A Digital Microfluidic Biochip Synthesis Framework

Daniel Grissom[1], Kenneth O'Neal[1], Benjamin Preciado[1], Hiral Patel[1], Robert Doherty[1], Nick Liao[2], Philip Brisk[1]

[1]Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521

[2]The Bishop's School
La Jolla, CA 92037-4799

***Abstract*—Synthesis of digital microfluidic biochips (DMFBs) is a crucial to the advancement and realization of miniaturized, automated, programmable biochemistry solutions; synthesis is performed in three steps: scheduling, placement and routing. In principle, algorithms for specific steps should be interchangeable with one another; however, different research groups typically develop algorithms for each step in isolation from one another. Thus, it is difficult to compare algorithms against one another, or to determine which algorithms for different steps share synergies.**

We introduce an *open source* DMFB synthesis framework to encourage collaboration between researchers working in the area. We introduce a common interface and describe the internal data structures that must be updated to ensure that the interfaces are adhered to. We also present and describe a number of high-quality 2D and 3D debugging tools that provide graphical output for each stage of synthesis.

Keywords-Digital Microfluidic Biochip (DMFB), Scheduling, Placement, Routing

I. INTRODUCTION

Digital microfluidics [11] is an emerging technology that will automate and miniaturize many chemical and biochemical analyses in the future. *Digital microfluidic biochips (DMFBs)* actuate discrete droplets of liquid on a 2-dimensional grid and are expected to play an important role in the development and evolution of fully integrated, programmable *laboratories-on-chip (LoCs)*. DMFB technology has been demonstrated as a viable solution for assays (biochemical protocols) including clinical pathology [14], protein crystallization [18] and DNA amplification and analysis [14], among others.

Fig. 1 shows a DMFB as a 2D array of control electrodes; a droplet is centered atop a control electrode, but overlaps its neighbors. A process called *electrowetting* induces droplet motion. As shown in the cross-sectional view on the right-hand-side of Fig. 1, the droplet is centered atop control electrode CE2. The droplet will remain in this position while CE2 remains activated and no neighboring electrodes are activated. Deactivating CE2 and activating adjacent electrode CE1 (CE3) moves the droplet left (right), providing transport. Droplets can be split, mixed, and stored, which provide the fundamental capabilities required for assay execution.

The space on top of a control electrode where a droplet may be stored is called a *cell*. Individual cells, or groups of cells, can perform other functions, such as heating or detection, if an external device is attached to (or placed in the appropriate vicinity of) a DMFB. The output of an assay may be one or more droplets and/or readings from sensors or other detectors.

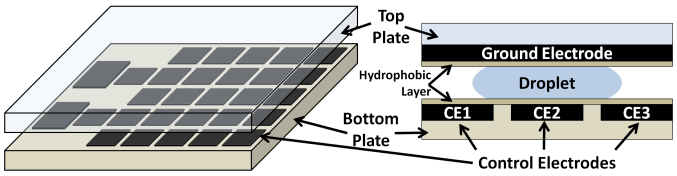

Figure 1. A DMFB with a 2D array of electrodes (left) and cross-sectional view of a DMFB (right).

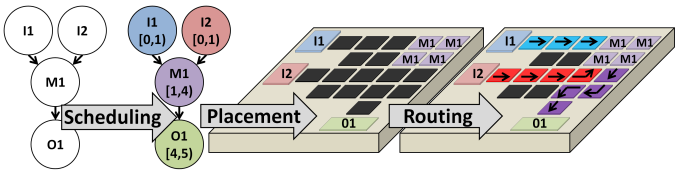

Figure 2. DMFB synthesis is composed of three, sequential, steps: operation scheduling, module placement and droplet routing.

A *droplet actuation cycle* is the time required to move a droplet from one cell to the next. During each actuation cycle, a new group of electrodes can be activated to induce motion. For a 100Hz DMFB [19], each droplet actuation cycle is 10ms, and thus, each droplet can change its location up to 100 times in 1s. From the DMFB's perspective, an actuation cycle is specified as a vector of 0s and 1s, one bit for each control electrode, which is essentially the *machine language* for the device. A sequence of vectors therefore defines an *executable program* (at the machine language level) that runs on the DMFB.

Chemists and biochemists, however, do not want to specify assays at such a low level, and have little interest in device-specific semantics. Consequently, higher-level languages have been introduced for specifying biochemical protocols [4]. *DMFB synthesis* is the process of converting a high-level assay specification into an executable program for a DMFB.

Fig. 2 illustrates DMFB synthesis. Assays are specified as directed acyclic graphs (DAGs). Synthesis involves three steps: scheduling the operations [7][9][10][12][14], placing modules onto the device [6][15][16], and routing droplets [13][19]; algorithms that perform multiple steps at once, to synergize cross-boundary optimization, have also been proposed [16].

II. MOTIVATION AND CONTRIBUTION

Researchers working on DMFB synthesis introduce new algorithms every year. In practice, these algorithms are developed in isolation; there appears to be no interoperability between the implementations, and little, if any, source code is publicly released. Ideally, the algorithms would be compatible with one another, and research groups that wish to work on one

problem (e.g., droplet routing) would not need to implement schedulers and placers to provide a basis for their research.

It is difficult to identify synergies between algorithms published by different research groups that have not been tested together. Both paper writing and software development is prone to human error; there is non-negligible likelihood that pseudocode in one paper does not precisely match what was implemented, or omits key details. Lastly, researchers who try to implement algorithms based on pseudocode written in other papers may inadvertently implement certain steps incorrectly.

To address these concerns, we developed a DMFB synthesis framework with well-defined internal data structures and interfaces between each step. We implemented several DMFB synthesis algorithms [12][13][14][15] within this framework, and compared them with new algorithms that we developed internally [6][7][10]. All algorithms that we have implemented are included in our source code release [1]. Our framework includes visualization tools that create high-quality graphical output after each synthesis step. In our experience, these tools have proven invaluable for debugging and comparison, and we believe that they will provide value to other researchers as well.

III. DMFB Synthesis Framework Overview

Fig. 3 illustrates our DMFB synthesis framework, including inputs, intermediate outputs between stages, final output, and graph visualization tools. Black boxes represent software modules (synthesis tasks) and/or data visualization tools; the document boxes represent human-readable, plaintext files that are produced and/or consumed by the software modules that comprise the simulator. Synthesis algorithms are implemented in C++, while visualization tools are implemented in Java; each stage of synthesis outputs a human-readable text file, which can be read as an input to the next stage or a visualization tool. An externally available, graph-drawing tool, *GraphViz* [2], can visualize the assay specification (as a DAG) as it is modified and annotated by the intermediate steps of the synthesis flow.

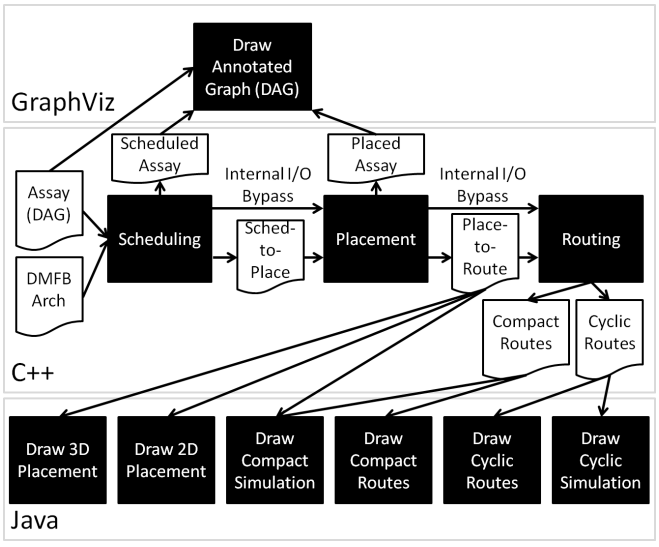

Figure 3. Flowchart of the synthesis toolflow. Black boxes represent synthesis stages and visualization tools; white document boxes represent I/O files. The synthesis steps performed are scheduling, placement, and routing.

The synthesis flow is modular. Each step can execute as a standalone command-line program, or all steps can execute atomically; in the latter case, output of intermediate files for visualization purposes is optional, as each stage can propagate its internal data structures to its successor. When one or more steps executes as a standalone program, the synthesis engine uses a built-in utility class to perform I/O and to construct and destruct all internal data structures. The utility function performs file I/O in a manner that is transparent to the user; thus, we do not describe the syntactical structure of the interface files; our source code [1] documentation describes the interface files' syntax in detail. Here, we focus on the internal data structures used by the framework and visualization tools.

A. Visualization Suite

Visualization provides two key capabilities that will assist users of our framework: debugging, and high quality visual output that can be integrated into papers and presentations. The remainder of this section highlights these capabilities. We use the *Polymerase Chain Reaction (PCR)* mixing stage [14] as a running example; PCR is also included with our source code release [1].

1) Input and Pre-Scheduling Visualization

The user specifies an assay as a DAG by writing a text file. In the future, we plan to provide interface support so that the user could specify the assay using the BioCoder language [4]; BioCoder's compiler would then be modified to output the DAG in a format that is compatible with our framework. Our framework translates the assay specification file into a .dot file format, which is compatible with GraphViz. With the GraphViz file, the user can visually verify the assay specification and update it if an error is found. As shown in Fig. 4 (left), each node in the DAG is annotated with information, including its operation type (e.g., dispense, mix, etc.), length of duration, and its name (if available).

2) Scheduling Visualization

Scheduling computes the start and stop time for each operation in the DAG [7][9][10][12][14]; this information is added to each node. The scheduler outputs an updated .dot file for visualization, as shown in Fig. 4 (center).

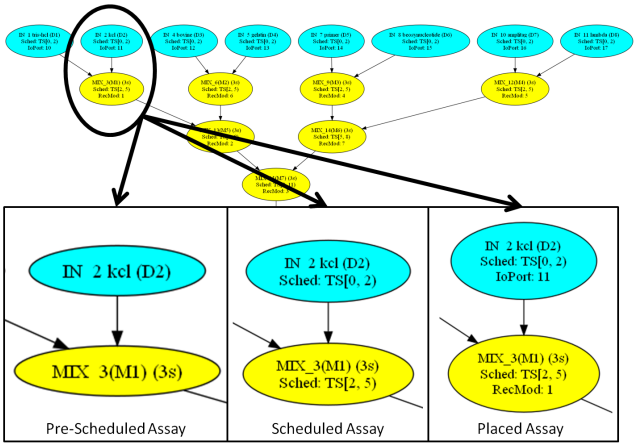

Figure 4. GraphViz visualization for the assay prior to synthesis (left), scheduled (center), and placed (right). All text is legible in native output files.

3) Placement Visualization

Placement determines the specific location on the DMFB where each assay operation will start, at the times computed by the scheduler [6][15][16]. The placer annotates each node in the DAG with the location of the module, and outputs a .dot file, as shown in Fig. 4 (right).

We have also implemented Java applications that can depict the placement in 2- or 3-dimensions. Schedules are computed on the granularity of *time-steps*, which are typically 1s or 2s for most assays. An assay operation must *start* at the *beginning* of a time-step, and must *finish* at the *end* of a time-step.

The 2D placement visualizer draws an image of the DMFB, with modules placed, for each time-step. Fig. 5 shows an example. The "TS 5" label in the upper left hand corner indicates that this placement occurs at the fifth time-step of the assay. The light blue cells depict two concurrent mixing operations. The dark ovals above each mixer provide information about the two mixing operations with respect to their location in the DAG.

The light red cells depict the *interference region (IR)* [6][15][17] of the mixing operations. Any droplet that inadvertently enters the interference region of an operation will mix with the droplet(s) engaged in the operation, which could result in contamination. Similarly, two operations that overlap with one another at the same time-step and location will cause inadvertent mixing. The visualization tool is thereby useful for debugging placement algorithms.

Fig. 5 depicts I/O reservoirs on the periphery of the DMFB, each of which displays the type of fluid it contains; the output reservoir is simply labeled "output." The cells with insignias—fire and magnifying glasses—indicate that external devices that perform heating and detection are available at those locations. The heater is essentially a physical element that is placed above or below the chip. The detector, for example, could be an infrared camera, completely external to the chip, but focused directly on those specific cells.

Fig. 6 depicts a 3-dimensional visualization, in which the third dimension (vertical axis) is time. This allows the user to view the scheduled and placed assay operations at each time step, as the assay proceeds. The DMFB is shown at the bottom; a red plane above the DMFB is drawn for each time-step, and time-steps are clearly labeled. Operations that have been placed on the DMFB stretch vertically above the cells that execute them. Modules are labeled with a module number, which can be used to find the corresponding assay operation in the DAG (e.g. Fig. 4, right). The placement is rotated so that it can be viewed from all angles; the user can also "fly" through the 3-dimensional space using keyboard controls to view the placement from any desired perspective.

4) Routing Visualization

The Java graphics suite uses two different approaches to display droplet routes. As shown in Fig. 7, the cyclic-route view draws an image for each droplet actuation cycle that droplets are in motion. The droplets are numbered, and the light red cells surrounding each droplet represent its interference region (IR). Similar to modules, two droplets will inadvertently mix if one enters the interference region of another.

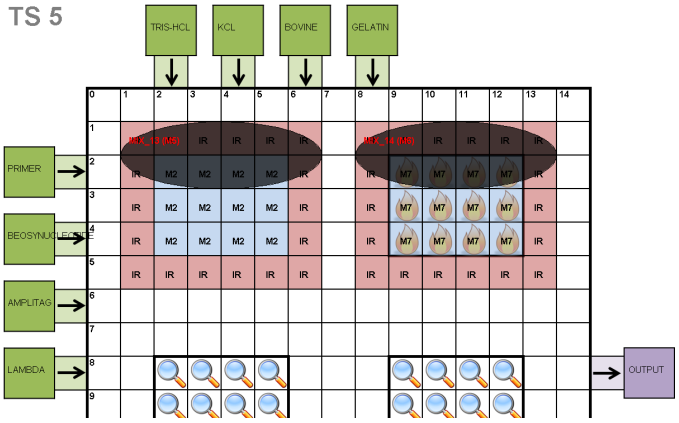

Figure 5. Sample 2-dimensional placement visualization (the bottom half of the DMFB is clipped for space). All text is legible in native output files.

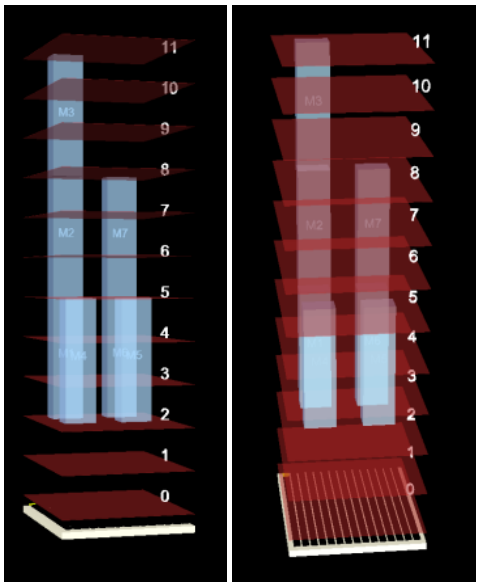

Figure 6. Sample 3-dimensional placement visualization.

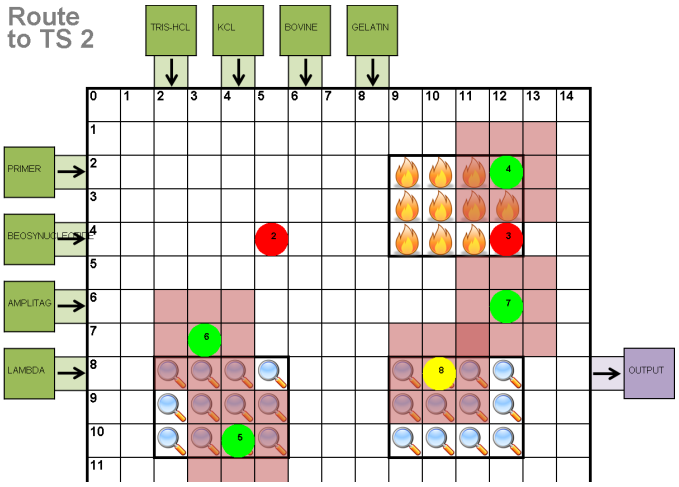

Figure 7. Sample cyclic-routing visualization depicting where each droplet is located at a particular droplet actuation cycle. Droplet interference regions are shown in transparent red, while the droplet colors indicate its status (green = free to move; yellow = waiting to avoid droplet interference; red = droplet has reached its destination). All text is legible in native output files.

The droplet color in Fig. 7 is a status indicator which displays whether the droplet is free to move forward along its route (green), is waiting for another droplet to move out of its way (yellow), or has reached its destination and has stopped (red). Other colors not shown in Fig. 7 indicate droplet merging or I/O operations.

The tool draws an image for each actuation cycle of each "*routing phase*" (i.e, the droplet routes computed for each time-step in the schedule). For each routing phase, the frames are compacted into a movie, where each frame equals 10ms, creating a real-time video for a 100Hz DMFB.

Lower-end machines, such as netbooks and tablets, may not be able to handle the computational complexity of drawing images for each droplet actuation cycle and creating a movie. Thus, the visualization suite includes a compact-route view, shown in Fig. 8, which draws a single image for each route. The electrodes are numbered so that the user can see the exact path the droplet is taking. Although the DMFB may contain multiple droplets at once, only a single droplet's path is drawn in any image; otherwise, the image would be too chaotic and aesthetically displeasing to the human user. Debugging a router using this tool would be tedious; however, it remains a useful tool for visualization on low-end computing devices.

5) Simulation Visualization

The graphics suite has two packages that can display the entire simulation process to the user:

The *cyclic simulator* draws an image for each droplet actuation cycle; it includes all images drawn by the cyclic routing visualizer, as described in the preceding subsection. It also adds images for the cycles between routing phases where assay operations occur; it shows droplets being processed inside the modules, as shown in Fig. 9. Once again, the tool can stitch the images together to form a movie. This is the most complete representation of the assay from the perspective of the DMFB, as all executing operations are visually shown.

The *compact simulator* is designed for low-end machines where the time required to draw images for all droplet actuation cycles may be prohibitive. This simulator interleaves the 2D placement images (e.g., Fig. 5) with the compact routing images (e.g., Fig. 8). For each routing phase, one image for each droplet is included. This provides a quick and efficient representation of the simulation in progress, but at a coarser granularity of detail than the cyclic simulator.

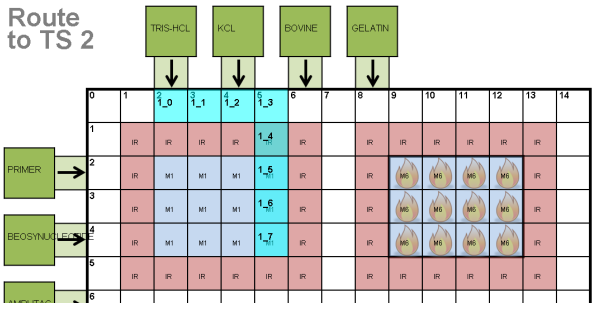

Figure 8. Sample compact-routing image showing the path a single droplet takes. All text is legible in native output files.

IV. Internal Data Structures

Each step of the synthesis framework can be performed in isolation as a standalone program, or the complete synthesis flow can execute atomically as a single program. In the latter case, the necessary internal data structures maintained by the synthesis engine are modified and passed along from the scheduler to the placer, and then to the router.

To ensure that any scheduling algorithm can be used with any combination of placement and routing algorithms, our framework defines four interfaces that impose common I/O formats for each step. As long as the implementation of a given algorithm adheres to these formats, it can execute automatically (i.e., as a standalone program). The framework outputs intermediate synthesis results as human-readable text files, which can be saved for later processing.

As shown in Fig. 10, the *Synthesis* engine (class) maintains an instance of a scheduler, placer and router. These instances inherit from global classes that force all algorithms implementing these steps to use the same functions, parameters and internal data structures for interfacing. This minimizes the number of changes that a user must make to the source code to implement a new algorithm in the framework.

As shown in Table I, the synthesis engine includes five internal data structures, which are passed between the scheduler, placer, and router. Table I shows which synthesis steps read (R) and write (W) these data structures, and which methods do not access them at all (-).

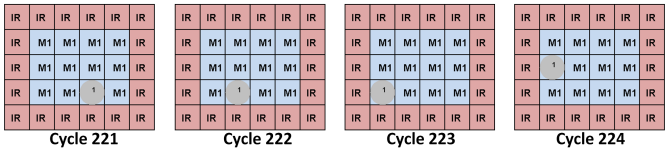

Figure 9. Illustration of droplet movement within a module.

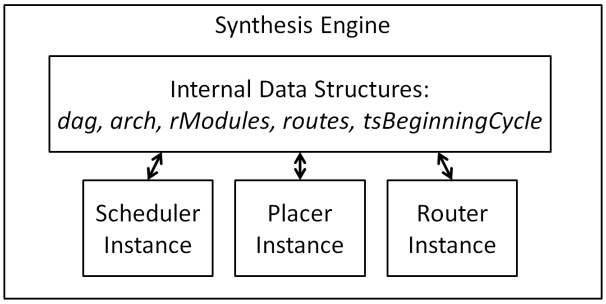

Figure 10. Synthesis engine contains an instance of a scheduler, placer and router, as well as internal data structures passed between synthesis steps.

TABLE I. Internal Data Structures of Synthesis Engine showing whether the scheduler (S), placer (P) or Router (R) reads (R) or reads and writes (W) to the data structure.

C++ type and name	S	P	R
`DAG *dag`	W	W	R
`DmfbArch *arch`	R	R	R
`vector<ReconfigModule*> *rModules`	-	W	R
`map<Droplet*,vector<RoutePoint*>*> *routes`	-	-	W
`vector<unsigned long long> *tsBeginningCycle`	-	-	W

All interface files are human-readable, well-structured text files. A utility class called *Util* handles all file I/O, according to the I/O specifications (see our source code download [1] for details). *Util* reads the appropriate file to populate all internal data structures when a stage of the flow begins, and outputs its result to a text file when the stage is finished.

A. Scheduler-Input Interface and Scheduling

The first interface defines the scheduler input. The scheduler accepts two input files: the assay specification file, and a DMFB architectural description file, as shown in Fig. 3. The assay specification file contains the basic information needed to construct the DAG; the *Util* class reads the text file and creates a data structure of type *DAG* called *dag* to represent the assay internally. *Util* annotates nodes with the operation type, length, and other relevant information to help understand the assay; the nodes do not (yet) contain any information about start times, stop times, or module placement.

The DMFB architecture file contains the dimensions of the chip, the locations of any input reservoirs (on the periphery) and fixed resources (e.g. heaters and detectors), the droplet actuation frequency of the DMFB, and the time-step length in seconds. The *Util* function creates a data structure of type *DmfbArch* called *arch* and populates it with this information. The scheduler is then called with *dag* and *arch* passed as parameters, as shown in Table I. The scheduler computes the start and stop times of each node, and the module type to which it is bound (e.g., a mixer, heater, detector, I/O, etc.).

B. Scheduler-to-Placer Interface and Placement

After scheduling, the *Util* class creates an output file that is provided to the placer; it does this by flattening *dag* and *arch* into a single text file, including all information added to the nodes by the scheduler.

Prior to placement, an empty vector of reconfigurable modules, *rModules*, is allocated, and *dag* and *arch* are either recreated by *Util* (if the placer runs as a standalone program), or passed along by the scheduler; *dag, arch,* and *rModules* are passed to the placer.

From the interface perspective, the placer has two tasks: (1) it creates a reconfigurable module, *rMod*, with a unique identification number for each non-I/O node in *dag* and adds it to *rModules*; and (2) it binds *rMod* to a node via pointers (e.g., *dag->node->module = rMod*); the placer also binds each I/O operation to a valid port (e.g., *dag->node->ioPort = port*).

C. Placer-to-Router Interface and Router

Once placement completes, the *Util* class flattens *dag* (which now contains references to specific modules in *rModules*), *arch* and *rModules* to produce a text file that can be passed to the router. Prior to routing, the *Util* class recreates *dag*, *arch* and *rModules* from the interface file. Then, two new empty data structures are created.

The first data structure is a list of droplet routes called *routes*, as shown in Table I. The *RoutePoint* structure represents the *(x, y)* coordinate of a droplet, the cycle number representing when the droplet is at that given coordinate, and the droplet's status (e.g. waiting, processing, etc.). Thus, the router maps each droplet to a vector of *RoutePoints*, which wholly characterizes the droplet's route. A droplet must have a *RoutePoint* for each actuation cycle along its route. This is certainly not the most time- and space-efficient representation of a droplet's route; however, it is easy to use and understand.

The second data structure is a vector of cycles called *tsBeginningCycle*, which dictates the cycle at which each time-step begins. Starting at time-step 0, a cycle should be added to *tsBeginningCycle* for each time-step. This information is determined in the router because a time-step cannot officially begin until all droplets have been routed to their destinations.

The router is called with 5 parameters: *dag* (scheduled and placed), *arch*, *rModules* (populated by the placer), *routes* (empty) and *tsBeginningCycle* (empty). To adhere to interface standards, the router creates a droplet-route pair (*pair(Droplet*, vector<RoutePoints*>*)*) for each droplet in the simulation and adds it to *routes*, adding a new *RoutePoint* for each cycle the droplet is on the DMFB. The router must also add the cycle number of the next time-step, which begins immediately after the routing phase ends. This data structure allows the visualization tool to display the time-step of each droplet actuation cycle during simulation.

D. Router Output Interface

Util is called when the router completes, and produces two output files for visualization. One file is used to create the cyclic-route and cyclic-simulation visualizations (Fig. 7), and the other produces the compact-route visualizations (Fig. 8).

E. Interface Bypass

The interface files created as output for each synthesis step are used as input to the visualization suite, which was written in Java. In practice, the visualization files are optional, and the entire synthesis flow can run as one atomic program, passing the internal data structures between steps; suppressing file I/O can reduce runtime and clutter in the file system.

V. Implementation Status and Usability Study

The framework compiles successfully under *gcc* in both Windows (using the *MinGW* toolchain [3]) and Linux; the visualization tools are written in Java, and, hence, are portable.

The simulator was used successfully in an undergraduate-level senior design project course at UC Riverside. Four undergraduate students were presented with the simulator, and access to the graduate student who was the primary developer. The initial version of the simulator had implemented list scheduling [14], a default placer in which all modules are placed on pre-defined locations on the chip, and a default router that transported one droplet at a time from its source to destination, using Dijkstra's algorithm to compute the route.

The students were given papers to read on scheduling, placement, routing, and DMFB technology in general. The instructor spent a significant amount of time with them to make sure that they understood the pseudocode for all algorithms that would be implemented. The instructor also explained algorithmic techniques, such as genetic algorithms and simulated annealing, which are used in some of the papers.

Within a 10-week period, the students were able to implement two genetic scheduling algorithms [12][14], a

simulated annealing-based placer [15], and one router of non-trivial complexity [13]. Several of the students have continued to work with the simulator, either as volunteer researchers or for independent-studies course projects, and other students (mostly undergraduates and M.S. students) have joined the project as well. Our present effort, which is ongoing, is to implement all existing scheduling, placement, and routing algorithms for direct-addressing DMFBs within the framework, in order to facilitate an honest and unbiased comparison.

The framework has also been used to produce new research results, including two new scheduling algorithms [7][10] and a fast online DMFB synthesis flow that is intended for dynamic interpretation, rather than static compilation [6].

The results of the algorithms implemented in the simulator are similar to previously published results; it is difficult to obtain the exact results reported by others for iterative improvement algorithms [12][14][15] because the random number seeds used for the experiments were not published. As the schedules and placements may differ as well, we did not obtain the same routing instances as prior work [13], so the routing times are likewise different. Unsurprisingly, the schedulers based on genetic algorithms [12][14] achieve better quality results than standard list scheduling [14]; however, the runtime of the genetic algorithms is significantly higher, as they are iterative improvement algorithms, whereas, list scheduling is a greedy heuristic. The new heuristic scheduling algorithms that we have since developed [7][10] have narrowed the solution quality gap considerably, although they run a bit slower than standard list scheduling.

VI. Related Work

BioCoder is a high-level language for biological protocols developed at Microsoft Research, India [4], and its compiler has been open-sourced. BioCoder creates a visual DAG output of each assay, similar to our visualization suite, and also outputs an unambiguous "cookbook-style" specification of each assay. The objective of this latter output option is to reduce ambiguity in assay specifications in peer-reviewed scientific literature, which is subject to human error. As mentioned earlier, we plan to modify BioCoder's compiler to generate DAGs that are compatible with our framework.

Micado [4] is an AutoCAD plug-in that automatically generates the control layer for continuous fluid-flow based microfluidic chips based on multi-layer soft lithography—a completely different microfluidic technology than DFMBs.

To the best of our knowledge, no other tools to support programmable microfluidics research are presently available.

VII. Conclusion and Future Work

We encourage researchers who want to study DMFB synthesis to download and use our framework. We hope that they will develop and contribute new algorithms using this framework, as it provides a common platform for comparison. The framework can also be used to create undergraduate and graduate courses on the topic of programmable microfluidics, and to support undergraduate senior design projects and graduate-level projects and theses.

In the future, we plan to add functionality to the simulator for cross-referencing and pin-constrained DMFBs that support constrained addressing schemes, and to introduce fault models to support research on fault tolerance, testing, and recovery [8].

Acknowledgment

This work was supported in part by NSF Grant CNS-1035603. Daniel Grissom was supported by an NSF Graduate Research Fellowship.

References

[1] www.microfluidics.cs.ucr.edu

[2] www.graphviz.org

[3] www.mingw.org

[4] N. Amin, W. Thies, and S. Amarasinghe, "Computer-aided design for microfluidic chips based on multilayer soft lithography," in *Proc. ICCD*, Lake Tahoe, USA, 2009, pp. 2-9.

[5] V. Ananthanarayanan and W. Thies, "Biocoder: a programming language for standardizing and automating biology protocols," *J. Biol. Eng.*, vol. 4, no. 1, pp. 1204-1216, Dec. 2010.

[6] D. Grissom and P. Brisk, "Fast online synthesis of generally programmable digital microfluidic biochips," in *Proc. Int. Conf. HW/SW Codesign and Sys. Synth. (CODES+ISSS)*, Tampere, Finland, 2012.

[7] D. Grissom and P. Brisk, "Path scheduling on digital microfluidic biochips," in *Proc. Design Automation Conference (DAC)*, San Francisco, CA, 2012, pp. 26-35.

[8] T. Ho, K. Chakrabarty and P. Pop, "Digital microfluidic biochips: recent research and emerging challenges," in *Proc. Int. Conf. HW/SW Codesign and Sys. Synth. (CODES+ISSS)*, Taipei, Taiwan, 2011, pp. 335-343.

[9] L. Luo and S. Akella, "Optimal scheduling of biochemical analyses on digital microfluidic systems," in *Proc. Conf. on Intelligent Robots and Systems*, San Diego, CA, 2007, pp. 3151-3157.

[10] K. O'Neal, D. Grissom and P. Brisk, "Force-directed list scheduling for digital microfluidic biochips," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integration (VLSI-SoC)*, Santa Cruz, CA, 2012.

[11] M. G. Pollack, A.D. Shenderov and R. B. Fair, "Electrowetting-based actuation of droplets for integrated microfluidics," *Lab Chip*, vol. 2, no. 2, pp. 96-101, Mar. 2002.

[12] A. J. Ricketts, K. Irick, N. Vijaykrishnan and M. J. Irwin, "Priority scheduling in digital microfluidics-based biochips," in *Proc. Conf. on Design Automation and Test in Europe (DATE)*, Munich, Germany, 2006, pp. 329-334.

[13] P. Roy, H. Rahaman and P. Dasgupta, "A novel droplet routing algorithm for digital microfluidic biochips," in *Proc. of the Great Lakes Symp. on VLSI (GLSVLSI)*, Providence, RI, 2010, pp. 441-446.

[14] F. Su and K. Chakrabarty. "High-level synthesis of digital microfluidic biochips," *ACM J. Emerging Tech. Comput. Syst.*, vol. 3, no. 4, pp. 16.1-16.32, Jan. 2008.

[15] F. Su and K. Chakrabarty, "Module placement for fault-tolerant microfluidics-based biochips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 682-710, Jul. 2006.

[16] F. Su and K. Chakrabarty, "Unified high-level synthesis and module placement for defect-tolerant microfluidic biochips," in *Proc. Design Automation Conference (DAC)*, Anaheim, CA, 2005, pp. 825-830.

[17] F. Su, W. Hwang and K. Chakrabarty, "Droplet routing in the synthesis of digital microfluidic biochips," in *Proc. Conf. on Design Automation and Test in Europe (DATE)*, Munich, Germany, 2006, pp. 323-328.

[18] T. Xu, K. Chakrabarty and V. K. Pamula, "Defect-tolerant design and optimization of a digital microfluidic biochip for protein crystallization," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 29, no. 4, pp. 552-565, April 2010.

[19] P-H. Yuh, C-L. Yang and Y-W. Chang, "BioRoute: a network-flow-based routing algorithm for the synthesis of digital microfluidic biochips," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 11, pp. 1928-1941, Nov. 2008.