Synthesis of Biochemical Applications with Operation Variability on Digital Microfluidic Biochips

Christian Ejdal Sjøgreen - 072429

March 18, 2011

Supervised by: Paul Pop

Abstract

Digital microfluidic biochips are small devices promising to replace much of the equipment found in biochemical laboratories today. A digital biochip is composed of a two-dimensional array of electrodes. A small drop of liquid (a droplet) can occupy one electrode. The droplets can be manipulated on the biochip using a technique called electrowetting-on-dielectric.

Any biochemical application (or bioassay), can be captured as a series of basic microfluidic operations performed using the droplets on a digital biochip. The bioassays are modeled using a directed graph, where each node is an operation. Apart from the operations themselves, the graph indicates the order in which they must be executed for the bioassay to be successful. Such a graph is called an application graph.

In order for a digital microfluidic biochip to execute the application graph of a bioassay the graph must be synthesized into a schedule for the biochip microcontroller. The schedule tells the biochip when and where to move the droplets to perform the bioassay.

Errors can appear during the execution of an application, such as a stuck droplet or an imperfect split operation resulting in imbalanced volumes. A simple scheme has been proposed in literature, where several recovery schedules are produced at design time and are stored in the microcontroller. However, in this thesis we consider that the recovery schedules are produced during runtime, based on the observed errors.

Researchers have proposed methods for schedule synthesis based on meta-heuristics, but they are very time-consuming. Hence, the existing methods cannot be used to synthesize a recovery schedule for the biochemical application at run time, as a reaction to errors. The objective of the thesis is to develop a fast and accurate heuristic for recovery schedule synthesis that can be applied online for fault-tolerance.

We present the types of errors that can appear. Different approaches for the biochip controller to react to errors are discussed. The existing synthesis method is presented, and a fast but accurate heuristic algorithm for recovery schedule synthesis handling is proposed and implemented.

Contents

List of Figures

1 Digital Microfluidic Biochip

This chapter introduces the principles of digital microfluidic biochips (DMBs) and the purpose of this technology. It includes information about how DMBs work, what kinds of different operations can be executed on them and how operations are selected to execute a given task. The final section introduces the different errors that may occur when executing operations on the chip.

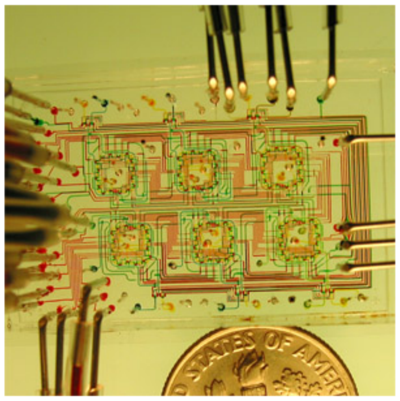
1.1 Lab on a Chip

Microfluidic biochips are the most recent addition to laboratory equipment. These new tools provide a miniature alternative to much of the equipment found in a laboratory today. A microfluidic biochip can be configured to perform many of the same kinds of bioassays that would otherwise require a fullscale laboratory. Because of this, a microfluidic biochip is often referred to as a lab-on-a-chip. These tiny laboratories only use a tiny amount of reagents contained in a small device. This provides a higher level of safety when working with hazardours materials and is more efficient for expensive or scarce samples.

1.2 Continous or Digital Flow

A regular microfluidic biochip is set up to mimic the flow of reagents through a laboratory. The reagents travel through a series of tiny pressurized tubes or carvings in the chip. Along the way they mix, dissolve and split in order to perform the desired bioassay. The droplets are moved using integrated microstructures, such as tiny pumps and valvels. These chips are the so called continous flow microfluidic biochips and they are already deployed in analytical equipment. A picture of a continous flow biochip is shown in Figure 1a

Continous flow chips excel at performing the same task over and over but the tubes, carvings and microstructures cannot be reconfigured. For this reason a seperate device is needed for each kind of bioassay.



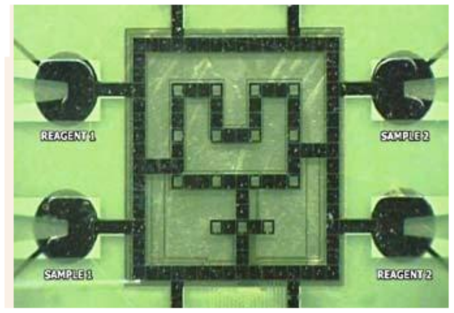


Figure 1: a) Continuous flow biochip[5]. b) Digital flow biochip[6].

Then what are *digital* microfluidic biochips? DMBs are set apart from continous flow biochips by operating in a discrete world. In this world the reagents exist as tiny volumes (droplets) occupying specific locations on the chip. On a DMB operations are performed on a two dimensional array of cells, using electricity. A droplet can move from one cell to any adjacent cell and may move in any direction. The cells are identical allowing a DMB to be reconfigured to perform many different bioassays. An example of a fabricated digital microfluidic biochip is shown in Figure 1b.

1.3 Working Principle of a DMB

A digital microfluidic biochip works by manipulating small amounts of liquid (or droplets) contained between two surfaces. The cross section of such a chip can be seen in figure 2.

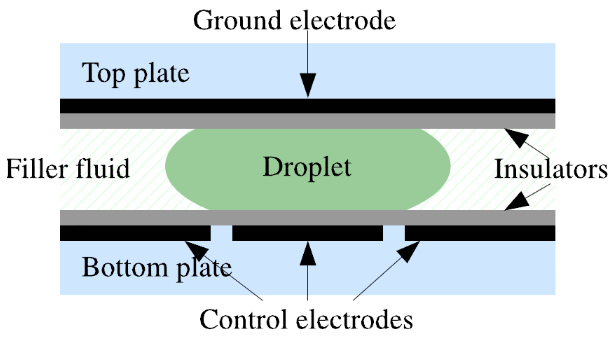


Figure 2: Biochip cross section[7].

A chip can consist of any number of individual cells and is connected to reservoirs for storing the liquid. Each cell is defined as the immediate area around a control electrode. A droplet can be moved from one cell to a neighbouring cell by activating the neighbouring cells control electrode. By activating the control electrode an electric field is created. This field pulls the closest edge of the droplet down towards the electrode thereby expanding its contact surface in that direction. This effect is called electrowetting. Due to surface tension the opposite end of the droplet has its contact surface reduced in an attempt to maintain the shape of a sphere. The droplets must overlap into neighbouring cells for this technique to work.

In addition to a control electrode some cells may be equipped with an optical device or other sensors to determine the characteristics of a droplet occupying that cell. The example in Figure 3 is a six by six DMB. Arrows indicate where droplets may enter and exit the chip that is, the input and waste reservoirs correspondingly. The chip in Figure 3 also contains an integrated sensor in one of the cells. Droplets are usually moved to sensor cells as the last step of a bioassay to validate the results.

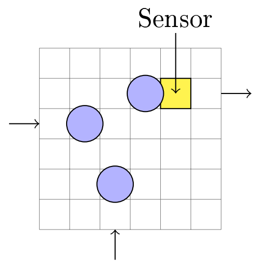


Figure 3: Top-down view of an example DMB.

Everything on a DMB is controlled by an embedded system.

1.4 Operations

A bioassay is performed by moving droplets around on the biochip. A bioassay consists of a number of operations arranged in a directed graph (the application graph). Each operation is considered a node and each link represents a droplet moving between operations. Because the links between nodes are directed an operation cannot be executed before all of its inputs have been fulfilled. These are the key operations available:

- dispense - a droplet enters the chip from one of the reservoirs. No parents, one child.
- detect - the characteristics of a droplet are determined. The droplet must remain in place during this operation. One parent, one child.
- merge - two droplets are united into a single droplet. Two parents, one child.

- mix - two droplets are moved around, split and merged until they are properly mixed. Two parents, one child.
- split - a droplet is ripped apart by activating electrodes in two opposite cells. One parent, two children.
- dilute (mix) - a droplet is diluted in a solvent using a sequence of mix and split steps. Two parents, two children.
- store - a droplet is stored in a cell on the chip until needed. One parent, one child.
- waste - a droplet is removed from the chip. One parent, no children.

The application graph in Figure 4 is comprised of some of these operations.

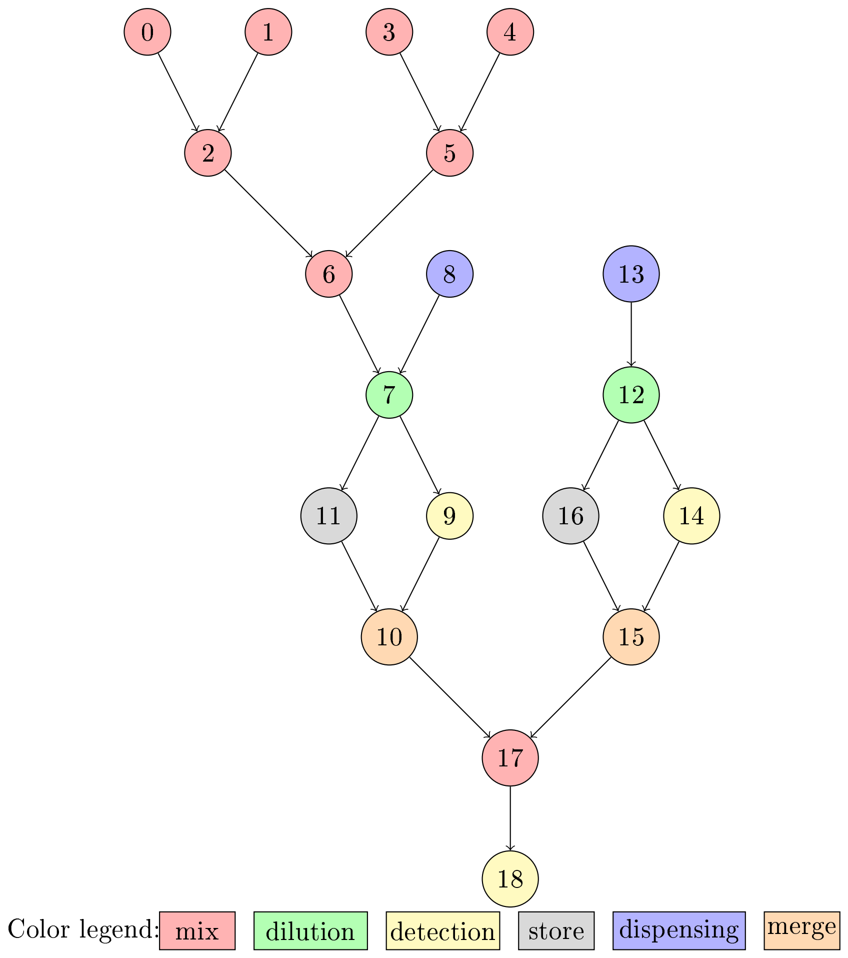


Figure 4: Example application graph.

1.5 Modules

There may exist more than one pattern of droplet movements that can implement a given operation. In order to reduce complexity of the synthesis process, a subset of movement patterns must be selected. For simplicity the selected movement patterns are often confined to rectangular shaped areas called modules. For some operations modules only represent a reserved area for that operation to execute within. Such modules can be placed anywhere on the chip and can even be moved during execution. But some operations, like dispensing, waste and detection can only occur at certain locations on the chip (i.e. detection can only be done on top of a sensor cell). In reality the number of sensor cells may be limited. Dispensing/waste would only be possible where reservoirs are connected to the chip. The number of reservoirs and sensors will be taken into account when creating the synthesis method.

The available modules make up a library. The library contains type, size and execution time of each module. This provides a higher level of abstraction than dealing with individual droplet moves. In this way it is also possible to test the correctness of the selected movement patterns.

Module type	Dimensions	Execution time
Mix	2x4	3
Mix	1x3	5
Dilute	2x5	4
Dilute	2x2	12
Detect	1x1	5
Dispense	-	7
Store	1x1	-

Figure 5: Example library of modules for a DMB.

Figure 5 shows an example library for a DMB. The dispense module has no dimensions since reservoirs are located outside the microfluidic array. The store module has no execution time because a droplet may be stored indefinitely.

1.6 Synthesis

The controller unit on the DMB conducts bioassays by executing operations in accordance with a schedule. The schedule is a list of commands denoting when, where and for how long the operations are executed on the biochip. A schedule is synthesised to minimize the execution time of the bioassay. The synthesis process consist of four tasks:

1. Allocation,
2. Binding,
3. Placement,

4. Scheduling.

1.6.1 Allocation

Allocation is choosing which modules from the library to use for the given bioassay. The number of special cells, like sensor cells and cells connected to reservoirs, is also specified during allocation. In this project allocation is given as input along with the application graph.

1.6.2 Binding

In order to be executed on the chip each operation must be bound to an allocated module. Since modules for the same operation may have different dimensions and execution times, the binding decision will affect the execution time of the schedule.

1.6.3 Placement

When an operation has been bound to a module, the module can be placed on the chip. The modules must be placed so that no two active modules take up the same cells. Additionally, to ensure that droplets do not accidently merge, the modules must be placed at least one cell apart. Cells separating droplets are called segregation cells. Figure 6 shows modules placed on a chip. The dark area represents the modules themselves and the lighter areas are segregation zones.

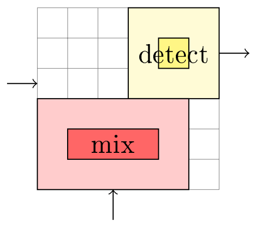


Figure 6: Modules placed on a chip.

1.6.4 Scheduling

In order to produce a correct schedule, the operations must obey the precedence constraints of the graph. This means that no child node may be scheduled before all operations of its parent nodes have finished. A node that fulfils this requirement is called a *ready node*. Using binding and placement the chosen scheduling algorithm must schedule the execution of the ready nodes as they become available.

The schedule must make use of all droplets on the chip at all times. If a droplet is on the chip but not part of an operation at any time, the schedule is invalid. Droplets that cannot be accounted for at all times during execution pose a danger to all other droplets on the chip. They may accidently mix and there is no way to know where to find the droplet, when it is needed again.

1.7 Solution

As mentioned, the goal of the synthesis is to provide an optimal solution to the problem of executing a bioassay on the DMB. But is it even possible to ensure an optimal solution for this kind of problem? Unfortunatly, the answer is no. The issue of placement alone could be considered a 2D packing problem with packages of rectangular shape. This kind of problem is classified as NP-complete. Since the synthesis also include the other choices of binding and, to some extend, allocation it may be even harder.

1.8 Errors

As is the case for many systems, errors may occur during some operations on the DMB. The errors of interest to this project are so called intrinsic errors. These occur when an operation produces a droplet of incorrect volume. For instance when a splitting operation produces two droplets of different size. Such an error in one operation will be carried over to any child operations. This will also cause the result of the child operation to deviate from the expected result. In this way errors occuring at the beginning of a long chain of operations may render the final bioassay results invalid. To guarantee the correctness of a bioassay, error limits (I) are put in place. This represents the worst case percentage offset from the target volume. A normal distribution can be used to calculate the accumulating effect of intrinsic errors. These are the equations for some key operations[]:

- Dispense (E_{Ds}). The dispensed droplet diviates from the expected volume. The offset E_{Ds} is found through testing. The error limit of a new droplet is then: $1 \pm E_{Ds}$
- Mix (E_{Mix}) is the child of two previous operations. If the error limit of these operations were I_1 and I_2 then the error limit at the end of the mixing operation is: $\sqrt{(0.5I_1)^2 + (0.5I_2)^2 + E_{Mix}^2}$
- Split (E_{Slt}) has one parent and two children. The input droplet has error limit I so *each* output droplet has error limit: $\sqrt{I^2 + (2E_{Slt})^2}$

By using these equations the error limit of an entire application graph can be calculated. For instance, the error limit of dilute (E_{Dlt}), which is the combination of mix and then split, is: $\sqrt{(0.5I_1)^2 + (0.5I_2)^2 + E_{Mix}^2 + (2E_{Slt})^2}$

1.9 Simplifications

In order to reduce the complexity of the synthesis process, the following simplifications are used during this project.

- How to move the droplets between operations (routing) is not considered. Routing is assumed to take no time.
- Each module is surrounded by one layer of segregation cells. This layer of segregation cells is even applied when the module is next to the edge of the chip or when the module is placed next to other segregation cells.
- Detect modules do not have to be fixed in place - there is no distinction between sensor cells and regular cells.

2 Existing Synthesis Method

Due to the small scale of the components and the complexity of the bioassays, computer aided design tools are used during production of microfluidic biochips. Such tools help optimize and verify the layout of a chip, simulate execution and error handling as well as performing the synthesis.

At the beginning of this project a method to synthesize an implementation was already in place[1]. The existing method uses a meta-heuristic called Tabu search in an attempt to find the optimal solution - the global minimum of schedule execution time.

2.1 Tabu Search

The principle of Tabu search is to navigate through the solution space of an NP-complete problem by visiting neighbouring solutions[1]. A neighbouring solution, in this context, is obtained in the current solution in two ways: either by re-binding one operation from the current solution to a new module or by swapping the priority of two operations. To prevent revisiting solutions the Tabu search algorithm maintains a history of recently visited solutions (a tabu list). The current implementation of Tabu search starts out with a solution in which operations are bound to modules at random.

The initial solution is primarily changed by rebinding. Only when the best solution has not changed for a number of iterations (num_{iter}) will priorities be swapped. The pseudo code for the implementation is displayed in Figure 7.

```
TabuSearch($\mathcal{G}$, $\mathcal{C}$, $\mathcal{L}$, $\mathcal{A}^\circ$, $\mathcal{B}^\circ$, $\Pi^\circ$)
$< \mathcal{S}^\circ, \mathcal{P}^\circ >$ = ScheduleAndPlace($\mathcal{G}$, $\mathcal{C}$, $\mathcal{A}^\circ$, $\mathcal{B}^\circ$, $\Pi^\circ$)
$\Psi^{best} = \Psi^{current} = \Psi^\circ = < \mathcal{A}^\circ, \mathcal{B}^\circ, \mathcal{S}^\circ, \mathcal{P}^\circ >$
$\delta_\mathcal{G}^{best} = \delta_\mathcal{G}^{current} = \delta_\mathcal{G}^\circ$ = GetCompletionTime($\mathcal{S}^\circ$)
$TabuList_{dev} = \emptyset$
$TabuList_{prio} = \emptyset$
$num_{iter} = 0$
while $timeLimit$ not reached do
  $N$ = GenerateNeighborhood($\Psi^{current}$, $\mathcal{L}$)
  $\tilde{N}$ = SelectAllowedMoves($N$)
  $(O_i, \mathcal{B}(O_i))$ = SelectBestMove($\tilde{N}$)
  PerformBestMove($\Psi^{current}$, $O_i, \mathcal{B}(O_i)$)
  RecordRebindMove($O_i$, $\mathcal{B}(O_i)$, $TabuList_{dev}$)
  $\delta_\mathcal{G}^{current}$ = GetCompletionTime($\mathcal{S}^{current}$)
  if $\delta_\mathcal{G}^{current} < \delta_\mathcal{G}^{best}$ then
    $\Psi^{best} = \Psi^{current}$; $\delta_\mathcal{G}^{best} = \delta_\mathcal{G}^{current}$
  else
    $num_{iter} = num_{iter} + 1$
    if $num_{iter} = num_{div}$ then
      $(O_i, O_j)$ = SelectSwapMove($\mathcal{G}, \Pi^{current}, TabuList_{prio}$)

      PerformSwapMove($\Psi^{current}$, $O_i, O_j$)
      RecordSwapMove($O_i$, $O_j$, $TabuList_{prio}$)
      $\delta_\mathcal{G}^{current}$ = GetCompletionTime($\mathcal{S}^{current}$)
      if $\delta_\mathcal{G}^{current} < \delta_\mathcal{G}^{best}$ then
        $\Psi^{best} = \Psi^{current}$; $\delta_\mathcal{G}^{best} = \delta_\mathcal{G}^{current}$
      end if
      $num_{iter} = 0$
    end if
  end if
end while
return $\Psi^{best}$
```

Figure 7: Tabu Search algorithm for digital microfluidic biochips[1].

The algorithm takes as input the graph $\mathcal{G}$, the two-dimensional array of cells $\mathcal{C}$, the module library $\mathcal{L}$ and the initial allocation $\mathcal{A}^\circ$, binding $\mathcal{B}^\circ$ and priorities Π°. The algorithm terminates when it has been running for a given amount of time.

2.2 Error Handling

Due to the accumulating intrinsic errors it may be necessary to implement error handling procedures. This is not done during synthesis but as part of the planning of the bioassay.

2.2.1 Error Detection Points

In order to handle an error it must first be detected. To do this, the designer adds error detection points to the application graph. An error detection point (or EDP) should be added whenever the error limit of the next operation exceeds the threshold set for the bioassay. Figure 8 shows an application graph with two error detection points, EDP 1 and EDP 2.

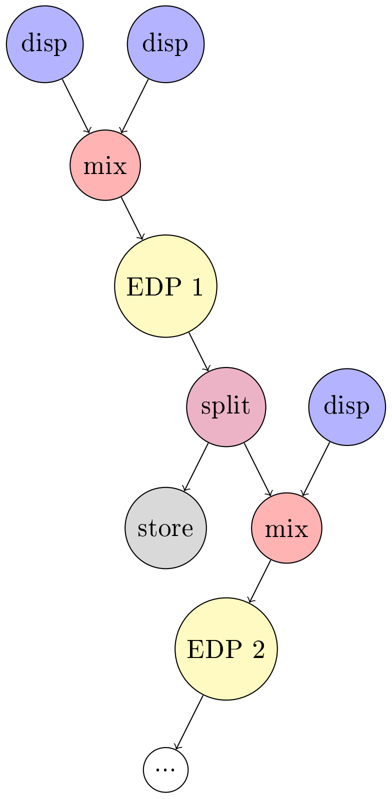


Figure 8: Application graph with error detection points.

2.2.2 Error Correction

When an error occurs the entire bioassay must be restarted - hoping that the error does not occur again. It may take several attempts to complete the bioassay with this approach.

An alternative solution is to pause the bioassay while the faulty droplet is reconstructed. This can be done by dispensing a new droplet from the reservoir or by using a backup droplet stored earlier. Either way, a new graph has to be created. This graph must contain the operations leading up to the error. Such a graph, called a Fault Tolerant (FT) Graph, must be available to the DMB controller for each error detection point along with a pre-synthesized schedule. Figure 9 shows one of the two possible FT graphs for errors occuring at the second EDP in Figure 8.

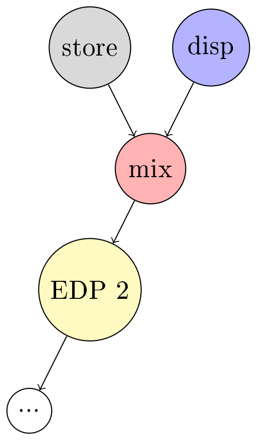


Figure 9: Example FT graph.

In this scenario the droplet from the storage is used to rerun the operations leading up to EDP 2. If EDP 2 detects an error again, the storage will be empty. In that case the only way to reconstruct the faulty droplet is to start over. The FT graph of that scenario would equal the application graph in Figure 8.

2.2.3 Online Sequential fault-tolerant Scheduling

An even better solution is to combine the remaining operations of the bioassay with the FT graph. This will allow for other parts of the bioassay to proceed during error correction. This method also requires a graph and a schedule to be computed for each error detection point. In fact, not only is an FT graph and a schedule needed for every EDP, but additonal graphs and schedules must be created to handle errors occuring in combination with other errors. Using the current synthesis method this is possible by creating a large library of error correcting graphs and schedules in advance. Due to the potientially huge amount of error combinations, this is not desirable.

To avoid pre-computing the many schedules, the synthesis procedure will have to run "online". This means that all operations on the DMB will have to be halted when an error is detected. They will remain halted until the error correcting schedule has been synthesized. This new schedule then replaces the original schedule and operations on the DMB can resume. This method is called Online Sequential fault-tolerant Scheduling (or OSS).

The problem in this setup is the execution time of the synthesis method. Because it uses the Tabu search heuristic it may take a long time to synthesize a good schedule.

Previous research[2] in OSS suggests that the synthesizer should spend no longer than two seconds working on the new schedule. If the Tabu search method is only allowed to run for two seconds, the resulting schedule will be close to the initial, random schedule. This is not a good result either. It must be possible to create a synthesis method that is faster than the Tabu search heuristic but provides a schedule that is better than a random schedule. Creating and implementing such a method is the objective of this project.

3 Fast Synthesis Method

The new synthesis method must give a good solution to an NP-complete problem without using time consuming search space techniques. In this project a heuristic technique called List Scheduling is used.

3.1 List Scheduling

List Scheduling is a general approach to scheduling problems[3]. It works by maintaining two lists. One list containing ready tasks and one containing available resources. Tasks could be jobs, program threads, trailers or ready nodes in an application graph. Resources would then be employees, cpu cores, trucks or modules on a chip. The list of tasks is sorted so that the task of highest priority is scheduled first. How to define the priority of tasks and the availability of resources depends on the context in which List Scheduling is used.

3.1.1 Priorities

A task in the context of the DMB scheduling problem is a node from the application graph. A node from the application graph cannot be executed until it is a ready node - all of its parents must have finished executing. Apart from this precendence constraint there are no requirements for the priorities. Setting the priorities is only a matter of optimizing the solution.

Getting a good solution is important though. If the schedule provided by this synthesis method is bad, the benefit of using OSS could be negated. Due to the complexity of the problem it is not possible to predict which priorities are optimal.

Looking at the application graph shown in Figure 4 it is clear that one branch is larger than the other. At some point during succesful execution all remaining operations will fit on the chip at the same time. I assume that the sooner this point is reached the shorter the final execution time of the bioassay will be. This point can only be reached when all remaining nodes in the graph are ready nodes. Since the execution time of each operation is assumed to be equal, because it is unknown at this point, the fastest way to make the last node ready is to execute the greatest grand parent in the graph. Finding the greatest grand parent is equivalent to finding the first node of the critical path (the longest path in the graph). The second greatest grand parent is then the first node of the second longest path and so on.

Priorities are assigned to nodes by giving them a value equal to the number of generations (or levels) of nodes below them. The initial ready nodes are then sorted based on this value.

3.1.2 Resources

The resources of a DMB are the modules. Whether a module is available or not depends on three things:

1. Has the module been allocated to this implemention?
2. Does the module fit the next operation in the ready list?
3. Can the module fit on the chip in the current state of the chip?

If a module fulfills these requirements it is available and can be placed on the chip. There may be more than one module available that completes the same operation. Some modules may be large but fast and some may be small and slow. At a given time in the schedule it may be more optimal to bind operations to smaller, slower modules. Or maybe choosing the larger, faster modules is better. As mentioned earlier this part of the problem is equivalent to a 2D packing problem and it is NP-complete. Choosing the optimal module is not guaranteed.

Taking a greedy aproach to the problem, the available module that provides the shortest execution time for a given operation is always chosen.

3.2 Algorithm

With priorities and resources for List Scheduling defined an algorithm for the new synthesis method can take shape. The algorithm will require three input parameters: an application graph, a module library and the dimensions of the chip. The first part of the algorithm can be seen in Figure 10.

```
Synthesis(graph, deviceLibrary, chip)
  1. priorities = SetPrioritiesBasedOnLevel(graph);
  2. readyNodes = FindInitialReadyNodes(graph);
  3. SortList(readyNodes, priorities);
  4. schedule = ListScheduling(priorities, graph, chip, deviceLibrary, readyN-
     odes);
  5. return schedule;
```

Figure 10: Fast synthesis

Based on its position in the graph, each operation in the bioassay is assigned a priority value. Then the inital ready nodes are defined and sorted. The initial ready nodes are the nodes that have no parents. The priorities and list of ready nodes is then sent to the ListScheduling part of the algorithm in Figure 11.

```
ListScheduling(priorities, graph, chip, moduleLibrary, readyNodes)

1. currentTime = 0;
2. schedule = empty;
3. activeOperations = empty;
4. while not empty (readyNodes and activeOperations) do
5.     selected = SelectFirst(readyNodes);
6.     bestModule = empty;
7      modules = GetModulesFor(moduleLibrary, selected);
8.     for every m in modules
9.         if CanPlaceDevice(chip, m) and FinishTime(m) < FinishTime(bestModule) do
10.            bestModule = m;
11.        end if
12.    end for
13.    if bestModule is not empty do
14.        Add(activeOperations, selected, bestModule);
15.        Add(schedule, selected, bestModule);
16.        Place(chip, bestModule);
17.        RemoveFirst(readyNodes);
18.    end if
19.    if readyNodes is empty or bestModule is empty do
20.        currentTime = FirstFinishTime(activeOperations);
21.        AddReadyChildrenOfFinishedOperations(activeOperations, graph, currentTime);
22.        RemoveFinishedOperations(activeOperations, currentTime);
23.    end if
24. end while
25. return schedule;
```

Figure 11: List scheduling for DMB

The List Scheduling algorithm maintains four lists while running:

1. readyNodes - The list of current ready nodes. Initially containing the nodes of the graph with no parents.
2. moduleLibrary - A list of modules allocated for this implementation.
3. activeOperations - The list of operations currently being executed on modules on the chip.
4. schedule - The result of the algorithm. Operations and modules are added as they are bound and placed.

The first step of every iteration of the large while-loop is selecting the next operation to schedule. This is done by picking the first node from the sorted list of ready nodes. All modules that can execute the selected operations are then found and added to the list *modules*. Those modules are then tested in a for-loop to see if they fit on the chip at this moment. If a module does fit and finishes faster than the fastest tested module so far, it is saved as the fastest module. At the end of the for-loop there are two possible scenarios:

1. False - None of the available modules can currently be placed on the chip.
2. True - The module with the earliest finishing time, that can be placed on the chip, has been found.

If no module was found the algorithm advances to line 19. The if-statement in this line is fulfilled if there are no remaining ready nodes or (as in this case) no available module for the selected operation. If fulfilled the variable *currentTime* is advanced to the earliest finishing time of any operation in *activeOperations*. Children of operations finishing at this time are added to the list of ready nodes and finishing operations are removed from *activeOperations*.

If the statement in line 13 is *true*, the selected operation and module are added to the list of active operations, added to the schedule and placed on the chip. The operations is then removed from the list of ready nodes. The algorithm does not terminate until there are no more ready nodes and no operations being executed on modules on the chip.

3.3 Problems and solution choices

The algorithm from Figure 11 will not guarantee an optimal solution. In fact it will not even guarantee a valid solution. These are the problems that have been identified.

3.3.1 One parent finishing before another

In the case of a node having two parents (mix operations for instance), one parent may finish before the other. At this point the output droplet of the finished parent cannot immediately be used in the child operation (since the child is not yet a ready node). This droplet must be stored temporarily or there is a risk that it will accidently come into contact with another droplet. To do this, a store operation must be added to the schedule even though it is not part of the graph. This operation has three unique features:

- This storage operation may not have been allocated and may not even be part of the device library.
- Additionally, this type of operation must receive the highest priority no matter the priority of its parent. This is in contrast to regular list scheduling in which all priorities are static during scheduling.

- And finally, the storage operation has a starting time defined by the finishing time of its parent and it is active on the chip until the child operation can be executed.

This problem must be addressed by the scheduling algorithm so the following changes to the algorithm in Figure 11 are required.

1. If an operation finishes and has children that do not qualify as ready nodes. A droplet must be stored for each of those children. This should be done by the *AddReadyChildrenOfFinishedOperations()* subroutine in line 21.
2. Before time is advanced (in line 20) a storage must be created for ready nodes that have not been placed at this point. This only applies to ready nodes that have parents since the first nodes in a graph have no incomming edges.
3. If the selected operation (the next operation to be bound and placed) uses a droplet being stored, the storage is removed. The storage is removed first so it does not block placement of the module for the selected operation. This should be done before the for-loop in line 8.

This solution gives rise to a new problem.

3.3.2 No room to store output droplet

If an output droplet cannot immediatly be part of another operation because the chip is out of space, the produced schedule will be invalid. If droplets are left unaccounted for in the schedule, they might accidently end up comming into contact with other droplets on the chip. The risk of running out of space depends on the distribution of operations in the graph. For the key operations this has the following implications:

- dispensing - this opreration takes up no space so when it finishes it may not be possible to store the output droplet.
- detection - the module for this operation requires at least one cell to execute (and 8 segregation cells) - the same as the storage module. There will always be room to store the output droplet.
- mixing - this operation only has one child and takes up more space than the storage module. The ouput droplet can always be stored.
- splitting - this operation produces two output droplets. Splitting modules are larger than a single storage module but smaller than two. There may not be enough room to schedule or store both output droplets upon completion.
- dilution (mixing) - since dilution includes splitting it has the same implications.

- store - if there ever is a need to store the output of a scheduled store operation the droplet can always be stored in the same space.
- waste - no output droplet so no problem.

Changing the priorities of new ready nodes can prevent the scheduling algorithm from overcrowding the chip with droplets. The priorities should be changed so the scheduling algorithm favors scheduling operations from the same path of the application graph over scheduling operations from new paths. The simplest way to do that is to insert all new ready nodes at the front of the priority list.

Because the initial ready nodes in many graphs are dispense operations, they pose the greatest risk. Fortunately the number of reservoirs is often limited, which will limit the number of simultanous dispense operations. This problem is left to be solved by the designer of the application graph and allocation. However, a good rule of thumb is to use DMBs with a number of cells greater than nine times the number of connected input reservoirs. This will ensure that there is enough space on the microfluidic array to store the dispensed droplets, if not used immediately as input to other operations. So a chip with four input reservoirs connected to it, should be larger than a 6x6 DMB.

4 Implementation

The purpose of this project is to implement a useful algorithm to schedule FT graphs during runtime. It is a 'proof of concept' so there is no need to consider how to interface with the DMB controller. The program is coded in Java like the existing synthesis method. This should make the source code and executable more easily available to people working on related projects.

4.1 Overview

The program is split into three parts:

1. Reading the input files.
2. Creating the schedule.
3. Verifying the output files.

The simple class diagram in Figure 12 shows how the classes associate.

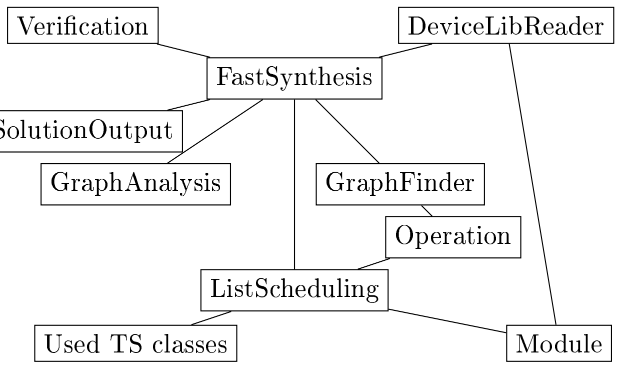


Figure 12: Class diagram for fast synthesis implementation.

4.2 Module class

The module class is used to store information about the different modules available to the scheduler. The class contains the following variables:

- String superType.
- String type.
- int exeTime.
- int length.
- int width.

The five types of information must be provided when the class is constructed.

4.3 Operation class

This class stores information about the operations of the bioassay. It uses the following variables and collections:

- String name - unique name of the operation (could be anything).
- String superType.
- String type.
- int sizeX - width of the operation when placed.
- int sizeY - height of the operation when placed.
- int posX - x-coordinate of operation when placed.
- int posY - y-coordinate of operation when placed.
- int level - an integer representing the relative position of the operation in the graph. A heigher value means the operation is closer to the end of the graph.
- int group - the group to which this operation belongs. The free nodes are sorted so that operations from the same group are scheduled together.
- ArrayList parents - list holding names of this operations parents.
- ArrayList children - list holding names of this operations children.
- ArrayList storage - list holding information about temporary storage of output droplets.

The constructor requires name, superType and type. Parents and children are added right after construction and sizeX, sizeY, posX, posY are added when the operation is scheduled. Storage operations are added to ArrayList storage if needed.

4.4 GraphAnalysis class

This class is the implementation of the algorithm from Figure 10 with additions from section 3.3.2. At construction it receives the graph as a HashMap with operation names as the key and intances of the Operation class as the value. It then performs a three step analysis of the graph.

1. The recursive method *findDepth* is called to identify at which level of the graph each node is located. The method starts at a random node of the graph. This node has $depth = 0$. Each parent of the node is assigned $parent.depth = node.depth - 1$ and for each child $child.depth = node.depth + 1$. The method then calls *findDepth* for each parent and child. If a node has no parents it is added to the list of initial ready nodes.

2. By going through a triple nested for-loop, parents of the same children are assigned similar group numbers.
3. The last step is to sort the list of initial ready nodes. The list is sorted into a list of increasing node depth using quicksort. Nodes in the list are then swapped so nodes from the same group appear in sequence.

The list of initial ready nodes is retrieved by calling the method *getReadyNodes()*.

4.5 ListScheduling class

This class is the implementation of the algorithm from Figure 11 with the additions from section 3.3.1. The constructor of this class takes six arguments:

1. HashMap containing the graph. Keys are operation names and values are operations.
2. ArrayList containing allocated modules.
3. Integer containing the width of the chip.
4. Integer containing the height of the chip.
5. Sorted ArrayList of initial ready nodes.

The schedule is created by calling the method *schedule*. The method *schedule* performs the same actions as the algorithm in Figure 11 using methods of the ListScheduling class. The methods are listed here along with a reference to a line in the algorithm.

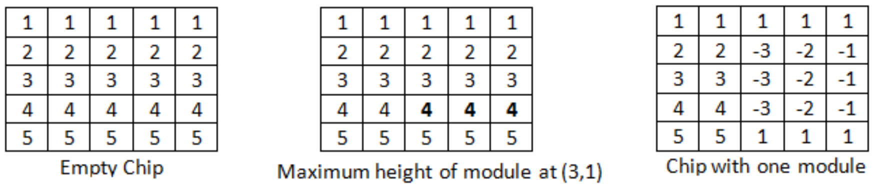
- *insertFreeChildren(Operation s, int currentTime)* - (Line 21) - Using the graph, which is a global variable in this class, this method inserts the children of the operation *s* if their parents have been scheduled and are not in the global list of active operations. The method *storeResult(s,currentTime)* is called for every child.
- *storeResult(Operation s, int currentTime)* - This method adds a special storage module (described in section 3.3.1) to the chip and a similar operation to the schedule.
- *updateStoredResults(Operation s, int currentTime)* - (Between line 4 and 5) - This method removes all special storage modules holding input droplets for operation *s* from the chip. If *currentTime* is equal to the starting time of the special storage operation then the storage operation was not needed. In that case the special storage operation is removed from the schedule.
- *insertAsActiveOperation(Operation s)* - (Line 14) - This method is called when an operation is bound to a module and placed on the chip. The method inserts *s* into the global list *activeOperations*, which is sorted by increasing finishing time of operations. The operation *s* is inserted so *activeOperations* remains sorted.

- *updateActiveOperations(int time)* - (Line 22) - This method is called when time is advanced. It removes operations from *activeOperations* if they finish at this time. The method *insertFreeChildren* is called from this method.

4.6 Other classes

Placing the rectangular modules on the chip, so that they do not overlap, requires an advanced algorithm. This algorithm is already being used for the implementation of the Tabu search heuristic. The implementation of the placement algorithm, called Placer, is imported into the implementaion for this project. A number of helper classes are also imported to support Placer.

The placer is based on the maximal empty rectangles algorithm proposed in [9] and is implemented using the area matrix data structure from [4]. Each point in the area matrix represents a cell on the chip and holds an integer. If the integer is positive, it means that the cell is unoccupied. The value of a positive integer indicates the number of unoccupied cells above it. If the integer is negative, it means the cell is occupied.



1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5

Empty Chip

1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	**4**	**4**	**4**
5	5	5	5	5

Maximum height of module at (3,1)

1	1	1	1	1
2	2	-3	-2	-1
3	3	-3	-2	-1
4	4	-3	-2	-1
5	5	1	1	1

Chip with one module

Figure 13: Area matrix example.

The maximum height of a module three cells wide at position (3,1) is revealed by finding the minimum value of the same row of three adjacent columns (see Figure 13).

4.7 Input files

The program uses two input files: a module library and a graph. If either of the files are not found, or if none of their contents match, the program produces an empty output file and terminates.

An example of a module library file is displayed in Figure 14 and a graph is displayed in Figure 15.

```
mix mix 2x4 3 4 6
mix mix 1x3 5 3 5
mix mix 2x5 2 4 7
mix mix 2x2 10 4 4
mix mix 3x3 7 5 5
mix mixDlt 2x4 5 4 6
mix mixDlt 1x3 7 3 5
mix mixDlt 2x5 4 4 7
mix mixDlt 2x2 12 4 4
mix mixDlt 3x3 10 5 5
opt opt 1x1 30 3 3 1
dis disS reservoirS 7 0 0 1
dis disB reservoirB 7 0 0 2
dis disR reservoirR 7 0 0 2
opt sensing 1x1 5 3 3 2
merge merge 1x1 2 3 3
storeDlt storeDlt 1x1 5 3 3
```

Figure 14: Module library input example.

Each line of the module library can be split into six or seven tokens: The first token is the super type of the module. The next is the type. The third is the dimensions of the module but it is ignored by the program. The fourth is the execution time of the module. The fifth and sixth are the width and height of the module (including segregation cells!). The inconsistent seventh token is the number of modules available but this is not considered in this project. For each line a `module` class instance is constructed and added to an ArrayList of modules.

```
0 mix mix
1 mix mix
2 mix mix
3 mix mix
4 mix mix
5 mix mix
6 mix mix
7 mix mixDlt
8 dis disB
9 opt sensing
10 merge merge
11 storeDlt storeDlt
12 mix mixDlt
13 dis disB
14 opt sensing
15 merge merge
16 storeDlt storeDlt
17 mix mix
18 opt opt
ARC a1_0 FROM t1_0 TO t1_2 TYPE 1
ARC a1_1 FROM t1_1 TO t1_2 TYPE 1
ARC a1_3 FROM t1_3 TO t1_5 TYPE 1
ARC a1_4 FROM t1_4 TO t1_5 TYPE 1
ARC a1_5 FROM t1_2 TO t1_6 TYPE 1
ARC a1_5 FROM t1_5 TO t1_6 TYPE 1
ARC a1_5 FROM t1_6 TO t1_7 TYPE 1
ARC a1_5 FROM t1_8 TO t1_7 TYPE 1
ARC a1_5 FROM t1_7 TO t1_9 TYPE 1
ARC a1_5 FROM t1_7 TO t1_11 TYPE 1
ARC a1_5 FROM t1_9 TO t1_10 TYPE 1
ARC a1_5 FROM t1_11 TO t1_10 TYPE 1
ARC a1_5 FROM t1_10 TO t1_17 TYPE 1
ARC a1_5 FROM t1_13 TO t1_12 TYPE 1
ARC a1_5 FROM t1_12 TO t1_14 TYPE 1
ARC a1_5 FROM t1_12 TO t1_16 TYPE 1
ARC a1_5 FROM t1_14 TO t1_15 TYPE 1
ARC a1_5 FROM t1_16 TO t1_15 TYPE 1
ARC a1_5 FROM t1_15 TO t1_17 TYPE 1
ARC a1_5 FROM t1_17 TO t1_18 TYPE 1
```

Figure 15: Graph input example.

The graph file contains two different formats of information. The first kind of lines contain three tokens. These are the names, super types and types of the operations to be scheduled. The second kind of lines represent the links between the operations. The program constructs an instance of the `Operation` class for each operation and adds them to a list. A link between two operations is listed in both objects - as child in one and parent in the other.

4.8 Output files

The program produces output files of the same format as the program of the existing synthesis method. Each line consists of at least 7 tokens:

1. Operation name.
2. Operation type.
3. Module width.
4. Module height.

5. Position on chip as (x,y)-point.
6. Starting time.
7. Finishing time.

A line may include any number of additional sequences of six tokens. Each of these sequences represent a temporary storage used to store the operations output. The six tokens of each sequence contain the following:

1. The string 'storage' to underline the reason for this additional sequence of tokens.
2. Width (always 3).
3. Height (always 3).
4. Position as (x,y)-point.
5. Starting time.
6. Finishing time.

The name of the output file is the name of the input file containing the graph with the width, height of the chip and the tag 'LS' added to the end.

4.9 Limitations

The final program has some weaknesses that sets it apart from the program for the tabu search synthesis method.

- The first weakness is related to the previously dicussed problem of running out of space to store unused droplets. Even though measures have been implemented to minimize the impact of this problem, this program may not be able to provide a solution for a given graph and chip even though a solution exists.
- The second weakness is the assumption that the given graph is a connected graph. If this is not the case only the operations of the first graph discovered by the program is scheduled.
- The resulting schedule from this synthesis method may take longer to execute than the resulting schedule of the tabu search synthesis method, shown in section 6.1. If this program is used to undo errors occuring early in large graphs, the penalty of using a less efficient synthesis method to reschedule the remaining graph may be greater than the speedup from the parallel execution of the droplet recovery sequence.

5 Testing

5.1 Evaluation

To show that the program can provide valid schedules an example will be synthesized.

The example is the biosassay using the FT graph from Figure 4. It is scheduled on a 10x8 DMB using the input files in Figure 14 and Figure 15. The output file contains the schedule seen in Figure 16.

```
4 mix 4 7 (0,9) 0 2
3 mix 4 7 (4,9) 0 2
5 mix 4 7 (0,9) 2 4 store 3 3 (0,2) 4 8
1 mix 4 7 (4,9) 2 4 store 3 3 (3,9) 4 6
0 mix 4 7 (3,6) 4 6
13 disB 0 0 (7,9) 4 11
8 disB 0 0 (7,9) 4 11
2 mix 4 7 (3,9) 6 8
6 mix 4 7 (0,9) 8 10 store 3 3 (0,9) 10 11
7 mixDlt 4 7 (0,6) 11 15
12 mixDlt 4 7 (4,9) 11 15
16 storeDlt 3 3 (0,9) 15 20
14 sensing 3 3 (0,6) 15 20
11 storeDlt 3 3 (0,3) 15 20
9 sensing 3 3 (3,9) 15 20
10 merge 3 3 (0,3) 20 22
15 merge 3 3 (0,9) 20 22
17 mix 4 7 (0,9) 22 24
18 opt 3 3 (0,9) 24 54
```

Figure 16: Output file containing schedule.

The diagram in Figure 17 provides a clear picture of the order in which the operations are being executed.

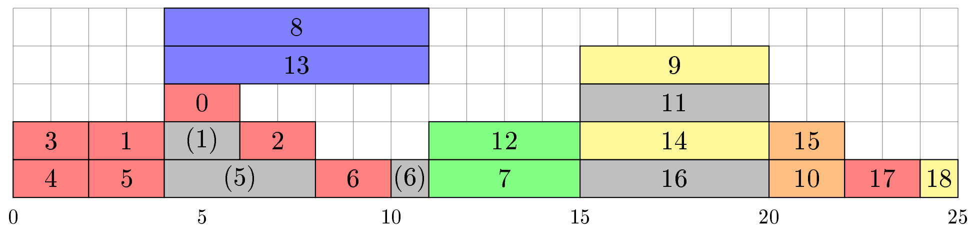


Figure 17: Diagram of example schedule.

The numbers are the operation names and the colors identify the type of operation. Numbers in brackets are the special storage operations. Comparing the schedule diagram with the graph of Figure 4 shows that the schedule fulfills the precendence requirements. The layout of the chip at three different points in time during the execution of the schedule can be seen in Figure 18.

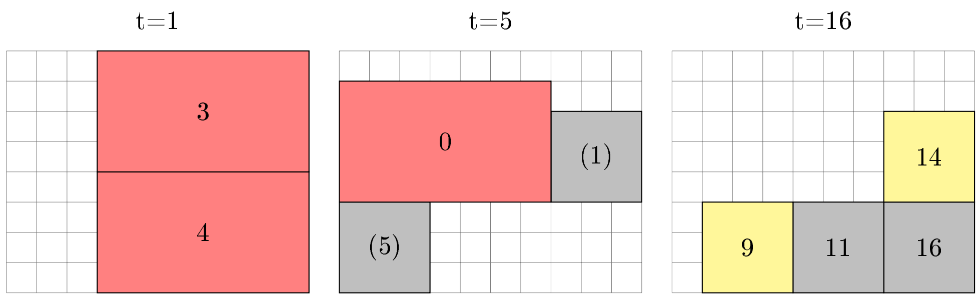


Figure 18: Chip layout for example schedule.

For this example the program terminates in a fraction of a second, which is well below the two second maximum for online sequential fault-tolerant scheduling to work.

5.2 Functional Testing

A verification procedure has been implemented to test if the schedules produced by the program are valid. The verification tool reads a schedule from a file and translates each line into an instance of the Operation class. The operations of the schedule are sorted by increasing start time and stored in a list. The relations between operations are obtained by reading the file containing the application graph. The verification tool determines if the schedule fulfills three requirements.

1. No operation is scheduled before all of its parents have finished. For a given operation s it is verified by finding all parents of s in the schedule and comparing their finishing times with the starting time of s.

2. No module is placed so it shares cells with other modules (this includes segregation cells). A two-dimensional array contains a list of timestamps for each cell. When a module is placed on the chip for a period of time, a new timestamp is created for each discrete moment in that period of time. The timestamp is added to each cell of the chip occupied by the module. If a cell already contains an indentical timestamp the schedule is invalid.

3. Output droplets produced by all operations must be consumed (used as input) by other operations. The droplet production and consumption must be synchronized so that droplets on the chip are constantly part of an operation. For a given operation s a Droplet class instance is created for each child of s. The droplet is indentified by the operation that created it and its time of creation. The droplets are added to a list. Other operations from the schedule can then search the list for input droplets matching their parents and their start time. Matching input droplets are removed from the list. The schedule is invalid if any droplets remain in the list after the last operation has been processed.

If a schedule is invalid the verification tool will print an error message in the command prompt.

6 Instructions of Use

The program is used by running the executable with the following arguments:

```
java FastSynthesis fileGraph fileDevices arrayLength arrayWidth
```

fileGraph and fileDevices are the filenames of the two input files, arrayLength and arrayWidth specify the size of the chip.

7 Conclussions

The thesis gives an insight into the working principles of digital microfluidic biochips. The synthesis process is introduced and the requirements for a valid schedule are established. The challenges of handling intrinsic errors during execution are introduced. Some of the existing solutions to these problems are described and the currently implemented solution is outlined.

The concept of using List Scheduling to provide a solution for the complex problem of synthesizing an implementation for a digital microfluidic biochip has been proven. Despite some limitations the program is working and is delivering satisfying results that enable online sequential fault-tolerant scheduling to be implemented.

The working principles of a functioning synthesis program are explained and the results are discussed.

Bibliography

1. Elena Maftei, Paul Pop, and Jan Madsen - *Tabu Search-Based Synthesis of Dynamically Reconfigurable Digital Microfluidic Biochips.* Manuscript, 2009.

2. Mirela Alistar - *Home Exam 2010 page 21.* Findings not published to my knowledge.

3. Oliver Sinnen - *Task Scheduling for Parallel Systems.* Wiley-Interscience.

4. M. Handa and R. Vemuri - *An Efficient Algorithm for Finding Empty Space for Online Placement.* Proceedings of the Design Automation Conference (2004) 960 – 965.

5. http://www.astrobio.net/pressrelease/2405/biologys-dark-matter.

6. F. Su and S. Ozev and K. Chakrabarty - *Concurrent testing of droplet-based microfluidic systems for multiplexed biomedical systems.* International Test Conference, 2004, p. 883-892.

7. Elena Maftei, Paul Pop, and Jan Madsen - *Routing-Based Synthesis of Digital Microfluidic Biochips.* Compilers, Architecture, and Synthesis for Embedded Systems Conference (CASES), 2010.

8. Zhao, Yang, Xu, Tao, Chakrabarty, Krishnendu - *Control-Path Design and Error Recovery in Digital Micro uidic Lab-on-Chip.* Journal of Emerging Technologies in Computing Systems 2009.

9. K. Bazargan and R. Kastner and M. Sarrafzadeh - *Fast Template Placement for Reconfigurable Computing Systems* - IEEE Design and Test of Computers, 2000, p. 68-83.