, G. Franco and R. Pagliarini. *NPQ phenomenon* [Online]. Available: http://mplab.scienze.univr.it/external/natcomp/NPQ_stepwise_tab4.html [Accessed Oct. 28, 2015].

Darius Kulakovskis received the B.Sc. and M.Sc. degree in telecommunications engineering from Vilnius Gediminas Technical University in 2012 and 2014, respectively.
He is a Ph.D. student at the Department of Electronic Systems of Vilnius Gediminas Technical University, Lithuania. His main research interests are metabolic P systems and field programmable gate arrays.
Address: Vilnius Gediminas Technical University, Faculty of Electronics, Naugarduko str. 41–413, Vilnius, LT-03227, Lithuania.
E-mail: darius.kulakovskis@vgtu.lt

Dalius Navakauskas received the honour diploma of a Radio-Electronics Engineer (in 1992) and the following scientific degrees from Vilnius Gediminas Technical University: the M.Sc. degree in electronics in 1994, the Ph.D. degree in electrical and electronic engineering in 1999, the habilitation degree in informatics engineering in 2005, after which he received the title of a professor in 2008.
He is a Professor and the Chair of the Department of Electronic Systems of Vilnius Gediminas Technical University, Lithuania. His main research interests include computational intelligence, signal and image processing, and bioinformatics.
D. Navakauskas is a senior member of the IEEE, an active member of The IEEE Computational Intelligence and Signal Processing Societies, and currently serves as the Chair of the IEEE Lithuania Section.
Address: Vilnius Gediminas Technical University, Faculty of Electronics, Naugarduko str. 41–426, Vilnius, LT-03227, Lithuania.
E-mail: dalius.navakauskas@vgtu.lt