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 to tell what happened when they do not match: was it an error in the model, or an error in the protocol? Often both the model and the protocol have unknown parameters: do we use the experimental data to fit the model or to fit the protocol? When most activities are automated, we need a way to answer those questions that is equally automated. We would like to start from an integrated description from which we can extract both the model of a phenomenon (for possibly automated mathematical analysis), and the step carried out to test it (for automated execution by lab equipment). This is essential if we want to carry out automated model synthesis and falsification by taking into account uncertainties in the both model structure and in equipment tolerances.

Here we present one such modeling framework, rooted in languages for chemical reaction networks [12] [15] [13], and for lab protocols [2]. Implementations are available at [4], [5].
2 The Language

2.1 Main properties

The Kaemika language is a function, nominal language with a monadic semantics.

2.1.1 Functional aspects

Functions that take values as parameters and produce values as results are the main structuring abstraction. Values, which are dynamically typed, include functions. Variables are lexically scoped.

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

```plaintext
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:

```plaintext
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 \( \lambda \)-calculus [9]). 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

KateMika 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 [16]. Given a result type $T$ and a collection $\text{Action}$ of output actions, including declarations of new species and chemical reactions between them, the output monad $(\text{Out}(T), \eta \in T \rightarrow \text{Out}(T), \Rightarrow \in (\text{Out}(T), T \rightarrow \text{Out}(U)) \rightarrow \text{Out}(U))$ is given by:

\[
\text{Out}(T) = T \times \text{List}(\text{Action})
\]

\[
\eta(x) = (x, [])
\]

\[
\Rightarrow = \lambda((x, s), f) \text{ let } (y, t) = f(x) \text{ in } (y, s++t)
\]

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

Via $(\eta, \Rightarrow)$ we can interpreted a functional computation intermixed with actions. For example, a basic functional computation $(\lambda(x, y) \cdot x + y)(3, 4)$ is interpreted in the output monad as $(\lambda(x, y) \cdot x \Rightarrow \lambda x' y \Rightarrow \lambda x'' \cdot \eta(x' + y''))(\eta(3), \eta(4)) = (7, []) = \eta(7)$. The occurrence of an output action $a \in \text{Action}$ can then be interpreted as $\text{emit}(a) = (\text{nil}, [a])$, where $\text{emit} \in \text{Action} \rightarrow \text{Out(Nil)}$, yielding a null result value and a singleton output stream. In a dynamically typed language all values have effectively a single type $\text{Value}$, so the output monad used here is $\text{Out(\text{Value})} = \text{Value} \times \text{List}(\text{Action})$, 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 structures 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 parameterization, conditional execution, and recursion.

Flows are used in three contexts. (1) in report statements to describe function of time that should be plotted: those function 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. (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 \( t \), representing the identity function \( \lambda t t \). Other flows can be produced by arithmetic operators like \( \sin(t) \), 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: \( a \cdot b = \lambda t a(t) \cdot b(t) \), is the product of the concentrations of \( a \) and \( b \) at any given time, and \( \text{cond}(a < b, a, b) = \lambda t \text{if } a(t) < b(t) \text{ then } a(t) \text{ else } b(t) \), is the minimum of the concentrations of \( a \) and \( b \) at any given time. The conditional flow operator \( \text{cond} \) should be contrasted with the ordinary boolean conditional \( \text{if } x \text{ then } a \text{ else } b \text{ end} \) 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. \( \text{var}(a) \) denotes the variance of species \( a \) over time, and \( \text{cov}(a, b) \) 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 \( \text{var}(a-b) = \text{var}(a) + \text{var}(b) - 2 \cdot \text{cov}(a, b) \). A useful stochastic flow is \( \text{poisson}(a) \) which, on a time plot, paints Poisson noise centered around the mean of \( a \): it can provide a visual benchmark to interpret \( \text{var}(a) \).

### 2.3 Simulation of chemical kinetics

A chemical reaction network can be simulated by the \( \text{equilibrate} \) instruction over a given time period. Optionally, that instruction can be preceded by \( \text{report} \) instructions to select the flows to plot (by default, all the single species flows are plotted). Reports and equilibrates, which are themselves Actions, can occur anywhere within a computation, and are affected by previous actions.

Figure 2 shows the output of this simple script:

```plaintext
species x1 @ 0.5 M   // prey
species x2 @ 0.5 M   // predator
```
Apart from simulation, one output that can be provided is the equations for the time derivatives of mean and covariance (from the Linear Noise Approximation) of the chemical species:

\[
\begin{align*}
\partial_t x_1 &= x_1 - x_1 \cdot x_2 \\
\partial_t x_2 &= x_1 \cdot x_2 - x_2 \\
\partial_t \text{var}(x_1) &= 2 \cdot \text{var}(x_1) - \text{cov}(x_1, x_2) \cdot x_1 \\
&\quad - \text{cov}(x_2, x_1) \cdot x_1 - 2 \cdot \text{var}(x_1) \cdot x_2 + x_1 + x_1 \cdot x_2 \\
\partial_t \text{cov}(x_1, x_2) &= \text{cov}(x_1, x_2) \cdot x_1 - \text{var}(x_2) \cdot x_1 \\
&\quad - \text{cov}(x_2, x_1) \cdot x_2 + \text{var}(x_1) \cdot x_2 - x_1 \cdot x_2 \\
\partial_t \text{var}(x_2) &= 2 \cdot \text{var}(x_2) \cdot x_1 + \text{cov}(x_1, x_2) \cdot x_2 \\
&\quad + \text{cov}(x_2, x_1) \cdot x_2 - 2 \cdot \text{var}(x_2) + x_1 \cdot x_2 + x_2
\end{align*}
\]
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 and 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*
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, \(r e^{-\frac{a}{RT}}\) 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 operation can be applied to samples.

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

\[
\text{split } B, C, D = A
\]

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 and equal split, but proportional splits can be expressed as \(\text{split } B, C, D = A \text{ by } 0.5, 0.3, 0.2\) (the last proportion can be omitted).

A number of samples can be mixed into one:

\[
\text{mix } E = B, C
\]

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. One operation allows time to pass:

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 reaction 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 $E$.

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:

$$\text{number } n = \text{observe}(f, F)$$

Here $f$ is a flow that provides a way of observing the contents of sample $F$. For example, $\text{observe}(a, F)$ 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 $\text{observe}(\text{time}, E)$ yields the current time in $E$’s timeline (which in the example above is the final simulation time of $E$).

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 actions discussed earlier. For example the execution of $\text{mix } E = B, C$ produces an action $\text{emit}(\text{mix } E = B, C)$ which records all the details of the three samples, before and after the mixing. In the final $\text{List(}Action\text{)}$ 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 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] [17], and can support a number of options for observation of the samples [7].

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. In particular, 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 be 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 virtualization, sample preparation is done off-device, and each sample is represented by a droplet on the device. Mixing, splitting, and disposing of samples is handled by appropriate routing of the droplets: this is automatic, and does not require further annotations in the protocol.

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:

\[
\text{regulate A to } 25 \text{ C}
\]

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:

\[
\text{concentrate B to } 1 \mu\text{L}
\]

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 [8], nuclear magnetic resonance [10], DNA hybridization detection [14], DNA amplification [6], DNA sequencing [17], single molecule detection [11]), 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.

**References**


