Replace this file with prentcsmacro.sty for your meeting,
or with entcsmacro.sty for your meeting. Both can be
found at the ENTCS Macro Home Page.

Chromar, a rule-based language of parameterised objects

Ricardo Honorato-Zimmer [a,1] Andrew J. Millar [b,2]
Gordon D. Plotkin [a,3] Argyris Zardilis [b,4]

[a] *School of Informatics*
University of Edinburgh
Edinburgh, U.K.

[b] *SynthSys and School of Biological Sciences*
University of Edinburgh
Edinburgh, U.K.

Abstract

Modelling in biology becomes necessary when systems are complex but the more complex the systems are the harder the models become to read. The most common ways of writing models are by writing reactions on discrete, typed objects (e.g. molecules of different species), or writing rate equations for the populations of such species. One problem (1) with those approaches is that the number of species and reactions is often so large that the model cannot be realistically enumerated. Another problem (2) is that the number of species and reactions is fixed, whereas biology often grows new compartments which means new reactions and species. Here we develop an extension to the representation of reactions where the objects carry variables that are defined by their type (for example objects of type Leaf all have a Mass variable). The dynamics are defined by rules about types, which means they work for all objects of that type. This compact representation solves problem 1. If we think of the object variables as the analogue of reaction/rate equation species, creating a new object of some type means we are also creating new species (solving problem 2). We also developed an embedding of Chromar in the programming language Haskell and showed its applicability to two examples. Having a more compact representation can help make models a tool for knowledge representation and exchange instead of just a simulation input. Embedding Chromar in a general purpose programming language lifts some of the constraints of modelling languages while still maintaining the naturalness of a domain-specific language.

Keywords: rule-based modelling, stochastic, representation, systems biology

1 Introduction

The notation we use to describe parts of the natural or artificial world can act as a tool for thinking about it. The characteristics that a notation for a specific domain should have in order to be a good tool for thought have been succinctly listed by Kenneth Iverson: ease of expression of common constructs in the domain, suggestivity, ability to subordinate detail, economy, and amenability to formal proofs [10]. A lot of excellent notations have been invented for biological models, some more general and some more domain specific. Of course a single notation cannot be used for everything and some specific models are hard to write in any existing notation in a way that satisfies the above criteria and makes it easy for people and computers to understand. To illustrate the problem consider the model of a growing array of cells, each having a concentration of some substance X diffuse among them, produced, and destroyed. The most common way of

[1] Email: r.honorato@sms.ed.ac.uk
[2] Email: andrew.millar@ed.ac.uk
[3] Email: gdp@inf.ed.ac.uk
[4] Email: A.Zardilis@sms.ed.ac.uk

©2016 Published by Elsevier Science B. V.

writing such systems is by writing reactions on the types (species) of molecules involved or writing the corresponding rate equations for the populations of the species. In our case we could write the following reactions for the molecules of X in each cell, where X_1 is an X molecule in the first cell, X_2 is an X molecule in the second cell and so on:

$$\begin{gathered} X_1 \xrightarrow{d} X_2 \\ X_2 \xrightarrow{d} X_1 \\ X_2 \xrightarrow{d} X_3 \\ \vdots \\ \varnothing \xrightarrow{\alpha} X_1 \\ \vdots \\ X_1 \xrightarrow{\beta} \varnothing \end{gathered}$$

There are two problems with the above description. The first problem is that it is not very compact and it grows with the number of cells since we have to write the diffusion reaction for every pair of cells in both directions and production/destruction reactions for every cell. The second problem is that it is impossible to describe the creation of new cells because we would need to create a new species of X for the new cell and new reactions for it, but there is no operator in this notation that allows that.

Ideally, there would be some notation that allows us to formally represent the above system in a way that satisfies our intuition, for example writing a generic diffusion reaction $X_i \to X_{i+1}$, a generic production $\varnothing \to X_i$ (similarly for destruction), and some way of generating new species. Our principal contribution is a notation that allows us to write systems like the above in a natural way, thereby solving the two main problems we noted: compactness of representation for larger systems and dynamic state-space. Specifically, our main contributions are:

- We define a rule-based notation with stochastic semantics. The main entities in the notation are objects with attributes that are defined at the type level, so that every object of that type has these attributes. For the above diffusion model we could have for example a type $X(n : \text{Int})$ with attribute n for the position of the X molecule in the array. Objects are instantiations of this type with concrete values for the attribute like $X(n = 1)$ for a molecule in the first cell, $X(n = 2)$ for a molecule in the second cell and so on. The rules describe how objects are added or removed (Section 3) at the type level, so that a rule applies for all objects of that type ($X(n = 1)$, $X(n = 2)$ etc.). This leads to a more compact representation of the model because each rule corresponds to multiple concrete reactions. If we make our species attributes of some type, for example in our case we could have $\text{Cell}(\text{pos} : \text{Int}, \text{x} : \text{Int})$, when we create new Cell objects we are also creating new species that will automatically be picked up by the Cell rules. This solves the second problem we noted. Our language is like a rule-based of Coloured Petri Nets. This rule-based textual representation becomes very important for the readability of larger models and our embedding in Haskell gives extra expressive power that is also crucial in practice (see Section 6 for full discussion).
- We describe an algorithm for the stochastic simulation of models written in this notation that acts directly on the attributed objects and the rules (Section 3)
- We have implemented the language, both the model definition and simulation, as an embedded Domain Specific Language (DSL) inside Haskell, a functional programming language (Section 4). The embedding means that we can use any valid Haskell expression where expressions are expected, for example in the rate expressions and in the right-hand sides of rules. From our experience this is very useful in model building, especially for more complex models.

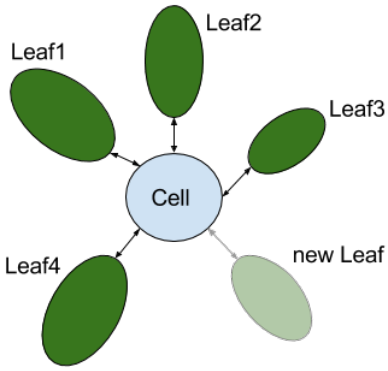


Fig. 1. Our simple plant development model. All the interactions, that in this case are transfer of carbon, happen between the central Cell object that represents the molecular state of the entire plant and the Leaf objects, which act as sinks of carbon. The carbon that goes to the leaves is either used for growth, in which case it is transformed into new material (increase of mass), or to maintain the already existing Leaf by fuelling its life sustaining processes. New leaves are also created creating new sinks and increased competition for carbon among the leaves but also increased production of carbon by providing new green area for photosynthesis.

- We show, using examples, the expressivity of our abstract notation but also the advantages of having an embedding in a general purpose programming language (Sections 2 and 5).

2 An example: Plant growth

We will now give an overview of our notation through an example from plant development. We will consider a very abstract view of plant development that has enough details to demonstrate the main features of our notation. Our model here is inspired by the Framework Model (FM) of Chew et al. [3], a modular whole-plant model that connects traditional Plant biology representations of molecular processes and representations of organ and whole-plant development processes. The above-ground part of an Arabidopsis plant before flowering has a simple architecture with a collection of leaves arranged in a circle. Each leaf photosynthesises, creating the main currency, carbon; uses some of it for maintenance, some of it for growth; and transfers anything left to the other leaves. In the Arabidopsis rosette (collection of leaves) there is no preference in the transfer and we have an all-to-all communication. Similarly to the FM, we will make all the molecular processes happen instead at a central plant 'cell' which allows us to keep the leaves as carbon sinks and track their growth, while avoiding the per-leaf molecular processes and their communication (see Figure 1).

We will think about all the processes that affect growth in the following way: we think of assimilation of carbon per leaf as increasing the carbon concentration of the central Cell depending on the photosynthesis level of a leaf (which will depend on its size); we think of maintenance respiration as the central Cell giving some carbon to a leaf; and we think of growth respiration as the central Cell giving some carbon to a leaf and the leaf mass increasing. We will also have creation of new leaves. There are interesting dynamics here such as the interaction between growth and assimilation: the more we grow, the more the leaves can photosynthesise, and the more carbon can go to the central Cell.

Since objects are the main entities in our language we can start thinking about what types of objects should we have to model the above system. We will need:

- A Leaf type with fields for the mass and index of appearance as a proxy to a Leaf's age: $\mathrm{Leaf}(\mathrm{age} : \mathrm{Int}, \mathrm{mass} : \mathrm{Real})$
- A Cell type that represents our main plant 'cell' with a field, carbon, to keep the current carbon level: $\mathrm{Cell}(\mathrm{carbon} : \mathrm{Real})$. There will only be one object of this type at any one point.
- A Ros type that represents the entire Rosette with a field, nl, to keep the current number of leaves: $\mathrm{Ros}(\mathrm{nl} : \mathrm{Int})$. There will also only be one object of this type at any point.

For the assimilation of carbon from one particular leaf we need to increase the carbon concentration of the central Cell. The bigger the leaf the 'faster' it contributes to the production of carbon:

$$\text{Leaf}(\text{mass} = m),\ \text{Cell}(\text{carbon} = c) \xrightarrow{f(m)} \text{Leaf}(\text{mass} = m),\ \text{Cell}(\text{carbon} = c + 1)$$

We can read this as saying that for any pair of Leaf, Cell the Leaf remains the same and the Cell increases its carbon content by one. Note that we assign the values of the fields for *any* Leaf, Cell pair to variables (m and c) so that we can refer to them in the right-hand side of the rule and the rate expression. If we were to write this in a traditional reaction notation we would have to write a reaction for every Leaf which leads to the compactness problem we have noted earlier. With the implicit 'for-all' here we can also pick up new leaves when they are created. For maintenance, we have the central Cell object giving some carbon to a Leaf object, with the amount of carbon needed for maintenance depending on the size of the Leaf:

$$\text{Leaf}(\text{mass} = m),\ \text{Cell}(\text{carbon} = c) \xrightarrow{g(m)} \text{Leaf}(\text{mass} = m),\ \text{Cell}(\text{carbon} = c - 1)\ [c \geq 1]$$

Another way to see these rules, which is actually how their meaning is defined later (see Section 3), is to think that any pair of Leaf, Cell objects can be removed and replaced by a Leaf object with the same mass as the one we removed and a Cell with a carbon decreased by one compared to the Cell object we removed. Since we are defining the replacement objects (right-hand side) in terms of the replaced objects (left-hand side) we need to assign their field values to some variables, here m and c, so we can refer to them again. The growth of a Leaf depends on its mass, its age (there is some limit on how much a leaf can grow so older leaves stop growing at some point), and the amount of carbon available:

$$\text{Leaf}(\text{mass} = m, age = i), \text{Cell}(\text{carbon} = c) \xrightarrow{h(i,m,c)} \text{Leaf}(\text{mass} = m{+}1), \text{Cell}(\text{carbon} = c{-}1)\ [c \geq 1]$$

Note that we use the condition $c \geq 1$ to make sure that the carbon levels do not go negative. Finally, for the creation of new leaves we have:

$$\text{Ros}(\text{nl} = n) \xrightarrow{k} \text{Ros}(\text{nl} = n + 1),\ \text{Leaf}(\text{age} = n + 1, \text{mass} = 0.0)$$

3 Chromar

In the previous section we got an idea of what the language looks like. Here we will make a more careful definition of the abstract syntax of the language and its semantics.

3.1 Syntax

Objects or agents are the main entities in the language. Each object is an instantiation of a type that provides the general structure of all objects of that type. Agent types have a name and a number of named fields for their attributes. Their syntax is:

```
agentType := AgentName ( fieldDecl_1, ..., fieldDecl_k )
fieldDecl := fieldName : type
```

An example of an agent type is the Leaf agent type that we have seen in the previous section that has mass and age fields: $\text{Leaf}(\text{mass} : \text{Real}, \text{age} : \text{Int})$. Any specific Leaf object is an instantiation of this agent type, for example: $\text{Leaf}(\text{mass} = 3.5, \text{age} = 3)$. The `types` for the fields are not fixed here and the language is parametric in them. In the next section, when we define the Haskell embedding, we will fix these to be the Haskell types.

The state of the system is a multiset of objects of the defined types. For example for the types of objects we had in the example in the previous section a possible valid state of the system is:

$$\{|\,\mathrm{Leaf}(\mathrm{mass} = 2.3, \mathrm{age} = 3), \mathrm{Leaf}(\mathrm{mass} = 3.1, \mathrm{age} = 2),\\ \mathrm{Leaf}(\mathrm{mass} = 3.5, \mathrm{age} = 1), \mathrm{Ros}(n = 3), \mathrm{Cell}(\mathrm{carbon} = 5.6)\,|\}$$

We write multisets using $\{|\ldots|\}$ brackets.

Rules have the following syntax:

```
rule       := lhs --> rhs at rate ([cond])
lhs        := agentPat_1, agentPat_2, ..., agentPat_n
agentPat   := AgentName (fieldPat_1, ..., fieldPat_n)
fieldPat   := FieldName = Var

rhs        := agentR_1, ..., agentR_n
agentR     := AgentName (field_1, ..., field_n)
field      := FieldName = expr

rate       := expr
cond       := expr
```

Rules have a left-hand side, which is matched against the state of the system. Any match can then be replaced by the right-hand side. The left-hand side is really simple: it can only select objects based on their type and bind the values for their fields to some variables which can then be used in the expressions for the values of the objects appearing on the right-hand side. The variables can also be used in the rate and condition expressions. We impose some constraints on the use of the variables: variables can only appear once in the left hand side of rules and the number of patterns, `fieldPat`s, for some `AgentName` should match the fields in the type declaration of `AgentName`. Variables appearing in the expressions of `rhs`, `rate`, and `cond` must appear on the `lhs` of the corresponding rule. The `cond` is an expression that evaluates to a Boolean value and determines the applicability of the rule. Again, we deliberately do not fix the `expr`s and `var`s to any specific sets and we could think of the language as being parametric on these. In the Haskell embedding we fix these to be Haskell variables and expressions.

For the rest of the text we will assume that we have accessor functions to the various parts of some rule with names coming from the syntax above. For example for some rule r, $lhs(r)$ is the accessor function for the left-hand side of the rule and so on.

3.2 Rule application

Rules can only be applied given a specific match of the rule to the state of the system. Matches occur between left-hand sides of rules and the concrete objects in the multiset that represent the state of the system. First we define a bind-map as an assignment of values to the variables in the `agentPat`s in the left-hand side of rules that we write like this: $[var_1/val_1, \ldots, var_n/val_n]$. A bind map can be used to produce a concrete realisation of the rule by substituting all the occurrences of the vars with the associated vals in the map. We will write this substitution as $[var_1/val_1, \ldots, var_n, val_n].r$ for some rule r. A bind-map, σ, for a particular rule is a *match* to the state of the system if $lhs(\sigma.r)$ is equal to some part of the state. The `agentPat`s in the left-hand side of rules become concrete Agents and two Agents are equal if they have the same name and they have the same values for their fields.

To illustrate the matching, consider the following system with types of agents $A(x : Int)$ and $B(y : Int)$, a multiset of objects of these types $\{|A(x = 5), A(x = 5), A(x = 2), B(y = 3), B(y = 4), B(y = 3)|\}$ and a rule, $A(x = x), B(y = y) \xrightarrow{f(x,y)} A(x = x - 1), B(y = y + 1)[g(x, y)]$ with $f : Int \to Int \to Real$ and $g : Int \to Int \to Bool$. The bind-map $[x/5, y/3]$ is a valid match

between the rule left-hand side and the state (multiset) since $A(x = 5), B(y = 3)$ exists in our state. Other bind-maps that are matches are for example: $[x/5, y/4]$, $[x/5, y/3]$, $[x = 2, y/3]$ and so on. The bind-map $[x/5, y/3]$ yields a concrete reaction from our rule by substitution:

$$[x/5, y/3].\left(A(x = x), B(y = y) \xrightarrow{f(x,y)} A(x = x - 1), B(y = y + 1)\ [g(x, y)]\right) \rightsquigarrow$$
$$A(x = 5), B(y = 3) \xrightarrow{f(5,3)} A(x = 4), B(y = 4)\ [g(5, 3)]$$

For some rule r and a match σ, applying the rule to a multiset M gives a new multiset M' given by:

$$M' = M \uplus rhs(\sigma.r) \setminus lhs(\sigma.r)$$

where $\uplus$ and $\setminus$ are multiset addition and difference respectively [23]. A rule can only be applied if $cond(\sigma.r)$ evaluates to True for the particular match σ.

3.3 Stochastic semantics

Since any rule and a match give a concrete reaction, any Chromar model can be expanded into an equivalent simple reaction system by considering all possible matches, that is all possible instantiations of the defined types. The stochastic semantics of Chromar is then the same as the stochastic semantics of the equivalent simple reaction system. Specifically, the stochastic process is a Continuous Time Markov Chain (CTMC) and the state-space consists of all possible multisets over concrete realisations of our types. The expanded system will, in most cases, give infinite reactions unless we constrain the types of the fields in our object types. However for a given state only finitely many of these reactions will apply, so we can still use the normal Stochastic Simulation Algorithm (SSA) to get sample paths from the CTMC.

Specifically our algorithm is the usual SSA, but with an extra step that dynamically creates the reactions based on the current state of the system:

(i) Find all reactions for every rule and every match:

$$R = \{\sigma.r | r \in Rules, \sigma \in \Phi(r),\ cond(\sigma.r) = True\}$$

where $\Phi(r)$ consists of all the matches of rule r in the current state.

(ii) Calculate the total rate $rate_T = \sum_{r \in R} rate(r)$.

(iii) Pick the waiting time for the next reaction event from the exponential distribution with cumulative distribution function $F(t) = 1 - e^{-rate_T t}$.

(iv) Pick exactly one of the reactions, choosing reaction i with probability $\frac{rate(r_i)}{rate_T}$.

(v) If reaction i is picked then update the state: $M' = M \uplus rhs(r_i) \setminus lhs(r_i)$ and iterate.

4 Haskell embedding

In any implementation of the language, eventually all entities have to become data structures in some programming language in order to set the model in motion on a computer. There are two extremes to this. At one end, we could make the model definition exactly like the abstract one presented in the previous section and then translate it to the programming language constructs. At the other end, the model definition could happen directly as constructs in some programming language. There are advantages and disadvantages to both but here we choose something in the middle: the model definition stays inside the programming language but we tweak the language a bit so that it understands the convenient rule syntax.

We chose the functional programming language Haskell for the implementation and for embedding our language. For an implementation we need to choose how to define the objects and

how to define the rules. The objects and their types are exactly record types in Haskell and our matching has the same semantics as Haskell's pattern matching so the definition of the types is easy. This is how the types of the objects in the Plant growth example are defined:

```
data Object = Leaf { mass    :: Double,
                     age     :: Int }
            | Cell { carbon  :: Double }
            | Ros  { nleaves :: Int }
```

The keyword `data` defines a new datatype and here we are defining a union type with three possible constructors separated by `|`.

The definition of rules is a bit less straightforward but we can think of rules as functions of the following type `Multiset a -> [Reaction a]` where `a` is a type variable that can stand for any user-defined type of objects. Each rule is, as we have seen, a generator of concrete reactions. We cannot expect the user to write the function doing the matching and creating the reactions (even though it is not too hard to write in Haskell since we can take advantage of Haskell's pattern matching) so we have made an easier definition of rules using Quasi-quotes. Quasi-quotes in Haskell allow for special syntax inside `[| ... |]` quotes as long as you provide a quoter, a function that takes the string inside the quotes symbols and produces Haskell abstract syntax that gets injected in the place of the quotes during compile time. Since all the rule function definitions that we want have a similar structure, it is easy to write such a quoter function that takes a rule written in the abstract syntax and creates a function of the correct type. This is how the growth rule from the Plant system (Section 2) is written:

```
growth = [rule| Leaf{mass=m, age=i}, Cell{carbon=c} -->
                Leaf{mass=m+1, age=i}, Cell{carbon=c-1} @f(m, i) [c-1>0] |]
```

This looks very close to how we have been writing rules in our abstract syntax, but with some minor syntactic differences such as the placement of the rate expression at the end of the rule preceded by the `@` symbol. Crucially, being inside a programming language means that we can use any valid Haskell expression in the places where expressions are expected, i.e. in the values of fields in the right-hand side of rules, rates, and conditions.

Our `simulate` function takes a list of rules, an initial state as a multiset, and the number of steps:

```
rules     = [growth, assimilation, leafCreation]
initState = ms [Leaf{age=1, mass=1.0}, Leaf{age=2, mass=1.5},
                Ros{nleaves=2}, Cell{carbon=4.5}]

simulate  rules initState 100
```

where `ms` is a function `[a] -> Multiset a` that creates a multiset from a List. The `simulate` function implements the simulation algorithm defined in the previous section.

4.1 Observables

The simulation algorithm is a way of getting sample paths from the state-space of the system and since our states are multisets over objects of the defined types, the path is just a time-indexed sequence of multisets. However the full state of the system is rarely what we want to know or at least it is rarely the only thing we want to know. For example given a multiset representing the state of our virtual plant from the previous example we might want to compute the mass of the entire plant or we might only want the carbon levels. Thinking of suitable query primitives on top of multisets the following two operations seem natural:

```
select    :: (a -> Bool) -> Multiset a -> Multiset a
aggregate :: (a->b->b) -> b -> Multiset a -> b
```

Fig. 2. Diffusion rule. Any molecule inside a cell can move to the cell to its right or left with equal probability.

These are just the select and aggregate statements in database query languages, where databases are often viewed as multisets [2] (more theoretically in [14]). In fact our object types with named fields are similar to database records. These query statements compose nicely with normal function composition which is primitive in Haskell so for example we can get the mass of the first three leaves by:

```
mass3 = aggregate sumM 0.0 . select isFirstThree . select isLeaf
```

with the functions `isFirstThree, isLeaf :: Object -> Bool` and the evident implementation. These are the most generic query constructs, but starting from them we can specialise to the most common use-cases with default filter and accumulation functions. For example, to `select` elements from the multiset we have special `select`s for objects with a specific type, or objects that have certain values for a field. And to `aggregate` we have special `aggregate` functions like sum, min/max, average, count.

5 Another example

We give another example here of a growing domain of cells with some substance diffusing between them. This is a more complicated version of the example given in the introduction where, instead of growing the domain of cells at one end, any cells at any position can divide. Assuming a one dimensional array of cells, each having a concentration of substance x that diffuses between them, we introduce the following types:

- $\mathrm{Cell}(\mathrm{pos} : \mathrm{Pos}, \mathrm{x} : \mathrm{Int})$
- $\mathrm{T}(\mathrm{ncells} : \mathrm{Int})$

A Cell object has a pos field that keeps positional information. In this case, to fully determine the cell's position in the array we need its identifier and the identifier of its neighbour. That way we can define the neighbour relation using equality between the identifier of one cell and the identifier in the neighbour field of another cell. A Cell also keeps track of the number of x molecules. We also have a T object, standing for a tissue with a field ncells, to keep track of the number of cells in the array. This is needed to give fresh identifiers to the cells created by division.

Going into the dynamics of the system, diffusion is the transferring of one molecule from one cell to the other and we assume it happens with equal probability to the left and right neighbours of the cell (see Figure 2). This gives the following rule:

$$\begin{aligned}&\mathrm{Cell}(\mathrm{pos} = p, x = x),\ \mathrm{Cell}(\mathrm{pos} = p', x = x') \xrightarrow{x}\\ &\mathrm{Cell}(\mathrm{pos} = p, x = x - 1),\ \mathrm{Cell}(\mathrm{pos} = p', x = x' + 1)\ [\mathrm{nextTo}(p) = \mathrm{id}(p')\ \&\ x - 1 > 0]\end{aligned}$$

where $\mathrm{id} : \mathrm{Pos} \to \mathrm{Int}$ and $\mathrm{nextTo} : \mathrm{Pos} \to \mathrm{Int}$ are accessor functions to the identifier of the Cell and its right neighbour respectively. Here we use a condition to limit our matches since the diffusion rule is not applicable to all pairs of Cell objects that are picked up by the left-hand side. For growth, we create a new cell on the right of the dividing cell and split the x molecules as

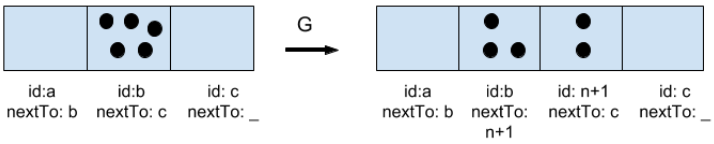
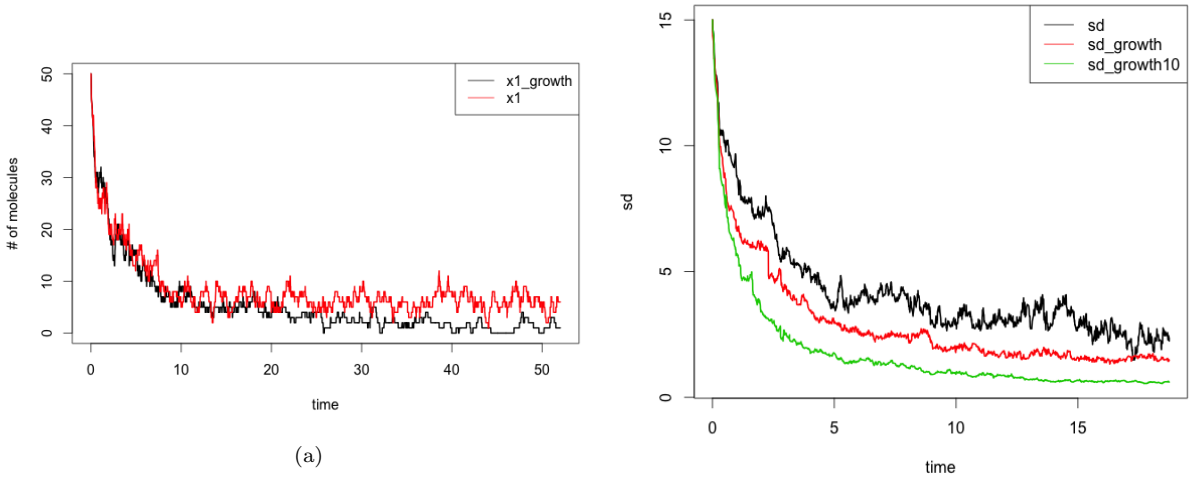


Fig. 3. Growth rule. Here cell b divides creating cell $n+1$ (assuming we had n cells before the division), cell b moves to the left of the new cell. The 5 X molecules of cell b get divided between itself and the newly created cell.



(a) (b)

Fig. 4. On the left the number of Xs in the first cell with and without growth. We start both processes at 10 cells and in the growth case we activate the growth rule with $G = 0.1$. In the beginning when the number of cells is close the trajectories are close but as the number of cells increases in the growth process, the number of molecules in the Cell on average is lower than the process without the growth. On the right, we plot the standard deviation of the cell contents (number of cells) in 3 different cases, diffusion only process, diffusion+growth at rate $G = 1$, and diffusion+growth at rate $G = 10$ all starting again with 10 cells. We can see that the faster the growth the faster the molecules get spread.

evenly as possible between the two cells:

$$\begin{aligned}
&\mathrm{T}(\mathrm{ncells} = n),\ \mathrm{Cell}(\mathrm{pos} = p, \mathrm{x} = x) \xrightarrow{G/n} \\
&\mathrm{T}(\mathrm{ncells} = n+1),\ \mathrm{Cell}(\mathrm{pos} = \mathrm{Pos}(\mathrm{id}(p), n+1), \mathrm{x} = \mathrm{ceil}(x/2)), \\
&\mathrm{Cell}(\mathrm{pos} = \mathrm{Pos}(n+1, \mathrm{nextTo}(p)), \mathrm{x} = \mathrm{floor}(x/2))
\end{aligned}$$

The dividing cells gets pushed to the left keeping its id and changing its neighbour identifier to the identifier of the new cell. The new cell gets a fresh identifier from our counter in T and a neighbour identifier the old neighbour of its mother cell (Figure 3). We assume there is a general growth rate G for the entire array so the rate for each cell is scaled to G/n.

It is interesting in this system to compare the behaviour with and without growth. Since we are not creating new molecules, we expect diffusion to spread the molecules among the cells. With growth we expect fewer molecules per cell since the same number of molecules is spread over a bigger number of cells - see Figure 4a for the number of X molecules in cell 1 in one realisation of the process with and without growth. Since diffusion spreads the molecules among the cells we expect the variability in the cell contents to go down with time. It is also interesting to see how fast variability is reduced in the diffusion only and diffusion+growth processes. In Figure 4b we plot how the standard deviation of the cell contents (number of molecules) is reduced over time in three different cases - diffusion only, diffusion + growth with rate $G = 1$, and diffusion + growth with rate $G = 10$. While we cannot compare the absolute numbers since we have different number of cells in each case, we can see that growth amplifies the effects of the diffusion spreading the molecules and reduces variability in the cell contents faster.

6 Relevant work

The idea of extending simple objects with fields to represent some of their attributes has been used before for example in Coloured Petri Nets [11] and in a more rule-based setting in CSMMR [19]. Our notation is inspired by both of these and we can think of our notation as a rule-based version of stochastic coloured petri nets where the richer types are first class and not merely a means of translation to a non-coloured version. We can also think of our notation as a simpler version of CSMMR with only the colours left. Our embedding in a programming language for increase of expressive power is also new, and fits with the availability of rich types. The use of the database inspired operations for the observables is also new and in practise we have found it very useful in model building. The declarative nature of our multiset query primitives makes the definition of the observables very intuitive. Similar database-inspired query operations on top of collections are used in LINQ [1] although the collections are usually taken to be lists not multisets. Buneman's comprehension syntax [2], again a collection query language similar to practical database query languages, considers other types of collections including multisets.

Colours can be used to encode the binding of species as in the example in the previous section. However, whenever we use them to encode binding we would probably be better off using a language that represents binding directly like Kappa [5]. Yet in the dividing cell and diffusion model of the previous section we use colours in other ways that can't be easily represented as binding. In particular, the division of the contents of a cell would be hard to express in Kappa. Also, counting how many X molecules we find at each position would be have to be done by manual inspection of the state.

In the next two sections we will focus on the comparison to Coloured Petri Nets since this is the most directly comparable system and on a comparison to a system coming from (primarily) a different domain (ecology) and a different paradigm – deterministic instead of stochastic.

6.1 Coloured Petri Nets

The closest formalism to our notation is Coloured Petri Nets. Petri Nets are a graphical network-based formalism often used to represent reactions. There are two type of entities in the nets: places and transitions. Places carry a population of tokens and transitions are a way of moving tokens from one place to the other. The state of the system is just the number of tokens at each place. Coloured Petri Nets (CPN) are an extension to Petri Nets that allows distinctions between tokens (colouring of tokens) by allowing them to have an associated data value adhering to the type (colourset) of their place [11]. For example, if a place has type $\mathrm{Leaf}(\mathrm{mass} : \mathrm{Real}, \mathrm{age} : \mathrm{Int})$, a token in that place might have value $\mathrm{Leaf}(\mathrm{mass} = 3.0, \mathrm{age} = 2)$. Our growth transition from the plant growth example in Section 2 would give the network in Figure 5. We have two coloursets: Leaf which is a product type over age and mass, and carbon. Our initial state has two tokens in the Leaf place, one with age 1 and mass 10 and another with age 2 and mass 5, and we have one token of carbon with value 10. A transition removes tokens from its pre-places (places with arcs going from them to the transition) and moves tokens to its post-places (places with arcs going into them from the transition). In this case pre and post places are the same so the effect of the transition is as in our system: to remove one Leaf and replace it with a Leaf with updated mass and remove the carbon token and replace it with a carbon token with an updated value.

The correspondence to our system is straight-forward, coloursets are our object types (records with named fields), tokens are our objects, and transitions are our rules. CPN transitions also have predicates that are the same as our conditions. One difference is that CPNs also allow union types instead of just product types as in our language. A stochastic version of this CPN formulation has also been used for biological modelling before for example for describing planar cell polarity in Drosophila wings [7], and other use-cases [22,8]. In these examples where the stochastic version was used its semantics are just given as a translation to the corresponding simple Petri Net.

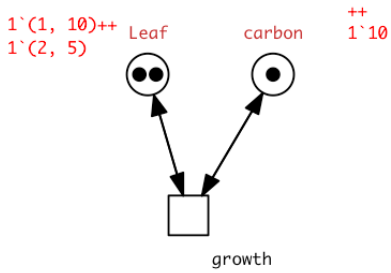


Fig. 5. The growth rule as a Coloured Petri Net transition.

The problem with that is that in a lot of cases the unfolded simple Petri Net has an infinite number of reactions. This means that in order to be able to do the unfolding at all the types have to be bounded to some finite set which further means that real values are not allowed. This is also reflected in the Coloured Petri Net tools implementation where one can define a Stochastic Coloured Petri Net but the definition is unfolded before it is run [9]. Here we have defined the semantics on the coloured stochastic version directly and have a simulation algorithm where reactions are generated as needed dynamically, instead of being created statically in the beginning. This allows us to have unbounded types for our fields, including real numbers. We can think of our language then as a stochastic rule-based (and therefore textual) version of Coloured Petri Nets. Our embedding in a general purpose programming language is also new, as most Petri Net tools have a graphical interface for defining the models although there is a hybrid approach where you can mix the graphical definition with programming language constructs (ML language) [12]. While graphical notations are intuitive for smaller models, we have found that for larger models they become hard to read whereas text-based approaches like our language produce much more readable representations. The embedding in Haskell also further helps to manage complexity in larger models since we can use Haskell's constructs for modularisation (from functions to modules). The ability to use any Haskell expression is also crucial since we can reuse existing libraries, have access to the full range of language primitives, and we are able to write any number of functions for the expressions (for rates, rule right-hand sides etc.) that helps hide some of the complexity from the rules.

6.2 Simile

Simile is another graphical language that has similarities to our approach [18]. Simile is used mainly in the domains of Ecology and Agricultural Sciences but has also been used in Systems Biology before (for the whole-plant model [3] that was the inspiration to our first example in Section 2) to exactly solve the kind of problems we noted in the Introduction. In Simile there are two levels of definition of a model, at the first level we have continuous variables with rate equations and at the second level we have discrete objects with discrete dynamics – adding/removing. The objects are grouped based on their types and their behaviour is given at the population level. Following from our plant growth example the growth of the leaves in Simile would be written as shown in Figure 6. We have a population of Leaf objects and a single Biochem object representing what we called Cell in our rules. Each Leaf in the population has a mass that grows as a continuous variable. In order to define the use of carbon for growth from the carbon variable in the Biochem object we have to do at the population level by summing the contribution of each Leaf. The population of Leaf objects also grows (see creation box).

The dynamics of the two types of entities, continuous variables and objects, are not integrated like our language. In Chromar creating new objects or updating the values at the fields of objects works in the same way by removing and adding objects. The main difference is though that an execution of a Simile model ends up as a system of ODEs whereas in Chromar we are in the stochastic world. Again the graphical notation of Simile becomes, in our experience, problematic for larger than a few variables models.

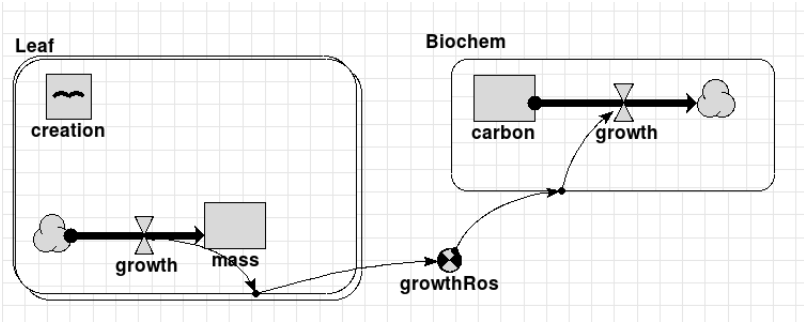


Fig. 6. The growth rule as a Simile model.

7 Discussion and Conclusions

We have defined an extension to the representation of reactions where we extend the simple, typed objects to objects employing rich types, namely records with named typed fields. Writing rules on these richer types yields a more compact representation than one would get by writing reactions on the simpler types in the traditional reactions setting. Moreover it sometimes helps writing systems that are impossible to write otherwise for example systems with dynamic number of species by allowing us to store variables as object attributes which means that creating new objects creates new species.

We have seen from our representative examples that the more compact model representation we get can be easier to read and more intuitive to write. In the end we write models to express our understanding of the world, but the more complex the models get, the further away their representation gets from our mental model of the world. The models then become just simulation inputs and the mental model becomes diagrams/pictures to capture the intuition of the process. Our work here is a step in the direction of closing the gap between mental models and formal executable models, at least for a class of models in Systems Biology.

On the implementation side, our embedding in Haskell lifts some of the constraints of modelling languages and we think gets the best of both worlds: it naturally and succinctly captures some elements in our domain of interest but at the same time when greater expressive power is needed we can turn to the programming language. This increase in expressive power might come at the expense of the ability to do general analysis of models since we cannot say much about what is happening in the Haskell `exprs` inside the rules. There seems to be a trend though in the direction of mixing domain-specificity and general purpose programming languages, for example Pedersen et al. [21] allow embedded F# scripts inside LBS-κ, and in PySB [15] Kappa models are defined inside Python. Embedding a domain-specific inside a programming language like we did is in some cases better than doing the opposite – embedding a programming language inside a domain-specific language – because we have less constraints on our definitions and more generally full access to the language for structuring our model definitions.

In terms of the simulation our implementation is simple and basically follows the steps we presented in Section 3. The idea about dynamic creation of reactions also appears in [20]. One area of improvement is the reaction generation step where we currently generate all the active reactions at every step. In practise though we do not need to completely regenerate them at every step since only a subset of them changes between steps. The performance gains will depend on the efficiency of calculating the change in the reactions set after a change in the state. Similar techniques where the matches are generated only once at the beginning of the simulation and then updated according to state changes are used in Kappa [6].

There are various ways our notation could be extended. Our rule left-hand sides are really simple and can only select based on type which means the only relations we can encode directly are products of types (we can think of types as sets containing all the objects of that type). For example in our array of cells example writing $\mathrm{Cell}(\ldots), \mathrm{Cell}(\ldots)$ on the left-hand side mean the

rule is applicable to any pair in the relation Cell $\times$ Cell. A lot of times this is okay, for example in our Plant system (from Section 2) *all* Leaf objects interact with *all* Cell (only one in this case) objects so writing the left-hand at the type level is okay because the rules are then applicable to exactly the pairs of objects we want, $\{(\text{Leaf}_1, \text{Cell}), (\text{Leaf}_2, \text{Cell}), \dots\}$. In other cases though the relation we want is some subset of the product of the types. For example in our array of cells example the diffusion rule is not applicable to all pairs of Cells so writing $\text{Cell}(\dots), \text{Cell}(\dots)$ on the left-hand side gives us more pairs than we want. In those cases we can restrict the applicability of the rules by our conditions and the only way to do that is by somehow in the fields of the types encode the relation information and use that in the condition.

In the array of cells case, we encoded the relation through identifiers and the next-to relation pairs were stored inside our objects. This works nicely in this simple case but what if we had more complex relations or had more than one relation? For example in a plant with a more complex architecture and some interaction between the leaves we will need to know which leaf is connected to which and if we further had a 'nested-in' relation between leaves and let's say cells then we would have a hard time keeping track of the relations. Ideally we would at least have the language keep track of some of these things for us and a special notation for the most common types of relations – for example in Biology the 'connects-to' and 'nested-in' relations seem natural. We could then write rule left-hand sides that say 'this rule is applicable to any two leaves that are connected' or 'this rules is applicable to any two cells inside the same leaf'. The 'connects-to' relation is the main driver of the models in Kappa for example giving a graph-like state to the system [4]. A version of Kappa with richer types like the ones we have shown here would be very powerful. Another system where both connection and nesting has been defined is Bigraph and Stochastic bigraphs in particular are applicable in the biological setting [17,13].

Our observables could be developed further and made into first-class entities in the language, for example by making them fields of types. Take for example the number of leaves in our Plant example that is a field of the `Ros` type. The value of the field at any given point in time is a function of some other part of the system, namely the leaves part of the state. This means that we have two representations of the same process at the population level and at the individual level inside the same model, which is problematic in some cases because we need to keep them consistent with each other. Ideally we would like this correspondence between the two representations to be made explicit so for example when declaring the `Ros` type we will able to say that its field `nleaves` is an observable and define it using our query primitives – `nleaves = count . select isLeaf`. Having the mapping there explicitly is good for the readability of the model and more practically means that the propagation of information to keep the two representations consistent can be automated. These mappings, defined as observables, would work particularly well with an extension for a native representation of levels (the 'nested-in' relation we noted earlier) because in that case we would definitely have representations of the same process at different levels of abstraction, in which case the idea of the fields of objects at a higher level being observables of objects at a lower-level would be really intuitive and powerful. The idea of multiple levels has already explored in the rule-based setting, for example in [16] and [19].

Finally, in a lot of use-cases the environment in which the rules are operating is not constant. This is especially important in Plant biology since plants are very adaptive to environmental inputs, for example our system in Section 2 is very detached from reality since the assimilation rule is always active whereas in reality it should switch on only during the day when the plant photosynthesises. Incorporating a changing environment means somehow incorporating time inside our language. Such an extension would be really powerful and very practically useful.

References

[1] Budiu, M., J. Galenson and G. D. Plotkin, *The compiler forest*, in: *European Symposium on Programming*, Springer, 2013, pp. 21–40.

[2] Buneman, P., L. Libkin, D. Suciu, V. Tannen and L. Wong, *Comprehension syntax*, ACM Sigmod Record **23** (1994), pp. 87–96.

[3] Chew, Y. H., B. Wenden, A. Flis, V. Mengin, J. Taylor, C. L. Davey, C. Tindal, H. Thomas, H. J. Ougham, P. de Reffye et al., *Multiscale digital arabidopsis predicts individual organ and whole-organism growth*, Proceedings of the National Academy of Sciences **111** (2014), pp. E4127–E4136.

[4] Danos, V., J. Feret, W. Fontana, R. Harmer and J. Krivine, *Rule-based modelling of cellular signalling*, in: *International Conference on Concurrency Theory*, Springer, 2007, pp. 17–41.

[5] Danos, V., J. Feret, W. Fontana, R. Harmer and J. Krivine, *Rule-based modelling, symmetries, refinements*, in: *Formal methods in systems biology*, Springer, 2008 pp. 103–122.

[6] Danos, V., J. Feret, W. Fontana and J. Krivine, *Scalable simulation of cellular signaling networks*, in: *Asian Symposium on Programming Languages and Systems*, Springer, 2007, pp. 139–157.

[7] Gao, Q., D. Gilbert, M. Heiner, F. Liu, D. Maccagnola and D. Tree, *Multiscale modeling and analysis of planar cell polarity in the drosophila wing*, IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) **10** (2013), pp. 337–351.

[8] Gilbert, D., M. Heiner, F. Liu and N. Saunders, *Colouring space-a coloured framework for spatial modelling in systems biology*, in: *International Conference on Applications and Theory of Petri Nets and Concurrency*, Springer, 2013, pp. 230–249.

[9] Heiner, M., M. Herajy, F. Liu, C. Rohr and M. Schwarick, *Snoopy–a unifying petri net tool*, in: *International Conference on Application and Theory of Petri Nets and Concurrency*, Springer, 2012, pp. 398–407.

[10] Iverson, K. E., *Notation as a tool of thought*, ACM SIGAPL APL Quote Quad **35** (2007), pp. 2–31.

[11] Jensen, K., *Coloured petri nets*, in: *Petri nets: central models and their properties*, Springer, 1987 pp. 248–299.

[12] Jensen, K., L. M. Kristensen and L. Wells, *Coloured petri nets and cpn tools for modelling and validation of concurrent systems*, International Journal on Software Tools for Technology Transfer **9** (2007), pp. 213–254.

[13] Krivine, J., R. Milner and A. Troina, *Stochastic bigraphs*, Electronic Notes in Theoretical Computer Science **218** (2008), pp. 73–96.

[14] Libkin, L. and L. Wong, *Query languages for bags and aggregate functions*, Journal of Computer and System sciences **55** (1997), pp. 241–272.

[15] Lopez, C. F., J. L. Muhlich, J. A. Bachman and P. K. Sorger, *Programming biological models in python using pysb*, Molecular systems biology **9** (2013), p. 646.

[16] Maus, C., S. Rybacki and A. M. Uhrmacher, *Rule-based multi-level modeling of cell biological systems*, BMC Systems Biology **5** (2011), p. 1.

[17] Milner, R., *Pure bigraphs: Structure and dynamics*, Information and computation **204** (2006), pp. 60–122.

[18] Muetzelfeldt, R. and J. Massheder, *The simile visual modelling environment*, European Journal of Agronomy **18** (2003), pp. 345–358.

[19] Oury, N. and G. D. Plotkin, *Coloured stochastic multilevel multiset rewriting*, in: *Proceedings of the 9th International Conference on Computational Methods in Systems Biology*, ACM, 2011, pp. 171–181.

[20] Paulevé, L., S. Youssef, M. R. Lakin and A. Phillips, *A generic abstract machine for stochastic process calculi*, in: *Proceedings of the 8th International Conference on Computational Methods in Systems Biology*, ACM, 2010, pp. 43–54.

[21] Pedersen, M., A. Phillips and G. D. Plotkin, *A high-level language for rule-based modelling*, PloS one **10** (2015), p. e0114296.

[22] Runge, T., *Application of coloured petri nets in systems biology*, in: *Proceedings of the Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Citeseer, 2004, pp. 77–96.

[23] Singh, D., A. Ibrahim, T. Yohanna and J. Singh, *An overview of the applications of multisets*, Novi Sad Journal of Mathematics **37** (2007), pp. 73–92.