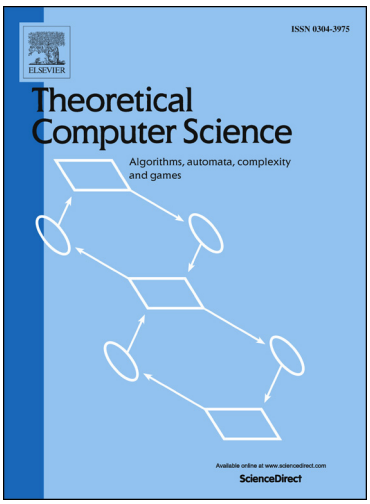
Chromar, a language of parameterised objects

Accepted Manuscript



Chromar, a language of parameterised objects

Ricardo Honorato-Zimmer, Andrew J. Millar, Gordon D. Plotkin, Argyris Zardilis

PII: S0304-3975(17)30599-6
DOI: http://dx.doi.org/10.1016/j.tcs.2017.07.034
Reference: TCS 11278

To appear in: *Theoretical Computer Science*

Received date: 31 March 2017
Revised date: 22 June 2017
Accepted date: 31 July 2017

Please cite this article in press as: R. Honorato-Zimmer et al., Chromar, a language of parameterised objects, *Theoret. Comput. Sci.* (2017), http://dx.doi.org/10.1016/j.tcs.2017.07.034

Chromar, a language of parameterised objects

Ricardo Honorato-Zimmer[a], Andrew J. Millar[b], Gordon D. Plotkin[a], Argyris Zardilis[b,*]

[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. However, the more complex the systems, the harder models become to read and write. The most common ways of writing models are by writing reactions on discrete, typed objects (e.g., molecules of different species), or by writing rate equations for the populations of such species. One problem with such approaches is that the number of species is often so large that the model cannot be realistically enumerated. Another problem is that the number of species and reactions is fixed, whereas biology often grows new compartments, which means new species and new reactions.

Here we develop a modelling language Chromar that provides an extension to the representation of reactions in which agents carry attributes with associated types (for example, Leaf agents all have a mass attribute). Dynamics are given by stochastic rules defined on groups of agents — for example all agents of a specific type — which means that enumerating the dynamics of each agent is not necessary. This compact representation addresses the first problem. Having such a more compact representation can also help make models a tool for knowledge representation and exchange instead of just simulation. Further, if we think of agents as the analogue of species in reactions, then creating a new agent of some type effectively creates new species, thereby addressing the second problem.

We have also developed an embedding of Chromar in the programming language Haskell and we demonstrate its applicability via two examples. Embedding Chromar in a general purpose programming language such as Haskell eases 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" [1]. Many excellent notations have been invented for biological models, some more general and some more domain specific. Nonetheless some specific models are hard to write in existing notations in a way that satisfies the above criteria. To illustrate the problem, consider modelling a growing array of cells, each having a concentration of some molecule X which diffuses between the cells, and which is produced and destroyed. The most common way to write such systems is by writing reactions on the types (species) of molecules

*Corresponding author
Email address: A.Zardilis@sms.ed.ac.uk (Argyris Zardilis)

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, with the diffusion occurring at rate d, and so on:

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

There are two problems with the above description. The first is that it is not very compact. 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 is that it is impossible to describe the creation of new cells because we need to create a new X-species for the new cell and new reactions, but the notation provides no way to express such a possibility.

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 $X_i \xrightarrow{d} X_{i+1}$, a generic production reaction $\varnothing \xrightarrow{\alpha} X_i$, and a generic destruction reaction, together with some way of generating new species. Our principal contribution is a notation that allows us to write such systems in a natural way, thereby solving the two main problems we noted, viz enabling compactness of representation for larger systems and providing a dynamic state-space. Specifically, our main contributions are as follows.

- We define Chromar, a rule-based notation with stochastic semantics (Sections 3 and 4) yielding a Continuous Time Markov chain (CTMC). The main entities in the notation are agents with attributes that are defined at the type level, so that every agent of that type has these attributes. For the above diffusion model we could have for example a type $\mathrm{X}(n : \mathrm{Int})$ with attribute n for the position of the X molecule in the array. Agents are instantiations of this type with concrete values for the attribute like $\mathrm{X}(n = 1)$ for a molecule in the first cell, $\mathrm{X}(n = 2)$ for a molecule in the second cell and so on. The states of the CTMC are multisets of agents. For example $\{\!|\mathrm{X}(n = 1), \mathrm{X}(n = 2), \mathrm{X}(n = 2)|\!\}$ would be a state corresponding to one X molecule in the first cell, and two in the second.

  The rules describe how agents are added to or removed from states at a more abstract level than individual agents. This is done by using patterns on the rule left-hand sides specifying the group of agents for which the dynamics are defined. In our example we could define a rule that applies on all agents of our X type ($\mathrm{X}(n = 1)$, $\mathrm{X}(n = 2)$ etc.). As each rule corresponds to multiple concrete reactions, this leads to a more compact representation of the model if we make our species correspond to attributes of some type. For example, in our case we could have $\mathrm{Cell}(\mathrm{pos} : \mathrm{Int}, \mathrm{X} : \mathrm{Int})$, then when we create new Cell agents we are creating new species that will automatically be picked up by the Cell rules. This helps us solve the second problem.

  We also define two extra features of Chromar: (i) *Fluents* — the incorporation of deterministically changing time-dependent values. These are important for modelling dynamics in a changing environment, and (ii) *Observables* — values calculated from a global view of the system. These are important in cases where a coarse-grained view of the system is needed. This may be because we cannot acquire atomistic data, or because we do not wish to model everything at the same level of detail. Observables

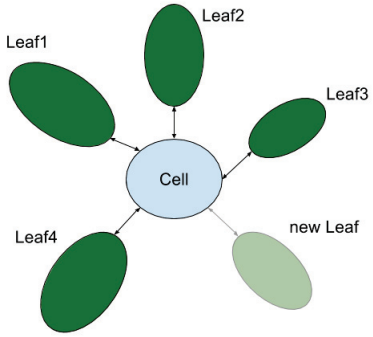


Figure 1: Our simple plant development model. All the interactions, that in this case are transfer of carbon, happen between the central Cell object that represents the molecular state of the entire plant and the Leaf objects, which act as carbon sinks. The carbon that goes to the leaves is either used 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, forming new sinks and increasing competition for carbon between the leaves, but also increasing the production of carbon by providing new green areas for photosynthesis.

also give us a flexible way to observe the state of the system that can be used to report the results of model simulations, as we often need time series of some observable on the state of the system rather than time series of the state itself. We refer to this nonconcrete version of the language as *abstract Chromar*.

- We then define a concrete realisation of Chromar as an embedded Domain Specific Language (DSL) inside Haskell, a functional programming language (Section 6)[2]. 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. Agent types, Fluents, and Observables are defined directly as Haskell expressions but rules are defined using abstract Chromar syntax via quotation [3]. The code for the language and simulator along with installation instructions is available online at: `https://github.com/azardilis/coloured-petri-nets/releases/tag/v0.2`

- We show the expressivity of our abstract notation as well as the advantages of having an embedding in a general purpose programming language via two examples (Sections 2 and 7).

2. An example: Plant growth

We will now give an overview of our notation using an example from plant development. The example considers a very abstract view of plant development that has enough details to demonstrate the main features of our notation. Our model is inspired by the Framework Model (FM) of Chew et al. [4], a modular whole-plant model that connects traditional plant biology representations of molecular processes with and representations of organ and whole-plant development processes.

The above-ground part of an Arabidopsis plant has a simple architecture before flowering: a collection of leaves arranged in a circle. Each leaf photosynthesises, creating the main currency, carbon; uses some carbon for maintenance and some for growth; and transfers any remaining carbon to the other leaves. The FM represents the Arabidopsis rosette (collection of leaves) with no preference in the transfer, thus we have an all-to-all communication. Similarly to the FM, in our model all the molecular processes reside in 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).

The processes that affect growth are as follows: we think of *carbon assimilation* 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 *new leaf creation*. 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 agents are the main entities in our language, let us consider what types of agents we should have to model the above system, given the above discussion. We will need:

- a Leaf type with attributes for mass and time of appearance: $\mathrm{Leaf}(\mathrm{age} : \mathrm{int}, \mathrm{mass} : \mathrm{real})$,
- a Cell type that represents our main plant 'cell' with an attribute, 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), and
- a Ros type that represents the entire Rosette with an attribute, n, to keep the current number of leaves: $\mathrm{Ros}(\mathrm{n} : \mathrm{int})$ (there will also only be one object of Ros type at any point)

We will use the current number of leaves relative to the index of a particular leaf as a proxy for the leaf's age.

For the *carbon assimilation* 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:

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

We can read this rule as saying that for any two Leaf and Cell agents, the Leaf agent remains the same and the Cell agent increases its carbon content by one. Note that we assign the values of the attributes of the the Leaf and Cell agents in the left-hand side of the rule to the variables i, 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 possible Leaf agent which leads to the compactness problem we have noted earlier (indeed, there would be infinitely many such possibilities). The implicit 'for-all' in the pattern on the left-hand side allows the rule to be applied to new leaves when they are created. For example if the state of the system has two leaves, the central cell, and the rosette agent:

$$\{|\mathrm{Leaf}(\mathrm{age} = 1, \mathrm{mass} = 10),\ \mathrm{Leaf}(\mathrm{age} = 2, \mathrm{mass} = 5), \mathrm{Cell}(\mathrm{carbon} = 20), \mathrm{Ros}(n = 2)|\}$$

then the rule is applicable to the two substates $\{|\mathrm{Leaf}(\mathrm{age} = 1, \mathrm{mass} = 10), \mathrm{Cell}(\mathrm{carbon} = 20)|\}$ and $\{|\mathrm{Leaf}(\mathrm{age} = 2, \mathrm{mass} = 5), \mathrm{Cell}(\mathrm{carbon} = 20)|\}$.

For *maintenance respiration*, the central Cell agent gives some carbon to a Leaf agent, with the amount of carbon needed for maintenance depending on the size of the Leaf:

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

Note the use of the condition $c \geq 1$ to make sure that the carbon levels do not become negative.

Next, *Leaf growth* depends on the mass of the leaf, its age (there is a limit on how much a leaf can grow so older leaves stop growing at some point), and the amount of carbon available:

$$\begin{aligned}&\mathrm{Ros}(\mathrm{n} = \mathrm{n}), \mathrm{Leaf}(\mathrm{mass} = m, \mathrm{age} = i),\ \mathrm{Cell}(\mathrm{carbon} = c) \xrightarrow{h(n-i,m,c)} \\ &\mathrm{Ros}(\mathrm{n} = \mathrm{n}), \mathrm{Leaf}(\mathrm{mass} = m + 1, \mathrm{age} = i),\ \mathrm{Cell}(\mathrm{carbon} = c - 1)\ [c \geq 1]\end{aligned}$$

Finally, for *leaf creation* we have:

$$\mathrm{Ros}(\mathrm{n} = n) \xrightarrow{k} \mathrm{Ros}(\mathrm{n} = n + 1),\ \mathrm{Leaf}(\mathrm{age} = n + 1, \mathrm{mass} = 0)$$

Fluents and Observables There are two problems with the above definition:

(i) The dynamics operate in a static natural environment making the model very detached from reality. For example our plants photosynthesise all the time whereas in reality they only do that during the day, and

(ii) We have two representations of the same process at two different levels of abstraction that have to be kept consistent with each other by the user. Specifically, we keep track of the number of leaves as an attribute of the Ros agent type that is just a function of the Leaves part of the system.

We next introduce two new notational features that solve these problems.

2.1. Fluents

To solve the first problem we introduce deterministically changing time-dependent values we call *Fluents*. These can be constructed through a combination of a small set of primitives and general expressions, taken from the Functional Reactive Programming (FRP) library in [5] (our Fluents are usually called Behaviours in FRP). For example we could write a fluent for light, which is a function from time to the Booleans, and another one for temperature that depends on light:

$$\text{light} = \textbf{repeatEvery}\ 24.0\ (\text{True}\ \textbf{when}\ (6 < \textbf{time} < 18)\ \textbf{else}\ \text{False})$$
$$\text{temp} = 21.0\ \textbf{when}\ \text{light}\ \textbf{else}\ 16.0$$

The assimilation rule can then be written as follows:

$$\text{Leaf}(\text{age}=i, \text{mass}=m),\ \text{Cell}(\text{carbon}=c) \xrightarrow{f(m,\text{temp})} \text{Leaf}(\text{age}=i, \text{mass}=m),\ \text{Cell}(\text{carbon}=c+1)\ [\text{light}]$$

Very often in Biology we have empirical relationships of various quantities with time. Fluents can be used to encode these as well. Such empirical relationships do not provide any mechanistic insight but they can be useful when we either do not have data or do not want to model more mechanistically using rules.

2.2. Observables

To solve the second problem, we introduce functions on the state of the system called *observables*. Observables can be constructed using database-inspired **select** and **aggregate** operations: we think of our state as a kind of database where we have a collection of agents rather than the more usual (and very similar) collection of records. The **select** operation specifies a part of the state, and the **aggregate** opereation specifies a combining function for 'folding' a collection of agents into a single value. For example, to count the number of leafs in a state, we could write:

$$\text{nl} = \textbf{select}\ \text{Leaf}\ ;\ \textbf{aggregate}\ \text{count}\ 0$$

where we first select anything that matches a Leaf pattern and then count the results by adding 1 for every Leaf starting from 0. The aggregation uses a simple count function $\lambda\ l\ n \rightarrow n+1$ which, given a Leaf l, increases the running count n by 1. Whenever an observable is used, one obtains its 'fresh' value, even when the underlying state has changed (e.g., by creating new leaves, in this case). For example our *Leaf growth* rule now becomes:

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

where we use our observable directly in the rule rate. There is no need to use the Ros agent any more as its only purpose was to keep track of the number of leaves.

Note that observables, like fluents, can be used to model parts of the system in cases where there is not enough knowledge about a process, or no desire to model at a more mechanistic level. In this case, for example, the leaves must have some mechanism for knowing their age but it is not central to our model so we use global knowledge via observables to abstract away from such details.

3. Abstract syntax of Chromar

In the previous section we gave some examples illustrating how the language works. Here we define the abstract syntax of the language. Regarding notation, we write $\underline{a}$ for a possibly empty sequence of objects, whose typical elements are given by mathematical expressions a. We write $\underline{a}.i$ for the ith element of the sequence and $|\underline{a}|$ for its length, and use commas for the composition of sequences. We assume we have a countable set of names N ranged over by n_a, n_f, id where n_a will be used for agent names, n_f for attribute names, and id for variables. For generality we assume that Chromar is parameterised by a set of expressions E (an expression language) ranged over by e, e_r, e_c and a set of attribute types (base types) $B \supseteq \{\text{bool}, \text{real}\}$ ranged over by b. We will use e_c as syntactic convenience for condition expressions and e_r for rate expressions. We further have agent types $\underline{n_f : b}$ that are sequences of typed attribute names. The set T is the union of base types and agent types ranged over by t (types, Figure 2). We will refer to Chromar where no choice is made about the expression language E and set of types T as *abstract Chromar* to distinguish it from any concrete realisation of the language, for example the Haskell embedding (described in Section 6) where we fix the set of expressions and set of types to be the set of Haskell expression and types respectively.

A base type environment $\Gamma = \underline{id : b}$ is a sequence of variable type bindings, and an agent type environment $\Delta = \underline{id : at}$ is a sequence of agent type bindings we require that the variables in environments are distinct. We write $\Gamma[\Gamma']$ for the new base type environment produced by extending or overriding Γ with new type bindings Γ'.

Regarding the expression language, E, we assume we have:

- A function $FI : E \to \mathcal{P}_f(N)$ that returns the finite set of free variables occurring in an expression.
- Typing rules of the form $\Gamma \vdash e : b$ that assign unique types for expressions in some base type environments.

The full syntax and typing rules for Chromar are given in Figure 2. We next go through the various syntactic constructs and explain their syntax and typing rules.

3.1. Agent declarations

Agent declarations define the sequence of typed attributes that all agents with some name n_a should have (agent decl, Figure 2). Using the model of the previous section as an example we had a Leaf type with attributes mass of type real and age of type int, that could be declared with:

$$\textbf{agent}\ \text{Leaf}(\text{mass} : \text{real}, \text{age} : \text{int})$$

We use the sequence of typed attributes $\underline{n_f : b}$ as the *agent type*. The typing rule (T-Intro) produces a new such type binding associating the agent name to the agent type.

3.2. Agent expressions

Agent expressions have a name and a finite sequence of attribute expressions (agent exprs, Figure 2). Agent expressions have a dual role: they are used in the left-hand side of rules as a way of binding values to variables and on the right-hand side of rules as a way of expressing the change in the left-hand side agents. We therefore have two distinct syntactic classes, one for each use:

(i) left-hand side binding agent expressions, a_l, where all attribute expressions are of the form $n_f = id$ (bind the value of an n_f to variable name id) and

(ii) right-hand side agent expressions, a_r, where all attribute expressions are of the form $n_f = e$ where the expression e indicates the change in the attribute values.

	Syntax		*Typing rules*
$a_d ::=$	$\textbf{agent}\ n_a(\underline{n_f : b})$	agent decl	$\vdash \textbf{agent}\ n_a(\underline{n_f : b}) : (n_a : \underline{n_f : b})$ (T-Intro) (all n_fs distinct)
$a_l ::=$	$n_a(\underline{n_f = id})$	left agent expr	$\Delta \vdash n_a(\underline{n_f = id}) : (\underline{id : b})$ (T-L-agent) $n_a : (\underline{n_f : b}) \in \Delta$, all ids distinct
$a_r ::=$	$n_a(\underline{n_f = e})$	right agent expr	$\dfrac{\Gamma \vdash \underline{e : b}}{\Gamma \mid \Delta \vdash n_a(\underline{n_f = e})}$ (T-R-agent) $(n_a : (\underline{n_f : b}) \in \Delta)$
$r ::=$	$\underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]$	rule expr	$\dfrac{\Delta \vdash \underline{a_l} : \Gamma' \quad \Gamma[\Gamma'] \vdash e_r : \text{real} \quad \Gamma[\Gamma'] \mid \Delta \vdash \underline{a_r} \quad \Gamma[\Gamma'] \vdash e_c : \text{bool}}{\Gamma \mid \Delta \vdash \underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]}$ (T-Rule)
$m ::=$	$\underline{a_d};\ \textbf{init}\,(\underline{a_r});\ \underline{r}$	model def	$\dfrac{\vdash \underline{a_d} : \Delta \quad \Gamma \mid \Delta \vdash \underline{a_r} \quad \Gamma \mid \Delta \vdash \underline{r}}{\Gamma \vdash \underline{a_d};\ \textbf{init}\,(\underline{a_r});\ \underline{r}}$ (T-Model)

Figure 2: Abstract syntax of Chromar with corresponding typing rules for each syntactic construct

The typing rule (T-L-agent) introduces new type bindings in the environment for the variables in the left agent expressions according to the agent type. Specifically if we have an agent type in the environment $n_a : (n_1 : b_1, \ldots, n_k : b_k)$ and a left agent expression $n_a(n_1 : id_1, \ldots, n_k : id_k)$, each $n_i : id_i$ in the interface of the left agent expression produces a type binding, $id_i : b_i$ according to the agent type resulting in bindings for the entire expression $(id_1 : b_1, \ldots, id_k : b_k)$. In Figure 2 we use a shorthand notation where we show the typing for a typical element of each sequence in the rule and then underline for the natural extension to the entire sequence. The typing rule (T-R-agent) makes sure that all expressions assigned to attribute names have the correct type according to the agent type. If we have an agent type binding $n_a : (n_1 : b_1, \ldots, n_k : b_k)$, the expression of each element, $n_i : e_i$, of the interface of a right agent expression with name n_a should have type b_i in a base type environment Γ. Again we use the shorthand sequence notation indicating the rule for typical elements of each sequence and then underline for the extension to the entire sequence. A further restriction implied by the typing rules is that agent expressions should define their interface fully, which means that they should define all the attributes indicated by their type.

3.3. Rule expressions

Rules, as we have seen from the examples, have a left-hand side (lhs) consisting of a sequence of binding agent expressions, a_l, a right-hand side (rhs) consisting of a sequence of agent expressions, a_r, a rate expression e_r, and a condition expression e_c (rule expr, Figure 2). The typing rule (T-Rule) first makes sure that the left-hand side agent expressions are well typed (i.e., that agents bind variables to all their attributes). Each left agent expression produces bindings for its variables and in $\Delta \vdash \underline{a_l} : \Gamma$, the environment Γ is the concatenation of the sequences of bindings produced by each element of the sequence. Then, in the environment extended with the type bindings produced by the lhs, we check that the rest of the rule is well formed. We need the extended type environment since ids on the rule lhs may appear in the rhs, rate and condition expressions.

3.4. Model definition

A full Chromar model is just a sequence of definitions. We have a sequence of agent declarations, $\underline{a_d}$ followed by the initial state definition, which is just a sequence of right agent expressions, and finally a sequence of rule expressions (model def, Figure 2). The typing rule (T-Model) constructs new agent type bindings for all the agent declarations in the model where Δ is the concatenation of agent type bindings produced by each agent declaration in the sequence. Then in a type environment extended with the defined agent types, the rule checks the initial state definition and the sequence of rules.

3.5. Syntactic extensions

Missing attributes We can relax the constraint of having to specify the full interfaces of agents in rules by defining a correspondence between left and right-hand sides of rules. The interface of any agent expressions on the left-hand side of rules with incomplete interfaces is filled with bindings to fresh names (*id*s) for the attribute names missing in the expression compared to the attribute names in its agent type. For example given an agent type $A(a : \text{int}, b : \text{int})$, an incomplete left agent expression $A(a = x)$ becomes $A(a = x, b = y)$, where y is a fresh identifier (i.e., one not occurring anywhere else in the lhs). For every $\underline{a_l}.i$ in the left-hand side we establish a positional correspondence to $\underline{a_r}.i$ in the right-hand side, if it exists. Then after the left-hand side agent expressions are completed, we complete the corresponding right-hand side expressions with the same fresh name bindings, if needed. Any right hand side agent expressions that do not have a corresponding lhs item need to be fully defined. Continuing with our example, the rule $A(a = x) \to A(a = x + 1)$ is completed to $A(a = x, b = y) \to A(a = x, b = y)$.

Attribute conditional expressions A common pattern to establishing a relation between agents is to use condition expressions to restrict the applicability of rules. Consider an agent type $A(a : \text{int})$. If we want a rule $A(\dots), A(\dots) \to \dots$ to be applicable to only a subset of $(A(\dots), A(\dots))$ pairs in the state we add a condition expression on the attributes values of the As to restrict the applicability of the rule: $A(a = x), A(a = y) \to \dots\ [y = f(x)]$. A syntactic extension that makes writing these conditions more readable allows writing conditions directly inside agent lhs expressions. In our example, instead of binding to y in the second lhs item and then putting the condition at the end, we can write the condition directly inside the agent making the lhs $A(a = x), A(a = f(x))$. This is just syntactic sugar as any rule lhs with conditional expressions can be rewritten as a binding-only lhs, plus a conditional expression. Any conditional expression in a rule lhs is rewritten as a binding to a fresh identifier and an equality condition on the fresh identifier taking us back to the form of our original rules.

4. Semantics of Chromar

For each base type constant $b \in B$ we assume we have an associated set of values $\mathrm{V_b}$, including $\mathrm{V_{bool}} = \mathbb{B}\ (= \{tt, ff\})$ for bool and $\mathrm{V_{real}} = \mathbb{R}$ for real. We have $\mathrm{V} = \bigcup_{b \in B} \mathrm{V_b}$ the union of these sets of values ranged over by v. A value environment $\sigma = id_1 : v_1, \dots, id_n : v_n$ with $v_1, \dots, v_n \in \mathrm{V}$ is a sequence of variable value bindings, where the variables are required to be distinct. We will sometimes treat value environments as functions with finite domain; for example to get the value of id_i in σ we may write $\sigma(id_i)$. We say that $\sigma = id_1 : v_1, \dots, id_n : v_n$ is a Γ-environment for a base type environment $\Gamma = id_1 : b_1, \dots, id_n : b_n$ if each $id_i : v_i$ in σ has an entry for that id_i in Γ of the correct type, i.e., $\sigma(id_i) \in \mathrm{V}_{b_i}$. For each Γ-environment σ, base type b, and expression e such that $\Gamma \vdash e : b$, the expression has an evaluation $[\![e]\!]_\Gamma(\sigma) \in \mathrm{V_b}$ (below, we often omit the environment subscripts on evaluation functions).

Regarding notation we write $\{a_1 \mapsto b_1, a_2 \mapsto b_2, \dots\}$ for finite partial maps, from A to B, with $a_1, a_2, \dots$ distinct elements of A and $b_1, b_2, \dots$ in B. For two such functions σ and σ' we write $\sigma[\sigma']$ for the partial function produced by using σ' to extend or overwrite the vaues produced by σ. We write $\{\!|a_1, \dots, a_n|\!\}$ for the (finite) multiset of elements of a set A whose elements, counting repetitions, are $a_i, \dots, a_n$, and take such multisets to be functions from A to $\mathbb{N}$ counting multiplicities (and so zero for all but finitely many arguments). We write $MS(\underline{a})$ for the multiset corresponding to the sequence $\underline{a}$. We write $m \preceq m'$ for the submultiset relation; this takes multiplicities into account and is defined pointwise. Finally, we write $\mathcal{M}[A]$ for the set of all finite multisets over a set A and $\mathcal{S}[A]$ for the set of all finite sequences over A.

4.1. Agents

Agent values are structures with a name and a finite partial map from attribute names to base values, $(n_a, \{n_f \mapsto v\})$. We write Av for the set of agent values, ranged over by av. We define the set of states, ranged over by s, to be $\mathcal{M}[\mathrm{Av}]$, the set of all finite multisets of agent values

To evaluate sequences of left agent expressions as multisets of agent values, suppose we have a typing $\Delta \vdash \underline{n_a(\underline{n_f = id})} : \Gamma$ of a sequence of left agents. Then, Γ is determined by Δ, and gives basic types for all the *id*s in the left agent expression, and so, for any Γ-environment σ we may define the evaluation $[\![\underline{n_a(\underline{n_f = id})}]\!]_\Delta(\sigma) \in \mathcal{M}[\mathrm{Av}]$ of the left agent sequence as follows:

$$[\![\underline{n_a(\underline{n_f = id})}]\!]_\Delta(\sigma) = MS(\underline{n_a(\underline{n_f = \sigma(id)})}$$

Similarly to evaluate sequences of right agent expressions to multisets of agent values, suppose that we have a typing $\Gamma \,|\, \Delta \vdash \underline{n_a(\underline{n_f = e})}$ of a right agent expression. Then for any Γ-environment σ we define its evaluation $[\![\underline{n_a(\underline{n_f = e})}]\!]_{\Gamma,\Delta}(\sigma)$ in $\mathcal{M}[\mathrm{Av}]$ as follows:

$$[\![\underline{n_a(\underline{n_f = e})}]\!]_{\Gamma,\Delta}(\sigma) = MS(\underline{n_a(\underline{n_f = [\![e]\!]_\Gamma(\sigma)})}$$

4.2. Rules

Rules induce (stochastic) reactions, and we discuss those first. Reactions are structures $\rho = l \xrightarrow{k} r$ with $l, r \in \mathcal{M}[\mathrm{Av}]$ and k a positive real number. We will assume that these names also act as accessors for the parts of the reaction structure, so for a reaction ρ, $l(\rho)$ returns its left-hand side and so on. We can apply the reaction ρ to a multiset of agent values s if $l(\rho) \preceq s$, when we obtain a new multiset $s - l(\rho) + r(\rho)$, which we write as $\rho \bullet s$.

Reactions give rise to stochastic matrices on $\mathcal{M}[\mathrm{Av}]$. A *stochastic matrix* on a set of states S is a *rate function* $Q : S \times S \to \mathbb{R}_+$ with Q zero on all but finitely many entries of any row. Note that the sum of two stochastic matrices is also a stochastic matrix, as is the zero matrix. Stochastic matrices encode the rate at which reactions occur in different states.

The rate at which a reaction ρ occurs in a state s depends on the number of ways in which $l(\rho)$ occurs in s. We define the *multiplicity* of a submultiset m of another (finite) multiset m' as the number of distinct times m appears as a submultiset of m', defined as:

$$\mu(m, m') = \prod_{x \in X} \binom{m'(x)}{m(x)}$$

Consider for example multisets $m = \{\!|a, a, b|\!\}$ and $m' = \{\!|a_1, a_2, a_3, b|\!\}$ where we label the a's in m' to distinguish them. The multiplicity of m in m' is three as m appears as a submultiset of m' in three distinct ways:

$$\mu(m, m') = |\{\{\!|a_1, a_2, b|\!\}, \{\!|a_1, a_3, b|\!\}, \{\!|a_2, a_3, b|\!\}\}|$$

With these preliminaries our of the way, we can define the semantics of rules as stochastic matrices (given suitable environments). Suppose that we are given a rule typing $\Gamma \,|\, \Delta \vdash \underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]$. Then we further have $\Delta \vdash \underline{a_l} : \Gamma'$ for a unique Γ'. We also have $\overline{\Gamma} \vdash e_r : \mathrm{real}$, $\overline{\Gamma} | \Delta \vdash \underline{a_r}$, and $\overline{\Gamma} \vdash e_c : \mathrm{bool}$, where $\overline{\Gamma} = \Gamma[\Gamma']$.

We first need to define the notion of a rule matching a state and the reaction denoted by a rule. For the first, we say that a Γ'-environment m is a *match* for the rule in a state $s = \{\!|av_1, \ldots, av_n|\!\}$ if $[\![a_l]\!]_\Delta(m)$ is a submultiset of s. Note that there are only finitely many possible such rule matches. For the second, suppose we are given a Γ-environment σ and a Γ'-environment σ', then we set

$$\mathcal{R}[\![\underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]]\!](\sigma, \sigma') \simeq \begin{cases} [\![a_l]\!]_\Delta(\sigma') \xrightarrow{[\![e_r]\!]_{\overline{\Gamma}}(\overline{\sigma})} [\![a_r]\!]_{\overline{\Gamma},\Delta}(\overline{\sigma}) & (\overline{\sigma} = \sigma[\sigma'], [\![e_r]\!]_{\overline{\Gamma}}(\overline{\sigma}) > 0, [\![e_c]\!]_{\overline{\Gamma}}(\overline{\sigma}) = tt) \\ \text{undefined} & (\text{otherwise}) \end{cases}$$

Note that this reaction only exists if (the denotation of) the rule rate is positive and (the denotation of) the rule condition holds. In case the reaction exists and σ' is a match for the rule in a state s, we say that the match is *proper* of the rule in state s, given σ.

We can now define the stochastic matrix associated to the rule, given a Γ-environment σ:

$$[\![\underline{a_l} \xrightarrow{e_r} \underline{a_r}\,[e_c]]\!]_{\Gamma,\Delta}(\sigma)(s,s') \;=\; \sum\{[\![e_r]\!]_{\overline{\Gamma}}(\overline{\sigma}) \cdot \mu([\![a_l]\!]_\Delta(m), s) \mid m \text{ is a match for } a_l \text{ in } s, \overline{\sigma} = \sigma[m], \\ s' = \mathcal{R}[\![\underline{a_l} \xrightarrow{e_r} \underline{a_r}\,[e_c]]\!]_{\Gamma,\Delta}(\sigma, m) \bullet s\}$$

For example, consider the left-hand expression $A(a = x), A(a = y)$ and state $\{\!|A(a = 1), A(a = 2)|\!\}$. We have two distinct matches $m_1 = \{x \mapsto 1, y \mapsto 2\}$ and $m_2 = \{x \mapsto 2, y \mapsto 1\}$. Next, consider the rule

$$r \;=\; A(a = x), A(a = y) \xrightarrow{f(x,y)} A(a = x + y), A(a = y - 1)\;[g(x, y)]$$

assumed well-typed given the agent type environment $A : (a : \text{int})$ and empty base type environment. Then, assuming, for example, that $f(x, y)$ denotes 3 in m_1 and the condition $g(x, y)$ holds there, m_1 yields a reaction from our rule r, namely:

$$\mathcal{R}[\![r]\!](\{\,\}, m_1) \;=\; A(a = 1), A(a = 2) \xrightarrow{3} A(a = 3), A(a = 1)$$

Assuming, further, that the condition does not hold in m_2, for the stochastic matrix of r we will have:

$$[\![r]\!](\{\})(\{\!|A(a = 1), A(a = 2)|\!\}, \{\!|A(a = 3), A(a = 1)|\!\}) = 3$$

If, on the other hand, the condition does hold in m_2 and $f(x, y)$ denotes 3 there, then the stochastic matrix of r will have value 6. This illustrates the fact that we may count the same reaction several times when applying a rule.

4.3. Stochastic semantics of models and a simulation algorithm

We now give the semantics of a model in terms of Continuous Time Markov Chains (CTMCs) over our set of states $\mathcal{M}[\text{Av}]$. A CTMC is a tuple (S, Q, I) with S being the set of all states, $Q : S \times S \to \mathbb{R}$ a stochastic matrix on S, and $I \in S$ the initial state.

A Chromar model $\underline{a_d};\ \mathbf{init}\,(\underline{a_r});\ \underline{r}$ generates CTMCs as follows. Suppose that $\Gamma \vdash \underline{a_d};\ \mathbf{init}\,(\underline{a_r});\ \underline{r}$. Then, for a unique Δ, we have $\vdash \underline{a_d} : \Delta$, $\Gamma \,|\, \Delta \vdash \underline{a_r}$, and $\Gamma \,|\, \Delta \vdash \underline{r}$. For any Γ environment σ we now define the evaluation $[\![\underline{a_d};\ \mathbf{init}\,(\underline{a_r});\ \underline{r}]\!](\sigma)_{\Gamma,\Delta}$ of the model as a CTMC by:

$$[\![\underline{a_d};\ \mathbf{init}\,(\underline{a_r});\ \underline{r}]\!]_{\Gamma,\Delta}(\sigma) = (\mathcal{M}[\text{Av}], \sum_{r \in \underline{r}} [\![r]\!]_{\Gamma,\Delta}(\sigma), [\![\underline{a_r}]\!]_{\Gamma,\Delta}(\sigma))$$

We remark that, in the implementation of Chromar, the Γ-environment needed for the semantics of a model, is supplied by the Haskell context in which the model is defined. We also remark that, with this definition, the same reaction may be counted more than once as it can occur in two different ways, either when applying a given rule (a possibility noted above), or when applying two different rules.

If we try to expand Chromar rules to the equivalent reactions for simulation, we may obtain infinitely many reactions (unless we constrain the types of the attributes in our agent types to be finite). 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, without constraining the available attribute types.

Specifically our algorithm is the usual SSA, but with an extra step (1) that dynamically creates the reactions based on the current state of the system. We assume a model and an environment σ, as above. The algorithm starts with state $s_0 = [\![\underline{a_r}]\!]_{\Gamma,\Delta}(\sigma)$ and then iterates the following sequence of steps as many times as desired.

1. For current state s, generate the multiset of all possible reactions for every rule:

$$R = \{\!|\mathcal{R}[\![\underline{r}_i]\!](\sigma, m) \mid i = 1, \ldots, |\underline{r}|,\ m \text{ is a proper match of rule } \underline{r}_i \text{ in } s, \text{ given } \sigma|\!\}$$

2. Calculate the total rate $k_T = \sum_{\rho \in R} R(\rho) \cdot (\mu(l(\rho), s) \cdot k(\rho))$. Halt if this is zero.
3. Pick the waiting time for the next reaction event from the exponential distribution with cumulative distribution function $F(t) = 1 - e^{-k_T t}$.
4. Pick exactly one of the reactions, choosing reaction $\rho \in R$ with probability $\frac{R(\rho) \cdot \mu(l(\rho), s) \cdot k(\rho)}{k_T}$.
5. If reaction ρ is picked then update the state to $s' = \rho \bullet s$.

Note that we take account of the multplicity of reactions here: the same reaction can occur from two different rules, or from one rule using different matches.

5. Other features

In this section we will define the Fluent and Observables features introduced in the example in Section 2. The new syntax and typing rules are summarised in Table 3.

5.1. Fluents

Fluents denote continuous functions of time and we borrow a syntax of Behaviours from Reactive programming to define them (fluent def, Figure 3). Most of the typing rules are obvious and will be made clearer when we give the semantics. The only subtlety is the (T-FDecls) typing rule for a sequence of fluent declarations. Since fluents can be used inside the definitions of other fluents, we cannot type check them independently. Therefore the typing rule (T-FDecls) proceeds in turn checking each fluent in a type environment that contains type bindings for the previous fluents in the sequence.

Each fluent expression evaluates to a total and continuous function of time and for some fluent f we define its evaluation $[\![f]\!] : \mathbb{R}_+ \to V$. First, we have the trivial **time** fluent, which is the identity fluent that gives a function that echoes back the given time:

$$[\![\mathbf{time}]\!](t) = t$$

The next conditional fluent constructor, is meant to be used to construct a new fluent as a piece-wise combination of other fluents. The conditional fluent chooses between two fluents f_1 and f_3 depending on the evaluation of a third fluent f_2. When f_2 evaluates to true it uses f_1 otherwise it uses f_3:

$$[\![f_1 \ \mathbf{when}\ f_2 \ \mathbf{else}\ f_3]\!](\sigma)(t) \ = \ \begin{cases} [\![f_1]\!](\sigma)(t) & (\text{if } [\![f_2]\!](\sigma)(t)) \\ [\![f_3]\!](\sigma)(t) & (\text{otherwise}) \end{cases}$$

The **repeatEvery** constructor provides a way to write the common motif of repeating behaviour over time:

$$[\![\mathbf{repeatEvery}\ e\ f]\!](\sigma)(t) \ = \ [\![f]\!](\sigma)(\mathrm{mod}(t, [\![e]\!](\sigma)))$$

Finally, any expression can be used as a fluent; the expression is 'lifted' to a time function:

$$[\![\mathbf{fexp}\ e]\!](\sigma)(t) \ = \ [\![e]\!](\sigma)$$

Note that in most of the examples in this text we use fluents directly in expressions without prepending the **fexp** constructor. For example we might write temp > 3 for some already define temp fluent instead of **fexp** (temp > 3).

To evaluate a sequence of fluents to a sequence of their values at some time suppose that $\Gamma \vdash \underline{f_d} : \Gamma_f$. Then for any Γ-environment σ, Γ_f-environment σ_f and a binding for time $(t : \tau)$ we define an evaluation of a sequence of fluents:

$$[\![\mathbf{fluent}\ n = f, \underline{f_d}]\!](\sigma, \sigma_f, t : \tau) = \big(n : f_\tau, [\![\underline{f_d}]\!](\sigma')\big)$$

where $f_\tau = [\![f]\!](\sigma, \sigma_f)(\tau)$ is a fluent evaluated at an environment σ, σ_f at time τ and $\sigma' = (\sigma, (\sigma_f, n : f_\tau), t : \tau)$ is the environment extended with the evaluation of the current fluent, f_τ so that its value can be used for the evaluation of subsequent fluents in the sequence that might be using it.

	Syntax		*Typing rules*
$f ::=$		fluent def:	
	time	identity	$\vdash \mathbf{time} : \text{real}$ (T-Time)
	f_1 **when** f_2 **else** f_3	condition	$\dfrac{\Gamma \vdash f_1 : b \qquad \Gamma \vdash f_2 : \text{bool} \qquad \Gamma \vdash f_3 : b}{\Gamma \vdash f_1 \mathbf{when}\ f_2\ \mathbf{else}\ f_3 : b}$ (T-cond)
	repeatEvery $e\ f$	repeat	$\dfrac{\Gamma \vdash e : \text{real} \qquad \Gamma \vdash f : b}{\Gamma \vdash \mathbf{repeatEvery}\ e\ f : b}$ (T-Repeat)
	fexp e	fluent expr	$\dfrac{\Gamma \vdash e : b}{\Gamma \vdash \mathbf{fexp}\ e : b}$ (T-Fexp)
$f_d ::=$	**fluent** $n = f$	fluent decl	$\dfrac{\Gamma \vdash f : b}{\Gamma \vdash \mathbf{fluent}\ n = f : (n : b)}$ (T-Fdecl)
			$\dfrac{\Gamma \vdash f_d : (n : b)}{\Gamma \vdash f_d, \underline{f_d} : (n : b), \Gamma[n : b] \vdash \underline{f_d}}$ (T-Fdecls)
$o ::=$	**select** n_a ; **aggregate** $e_1\ e_2$	obs def	$\dfrac{\Gamma \vdash e_1 : b' \qquad \Gamma \vdash e_2 : (\underline{n_f : b} \to b' \to b')}{\Gamma \mid \Delta \vdash \mathbf{select}\ n_a\ ;\ \mathbf{aggregate}\ e_1\ e_2 : b'}$ (T-Obs) $(n_a : (\underline{n_f : b}) \in \Delta)$
$o_d ::=$	**obs** $n = o$	obs decl	$\dfrac{\Gamma \mid \Delta \vdash o : b}{\Gamma \mid \Delta \vdash \mathbf{obs}\ n = o : (n : b)}$ (T-Odecl)
$m ::=$	$\underline{a_d};\ \underline{f_d};\ \underline{o_d};\ \mathbf{init}\ (\underline{a_r});\ \underline{r}$	model	$\dfrac{\vdash \underline{a_d} : \Delta \qquad \Gamma \vdash \underline{f_d} : \Gamma_f \qquad \Gamma \mid \Delta \vdash \underline{o_d} : \Gamma_o \qquad \Gamma \mid \Delta \vdash \underline{a_r} \qquad \Gamma[\Gamma_f, \Gamma_o] \mid \Delta \vdash \underline{r}}{\Gamma \mid \Delta \vdash \underline{a_d};\ \underline{f_d};\ \underline{o_d};\ \mathbf{init}\ (\underline{a_r});\ \underline{r}}$ (T-Model)

Figure 3: Abstract syntax of fluents, observables and corresponding typing rules.

5.2. Observables

Observables consist of a select statement followed by an aggregate statement (obs def, Figure 3). Observables give us a way to do reflection but in order to be able to do that we need a more powerful expression language E. Specifically in the expression language we need to be able to express the combining functions of aggregate statements, which means that the expression language should allow functional abstractions and have some representation for agents. Consequently the typing rules for the expression language should also handle both function and agent types.

Observables denote functions from states (multiset of agent values) to base values in V. Suppose we are given a typing rule $\Gamma \mid \Delta \vdash \mathbf{select}\ n_a\ ;\ \mathbf{aggregate}\ e_1\ e_2 : b'$. Then for any Γ-environment σ we may define

the evaluation $[\![\textbf{select } n_a \ ; \textbf{aggregate } e_1\ e_2]\!] \in \mathcal{M}[\mathrm{Av}] \to V$ of observable definitions as follows:

$$[\![\textbf{select } n_a \ ; \textbf{aggregate } e_1\ e_2]\!](\sigma) = [\![\textbf{aggregate } e_1\ e_2]\!](\sigma) \circ [\![\textbf{select } n_a]\!](\sigma)$$
$$[\![\textbf{select } n_a]\!](\sigma) = \lambda\, s \to \mathrm{plusMap}\ s\ \left(\lambda\, e \to \text{if } (n_a = \mathrm{nm}(e)) \text{ then}\{\!|e|\!\}; \text{else } \{\!|\}\!\right)$$
$$[\![\textbf{aggregate } e_1\ e_2]\!](\sigma) = \lambda\, s \to \mathrm{mFold}\ [\![e_1]\!](\sigma)\ [\![e_2]\!](\sigma)\ s$$

The plusMap function is the multiset equivalent of the list concatMap and mfold is the multiset equivalent of the list fold. The function nm acts as an accessor to the name part of an agent value. To evaluate a sequence of observable declarations suppose that $\Gamma \,|\, \Delta \vdash \underline{o_d} : \Gamma_o$. Then for any Γ-environment σ we have evaluation of a sequence of observables:

$$[\![\underline{\textbf{obs } n = o}]\!](\sigma) = \lambda\, s \to \underline{n : [\![o]\!](\sigma)(s)}$$

5.3. Chromar with Fluents and Observables

The definition of the language with the Fluent and Observable features follows from the definition of the core language in the previous sections. The abstract syntax of a Chromar model is extended with observable and fluents declarations (model, Figure 3). Since we are allowed to use time-dependent values inside rules, the transition rates between states are also time-dependent. This means the stochastic matrix of a rule are also parameterised by time: $Q : S \times S \times \mathbb{R}_+ \to \mathbb{R}_+$. Suppose that we are given a rule typing $\Gamma[\Gamma_f, \Gamma_o] \,|\, \Delta \vdash \underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]$. Then we further have $\Delta \vdash \underline{a_l} : \Gamma'$, $\Gamma \vdash \underline{f_d} : \Gamma_f$, $\Gamma \,|\, \Delta \vdash \underline{o_d} : \Gamma_o$, $\overline{\Gamma} \vdash e_r : \mathrm{real}$, $\overline{\Gamma} \vdash e_c : \mathrm{bool}$, where $\overline{\Gamma} = \Gamma[\Gamma_f, \Gamma_o, \Gamma']$. Then for any Γ-environment σ the stochastic matrix of a rule is given by:

$$[\![\underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]]\!]_{\Gamma,\Delta}(\sigma)(s, s', t) \quad = \quad \sum\{[\![e_r]\!]_{\overline{\Gamma}}(\overline{\sigma}) \cdot \mu([\![a_l]\!]_\Delta(m), s) \mid m \text{ is a match for } a_l \text{ in } s, \overline{\sigma} = \sigma[\sigma_f, \sigma_{obs}, m],$$
$$s' = \mathcal{R}[\![\underline{a_l} \xrightarrow{e_r} \underline{a_r}\ [e_c]]\!]_{\Gamma,\Delta}(\sigma[\sigma_f, \sigma_{obs}], m) \bullet s\}$$

where $\sigma_{obs} = [\![\underline{o_d}]\!](\sigma)(s)$ is a Γ_o-environment produced by evaluating all observables at current state s, $\sigma_f = [\![\underline{f_d}]\!](\sigma, t : \overline{t})$ is a Γ_f-environment produced by evaluating all fluents at current time t. The stochastic matrix is the same as before with the only difference that the given σ environment is extended with the evaluation of all fluents and observables at state s and time t. Note that using fluents means that the Continuous Time Markov Chain that a Chromar model generates becomes inhomogeneous in time. Therefore the standard Stochastic Simulation algorithm (SSA) does not apply since it assumes constant propensities between reactions. In practise, however, many times this discrepancy is not prohibitive and gives very similar results while avoiding the extra computational cost added by non-homogeneity. There are promising new methods for simulating CTMCs with time-varying propensities that alleviate some of the computational cost [6].

Here we use the approximate simulation method where the fluents and therefore all time-varying expressions in rules are only evaluated at the timepoints the usual SSA visits. This keeps their values constant between reactions at the expense of accuracy. The only change in our algorithm is the reaction generation step (1.) where the reaction generation happens in an environment extended with the evaluation of all fluents and observables at state s and time t:

$$R = \{\!|\mathcal{R}[\![\underline{r_i}]\!](\sigma[\sigma_f, \sigma_{obs}], m) \mid i = 1, \ldots, |\underline{r}|,\ m \text{ is a proper match of rule } \underline{r_i} \text{ in } s, \text{ given } \sigma|\!\}$$

where σ_f and σ_{obs} are the environments produced by evaluating all fluents at current time t and observables at current state s respectively. Since fluents are only sampled according to the discrete time-jumps followed by the simulation clock that means that in practise we only get a discrete approximation of the continuous functions denoted by the fluent definitions (see for example Figure 4). The accuracy of the approximation will depend on the sampling interval and how fast the fluent changes. If, for example, the fluent changes on a faster timescale than that of the model then the approximation will be poor. In practise, however, usually the fluents are on the same or slower timescale than that of the model since they are usually used to model

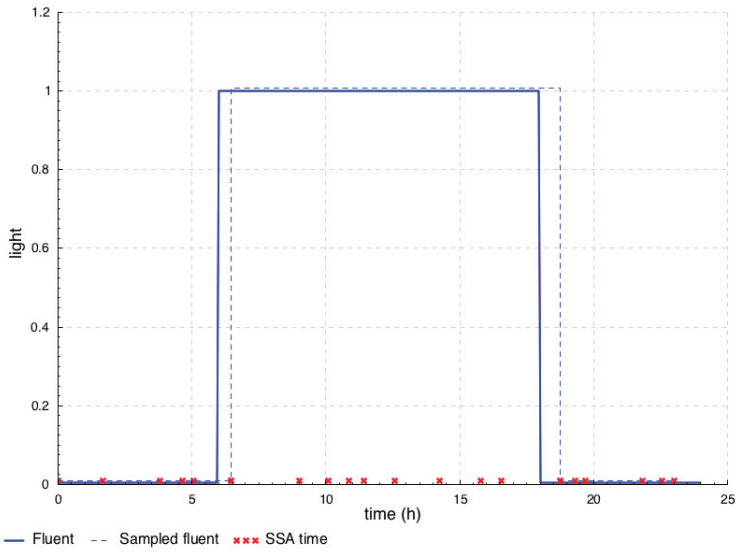


Figure 4: Ideal (solid line) versus practical approximation during simulation (dotted line) of the fluent for the light conditions in a day: True **when** ($6 <$ **time** < 18) **else** False, where we take True to be 1.0 and False to be 0.0. The fluent definition denotes a continuous function of time but in practise the function will only be sampled at the time points the SSA visits (red points). This means that practically we only get a discrete approximation of the fluent.

the context of the model, which means we get acceptable approximations. In the case of boolean fluents (like our light fluent from Figure 4) similar decisions on how to handle discontinuous changes have to be made in simulators that support SBML events for example. A common approach there is to have explicit handling of events in the simulation loop. The stochastic simulation in the 'iBioSim' tool, for example, checks the time of the next event and executes it if it happens before the time of the next reaction (algorithm 7 in [7]). It should be possible to extend these methods to Fluents of any type and add to our simulation algorithm if increased accuracy is needed.

5.4. Abstract Chromar

We refer to Chromar (defined in Sections 3, 4, and the current section) where no particular choice is made regarding the expression language E and set of types T as *abstract Chromar*. We say that *abstract Chromar* is parametric in these. This is in contrast to a particular concrete realisation of Chromar where we fix the expression language and set of types. Apart from the choice of expression language and types there is the more practical issue of the implementation of some concrete realisation of the language. In any implementation of the language (regardless of the choice of expressions and types), eventually all entities of the language have to become data structures in some programming language in order to set the model in motion on a computer. At the other extreme, the model definition could be written directly as constructs in some programming language (embedded DSL). In the following section we describe one concrete realisation of Chromar where we fix the expression language and set of types to be Haskell expressions and types respectively. We then describe the particular implementation choice via embedding.

6. Haskell embedding

In this section we will describe the implementation via embedding of one concrete realisation of Chromar where we fix the expression language E and set of types T to Haskell expressions and types respectively. In Section 5.4 we outlined the dichotomy between the two extremes of an implementation of any language. There are advantages and disadvantages to both approaches. Here we choose a middle ground: the model definition happens directly in programming language syntax except in the case of rules where we allow

their writing using the abstract syntax. This means that we get the convenience of the abstract syntax while allowing the full flexibility of the general programming language. In particular here we fix the set of expressions to be any valid Haskell expression and the types to be Haskell types. We will go through the different entities in Chromar and see how they are represented in Haskell. We will use `typeface` font for Haskell code to distinguish from the abstract Chromar syntax.

6.1. Agents

Agent type declarations of the form **agent** $n_a = (\underline{n_f : b})$ become Haskell types with n_a as the name of the type constructor and a sequence of named, typed fields (records). Sequences of agent declarations, $\underline{a_d}$, can be collected in a union type. For example our Leaf and Cell agent declarations from Section 2 are written thus:

```
data Agent
    = Leaf { age :: Int
           , mass :: Double}
    | Cell { carbon :: Double}
```

where the `data` keyword introduces a new datatype. Agent values are values of the `Agent` type, for example `Leaf{age=1, mass=1.5}` and `Cell{carbon=1.2}`. For multisets of Agents we introduce a type `Multiset a = [(a, Int)]` that maps agent values to their counts. The multiset type is parametric to an agent type `a`. Given a function `mset` that constructs a multiset from a list (analogous to our MS from Section 4), our initial state definition becomes:

```
init = mset [ Leaf{age=1,mass=0.2},
              Leaf{age=2, mass=0.1},
              Cell{carbon=0.5} ]
```

6.2. Rules (via Quasiquotation)

The conversion of rules from abstract Chromar to Haskell is less straightforward but we can think of rules as reaction generators: given a state (multiset of agent values) a rule denotes a reaction for each match to the state. A reaction is naturally defined as a record type parameterised on an agent type:

```
data Reaction a = Reaction
   { lhs :: Multiset a
   , rhs :: Multiset a
   , rate :: Double
   }
```

We cannot expect the user to write this rule (reaction generation) function though so we have made an easy definition of rules close to their abstract Chromar syntax, using Quasi-quotes. Quasi-quotes is a Haskell language extension that allows the use of special syntax (i.e., non standard Haskell syntax) inside `[| ... |]` brackets as long you provide a quoter, that is a function that takes the string inside the brackets and produces Haskell abstract syntax. The produced abstract syntax is injected by the Haskell compiler in place of the quotes. Our quoted rules have the following syntax that is almost the same as the abstract one we presented earlier:

$$\mathrm{r_q} ::= \underline{n_a\{\underline{n_f{=}id}\}} \rightarrow \underline{n'_a\{\underline{n'_f{=}e}\}} \; @e_r \; (e_c)$$

For example our growth rule from the plant model example becomes:

```
growth = [| Leaf{mass=m}, Cell{carbon=c} -->
            Leaf{mass=m}, Cell{carbon=c-1} @g m (c >= 1) |]
```

This looks very close to 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. A very wide range of expressions are therefore supported, without further effort.

Rule quoter function The quoter function takes a quoted rule expression and produces a Haskell function of type `Multiset a -> Multiset (Reaction a)` where `Multiset (Reaction a)` is a multiset of reactions (in fact we are producing the abstract syntax of the rule function so that it can be spliced inside the rest of the program at compile time). The rule function needs to find all matches (according to the rule lhs) and then generate a concrete reaction for each. This leads to the two parts of the rule functions we need to construct: the query part — for finding the matches — and the reaction generation part. For the query part we use list comprehensions and pattern matching for binding the variables in the rule lhs. The following example illustrates the translation from quoted rule expression to rule functions. Consider the agent declaration **agent** $A(a : \text{int})$ and rule $A(a = x), A(a = y) \xrightarrow{f(x)} A(a = x)\ [x > y]$. For the query part we make one comprehension generator statement for each element on the left-hand side. The statement for the first element becomes: `(A{a=x}, _) <- s` for some state s of type `Multiset Agent`. Pattern matching ensures that we only look at A's in the state and that we also bind the `x` variable. Note that we ignore the count of each agent-count pair since we later calculate the multiplicity of the entire lhs inside the state in the reaction generation part of the comprehension. For each such successful match of the first statement we continue to the second statement with `x` bound. The statement for the second element of the rule lhs is similarly `(A{a=y}, _) <- s`. With `x` and `y` bound we have our match m (as defined in Section 4.2) and we produce a reaction if the condition is true in the environment extended with m. The full comprehension statement is thus:

```
r :: Multiset Agent -> Multiset (Reaction Agent)
r = \s -> mset
     [ Reaction
      { lhs = mset [A{a = x}, A{a = y}]
      , rhs = mset [A{a = x} ]
      , rate = (g x) * m (mset [A{a = x}, A{a = y}]) s
      }
     | (A {a = x}, _) <- s
     , (A {a = y}, _) <- s
     , x > y ]
```

Function `mset` creates a multiset from a list. Each rule generates a multiset of reactions, as noted earlier, since the same reaction can occur from two different matches. Function `m` is the μ function that calculates the multiplicity of the match (defined in Section 4.3).

6.3. Fluent and Observable features

Fluents are provided as a combinator library and the implementation follows the abstract semantics and the definitions in [5]. The Fluent datatype is parametric to the type of values that the time function it represents returns: `data Fluent a = Fluent { at :: Time -> a }` with `Time` just a synonym for `Double`. We will say 'integer fluent' for a fluent of type `Fluent Int`, 'boolean fluent' for a fluent of type `Fluent Bool` and so on. The fluent constructors become higher-order functions that construct fluents from other fluents:

```
constant :: a -> Fluent a
time :: Fluent Time
when :: Fluent Bool -> Fluent a -> Fluent (Maybe a)
else :: Fluent (Maybe a) -> Fluent a -> Fluent a
repeatEvery :: Time -> Fluent a -> Fluent a
```

The condition fluent constructor becomes two functions here that are meant to be used one in prefix form (`when`) and one in infix (`else`) so that they read together in a similar way to the abstract syntax (`condition`,

Figure 3). Consider for example a fluent defining temperature that changes from 20.0 during the day to 10.0 during the night. The definition becomes:

```
temp = when day (constant 20.0) `else` (constant 10.0)
```

Note that we cannot use value expressions directly as fluents but we can 'lift' simple values to fluents using the `constant` constructor for example. The same goes for function expressions; in order to use a simple function between fluents we have to 'lift' it. For example a lifted integer addition function (`***`) applied to two integer fluents results in another fluent that at every time t returns the addition of the value of the two fluents at that time:

```
f1 *** f2 = Fluent { at = \t -> (at f1 t) + (at f2 t) }
```

We generalise lifting of functions by making our fluent type an instance of the Applicative typeclass. In case we are using fluents the rule functions produced by our rule quoter also take time as an argument so they have type `Time -> Multiset a -> [Reaction a]`. In order to allow fluents to be used directly inside expressions in rules, we translate all fluent variables to their interpretation in the quoted rule compilation. For example if we have a defined fluent `temp`, the expression `temp + 1.0` inside a rule will be compiled to `at temp t + 1.0`.

Observables are also given as a library of two functions:

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

These can be composed with normal function composition. For example for the number of leaves observable from our example we write:

```
nl = aggregate ((+) . const 1) 0 . select isLeaf
```

where `isLeaf` is a function of type `Agent -> Bool`. Note that instead of using Agent names to do the selection as in the abstract definition of the observables we use a boolean function on the Agent. Similarly to fluents, in order to be able to use observables directly in rule expressions our rule compilation translates observables to their interpretation at the current state. For example the observable for the number of leaves `nl` becomes `nl s`.

7. Second example: diffusion in a growing array of cells

We next present another example, that of a growing (one-dimensional) array of cells with a molecule X diffusing between them. This is a more complicated version of the example given in the introduction where, instead of growing the array of cells at one end, a cell at any position can divide. To do this, we employ the Cell type Cell(id : int, nextTo : int, X : int). Apart from its concentration of X molecules, a Cell agent also keeps some positional information. To fully determine the cell's position in the array we use its identifier attribute id, and that of its neighbour (given by its nextTo attribute). We can then define the neighbour relation using equality between the identifier of one cell and the identifier in the neighbour attribute of another cell.

In order to be able to give fresh identifiers to cells created by division we also need to know the current number of cells in the array. We do this using an observable:

$$\text{ncells} = \textbf{select}\ \text{Cell}()\ ;\ \textbf{aggregate}\ \text{count}\ 0$$

Similarly to our plant example in Section 2, we use a count function that, given a Cell c, increases the running count by 1: $\lambda\, c\, n \to n + 1$.

Turning to the dynamics of the system, diffusion is the transferance of molecules from one cell to another; we assume a cell's rate of diffusion to its left neighbour is equal to that to its right neighbour (see Figure 5). The rule for diffusion to the right is:

$$\text{Cell}(\text{nextTo}{=}p, \text{X}{=}x),\ \text{Cell}(\text{id}{=}p, \text{X}{=}x') \xrightarrow{k_d x} \text{Cell}(\text{nextTo}{=}p, \text{X}{=}x-1),\ \text{Cell}(\text{id}{=}p', \text{X}{=}x'+1)\ [x > 0]$$

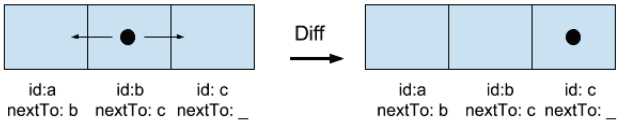


Figure 5: Diffusion rule. Any molecule inside a cell can move to the cell to its right or left with equal probability.

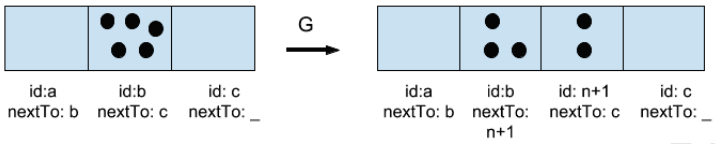


Figure 6: Growth rule. Here cell b divides creating cell $n+1$ (assuming we had n cells before the division), cell b moves to the left of the new cell. The 5 X molecules of cell b get divided between itself and the newly created cell.

where the rate of diffusion is proportional to the number of molecules, x, following standard mass-action kinetic rates, with constant of proportionality the diffusion constant k_d. Here we are using the syntactic extension of conditional attribute expressions (Section 3.5) for the id attribute of the second cell in order to limit our matches, so that the diffusion rule only applies to neighbouring pairs of Cell agents. There is an evident symmetric rule for diffusion to the left:

$$\mathrm{Cell}(\mathrm{nextTo}{=}p, \mathrm{X}{=}x),\ \mathrm{Cell}(\mathrm{id}{=}p, \mathrm{X}{=}x')\ \xrightarrow{k_d x'}\ \mathrm{Cell}(\mathrm{nextTo}{=}p, \mathrm{X}{=}x+1),\ \mathrm{Cell}(\mathrm{id}{=}p', \mathrm{X}{=}x'-1)\ [x'>0]$$

For growth, we create a new cell on the right of the dividing cell and split the X molecules as evenly as possible between the two cells:

$$\begin{aligned}&\mathrm{Cell}(\mathrm{nextTo}=p, \mathrm{X}=x)\ \xrightarrow{g(\mathrm{S})}\\ &\mathrm{Cell}(\mathrm{nextTo}=\mathrm{ncells}+1, \mathrm{X}=\mathrm{ceil}(x/2)),\\ &\mathrm{Cell}(\mathrm{id}=\mathrm{ncells}+1, \mathrm{nextTo}=p, \mathrm{X}=\mathrm{floor}(x/2))\end{aligned}$$

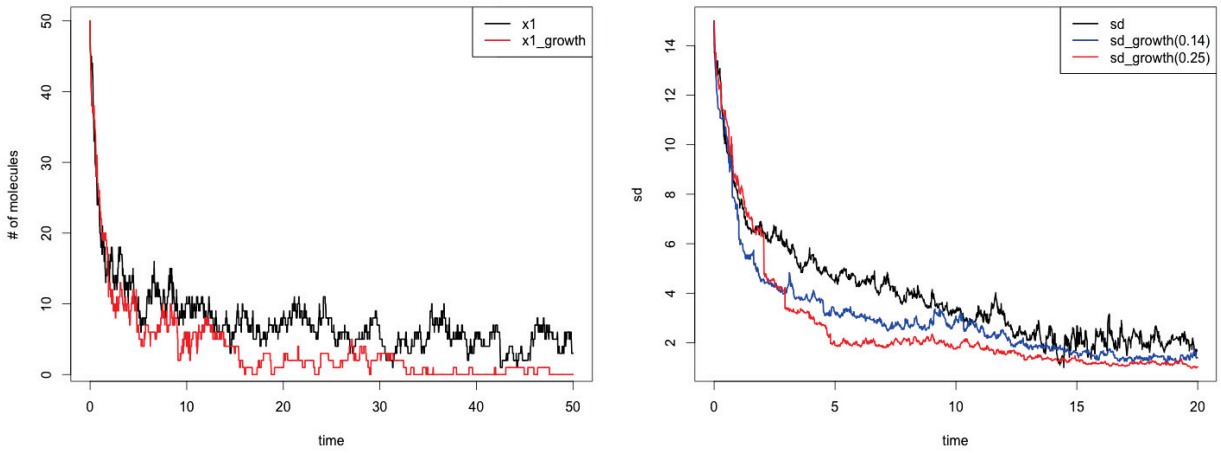
The dividing cell gets pushed to the left, keeping its id and changing its neighbour identifier to the identifier of the new cell. The new cell gets a fresh identifier from our cell counter observable ncells and a neighbour identifier from the old neighbour of its mother cell (Figure 6). We assume that growth is limited by the availability of some nutrient S through Monod's equation:

$$g(S) = g_{max}\frac{S}{K_s + S}$$

where g_{max} is the maximum growth rate. Availability of the nutrient cycles every 5 time units from some maximum value to zero. We define this as a fluent:

$$\mathrm{S} = \textbf{repeatEvery}\ 10.0\ \left(\mathrm{S}_{\max}\ \textbf{when}\ (0.0 \le \mathrm{time} \le 5.0)\ \textbf{else}\ 0.0\right)$$

It is interesting to compare the behaviour of this system with and without growth. Since we are not creating new molecules, we expect diffusion to spread the X molecules among the cells. With growth we expect fewer molecules per cell since the same number of molecules is spread over a larger number of cells - see Figure 7a 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 versus the diffusion + growth processes. In Figure 7b 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



(a) (b)

Figure 7: 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 $g_{max} = 0.14$ with a diffusion constant $k_d = 1$. 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 maximum growth rate $g_{max} = 0.14$, and diffusion+growth at maximum growth rate $g_{max} = 0.25$ all starting again with 10 cells. We can see that the faster the growth the faster the molecules get spread.

maximum growth rate $g_{\max} = 0.14$, and diffusion + growth with rate $g_{\max} = 0.25$. 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 reducing variability in cell contents more quickly.

We used our observables to extract and compute relevant quantities, such as the number of molecules in the first cell in the array Cell 1, and the standard deviation of the molecular counts across the array, thus obtaining a timeseries of observables. For example, to obtain the timeseries of X molecule counts in Cell 1 we use the following observable:

select Cell ; **aggregate** sumx1 0

where the combining function sumx1 is $\lambda\ c\ x \rightarrow$ **if** (id c) $= 1$ **then** $x + (X\ c)$; **else** n. Given a cell c, this function adds the cell's X contents to the running total x if the cell's id is equal to 1.

8. Comparison to related 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 [8] and, in a rule-based setting, in CSMMR [9]. Our notation is inspired by both of these. One can think of iy 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; one can also think of it as a simpler version of CSMMR with only colours left. Our embedding in a programming language increases expressive power, and fits with the availability of rich types.

However, there have been other languages that, while not giving the full power of a programming language, still allow for complex expressions to appear, e.g., inside rate expressions. For example in React(C) [10] rate expressions can be build from a subset of a functional programming language with a reflection option to get access to the full state of the system. This allows conditions to be encoded inside rates by setting the rates to zero if they are not met. Simulators for other widely used languages like KaSim (the simulator for

Kappa [11]) and BioNetGen [12] also allow more sophisticated rate expressions than traditional mass-action kinetic rates.

Our use of database inspired operations for the observables is also new and we have found it very useful in model building. The declarative nature of our multiset query primitives makes the definition of observables very intuitive. Similar database-inspired query operations on top of collections are used in LINQ [13] although the collections are usually taken to be lists not multisets. Buneman's comprehension syntax [14], a collection query language similar to practical database query languages, considers other types of collections, including multisets.

Attributes 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 such as Kappa [11] or BioNetGen [12]. However, note that 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.

From a more practical viewpoint, the issues we identified in the Introduction regarding dynamic creation of entities and compactness of representation are usually handled by modellers by either writing custom simulation software or by using a simulation framework. Agent-based modelling (ABM) simulation frameworks, in particular, share some characteristics with our work. In ABM frameworks the description of the process/model also happens declaratively through the description of the behaviour of classes of agents (see for example [15] used in systems biology).

The main difference from our work is that, whereas in ABM systems the main unit of description is the entire behaviour of each individual agent, in Chromar it is the rule, which can both describe a (possibly partial) behaviour of an individual agent and a synchronised action of two or more agents. We think this ability to specify sychronisation makes our models more natural and more easily written in a modular fashion. This improves readability. and, more practically, makes the resulting models easier to change and combine.

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

8.1. 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 (CPNs) 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 [8]. 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 8. We have two colourssets: 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 straightforward, coloursets are our agent types (records with named attributes), tokens are our agents, 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 [16] (and see [17, 18] for other examples). In these examples where the stochastic version was used its semantics are just given as a translation to the corresponding simple stochastic Petri Net.

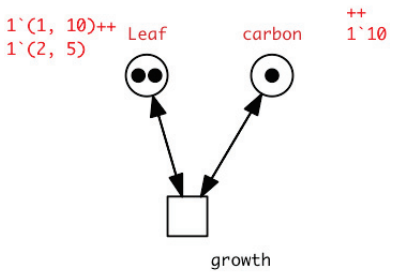


Figure 8: The growth rule as a Coloured Petri Net transition.

The problem here is that in many cases the unfolded simple Petri Net has an infinite number of reactions. This means that in order to do the unfolding at all, the types have to be finite sets, which further means that real values are not allowed. This is also reflected in the implementation of Coloured Petri Net tools where one can define a Stochastic Coloured Petri Net but the definition is unfolded before it is run [19]. Here we have defined the semantics on the coloured stochastic version directly and have a simulation algorithm where reactions are generated dynamically, instead of being created statically in the beginning. This allows us to have unbounded types for our attributes, including real numbers. Dynamic creation of reactions already appears in [20].

We can think of Chromar then as a stochastic rule-based (and therefore textual) language on top 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 that allows mixing the graphical definition with programming language constructs (ML language) [21]. 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 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.). This helps hide some of the complexity from the rules.

8.2. Simile

Simile is another graphical language that has similarities to our approach [22]. 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 [4] 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. Object relations can also be encoded through conditions to restrict the dynamics to a subset of the object types. This is similar to our conditional expressions that can also be used for the same purpose, for example in our Cell file model (section 7) we use a condition to restrict the diffusion rule to only neighbouring cells (neighbour relation).

In our plant growth example the growth of the leaves in Simile would be written as shown in Figure 9. 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 work 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 as they are in our language. In Chromar creating new objects or updating the values at the attributes of objects/agents work in the same way: by removing and adding objects. The main difference is that an execution of a

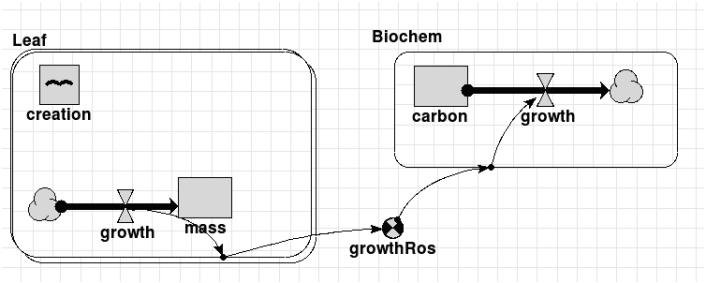


Figure 9: The growth rule as a Simile model.

Simile model ends up solving a system of ODEs whereas in Chromar we are in a stochastic setting. Again, the graphical notation of Simile becomes, in our experience, problematic for models with more than a few variables, thereby requiring further visualisation tools.

9. Discussion and Conclusions

We have defined an extension to the representation of reactions, where we extend simple named species to agents employing rich types, namely records with named typed attributes. 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 reaction setting. Moreover it sometimes helps writing systems that are difficult to write otherwise by allowing the dynamic creation of objects and therefore their attributes.

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. [23] allow embedded F# scripts inside LBS-κ, and in PySB [24] Kappa models are defined inside Python. Embedding a domain-specific language 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 fewer 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 4. One area of improvement could be in the reaction generation step where we currently generate all the active reactions at every step. In practise we do not need to completely regenerate the reactions 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. Techniques where the matches are generated only once at the beginning of the simulation and then updated according to state changes were already used in Kappa [25].

There are various ways our notation could be extended. Our rule left-hand sides are simple and can only select naturally 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). This is often unproblematic, 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 works 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 that case we had to encode the relation through identifiers and the next-to relation pairs were stored inside our objects. The selection was then restricted using attribute conditional expressions. 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. If we further had a 'nested-in' relation between cells and leaves then tracking the relations would need additional support. Ideally the language would track such relations and have 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 [26] and the 'nested-in' relation appears in [27] and [9]. 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 have been defined is Bigraphs [28]. Stochastic bigraphs in particular are applicable in the biological setting [29].

Our observables could be developed further and made into first-class entities in the language, for example by making them attributes of types. For instance in our plant growth example we would keep our initial Ros type and use our observable primitives to define its n attribute instead of defining the observables outside our agents. First-class observables would work particularly well with an extension for a native representation of levels (the 'nested-in' relation we noted earlier), in which case the idea of the agent attributes at a higher level being observables of objects at a lower-level would be both intuitive and powerful.

In conclusion, abstract Chromar and its concrete realisation in Haskell give a highly expressive language that is particularly good for the description of models with a simple structure, like the Plant example in Section 2 and the Cell file example in Section 7. The main ideas of Chromar like: two levels of dynamics (attribute and agent levels), flexible system of observation, combination of domain-specific and general programming languages should carry to other frameworks/languages, for example ones with more natural descriptions of more complex structure like the ones we described above.

References

[1] K. E. Iverson, Notation as a tool of thought, ACM SIGAPL APL Quote Quad 35 (1-2) (2007) 2–31.

[2] J. Gibbons, Functional programming for domain-specific languages, in: Central European Functional Programming School, Springer, 2015, pp. 1–28.

[3] G. Mainland, Why it's nice to be quoted: quasiquoting for haskell, in: Proceedings of the ACM SIGPLAN workshop on Haskell workshop, ACM, 2007, pp. 73–82.

[4] Y. H. Chew, 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 (39) (2014) E4127–E4136.

[5] Z. Wan, P. Hudak, Functional reactive programming from first principles, in: Acm sigplan notices, Vol. 35, ACM, 2000, pp. 242–252.

[6] M. Voliotis, P. Thomas, R. Grima, C. G. Bowsher, Stochastic simulation of biomolecular networks in dynamic environments, PLoS Comput Biol 12 (6) (2016) e1004923.

[7] L. H. Watanabe, C. J. Myers, Hierarchical stochastic simulation algorithm for sbml models of genetic circuits, Frontiers in bioengineering and biotechnology 2.

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

[9] N. Oury, 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.

[10] M. John, C. Lhoussaine, J. Niehren, C. Versari, Biochemical reaction rules with constraints, in: European Symposium on Programming, Springer, 2011, pp. 338–357.

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

[12] M. L. Blinov, J. R. Faeder, B. Goldstein, W. S. Hlavacek, Bionetgen: software for rule-based modeling of signal transduction based on the interactions of molecular domains, Bioinformatics 20 (17) (2004) 3289–3291.

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

[14] P. Buneman, L. Libkin, D. Suciu, V. Tannen, L. Wong, Comprehension syntax, ACM Sigmod Record 23 (1) (1994) 87–96.

[15] A. Solovyev, M. Mikheev, L. Zhou, J. Dutta-Moscato, C. Ziraldo, G. An, Y. Vodovotz, Q. Mi, Spark: a framework for multi-scale agent-based biomedical modeling, in: Proceedings of the 2010 Spring Simulation Multiconference, Society for Computer Simulation International, 2010, p. 3.

[16] Q. Gao, D. Gilbert, M. Heiner, F. Liu, D. Maccagnola, D. Tree, Multiscale modeling and analysis of planar cell polarity in the drosophila wing, IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) 10 (2) (2013) 337–351.

[17] T. Runge, 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.

[18] D. Gilbert, M. Heiner, F. Liu, 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.

[19] M. Heiner, M. Herajy, F. Liu, C. Rohr, 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.

[20] L. Paulevé, S. Youssef, M. R. Lakin, 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] K. Jensen, L. M. Kristensen, L. Wells, Coloured petri nets and cpn tools for modelling and validation of concurrent systems, International Journal on Software Tools for Technology Transfer 9 (3-4) (2007) 213–254.

[22] R. Muetzelfeldt, J. Massheder, The simile visual modelling environment, European Journal of Agronomy 18 (3) (2003) 345–358.

[23] M. Pedersen, A. Phillips, G. D. Plotkin, A high-level language for rule-based modelling, PloS one 10 (6) (2015) e0114296.

[24] C. F. Lopez, J. L. Muhlich, J. A. Bachman, P. K. Sorger, Programming biological models in python using pysb, Molecular systems biology 9 (1) (2013) 646.

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

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

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

[28] R. Milner, Pure bigraphs: Structure and dynamics, Information and computation 204 (1) (2006) 60–122.

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