Frabjous Language Reference
v0.3

Ivan Vendrov

University of Saskatchewan

Abstract. Agent-based modeling (ABM) is a powerful tool for the study of complex 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 by using functional reactive programming (FRP), a paradigm for describing dynamic systems. We describe Frabjous, a new language for agent-based modeling 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 modeling

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.

There are, however, a number of processes that are difficult to express with differential equations, such as those involving networks, history dependence, or heterogenous populations**??**. The need to model these processes is addressed by agent-based modeling (ABM), a more general approach which involves specifying the behavior of individuals and allowing the global dynamics to emerge from their interactions.

This generality comes with a number of costs. With existing tools and frameworks, agent-based models are significantly harder to create, extend, and understand; and significantly more expensive to calibrate and run 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 their increased complexity and generality, we argue that these costs have been greatly compounded by the use of imperative object-oriented languages such as Java and C++ to express model logic. As used in modern ABM frameworks, these languages force modelers and users to think at a low level of abstraction, and fail to cleanly separate domain-level structure from implementation details such as input/output, the

time-stepping mechanism, and the data structures used, which obscures the essential model logic **?**.

On the other hand, the underlying language of ODE models is not imperative but declarative: rather than describing 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 ODE models accounts for much of their success by simplifying model creation, modification, and analysis. It then stands to reason that ABM could be similarly simplified by basing it on an appropriate declarative language. To support this hypothesis, we developed Frabjous, a new declarative language for ABM. In this paper, we describe Frabjous and demonstrate its benefits on a standard example model.

2 Background

In this section, we briefly describe the existing languages and technologies we employ 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.

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, Haskell is the base language of Frabjous: 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 time-varying values 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***?**.

The simplest AFRP operator is `constant`, which defines a constant signal function. So the output of the signal function `constant 1` is `1` at all times, and for all inputs. Integration over time 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 starts producing values after a given delay. 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.

We base Frabjous on FRP because its declarative nature provides the clarity and concision associated with declarative modeling **??**. The utility of FRP as a specification language for complex systems has been demonstrated in a number of domains, including graphics **?**, robotics **?**, and games**?**.

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 **?** and arrows **?**. While these concepts allow for a great deal of conceptual elegance and generality, a domain-specific language that packages those portions relevant to ABM is desirable.

To explore this, the original version of Frabjous**?** realized concision and clarity compared to the popular AnyLogic framework. However, it placed severe restrictions on agent behavior and network structure. In this paper, we completely redesign 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 Modeling Language

At the top level, a Frabjous model consists of a set of populations evolving in time. Each population is a dynamic collection of agents. Each agent comprises a set of time-varying values (such as income, age, or educational level) called *attributes*. Agents can be added to and removed from populations by processes such as birth, death, and migration. Any pair of populations can be linked together by a network, which represents relationships between agents.

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 whose input is the entire agent.

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 the amount of time elapsed since the agent was added to the model:

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

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* between signal function, which is declared by appending `(t)` to all signal functions in the declaration:

```
isAdult(t) = age(t) ≥ 18
```

which makes explicit the signal functions' dependence on time.

4 An Extended Example: The SIR Model

In this section, we use Frabjous to implement an adaptation of the classic Susceptible, Infectious, Recovered (SIR) model of the spread of infectious disease, then extend it in a number of directions. Our purpose is to give examples of the clarity, concision, and flexibility of Frabjous models, paving the road for a deeper discussion in Section 5.

Our basic agent type is called `Person`, with an attribute for the agent's current 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 `enum`, `agent` declares a new agent type with the given name 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 capture the structured character of human contact patterns, we introduce a *neighbor* relation between people: a network which has an edge between two people if they come into contact on a regular basis. We do this by amending the agent declaration for `Person`:

```
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 the core dynamics of the model can be specified by defining `infectionState`:

```
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 its count of infected neighbours by `per_contact_rate`. An `Infectious` person will recover at a constant rate of `recovery_rate`, and if the current state is `Recovered`, the person's `infectionState` will never change.

Finally, the first line defines the overall behavior of `infectionState`: an evolution path 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.

4.1 Adding Time-Varying Infectiousness

An implicit assumption of the SIR model is that all `Infectious` people are equally infectious at all times. In ODE models, 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 HIV**?**; how can we relax it in Frabjous?

The first step is to add a new attribute, `infectiousness`, to `Person`:

```
agent Person { ... , infectiousness :: Double}
```

Then we specify infectiousness after infection as an explicit function of time, perhaps linearly decreasing over three days:

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

then convert it to a time-varying value with the Frabjous operator `timeFunction`, which yields the following definition:

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

where `edge` is an FRP operator that only produces a value at the instant that its argument becomes `True`, and `trigger` produces nothing until its first argument (the "trigger") produces a value, then acts as dictated by 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 neighbors:

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

4.2 Adding Dynamic Networks

So far we have not bound the `neighbors` attribute to any value. In fact, if all we want is a static network, we need not specify it at all, since a Frabjous model only describes change, not 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 specify the dynamics of a network by binding `neighbors` to a time-varying value as we would with any other attribute, since agents need to agree on network structure. Instead, we recognize that dynamic networks involve global interactions between agents, and 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 a Frabjous standard library operator that creates dynamic networks by connecting two agents with a given probability. Using the standard Haskell `const` function gives equal weights to all pairs; a different function could be used to implement preferential mixing.

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

4.3 Usage

The Frabjous compiler currently generates, for every model, a single Haskell function 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.

5 Discussion

As a language for ABM, Frabjous is distinguished by two key properties: the high-level constructs it provides to hide the computational details of common ABM mechanisms (state-charts, event queues, functional dependencies), and the language's declarative nature. These properties provide a number of important benefits, which we discuss below:

Concision. The use of a high-level language allows models to be expressed more concisely, as can be seen from the example implementation of a fairly sophisticated SIR model in only 14 lines of code. The program reads like an executable specification, going to the heart of the unique, defining characteristics of the model rather than low-level implementation details.

This offers a major benefit for scientific communication. One of the great challenges of conducting research with agent-based models is that they are hard to communicate in a fully transparent and reproducible way. By contrast, models written in Frabjous are sufficiently short and free of boilerplate that many of them can be feasibly provided in complete form within papers introducing them.

Clarity. The key benefit of Frabjous' declarative nature is that models are expressed directly in terms of *processes* rather than as sequences of imperative state changes. Since Frabjous models are composed of equations linking processes, the dynamic hypotheses made about the world are laid clear.

Correctness. The encapsulation of common ABM mechanisms together with greater code clarity both help reduce the risk of error during model creation. Concision and clarity together lead to fewer places where bugs can arise, and make it easier to perceive the essential governing logic of the model, which eases developing confidence in a model's correctness. Contrast this to an approach which interleaves the model logic with implementation and visualization details, where the low-level code must be understood in order to gauge correctness.

Flexibility. Modeling is typically undertaken for the purpose of discovery, which means that the model will frequently evolve in unexpected directions. The flexibility of Frabjous means there is less inertia when adding features or changing directions. For example, adding time-varying infectiousness, a drastic change of one of the core SIR mechanics, required only 4 lines of additional code.

This flexibility is largely due to the modularity enforced by a declarative specification: all the code that can modify a particular agent attribute *must* appear in the attribute definition, easing the identification of code that needs to change and minimizing unintended side effects from modification.

6 Future Work

The primary area for future work is to make Frabjous a more complete framework, with the normal features and conveniences modelers expect, including support for collection of statistics over agent populations, parameter calibration and sensitivity analysis, and graphical visualization of model outputs.

This will make it Frabjous 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. The declarative nature of the language provides many opportunities for optimization and parallelization. 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.