Towards Frabjous:

A Two-Level System for Functionally Reactive Agent-Based Simulation

Author name(s) withheld for anonymous review

Abstract

Computational epidemic models, such as the simulation of a disease moving through a population, are more frequently employed by health policy-makers. However, these models present several obstacles to widely adoption. They are complex entities, and hence have a high cost of development and maintenance. Current tools can be opaque, requiring multidisciplinary collaboration between a modeler and an expert programmer, with the attendent penalties for miscommunication and scheduling drastically slowing research. In this paper, we describe the use of Functional Reactive Programming (FRP), a programming paradigm created by imbuing a functional programming language with a sense of time, to represent agent-based models in a concise and transparent way. We document the conversion of several agent-based models developed in the popular hybrid modeling tool AnyLogic [16] to a representation in FRP. We also introduce Frabjous, a programming framework and domain specific language (DSL) for computational modeling. Frabjous is a unique system that generates human-readable and modifiable FRP code from a model specification, allowing modelers to have two transparent representations in which to program: a high-level model specification, and a full programming language with an agent-based modeling framework.

Keywords Functional reactive, simulation, dynamic model, domain-specific language, agent-based simulation, agent-based modeling

1. Introduction

While computational health models have proven valuable lenses for understanding public health issues, such models are only now beginning to be applied to many pressing public health issues [13, 14]. In part, this reflects limits in the software support for modeling; as a result of which, the development of these models can be cumbersome and slow . The application of dynamic models to public health is further limited by the fact that such models are frequently challenging to build, difficult to understand and evaluate, computationally expensive, and are difficult to share and collaboratively explore in policy decision team.

A key aspect of this problem lies in how the models are specified and developed. Many are represented by imperative programming languages such as Java and Objective-C [14]. Despite the use of frameworks and problem solving environments (PSEs) to handle the complexity of the code, these representations are fundamentally opaque and verbose. Developers must understand the programming language, the framework, and the model, which is challenging, and frequently demands considerable skills in software engineering.

Functional programming (FP), on the other hand, is well-documented for being concise and transparent[**? ?**]. As well, a declarative language are frequently easier to parallelize[**?**]. For an agent-based model, which has hundreds or thousands of agents working in parallel, taking advantage of parallel computing can lead to great gains in turnaround, hence facilitating experimentation and domain insight.

Though functional programming appears well-suited to representing simulation models, FP presents special difficulties for represent a time-varying system. This is because a declarative style avoids a direct mutable representation of the system state. In purely-functional programming languages like Haskell, system state is represented as a hidden parameter using *monads*[**?**]. In a non-purely-functional programming language like ML, code must take on an imperative style, reducing the elegance and transparency of functional programming language, by breaking equational reasoning. As an alternative, we adopt functional reactive programming (FRP), originally developed by Elliot and Hudak [5], to represent time as an intrinsic part of the paradigm.

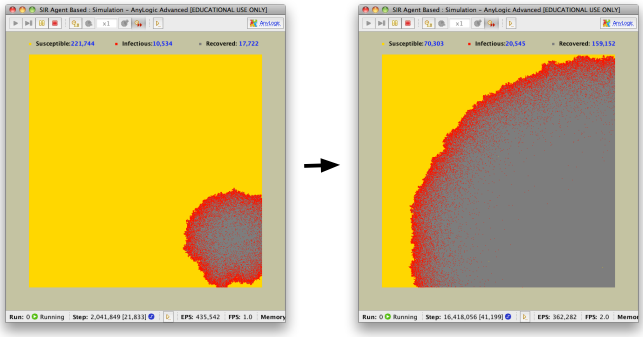

Figure 1. SIR Model. Yellow squares are Susceptible agents, Red are Infectious, and Grey are Recovered.

In this work, we provide two main contributions. First, we explore a promising method for specifying agent-based epidemiological models with functional reactive programming by translating several models into Haskell/Yampa, a FRP framework. This adds to the limited literature of this growing programming paradigm. Second, we present a domain-specific language, Frabjous, that generates readable and concise Haskell/Yampa code. This helps to overcome the steep learning-curve involved with using a functional programming language. Model specifications can be written, modified, or extended in either the Frabjous language or directly in Haskell/Yampa, leading to a flexible system that is always transparent for

model structure and parameters, but that still allows for an overview or for more detail and power.

We begin with a description of agent-based epidemiological simulation, and an overview of of FRP. We then show how a simple model can be represented in FRP. We then give an overview for the Frabjous language and system, including a description of the same model represented with Frabjous. We conclude with our overall findings and future directions for this work.

2. Computational Health Models

In this section, we provide a brief overview of computational health models, current tools, and challenges encountered by modelers.

2.1 The SIR Model

One example of an agent-based health model is the Susceptible, Infectious, Recovered (SIR) model (see Figure 1), a class-stylized abstraction of the dynamics of infectious diseases such as measles, chickenpox, pertussis [1]. Every agent in this system has one of three states - Susceptible, where the agent is not protected from the disease and may be infected; Infectious, where the agent is infected with the disease and may pass it on to Susceptible agents; and Recovered, where the agent no longer has the disease (and is temporarily or permanently protected from being infected). A simple version of this model has agents positioned on a two-dimensional grid, where each Susceptible neighbour of an Infectious agent has defined probability of being infected. We will use this mode as an example to illustrate our process and findings.

2.2 Current Tools

The past two decades have seen a rapid rise in the number of modeling packages supporting agent-based modeling. These include REPAST/Simbuilder, SWARM, SDML, NETLOGO and, most recently, AnyLogic [7, 8, 10, 11, 16, 17].

Most of these systems require custom coding to describe agent behavioural rules, network structure, as well as to constructing facilities to analyze model operation, etc. Traditionally, agent-based models have been created in imperative programming languages such as Java and Objective-C[13, 14, 16, 17]. Many researchers interested in agent-based modeling find themselves dissuaded by the need to learn general-purpose programming languages and principles and practices of software engineering in order to apply the tools. While the modeling environments provide much power, the use of such languages can distract the modeler from the high level characterization of the problem at hand in terms of domain concepts (the "what" of the simulation), and often require the modeler to operate among a distracting welter of low-level details concerning how the dynamic behaviour is to realized.

The languages also offer poor support for important domain-specific logic and metadata, such as are required for dimensional analysis, or for maintaining information on the quality of data used. In an unfortunately high fraction of cases [14], the modeler must serve as a software developer as well a task for which they typically lack training. Taken together, these factors result in longer development times, lack of model transparency and a higher risk of human error in models.

By contrast, functional languages and closely related declarative techniques have recently been demonstrated to offer admirable expressiveness, transparency of reasoning, and economy of effort in specifying dynamic systems[13, 14].

2.3 SIR Model Implementation in AnyLogic

AnyLogic is a Java-based modeling problem solving environment (PSE). It provides a graphical user interface (GUI) for graphical manipulation of models, but much of the functionality to build a model consists of Java code. Furthermore, because this code is hidden behind the GUI, a modeler must both understand Java and the AnyLogic object-oriented framework before they can build anything beyond a basic model. Even a programmer experienced with this tool must sometimes scan across properties associated with multiple components of the model to find salient features from the model. See Figure 2 for a screenshot of AnyLogic in use.

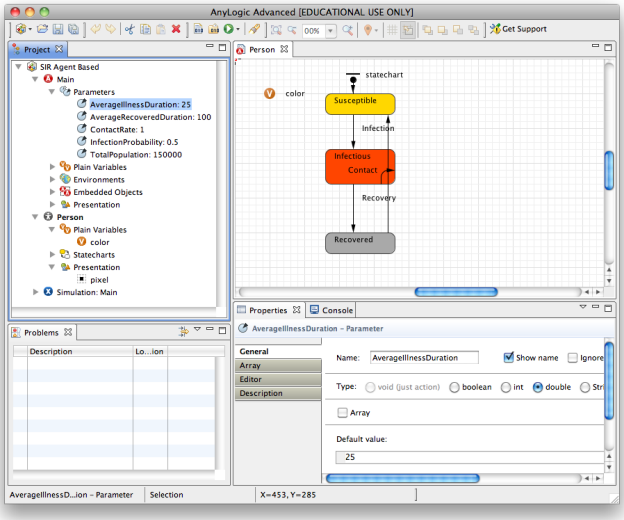

Figure 2. SIR Model implemented in AnyLogic. Note how, through hiding of information, the user must hunt to find details. On the other hand, if AnyLogic were to show all of the Java code, the result would not be much better.

3. Functional Reactive Programming

In this section, we summarize the technique of functional reactive programming (FRP) and describe its use to represent agent-based computational health models.

3.1 Functional Programming

Functional programming is a programming paradigm in which the computer languages define pure functions, without mutable state or other computational effects tying them together. Hence, sequencing of program execution, which is mandated by these effects, is only provided by function calls which yield constructed results. Hence, these languages are often called declarative languages. Examples of functional programming languages include Scheme, ML, and Haskell.[1]

Functional programming enriches the collection of concepts and structures available to the programmer, and thereby improves the programming exercise and results. In particular, it provides concise and expressive code, often orders of magnitude shorter, and often structured so the fundamental algorithm is clearly exposed. This means that functional programs can be transparent, enabling a computational modeler to debug by inspection. Essentially, reasoning about functional programs is simpler, involving less of the global execution state, because of the reduced and explicitly constructed (side) effects. Any sequencing is encapsulated into explicit sequencing constructs, generalizing function calls (e.g. monads). An important result of this that only necessary sequencing is

[1] In fairness, Scheme and ML provide some imperative features, but use of these is discouraged.

present, and for the remainder of the program (usually a large fraction), the runtime system can sequence it, potentially in parallel.

3.2 Functional Reactive Programming

Functional Reactive Programming (FRP) endeavors to eliminate this awkwardness by imbuing a functional language with a notion of time [2, 9]. This allows us to overcome the second problem of functional languages: the lack of a dynamic state. This gives us a suitable mechanism for time-based simulation. In addition, FRP gives us an effective mechanism for visualization of output - indeed, its original purpose was to produce animation [5].

In a functional paradigm, the user works with *functions* and *values*. In FRP, however, these concepts are lifted to *signal functions* and *signals*. A signal is a time-varying value, that is, a function of the type $Time \rightarrow \alpha$, and a signal function is a transformer with the type $Signal\ \alpha \rightarrow Signal\ \beta$ [2].[2] The result is a functional program that executes over time and that can react to different input.

To improve FRP code, Hughes developed the concept of the *arrow* [6]. An arrow is a generalization of *monads* that are used in non-reactive functional programming for sequencing. Monads encapsulate computation, forming a list or pipeline of computation that can make a programmer's job easier. Examples from Haskell include the Maybe monad, which allows computation to fail at any point in the pipeline, and the IO monad, which is used to transmit the external state of the world as a hidden parameter.

While monads form a pipeline, arrows form a graph. One can have multiple arrows converge and split. This leads to a natural representation of circuits and dynamic systems. Functional reactive programming uses arrows to have an *intrinsic sense of time*, making the programming of such systems much simpler. Paterson made this approach more transparent with his excellent syntactic sugar - arrows make almost graphical diagrams in the code, improving readability [15]. See Figure 3 for an example.

```
alarm :: SF WInput String
alarm = proc inp -> do
        dMon <- doorMonitor -< inp
        wMon <- windowMonitor -< inp
        tMon <- tempMonitor -< inp
        returnA -< result (dMon
                           `lMerge` wMon
                           `lMerge` tMon)
```

Figure 3. An example of FRP modeling an alarm system for a building, written in Haskell/Yampa. `doorMonitor`, `windowMonitor`, `tempMonitor`, and `alarm` are all signal functions (SFs). Paterson's syntactic sugar for arrows [15] makes the relationships between SFs clear.

3.3 SIR model written with FRP

We implemented the SIR model in the functional programming language Haskell, using Yale's Yampa [2] environment for FRP (see Figure 5). For visualization, we used wxFruit, an implementation of Fruit (Functional Reactive User Interface Toolkit) using wxWidget bindings [3, 4]. We used the Glasgow Haskell Compiler (GHC) version 6.12.3.

Computational models written in Haskell/Yampa were orders of magnitude shorter than their Java counterparts produced by AnyLogic (see Figure 4 for a comparison of source lines of code). In fact, many of the models were able to fit on one page of code. However, this conciseness comes at a cost. Though transparent to a programmer familiar with functional reactive programming, much of the syntax is difficult to understand by the layperson. *Higher order functions*, functions that return functions, as well as their signal function counterparts made the language complicated to those unfamiliar with this programming paradigm (See Figure 5 for an example).

To overcome this obstacle, we chose to develop an agent-based modeling framework and domain-specific language (DSL) in Haskell.

[2] In this context, α and β are type variables

Model	AnyLogic	Haskell/Yampa
Game of Life	839	70
SIR	1282	70
ESRD/TB	2222	166

Figure 4. Source lines of code for model implementations comparing Java code generated by AnyLogic for an executable file and our hand-programmed Haskell/Yampa implementations.

```
--states of agents
susceptible :: StateSF
susceptible = dSwitch (constant Sus &&&
                       (arr (/=[]) >>> edge))
              (\_ -> infected)

...

--runs the simulation with a step time of 'step'
runSimulation :: Time -> SF () StateGrid
runSimulation step = proc _ -> do
    rec
        e <- repeatedly step () -< ()
        let ev = catEvents events
        results <- parB board -< e `tag`
                                 (event [] id ev)
        let (kstates, events) = unzip results
    returnA -< kstates
```

Figure 5. Excerpts from the Haskell/Yampa implementation of the SIR model.

4. Frabjous

We present an initial specification of Frabjous, our domain-specific language and framework for developing functional reactive agent-based simulation ("FRABjouS"). Frabjous provides a high-level natural language to generate Haskell/Yampa code, which can then be further tweaked. Although our system is still a prototype, initial results show promise. The Haskell/Yampa code is concise and human-readable, a rarity for generated code. See Figure 10 for a high level system description, and Figures 6 and 7 for examples of the code.

4.1 Language Specification

Here we describe the Frabjous DSL specification. Please refer to Figure 10 for an overview of the specification. Frabjous describes a labelled transition system. Programmers define a *model*, with an *environment*, a set of *agents*, and a set of *networks* of agents.

A *model* is the top-level system abstraction described by Frabjous. It has a defined name, and encapsulates everything required to run an agent-based simulation.

```
discrete model sir

    on startup send "infect" to anyone

    network connections of people
        by vonneumann

    diagram flu starting with susceptible

        state susceptible displays yellow
            on receive "infect" switch to
                        infectious

        state infectious displays red
            on timeout 10 switch to recovered
            on rate 1 send "infect" to
                        neighbour connections

        state recovered displays blue

    agent people with
        flu
        population 100
```

Figure 6. SIR model written in Frabjous.

```
-- state definitions for diagram flu
diagram_flu = state_flu_susceptible

state_flu_susceptible = state output
                transitions
      where
        output = State_flu_susceptible
        transitions = transition1
        transition1 = receive "infect"
                        state_flu_susceptible
```

Figure 7. Excerpt of the Haskell/Yampa code generated by the Frabjous system prototype for the SIR model.

An *agent* is the bottom-level system abstraction. Several agents exist in the model. Each has a unique identifier, as well as a list of associated networks, confounders, diagrams, and populations (described later).

A *diagram* is shorthand for a state-transition diagram. A diagram has a unique identifier, a starting state, and a list of associated states.

A *state* is part of a diagram. Each state describes a state of an agent with respect to the distinctions captured in this diagram; for example, in the SIR model, there are three states: Susceptible, Infectious, and Recovered. Every state has unique identifier, a list of messages, a list of variables, a list of transitions to other states in the same diagram, and displays information for visualization when the model is in execution.

A *message* is the means by which agents communicate to each other. For example, in a SIR model, an Infectious agent might send a message to another agent to represent contact. If the receiving agent is in the Susceptible state, it will have a probability to switch states to Infectious. Each message has a trigger, which will send the message; a target, to which the message while send; a string representation of the message contents; and a variable.

A *trigger* is an event that will initiate the sending of a message or transition from one state in a diagram to another. A trigger can be: "startup", where the event will trigger when the simulation starts running; "rate", a probabilistic method representing a poisson process where a mean of *n* triggers will occur per unit of time; "timeout", where a trigger will occur after *n* time units; "expression", where a trigger will occur when a boolean expression evaluates to true; "enter", where a trigger will occur when an agent transitions to the specified state; and "leave", where a trigger will occur when an agent transitions from the specified state. An example of a trigger in the SIR model is the infection of patient 0 on startup.

A *variable* is stored numerical or text data that can be used to coordinate triggers.

A *Transition* is the means by which an agent switches between states. Each transition has a trigger, which determines when the state change occurs, and the identifier of the target state.

A *Target* is a set of agents to receive a message. At this point in time, we limit it to: Anyone, where one agent at random is selected to receive the message; Everyone, where every agent receives the message; Neighbour, where an agent's neighbor is selected at random from a specified network; and Neighbours, where an agent's entire neighborhood from a specified network receives the message. See Figure 9 for neighborhood information. For example, with the SIR model, patient 0 is Infected on startup by sending a message to Anyone, and each Infected agent sends a message to`Neighbour connections`, where `connections` is the only network in the model.

A *Confounder* is a way of combining diagrams, the internal state abstraction for each agent. Each confounder has a list of diagrams to combine, and a list of messages that provide greater control over ways these diagrams interact. In this way, health modelers can study the interaction between different diseases. For example, in the ERSD/TB model, modelers can make explicit that the two different state diagrams influence each other. In AnyLogic, this relationship is not clear from the graphical representation of the diagram (see Figure 8). Modelers must click on every transition arrow to find this relationship. In Frabjous, this interaction is explicit and modular, allowing the interaction to be tuned separately. This also supports reuse of code.

A *Network* is an arrangement of agents, determining how they are connected to each other. (elaboration needed!) Each network has a unique identifier, the identifier of a type of agent, and a structure detailing how neighborhoods are formed.

A *Structure* is a defined method for creating *neighborhoods* in a network. See figure 9 for more details. (elaboration)

An *Environment* is either discrete or continuous. This defines the type of space within which the agents exist, and determines the possible structures for a network. A discrete environment's agents form an *m* by *n* grid (lattice?) with each position occupied by a single agent. A continuous environment allows any number of agents to contain a position on a continuous 2-dimensional plane, usually determined randomly by some distribution. See figure 9 for more details.

4.2 System Description

The Frabjous system has two components: a compiler, which generates Haskell/Yampa code from a Frabjous model specification, and a library intended to make the code more readable. We hope to use the Frabjous specification as a way to rapidly develop initial models, and then provide the full programming environment of Haskell/Yampa to tweak or further improve more sophisticated models (although the need for this should be rare). The library is

```
-- Top level Model Structure
TModel = Model String TEnv [TMsg] [TNetwork] [TDiagram] [TConfounder] [TAgent]

-- Agent structure
data TAgent = Agent String --name [String]      --diagrams/confounders/population

-- Diagram structure
data TDiagram =         Diagram
                                String          --name
                                String          --starting state
                                [TState]        -- states

-- State structure
data TState     =       State   String [TMsg] [TVar] [TTransition] TDisplay

-- Message structure
data TMsg       =       Msg     TTrig TTarget String [TVar]


-- Trigger structure
data TTrig      =       Startup | Rate Double | Timeout Double
                        | ExprT BoolExpr
                        | EnterState String | LeaveState String | Receive String

-- Variable structure
data TVar       =       VInt Int | VString String | VDouble Double

-- Transition structure
data TTransition        = Transition    TTrig String    -- target state

-- Target structure
data TTarget =  Anyone | Everyone | Neighbour String | Neighbours String


-- Boolean expression
type BoolExpr = String

-- Confounder structure
TConfounder = Confounder        [String]

-- Network structure
TNetwork = Network      String  -- name
                        String  -- agentType
                        TStructure

-- Network structure definition
TStructure =    ByDistance Double       -- distance
              | Random Double           -- connectivity between 0 and 1
              | Moore
              | VonNeumann
              | Manhattan Int

-- Model environment type
TEnv = Discrete | Continuous
```

Figure 10. The Frabjous language specification.

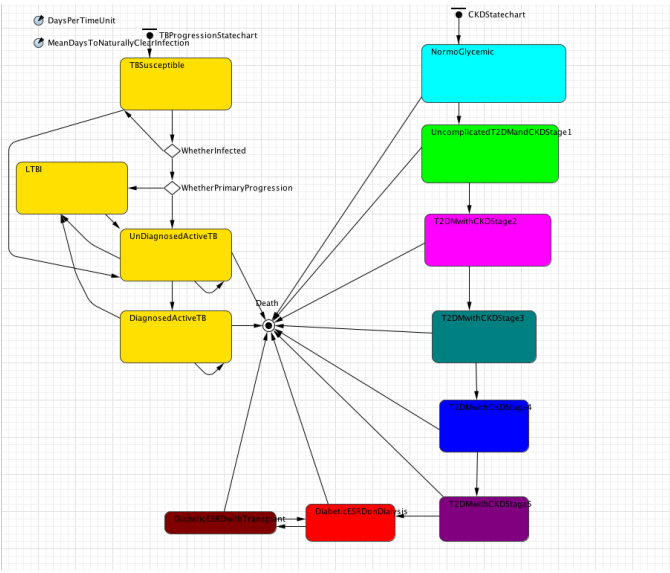

Figure 8. The state diagram for the ESRD/TB model in AnyLogic. This model has two disease that influence each other - the rate of progression in the ESRD model on the right depends on the current state in the TB model on the left. This is not immediately clear from the structure.

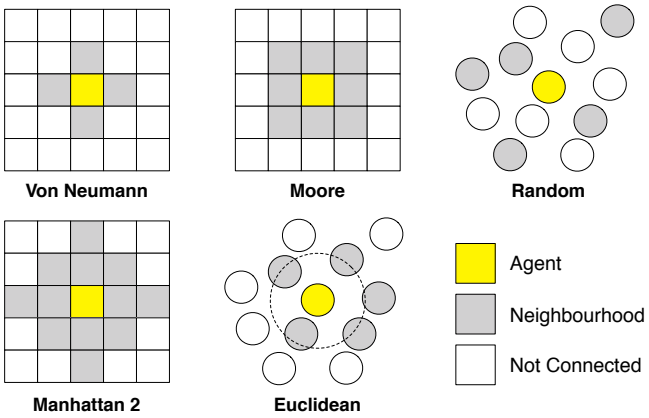

Figure 9. Examples of different types of neighbourhoods. Moore, Von Neumann, and Manhattan neighbourhoods are only available in a discrete environment where agents form a grid. Euclidean and Random neighbourhoods are only available in a continuous environment, where agents are distributed randomly in 2D space.

used to make the generated Haskell/Yampa code easier to work with for both those familiar and unfamiliar with FRP. Our goal is to have modelers develop at both levels, and be able to see the entire model at a glance from either view. See Figure 11 for a high-level overview of the system.

The Frabjous compiler was developed in Haskell/Yampa using the Parsec library to facilitate parsing[**?**]. The system is partially implemented; the parsing module is complete along with rudimentary type-checking, and the code generator is partially implemented. The Frabjous library is still under development. Similar to our purely Haskell/Yampa implementations, the Glasgow Haskell Compiler (GHC) will be used to compile generated code into a binary executable, although end-programmers will be able to use their preferred Haskell compiler.

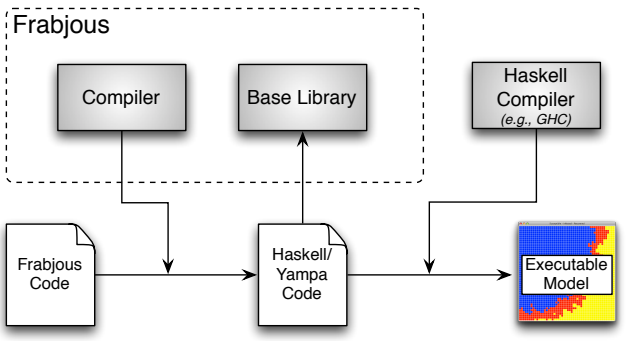

Figure 11. A high-level overview of the Frabjous system.

Frabjous's implementation was accomplished over the course of two to three weeks by a programmer with only 6 months of functional programming experience. Once the paradigm of functional reactive programming was understood and working with FRP implementations of existing models, development of a domain-specific language was extremely rapid. Functional languages are very amenable for developing domain-specific languages[**?**].

4.3 Language Features

Our partial implementation shows promising results in human-readable generated code. In addition, both representations are extremely concise (see Figure 4). Hand-crafted Haskell/Yampa code is approximately four times shorter than the Java code generated by AnyLogic. Frabjous-generated Haskell/Yampa code length is expected to be similar to that of our hand-crafted examples.

The higher-level character of the specification helps in maintaining modeler reasoning at the domain level, which is important in creating, extending, debugging, and seeking to understand a model. Through Frabjous, modelers can think in terms of the domain rather than at the level of a programming language. They can then fine tune the generated Haskell/Yampa code, still concise and transparent, but with the power of an entire programming language and supporting libraries.

4.4 The Confound Statement

One particularly exciting feature of Frabjous is the *confound* statement. This statement allows two different models to be combined, with additional members that override individually specified members. We hope to provide modelers with the ability to confound agent-based models in a similar fashion that functional programmers *compose* functions. The combinatorial explosion of combining different simulations (for example, simulating two or more diseases interacting within a single agent) is not well handled in aggregate simulation models [12]. While agent-based tools avoid this combinatorial explosion, modular specification of comorbidities is not supported in a modular fashion in existing agent-based simulation tools. We hope to provide increased manageability of large model sets; indeed, one can even imagine modelers releasing models to an online repository. With such a repository, other modelers would be able to take advantage of existing models to develop new ones and investigate interactions. Though such a tool remains firmly in the future, we hope that Frabjous's confound statement is a first step in that direction.

5. Summary

We have demonstrated a need for more transparent and efficient tools (in terms of both computational resources as well as human skills) in the domain of agent-based simulation. We have provided

a case for using functional reactive programming to meet this need. Initial exploration has demonstrated that this approach yields concise and transparent code for expert programmers, but a language barrier for less experienced programmers. The development of a domain-specific language and domain library (Frabjous) has compensated for these problems and presents a step towards a viable solution. As well, development of a DSL was extremely rapid, further supporting FRP as a promising approach for supporting modelers.

6. Future Work

We plan to complete the implementation of the Frabjous system by finishing the code generation and base library modules. Iterative design and evaluation with additional computational health modelers will provide a deeper understanding of how we can effectively develop a tool that is transparent, effective, and easy to use by computational modelers.

To address this need, there are planned improvements to evaluate the computational efficiency by parallelization. We plan to use CUDA to compute each agents' actions and interactions in parallel at each time step. Preliminary investigation suggests that this will be a simple conversion[**?**], but one that could offer important incentives for using the Frabjous framework.

Acknowledgments

Acknowledgements removed for anonymous review

References

[1] R. M. Anderson and R. M. May. *Infectious Diseases of Humans Dynamics and Control*. Oxford University Press, 1992. URL `http://www.oup.com/uk/catalogue/?ci=9780198540403`.

[2] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, pages 7–18, 2003.

[3] Antony Courtney, Bart Robinson, and Paul Hudak. wxfruit, March 2009. `http://www.haskell.org/haskellwiki/WxFruit`.

[4] Antony Alexander Courtney. *Modeling user interfaces in a functional language*. PhD thesis, Yale University, New Haven, CT, USA, 2004. AAI3125177.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. *International Conference on Functional Programming*, pages 263–273, 1997.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 2000.

[7] Nelson Minar, Rogert Burkhart, Chris Langton, and Manor Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Working Papers 96-06-042, Santa Fe Institute, June 1996. URL `http://ideas.repec.org/p/wop/safiwp/96-06-042.html`.

[8] Scott Moss, Helen Gaylard, Steve Wallis, and Bruce Edmonds. Sdml: A multi-agent language for organizational modelling. *Computational and Mathematical Organization Theory*, 4:43–69, 1998. ISSN 1381-298X. URL `http://dx.doi.org/10.1023/A:1009600530279`. 10.1023/A:1009600530279.

[9] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive progamming, continued. *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[10] M J North, T R Howe, N T Collier, and J R Vos. The repast simphony runtime system. In C M Macal, M J North, and D Sallach, editors, *Proceedings of the Agent 2005 Conference on Generative Social Processes Models and Mechanisms*, number 1, pages 151–158, 2005.

[11] Michael J. North, Nicholson T. Collier, and Jerry R. Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16:1–25, January 2006. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/1122012.1122013. URL `http://doi.acm.org/10.1145/1122012.1122013`.

[12] Nathaniel Osgood. Representing progression and interactions of comorbidities in aggregate and individual-based systems models. *Proceedings, The 27th International Conference of the System Dynamics Society*, page 20, July 2009.

[13] Nathaniel Osgood. Systems dynamics and agent-based approaches: Clarifying the terminology and tradeoffs. *Proceedings of the First International Congress of Business Dynamics*, 2006.

[14] Nathaniel Osgood. Using traditional and agent based toolsets for system dyamics: Present tradeoffs and future evolution. *Proceedings of the 2007 International Conference on System Dynamics*, 2007.

[15] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001. URL `http://www.soi.city.ac.uk/ ross/papers/notation.html`.

[16] XJ Technologies. *AnyLogic (Version 6)*. XJ Technologies, St. Petersburg, Russia, 2007.

[17] Seth Tisue. Netlogo: Design and implementation of a multi-agent modeling environment. In *In Proceedings of Agent 2004*, 2004.