The Kaemika* Apparoach to Integrated Modeling of Reaction Networks and Protocols

Luca Cardelli

Department of Computer Science, University of Oxford

December 2019

1 Introduction

The classical cycle of observation, hypothesis formulation, experimentation, and falsification, which has driven scientific and technical progress since the scientific revolution, is lately becoming automated in all its separate components. Data gathering is conducted by high-throughput machinery. Models are automatically synthesized, at least in part, from data. Experiments are selected to maximize knowledge acquisition. Laboratory protocols are run under reproducible and auditable software control. However, integration between these automated components is lacking. Theories are not placed in the same formal context as the (coded) protocols that are supposed to test them: theories talk about changing quantities, while protocols talk about steps carried out by machines. Neither knows about the other, although they both try to describe the same process. The consequence is that often it is hard tell what happened when something fails: was it an error in the model, or an error in the protocol? In parameter inference, both may have unknown parameters that must be fit to data. When all those activities are automated, we need a way to answer those questions that is equally automated.

We should ideally start from an integrated description from which we can extract both the model of a phenomenon, for possibly automated mathematical analysis, and the steps carried out to test it, for automated execution by lab equipment. This is essential to carry out automated model synthesis and falsification by taking into account uncertainties in the both model structure and in equipment tolerances.

*/'kimika/

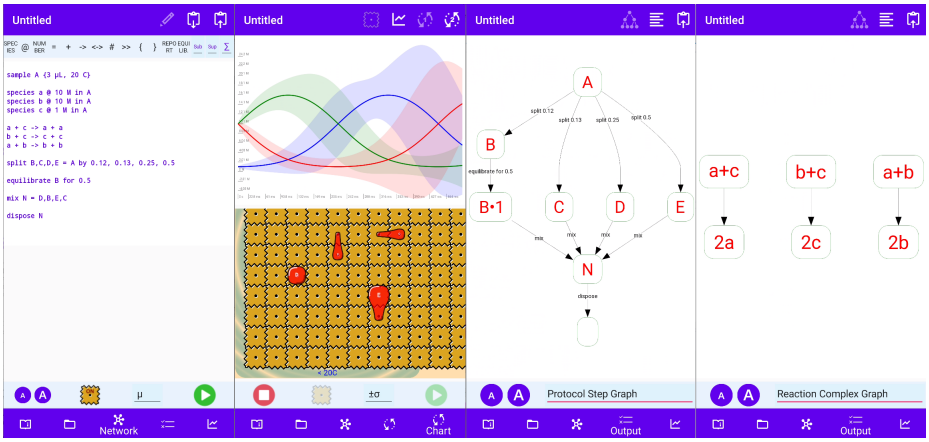

Figure 1: Output of a Kaemika program under Android [4]. **Left:** Source code for a simple chemical oscillator and some liquid handling operations. **Center left:** A view of a stochastic simulation (top) followed by droplet movement on a simulated digital microfluidics device (bottom). **Center right:** A view of the liquid-handling protocol. **Right:** A view of the chemical reaction network inside the droplets.

Here we present one such modeling framework, rooted in languages for chemical reaction networks [13] [16] [14], and for lab protocols [2]; implementations are available at [4], with sources in [5].

Integrated modeling of networks and protocols is not common because, on one hand, reaction networks are usually observed from nature (or at best are engineered from natural components), while on the other hand protocols are meant to be designed. But the opposite situation is also possible: reaction networks can be engineered from scratch [19], while protocols can be seen as something that ”spontaneously” happens [18]. In general, it seems useful to be able to codify networks and protocols in a unified way, to be able to describe them, analyze them, and automate them as a whole.

2 The Kaemika Language

2.1 Main properties

The Kaemika modeling language is a *function*, *nominal* language with a *monadic* semantics, whose purpose is to describe dynamical systems and the way they are manipulated in a wet lab.

2.1.1 Functional aspects

The main structuring abstraction consists of functions that take values as parameters and produce values as results. Values, which are dynamically typed, include functions: these higher order capabilities are based on lexical scoping of variables.

A fairly conventional syntax allows for the definition and use of functions over basic values such as numbers and, less conventionally, chemical species:

```
function F(number x, species a) { x * observe(a) }
species A @ 3 mM   // A at 0.003 Molar concentration
number n = F(2,A)  // n = 0.006
```

A similar syntax allows for encapsulating networks of chemical reactions:

```
network N(species a, number r) { a ->{r} a + a }
species A @ 3 mM
N(A,2)   // produces the 1-reaction network A ->{2} A + A
```

A *network* is nothing more than an abbreviation for a function that returns a trivial value. Still, a network (or a function) can *produce* chemical reactions. The way in which species and reactions can occur in functional computation, and the sense in which a network of reactions can be 'produced', is explained below regarding nominal values (for species) and monadic output (for reactions).

Species are values that can be passed to functions and networks, and returned from functions. Reactions are not values, but reaction-producing networks (like `N` above) are values that can be passed to functions and networks, and returned from functions. Functions can be constructed from basic values (booleans, numbers, species) via boolean and arithmetic operators, conditionals, lists, and recursion. This way, reaction networks can be generated whose number of species and reactions depends on parameters. The nominal aspects of the language are essential for making sense of dynamically generated species.

2.1.2 Nominal aspects

An unbounded number of distinct *species* can be generated during a computation via *species definitions* such as `species A @ 3 mM`. Any such definition generates a new species with a unique *species-name* and at the same time binds that species to a *variable* (`A`). The species is uniquely characterized by its species-name, which however can never be known or mentioned. The variable denoting the species can be mentioned within its lexical scope, but the species itself is a value that can escape that scope. Hence a species-name has an identity that goes beyond the lexical name of the variable (*c.f.* the nominal λ-calculus [10]). A `report` instruction can be used to mark dynamically generated species for plotting during a later simulation.

```
network N(species a) {
   species B @ 1 mM   // a species declaration
   a + B ->{1} a      // repeated at each invocation of N
   report B           // reporting *that* species
}
species A @ 3 mM     // another species declaration
N(A)   // produces a reaction A + B ->{1} A
N(A)   // produces a different reaction A + B•1 ->{1} A
```

Here two different reactions are produced because a (species-name for) `B` is generated each time the network abstraction is invoked. Each `B` is subject to its own `report`, so they can both be plotted at a later time. When displayed as outputs, they are distinguished by a decoration like `B•1` based on the original variable name. These decorations are not part of the syntax, and are used only to distinguish species with distinct species-names that happen to have been initially bound to equally named variables.

2.1.3 Monadic aspects

Kaemika functions return (primarily) values, but can also produce (on the side) an output stream of chemical actions. These actions can involve the simulation of a chemical reaction network that has been produced. In fact, computation and simulation can be interleaved: see the section on Protocols.

More precisely, an *output monad* is implicit in the language, and explicit in its execution [17]. Given a result type T and a collection $Action$ of output actions, including declarations of new species and chemical reactions between them, the output monad $(Out(T), \eta \in T \to Out(T), \Rightarrow \in (Out(T), T \to Out(U)) \to Out(U))$ is given by:

$$
\begin{aligned}
&Out(T) = T \times List(Action)\\
&\eta(x) = (x, [])\\
&\Rightarrow\ = \lambda((x,s),f)\,\mathit{let}\,(y,t) = f(x)\,\mathit{in}\,(y, s{+}{+}t)
\end{aligned}
$$

where $[]$ is the empty list and $++$ is list concatenation.

Via $(\eta, \Rightarrow)$ we can interpret a functional computation intermixed with actions. For example, a basic functional computation $(\lambda(x,y)\,x+y)(3,4)$ is interpreted in the output monad as $(\lambda(x,y)\,x \Rightarrow \lambda x'\,y \Rightarrow \lambda y'\,\eta(x'+y'))(\eta(3),\eta(4)) = (7,[]) = \eta(7)$. The occurrence of an output action $a \in Action$ can then be interpreted as $emit(a) = (nil, [a])$, where $emit \in Action \to Out(Nil)$, yielding a null result value and a singleton output stream. In a dynamically typed language all

values have effectively a single type $Value$, so the output monad used here is $Out(Value) = Value \times List(Action)$, which represents the outcome of any computation.

2.2 Flows

A *flow*, in the sense of time-flow, is a function of time represented as a data structure rather than as a proper function. The handling of flows is always staged: in a first stage the flow is assembled, and in a later stage, after its computation has completed, the flow is used. A flow can be assembled by any computational means available, including paremeterization, conditional execution, and recursion.

Flows are used in three contexts. (1) in *report* statements to describe functions of time that should be plotted: those functions typically depend on the concentrations of species over time. The flows themselves are used as labels in the plot legend. (2) to describe *general rates* for reactions, in particular ones that may be time-dependent in a non-mass-action way. This enables also fixing arbitrary functions of time as concentration profiles for selected species. (3) to *observe* the results of chemical evolution, and to feed those back into algorithms (this is discussed in the section on Protocols). When a flow is used, it is typically queried at some frequency from time 0 to some time bound.

Syntactically, a flow is represented using the same syntax as expressions, but the interpretation of these expressions is different in an normal context vs. a *flow context* (i.e. (1),(2),(3) above). For example, in a normal context a variable *x* bound to a species denotes that species, but in a flow context it denotes the concentration of that species at a given time.

The simplest flow is `time`, representing the identity function $\lambda t\, t$. Other flows can be produced by arithmetic operators like `sin(time)`, representing $\lambda t\, sin(t)$, or by constants like `pi`, representing $\lambda t\, \pi$: these can be useful as reference lines in plots.

The most common flows involve species: `a*b` = $\lambda t\, a(t) * b(t)$, is the product of the concentrations of `a` and `b` at any given time, and `cond(a<b,a,b)` = $\lambda t\, if\, a(t) < b(t)\, then\, a(t)\, else\, b(t)$, is the minimum of the concentrations of `a` and `b` at any given time. The conditional flow operator `cond` should be contrasted with the ordinary boolean conditional `if x then a else b end` which, still in a flow context, assembles either the flow $\lambda t\, a(t)$ or the flow $\lambda t\, b(t)$, depending on the (timeless) boolean value of variable `x`.

In conjunction with Linear Noise Approximation simulation, stochastic flows can be used. `var(a)` denotes the variance of species `a` over time, and `cov(a,b)` the covariance. Here `a` and `b` need not be single species, but are restricted to linear combination of species, so that they can be correctly evaluated; for example `var(a-b)`

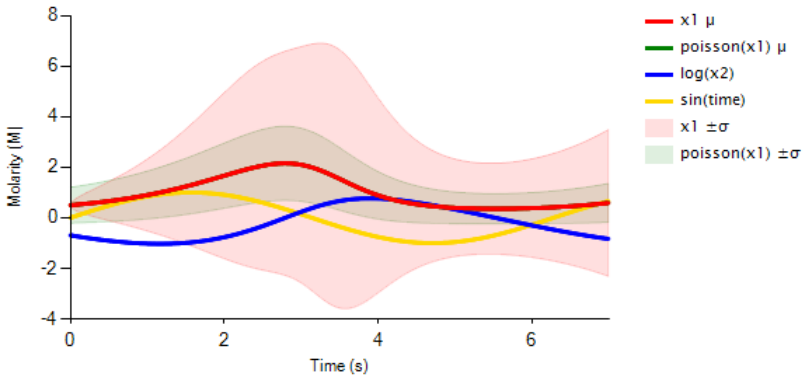

Figure 2: Reporting the flows `x1`, `poisson(x1)`, `log(x2)`, `sin(time)`

= `var(a)+var(b)-2*cov(a,b)`. A useful stochastic flow is `poisson(a)` which, on a time plot, paints Poisson noise centered around the mean of `a`: it can provide a visual benchmark to interpret `var(a)`.

2.3 Simulation of chemical kinetics

A chemical reaction network can be simulated [7] by the `equilibrate` instruction over a given time period. Linear Noise Approximation (LNA) simulation can be enabled, providing both mean and variance of species concentrations. Equilibrate instruction can be preceded by `report` instructions to select the flows to be plotted (by default, all the single species flows are plotted). Report and equilibrate, which are themselves *Actions*, can occur anywhere within a computation, and are affected by previous actions.

Figure 2 shows the LNA output of this simple script:

```
species x1 @ 0.5 M      // prey
species x2 @ 0.5 M      // predator
x1 -> x1 + x1           // prey reproduces
x1 + x2 -> x2 + x2      // predator eats prey
x2 -> #                 // predator dies
report x1, poisson(x1), log(x2), sin(time) // flows (what to plot)
equilibrate for 7                          // simulate and plot
```

Apart from simulation, differential equations for the species concentrations can

be produced, including equations for covariances from the LNA:

$$\begin{aligned}
\partial_t x1 &= x1 - x1 * x2 \\
\partial_t x2 &= x1 * x2 - x2
\end{aligned}$$

$$\begin{aligned}
\partial_t var(x1) &= 2 * var(x1) - cov(x1, x2) * x1 \\
&\quad - cov(x2, x1) * x1 - 2 * var(x1) * x2 + x1 + x1 * x2 \\
\partial_t cov(x1, x2) &= cov(x1, x2) * x1 - var(x2) * x1 \\
&\quad - cov(x1, x2) * x2 + var(x1) * x2 - x1 * x2 \\
\partial_t var(x2) &= 2 * var(x2) * x1 + cov(x1, x2) * x2 \\
&\quad + cov(x2, x1) * x2 - 2 * var(x2) + x1 * x2 + x2
\end{aligned}$$

LNA simulation handles, numerically, both mass action rates and flows; that is, arbitrary rate functions can be used in stochastic simulation. When requesting symbolic LNA output, as above, flows appearing in reaction rates are handled as long as they are differentiable.

2.4 Visualization of chemical reaction networks

Chemical reaction networks are usually visualized as either (A) graphs whose nodes are complexes (left hand sides and right hand sides of reactions), and whose edges represent the related reactions, or (B) as multigraphs whose nodes are species and whose multiedges represent the reactions. In the biological literature a multitude of other informal representations are used, the vast majority of which are specific to a single article and non-invertible (it is not possible to exactly recover the reaction network from the graph). In the bioinformatics literature, invertible representation have been achieved and standardized for use in tools. Still, the automated layout of such graphs, even when using state-of-the-art algorithms, is highly unsatisfactory in the sense of not bringing out the symmetries of the network, and awkward in the sense of requiring constant panning and zooming.

Kaemika uses a new graphical representation of directed multigraphs with multiplicities, which are those needed to unambiguously represent chemical reactions. In first instance, the problem is the same as visually representing Petri nets. Even here we appear to be making an original contribution, although the mapping of Petri nets to our representation is fairly straightforward. Beyond that, catalytic aspects of chemical reaction networks are important to convey meaning, and in this area we introduce a more compact visual notation that extends the basic one.

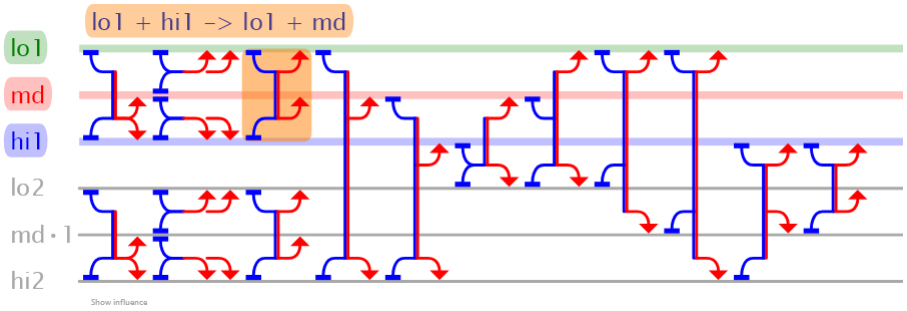

Figure 3: Mozart style. This network is a chemical oscillator consisting of two similar subnetworks interconnected in an alternating pattern, with 6 species and 16 reactions. Blue/blunt tips are reagents, red/sharp tips are products, and vertical (or empty) stems are reactions. An optional bin-packing has been applied, so that the two similar subnetworks are stacked on top of each other on the left. A reaction can be highlighted by hovering over it: its standard chemical wording is then shown at the top, and the species involved in the reaction are highlighted as well, according to a color scheme for reagents (blue) products (red) and catalysts (green). The highlighted lines point out connections between the highlighted reaction and other reactions.

We call this new representation a *reaction score*: like a musical score it has a set of horizontal lines, each associated with a chemical species rather than a frequency. Reactions are added to the score in vertical orientation. Neither the horizontal or vertical orders are important (unlike in musical notation), and it is useful to be able to manually or automatically reorder species and reactions to cluster them in different ways.

The basic Petri-nets capable representation is shown in Figure 3. Each chemical reaction is seen as a Petri net transition from input (blue) blue species to output (red) species. Each Petri net place (species) is stretched out into a horizontal line, and the transitions are laid out vertically as *stems*. Otherwise, the connectivity of the underlying Petri net is respected by connecting transitions to places as appropriate. Each reaction occupies only a bounded horizontal region, the width of which is given by the multiplicities of the incoming and outgoing edges. This results in a regular and compact layout that fits naturally into a rectangular region of a display. The area occupied is given by the number of species, vertically, times the number of reactions, horizontally, each adjusted by its multiplicities. In our implementation we always rescale the whole score to fit exactly in the available space, but zooming and panning is then available.

Just to give it a name, we call this first representation style ‘Mozart’. The next

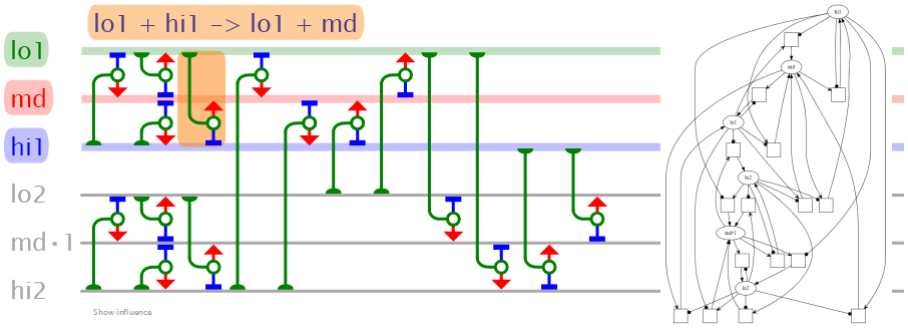

Figure 4: Bach style. This is the same network as in Figure 3; we can now see clearly its reliance on catalytic reactions. This representation is more compact, in part due to straightening connections onto the stem. We refrain from similarly compacting the Mozart case, which is meant mostly as a tutorial for the Bach style. All the stems have been reduced to size 1, but this depends on the vertical order of the species, which can be changed by dragging. On the right, for comparison, is the same multigraph rendered by GraphViz, with ovals for species and squares for reactions, already reduced as in Bach style via catalytic edged (round tips).

step is 'Bach', where we factor out the catalysts from the reactions. Each reaction $A \to B$ is first recast in the form $C >> A' \to B'$ where for each species s if $n * s$ occurs in A and $m * s$ occurs in B, then $min(n, m) * s$ are moved into C, and the rest are left in A' or B' (not both). The reaction $A' \to B'$ is laid out in Mozart style (with some additional compaction of connections onto the vertical line of the stem, when possible). Additional catalytic connections, using a different visual style (green), are introduced between the species in C and the *stem* of $A' \to B'$. That is all, modulo taking care of some special cases when A' or B' are empty, and routing all the connections so that they do not interfere with each other (Figure 4).

We emphasize that this representation is complete (any reaction network can be automatically laid out) and unambiguous (the original reaction network can be recovered from it, except for the reaction rates and initial conditions).

3 Protocols

Chemical reaction networks can be localized to *samples* (representing, e.g., test tubes or droplets), and combined into protocols that perform the equivalent of liquid handling in a wet lab. Samples are values in the language: there can be an unbounded number of them, and their names are handled in the *nominal* style similarly to species.

3.1 Samples

A *sample* is a named container for chemical species characterized by a volume and a temperature for a quantity of water in which the chemicals are dissolved. Both the volume and temperature of a sample are assumed to remain constant over time, but samples can be turned into other samples with different volumes and temperatures.

```
sample A {1 µL, 20 C}
sample B {2 µL, 37 C}
```

The same chemical species can occur in different samples; to this end species should first be declared without referring to their initial concentrations:

```
species {a, b#10}         // b has molar mass of 10 g/mol
```

Here `b` is assigned a molar mass of 10, while `a`'s molar mass is unspecified. Those species can then be added to various samples, in specified amounts:

```
amount a @ 0.001 nmol in A    // initialized in mol
amount a @ 1 µM in B          // initialized in Molar = mol/L
amount b @ 0.02 ng in B       // initialized in grams
```

These initializations along with the previous declarations result in 1 µM of `a` in both `A` and `B`, and 1 µM of `b` in `B`. Whenever the dimension of an initial quantity is Molar, the assigned value directly determines the concentration within the sample. When the dimension is mol or grams, the outcome depends on the volume of the sample, and in the case of grams, it requires a declaration of the molar mass of the species.

Chemical reactions should be listed after the samples that they are meant to affect, but they do not reference samples explicitly. Each reaction applies to all the *relevant* samples, that is to all previously declared samples that contain some `amount` (even if zero) of all the species that are involved in the reaction:

```
a -> #           // relevant to samples A and B
b -> #           // relevant to sample B but not sample A
a ->{2} a + b    // relevant to sample B but not sample A
```

(`#` is the empty multiset of species, `{2}` is a reaction rate, defaulting to `{1}`.)

We have seen that the volume of a sample can affect the concentration of the species placed into it. The temperature of a sample, instead, can affect the rates of the reactions. A reaction rate like `{r}` indicates a constant (temperature-independent) rate where r is the *productive collision frequency* of the reagents, which is multiplied by the concentration of the reagents to obtain the usual mass action kinetics.

A reaction rate like `{r,a}` instead indicates a collision frequency of r, as before, and an *activation energy* of a. These quantities feed into the temperature-dependent Arrhenius' formula, $re^{-\frac{a}{RT}}$ where R is the universal gas constant, and T is the temperature of the sample. The resulting value is again multiplied by the concentrations of the reagents to obtain a temperature-dependent mass action kinetics.

In the initial sections we used species and reactions, but no samples. In fact, there is a default sample called `vessel` that collects all the species that are not assigned to other samples. The notation we used previously, `species a @ 1 mM` is thus an abbreviation for `species a @ 1 mM in vessel`, and that in turn is an abbreviation for `species {a}; amount a @ 1 mM in vessel`. Similarly, if a species is used in a single sample it can be declared and initialized at once, as in: `species a @ 1 mM in A`.

3.2 Liquid handling

In addition to sample preparation, explained in the previous section, a number of operations can be applied to samples.

A sample `A` can be split into a number of other samples `B,C,D`:

```
split B, C, D = A
```

The resulting samples have the same temperature as `A`, and their volumes sum up to that of `A`. The concentration of the species from `A` remains the same in the new samples. The above instruction indicates an equal split, but proportional splits can be expressed as `split B, C, D = A by 0.5, 0.3, 0.2` (the last proportion can be omitted).

A number of samples can be mixed into one:

```
mix E = B, C
```

The temperature of the resulting sample is the volume-weighted average of the contributing samples, and its volume is the sum of their volumes. The concentrations of the species from `B` and `C` become typically diluted, contributing to `E` proportionally to the ratio of their volumes to the volume of `E`.

Samples can be discarded:

```
dispose D
```

Mix, split, and dispose operations, as well as sample preparation, happen instantly, that is without any reactions firing within the samples. Another operation allows time to pass in a sample (or in a collection of samples in parallel), while any other samples are assumed to remain unaffected:

```
equilibrate F = E for 10
```

During the next 10 second, the initial state of `E` is allowed to evolve according to the kinetics of the relevant reactions over the species contained in `E` (it need not actually reach equilibrium). This evolution is carried out by integrating the differential equations resulting from the chemical reactions, starting from the initial conditions of `E`. At the end of the integration, a new sample `F` is produced whose concentrations equal the final ones of `E`. LNA stochastic state is propagated through multiple simulations, as well as through mix and split operations.

The input samples of all these operations are *consumed*, that is rendered unusable to any further protocol operations, but they can remain available for observation. Samples can be *observed* by applying flows to them; an observation always yields a number:

```
number n = observe(f, F)
```

Here `f` is a *flow* that provides a way of observing the contents of sample `F`. For example, `observe(a, F)` yields the concentration of `a` in `F` (at the current time in `F`'s timeline, which in the example above is time 0 because `F` has not been further equilibrated), and `observe(time, E)`, `observe(volume, E)`, `observe(kelvin, E)` yields the current time in `E`'s timeline (which in the example above is the final simulation time of `E`) and its volume and temperature. It is possible to use `observe` within a flow, e.g. to make a reaction rate in a sample depend on the state of a different sample.

Subsequently, we can keep applying protocol operations to `F` and its derivates, including further equilibrations. All these operations on samples can be freely embedded in functions and networks, and combined algorithmically, providing the ability to encapsulate complex parameterized protocols.

Protocol operation provide some of the output monad *actions* discussed earlier. For example the execution of `mix E = B, C` produces an action $emit(mix\, E = B, C)$ which records all the details of the three samples, before and after the mixing. In the final $List(Action)$ we obtain a detailed trace of the protocol steps that have been executed, and their effects. This trace can be displayed, e.g., as a graph, or otherwise inspected (Figure 1).

3.3 Digital microfluidics

The Kaemika system provides an interpreter for the Kaemika language, including simulation and plotting for the evolution of chemical reaction networks. In addition, it provides a virtual liquid handling device for the simulation and visualization of protocols. Many flavors on liquid-handling devices exist, but we focus on digital

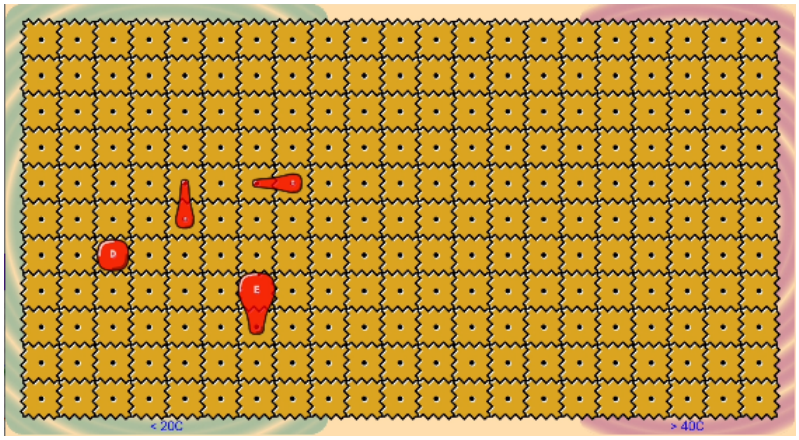

Figure 5: Simulated digital microfluidics device.

microfluidics because of its generality, simplicity, and programmability, in that a single device can execute all basic liquid-handling protocols (within practical size limitations) [3] [1] [20], and can support a number of options for observation of the samples [8].

A Kaemika protocol contains information both about the kinetics of reactions that naturally occur within samples, and about laboratory manipulations performed on samples. The two are linked because lab operations affect concentrations, volumes, and temperatures, which affect kinetics. Correspondingly, the execution of a kaemika protocol intertwines the simulation of individual reaction networks with the microfluidic manipulation of the samples, including intertwining the plotting of simulations and the visualization of liquid handling. The state of a sample at the end of a chemical simulation is propagated to the following liquid handling operations, and vice versa.

A typical digital microfluidics device has a rectangular array of electrically controlled pads, and some means of adding and removing liquid droplets over its surface. Injection and extraction may by done hand, or by extruding standard-size droplets from larger on-device reservoirs, or by pumps at the device's periphery. The standard droplet size is around 1μL. Droplets can be moved by changing the electrical properties of adjacent pads, and multiple droplets can be moved in parallel. Droplets can be merged by causing one to move over the pad of another, and split by electrically pulling them in opposite directions. An overhead camera or an on-surface sensing apparatus may provide feedback about the position of the droplets.

In the Kaemika droplet simulation, sample preparation is done off-device, and

each completed sample is represented by a droplet on the device. Mixing, splitting, and disposing of samples is handled by appropriate routing of the droplets over the device surface: this is automatic, and does not require further annotations.

Some further assumptions are needed for equilibration, for observation, and for the handling of temperatures and volumes. We assume that a region of the device is maintained at a *cool* temperature. All the staging and mixing operation are executed in this region, because chemical reactions are not supposed to be happening during liquid handling, and a cool temperature can approximate that condition. We also assume that another region of the device is maintained at a *hot* temperature, and an intermediate region is at *warm*, ambient temperature. Equilibrate operations move droplets into one of the warm or hot regions, according to need, hold them there for the prescribed time, and then them move them back to the cool region.

The `mix` operation can be used to dilute a sample, by mixing it with pure water, and even to change its temperature by mixing it with hot or cold water. But this is not very convenient because a change of temperature thus requires a change in volume or concentration. Moreover, the inverse of dilution, evaporation, is not available. To help with the handling of temperatures, a `regulate` instruction can be used that does not affect sample volume:

```
regulate A to 25 C
```

This simply moves a droplet to a region appropriate for that temperature, and may instruct the device to achieve that temperature. For the handling of volumes, more speculatively, we allow a `concentrate` operation that causes a change of volume by removing (evaporating) or adding (diluting) water, without affecting temperature:

```
concentrate B to 1 µL
```

Finally, we already discussed using `observe(f, A)` to observe the properties of a sample during the execution of a protocol, possibly then altering the protocol in response. It is useful, for example, to iterate a subprotocol until some concentration is achieved. The kind of observations that are practical depend on the hardware capability of a device; there are for example several optical means to detect particles or concentrations. More powerful observation (mass spectroscopy [9], nuclear magnetic resonance [11], DNA hybridization detection [15], DNA amplification [6], DNA sequencing [20], single molecule detection [12]), require separate techniques, although many of these can be integrated or carried out on the microfluidic device itself. The Kaemika language supports powerful observation based on concentrations, but their practicality will always depend on the particular hardware that is available.

3.4 Example

In the following example, we prepare a sample A, we let it react for a bit, we prepare a sample B, we let it react for a bit, we then mix A with B into C, and we let C react by the union of the species and reactions of A and B. Both A and B are relatively inert, but an oscillation is triggered after they are mixed into C.

```
species {c}                 // occurs in multiple samples

sample A {1µL, 20C}         // prepare sample A
species a @ 10mM in A       // species initially local to A
amount c @ 1mM in A
a + c -> a + a              // reactions in A
equilibrate A for 100       // let A evolve

sample B {1µL, 20C}         // prepare sample B
species b @ 10mM in B       // species initially local to B
amount c @ 1mM in B
b + c -> c + c              // reactions in B
equilibrate B for 100       // let B evolve

mix C = A, B                // mix A and B
a + b -> b + b              // further reactions

equilibrate C for 1000      // let C evolve by all reactions
```

From this unified script we can extract the dynamical system for each sample, separately from the protocol:

```
KINETICS for STATE_0 (sample A) for 100 time units:
∂a = a * c
∂c = - a * c

KINETICS for STATE_1 (sample B) for 100 time units:
∂b = - b * c
∂c = b * c

KINETICS for STATE_3 (sample C) for 1000 time units:
∂a = a * c - a * b
∂c = b * c - a * c
∂b = a * b - b * c
```

We can also extract the sequence of protocol steps, separately from the dynamical systems:

```
sample A {1µL, 293.2K}
amount a @ 10 mM in A
amount c @ 1 mM in A
equilibrate A•1 = A for 100

sample B {1µL, 293.2K}
amount b @ 10 mM in B
amount c @ 1 mM in B
equilibrate B•1 = B for 100

mix C = A•1, B•1
equilibrate C•1 = C for 1000
```

References

[1] Printeria. http://2018.igem.org/Team:Valencia_UPV.

[2] Abate A., Cardelli L., Kwiatkowska M., Laurenti L., and Yordanov B. Experimental biological protocols with formal semantics. In Češka M. and Šafránek D., editors, *Computational Methods in Systems Biology*, volume Lecture Notes in Computer Science 11095, pages 165–182. Springer, 2018.

[3] Mirela Alistar and Urs Gaudenz. Opendrop: An integrated do-it-yourself platform for personal use of biochips. *Bioengineering (Basel)*, 4(2):45, 2017.

[4] Luca Cardelli. Kaemika apps. Windows: https://www.microsoft.com/en-us/p/kaemika/9n258rnwv8pr. Android: https://play.google.com/store/apps/details?id=com.kaemika.Kaemika&hl=en_GB. iOS: https://apps.apple.com/app/id1491803017. macOS: https://apps.apple.com/us/app/kaemika/id1493299038.

[5] Luca Cardelli. Kaemika software sources. https://github.com/luca-cardelli/KaemikaXM.

[6] Beatriz Coelho, Bruno Veigas, Elvira Fortunato, Rodrigo Martins, Hugo Águas, Rui Igreja, and Pedro V. Baptista. Digital microfluidics for nucleic acid amplification. *Sensors*, 17(7):1495, 2017.

[7] Neil Dalchau. Open solving library for ODEs. https://www.microsoft.com/en-us/research/project/open-solving-library-for-odes/.

[8] Sergio L.S. Freire. Perspectives on digital microfluidics. *Sensors and Actuators A: Physical*, 250:15–28, 2016.

[9] Andrea E. Kirby and Aaron R. Wheeler. Digital microfluidics: An emerging sample preparation platform for mass spectrometry. *Analytical Chemistry*, 85(13):6178–6184, 2013.

[10] Roy L.Crole and Frank Nebel. Nominal lambda calculus: An internal language for fm-cartesian closed categories. *Electronic Notes in Theoretical Computer Science*, 298:93–117, 2013.

[11] Ka-Meng Lei, Pui-In Mak, Man-Kay Lawa, and Rui P. Martinsab. NMR–DMF: a modular nuclear magnetic resonance–digital microfluidics system for biological assays. *Analyst*, 139:6204–6213, 2014.

[12] Kühnemund M, Witters D, Nilsson M, and Lammertyn J. Circle-to-circle amplification on a digital microfluidic chip for amplified single molecule detection. *Lab on a Chip*, 16:2983–2992, 2014.

[13] Pedersen M. and Plotkin G. A language for biochemical systems. In Heiner M. and Uhrmacher A.M., editors, *Computational Methods in Systems Biology*, volume Lecture Notes in Computer Science 5307, pages 63–82. Springer, 2008.

[14] Vasic M., Soloveichik D., and Khurshid S. CRN++: Molecular programming language. In Doty D. and Dietz H., editors, *DNA Computing and Molecular Programming 24*, volume Lecture Notes in Computer Science 11145, pages 1–18. Springer, 2018.

[15] Lidija Malic, Teodor Veres, and Maryam Tabrizian. Biochip functionalization using electrowetting-on-dielectric digital microfluidicsfor surface plasmon resonance imaging detection of dna hybridization. *Biosensors and Bioelectronics*, 24:2218–2224, 2009.

[16] Michael Pedersen and Andrew Phillips. Towards programming languages for genetic engineering of living cells. *Journal of the Royal Society Interface*, 6:S437–S450, April 2009.

[17] Tomas Petricek. What we talk about when we talk about monads. *The Art, Science, and Engineering of Programming*, 2(2):12, 2018.

[18] Santiago Schnell. Ten simple rules for a computational biologist's laboratory notebook. *PLoS computational biology*, 11:e1004385, 09 2015.

[19] David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

[20] Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. Puddle: A dynamic, error-correcting, full-stack microfluidics platform. In *ASPLOS'19*.