ATOMIC: Automatic Tool for Memristive IMPLY-based Circuit-level Simulation and Validation

Fabian Seiler
TU Wien, Institute of Computer Technology
Vienna, Austria
fabian.seiler@student.tuwien.ac.at

Nima TaheriNejad
Institute of Computer Engineering, Heidelberg University
Heidelberg, Germany
nima.taherinejad@ziti.uniheidelberg.de

ABSTRACT

Since performance improvements of computers are stagnating, new technologies and computer paradigms are hot research topics. Memristor-based In-Memory Computing is one of the promising candidates for the post-CMOS era, which comes in many flavors. Processing In memory Array (PIA) or using memory, is on of them which is a relatively new approach, and substantially different than traditional CMOS-based logic design. Consequently, there is a lack of publicly available CAD tools for memristive PIA design and evaluation. Here, we present **ATOMIC**: an **A**utomatic **To**ol for **M**emristive **I**MPLY-based **C**ircuit-level Simulation and Validation. Using our tool, a large portion of the simulation, evaluation, and validation process can be performed automatically, drastically reducing the development time for memristive PIA systems, in particular those using IMPLY logic. The code is available at https://github.com/fabianseiler/ATOMIC.

ACM Reference Format:
Fabian Seiler and Nima TaheriNejad. 2024. ATOMIC: Automatic Tool for Memristive IMPLY-based Circuit-level Simulation and Validation. In *Proceedings of ESWEEK 2024 Embedded System Software Competition (ESSC'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/0000001.0000001

1 MOTIVATION

With the stagnating computer performance, there is a rising emphasis on new computing technology and paradigms such as In-memory Computing (IMC) with memristors and the approximation of calculations or Approximate Computing (AxC). Memristor-based In-Memory Computing is one of the promising candidates for the post-CMOS era, which comes in many flavors. Processing In memory Array (PIA) or using memory, is on of them which is a relatively new approach, and substantially different than traditional CMOS-based logic design. Consequently, there is a lack of publicly available CAD tools for memristive PIA design and evaluation. We illustrated the typical development process for (approximated) algorithms for IMPLY, a common logic form for memristive computing, in Figure 1. When carried out manually, the process can take multiple hours to days, even for experienced researchers.

ESSC'24, September 29 – October 4 2024, Raleigh, NC, USA,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/0000001.0000001

Since there is a lack of available tools that help automate the development we propose **ATOMIC**, an automated Python tool based on the PyLTSpice framework. With this tool, a large portion of the validation, simulation, and evaluation process, as well as the processing and illustration of the data, is now completely automated. Since real-world memristors experience non-idealities such as variation, studying their effect is an important topic. The deviation of the memristor's resistive state is one of the most important variations that is often disregarded in many State-of-the-Art (SoA) papers. Therefore, we placed a strong emphasis on this topic to facilitate simulation and evaluation of memristive circuits under such variations. We approached this project in a generic fashion to allow compatibility with exact and approximated algorithms in various topologies and provide an environment that can be easily expanded to beyond what is presented here.

2 OVERVIEW OF ATOMIC

In this section, we will give an overview of the project structure and briefly explain the functions of the individual classes. The project was written in Python and consists of four classes: Functional Validation, Control Logic Generator, Simulator, and Plotter. We also implemented a logger in a singleton design pattern that is embedded in all classes to store information vital for debugging. An overview of the information and process flow is illustrated in Figure 2.

2.1 Files

2.1.1 Inputs Files. For this project, two files have to be configured beforehand. The first one is the "config" file where the essential information of the algorithm, used topology, required memristors, and expected outputs are stored in JSON format. The exact sequence of the algorithm has to be stored in an additional plain-text file. The specific format for each of the implemented topologies will be explained in more detail in Section 3.

2.1.2 Pre-configured Files. We stored important information such as the memristors, switches, and corresponding control voltages for each implemented topology in JSON files. Since the simulations are based on SPICE, we prepared circuit (.asc) and netlist (.net) files for each topology. The IMPLY-specific parameters are configured with values commonly used in the SoA. They can be adjusted quickly in the "IMPLY_parameters.json" file. All of the pre-configured files can be found in the "Structures" folder.

2.1.3 Outputs Files. Since ATOMIC automatically evaluates many steps in the design process we store intermediate and final results to allow for easy debugging. All intermediate and final results are stored in sub-folders of the "outputs" folder. More details on the

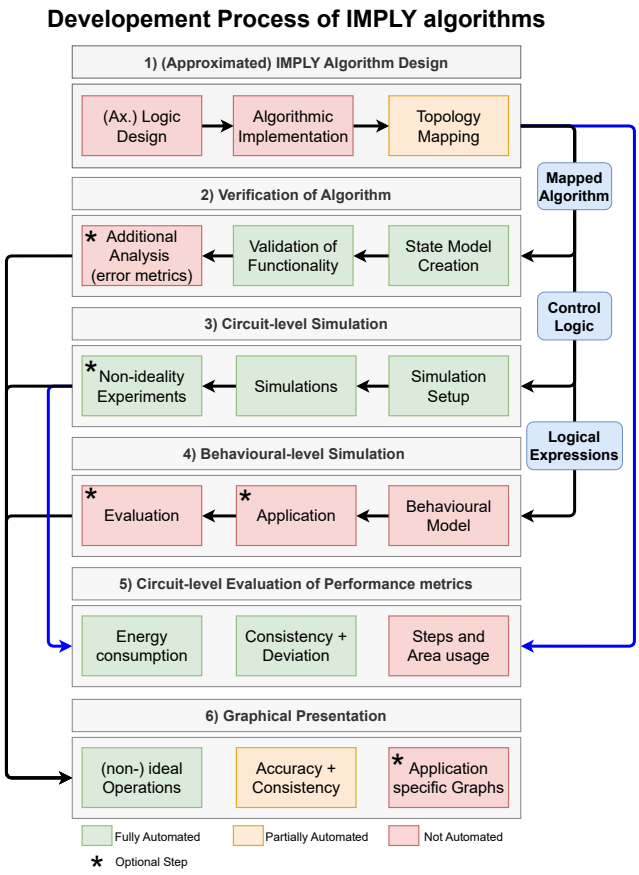

Figure 1: Design process for (exact and approximated) algorithms in memristive IMPLY logic. The blocks in green are fully, and the blocks in orange are partially automated in this project.

individual result files can be found in the class description that creates them.

2.2 Classes

2.2.1 Functional Validation. The first class in the pipeline is responsible for the creation of a state model and the validation of the functionality for the given algorithm. From the configuration file, the number of inputs and outputs are extracted, and logic vectors that represent each input combination are initialized. With the method `calc_algorithm`, each line of the algorithm files is read and processed. Depending on the topology, this method extracts information from each line and creates either IMPLY or FALSE operations. These operations are then applied to the state model in a vectorized fashion via `imply_op` and `false_op`. The operation and updated state model are stored in the "State_History.txt" file for later examination. At the end of the algorithm, the equivalence of the expected and simulated logic vectors is checked.

2.2.2 Control Logic Generator. This class is responsible for generating control logic in the form of PWM signals, that are then applied in the SPICE simulation. When initialized the class creates ".csv" for each memristor and switch that is used in the given algorithm and stores them in the "PWM_output" folder. The method `eval_algo` iterates over every step in the algorithm file and stores time steps for each memristor and switch individually. The parameters like cycle time and control voltages for different operations are read from the "IMPLY_parameters.json" file. After the last step the complete PWM signal is written in the previously created files.

2.2.3 Simulator. The simulator class is responsible for manipulating netlists, simulations with LT-SPICE, and calculating the energy consumption. When initialized, the parameters of the configuration and the topology file are extracted, and the netlist is selected with the `SpiceEditor` class from the PyLTSpice framework. With the `set_parameters` method, a list of parameters in the format `param_values:= list(memristor values, R_on, R_off)` is accepted as an input and the netlist is manipulated accordingly. But instead of changing our preset netlist, we store the intermediate netlist and the other manipulated simulation files in a temporary folder to allow for better debugging. With the method `run_simulation` a transient (as configured) simulation with SPICE is started. The resulting waveforms can be extracted with `read_raw` and saved as a file with `save_raw`. The energy consumption is calculated with `calculate_energy`. With the method `evaluate_deviation` we automated resistive deviation experiments that calculate different input combinations. For each input, the state deviation of memristors is varied, and the results are stored in the "Waveforms" folder. Since we are interested in the state of the memristors at the end of the algorithm, we store the resulting logic states of the output memristors in the "deviation_results" folder. This is also done with different combinations of resistive deviation.

2.2.4 Plotter. This class is responsible for post-processing the simulation data and illustrating the results. The state of individual memristors in an IMPLY algorithm can vary depending on the resistive state and its deviation. To visualize the impact of these deviations on the algorithm we implemented the method `plot_waveforms_with_deviation` that extracts the range of the waveforms and plots the result. An example is shown in Figure 3, where the deviation range is illustrated as the shaded area around the exact waveform. To analyze the functionality of algorithms with increasing deviation, the method `plot_deviation_scatter` was implemented. An example plot is shown in Figure 4, where the incorrect output states are colored red. `plot_deviation_range` calculates the range of the different outputs and compares them over increasing deviation. An example is shown in Figure 5. The resulting figures are stored in the folder "Images". To compare with other algorithms the output state ranges are stored in the file "deviation_range.txt".

3 HOW TO USE ATOMIC

3.1 Requirements

This project was developed for Python 3.12, LT-SPICE version 17.1.6.0, and PyLTSpice version 5.3.1. The required Python packages are summarized in the "requirement.txt" to allow for a fast setup.

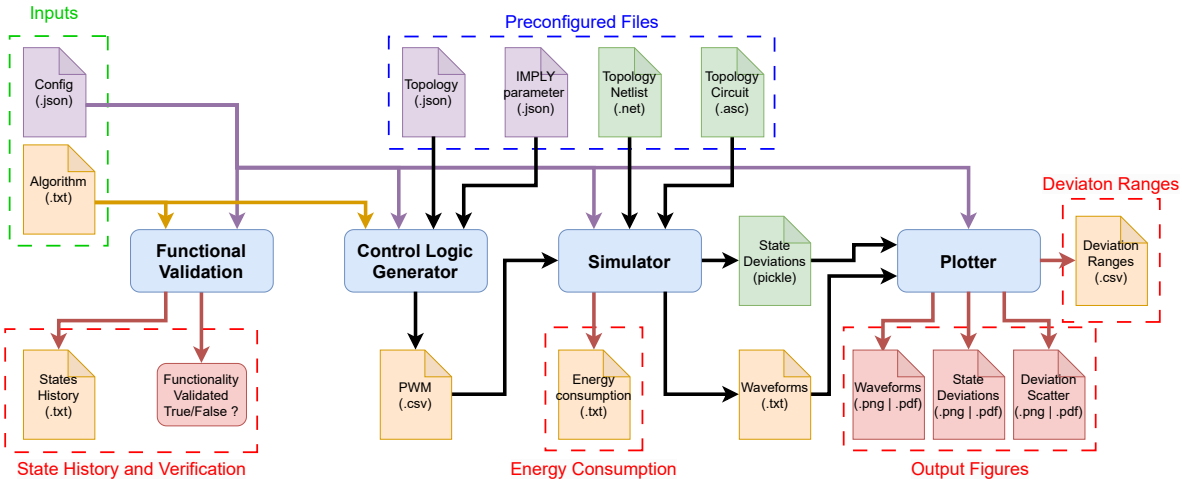

Figure 2: Overview of the ATOMIC pipeline and dataflow.

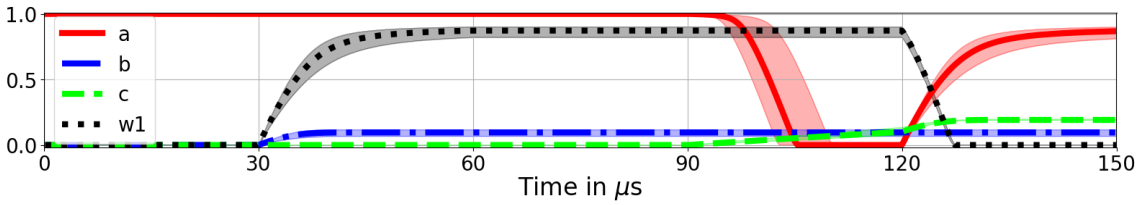

Figure 3: Example waveform with a deviation of ±20% illustrated as the shaded area.

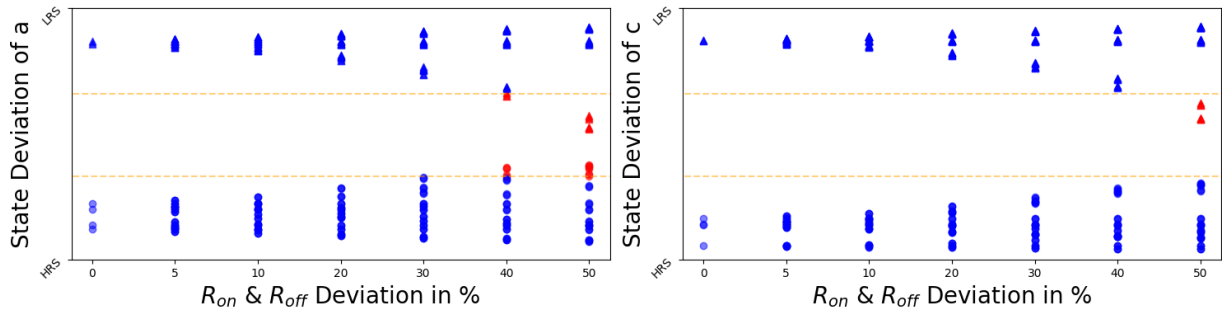

Figure 4: Scatter plot of output states with increasing deviation range. The markers in red are marked as incorrect results.

3.2 Setup algorithm file

Since there is no uniform way of writing down IMPLY-based algorithms, we propose a simple and easily extendable format for storing algorithms in various topologies. Our framework is currently compatible with the serial, semi-serial, and semi-parallel topologies but can be extended quite easily. Each section (depending on the topology) that can compute operations must be set to either FALSE, IMPLY, or NOP (No Operation) and the sections have to be separated by the "|" symbol. The FALSE operation can reset up to three different memristors, which is written in the form **F***j* **F***j*, *k* or **F***j*, *k*, *l* where *j*, *k*, and *l* correspond to the number each used memristor is given. The IMPLY operation is written as **I***j*, *k*, which is equal to the operation $M'_k = M_j \rightarrow M_k$ where memristor *j* implies memristor *k*. The memristor number resembles an enumeration of a list of used memristors (e.g. ["a", "b", "c", "w1"] have the numbers [0, 1, 2, 3] so the operation "**I**0,2" is equal to $c' = a \rightarrow c$). We implemented a few SoA IMPLY algorithms to give the user a reference on how the algorithms can be written in each topology.

3.3 Setup config file

The configuration file has to be created separately for each algorithm. It contains information on the algorithm's intended behavior and declares the use case of each memristor. Additionally, the expected output states have to be declared to allow for validation of

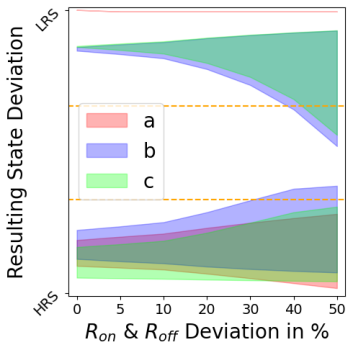

Figure 5: Range of the resulting states for each output over increasing deviation.

the algorithm's functionality via the state model or on circuit-level. We kept this configuration as general as possible to also allow approximated algorithms in various forms. We prepared a template here and in the project files to speed up the preparation. The unfilled template can be seen here:

```
{
"topology": "",
"algorithm": "",
"memristors": ["", "", "", ""],
"inputs": ["", "", ""],
"work": [""],
"outputs": ["", ""],
"switches": ["", "", "", ""],
"steps": ,
"output_states": {"": [0, 0, 0, 0, 0, 0, 0, 0],
                  "": [1, 1, 1, 1, 1, 1, 1 ,1]}
}
```

The topology name must be either be "Serial", "Semi-Serial", or "Semi-Parallel" and the algorithm should be the name of the prepared algorithm file. In "memristors" the name of all used memristors must be declared. If they are used as input, work, or output they have to be declared in the corresponding place. We note here that memristors can be used for both input and output, as well as work and output, as this is a common design property of SoA IMPLY algorithms. In "switches" each used switch has to be declared, as they can vary between algorithms. More information on the available switches and memristors for each topology can be found in the "Structures" folder. "output_states" is a dictionary that includes the name of the outputs and the expected bit vectors. The number of outputs is variable to allow for a more flexible design, which is necessary in this design space. More information and example algorithms can be found in the project.

3.4 Run the pipeline

We implemented two variants on how an algorithm can be evaluated. The first one is a Jupyter Notebook (`Pipeline.ipynb`) where each step of the pipeline can be executed individually. We suggest this method for the initial stages of the development so possible mistakes can be better detected. When only parts of the project are required we refer the reader to Section 2.2 for more information on the implemented classes. To run the whole pipeline (algorithm validation, simulation, deviation experiments, and illustration) we prepared the command:

`python Pipeline.py – config_file=CONFIG_FILENAME.json`

We configured additional flags so the user can customize various settings, which can be seen in the project. We implemented a few SoA algorithms and corresponding configuration files, to give some examples. To evaluate all the implemented algorithms at once, run the command: `python evalute_soa.py`

4 HOW TO EXTEND THIS PROJECT

4.1 Appending topologies or other structures

We built this project in a highly modular fashion to allow for easy extensibility. As interest in this area of research is increasing we also expect many new topologies and algorithms to be published. We marked areas where new topologies can be added in all implemented classes. It is also required to create a new circuit with the used naming convention for all circuit elements and to parameterize them like the implemented topologies. To append new topologies or other memristive structures to the functional validation module, add a new branch in `FunctionalValidation.calc_algorithm()` that converts the algorithm to IMPLY and FALSE commands. Since the control logic can differ drastically between topologies, the creation of PWM signals is highly specific. We recommend that a new method: `ControlLogicGenerator.write_timestep_TOPOLOGY` is created and the specifics are handled there. We again marked the spaces where branches should be added in all methods. As the simulator only manipulates the netlist and runs SPICE commands, not much change is required for new topologies. We note here that the `calculate_energy()` and `evaluate_deviation` functions are limited to three inputs by design since we only evaluated adder circuits. To run simulations with arbitrary inputs first prepare a list of parameter values and then utilize the `run_simulation` and `save_raw` methods to simulate certain inputs and store the resulting waveforms. In the plotter module all methods are generalized for an arbitrary number of output states. We note here that the figure size may have to be adjusted.

4.2 Other logic forms and memristor model

There exist various forms of logic for memristive circuits. To add other logic forms, the operational functions of the Functional Validation class must be extended to subject to the new logic. As the Control Logic Generator class is designed for IMPLY, another class that handles the generation of PWM signals would be the best option. The Simulator and Plotter can handle arbitrary logic forms.

To utilize another model for memristors, the sub-module in the selected topology has to be exchanged. As these models use different parameters, the method `Simulator.set_parameters()` and all the dependent methods must also be updated accordingly.