Design of the System Dynamics Longitudinal Analysis System: Quantifying the Hidden Trajectories of System Dynamics Models

Xiang Meng[1] and Nathaniel Osgood[2]

[1] Department of Electrical Engineering, Columbia University, New York, 10027, USA. meng@ee.columbia.edu
[2] Department of Computer Science, University of Saskatchewan, Saskatoon, S7N 5C9, Canada. osgood@cs.usask.ca

Abstract: The purpose of this document is to demonstrate a new set of algorithms which utilize dynamic programming techniques to solve two typical longitudinal statistics for aggregate population models. The paper first discussed some significant values of longitudinal statistics for system dynamic model calibrations and individual history-targeted interventions. It also addressed the current limitations of people who are facing difficulties in order to access the underlying dynamics of individual health patterns within populations. A detailed example of how to quantify the hidden trajectories for a basic SIR (non-linear) model has been demonstrated in this paper. The future work may involve in developing more algorithms to identify the trajectory statistics and implement one or more GUI-based software systems to permit application of algorithms.

Keyword: aggregate models, longitudinal statistics, dynamic programming and software engineering

1. Introduction

When faced with understanding the underlying dynamics of individual health patterns within populations, and the impact of health interventions on such populations, the value of individual-level longitudinal data has long been recognized. Such data – which provides a depiction of individual trajectories over time in the form of successive events– can offer great insight into the characteristic patterns of behavior and natural history of conditions in individuals. For example, such information can aid in identifying time constants associated with behavior change or declining health, and timing separating events such as successive infections, treatments and benefits. Therefore, developing the methods to perform the longitudinal data analysis will do a great help for improving the performance of the tools that are currently available in modeling calibrations, validations and policy design. This paper will introduce the new methods for analyzing individual-level longitudinal data for aggregate models. It outlines the motivations, objectives, along with a detailed description of how to perform and interpret the various algorithms. In addition, the paper will provide a brief

example showing how to quantify the hidden trajectories for a basic SIR model. The paper will conclude with a discussion of the most important tasks ahead, on the prospects of algorithms improvements and software implementations in the future, for more widespread applications.

2. Background & Motivations

Traditionally, health interventions and policy designs for system dynamics models are relied on cross-sectional data analysis. However, such analysis technique has the limited abilities to access the individual behaviors. At the population level, a longitudinal sample – composed of follow-up data on many individuals – can offer a far richer picture of the health of those individuals than is afforded by cross-sectional data. For example, for an infectious illness, researchers limited to using cross-sectional data could experience difficulties in distinguishing between situations in which an ongoing low level of prevalence of an infection reported at the population level is sustained in a small group of repeatedly infected individuals, or whether such infection is maintained instead in a succession of distinct individuals (with those individuals much larger in number). For instance, consider selecting prenatal Chlamydia screening programs in the following two different cases: (1) over the course of 5 years 1% of the population undergoes a suffering of 20 infections each; and (2) over the course of 5 years 20% of the population undergoes a suffering of 1 infection each. It became obvious to us that cross-sectional data analysis experiences difficulties in distinguishing between those two situations. This limitation could in turn translate into large uncertainties regarding to the tradeoffs between different interventions.

The value of longitudinal data in the health research arena is attested to by the heavy investment made by agencies in collecting such data. Despite very high costs associated with follow-up efforts (relative to those imposed by cross-sectional survey instruments), longitudinal datasets remain critical to health research.

For simulation modelers, the value of longitudinal is especially high. Because those building simulation models seek to capture the population dynamics, understanding the dynamics at an individual level can be important for building and calibrating an effective model – regardless of its level of aggregation. To offer face validity (and corresponding credibility with stakeholders), or to properly capture the impact of a given intervention, it will frequently be important that a model offers fidelity to the dynamics captured within the longitudinal datasets. For example, a modeler may judge it important that the model's depiction of the timing separating two types of events (or transitions), or the model's depiction of the fraction of individuals who experience a certain series of events (e.g. three separate relapses of illness prior to clearance) be consistent with those of the underlying data. The ability to validate a model such that its estimates agree with estimates from external data

can add significant consistency checks compared with what can be accomplished with purely cross-sectional data.

While longitudinal data is valuable for model validation and calibration, availability of such data offers additional opportunities on the policy design front. Because longitudinal data provides a modeler access to information on the distribution of individual histories, it opens the capacity for a modeler to prospectively evaluate the tradeoffs between history-informed policies. For example, in the health service delivery context, a modeler might investigate policies that differentially treat individuals based on past treatment history. In other contexts, individuals with particular longitudinal health profiles might be singled out as being at unusual risk, and represent candidates for targeted treatment. In some cases, this capacity to formulate policies not only based on the current health state classification of an individual, but on aspects of their history, can significantly enhance policy effectiveness.

3. Model Structure

While longitudinal data can offer great insight into system structure and policy tradeoffs, different model types offer different capacities to work with such information. Individual-based models (including agent- based models, or individual-level models created using traditional System Dynamics modeling packages) generally offer great flexibility in accumulating history information. For example, an agent-based model might maintain information on (i.e. a counter for) the number of times an individual has experienced re-infection, or maintain a detailed record of past treatments rendered (Axelrod, 1997). Such information could be used to tailor treatment strategies prospectively, or used in model calibration or validation via comparison with similar data from the population being modeled in the real-world system. However, in large-scale models with perhaps hundreds of state variables with thousands of individuals, the individual-based models approach shows significant limitations. There is a practical upper limit to the size of the parameter space that can be checked for robustness, and this process can be extremely computationally intensive, thus time consuming (Ferrer, 2008). Although computing power is increasing rapidly, the high computational requirement of individual-based models remains a limitation when modeling large systems. Furthermore, individual-based models can be more difficult to analyze, understand and communicate than traditional analytical models (Grimm, 1999), as it is difficult to provide detailed descriptions of the inner workings of such models.

By contrast, aggregate models traditionally offer limited ability to reason about longitudinal trajectories. For example, while aggregate models can readily capture simple discrete history information on model population members by disaggregating the population according to their history characteristic into different sets of stocks (e.g. often best captured via subscripting), this strategy scales poorly when multiple aspects of history must be

maintained. For example, while we may readily stratify a model according to a mother's dichotomous glycemic status during pregnancy (diabetic vs. non-diabetic), or by the count of her past bouts of gestational diabetes, the computational effort and space required to represent such models rises geometrically with the number of pieces of history information so maintained. It can also be highly expensive to maintain many discrete categories pieces (as might be desirable for approximating continuous history information, e.g. birthweight). Moreover, it is complex or infeasible to maintain arbitrarily large amounts of history information.

Other types of statistics summarizing history information can in principle be derived from aggregate models, but traditionally remain very difficult to access. For example, given a fixed mean residence time in an "aging chain" of model stocks, the time separating entry to one stock and to another can be readily calculated. However, this calculation grows more complicated in the presence of multiple outflows and (especially) dynamic per-individual transition rates. Other questions require more care even for relatively simple models, and involved calculation for complex models: During the operation of a model, what fraction (or count) of individuals traverse a given path of one or more (possibly repeated) stocks? During a specified period of time, what fraction (count) of individuals enter (exit) a stock through inflow (outflow) *X* vs. (outflow) inflow *Y*? These and similar questions have well-defined answers (for a given scenario), and knowing the answers would be valuable for validating a model and when designing policies.

4. Methods

In this contribution, we describe the design of a software system that will allow System Dynamics modelers to pose queries regarding statistics on model trajectories, such as those given above. The user can request longitudinal metrics to assess for a scenario, and the pieces of the model and time horizon over which those metrics are to be estimated.

In addition to providing retrospective use (using previously run scenarios), the system is being designed so as to allow a model in operation to change its behavior based on reported statistics accumulated from the run currently in operation. We anticipate the usefulness of such information for at least two purposes. Firstly, such information could be used during model calibration to evaluate closeness of fit between statistics on longitudinal progression within the model with those obtained from the external world. For example, during calibration, parameters sets could be favored according to how closely the times separating re-infection derived from the model matched those in the observed data.

Secondly, such information could be accessed during model operation in order to change model behavior. While it is not in general possible for an aggregate model to directly capture the effects of performing differential policies based on detailed information on an individual's trajectory (e.g. based on the number of times that individual has previously been infected), the capacity to access statistics on such information could be helpful for policy analysis. For example, a model that sought to investigate the impact of policies targeting individuals with certain history characteristics might compute what fraction of individuals share such characteristics. While the model would not directly reify the set of individuals with such characteristics (as is possible in an individual-based model), the information gained from the system presented here could be used to attempt to approximate the impact of differential treatment by applying a certain assumption concerning the effectiveness of the policy on those with that history characteristic, and on those without that characteristic.

According to the distributions of aggregate models on longitudinal statistics, we are able to identify a way to calculate the fraction of individuals with some history characteristic for a given model structure. Typically, longitudinal statistics are shaped by a large number of possible time-transition paths; calculating statistics by exhaustively enumerating all possible paths is typically infeasible. Intuitively, based on the memoryless character of stocks and the tremendous common substructure, we can compute shared structure only once and piece together a full computation out of these pieces.

4.1 Algorithms

In general, there are many techniques that can be used for calculating the longitudinal statistics, among those techniques, memorizations and dynamic programming could be the simple and time efficient solutions. In the following section, we are going to demonstrate a dynamic programming solution that can be used for finding the longitudinal statistics for nonlinear aggregate models, such as the average time or the fractions of people, who start in state A and reach state B over a certain period of time.

Assume there is a directed graph G which represents the aggregate model that has *n* states. The nodes of the graph are the state in the model, and each state has the private fileds named *fraction*. The edges of the graph represent the transition paths in the model from one state to the other, and each edge has the private field named *flowRate,* which stores the transition rate for such model. Here is a pseudocode version of the algorithm that computes the fraction of population, who starts in the state *Source* and ends in the state *Sink* at a given time step:

Algorithm 1:

FIND-FRACTION-HELPER (*G, Source, Sink*)
1 Let *Q* be the queue
2 *Q*.push(*Sink*)
3 **do** until the *Source.fraction* has been calculated

```
4        CurrentNode := Q.pop()
5        Q. push( all the nodes in G that pointed to CurrentNode )
6        for each node N in Q
7            N.fraction := CurrentNode.fraction * EdgeNtoCurrentNode.flowRate +
                           N.fraction *  (1 – Σ all outflow rates associated with N)
```

Similarly, if each state in *G* has the private fields named *b_factor* and *aveTime*, we can also develop the algorithm that computes the average time taken for the populations, who start in state *Source* and travel to the state *Sink* at a given time step:

Algorithm 2:

```
FIND-AVERAGE-TIME-HELPER (G, Source, Sink)
1        Let Q be the queue
2        Q.push( Sink )
3        do until the Source.aveTime has been calculated
4          CurrentNode := Q.pop()
5          Q. push( all the nodes in G that pointed to CurrentNode )
6          for each node N in Q
7            N.b_factor := N.b_factor * (1 – Σ all outflow rates associated with N)
                                      + Σ all outflow rates associated with N
                                      - EdgeNtoCurrentNode.flowRat
8            N.aveTime := {EdgeNtoCurrentNode.flowRate * (1 + CurrentState.aveTime)
                         + [(1 – Σ all outflow rates associated with N)  – N.b_factor]
                         * ( 1 + N.aveTime) }
                         / (EdgeNtoCurrentNode.flowRate + rateOfStay – N.b_factor)
```

In the main procedure *COMPUTSTAT()*, it will take the specified *startTime* and *endTime*. The detailed algorithm is shown below. Based on the step time *interval*, it will call the above algorithms *n* times, as indicated in line 1, where *nSteps* = (*finish time – start time*) / (*time interval for each step*). AnalysisMethod is a user defined method. Based on the users' specifications, the structure of the mode will be constructed as a type of AnalysisFrontier. For more detailed interface structures about AnalysisMethod and AnalysisFrontier, please refer to *Section 4.3*. Line 3 performs the initializations for the model, so that it is ready for the analysis. For each time step (line 4), the model will load the useful data from the external simulation tool, can calculate the result at the certain time step.

Main Algorithm:

```
COMPUTESTAT (Pair<Double, Double> Time, Pair<Node, Node> State,
                 AnalysisMethod specifiedMethod)
1    int nSteps = askVensim (Time)
2    AnalysisFrontier M = specifiedMethod.contructModel()
```

```
3   specifiedMethod.init(M, State)
4   for int t = (nSteps – 1) to 1
5           M.loadNewData(t +1)
6           specifiedMethod.calculate(M, State)
7   return specifiedMethod.result()
```

4.2 Visual Examples - SIR Model

In the following section, we will have a close look on how the algorithm works for a simple non-linear SIR model (Epstein, 2007), which contains three states, namely the susceptible (S) state, infected (I) state, and recovered (R) state. As a variant, the R state can also stand for "removed" state if people die from a disease. This simple model can represent various diseases. For example, in most cases, chicken pox fits the category of Susceptible – Infected – Recovered. Since the person who contracts this disease will become immune to future infections after recovery, Recovered state will be a terminated state in this model. HIV, on the other hand, fits the category of Susceptible – Infected – Removed, due to most people dying from contracting the disease. For some diseases, it is possible for the person who is in recovered state to get back to susceptible state again. The SIR model can be represented mathematically as a set of ordinary differential equations. If a constant population is under observation, the deterministic form of the SIR model is:

$$dS/dt=-c\beta S(I/(S+I+R))+\delta R$$

$$dI/dt=-\mu I+c\beta S(I/(S+I+R))$$

$$dR/dt=-\delta R+\mu I$$

Where:

- c is the average number of contacts per person unit time.
- β is the probability that any one such contact will transmit infection.
- μ is the rate of recovery/removal.
- δ is the rate of susceptible. With removed state or immune cases, $\delta = 0$.
- t is time, the unit of measurement for the rate of change of S, I, and R.

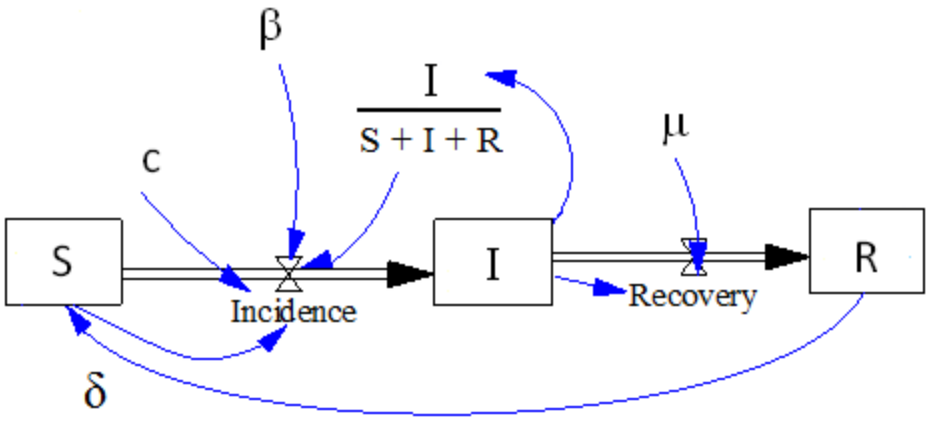


Figure 4.1 - *SIR* (non-linear)Model

Based on the model above, we can use a simulation tool such as Vensim to simulate the following situation: Consider a constant initial (t=1) population of size 10000 that is partitioned into 9000 susceptible, 1000 infected, and 0 recovered. The number of susceptibles per unit time to whom one such infective person will transmit infection when surrounded by susceptibles is $c\beta = 0.5$; the rate of recovery is $\mu = 0.8$; and the rate of waning immunity is $\delta = 0.2$. The algorithms must be built on the actual simulation run from external software. After a simulation performed for six unit times, we can obtain the following table of transition rates for each path at each time step. Based on those information, we will attempt to answer the following questions by using our algorithms.

Table 4.1: The Simulation Results of an SIR Model

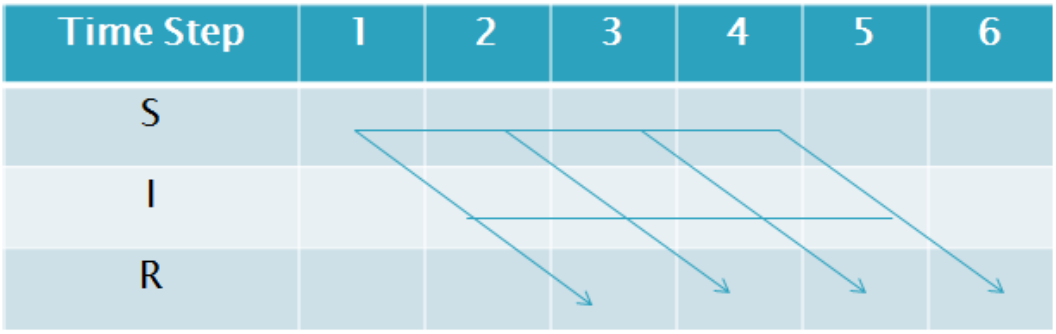
Time Step	1	2	3	4	5	6
S→I	0.0500	0.0325	0.0204	0.0127	0.0079	0.0050
I→R	0.8	0.8	0.8	0.8	0.8	0.8
R→S	0.2	0.2	0.2	0.2	0.2	0.2

Problem: What is the fraction for those people who start in state S at Time 1 and get to state R by no later than Time 6?

Assume there are *t* time steps, and *n* states. The time complexity for exhaustive analysis is a combination of *t* and *n*. The corresponding space complexity should be $O(nt)$. Obviously, solving this type of problems by exhaustively enumerating all possible paths is typically

infeasible. So we would like to propose a solution using our dynamic programming algorithms. The following figures demonstrated all the possible paths for exhaustive analysis.

Table 4.2 Paths for Exhaustive Analysis



Solution via Dynamic Programming: COMPUTESTAT() will start analyzing the model at *Time 6*. For those people who is in state R_6 , 100% of them will reach state *R* by $t = 6$; similarly for people in state of R_n where $n = 5,4,3,2,1$. In the meantime, the people who are in states I_6, S_6, can never reach the state of *R* by $t = 6$. Hence, 0% of those people will reach state *R*.

After we set up the conditions at ending *Time 6*, let us have a close look at how we find the fraction associated with each state at *Time 5* by applying algorithm 1. Before the algorithm start, it will load the model information at *Time 6* to construct *SIR Model*.

Time 5

FIND-FRACTION-HELPER (*SIR Model, S, R*)

1 Let *Q* be the queue
2 *Q*.push(R) // Q = {R}
3 **do** until the *S.fraction* has been calculated
4 *CurrentNode* = Q.pop() // *CurrentNode* = *R*
5 *Q*.push(I) // Q = {I}
6 **for** each node *N* in *Q*
7 *I.fraction* = *R.fraction* * *0.8* + I.*fraction* * (1 – 0.8)
= 100% * 0.8 + 0% * 0.8 = 80%
3 **do** until the *S.fraction* has been calculated
4 *CurrentNode* = Q.pop() // *CurrentNode* = *I*
5 *Q*.push(S) // Q = {S}
6 **for** each node *N* in *Q*
7 *S.fraction* = *I.fraction* * *0.005* + *S.fraction* * (1 – 0.005)
= 0% * 0.005 + 0% * 0.995 = 0%

Similarly, we will be able to find the fraction associated with each state at *Time 4, 3, 2* and *Time 1* respectively. The information in Table.2 should be available. As we can see from the table, there are 6.95% of the populations who start in state S at *Time 1* and get to state R by no later than *Time 6*. Assume there are t time steps, and n states. As we can see, the time complexity for the solution via dynamic programming is $O(nt)$. The corresponding space complexity has been reduced to $O(n)$.

Table 4.3: The Analysis Results from Time 6 to Time 1

Time Step	1	2	3	4	5	6
S.fraction (%)	6.95	3.83	1.84	0.63	0.00	0.00
I.fraction (%)	99.97	99.84	99.20	96.00	80.00	0.00
R.fraction (%)	100.00	100.00	100.00	100.00	100.00	100.00

4.3 Software Structures

As mentioned above, the aforementioned algorithms must operate on observations from actual simulation run from external software (Vensim). Thus, we need a set of tools, which use the Vensim Application Programming Interface to access model output for further calculations and analysis. In addition, a GUI-based wrapper for Vensim is also needed.

We have been conceptually designed the interface for *AnalysisMethod*, as indicated in Figure 4.2. There are two classes – *FindFraction* and *FindAverageTime*, which implements this interface. Each class is for solving one type of problems. The class can have subclasses to handle differentiate boundary cases.

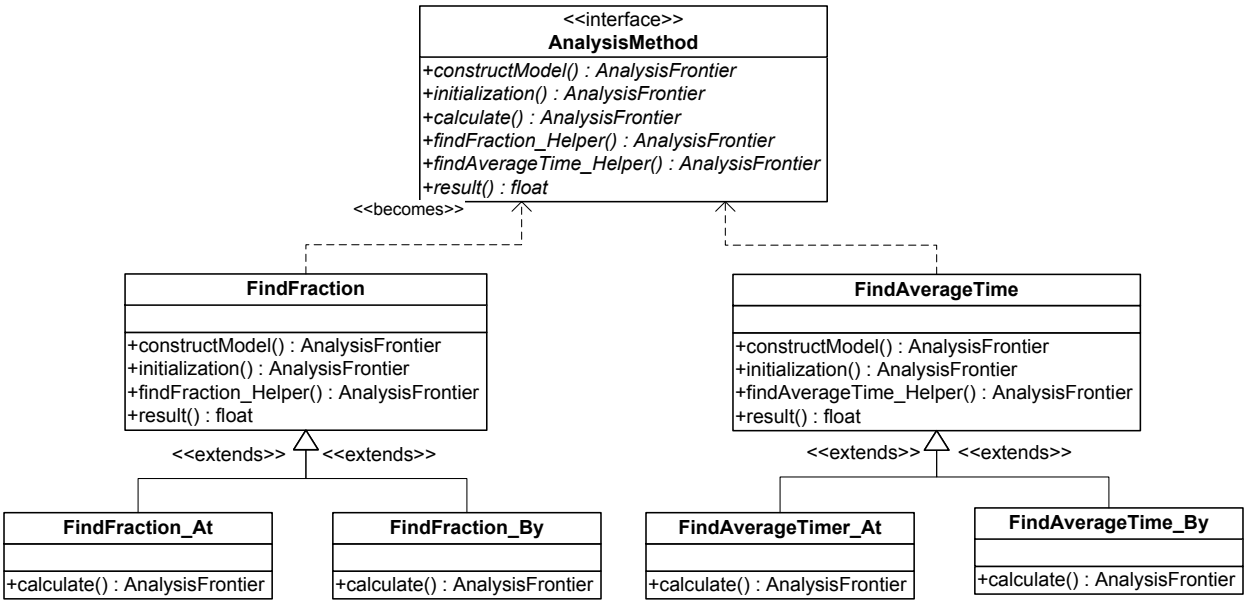


Figure 4.2 Interface Structure for methods of analysis

It is worth noting that the algorithms for different analysis methods require similar structures to represent to models. Thus we can develop a common structure as the *AnalysisFrontier*, which stores the information related to the analysis and their temporary results. In Figure 4.3, it shows a UML diagram of the interface for model representations.

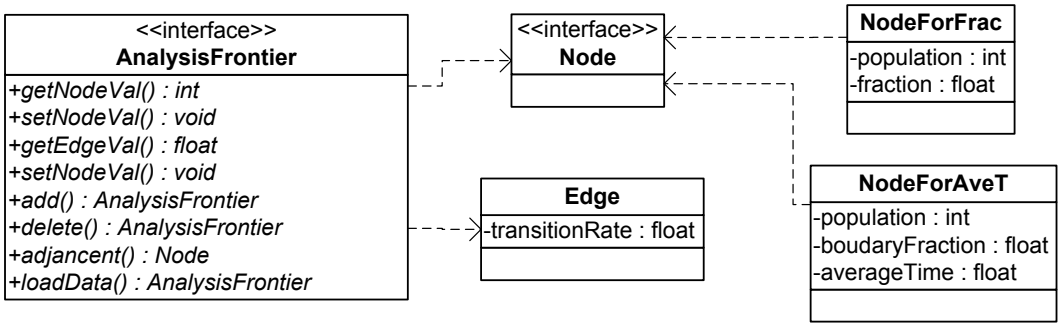


Figure 4.3 Interface for Model Structure

Future Work

Understanding how and why the algorithms work the way they do is crucial. Before one invests the effort of turning the algorithms described in this article into polished software

packages, it will require further mathematical understanding and development from at least the following perspectives:

(1) Additional methods for longitudinal statistics;
(2) How those different methods are related;
(3) Numerical computational efficiency and accuracy, such as pruning; and
(4) Extensive testing of the methods to an array of different models.

Only after these aspects have been further explored will the time come to translate the methods into the software application, by following the interface structures suggested in previous section. The system is designed to use the Vensim Application Programming Interface to access model output from a past or current Vensim scenario selected by a user. (This scenario must have a SAVEPER equal to the timestep of the model). Using this interface, the system derives data on the values of stocks and flows over the time horizon of user interest – data of key use during the system operation. A key intermediate quantity used is the fraction of individuals within a stock who flow out of each outflow in the course of a given timestep. A central challenge in the design of the system concerns the need to reason in a time-efficient fashion regarding statistics that are quantified over all possible specific trajectories by which an individual could pass. While the number of possible time-specific trajectories from one stock to another is very large, the algorithms can take critical advantage of the memoryless character of transitions (i.e. the fact that the likely of a given transition is independent of the amount of time spent in that transition) to reduce the amount of work involved in computing the statistics. Algorithms based on dynamic programming exploit the common substructure of multiple paths to compute required statistics in a rapid but exact fashion.

Conclusions

In summary, we have been developed a set of algorithms, which can calculate the longitudinal statistics for aggregate population model. The algorithms are able to satisfy the growing interest among system dynamics researchers, which can help them discover the underlying dynamics of individual health patterns within populations and examine the impact of health interventions on such populations. A polished GUI - software package will be developed to permit application of algorithms to Vensim models.

Appendix I - Interface

```
interface AnalysisMethod
{
        AnalysisFrontier init(AnalysisFrontier g. Pair<Node, Node> State);
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State);
        AnalysisFrontier FindFraction_Helper(AnalysisFrontier g, Pair<Node, Node> State);
        AnalysisFrontier FindAverageTime_Helper(AnalysisFrontier g, Pair<Node, Node> State);
        double result();
}

abstract class FindFraction implements AnalysisMethod
{
        public FindFractiont() {};
        AnalysisFrontier init(AnalysisFrontier g, Pair<Node, Node> State);
        AnalysisFrontier FindFraction_Helper(AnalysisFrontier g, Pair<Node, Node> State);
        double result();
}

class FindFraction_At extends FindFraction
{
        public FindFraction_At() {super()}
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State);
}

class FindFraction_By extends FindFraction
{
        public FindFraction_By() {super()};
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State);
}

abstract class FindAverageTime implements AnalysisMethod
{
        public FindAverageTime() {};
        AnalysisFrontier init(AnalysisFrontier g, Pair<Node, Node> State);
        AnalysisFrontier FindAverageTime_Helper(AnalysisFrontier g, Pair<Node, Node> State);
        double result();
}

class FindAverageTime_At extends FindAverageTime
{
        public FindAverageTime_At() {super()};
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State);
}
```

```
class FindAverageTime_By extends FindAverageTime
{
        public FindAverageTime_By() (super());
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State);

}
```

Appendix II – Algorithms (Draft)

```
abstract class FindFraction implements AnalysisMethod
{
        public FindFraction () {};
        AnalysisFrontier init(AnalysisFrontier g, Pair<Node, Node> State)
        {
                // Each node in graph should have a value of fraction
                assign the fraction for the Sink to be 1
                assign the fraction for other node to be 0

        }
        AnalysisFrontier FindFraction_Helper (AnalysisFrontier g, Pair<Node, Node> State)
        {
                Temporary duplicate graph g, as we need old values to calculate new ones
                 //Q is the queue
                Q := push(Sink)
                do until Q is empty
                   CurrentNode := Q.pop()
                   Push all other nodes pointed at CurrentNode to Q
                   for each node N in Q
                      N.fraction = CurrentNode_oldFrac * outFlowrateToCurrentNode +
                                   N.fraction *  rateOfStay
                   endfor
                endwhile

        }
        double result()
        {
                return Source.fraction;
        }

}

class FindFraction_At extends FindFraction
```

```
{
	public FindFraction_At() {super()};
	AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State)
	{
		AnalysisFrontier newFrontier = super. FindFraction_Helper (AnalysisFrontier g,
Pair<Node, Node> State);
		override the new fraction value for the Sink to be 1
	}
}

class FindFraction_By extends FindFraction
{
	public FindFraction_By() {super()};
	AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State)
	{
		AnalysisFrontier newFrontier = super. FindFraction_Helper (AnalysisFrontier g,
Pair<Node, Node> State);
	}
}

abstract class FindAverageTime implements AnalysisMethod
{
	public FindAverageTime()
	AnalysisFrontier init(AnalysisFrontier g, Pair<Node, Node> State)
	{
		//Each node in graph should have for averageTime and boundaryFactor
		assign the averageTime for all the node to be 0
		assign the boundaryFactor for all the node to be 1
	}
	AnalysisFrontier FindAverageTime_Helper (AnalysisFrontier g, Pair<Node, Node> State)
	{
		Temporary duplicate graph g, as we need old values to calculate new ones
		 //Q is the queue
		Q := push(Sink)
		do until Q is empty
		   CurrentNode := Q.pop()
		   Push all other nodes pointed at CurrentNode to Q
		   for each node N in Q
		      // BF – BoundaryFactor; AT - AverageTime
		      N.b_factor = N.b_factor *  rateOfStay +
		                        Σ all outFlowRate excluded the one to CurrentNode
		      N.averageTime = ( outFlowRateToCurrentNode * ( 1 +
		   CurrentState.averageTime ) +
```

```
                                    ( rateOfStay – N.b_factor ) * ( 1 + N.averageTime ) ) / (
                                    outFlowRateToCurrentNode + rateOfStay – N.b_factor)
                    endfor
                endwhile

        }
        double result()
        {
                return Source.averageTime;
        }

}

class FindAverageTime_At extends FindAverageTime
{
        public FindAverageTime_At() {super()};
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State)
        {
                // More thoughts should be put on this type of question
                AnalysisFrontier newFrontier = FindAverageTime_Helper
                                                (AnalysisFrontier g, Pair<Node, Node> State);
                override the new average time for the Sink to be ( Sink.averageTime + 1 )
        }

}

class FindAverageTime_By extends FindAverageTime
{
        public FindAverageTime_By() {super()};
        AnalysisFrontier calculate(AnalysisFrontier g, Pair<Node, Node> State)
        {
                AnalysisFrontier newFrontier = FindAverageTime_Helper
                                                (AnalysisFrontier g, Pair<Node, Node> State);
        }

}
```

References

Axelrod, R., The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration, Princeton University Press, 206-222 (1997)

Ferrer, J., Prats, C., and López, D.: Individual-based Modeling: An Essential Tool for Microbiology. J Biol Phys 34:19–37 (2008)

Grimm, V.: Ten years of individual-based modeling in ecology: what have we learned and what could we learn in the future? Ecol. Model. 115(2–3), 129–148 (1999)

Epstein, J.M., Generative Social Science: Studies in Agent-Based Computational Modeling, Princeton University Press, 271-306 (2007)