Frabjous : A Declarative Domain-Specific Language for Agent-Based Modelling

Ivan Vendrov, Christopher Dutchyn, and Nathaniel Osgood

University of Saskatchewan

Abstract. Agent-based modelling (ABM) is a powerful tool for the study of complex, dynamic systems; but agent-based models are notoriously difficult to create, modify, and reason about, especially in contrast to system dynamics models. We argue that these difficulties are strongly related to the choice of specification language, and that they can be mitigated through use of functional reactive programming (FRP), a paradigm for describing dynamic systems, to specify agent behavior. We describe Frabjous, a new declarative language for agent-based modelling based on FRP, and discuss its software engineering benefits and their broader implications for language choice in ABM.

Keywords: functional reactive, functional programming, simulation, dynamic model, domain-specific language, agent-based simulation, agent-based modelling

1 Introduction

For systems that evolve continuously in space and time, the language of differential equations - honed by centuries of application to the physical sciences - has no substitute. Its syntax is extremely terse, with precise mathematical semantics that permit sophisticated analysis. While differential equations are not a natural fit for modelling populations of individuals (these being essentially discrete), the System Dynamics community has used them with great success to model the dynamics of large populations.

There are, however, a number of processes that are difficult to express with differential equations, such as those involving networks or a high degree of heterogeneity in the populations being modelled **?**. The need to model these processes is addressed by agent-based (AB) modelling, a more general approach, which involves specifying the behaviour of each individual in the population and allowing the global dynamics to emerge from the interaction of individuals.

The generality of agent-based modelling comes with a number of costs. With existing tools and frameworks, agent-based models are significantly harder to create, extend, and understand; significantly more expensive to calibrate and run; and significantly harder to mathematically analyze relative to models based on systems of differential equations **?**.

Although the increased cognitive and computational costs of agent-based models are to some degree unavoidable due to the models' increased complexity

and generality, we argue that these costs have been exacerbated by the use, in many AB modelling frameworks, of imperative languages like Java and C++ to express the model logic. While these languages are well-suited for general-purpose programming, they are not good specification languages, due to their verbose nature and hiding of essential details. They generally force modellers and users to think at a low level of abstraction, and fail to cleanly separate the relationships at the heart of the model from implementation details such as input/output, the time-stepping mechanism, and the data structures used **?**.

On the other hand, the underlying language of DE models is not imperative but declarative - rather than explicitly specifying rules by which model variables change, differential equations specify relationships between model variables that hold at all times. We believe that the declarative nature of DE models accounts for much of their success by simplifying model creation, modification, and analysis. It then stands to reason that AB models could be similarly simplified by basing them on an appropriate declarative language. To support this hypothesis, we develop such a language and use it to implement a number of standard models from the literature.

2 Background

In this section, we briefly describe the existing languages and technologies we used to create Frabjous, as well as explain why we chose them.

2.1 Haskell

Haskell is a purely functional programming language; that is to say, a Haskell program is a list of equations, each defining a value or a function.

As an example, consider the following Haskell code:

```
b = True
f x = 2 * x + 1
g = f 2
```

Here b is defined to be the Boolean value True, f is defined to be the function $f(x) = 2x + 1$, and g is defined as f(2), i.e 5.

Note that the equal sign in a Haskell definition denotes mathematical equality, not the assignment of a value to a variable. The values of b, f, and g, once defined, cannot change for the duration of the program.

Since Haskell lacks a mechanism for changing the value of a variable, it comes very close to the declarative ideal - specifying what things are, not how they change - and reaps the associated benefits: Haskell programs are often an order of magnitude shorter than programs written in imperative languages, are clearer to read, and are much easier to analyze mathematically. For these reasons, we have chosen Haskell as the base language of Frabjous, in that Frabjous code is largely composed of segments of Haskell code, and compiles directly to Haskell.

2.2 Functional Reactive Programming

An apparent weakness of Haskell is the difficulty of representing systems that vary with time, since there is no mechanism for changing state. As pioneered by Elliott and Hudak, functional reactive programming (FRP) is a paradigm that augments functional programming with "behaviours" - values that change over time - as well as a set of primitive operations on these values **?**. Arrowized functional reactive programming (AFRP) is a version of FRP that shifts the focus onto functions between time-varying values, called signal functions**?**. AFRP allows complex, time-varying, locally stateful systems to be written in a declarative fashion, as demonstrated by Courtney et al **?**.

The simplest AFRP operator is `constant`, which defines a constant function with a given value. So the output of the signal function `constant 1` is `1` at all times, and for all inputs. Integration can also be viewed as a signal function, since it operates on a function of time and produces a function of time. So

```
integral . constant 1
```

is a signal function that ignores its input and outputs the current time (`'.'` is the Haskell function composition operator).

Following the Netwire version of AFRP**?**, we allow signal functions to sometimes not produce values. For example, `rate` is a signal function that takes a time-varying number and produces a value at a rate specified by that number, so `rate . constant 2` produces twice (on average) in a given time unit. This allows us to model, for example, the Poisson process:

```
poisson lambda = count (rate . constant lambda)
```

where `count` is an operator that counts the number of instants that its argument produces a value, and `lambda` is the rate parameter of the Poisson process.

Another common signal operator is `after`, which produces after a given length of time. Two signal functions can be combined in parallel using the <|> operator, which acts like its left hand side when it produces, and like its right hand side otherwise, so

```
constant 1 . after 3 <|> constant 0
```

is a signal function that produces `0` for the first three time units, then `1` forever.

FRP can in fact be viewed as a generalization of differential equations. An FRP program is essentially a set of equations between time-varying values (in other words, functions of time). But where differential equations are limited to the standard mathematical operations such as addition, multiplication, and the differentiation operator, FRP adds a number of operators (such as `rate` and `count`) that act on and produce a much richer variety of functions. The differentiation operator is only defined on smooth functions in a vector space, but FRP operators can produce piecewise and step functions of arbitrary sets, which allows us to model discrete processes like the Poisson process above.

We use FRP as the basis for our ABM language because its declarative nature provides the transparency, clarity, and concision usually associated with

declarative modelling **??**, and because FRP has an explicit, formal semantics **?** that simplifies mathematical reasoning.

2.3 Frabjous

The generality of FRP comes at a cost, however. Understanding the syntax used in existing FRP libraries such as Netwire or Yampa **?** requires familiarity with advanced functional programming concepts such as monads **?**, arrows **?**, and applicative functors **?**. While these concepts allow for a great deal of conceptual elegance and generality, much of that generality is not necessary for ABM, and so a less general language with syntax oriented towards modellers rather than functional programmers would be desirable.

This was precisely the motivation for the development of the original Frabjous **?**, which showed that key mechanisms of ABM could be expressed with FRP in a much more concise and human-readable form than that of Java code generated by the popular AnyLogic framework. But Frabjous was a proof of concept, not a usable language for ABM; it placed a number of restrictions on agent state, behaviour, and network structure that drastically reduced its generality. In this paper, we redesign and extend Frabjous to yield a language that is still concise and readable, but is general enough to describe, in principle, any agent-based model.

3 The Frabjous Modelling Language

At the very highest level, a Frabjous model consists of a set of populations evolving in time. Each population is a dynamic collection of agents; agents can be added to and removed from populations by processes such as birth, death, and immigration. Furthermore, any pair of populations can be linked together by a network, which represents relationships between agents.

An agent consists of a set of attributes such as income, age, or level of education. Attributes are time-varying values which together comprise the evolving agent state, and are the main mechanism through which dynamic behaviour is specified.

The direct use of time-varying values is the only truly novel language feature of Frabjous, as the handling of populations and networks is mostly an issue of library design. Since time-varying values are the fundamental building blocks of Frabjous models, we proceed to discuss them in depth before presenting the full language.

3.1 Time-Varying Values

Time-varying values are not specified in Frabjous directly, but implicitly by means of signal functions (introduced in). In particular, the dynamics of an agent attribute are specified by a signal function from the entire agent to the

attribute; thus the value of an attribute at a particular time depends only on the values of other attributes up to that time.

Normally one defines signal functions by combining simpler functions with one of the provided operators. For example, we might define the attribute age of an agent as follows:

```
age = integral . constant 1
```

so the age of an agent is the amount of time elapsed since the agent was added to the model.

But how would we declare an attribute `isAdult`, which should be **`False`** during the first 18 years of the agent's life, and **`True`** from the 18th birthday on? This is a special case of a functional dependency, when the value of an attribute is a function of other attributes. To express functional dependencies in Frabjous, write it as a function, appending `(t)` to the declared signal function and the signal functions it depends on. This serves to make the dependence on time explicit:

```
isAdult(t) = age(t) ≥ 18
```

As a more elaborate example of this notation, the declaration

```
income(t) = if (isAdult(t))
                  then 1000 * age(t) * (uniform (0.5, 1.5))(t)
                  else 0
```

defines the income of adults at any given time to be 1000 times their age multiplied by a random number drawn from a uniform distribution between 0.5 and 1.5, and the income of non-adults to be 0.

4 An Extended Example: The SIR Model

To emphasize our approach to agent-based modelling as a generalization of system dynamics, we begin with a classic system dynamics model, generalize it to an agent-based model, and extend it in a number of directions, presenting the key features of Frabjous in an example-driven manner.

4.1 The System Dynamics SIR Model

The classic Susceptible, Infectious, Recovered (SIR) model, also known as the Kermack-McKendrick model of infectious disease, proposes to explain the dynamics of infectious diseases such as measles, chickenpox, and pertussis **?**. It is described by a system of three nonlinear differential equations:

$$\dot{S} = \beta SI \tag{1}$$

$$\dot{I} = \beta SI - \gamma I \tag{2}$$

$$\dot{R} = \gamma I \tag{3}$$

Here $S(t)$ is the number of Susceptible (non-infected but vulnerable) people, $I(t)$ is the number of Infectious people, $R(t)$ is the number of Recovered (previously infected and now immune) people in the population, β the infection rate, and γ the recovery rate.

The model makes a number of simplifying assumptions, first and foremost that the population exhibits homogenous mixing - infection is equally likely to spread between any pair of people in the population, as if everyone is milling around randomly in the town square. Unfortunately, in system dynamics there is no clean way to lift this assumption: homogeneity is unavoidable since there is no way to distinguish two people with the same infection state. To model heterogeneous populations, we must turn to agent-based modelling.

4.2 The Agent-Based SIR Model

There are a number of ways to generalize SIR to the agent-based domain; here we choose the one that seems most natural.

Instead of partitioning the population into three groups by infection state, we view it as a collection of individual people, each with an attribute that denotes the person's infection state. In Frabjous we declare this as follows:

```
data State = Susceptible | Infectious | Recovered
agent Person { infectionState :: State}
```

where **data** is a Haskell keyword that creates a new type with a given set of named values, similar to C++ or Java enums, **agent** declares a new agent type with the given name (Person) and list of attributes, and :: means "is of type". We also declare, for convenience, a boolean-valued helper function that determines whether a given Person is currently infectious:

```
infectious person = (get infectionState person) == Infectious
```

To relax the assumption that each person comes into contact with every other person in the population, we introduce a "neighbors" relation between people - a network which has an edge between two people if and only if they come into contact on a regular basis, i.e. if they are neighbors. We do this by amending the agent declaration for `Person` to the following:

```
agent Person { infectionState :: State, neighbors :: Vector Person}
```

where `Vector` is a standard Haskell collection, similar to a C++ vector - so each person has a reference to a collection of other people in the population. Now a susceptible person's chance of being infected depends not on the number of infected agents in the population at large, but only on its number of infected neighbours; everything else carries over from the system dynamics model. We can specify this by declaring infectionState to be a time-varying value as follows:

```
infectionState = hold . repeatedly transition
    where
        transition person =
```

```
    case (get infectionState person) of
      Susceptible → constant Infectious . rate .
                    infectionRate
      Infectious → constant Recovered . rate .
                    constant recovery_rate
      Recovered → never
infectionRate = per_contact_rate ∗ numContacts
numContacts(t) = count infectious neighbours(t)
```

The interesting part here is the transition function, which selects (using Haskell's `case` statement, an analogue to C++ or Java switch) between three possible evolution paths depending on the current state of the person.

A `Susceptible` person becomes `Infectious` with a rate determined by multiplying the number of infected neighbours it has by `per_contact_rate` (β in the differential equations model). An `Infectious` person will recover at a constant rate of `recovery_rate` (γ in the differential equations model), and if the current state is `Recovered`, the person's `infectionState` will never change.

Finally, the first line defines the overall behaviour of infectionState: an evolution path for `infectionState` is repeatedly selected using the transition function, holding the most recently produced value (the value of the last transition taken). Both `hold` and `repeatedly` are FRP operators in the Frabjous standard library.

4.3 Adding Time-Varying Infectiousness

A more implicit assumption of the SIR model is that all `Infectious` people are equally infectious at all times. In system dynamics, relaxing this assumption and allowing infectiousness to vary over time has been shown to yield a more accurate model for the spread of diseases such as norovirus **?**; how can we relax it in Frabjous?

First of all, we add a new attribute, `infectiousness`, to `Person`:

```
agent Person { infectionState :: State,
               neighbors :: Vector Person,
               infectiousness :: Double}
```

Now suppose that an agent's infectiousness starts at 0, jumps to 1 immediately after infection, then declines linearly to 0 in the next three days. We can specify infectiousness after infection as an explicit function of time:

```
after_infection t =  if t < 3 then 1 - 3∗t
                                   else 0
```

We can convert this function to a time-varying value (an implicit function of time) with the Frabjous operator `timeFunction`, which yields the following definition for infectiousness:

```
infectiousness = trigger (edge infectious)
                         (timeFunction after_infection)
```

where `trigger` is an FRP operator that produces nothing until its first argument (the 'trigger') produces a value, then acts like its second argument. In this case, `infectiousness` will stay at its initial value (presumably 0) until the agent first becomes infectious, at which point it will behave like `after_infection` - jumping to 1, then declining back to 0.

Finally, we change the calculation of `infectionRate` to be the sum of the infectiousness values of the person's neighbours (rather than just the count of infectives):

```
infectionRate(t) = sumBy (get infectiousness) neighbors(t)
```

4.4 Dynamic Networks

Up until now we have not bound the `neighbors` attribute to any value, which may seem like a cause for concern. In fact, if all we want is a static network, we need not specify it at all, since a Frabjous model only describes how the model changes, not its initial state. But we often do want the network to vary with time, whether randomly or in response to local changes in the agents.

We cannot, however, specify the dynamics of a network by binding `neighbors` to a time-varying value, like we would with any other attribute, since changing the network from the perspective of a single agent could create inconsistencies. Instead, we recognize that dynamic networks involve global interactions between agents, and thus specify them at global scope. For example, suppose we want neighbors to describe a random, dynamic network where each link has a 30% probability of existing at any point in time. We start by attaching an explicit name, `people`, to a population of `Person`s:

```
population people of Person
```

and declare the network as follows:

```
network people neighbours by randomLinks (const 0.3)
```

where `randomLinks` is one of the Frabjous standard library operators for creating dynamic networks. Its argument is a function from a pair of People to a probability (the probability that there will be a link between them at any point in time), and `const` is a standard Haskell function that ignores its second argument.

Networks can also be described between two different populations, which allows the specification of hierarchical models (e.g. people contained within cities).

5 Discussion

6 Usage

Although we've discussed the Frabjous language in some depth, we have not explained at all how a model written in it can be made to actually run. This is

intentional; the novelty of our approach lies entirely in how agent-based models are specified (the language), not how they are run and analyzed (the framework). For completeness, however, we briefly describe the current state of the Frabjous framework and the tradeoffs made in its design.

Although the language specifies a system evolving in continuous time, it is impossible, in general, to simulate it exactly, just as it is impossible in general to analytically solve a system of differential equations - so we must approximate. The approximation scheme we have chosen is a generalization of Euler integration: we run the model in discrete steps of user-specified length, interpolating the change that happens between steps. The user thus faces an explicit trade-off between accuracy and performance.

The Frabjous compiler presently generates a single Haskell function for every model, that takes four arguments - the initial state (all the agent populations), the timestep, the amount of time for which to run the model, and a function that specifies the desired output (e.g. all the agent states, or the percent of agents currently infected) - and returns an array of the desired outputs at each timestep.

7 Future Work

The primary area for future work is to make Frabjous a more complete framework, with the normal features and conveniences modellers expect, including support for

- collection of statistics over agent populations
- parameter calibration and sensitivity analysis
- convenient generation of initial populations
- graphical visualization of model outputs

In parallel, the Frabjous standard library needs to be refined and extended to cover a wider variety of agent behaviours and network types.

Expanding Frabjous in these directions will make it a useful language, at least for the purposes of pedagogy and communication; it will also pave the way for a direct quantitative and qualitative comparison to existing ABM frameworks.

To make Frabjous an industrial-strength ABM framework, performance issues must also be addressed. Thankfully, the declarative nature of the language provides many opportunities for optimization and parallellization. In particular, we are exploring the possibility of leveraging Data Parallel Haskell**?** as well as GPU acceleration to speed up the execution of Frabjous models.

8 Acknowledgements

This work was funded in part by a Natural Science and Engineering Research Council of Canada (NSERC) Undergraduate Student Research Award and NSERC Discovery Grant ¡FILL THIS IN¿.