Goals for Thylacine, a language suite for Digital Microfluidics Platforms

Evan Kirshenbaum

Thursday, March 25, 2021

Introduction and Paper Goals

In this informal paper, I will try to accomplish two things. In the first section, I will give my current understanding of how the HP microfluidics platform works, at least from the point of view of a user who wants to run experiments on it. That is, I will not be concerned with specifics of how the platform is built, but rather the capabilities it provides. **Note: I fully realize that my understanding will be wrong, and I am counting on readers of this paper to point out my misconceptions.** I am neither a chemist nor a mechanical engineer, and I have only recently become at all familiar with the notion of digital microfluidics. I will also try to mention directions it seems likely that such platforms will evolve in the future. (These will also likely be wrong.)

In the second section, I will sketch out my current vision for a suite of languages and associated software for running such platforms. In keeping with what appears to be the current local naming convention, I'm tentatively calling this suite *Thylacine* after the Australian Tasmanian tiger.

10,000-foot view of Thylacine

The core of Thylacine is the design of four languages:

- **MPDL**, the *Microfluidics Platform Description Language*, is a language for describing and characterizing specific DMF platforms.
- **MPAM**, the *Microfluidics Platform Abstract Machine*, is a low-level model (and associated language) for operations to be performed on DMF platforms characterizable by MPDL.
- **MRL**, the *Microfluidics Recipe Language*, is an intermediate-level programming language for specifying DMF recipes independent of a particular DMF platform.
- **MPSL**, the *Microfluidics Protocol Specification Language*, is a high-level (e.g., near-natural-language) specification language for specifying microfluidics protocols close to the way they are described in the literature, with little or no programming expertise required

To support these languages software tools will be designed and implemented:

- An **MPAM API** that allows MPAM operations to be initiated from programming languages such as Java, C++, and/or Python

- An **MPAM emulator**, likely graphical, that implements the MPAM API and can emulate DMF platforms specified by MPDL descriptions.
- An **MPAM interpreter** that reads an MPAM program and executes it by making calls to the MPAM API
- An **MPAM assembler** that converts MPAM programs to equivalent programs in other programming languages
- An **MRL** compiler that takes as input an MPDL description and one or more MRL recipes and outputs an MPAM program that implements the recipes for the platform
- An **MPSL** compiler that takes as input an MPSL specification and translates it into an MRL recipe (and, optionally, from there to an MPAM program).

There will also be MPDL descriptions and MPAM API implementation libraries for various concrete DMF platforms

Caveat

This is, obviously, a lot of work, and there is, at present, only one of me, so it may well not all get done. I do, however, think that I see how to do all of it (even the MPSL compiler), so for now I'm going to work under the assumption that it is all possible.

My DMF Platform Understanding

In this section I will present a (flawed) description of digital microfluidics (DMF) platforms in general and the current HP prototype in specific. As stated above, I will be primarily concerned with the capabilities the platform presents to the user.

I should note up front that at the time I'm writing this, I know almost nothing about IonTouch or what it means for DMF platforms. As I learn more, I will revise this section (and likely my design choices for the Thylacine languages).

What is a DMF platform?

At its most basic level, a DMF platform presents an array of *pads*,[1] each of which can contain a drop of fluid. In the current platforms, the basic drop size is 0.5 µL, and each pad can hold a volume of several drops. In some cases, some of the pads may be *unusable* (e.g., because that area contains a support for the board's cover) and must be routed around.

Electrodes and drop motion

Integrated into each pad is an *electrode*. When a pad's electrode is on, fluid in neighboring pads is enticed to move to the pad. This can be used in several ways:

- Most basically, it can be used to make a drop move from pad to pad across the board.

[1] I'm tempted to call them "cells", but that might be confusing for applications that involve detection or manipulation of biological cells.

- When an empty pad is between two drop-containing pads, turning on its electrode attracts both drops and causes them to mix.[2] My current understanding is that the mixing happens essentially immediately and completely.
- Finally, if neighboring pads to a drop-containing pad turn on their electrodes at the same time, the drop will be split between the neighboring pads. My understanding is that the drop is split into essentially equal parts, or close enough to equal for practical purposes, as long as the neighboring pads are across a pad edge from the drop-containing pad (as opposed to being neighbors across corners of the pad[3]). I presume that if the electrode of the drop-containing pad is also on, that pad also participates in the splitting and retains a portion of its initial volume.

The DMF board has a *clock rate* that specifies the frequency with which droplet motion events can take place. In the current platform this rate is 10 Hz, meaning that it takes 0.1 seconds for a drop to move (or split) to a neighboring pad, and one clock cycle is sufficient for all operations regardless of the volume or content of the drop.

Electromagnets, magnetic beads, and drop separation

Some or all pads may have *electromagnets* underneath them, which may be turned on or off. If a drop contains substances bound[4] to *magnetic beads* and the electromagnet under its pad is turned on when neighboring pad entice the drop to move, the beads—and, therefore, the bound substance—remain behind and may be transferred to another drop that later occupies the pad when the electromagnet is turned off.

Thermal regions

Some regions of the pad array have the property that their temperature can be controlled. By remaining in such regions for a number of time steps, the drop may be heated to a desired temperature. My understanding is that the regions are thermally isolated to the extent that pads next to thermal regions do not have to worry about the temperature.

Wells

Along the edges of the board are some number of *wells* (or *reservoirs*), each of which can contain a (relatively) large volume of fluid. The current roadmap posits between 32 and 96 wells. In the OpenDrop system, wells can contain 120 µL. In the roadmap, the goal is for most wells to contain 250 µL, with a subset able to contain 2 mL. Each

[2] This would also seem to happen if you turn on the electrode of a non-empty pad with a drop in a neighboring pad, but my understanding is that it's considered a bad idea to have drops in neighboring pads because of possible interactions.

[3] The current pad arrays are rectangular, but this constraint would seem to mean that it might be advantageous in the future to consider hexagonal pad tilings, as then each pad would have six neighbors and all of a pad's neighbors would be across an edge rather than across a corner.

[4] I'm really fuzzy on how this works.

well is associated with an *entry pad*[5] on the one of the edges rows or columns of the board.

At least on OpenDrop, the well itself has a number of electrodes (five) that can be used to either dispense a drop from the well to its entry pad or to remove a drop from its entry pad to the well. If I understand the logic,[6] it looks as though it should take three or four time steps in order to perform either operation.

Wells have the property that when they dispense their entire content, there is no residue left behind, so the well me be reused to hold drops of another fluid without worrying about contamination.

Although the current platforms do not yet do this, it is considered desirable that wells be able to perform operations that are more efficiently performed on volumes of fluid larger than a drop. The most likely such operation is *incubation*, in which a well may be heated to a desired temperature and held at that temperature for a specified period of time. This would be considerably more efficient than the current paradigm in which incubation is performed by walking drops through thermal regions for the desired amount of time. (It would also free up those pads for other use.) The roadmap specifies that wells will also be able to cool to 4 C and perform “magnetic actuation” and “fluorescence quantification”.[7]

It’s not clear to me what other operations might be able to be performed at wells.[8]

Sensors

The above components suffice for applications which don’t involve decision making, but future DMF platforms will almost certainly want to incorporate *sensors* to allow programs that branch or loop based on what’s found. The most likely sensors would be downward facing cameras. Cameras with modest resolution could look over the board as a whole and detect color changes in drops. High-resolution cameras could focus on individual pads and perform operations like detecting and counting biological cells of a certain type.

[5] I don’t know if it’s possible for a well to have more than one exit path onto the board. It would probably simplify some operations.

[6] Narrator: He doesn’t. He’s looking at a picture of an OpenDrop board and trying to reverse engineer what happens.

[7] Which I will understand as soon as I ask somebody about them.

[8] Just off the top of my head, it might be useful to be able to extract fluid from the well to off-board equipment to perform operations like centrifuging or agitation. Extraction would also be useful for applications in which a sample is prepared for later use. Another possibility would be for a well to be able to act as a *drain*, to which you could route any fluid (in any quantity) that you no longer care about.

It's possible that other sorts of sensors could be added to certain pads to detect other physical properties of a drop. (Conductance? Presence of magnetic beads?)

Current software

There are currently two software development activities related to the prototype DMF platform.

The Hardware Abstraction Library (HAL)

Aaron Bowdle is developing/has developed the *Hardware Abstraction Library (HAL)*, which I currently know very little about, but I hope to learn a lot more about very shortly. From what I understand from Viktor, it accepts a sequence of descriptors, one per time step, that specify the coordinates of pads whose electrodes and electromagnets should be on and the temperatures for the various thermal regions. The HAL then takes this sequence and interfaces directly with the board to realize the specified sequence.[9]

Wombat

Viktor Shkolnikov is developing *Wombat*, a Python-based library for higher-level DMF programming. Wombat-based programs are defined in terms of *super operations*, such as mixing, washing, and incubating volumes of reagents, and each super operation takes its input reagents from wells and stores its output reagents into (previously empty) wells. Given a sequence of super operations, Wombat determines the wells that the initial reagents need to be in, the output wells for each super operation, and the sequence of low-level steps that are needed to realize each super operation.

Wombat also includes a rudimentary graphical emulator that takes a generated (HAL-like?) state sequence and visualizes movement of drops on the board.

Thylacine

Next, I'll describe the suite of languages and tools that I envision building. Note that this is essentially the plan that I came up with before I came on board and had access to any internal information. While I haven't yet heard anything that would cause me to think that this is not the right approach, I am fully prepared to be convinced that it needs to change.

Assumptions and expectation setting

Before describing the languages and tools themselves, it's probably worthwhile to lay out some of the assumptions that I'm making.

Target users

I have two primary users in mind in my designs. The first understands the domain and has some minimal programming experience, likely in a language like Python or maybe basic Java. They're happy to be trained on a special-purpose language for this new platform, but they don't want the code they write to be too closely tied to one particular brand or version of the platform. I expect this user to write *recipes* in MRL and run them through the MRL compiler to generate code to run on their particular hardware (specified by the MPDL specification provided by the hardware manufacturer).

[9] It probably does a lot more than that. My apologies, Aaron.

The second user is a domain expert who has no interest in learning to program. This user wants to take published protocols and do the minimal translation into MPSL specifications, which they will be able to read and be confident in. They will then run the MPSL compiler to generate code for their platform. Alternatively, the published protocols themselves could be specified in MPSL, downloadable from the publication's website or a publicly available database.

Interactive MPAM API

Some users will want to drive the board interactively. The MPAM API will be directly callable from at least Python, allowing interactive control of the board at a level higher than needing to worry about specific electrodes, but lower than platform-independent recipes.

Implementation languages

For the tools themselves, I'm assuming that I can choose the appropriate programming languages for each, and the code doesn't need to be understandable to users who aren't professional programmers. (It does, however, need to be maintainable.) My current expectation is that I will use a combination of C++ and Java and will use ANTLR to generate parsers.

Code quality

My current code quality target is what I'd call "high-quality prototype". It will be as efficient and robust as I can make it, and it will be commented and documented such that others can (hopefully) understand it. As I won't have a Q/A team beating on it, and because it will venture into some areas (e.g., routing algorithms), I'm not an expert in, it will probably not be product quality.

It will certainly be usable and releasable as an open-source reference implementation (if desired), but a larger team with different skill sets would be required to turn it into a revenue-generating product or, likely, even an open-source project that customers of the hardware platform could rely on for their own mission-critical applications.

MPDL: the Microfluidics Platform Description Language

The first language will be a relatively straightforward description language that will be used to describe the characteristics and capabilities of a specific hardware platform. I'm calling this, for the moment,[10] *MPDL*, for *Microfluidics Platform Description Language*.

An MPDL description will capture things like

- The overall size and shape of the pad array (including, possibly, whether it is a square or hexagonal mesh).
- The location of any "dead pads" (e.g., coordinates that contain struts rather than pads.
- The positions, capacities, and capabilities (e.g., heating or cooling ranges) of wells.

[10] I obviously haven't put any thought (yet) into coming up with clever names for these languages.

- The basic drop size, the size of drop dispensed from a well (if different), the capacity of a pad in terms of drops, and the minimum content of a pad to ensure movement.
- The positions (and strengths?) of any electromagnets.
- The positions and ranges of any thermal regions.
- The positions and capabilities of any sensors.
- The basic clock speed of the board and the time required for operations that take more than a single clock tick.
- The names, parameters, and timing of any platform-specific MPAM operations (PSOs) provided by the board.

In addition, it may be useful to allow a description to contain *implementation clauses*, which are snippets of MPAM code that describe how to perform certain operations. This might be useful if a particular board comes with a generic MPAM driver rather than one specifically tailored to it.

Another potential use for implementation clauses would be to allow the description to specify how you can implement MRL operations in terms of PSOs (which the MRL compiler would be otherwise unable to use).

MPDL Platform Descriptions

It's expected that every platform developer will make an MPDL description of the platform available to their users, so most users will never have to think about this language.

I expect that I/we will have MPDL descriptions for OpenDrop as well as different versions of the HP DMF boards under development.

***MPAM*: the Microfluidics Platform Abstract Machine**

Actual operation of a board is described in terms of low-level programs written to the *Microfluidics Platform Abstract Machine* (*MPAM*). Such programs are meant to be executable on a *specific* board, described by MPDL.

At the lowest level, MPAM operations include things like

- Turn on the electrode at pad (5, 2).
- Dispense a drop from well 7 to pad (3, 0).
- Take well 3 to 39 C and hold for 20 sec.
- Turn off the electromagnet at pad (20, 37).
- Check whether the drop at pad (5, 2) is yellow.
- Perform platform-specific operation (PSO) 5 with parameters pad (8, 1) and well 4.

Slightly higher, MPAM has operations that are, strictly speaking, defined in terms of those low-level operations but are sufficiently common to want to reify, especially for interactive use, things like

- Move a drop from pad (5, 3) to pad (5, 4).

- Split the drop at pad (10, 2) between pads (9, 2) and (11, 2).
- Merge the drops at pads (14, 7) and (14, 9) into pad (14, 8).

In one sense, drops at the MPAM level are purely theoretical. "Moving a drop" will likely just mean turning on the electrode at the destination pad, regardless of whether the source pad contains a drop. In another sense, however, it will probably be useful for the MPAM interpreter to keep track of volumes in pads and wells so that it can raise an error condition if it is asked to move a hypothetical drop that doesn't exist. This will also allow higher-level operations like "repeat until well 5 is empty".

Higher still are operations that involve *paths*—sequences of pads characterized by a start pad/well, an end pad/well and a sequence of intermediate pads (or straight segments composed of pads)—and *trains*—sequences of drops, possibly with gaps between them. A train moves along a path until the lead drop reaches the end (unless the end is a well) or until an operation on the train produces another train.

This will allow things like

- Move the drop from pad (21, 4) along path P1 to well 42.
- Dispense 50 µL from well 2, with 2-pad gaps between drops, as train 2.
- Move train 14 along path 25.
- Merge train 2 with train 7 as train 41 at pad (51, 23)
    - This assumes their endpoints of the trains can be merged at that pad
- Split train 19 into train 20 at pad (7, 39) and train 21 at pad (9, 39).
    - This assumes that train 19 ends at pad (8, 39)
- Incubate train 24 along path 48 with thermal region 9 held at 98 C.
- Detect yellow drops in train 4 using sensor 4 and call them train 16.[11]

The time required for a train operation is the amount of time it takes for the last drop in the train to be processed.

In many cases (probably most), the MPAM interpreter or assembler will be to compute a shortest (or at least shortish) path from a train's current endpoint, avoiding any other active trains and having the appropriate length and cadence to make an operation possible. (For example, for merging operations, ensuring that the lead drops for the two trains arrive at the merge point at the same time.)

Next, there is the temporal aspect. MPAM operations will need to be able to specify that they must (or may) happen at the same time as other operations or that they must

[11] Not that such filtering operations will imply that trains may have "missing" drops in them. This will imply that either such trains should not participate in mixing operations or visual surveillance should be used to ensure that there are drops present along both trains. If the latter is used, there will need to be a notion of specifying what to do with the excess from the other train. (That would also be useful in the cases in which the trains aren't of the same length.)

begin (exactly or at least) a certain amount of time after the start or end of another operation.

For state, I presume that most decisions will be based on current properties of the board, e.g., whether a pad is (presumed to be) occupied by a drop, the (presumed) volume of fluid in a well, the (measured or presumed) temperature of a pad or well, or information given by a sensor. But we will also want some notion of global (or perhaps stack-based) scratch registers that can be used to store data, and we will need operations to perform arithmetic on them.

Finally, there are control operations. Much of such notions, I believe, can be accounted for with train-focused operations, but we will also want simple conditionals and loops.

The MPAM API

In addition to a language, there will be a *programmatic API* with function calls for each operation (with the possible exception of control operations). The API will have data types for board components such as pads and wells and for paths and trains, and it might be a good idea to also include data types for things like drops and fluids. I expect the API to be defined for at least Python, Java, and C++.

This API will allow programs written in any of these languages to drive a board directly, and it will also allow users to drive a board interactively in languages like Python.

MPAM API implementations

In order to actually drive a board, somebody will have to write a library that implements the MPAM API in terms of calls to whatever low-level functions are required to interface with a specific board. I presume that such libraries will be provided by board manufacturers or the open-source community, likely written in C++ (or C), with wrappers allowing them to be called from other languages such as Java and Python.

Thylacine will include wrappers for Java and Python (in terms of a C++ implementation) and will include libraries for OpenDrop and HP platforms.

To simplify (and standardize) the creation of MPAM implementation libraries, I plan on writing an incomplete library with default implementations for higher-level operations in terms of lower-level operations (such as turning on and off electrodes), which would still need to be implemented for a specific board. This incomplete implementation may require (or, at least, be able to take advantage of) an MPSL description of the board.

The MPAM assembler

The *MPAM assembler* will be a standalone program which reads an MPAM program and translates it into equivalent code in another programming language which implements the MPAM program in terms of calls to the MPAM API. This code can be run directly as a standalone program or linked together with other code. I plan on implementing the MPAM assembler such that different back ends can be written to emit code for different programming languages.

The MPAM interpreter

The *MPAM interpreter* will read in an MPAM program and execute it by making calls on the MPAM API. The interpreter can be run as a standalone program or linked into other code. (This implies that it needs to be at least callable from various programming languages).

The MPAM emulator

The *MPAM emulator* will allow for the visualization of an MPAM program running on an MPDL-described board. It will read in the MPDL description of the board (and a description of, e.g., the initial contents of various wells) and display a simple[12] representation of the board and implement the MPAM API in terms of making changes to this representation. It will therefore be able to be used by the MPAM interpreter, by code generated by the MPAM assembler, or by users interacting with the MPAM API directly.

The emulator will be useful for trying things out for boards under development or to develop code before hardware becomes available. Since many recipes take a long time to run, I expect that the emulator will allow control of virtual time, allowing things like

- Run the program at 5x speed.
- Run this segment of the program a quarter speed so that it's possible to follow what's going on.
- Suppress (but flag) "and then nothing happens for the next 60 seconds" gaps.

***MRL*: The Microfluidics Recipe Language**

Whereas MPAM programs are entirely concerned with driving boards and their pads, wells, electrodes, and the like, recipes written the *Microfluidics Recipe Language* (*MRL*) don't care about such things at all—and, therefore, can be compiled to run on any platform as long as it provides sufficient resources.

An MRL recipe is, instead, written in terms of *reagents* and *samples*.[13] It will start by declaring the reagents available at the beginning (and, if necessary, their thermal storage requirements). Recipes include operations like

- Draw 70 µL of reagent 1 as sample 4.
- Mix sample 4 with 50 µL of reagent 2 to form (by default, 120 µL of) reagent 3 as sample 5.
- Incubate sample 5 at 90 C for 10 seconds
- If sample 7 is yellow, …
- Repeat 10 times: …

It will probably be worthwhile to flag reagents that are considered to be the output of the recipe (so that they are not inadvertently consumed or discarded).

---

[12] Likely *very* simple. I'm not a graphics person.

[13] I'm not sure "sample" is the right word here. What I mean is "a particular quantity of a particular reagent, possibly split among several pads and wells, but with the same history"

Recipes will also specify *temporal constraints*, either by explicitly declaring several steps to be performed in sequence or by declaring that *synchronization points* (either explicitly specified or implicitly given as the beginning or end of recipe steps) must take place in a specified temporal order, optionally with minimal and/or maximal time bounds. Anything not so specified can be assumed to be parallelizable and reorderable.

The MRL compiler

The *MRL compiler* will be a standalone program that takes as input an MPDL description of a platform and one or more MRL recipes and outputs an MPAM program equivalent to running all of the recipes on that platform—or inform the user that it isn't possible to run the recipes on the platform. The intent is that it should be possible to use the same board to run several recipes (e.g., perform several tests) at the same time, ideally substantially in parallel.[14]

While the compiler will attempt to find a substantially optimal equivalent MPAM program, I expect that for most programs, the overall runtime will be swamped by incubation operations of seconds or minutes, so I don't plan on spending too much time on optimizations other than trying to find reasonably short paths and assigning nearby wells.

In addition to the MPAM program, the compiler will output a specification of which input reagents (in what quantities) should be placed in each well at the beginning of the program and which output reagents (in what quantities) can be found in each well at the end of the program. Unless otherwise instructed, the compiler will be licensed to

- determine the required (and, therefore, expected) initial quantities of reagents,
- combine identical reagents (for different recipes) into the same input well, and
- split the initial quantity of a given reagent among multiple wells.

***MPSL*: The Microfluidics Protocol Specification Language**

It may well be that MRL recipes can be made sufficiently close to the way chemists talk about their protocols that they can be written (and read) even by people with no experience in programming. If that isn't the case, the next step is to design a *Microfluidics Protocol Specification Language* (*MPSL*) whose design goals include

- Syntax that is essentially[15] English sentences.
- Non-rigid syntax, allowing multiple ways to say the same thing.
- Immediately readable by practitioners in the field with no training, to the point where you could expect one to look at a protocol, point to a line, and say, "No, that's wrong".

[14] I'm presuming that this is a useful use case and worth pursuing. If not, focusing on a single recipe would be a bit simpler, but not much, as multiple recipes can be easily combined into a single recipe with no temporal constraints between them.

[15] Note that I don't intend to do actual natural language processing. This will still have a deterministically-parsable grammar. It will just look like sentences.

- Easily writable by practitioners with very minimal training.

The MPSL compiler

The *MPSL compiler* will be a standalone program that translates a protocol specified in MPSL into an MRL recipe (which can then be compiled further into an MPAM program for a given platform). I suspect that the only hard part about this compiler—and it will be difficult—will be writing the parser. The actual translation should be exceedingly straightforward.