Frabjous|| : A Declarative Framework for Agent-Based Modelling

Ivan Vendrov
Dept. of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
ivan.vendrov@usask.ca

Christopher Dutchyn
Dept. of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
dutchyn@cs.usask.ca

Nathaniel Osgood
Dept. of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
osgood@cs.usask.ca

ABSTRACT

-todo-

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications - *functional, parallel, data-flow*; I.6.5 [**Simulation and Modeling**]: Model Development; Modeling Methodologies; J.3 [**Life and Medical Sciences**]: Health

General Terms

Languages, Experimentation

Keywords

Functional reactive, functional programming, simulation, dynamic model, domain-specific language, agent-based simulation, agent-based modelling, data parallel programming

1. INTRODUCTION

For systems that evolve continuously in space and time, the language of differential equations (DE's) - honed by centuries of application to the physical sciences - has no substitute. Its syntax is extremely terse, and it has a precise mathematical semantics that permits 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 *(insert citations / examples)*.

There are, however, a number of processes that are difficult if not impossible to feasibly express with differential equations, such as those involving networks or a high degree of heterogeneity in the populations being modelled [17]. The need to model these processes is addressed by agent based modelling (ABM), a far more general approach to modelling populations, which involves specifying the behaviour of each individual in the population and allowing the global dynamics to emerge from the interaction of individuals.

With the added flexibility and power of agent-based modelling there come 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 [21].

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 ABM frameworks, of imperative languages like Java and C++. 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 lower 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 [17].

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 ABM could be similarly simplified by basing it 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.

possibly add more details about our contribution.

2. BACKGROUND

In this section, we briefly describe the existing languages and technologies, largely developed by the functional programming community, that we used to create Frabjous||.

2.1 Haskell

Haskell is a purely functional language; that is to say, a Haskell program is a set of equations defining values and functions. (as close as we've gotten to the declarative ideal)

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 [16, 17]. This in part reflects limits in the software support for modeling; the development of these models can be cumbersome, slow, and error-prone.

The application of dynamic models to public health is further limited because such models are frequently challenging to build, difficult to understand and evaluate, computationally expensive, and difficult to share and collaboratively explore in policy decision teams.

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 [17]. Despite the use of frameworks and problem-solving environments 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 frequently demands considerable skills in software engineering.

Functional programming (FP), on the other hand, is well-documented for being concise and transparent [2, 8]. Also, functional languages are frequently easier to parallelize [3, 20]. For an agent-based model with hundreds or thousands of agents working in parallel, parallel computing can greatly reduce the delay between scenario specification and the availability of results, facilitating experimentation and domain insight.

Although functional programming appears well-suited to representing simulation models, FP presents special difficulties for representing a time-varying system, since a functional style avoids a direct, mutable representation of the system state. As an alternative, we adopt functional reactive programming (FRP), originally developed by Elliot and Hudak [7], to represent time as an intrinsic part of the paradigm.

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, an FRP framework. Second, we specify a domain-specific language, Frabjous, that generates readable and concise Haskell/Yampa code. 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 transparent with respect to model structure and parameters.

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

3. COMPUTATIONAL HEALTH MODELS

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

3.1 The SIR Model

One example of an agent-based health model is the classic *Susceptible*, *Infectious*, *Recovered* (SIR) model (see Figure 1), which presents a stylized abstraction of the dynamics of infectious diseases such as measles, chickenpox, and 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 a defined probability of being infected. We will use this model as an example to illustrate our process and findings.

3.2 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

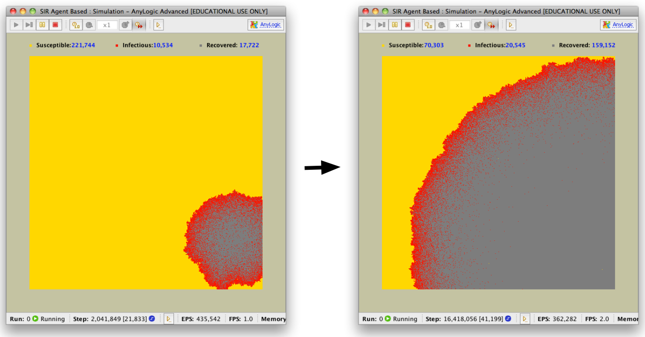


Figure 1: SIR Model. Yellow squares are Susceptible agents, Red are Infectious, and Grey are Recovered.

recently, AnyLogic [11, 12, 14, 15, 22, 23].

Most of these systems require custom programming to fine-tune simulation behaviour and analyze model operation. Traditionally, agent-based models have been created in imperative programming languages such as Java and Objective-C [16, 17, 22, 23]. Many health 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 necessary tools. These programming environments also offer poor support for important domain-specific logic and metadata, which are required for dimensional analysis or maintaining information on the quality of data used. In a high fraction of cases [17], the modeler must serve as a software developer, 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 [16, 17].

3.3 SIR Model Implementation in AnyLogic

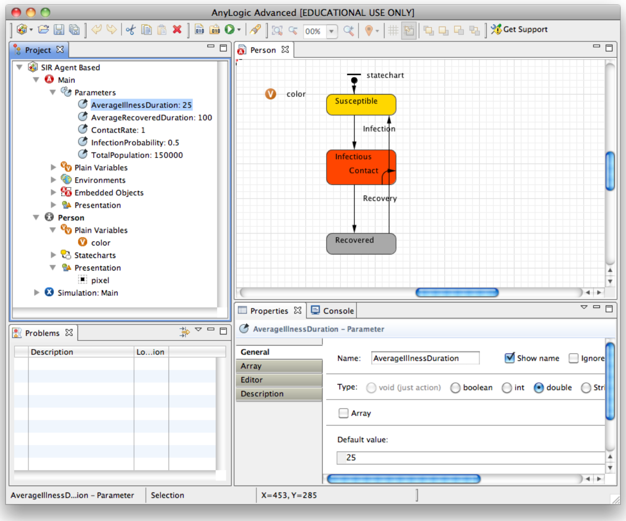
AnyLogic is a Java-based modeling problem-solving environment. It provides a graphical user interface (GUI) for graphical manipulation of models, but much of the functionality to build a model consists of Java code. Furthermore, because this code is hidden behind the GUI, a modeler must both understand 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.

4. FUNCTIONAL REACTIVE PROGRAMMING

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

4.1 Functional Programming

Functional programming is a programming paradigm in which 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. These languages are often called



AverageIllnessDuration - Parameter

Name: AverageIllnessDuration [✓ Show name] Ignore

Type: void (just action) boolean int [✓ double] Stri

Array

Default value: 25

Figure 2: SIR Model implemented in AnyLogic. Note that, 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.

declarative languages. Examples of functional programming languages include Scheme, ML, and Haskell.[1]

Functional programming provides concise and expressive code, often orders of magnitude shorter than imperative languages, and with a structure that often exposes the fundamental algorithm more clearly. 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 side effects are reduced and explicit.

However, by avoiding explicit manipulation of the global execution state, it can be difficult to represent a time-varying system in a functional paradigm. In purely-functional programming languages like Haskell, system state is represented as a hidden parameter using monads [24]. In a non-purely-functional programming language like ML, code must take on an imperative style, compromising elegance and transparency. This makes traditional functional programming poorly suited to agent-based modeling.

4.2 Functional Reactive Programming

Functional reactive programming (FRP) endeavors to accommodate sequencing in functional programs by imbuing a language with an intrinsic notion of time [4, 13]. This allows us to overcome the problem of representing a time-varying system in a functional language. In addition, FRP provides an effective mechanism for visualization of output - indeed, its original purpose was to produce animation [7].

To improve FRP code, Hughes developed the concept of the *arrow* [9]. Arrows are a generalization of *monads* that are used in functional programming for sequencing. Monads encapsulate computation, forming a list or pipeline of computation that can make a programmer's job easier. An example from Haskell is the IO monad, which is used to transmit the external state of the world as a hidden parameter.

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

While monads form a pipeline, arrows form a graph; multiple arrows can converge and split. This leads to a natural representation of circuits and dynamic systems. In the case of FRP, arrows are used to implicitly interpret time. Paterson made this approach more transparent with his excellent syntactic sugar - arrows essentially make graphical diagrams in the code, improving readability [19].

4.3 SIR model written with FRP

We implemented the SIR model in the functional programming language Haskell, using Yale's Yampa [4] environment for FRP (see Figure 3). We used the Glasgow Haskell Compiler (GHC) version 6.12.3. For visualization, we used wxFruit, an implementation of Fruit (Functional Reactive User Interface Toolkit) using wxWidget bindings [5, 6].

Computational models written in Haskell/Yampa were orders of magnitude shorter than their Java counterparts produced by AnyLogic (see Table 1 for a comparison of source lines of code). Note that while the SIR model is used here as a unifying example, additional models were implemented in Haskell/Yampa and Frabjous but cannot be reported in similar detail due to space limitations. These include Conway's Game of Life, an early example of cellular automata, and ESRD/TB, a model examining the representation and potential outcome of comorbidities of end-stage renal disease and tuberculosis.

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

Figure 3: Excerpt from the Haskell/Yampa implementation of the SIR model.

Many of the models 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 and FRP syntax made the language complicated to those unfamiliar with this programming paradigm (see Figure 3 for an example). To overcome this obstacle, we chose to develop an agent-based modeling framework and domain-specific language in Haskell.

5. 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

Table 1: Source lines of code for model implementations comparing Java code generated by AnyLogic for an executable file and our hand-programmed Haskell/Yampa implementations.

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

```
-- 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 4: Excerpt of the Haskell/Yampa code generated by the Frabjous system prototype for the SIR model.

be further fine-tuned. 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 Figures 4 and 5 for examples of generated code and its Frabjous specification.

5.1 Language Specification

Here we detail the Frabjous language 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.

An *agent* is the bottom-level system abstraction. 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 display information for visualization when the model is in execution. A *transition* is the means by which an agent switches between states. Each transition has a trigger (described below) and a target state.

A *message* is the means by which agents communicate to each other. Each message has a *trigger*, an event that will initiate the sending of a message or the 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 *x* triggers will occur per unit of time; “timeout”, where a trigger will occur after *x* 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. For example, in the 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. An example of a trigger in the SIR model is the infection of patient 0 on startup.

A *target* is a set of agents designated 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 neighbourhood from a specified network receives the message. For example, each Infected agent sends a message to `neighbour connections`, where `connections` is the only network defined in the model. We support a number of common neighbourhood definitions (Figure 6).

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. We defer a discussion of this to Section 5.4.

A *network* is an arrangement of agents describing how they are connected to each other. Each network has a unique identifier, the identifier of a type of agent, and a *structure*, a defined method for creating *neighborhoods* in a network. Network structure depends on the model’s *environment*, which is either discrete or continuous.

5.2 System Description

The Frabjous system has two components: a compiler, which generates Haskell/Yampa code from a Frabjous model specification, and a Haskell/Yampa library intended to make generated 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. Our goal is to have modelers develop at both levels, and be able to see the entire model at a glance from either representation.

The Frabjous compiler was developed in Haskell/Yampa using the Parsec library [10]. The system is partially implemented; the parsing module and rudimentary type-checking are complete and the code generator is partially implemented. The Frabjous library is still under development. The Glasgow Haskell Compiler (GHC) will be used to compile generated code into a binary executable.

5.3 Language Features

Our partial implementation shows promising results in human-

```
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 5: SIR model written in Frabjous.

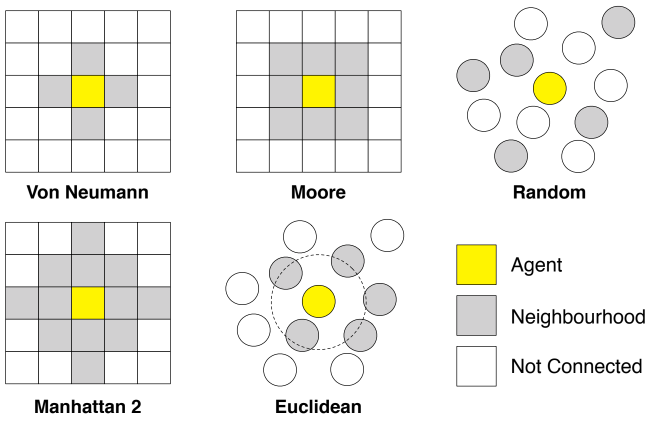


Figure 6: Examples of different types of neighbourhoods. Moore, Von Neumann, and Manhattan neighbourhoods are only available in a discrete environment where agents form a grid. Euclidean and Random neighbourhoods are only available in a continuous environment, where agents are distributed randomly in 2D space.

readable generated code. In addition, both representations are extremely concise (see Table 1). 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.

5.4 The Confound Statement

One particularly exciting feature of Frabjous is the *confound* statement. This statement allows two different models to be combined and interact in a defined way. We hope to provide modelers with the ability to confound agent-based models in a fashion similar to that used by functional programmers to compose functions. The combinatorial explosion of combining different simulations (for example, simulating two or more diseases interacting within a single agent) is poorly handled in aggregate simulation models [18]. While agent-based tools can avoid this combinatorial explosion, modular specification of comorbidities is not supported in a modular fashion in existing agent-based simulation tools. Comorbidities are common, and the opportunity to capture their dynamics in a clean fashion is a significant motivator for using agent-based simulation. Through the confound statement we hope to provide increased manageability of large model sets and better support collaboration between modelers.

6. SUMMARY

We have demonstrated a need for more transparent and efficient tools (in terms of both computational resources and human skills) in the domain of agent-based health simulation. We have provided a case for using functional reactive programming to meet this need. Initial exploration revealed 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) compensates for these problems and presents a step towards a viable solution.

7. 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 health modelers will provide a deeper understanding of how to effectively develop a tool that is transparent, effective, and easy for modelers to use.

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 [3], but one that could offer important incentives for using the Frabjous framework.

8. ACKNOWLEDGEMENTS

This work was funded in part by a Natural Science and Engineering Research Council of Canada (NSERC) Undergraduate Student Research Award (USRA), an NSERC CGS M scholarship, and the NSERC Discovery Grants program.

References

[1] R. M. Anderson and R. M. May. *Infectious Diseases of Humans Dynamics and Control*. Oxford University Press, 1992.

[2] J. Backus. Can functional programming be liberated from the von Neumann style? *Comm. ACM*, 21(8):613–641, 1978.

[3] M. M. Chakaravarty, R. Leschinskiy, S. Peyton-Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *Workshop on Declarative Aspects of Multicore Programming*, pages 10–18. ACM Press, January 2007.

[4] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, pages 7–18, 2003.

[5] A. Courtney, B. Robinson, and P. Hudak. wxfruit, March 2009. `http://www.haskell.org/haskellwiki/WxFruit`.

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

[7] C. Elliott and P. Hudak. Functional reactive animation. *International Conference on Functional Programming*, pages 263–273, 1997.

[8] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(8):98–107, 1989.

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

[10] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[11] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Working Papers 96-06-042, Santa Fe Institute, June 1996.

[12] S. Moss, H. Gaylard, S. Wallis, and B. Edmonds. Sdml: A multi-agent language for organizational modelling. *Computational and Mathematical Organization Theory*, 4:43–69, 1998. 10.1023/A:1009600530279.

[13] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive progamming, continued. *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 51–64, 2002.

[14] M. J. North, N. T. Collier, and J. R. Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16:1–25, January 2006.

[15] 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.

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

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

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

[19] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

[20] S. Peyton-Jones, R. Leshchinskiy, G. Keller, and M. M. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 1–32, 2008.

[21] H. Rahmandad and J. Sterman. Heterogeneity and network structure in the dynamics of contagion: Comparing agent-based and differential equation models. *Proceedings of the 2004 International Conference on System Dynamics*, 2004.

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

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

[24] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.