Cpppc Project: Microfluidic-Large-Scale-Integration (mLSI) Simulator

Mengchu Li

October 12, 2018

1 Introduction

Microfluidic Large-Scale Integration (mLSI) is a promising platform for biological/biochemical applications. An mLSI chip has a multi-layered structure consisting of micro-channels and micro-mechanical valves. The flow layer is responsible for fluid transportation and operation execution, and the control layer is responsible for valve actuation that manipulates the direction of the fluid-flow. Figure 1 shows a photo of an mLSI chip where control channels are filled with green food dye and flow channels are filled with red food dye.

MLSI is a developing research filed. Currently there is no simulation tool that can predicts the flow status on an mLSI chip. Different from existing microfluidic simulators, the main objective of which is to accurately predict the movement of small fluid particles, mLSI simulation should focus on the correlating valves and channels. More specifically, it should predict the channel status, i.e. whether a channel is blocked, vacant, or filled with fluids at a specific time point. In other world, mLSI simulation predicts the fluid behavior on a system-level.

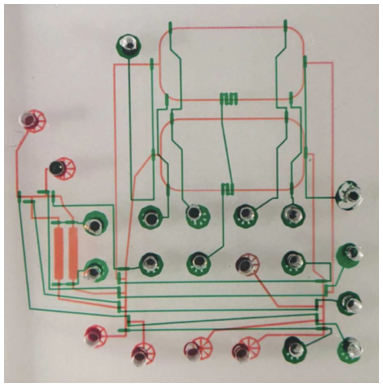
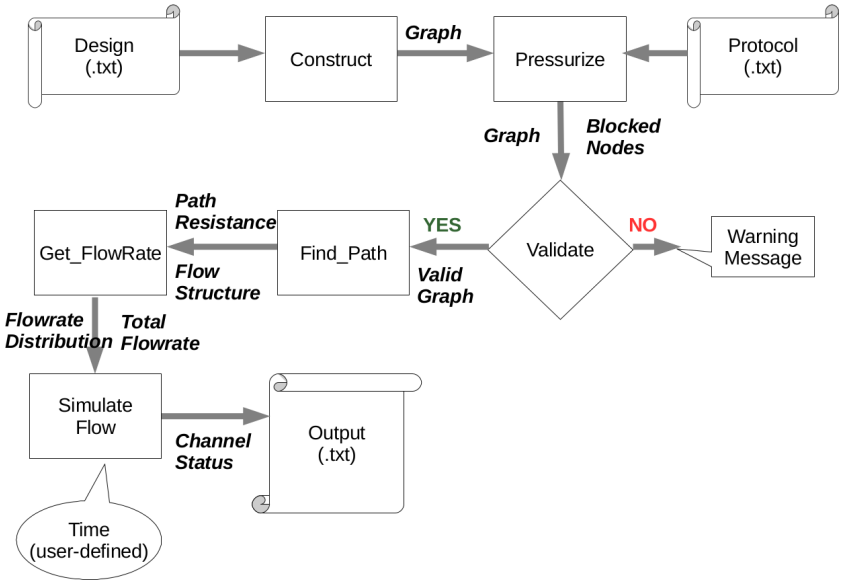


Figure 1: An mLSI chip for kinase activity applications.

Figure 2: System diagram of the Project



2 Overview of the Project

This project aims to develop a prototype of mLSI simulator. Formally:

Input

- A description of the physical structure of an mLSI chip (.txt file);
- A description of the application protocols (txt.file);
- A specific time point (a floating point number).

Output

- The channel status, i.e. whether it is blocked, vacant or filled with fluids, at a specific time point (txt.file).

Figure 2 shows the system diagram of the simulator. The simulator first takes a description of the chip *design* as its input, and *constructs* a **graph** structure consisting of nodes and edges. Each node may represent an inlet, an outlet, a valve, or a channel branching point of the mLSI chip, and each edge represents a channel segment that connects two nodes. After that, the simulator reads a description of application *protocols*, based on which some of the control channels will be *pressurize*d, and thus some nodes will be blocked. The **graph** and the **blocked nodes** are then fed to the next process that *validate*s whether

Figure 3: Code demonstration of the system diagram

```
int main(int argc, char* argv[]) {
  Graph basic_graph = construct(argv[1]);
  std::map<int, Graph::Node *> blocked_nodes = pressurize(argv[2], basic_graph);
  Graph valid_graph = validate(blocked_nodes, basic_graph);
  if (valid_graph.flag() == false)
    return 1;
  std::tuple<std::map<Graph::Edge, std::pair<Graph::Edge *, double>>, std::map<std::pair<Graph::Node*, Graph::Node *>, (
  std::pair<std::map<Graph::Edge *, double>,double> flow_collection = get_flow_rate(seq_par_res);
  std::cout<<"Please give the execution time: ";
  double time;
  std::cin>>time;
  std::map<Graph::Edge *, double> utilization = simulate_flow(flow_collection, valid_graph, time);
  output(valid_graph, seq_par_res, flow_collection, utilization, time);
}
```

there is any feasible flow paths on the chip. If there is no feasible flow path, the simulator will output a *warning message* and terminate. If there is one or more feasible flow paths, the simulator will generate a **valid graph** where blocked nodes and channels are removed. Based on the valid graph, the simulator will *find* the directed flow *paths* and calculate their resistance. The **flow structure** and the **path resistance** are then used to *get* the **flow rate distribution** of each edge as well as the **total flow rate** of the chip. Finally, the simulator *simulates* the fluid *flow* by outputting a text file describing the **channel status**.

Each rectangle in the system diagram represents a major function in the program, and the arrows going in to and going out from the rectangles are denoted with the input and the output of these functions, respectively.

Figure 3 demonstrates the implementation of the system diagram as codes.

3 Insight into the Program

The program is implemented based on a **graph** concept, which models the chip structure as well as the flow paths.

A graph consists of two essential members, namely nodes and edges. A graph is called *valid*, if it consists of a valid sequence of edges that connects an inlet node to its destined nodes.

From the algorithmic aspect, there are three essential functions based on graph parameters: *validate*, *find_path*, and *simulate_flow*. In the following I will explain these functions in more detail.

3.1 Validate

Figure 4 demonstrates the implementation of the validate function. The validate function takes two input parameters, namely a graph and a container of blocked nodes. A new graph object will be initialized based on the unblocked nodes in the graph. Once a node is blocked, the edge connected to it will also be blocked; and once an edge is blocked, the other node of this edge will also be

Figure 4: Code demonstration of the validate function.

```
Graph validate(std::map<int, Graph::Node *> & blocked_nodes, Graph & g){
  Graph valid_graph = initialize(blocked_nodes, g);
  int count = 0;
  while (count != valid_graph.nodes().size()){
    count = valid_graph.nodes().size();
    delete_invalid_nodes(valid_graph);
    delete_invalid_branch(valid_graph);
    delete_intermediate_node(valid_graph);
  }
  Graph::Node * departure = find_departure(valid_graph);
  if (departure == nullptr){
    std::cout<<"There is no open inlet and thus no feasible flow path."<<std::endl;
    valid_graph.set_flag(false);
    return valid_graph;
  }
  refine_edges(valid_graph);
  guide_direction(valid_graph, departure);
  return valid_graph;
}
```

Figure 5: Code demonstration of the find_path function.

```
std::tuple<std::map<Graph::Edge, std::pair<Graph::Edge*, double>>, std::map<std::pair<Graph::Node *, Gr
  std::map<Graph::Edge, std::pair<Graph::Edge*, double>> seq_res;
  std::map<std::pair<Graph::Node *, Graph::Node *>, double> par_res;
  std::map<std::pair<Graph::Node *,Graph::Node *>, std::pair<Graph::Edge *, double>> resistance = initi
  refine(resistance, seq_res, par_res);
  double total_resistance = resistance.begin()->second.second;
  return std::make_tuple(seq_res, par_res, total_resistance);
}
```

blocked, unless this node is at a channel branching point. What's more, if a node which is neither an inlet nor an outlet node, and it is connected to only one edge, there is no way for this node to form a valid flow path and thus it can also be considered as blocked. This can be referred to the *propagation of the blocked status*. The validate functions applies an iterative process to find all the blocked nodes and edges. Besides, if a node is connected to exactly two edges, it indicates that the node is an intermediate node of a longer edge. In this case, the delete_intermediate_node function will update the graph by merging the edges into a new edge and deleting the intermediate node.

Figure 6: Code demonstration of the simulate_flow function.

```
std::map<Graph::Edge *, double>
simulate_flow (std::pair<std::map<Graph::Edge*,double>,double> & flow_collection, Graph & g, double time){
  double flow_volumn = get_volumn(flow_collection.second, time);
  std::set<Graph::Edge *> processed;
  std::map<Graph::Edge *, double> utilization;
  std::map<Graph::Edge *, double> remain;
  int departure = find_departure(g)->index();
  int cur_node = departure;
  while (cur_node != 0){
    int next_node = 0;
    double cur_volumn = get_cur_volumn(flow_volumn, departure, cur_node, remain, g);
    for (auto & i : g.nodes()[cur_node].out()){
      check_capacity(cur_volumn, i, utilization, remain, flow_collection.first);
      processed.insert(i);
      next_node = find_next(cur_node, processed, g);
    }
    cur_node = next_node;
  }
  return utilization;
}
```

3.2 Find_Path

Figure 5 demonstrates the implementation of the find_path function. This function takes a valid graph as input parameter and outputs a tuple that specifies both the flow structure and the flow resistance of the edges. The solution is implemented in an iterative manner. First, if two edges share the same end nodes, it means that these two edges are in parallel. In this case, these edges can be considered as a wider edge with relatively smaller resistance. Thus, the simulator will merge the two edges as a new edge with re-calculated flow resistance. After merging the edges that are in parallel, there can be sequentially connected edges which did not exist in the input valid graph. Thus, the simulator will merge these sequentially connected edges and delete the intermediate node. This action might again introduce new edges that are in parallel, and thus the iteratively refinement process will only terminate when no more edges can be merged. In this process, we will keep updating the resistance of the edges, and record all the edges generate intermediately to keep a track of the flow structure.

3.3 Simulate_Flow

Figure 6 demonstrates the implementation of the simulate_flow function. This function takes the records of the flow rate and the graph as its inputs, together with a user defined time parameter. Based on the time parameter, the simulator calculates the total volume of fluids that can be injected into the chip in the given time interval. Based on the flow structure, the transportation paths of the fluids are known. Thus, the simulation starts from the departure node and

the edge connected to it. If the the volume of the fluids surpasses the capacity of the channel represented by this edge, we know that the fluids will proceed to the next edges connecting to this edge. If an edge is connected to more than one channels, the distribution of the fluids into these edges will follow the distribution of the flow rates. This process repeats until the remained fluids are not enough to fill a given edge. In this case, we record the percentage of the fluids to the capacity of the edge.

4 Conclusion and Future Work

The proposed simulator is able to simulate the fluid flow on mLSI chips at a given time point based on the chip design and the application protocols. Currently the simulation results are text files and cannot be interpreted simultaneously. The next step is to develop a visualization method to demonstrate the results. Besides, from functional aspect, the simulation tool is expected to support dynamically adjustments of the protocols.