Aspects Enhance the Flexibility and Modularity of Simulation Models

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By

Priyasree Bhowmik

©Priyasree Bhowmik, Apr 2016. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

While the popularity of simulation models as a tool to address complex problems has increased in recent years, issues of flexibility and modularity associated with simulation models are yet not well explored. These two issues emerge from software engineering challenges arising from implementation and management of model execution, maintenance of metadata corresponding to scenario results, inter-dependency of modelers and end-users to modify model output for exploring patterns of interest, a frequent need to debug and the occasional unavailability of sufficient data to offer effective estimates for model parameters. These challenges have often led simulation modelers to adopt to various mechanisms like manual documentation, tracing, calibration etc., but not to much success due to the other limitations associated with each of these processes. We present here techniques to enhance flexibility, modularity, usefulness and effectiveness of simulation modeling by using Aspect Oriented Programming. The core concepts of Aspect Oriented Programming have been utilized to implement two aspect-based frameworks first, a logging and tracing tool for capturing the high-level execution results and, separately, low-level details associated with model executions, and second, a MCMC tool for estimating model parameters by sampling from their joint posterior distributions using a rigorous statistical approach formed by combining Bayesian Markov Chain Monte Carlo (MCMC) methods with dynamic models. We describe here both the frameworks, including their implementations and functioning, experiments conducted, and results obtained.

// Acknowledgements

I would like to take this opportunity to extend my heartfelt gratitude to several people who contributed to the completion of this work.

- I extend special thanks to my supervisors Dr. Nathaniel Osgood and Dr. Christopher Dutchyn - who encouraged, guided and supported me from the beginning to the end in order to help me develop a deep understanding of the subject of my thesis and proceed systematically with my research in the right direction. I greatly appreciate all the time, effort and energy my supervisors bestowed upon me that improved my reading, writing skills and analytical skills.
- I would like to thank my committee members - Dr. James Nolan, Dr. Ian Stavness, Dr. Kevin Stanley along with Dr. Christopher Dutchyn and Dr. Nathaniel Osgood for their valuable time, suggestions and insights that helped enrich my thesis.
- I would also like to thank my several course instructors - Dr. Kevin Stanley, Dr. Chanchal Roy, Dr. Christopher Dutchyn and Dr. Nathaniel Osgood who made it easier for me to understand important concepts related to my research.
- I wish to convey my regards to my lab mates - Winchell Qian, Kurt Kreuger, Narjes Shojaati, Rahim Oraji, Allen McLean and fellow researchers, friends who helped me with their ideas and feedback and and supported me while evolving as a researcher. I would also take this opportunity to thank the computer science department stuffs - Gwen Lancaster, Heather Webb, Linda Gesy and Shakiba Jalal for the help and supported they bestowed on me in their capacity in various ways during the completion of my thesis.
- I wish to express my love and gratitude to my beloved husband Dr. Koushik Pal who has been with me in all ups and downs, constantly inspiring me with his valuable advice, suggestions, encouragement and support.
- Finally, I would take this opportunity to express my gratitude to my beloved parents Mr. Prashanta Kumar Bhowmik and Mrs. Purnima Bhowmik without whom I would have never reached here. I would also like to express my gratitude to my parents-in-law Dr. Bata Krishna Pal and Mrs. Chhaya Pal for their continuous encouragement and support.

I dedicate this thesis to my parents Mr. Prashanta Kumar Bhowmik and Mrs. Purnima Bhowmik whose inspiration helped me accomplish everything in my life.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables vii

List of Figures viii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Problem Statement 4
- 1.3 Thesis Statement 6
- 1.4 Thesis Contributions 6
- 1.5 Related Publications 7
- 1.6 Thesis Organization 8

2 Background 9
- 2.1 Java Reflection API 10
- 2.2 Limitations of OOP overcome by AOP 10
  - 2.2.1 Crosscutting Concerns 12
  - 2.2.2 Aspects' role in eliminating crosscutting concerns 13
- 2.3 AspectJ 13
  - 2.3.1 Join Points 13
  - 2.3.2 Pointcuts 14
  - 2.3.3 Advice 15
  - 2.3.4 Inter-type declarations 16
  - 2.3.5 Aspects 16
- 2.4 Simulation Modeling and AnyLogic 17
  - 2.4.1 Simulation Model 17
  - 2.4.2 AnyLogic 18
  - 2.4.3 Agent-based Modeling 18
  - 2.4.4 System Dynamics Modeling 19
  - 2.4.5 Discrete Event Modeling 19
- 2.5 How a Simulation Model Works 20
  - 2.5.1 Logging 21
  - 2.5.2 Tracing 21
- 2.6 Markov Chain, Monte Carlo and MCMC 23
  - 2.6.1 Metropolis Hastings Algorithm 24
- 2.7 Prior, Likelihood and Posterior 25
  - 2.7.1 Prior 25
  - 2.7.2 Likelihood 25
  - 2.7.3 Posterior 25

3 The Logging and Tracing Tool 27
- 3.1 Logging and Tracing Tool Design 28
- 3.2 Logging and Tracing Tool Implementation 29

3.2.1 Implementation Overview . . . 29
3.2.2 Logging and Tracing Tool Framework . . . 30
3.3 Output from Logging and Tracing Tool . . . 39
3.3.1 Run Log . . . 39
3.3.2 Trace Log . . . 41
3.3.3 Output Canvas . . . 43
3.3.4 Copy of the Model . . . 43
3.4 Logging and Tracing Tool Configuration . . . 44
3.5 Logging and Tracing Tool Functionality and Service . . . 45

4 Experiments and Evaluations related to the Logging and Tracing Tool 46
4.1 Experiment with the MinimalistSIRNetworkABM Model . . . 46
4.2 Experiment with the Post-Partum Depression Model . . . 47
4.3 Experimental Observations . . . 48
4.3.1 Role of run log, trace log and output canvas as future reference guide . . . 49
4.3.2 Role of the trace log for clear understanding and debugging . . . 49
4.3.3 Role of the model copy and run log and output canvas for recreation of output . . . 50
4.3.4 Logging and Tracing Tool Modularity Assessment . . . 51

5 The MCMC Tool 55
5.1 MCMC Tool Design . . . 55
5.2 MCMC Tool Implementation . . . 57
5.2.1 Implementation Overview . . . 57
5.2.2 GUI for User Input . . . 58
5.2.3 MCMC Tool Model Requirements . . . 59
5.2.4 MCMC Tool Framework . . . 59
5.3 Output from the MCMC Tool . . . 64
5.3.1 MCMC Results . . . 64
5.4 MCMC Tool Configuration . . . 66
5.5 MCMC Tool Functionality and Service . . . 67

6 Experiments and Evaluations related to the MCMC Tool 68
6.1 Experimental Set up . . . 68
6.1.1 Analyzing the SEIR Model . . . 68
6.1.2 Handling the attributes of the Random Walk Metropolis Hastings . . . 71
6.1.3 Overview of the empirical data . . . 72
6.2 Experimental Observation and Analysis . . . 73
6.2.1 Statistical Analysis . . . 73
6.3 Advantages of the MCMC Tool . . . 75
6.3.1 MCMC Tool Modularity Assessment . . . 75
6.3.2 MCMC Tool Simplification Assessment . . . 78
6.3.3 MCMC Tool Correctness Assessment . . . 80
6.3.4 Learning . . . 81

7 Conclusion 82
7.1 Summary . . . 82
7.2 Contributions from this research . . . 83
7.3 Limitation of our tools . . . 83
7.3.1 Limitations of the logging and tracing tool . . . 84
7.3.2 Limitation of the MCMC tool . . . 84
7.3.3 Future Work . . . 85

References 86

List of Tables

1.1 Features provided by Simulation Frameworks) 4

2.1 Description of Join Points 16
2.2 Description of Pointcuts 17
2.3 Description of Advice 18

3.1 Mapping of Items in the Trace Log with the information it provides 42

4.1 List of experiments conducted [9] 47

6.1 Quantiles for parameters p & c (Here Par stands for Parameter and Emp for Empirical Value) 77
6.2 Mean, Standard Deviation and Credibility Intervals for parameters p & c (Here Par stands for Parameter, Emp for Empirical Value and Stdev for Standard Deviation) 77

List of Figures

2.1 Issues with OOP 12
2.2 AOP prevailing over OOP 13
2.3 Example of crosscutting concern 14
2.4 Resolution of crosscutting concerns via aspects. 15
2.5 Example of a named pointcut 15
2.6 Example of a unnamed pointcut 15
2.7 Random Walk Metropolis Hastings (single-walker version) 25

3.1 Implementation Overview of the Logging and Tracing Tool 29
3.2 Enable/Disable the Logging and Tracing Tool 30
3.3 The Run Log 39
3.4 The Trace Log 41
3.5 Initial and Final state of the Output Canvas. 43
3.6 Copy of the Model 43
3.7 The Run Log and the Trace Log. 44

4.1 The recreation of output using the logging and tracing tool 50
4.2 Left: Simulation model with manual methods of logging and tracing ; Right: Simulation model with the Logging and Tracing Tool 51
4.3 Coding required for implementing logging and tracing without aspects 53

5.1 Implementation Overview of the Aspect-based MCMC tool 57
5.2 User Interface of the Aspect-based MCMC Tool 58
5.3 Enable/Disable the MCMC Tool 60
5.4 Plots generated by the MCMC Tool 65
5.5 MCMC simulation data set generated by the MCMC Tool 66
5.6 MCMC output data set generated by the MCMC Tool 67

6.1 Left: SEIR model in AnyLogic ; Right:Pictorial depiction of simplified SEIR Model significant for our project 69
6.2 Trace Plot of of parameter p and c when the actual values are 0.8 and 350, respectively. . . . 74
6.3 Trace Plot of of parameter p and c when the actual values are 0.7 and 300, respectively. . . . 74
6.4 Joint distribution of parameters p and c when the actual values are 0.8 and 350, respectively. 75
6.5 Joint distribution of parameters p and c when the actual values are 0.7 and 300, respectively. 75
6.6 Marginal Density plots of parameters p and c when the actual values are 0.8 and 350, respectively 76
6.7 Marginal Density plots of parameters p and c when the actual values are 0.7 and 300, respectively 76
6.8 Left: Simulation model with code written manually to incorporate the MCMC functionality; Right: Simulation model with the MCMC Tool 77
6.9 Coding required for incorporate the MCMC functionality without aspects 79

7.1 Model View Controller Architecture for decoupling view from model 85

Chapter 1

Introduction

1.1 Motivation

Simulation modeling is becoming increasingly popular in academia and industry for various research, policy making or business purposes. The upward rise in its user base is paralleled by a rise in public expectation of simulation modeling frameworks in terms of flexibility, usability and correctness. People are seeking – or need – efficient (efficiency discussed in this thesis is with respect to ease of use and not computational efficiency) ways to ensure reproducibility, generate metadata, store simulation output, and minimize the effort involved with execution of simulations. To date, there are quite a few popular and efficient simulation modeling frameworks that provide both *modelers* and *users* (one who develops the simulation model is referred to as a modeler whereas the one who simply uses it for investigation is referred to as a user) a platform to simulate, analyze and study complex real-life situations or problems. However, these platforms are hampered by the absence of developed functionality of automated documentation, tracking and information-storage [9].

The primary goal of simulation modeling is to explore outcomes to scenarios by simulating them. Often modelers need to execute a series of experiments with a varied set of inputs to explore different test case scenarios. Maintaining associated *metadata* (associated information that describes the generated output and how it was produced) is absolutely crucial to ensure easy reproduction of simulated results in the future – reproduction which is important for the scientific process. Such information is also important to lower the risk that the same scenario will be needlessly re-simulated. The model version, parameter assumptions, and random number seed are some of the metadata that must be preserved for future reference and for guaranteeing easy re-creation. Without an appropriate documentation process in place, such preservation is unlikely. While current simulation modeling practice places great emphasis on convention and manual re-execution of simulation models, such processes consume a fair amount of time and attention. Thus, even in the presence of a well-fitting manual process, preservation of all required information becomes solely dependent on the modelers and their schedule; busy schedules associated with stringent deadlines often lead to incomplete manual recording of metadata. Furthermore, certain information – such as the random number seed auto-generated by the model in a particular simulation run – is not easy to obtain manually. Hence modelers are often forced to run the same experiments over and over again as and when information from the same scenarios are required for some study or analysis, leading to unnecessary waste of time and resources.

An automated documentation process taking care of these vital pieces of information is a potent way to avoid these problems.

While it is valuable for a modeler or user to have the option of continuously observing the entire execution of a simulation model, it is relatively rare that this option is exercised. A lot of understanding is derived from insights models provide while they run. In order to gain the most from simulation models, it is important to monitor model behaviour through the course of execution (*run-time*). Although modelers are usually the ones who observe the execution from start to finish, they are often not the knowledge-users, or other stakeholders or sponsors in many cases. Knowledge-users are those individuals who need to have a clear understanding of all the basics of a simulation model in order to study, analyze or use the model for actual decision-making. But since these individuals sometimes do not execute the model themselves, mostly all they can do is check or examine the final state of the model, in contrast to verifying the action of several factors within the model over a period of time. The limited domain knowledge or interpretation of modelers limits their insightfulness of the *renderings* (visualization of the information from the model) that they expose to end-users. The resulting dependence of knowledge-users on modelers often further obscures the insights available to the knowledge-user. In addition, the frequent need of knowledge-users to convey requests and queries to the modeler creates a bottleneck in the process around the modeler, and further delays the learning process. Moreover, some simulation models take a long time to execute and as such observing the entire execution may be tiring, or even infeasible. Tracing certain kinds of events or changes taking place in the model can be of value and at times proves to be beneficial both to the users and the modelers. There are times when each of knowledge users and modelers need to understand the sequential flow of certain types of state transitions. This is of utmost use when an unexpected behaviour is noticed, since it simplifies the activities of debugging and analysis.

The traditional way of adding the tracking functionality to a simulation model is via manual logging, such as with calls to print or logging functionality in language libraries. Some demerits of this process are as follows:

- Tracing in order to diagnose a particular activity essentially involves writing additional lines of code. This also involves removing these lines of code in future when the requirement no longer exists.
- Manual tracing is a time consuming activity, especially given that it is recurring in nature, since it involves both insertion of code and its removal when tracing is not required at a later time.
- When the traced output is generated in the console, this process leads to unnecessary cluttering of that console.
- Often manual tracing gives rise to undesirable inter-connections among code segments. While the tracing functionality is logically related to only one type of functionality or concern of tracking, the code to introduce the tracing functionality will often span across several modules in the model code. For instance — when there is a requirement to trace any change of state of an agent, all code segments

concerning each of the states are impacted because the same print statements have to be introduced into each state, thereby inadvertently creating a coupling effect. The same is true for logging related to a given message (both sending and receipt, potentially by different agents) as this typically requires code at several locations.

- This process lacks modularity, as a result of which any modification in the tracing functionality such as outputting the traces to a file instead of the console usually involves changes across several modules of the model.
- Each change involving any of these affected code segments across different modules in the code-base potentially requires updating the tracing code.
- The correctness of the process depends on the memory of the modeler since whenever new modules are added to the code-base, the modeler needs to remember to add the tracing functionality to the new module. Along similar lines, when such logging has to take place across multiple locations of significance in the code, it can be quite easy to forget to add in such annotations even when wanting to add what is conceptually a single logical event. As a result, logging messages can be incomplete.

Developing a simulation model is often a challenging task. One common contributing factor to this challenge is that typically modeling teams have limited access to data regarding the particular context being simulated by the model. This is true, for instance, in cases where a modeler needs to develop a model about the spread of a disease, newly introduced into a community by immigrants. The modeler in this case has to take note of the new combination of the community, their behaviour along with the source and communication factors of the disease. In cases like this, empirical data available to the modeler often is inadequate to offer confidence in model parameter estimation [35]. Hence, correctly quantifying individual parameters affecting the model can be quite demanding.

Modelers, therefore, need to rely upon some technique or tool to estimate parameters for which direct evidence is lacking. One important and extensively used approach is *calibration*. Calibration mechanisms are used for estimating model parameters based on the *emergent behavior* of the system. Here parameters are adjusted such that the emergent behavior of the model closely matches the empirical emergent behavior of the system in reality. Emergent behavior of a system does not depend on the individual elements of the system, but rather it is based on their interactions with each other. Thus emergent behavior refers to the global behaviour of the system caused by the local relationships between its individuals. Unfortunately, calibration mechanisms are accompanied by several drawbacks, as mentioned below [35]:

- Calibration tools often eliminate or provide ineffectual information regarding other uncertainties or factors involved.
- When such uncertainty bounds are quantified, calibration tools tend to focus on local bounds and fail to capture the broader characteristics associated with the parameter distribution, thereby distorting the

resulting insight for systems dealing with multi-modal distributions (e.g., cases in which the observed data suggest multiple interpretations).

- When calibration mechanisms yield parameter uncertainty estimates for non-linear models, such estimates cannot be translated into estimates for model outcomes.

- Calibration mechanisms cannot help resolve uncertainty involving the choice of model.

1.2 Problem Statement

Simulation modeling is an incremental process. While developing complex models, ongoing re-examination of results for the same scenarios is frequently required. Modelers therefore need to generate metadata, or meticulously preserve information generated from model execution. At times, they also need to recreate the exact same output created at an earlier point of time. When results are not as per expectation, they also need to track exactly what is going on in the model. These requirements further clarify the importance of generating summary level details required to recreate a simulation run – or *Logging* – and tracking the intrinsic details of the simulation or– *Tracing* – as a proper logging and tracing mechanism is capable of resolving most of these issues. Logging and tracing is supported to some extent by a few simulation frameworks. Table 1.1 provides a comparison of the features in question, of some popular simulation modeling frameworks. Here we have considered five simulation frameworks to find out whether the framework prints the seed randomly generated by the model to ensure reproducibility, whether it provides logging and tracing to the extent required for reproducibility and debugging and if it contains a scripting environment.

Frameworks	Allows Inbuilt mechanism to print the seed randomly generated	Logging and Tracing	Scripting
AnyLogic	No	Limited, newly added	Yes
Netlogo	No	Limited	No
Vensim	No	Yes	Yes
Simio	No	Limited	No
Flexsim	No	Limited	Yes

Table 1.1: Features provided by Simulation Frameworks)

Frequently, deficiency of empirical data emerges as an challenge in model building, predominantly when enough information about different factors of the system being simulated is unavailable. This compels modelers to seek a technique that can help estimate parameter values in light of the limited amount of empirical data available. Outside of dynamic modeling, Bayesian *Markov Chain Monte Carlo (MCMC)* method is a popular technique that offers a promising combination with dynamic modeling [35].

In this thesis we have focused on system simulation although similar requirements are present in cognate areas of simulation. Simulation models are profusely affected by crosscutting concerns [11] (discussed in section 2.2.1). Almost all models deal with visualizations and presentations. These features are responsible for a bulk of crosscutting concerns in the simulation models. Aspect Oriented Programming (AOP) Paradigm [22, 26] is a way to address crosscutting concerns. The utilization of Aspect Oriented Programming (AOP) Paradigm in simulation modeling has the potential to improve modularity, reduce complexity and eliminate redundancy [11], as such it is considered as a promising research area. Aspects can help with epiphenomenal components like displays, presentations, visualizations, security, tracing, logging, sampling etc. AOP has been applied across several systems to provide a modular implementation of crosscutting concerns. A few of them are mentioned below:

- Applying aspect-oriented programming to database systems for designing tailorable and maintainable databases [44].
- Combining aspect-oriented programming with Feature-oriented analysis to enhance reusability, adaptability, and configurability of product line assets [27].
- Applying aspect-oriented programming to modularize specific features of the JUnit framework [24].
- Using aspect-oriented programming for the testing of embedded software in the context of an industrial application [28].
- Using aspect-oriented programming to benefit trustworthy software development [41].
- Using aspect-oriented programming to address modularity-issues in compilers [37].
- Aspect-Oriented Programming in Spring Framework [5]. The Spring Framework is an application framework that allows developing complex enterprise applications. It uses AOP as one of the fundamental technologies. Although using AOP is optional, AOP complements Spring to provide a very capable middle ware solution.
- Applying aspect-oriented programming to security to separate the concerns [45].

Our research is focused on a few predominant challenges prevailing among the modeling community that aspects can resolve. We aim at addressing issues of automated documentation (ensuring reproducibility), storage, tracking and sampling from parameter distributions while improving the modularity and the flexibility of simulation modeling techniques. The goal is to establish how crosscutting concerns associated with simulation modeling can be encapsulated in a modular manner, thereby easing common hurdles (such as modifying the model every time a new module is added) associated with modeling and improving the modularity of simulation modeling techniques. Another goal is to improve flexibility provided to modelers by the simulation modeling environment such that solutions related to the issues of automated documentation,

storage, tracking and sampling are available as and when the requirement arises, or in other words, can be enabled and disabled easily.

As a proof of concept of the benefits of applying aspects to simulation modeling, we enhance the functionality and usefulness of a simulation modeling framework by introducing aspects to:

- Automatically document and store essential information and results from simulation runs, required for re-creation, analysis and re-use; and
- Automatically estimate model parameters using a theoretically grounded approach.

These approaches of introducing logging, tracing and the use of MCMC in simulation models will help reduce the cognitive burden of understanding hidden processes in a model, which in turn, clarifies the insight gained from these models and subsequently helps to analyze and find errors, if any prevail in the model. It also enhances simulation modeling techniques by supporting an automatic documentation process that helps improve reproducibility by enabling the recreation of the exact same output for most simulations. It also enriches the simulation modeling framework by introducing an easy-to-use mechanism to address the issue of parameter estimation.

1.3 Thesis Statement

Introducing aspects into simulation modeling can help implement and simplify the representation of Logging, Tracing and MCMC algorithm, thus paving the road for modular, flexible, interactive and robust simulation modeling techniques.

1.4 Thesis Contributions

In this research, we have worked towards establishing the hypothesis that a popular Java based simulation modeling framework can be extended to meet three common challenges faced by modelers, with the help of the Aspect Oriented Programming (AOP) Paradigm [22, 26]. Our project deals with the application of *aspects*, which are basic structures or constructs of aspect oriented programming. When tracking a repetitive event or activity in the simulation model is needed, there tends to be undesirable inter-connections between different segments of the code-base. An aspect-based tracing mechanism is an efficient solution to the problem of unwanted coupling between code segments associated with traditional tracking methods. Aspects extract the tracking code (in this case) from the core business logic and rearrange the same code in a single unit or code location. Furthermore, aspects are implemented such that other areas of the code matching a specific pattern can automatically make use of it; as such, when new code matching the same pattern is added to the code-base, that new code can automatically make use of the existing aspect-based functionality (tracing mechanisms in this case).

To evaluate our hypothesis, we have enhanced the leading Java based simulation modeling framework *AnyLogic* with the help of *AspectJ*, a popular, well-integrated, compatible, aspect oriented extension of the popular object oriented programming (OOP) language Java [21, 16]. Specifically, we developed a *Logging and Tracing tool*. To cope with the issue of undesired coupling, this tool uses a set of aspects to implement the new tracing functionality so that the tracing logic can be independently placed in a small region of the code-base, thereby preventing unwanted intertwining of application logic with the logic of tracing. AspectJ has the capacity to open up closed-source applications via load-time weaving that allows injection of aspect instructions to the byte code during class loading. Thus aspectJ can be used with AnyLogic models whose source code is not readily available. Even when the source code is available, the functionalities such as logging can be introduced by manual adhoc method which lack modularity and flexibility.

This project is also an effort to address the challenges faced by modelers due to insufficient data. We demonstrate here an approach by which dynamic simulation models can be easily complemented with a statistical computation technique known as the Bayesian Markov Chain Monte Carlo (MCMC) method. In effect, MCMC samples from joint posterior distributions (discussed in 2.7.3) associated with model parameters. It also samples from model states, outcomes, and intervention results over time. MCMC approaches are based around a well defined but generic probabilistic model and thus it can be reused for newly arrived data points. Two of the most common MCMC mechanisms are Metropolis Hastings and Gibbs sampling, Gibbs sampling is a special case of Metropolis-Hastings [10].

As a demonstration of this potential, we present a user-friendly implementation of a particular MCMC algorithm (the random-walk Metropolis Hastings) to address this limitation. In particular, we designed an aspect based framework supporting MCMC to automate sampling from the joint distribution over specified parameters in dynamic models built in AnyLogic.

These tools can be configured easily in the AnyLogic framework and can be used with simulation models to automate documentation, tracing and parameter estimation via the MCMC functionality. As we shall see, users have the choice to use the tool or skip it by including or excluding a java machine argument.

1.5 Related Publications

The portion of this thesis about the Logging and Tracing tool has been published. The portion of this thesis developing the MCMC tool is in press. In each of these publications, I am the primary author and the associated studies and experiments were conducted under the supervision of Dr. Nathaniel Osgood and Dr. Christopher Dutchyn. The list of publications (two conference papers) is given below.

- Priyasree Bhowmik, Nathaniel Osgood, and Christopher Dutchyn. Improving the flexibility of simulation modeling with aspects. In *Proc. Symposium on Theory of Modeling and Simulation (TMS/DEVS 2015)*, Alexandria, Virginia, April 2015. (Published)

- Priyasree Bhowmik, Christopher Dutchyn, and Nathaniel Osgood. An aspect oriented framework to ap-

plying markov chain monte carlo methods with dynamic models. In *Proc. of the Symposium on Theory of Modeling and Simulation (TMS/DEVS 2016)*, Pasadena, California, April 2016. (Forthcoming)

1.6 Thesis Organization

The organization of this thesis is as follows:

- Chapter 2 introduces the background material relevant to the thesis. We discuss different concepts that form the backbone of this thesis, including — but not limited to — Aspect oriented programming, simulation modeling, logging, tracing and MCMC methods. We also discuss other literature and research related to this area.
- In Chapter 3, we discuss the Logging and Tracing tool in depth, including the concepts of functionality, implementation and configuration.
- Chapter 4 discusses the Experiments and subsequent Result Evaluations related to the Logging and Tracing tool.
- In Chapter 5, we discuss the MCMC tool in depth. As above, this includes a discussion of its functionality, implementation and configuration.
- In Chapter 6, we discuss Experiments and subsequent Result Evaluations related to the MCMC tool.
- In Chapter 7, we summarize the thesis and provide a vision for future work.

Chapter 2

Background

This thesis focuses on the various challenges faced by dynamic modelers while developing and maintaining a simulation model, and also how the aspect oriented programming paradigm can be leveraged to address such limitations. The challenges referred to here emerge from the fact that most of the widely used simulation modeling frameworks lack potent logging and tracing mechanisms. One such extensively used and versatile simulation modeling software is AnyLogic, which is capable of supporting a broad set of modeling tasks while allowing the development of complicated models. However, there remain some limitations that curtail its versatility. One of these missing pieces of functionality is the capacity to maintain execution logs for future reference [34]. Another is the lack of an efficient way to document run details. The package also lacks a simpler way of debugging complicated models [36] and an effective way of coping with situations in which there is a deficiency of empirical data. The traditional mechanism for tracing provided by AnyLogic consists of inserting new lines of code within the model, using methods such *trace* or *traceln* [17]. These are some of the critical issues that have been targeted in this research. Another vital issue addressed here is a general way of parameter estimation for unknown parameters in a dynamic simulation model. When complex systems need to be modeled, frequently the modeler faces challenges posed by the deficiency of data to quantify all factors involved. In order to understand clearly the work done in this research, it is useful to characterize relevant concepts of simulation modeling and aspect oriented programming. It will also bear discussion of the simulation modeling framework AnyLogic and aspect oriented programming language AspectJ, as these are the software packages used in this research. As mentioned in the 1.4, aspect oriented programming is used with a Java based simulation modeling framework to resolve some of the existing difficulties with simulation modeling. In order to motivate this work, we start this chapter discussing a drawback of OOP and how AOP helps to address it. Subsequently, we discuss the important concepts of logging, tracing and MCMC as they form the backbone of our solutions to the problems we address in this thesis. Moreover, we mention other related work that inspired us and helped us with this research. Finally, we highlight the novelty of our work in the context. In this light, it also bears emphasis that our approach of addressing these issues is based on the use of aspect oriented technology although AOP has been utilized marginally in this context as of now. As a consequence, the existing related literature is limited.

2.1 Java Reflection API

The ability of a program to examine itself and its software environment at run-time is known as Reflection [12]. Reflection is commonly used by programs requiring this ability to inspect and then modify run-time behavior of an application depending on the results returned from the inspection. The Java reflection API is an advanced feature of Java that enables the inspection of (*reflects*) the classes, interfaces, methods and fields at run-time [39, 18]. Reflection in Java empowers the developer by enabling him/her to perform run-time actions like creating objects given their class names, calling methods by their name, and access object fields given their name [12]. With the help of reflection, applications can perform certain operations which would otherwise be impossible. As such, it is a very powerful feature. We have made strong use of Java reflection APIs along with other Java features to support the generic features of the developed frameworks within the context of AspectJ.

Some of the prominent uses of java refection that have been utilized in our research are highlighted below:

- Inspect Java classes of an application (simulation model in our case) at runtime — This feature is being used to obtain the name of the class, package, methods, fields and parameters.
- Inspect methods of a class and invoke them at runtime — This feature is being used to invoke a modeler-provided *posterior* (explained later) method in the MCMC tool.
- Inspect the fields of a class and get / set them at runtime — We are using this feature to retrieve the name of the experiment of the simulation model being executed.

2.2 Limitations of OOP overcome by AOP

Quality maintenance has been a central issue for software developers. As such, a substantial amount of time and study has been invested to lessen the complexity of code underlying developed software in order to enhance ease of understanding, re-usability, robustness and maintainability. These characteristics are important to software projects, as these are the characteristics directly related to overall software quality. For instance, a future enhancement over any developed software would require changes in the existing code. A code base that has been developed with maintainability in mind is easier to modify, implement and less prone to errors. Thus, software developers emphasize building applications adhering to both functional as well as quality requirements, which makes it a more complex activity and also requires appropriate programming languages and development paradigms. In order to cope with this complexity, Computer Science researchers have contributed to an evolving set of development methodologies, design approaches and programming languages, including object oriented languages. The principal philosophy of object oriented programming (OOP) is to deal with objects. To that end, object oriented programming use classes. In fact, object oriented programming paradigm has been widely used over the last several decades. But there are times

when concerns in code need to span beyond a single module or class. If object oriented programming is used in these scenarios, then a single design pattern is implemented by many classes together, often making those classes tightly coupled and hence difficult to read, understand, maintain and reuse. So although object oriented programming methodology has provided good service in encapsulating core concerns into modules or classes, it has still failed to address the following two fundamental issues:

- The issue of Code Tangling : When a single module is associated with multiple concerns. This refers to the common situation in which a single module is responsible for more than one functionality. While in OOP, each class should ideally focus on a offering a single variety of functionality, this is not always feasible. Common offenders include logging, transactioning and security functionalities. For example, consider a banking system in the production environment where the module for balance enquiry is developed in OOP. We will find that although the module is responsible for balance enquiry — meaning that it will provide the customer with their balance information — it also contains the functionality of logging and security. Each time a balance enquiry is executed, the activity needs to be logged and before any activity the customer's credentials must be authenticated for security purposes. This gives rise to the issue of the tangling of code, whereby the code for logging and security remains intertwined in the code dealing with the primary business logic of balance enquiry.

- The issue of Code Scattering: When a single concern is spread across multiple modules. "Code Scattering" refers to the situation in which a single functionality is required by multiple modules. This is also a very basic problem that directly affects maintenance. The common offenders — logging, transactioning and security functionalities — discussed in the last point are guilty in this case as well. Considering the banking example developed in OOP once again, let us examine the module responsible for crediting accounts and the one for debiting account along with balance enquiry. All three modules require the functionality of logging, transactioning and security along with the core functionality that each are supposed to support since each action related to banking needs to be logged and before any activity a customary security check must be performed. Transactioning is required to ensure atomicity of debits, independence and other properties. This generates scattering of code whereby the code for logging, transactioning and security remains intertwined with the main business logic of the individual modules, creating an inter-dependency between modules.

Figure 2.1 is the pictorial illustration of both these issues. In this figure, the purple blocks represent the logging and security code.

Aspect oriented programming is a way of resolving these issues by decoupling such intertwined concerns, thereby enhancing modularity of the system [25]. Figure 2.2 depicts how aspects can extract the code responsible for inter-dependency among modules — represented by the purple blocks — and instead form a distinct building block which can then be referred to by all concerned modules. In effect, aspect oriented programming is a step in the continuous evolution of software techniques to ensure higher efficiency in

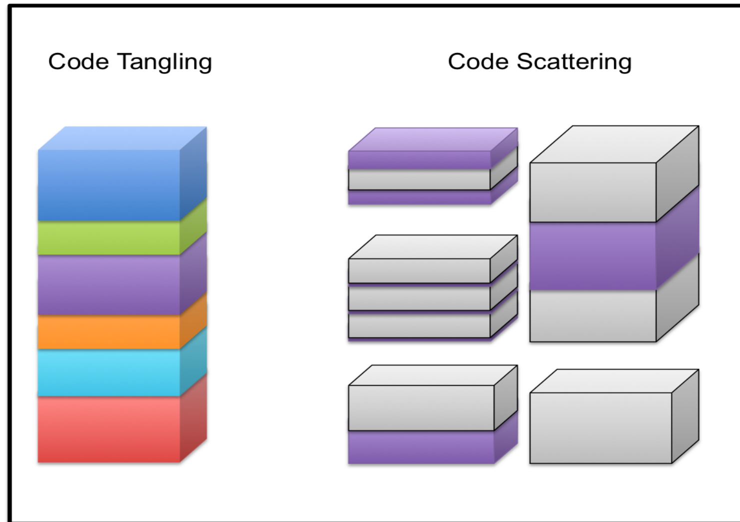


Figure 2.1: Issues with OOP

software development and improved software quality. But we note that aspect oriented programming is not a replacement for object oriented programming. Instead, it is a solution to some of the problems inherent in object oriented programming. It provides new constructs called aspects that can aid the refactoring of programs. AOP complements OOP by providing an enhancement in modularity that pulls together scattered implementation of a concern into a single unit, enhancing OOP modularity.

2.2.1 Crosscutting Concerns

A particular instance of code scattering is also referred to as a *crosscutting concern*. A crosscutting concern refers to a construct dealing with a single responsibility but scattered across a code-base. As a consequence, there is needless coupling between areas of the code. The issue of one part of the code affecting another part, is the issue of a crosscutting concern — common examples (again) are logging, security and tracing [7]. When the same functionality is spread across multiple classes, any change in this particular functionality can affect all of the classes. Crosscutting concerns emerge when the same functionality is required by multiple classes of the system.

Figure 2.3 [9] provides an illustrative example of crosscutting concerns in the context of simulation modeling. Here, the code related to tracing the entrance to individual states is intertwined with the core application logic. This example consists of a *statechart* with four different *states*. The tracking is intended to trigger a report every time an agent enters any of these states. As a result of this requirement, the tracking code is present in all the four states, thus generating crosscutting concern.

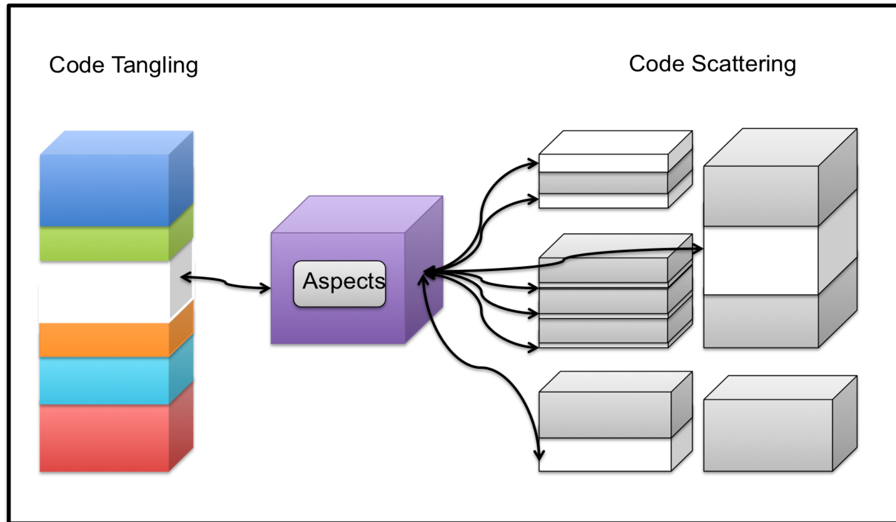


Figure 2.2: AOP prevailing over OOP

2.2.2 Aspects' role in eliminating crosscutting concerns

Aspects identify, separate and reduce the widespread implementation of crosscutting-concern code into a single modular unit. Aspects combine different building blocks of AOP, about which we will discuss in detail in later sections of this chapter, as well as provide a modular implementation for a crosscutting concern. Figure 2.4 [9] is based on Figure 2.3 and illustrates how the aspect oriented approach extracts the crosscutting-concern code from the core logic and reorganizes the same code within a new construct to create a simplified implementation of the required functionalities, absent crosscutting concerns. In this case, the code associated with tracing is contained in a related set of aspects. This aspect is used by different modules to take care of the tracing functionality.

2.3 AspectJ

AspectJ is the aspect oriented extension to the object oriented programming language Java and provides support for modular implementation of common crosscutting concerns. AspectJ provides new constructs that can aid aspect-based refactoring of Java programs [3, 37, 29]. The constructs are of two types: dynamic and static [14, 25]. The dynamic constructs are *aspect*, *pointcut*, and *advice*, while the static construct is the *inter-type declaration*. Each of these will be described in the next few subsections.

2.3.1 Join Points

Join Points are specific points in the execution of the program where crosscutting code can be woven. They are distinct junctures where crosscutting concerns are found. They form the basic entity and the most important concept of AspectJ, and the entire infrastructure of aspect oriented approach revolves around

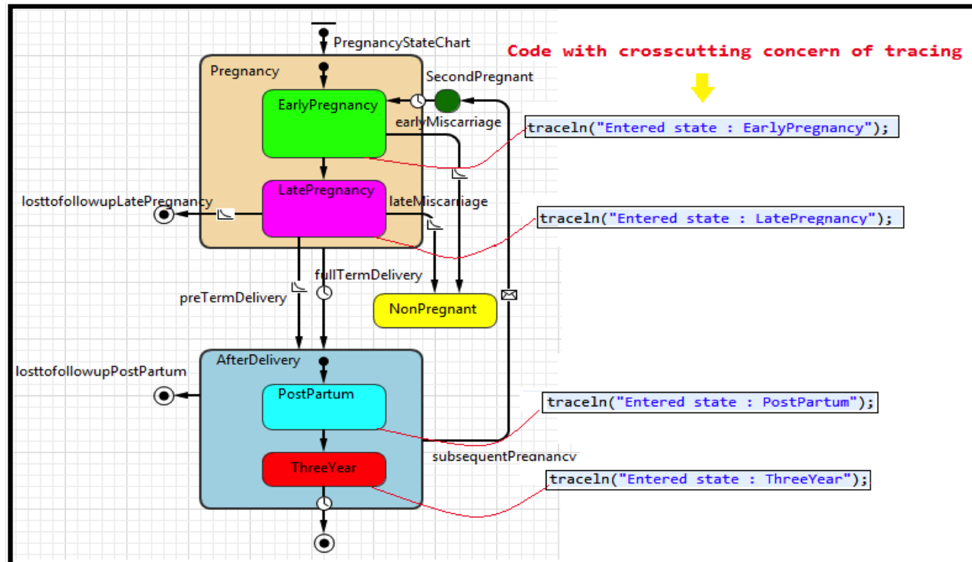


Figure 2.3: Example of crosscutting concern

them. Table 2.1 presents several join points that can be used while working with AspectJ [29, 6]:

It is important to note that the execution of the methods and constructors are associated with two different time frames. One is when they are called – this is usually when other methods calling the method in question method executes and is represented by the *call* join point. These join points have access to the execution context information prior to the actual call of the method. The other is when the methods are actually executed – this is usually when the code within the particular method executes. This is represented by the *execution* join point. Execution join points are associated with executing a method and have access to the execution context information inside the body of the method.

The next section discusses pointcuts, the mechanism AspectJ uses to refer to join points.

2.3.2 Pointcuts

Pointcuts are AspectJ constructs that identify and specify where and when to apply crosscutting code in the program flow by selecting a set of join points. The aspect present in figure 2.4 can be considered as an example to understand these concepts. Figure 2.5 provides an example of named pointcut. Here *scope* is the pointcut. Figure 2.6 provides an example of unnamed pointcut. Here the pointcut *scope* is also referenced from this anonymous pointcut. It is also clear from this example that AspectJ supports compound pointcuts, a number of pointcuts have been combined here with the AND operator (&&) to form a composite pointcut. The OR (||) operator and the NOT(!) operator are also used for creating composite pointcuts. In Figure 2.5 we see NOT(!) being used with the *within* pointcut.

In AspectJ, both anonymous and named pointcuts are used [25]. Anonymous pointcuts are unnamed, are defined as a part of advice and are defined at the position where they are used or with other pointcuts. As such they are not reusable, in contrast to named pointcuts that can be referred from different places.

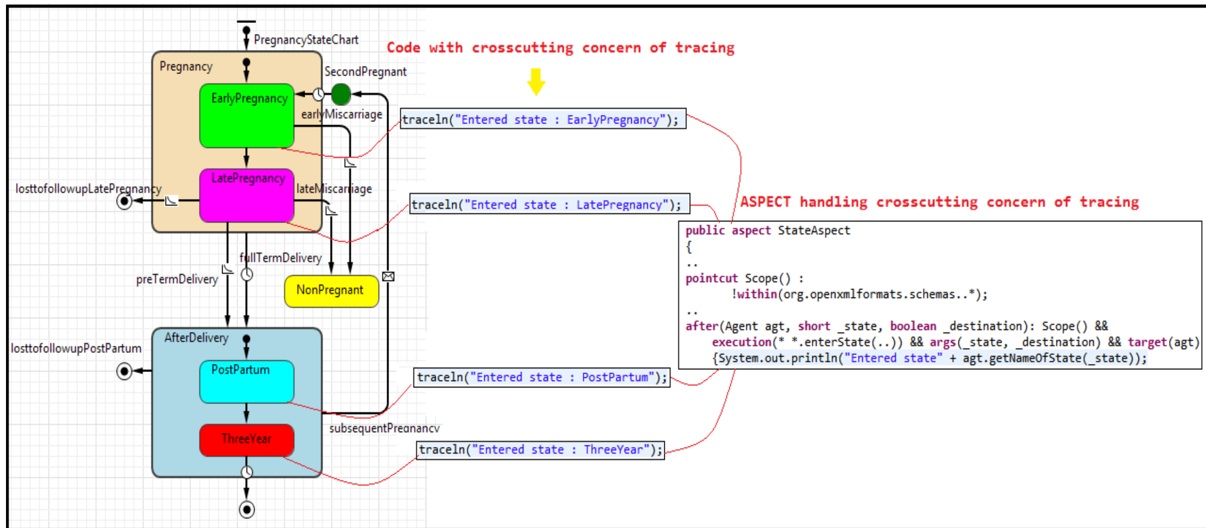


Figure 2.4: Resolution of crosscutting concerns via aspects.

```
pointcut Scope() :
        !within(org.openxmlformats.schemas..*);
```

Figure 2.5: Example of a named pointcut

Some of the most commonly used pointcuts in AspectJ are presented in table 2.2 [25, 6]:

2.3.3 Advice

An advice is an AspectJ construct that consists of a piece of code to be executed when the application or executing code reaches a join point, similar to a method in Java code [9]. It is the action and decision segment of the applied aspect that defines the action that needs to be completed [25]. The action code or the *code to be executed* is placed within the advice. The advice connects *code to be executed* to the join points specified by the corresponding pointcuts. The action or event associated with the join points specified by the pointcuts may be altered by having the advice code executed before, after, or instead of the action or event. There are three forms of advice identifiers that define the nature of the advice. These are presented in table 2.3 [25].

```
after(Agent agt, short _state, boolean _destination): Scope() &&
    execution(* *.enterState(..)) && args(_state, _destination) && target(agt)
    {System.out.println("Entered state" + agt.getNameOfState(_state));
```

Figure 2.6: Example of a unnamed pointcut

Join Points	**Description**
Method Call	When the method is called
Method Execution	When the method executes
Constructor Call	When an object's constructor is called — the object being constructed is returned and the object itself can be accessed with *after returning* advice (we will discuss about advice in a later subsection), it is present in the point where the logic is invoked
Constructor Execution	When an object's constructor is executed
Field Assignment	When a field is assigned with some value — these join points usually carry one argument (the assignment value) and their return type is void as no value is returned.
Field Reference	When a non-constant field is referenced
Handler Execution	When an exception handler executes — these join points also carry one argument (the exception) and their return type is void as no value is returned

Table 2.1: Description of Join Points

Figure 2.6 illustrates an example of the *after* advice. Here, the call to print serves as the *code to be executed* after the join points captured by the defined unnamed pointcuts are reached.

2.3.4 Inter-type declarations

AspectJ provides aspect constructs that affect the dynamic behavior or the execution behavior of a program. In addition to that, AspectJ features a powerful characteristic that modifies the static structure of the program — the classes, interfaces, and their compile-time behavior [25] – thereby allowing the addition of new functionality to an application. These additional declarations of new methods or fields to an existing Java class or type, contained within aspects, are known as inter-type declarations (ITD). This is also commonly referred to as *member introduction.*

2.3.5 Aspects

An aspect is the basic unit of modularity for Aspect oriented programming; it is analogous to a class of Java code. Aspect oriented programming is a way of resolving the issue of crosscutting concerns by decoupling them, and thereby enhancing modularity [25, 17]. Aspects are the aspect oriented programming constructs that abstract the modular implementation of crosscutting concerns. Aspects encapsulate pointcuts, advices, and inter-type declarations for a single concern into a single module [20]. When a Java program that has been extended by aspects is compiled, aspects are woven into the program as if they were part of it.

Pointcuts	Description
call	used to select join points whenever the specified method or constructor is called.
execution	used to select join points whenever the specified method or constructor is executed.
get	used to select join points whenever the specified field is read.
set	used to select join points whenever the specified field is written to.
handler	used to select join points whenever the exception handler associated with the Throwable type-pattern is executed.
within	used to select join points concerned with the executing code defined in the specified classes/ package.
this	used to select join points whenever the the currently executing objects are instances of the specified type.
args	used to select join points whenever the arguments are instances of the specified types.
target	used to select join points whenever the target objects (the objects on which the operation is applied to) are instances of the specified types.
&& *(AND)*	used to select those join points which are selected by both the pointcuts specified on either side of && operator.
|| *(OR)*	used to select those join points which are selected by either of the pointcuts specified on either side of || operator.
! (NOT)	used to select those join points which are not selected by the pointcut associated with ! operator.

Table 2.2: Description of Pointcuts

2.4 Simulation Modeling and AnyLogic

2.4.1 Simulation Model

A simulation model is a simplified representation of a system that exists in reality. Construction of simulation models enable theorizing concerning interactions between different elements of this world in order to facilitate understanding of complicated systems [31]. It serves as a tool to reason about issues that we face in the real world, especially cases in which it is difficult or infeasible to experiment with real-life complex systems such as organizations, health care systems, diseases Simulation models are well recognized as helping researchers to address many challenges faced by traditional statistical methods, like capturing time-related constraints and delays, complex and reciprocal casual dependencies, and non-linearities [19]. Simulation modeling approaches have increasingly been used in recent years to understand multi-level determinants of complex system challenges across different fields of study. *System Dynamics*, *Discrete Event* and *Agent-Based* Modeling are three well known computational modeling methods.

Advice	Description
before()	used to execute the code in the advice before the execution of the captured join points.
after()	used to execute code in the advice after the execution of the captured join points.
around()	used to execute code in the advice at the location of the captured join points. This construct surrounds the join points and allows changes to the selected join points, ignores it or continues with it. It also enables executing the captured join points multiple times with different arguments.

Table 2.3: Description of Advice

2.4.2 AnyLogic

AnyLogic is a multi-method Java-based simulation modeling framework that supports the three common simulation methodologies — System Dynamics, Discrete Event modeling and Agent Based, and combinations thereof. AnyLogic uses an object oriented approach. It is written in Java and it processes models which are also written in Java. In addition, AnyLogic is based on the Eclipse framework. It allows incremental development and provides the flexibility required for the construction of complicated models involving various dynamics and diversity [2]. It supports a well-designed enriched graphical interface that guarantees ease of use for most modelers.

The below subsections contain brief descriptions about the three modeling techniques supported by AnyLogic.

2.4.3 Agent-based Modeling

Agent-based Modeling is a computational methodology that enables a modeler to create, analyze, and experiment with models composed of active individual entities or agents interacting in an environment [14, 38]. Agents are used to represent social actors such as individual people, organizations or bodies and are programmed to react to the computational environment in which they are located in a way that typically mimics hypothesized elements of the behavior of corresponding agents in the real world environment [14]. A significant component of agent-based modeling in AnyLogic is the statechart. A given statechart represents the various states an agent can find itself in with respect to a given concern. A statechart also includes transitions associated with changes from one state to another. By viewing the system as a population of heterogeneous agents, agent-based simulation offers the possibility of modeling individual heterogeneity, representing explicitly agents decision rules, characterizing individual agent history, and situating agents in a geographical, network or other type of virtual space [14, 43]. Agent-based modeling models the ways individuals interact with each other and their environment and change and adapt in response to these interactions as well as the ways the environment changes in response to the actions of individuals. Agent-Based Models do not resolve the difficulties associated with obtaining empirical data. However, the development of these models benefit from information on empirical relationships in order to specify realistic parameters and algorithms, and can

help highlight areas for which empirical data are lacking, encouraging the collection of additional data. By comparing the emergent behavior of the model with empirical data, researchers can also more quickly spot inconsistencies between their hypothesized understanding of the world (as captured by the model) and what is shown by the empirical evidence.

2.4.4 System Dynamics Modeling

System Dynamics Modeling is a computational method that helps understand complex systems through the use of models that focus on feedback- and accumulation process, with model mechanisms expressing the hypothesized cause-and-effect relationships between variables of an entire system and its sub-systems [14]. It involves the application of both qualitative and quantitative methods. The qualitative method leads to the process of analyzing the system by finding loops and interlinks between variables (sometimes also including stock-and-flow structure), while the quantitative method uses fully specified stock and flow diagrams in order to simulate the system over time [42]. System Dynamics modeling has a mathematical foundation in differential equations and their numerical solution over time. This technique involves the use of *stocks* — which can be considered as accumulators — and *flows*, which can be considered as the pathways of inflow/outflow from the stock. System Dynamics mechanisms are commonly concerned with evolving conceptual understanding on the part of the modeler – and often on the part of stakeholders. System Dynamics will often approach understanding the system at an aggregate rather than at an individual level, meaning that individuals are not actually represented independently from one another, and instead groups of like individuals are aggregated together as stocks [30], with the value of the stock corresponding to the count of individuals within a certain state and/or with a particular set of characteristics. As a result, the System Dynamics approach is most commonly used for modeling aggregate behavior or for those circumstances where there are large populations of behaviorally similar agents, but challenges exist for modeling heterogeneity or in trying to address unique behavior and dynamics among individual agents [46, 14]. As such, representing agent behaviors that depend on the agents past experience, memory, or learning is also a challenge in a System Dynamics model [14].

2.4.5 Discrete Event Modeling

Discrete Event Modeling is another computational simulation method that focuses on the process flow of events in a system, that portrays the flow of individual entities through an ordered sequence of events or activities — commonly a defined and resource-limited workflow — involving different elements along with the time period associated with the completion of each activity. Discrete event modeling fundamentally represents the state of each entity in the system at each point in time, and performs bookkeeping on the resources needed for that entity to proceed. Hence it is suitable for systems where changes take place at discrete points in time [15]. Discrete event models commonly include representation of entity movement through quite detailed spaces, where movement times are endogenous features of the system, and result from

resource and entity placement and availability. A significant aspect of discrete event modeling is the key role of resources in mediating the flow of entities through the system. Entities are en-queued when such resources are unavailable. In contrast to agent-based modeling, discrete event modeling does not typically allow entity-entity interaction other than by resource depletion. For several decades Discrete event modeling has been largely used in the area of Operational Research [42]. Activity diagrams serve as a conceptual model in order to describe the interaction between entities within a certain event in discrete event modeling techniques. In turn, this helps modelers to understand the logical sequence linking entities to events and also to resources. Discrete Event Modeling is relatively easy particularly for users to understand with the aid of animation and graphics.

2.5 How a Simulation Model Works

A simulation model typically mimics in an abstracted way a hypothesized system, often (but not always) a real world system. Systems or processes in the real world are often extremely complicated, involving large numbers of factors and dependencies. In some cases, simulation models instead focus on a thought-experiment world, constructed to observe the logical implications of certain assumptions. The facts that are most important when developing a simulation model are that the processes needing to be simulated are unusually complex and that the overall behavior of the entire system is usually very different from that of its individual parts. A simulation model attempts to consider all important factors related to a system and then mimic it in a computerized version to carry out various checks, balances and measures virtually that are otherwise not feasible to do in real life. One well known example is the study of the spread of an infectious disease and developing ways to minimize the spread. A simulation model is usually developed by a modeler. Often the modeler is not always well acquainted with domain knowledge of the system he/she is simulating. For this purpose, the development of simulation models usually involve both modelers, who are knowledgeable about software used to develop the model, and end-users, who are the domain experts and are knowledgeable about the functioning of the real life system. Empirical data and existing field literature and body of evidence also has a major role to play in this context. Once the simulation model is developed using knowledge from different sources, the model is executed or run and results obtained. The output is then analyzed – particularly over historical periods – to check if these are plausible results, whether these match observations and data familiar to and expectations or predictions of the domain expert; in some cases, the results are also validated against empirical observations – particularly concerning the behaviour of the system – to find out how well model behaviours align with real-life results.

The goal of this thesis is to enhance the functioning and ease of use of simulation models. We seek to provide solutions that will overcome some of the constraints that a modeler comes across while seeking an inbuilt mechanism of documentation, preservation of information for future use, easy reproduction of results and a streamlined way of detecting flaws and (alternately) developing confidence in the simulation model.

Since the motivation and the purpose of the thesis revolves around simulation modeling, we discuss the workings of a simulation model to better motivate the research.

One of the assumptions made in this research is that an automated structured framework, including logging and tracing, will address an important need of simulation modelers. At this juncture, it is necessary to discuss these features and to emphasize why they are important in this context and how they can take care of the limitations that we are trying to address.

2.5.1 Logging

The term logging refers to a specialized mechanism of recording the high-level details of a system's execution. Logging is the process of systematically storing summary level information related to each simulation, including but not limited to, high-level information related to model outcomes and the meta-data containing specifics of the particular simulation that is required for recreating identical model output in the future. The following considerations motivate logging in the context of simulation modeling:

- Re-running the same experiment with different sets of data is a typical practice among modelers. While meticulously saving complete information is required to better reproduce the scenario for study, analysis and reference, often this is not done due to absence of a due process.
- In lieu of an automated logging process, the only options are manual documentation and storing, or, reworking. However both these processes consume a substantial amount of time, and reworking essentially means wasting of valuable resources that could otherwise be devoted towards more analysis and study. Both of these processes can also introduce the risk of error into the process.
- While a crucial element for ensuring the reproduction of a scenario, the random number seed that is an integral part of the models are difficult to obtain manually, as they are typically auto-generated by the code itself.

Each of the above points underscores the need for an automated logging mechanism when conducting simulations. An automated logging mechanism directly preserves information about the assumptions underlying observed behavior, thus permitting better reproduction of the experiment (and variants thereof) for further study. And retaining such information saves the modeler from reworking, and saves both human and computer time, and reduces the risk of error.

2.5.2 Tracing

Tracing is the process of recording low-level details of a system – such as a simulation model – execution. Within the scope of simulation modeling, tracing takes note of individual actions taking place within the model in sequence as they appear. This facilitates the capturing of events occurring in the system. Tracing effectively opens a window to the modeler or the user providing a glimpse of the dynamics occurring "behind the scenes"

during model execution. The points listed below highlight the reasons for supporting the development of a tracing mechanism in the context of simulation modeling:

- Many modeling frameworks, including AnyLogic used in this research, do not provide a standardized automated tracing mechanism. To introduce any kind of tracking or tracing functionality, modelers need to include new code. This process is not only time consuming, but also cumbersome as it creates a cluttered console. This reflects the fact that modelers need to enable tracing only in the areas required at that particular time, meaning they have to constantly get involved with the process of adding new lines of code for tracing and later deleting the same, thus wasting time. Moreover, AnyLogic users face two more issues in this context:
  - within AnyLogic, traces are printed in the console, and the console has a threshold beyond which data is lost from the console.
  - events do not appear in the actual sequence in which they are occurring.
- The manual tracing mechanism is likely to be error prone since it essentially depends on the memory of the modeler, and the modeler needs to remember to add the tracing code consistently every time a new module is added to the model. It also relies on the modeler's compliance to diligently add the code in all the required locations.
- Often the individuals who develop models (modelers) are not the ones who use the models for policy analysis or decision making (end-users). And for this reason the end-users often fail to grasp a lot of information and learning that is gained by the modelers, even though the latter are the ones who need to understand model elements thoroughly in order to make correct use of that model. But since most end-users do not participate in the process of development and execution, they often miss the insight gained by modelers while watching the models run from the start to end. In some cases, a detailed trace log can provide end-users with a suitable idea of the intricate details in complex models.
- Simulation models are usually complicated and involve many factors. Without a document capturing the inner details of the model, authenticating the correctness of a model remains difficult. Recourse to trace information enhances clarity and depth of understanding concerning model behavior. In case of error, tracking down the system and finding the source of the error is also quite challenging.

The above points stress and motivate use of an automated tracing mechanism to facilitate the smooth functioning of a simulation modeling framework. Tracing empowers modelers as well as end-users to better understand what is going on in the modelled system and to discern whether something is incorrect. Thus, it also simplifies the complex job of debugging and raises confidence in the accuracy of a model's implementation. Tracing information can further serve to allow systematic investigation of *why* certain high-level model results were obtained.

2.6 Markov Chain, Monte Carlo and MCMC

The second part of this thesis is concerned with exploring the use of aspect-based support for the estimation of dynamic simulation model parameters using a powerful Bayesian Machine Learning approach known as Markov Chain Monte Carlo methods, or MCMC.

A Markov chain is a mathematical model representing a random phenomenon with time, capturing the fact that the future is only affected by the past via the present [23]. Here time can be discrete, continuous or an ordered set. A Markov chain actually denotes states governed by a transition probability, where the current state depends only on the most recent previous state [1]. Basically, it is a technique used to generate samples where the most recent sample generated is reflected only by the current state of the Markov Chain. This method is termed the Monte Carlo method since samples from a posterior distribution on parameters (θ) are generated repeatedly. Samples generated from the posterior can be used to compute statistics or credibility intervals over θ, or over other model quantities computed using those parameters. MCMC is a general purpose technique for generating samples from a probability distribution in space based on some constraints.

Our goal in this part of the thesis is to aid modelers when simulating systems with missing or insufficient data. To that end, we attempt to cross-leverage the Markov Chain Monte Carlo techniques with dynamic simulation models as this latter technique allows sampling from posterior distributions of model parameters. MCMC combined with dynamic modeling can be utilized for a wide range of dynamic models, and thus it is a highly generalizable technique that has the capability to overcome the challenges faced by other techniques in this context [35], including those discussed in the next few paragraphs.

The complex and disjoint relationship between parameter values and emergent behavior makes it almost impossible to estimate parameter values directly from emergent behavior observed in the real world. Moreover, often while knowledge of a high-level system is often available, we might lack data based on specific model parameters. This hinders the process of specific parameter estimation. Calibration and Parameter Estimation [40] have been used to estimate parameter values and model outcomes by making use of empirical data regarding the emergent behavior of a system characterized by a particular simulation model. The most common use of Calibration and Parameter Estimation process is to arrive at "point estimates" concerning model parameters. Unfortunately, calibration results obtained from point estimates do not usually provide a clear picture of the degree of confidence associated with the results: whether there are other parameter estimates – perhaps very different – that might be nearly as favourable, and how broad the uncertainties are around the error estimates. Partly in order to achieve this kind of insight, another process known as "sensitivity analysis" is often used. When provided, associated error bounds do not reflect the resultant variability to be expected in the model outcomes. Decoupling sensitivity analysis from calibration suffers from the drawback of ignoring the effects of modified assumptions regarding one variable on the calibrated value for another.

The approach of cross-leveraging the MCMC technique with dynamic simulation model caters to a more rigorous way of deriving posterior distributions for model parameters [35]. The advantage of this approach is that it is capable of providing insights that are otherwise sought from the combination of calibration and sensitivity analysis techniques. In fact, this approach tends to overcome many of the limitations accompanying these two traditional techniques by helping to identify a form of the posterior parameter distribution. This is an eminently generic approach that can be used for a wide range of result-oriented analysis. Furthermore, MCMC can be reused for newly arrived data points, thus providing an alternative to the requirement of recalibrating models when new data arrives.

2.6.1 Metropolis Hastings Algorithm

In this research, we implement the random walk Metropolis Hastings algorithm [35]. The number of iterations as well as the duration of the burn-in period (the number of "throw away" MCMC iterations before samples should be drawn from the distribution) can be set by the user. The number of iterations play a significant role in ensuring convergence of the Markov Chain. Ensuring convergence of the MCMC sampling process is a vital part of this technique. The burn-in period is usually specified so that the Markov Chain can utilize this time to reach an equilibrium. Here, we carried out experiments where the total number of iterations including the burn-in period was in the range of 10,000 to 100,000, while the burn-in period was in the range of 2,000 to 20,000. These are typical for this type of computational analysis.

The algorithm is as follows [32] :

1. Initialize x_0.
2. For $t = 0, 1, 2, ...$ do
3. Set $x = x_t$
4. Sample $x' \sim q(x'|x)$
5. Compute Acceptance probability

$$\alpha = \frac{\tilde{p}(x')q(x|x')}{\tilde{p}(x)q(x'|x)}$$

6. Compute $r = \min\{1, \alpha\}$
7. Sample $u \sim U(0, 1)$
8. Set new sample to

$$x^{t+1} = \begin{cases} x' & \text{if } u < r \\ x^t & \text{if } u \geq r \end{cases}$$

Note: Here the candidates are denoted by x. However, later in the paper, we use θ to represent the candidates or parameters.

The random walk metropolis algorithm is pictorially depicted in Figure 2.7 [8].

The important steps of the algorithm are

- Choose an initial value of the parameters.
- Choose the step size for perturbing the current candidate to generate a new candidate.

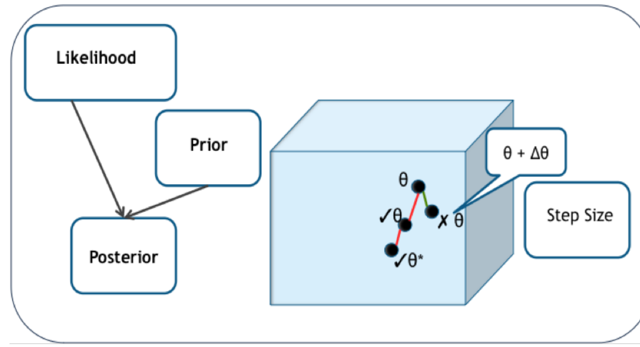


Figure 2.7: Random Walk Metropolis Hastings (single-walker version)

- Compute the posterior probability of the new candidate using the prior density and the likelihood functions.
- Use the Posterior probability to decide whether to accept the new candidate.

2.7 Prior, Likelihood and Posterior

2.7.1 Prior

"Prior" refers to any prior knowledge related to an unknown parameter of a system before any (i.e., prior to the point where) experimental observations are taken into account [32]. In the context of Bayesian approach, a prior $p(\vec{\theta})$ stands for the belief, often emerging from domain expertize or system knowledge, pertaining to the distribution of an unknown parameter before any observations are made.

2.7.2 Likelihood

"Likelihood" refers to the chances of observing a set of data with given parameters [32]. Likelihood in literary terms means the chance or probability of an action taking place. In the context of the Bayesian approach, likelihood $p(\vec{y}|\vec{\theta})$ refers to the conditional density of the data, that is, the probability or chance of observing a set of data when observed data and parameters are specified.

2.7.3 Posterior

"Posterior" refers to the conditional probability density of the parameters, given the data [32]. In other words this reflects the chances that a certain value of the parameter obtains, given a set of data. Since the posterior comes in the picture post or after the observations are made, the term "posterior" is used. Posterior distributions can be used to reach conclusions regarding unknown parameters, to make predictions, and thus make decisions based on them. In the context of the Bayesian approach, given a set of observed data, the

posterior function $p(\vec{\theta}|\vec{y})$ for a set of model parameters can be computed from the prior function and the likelihood function for the same set of parameters, given the observed data.

The posterior $p(\vec{\theta}|\vec{y})$ is computed from the prior $p(\vec{\theta})$ and the likelihood $p(\vec{y}|\vec{\theta})$ functions by using Bayes' Rule as follows:

$$p(\vec{\theta}|\vec{y}) = \frac{p(\vec{y}, \vec{\theta})}{p(\vec{y})} = \frac{p(\vec{y}|\vec{\theta})p(\vec{\theta})}{p(\vec{y})}$$

$p(\vec{y})$ is a constant which can be determined by the empirical data and is not related to $\vec{\theta}$. Thus,

$$p(\vec{\theta}|\vec{y}) \propto p(\vec{y}|\vec{\theta})p(\vec{\theta}).$$

Thus posterior is proportional to the product of the prior and the likelihood:

Posterior $\propto$ Likelihood $*$ Prior

Chapter 3

The Logging and Tracing Tool

We have utilized the core concepts of aspect oriented programming to design and implement a framework that can be easily applied on the java based simulation modeling framework, AnyLogic. The framework has been termed the logging and tracing tool, since it creates both a run log and a trace log along with several other elements important for addressing the challenges related to documentation and storage, re-creation of results and debugging as discussed in section 1.2. In this section, we will shed some light on the design and implementation of the tool. Along with this, we will also discuss each output element in detail, focusing on what information they contain, why they are important in terms of improving the flexibility of the simulation modeling framework and how they can help modelers and users with some of the challenges that exist today.

The primary goal of the logging and tracing tool involves the two essential concepts of documentation and storage. The tool documents automatically the high-level details in the run log and the low-level details in the trace log related to execution and places these documents in the same location in the computer where the model resides. The run log provides scenario metadata while the trace log provides metadata concerning the temporal process of execution. The run log has a major role to play when basic information like date, time, high level model assumptions etc. related to a past model run is required, and – more specifically – when future re-creation of the exact same output is required. The trace log enables the user to gain more insight about intricate details of the actions taking place in the model over time. Additionally, the logging and tracing tool is concerned with auto-generation of some other useful information related to the model, including the location of the model being executed. Both the run log and the trace log offer value as reference documents for future use. They provide notable support to modelers as they try to analyze model behaviour. In the event some unexpected issue is encountered in the model, these logs provide a great deal of useful information, helping to ease the complicated task of debugging. Although the logging and tracing tool can handle different kinds of AnyLogic models, it is most useful for agent based models. This is the case because the run log can be generated for a system dynamic or a discrete event model but the trace log will not contain much relevant information, as none of these latter models use states or statecharts, the latter forming a vital part of tracking the activities of agent based models. In order to use the tool, the JVM argument must be updated for each simulation. We provide the details in the section 3.4.

3.1 Logging and Tracing Tool Design

Implementation of the logging and tracing tool involves AspectJ, Java and AnyLogic. We used AspectJ 8 and Anylogic version 6.9 in developing this framework. However, we updated the tool to be compatible with Anylogic version 7.1. Our goal is to build a generic tool that can be used by any agent based model developed in AnyLogic. The steps that have been followed while designing this tool are as follows:

- A number of agent based models were selected for analysis. The list contains both example models provided by AnyLogic as well as third-party models. These third-party models included some that have already been developed by other modelers and which remain of research interest, along with newly developed models, i.e. models more recently developed by graduate students. AnyLogic auto-generates the Java code for each model and because this code must be instrumented with AspectJ, the structure of the Java code had to be analyzed.

- The AnyLogic-generated Java code for agent based models are evaluated with care to help identify common join points corresponding to specific functionalities that are responsible for various actions taking place in the model, including model initialization, model parameter set up, agents entering new states, and so on. The reason we need to identify these join points is that our aspect framework must be able to capture the important action points of the model in order to extract and document information about the intricate and the overall execution details.

- After the required join points are enlisted, a number of pointcuts are defined to select these join points.

- The next step is to declare the advice and Inter-Type Declarations (ITDs) required to execute the actions required for the functioning of the tool. This is a crucial step responsible for the core functioning of the aspect-based logging and tracing tool.

- Once the join points, pointcuts, advice and ITDs are identified and declared, the subsequent step of designing any aspect-based framework requires declaration of the aspects combining the individual AOP constructs. This marks the completion for handling the aspect related part of the tool.

- In addition to the aspect, a number of Java features like writing in a file, creating a new folder etc, have also been utilized while building the tool. The generic feature of the tool — again, designed to handle any agent based model built in AnyLogic — relies heavily on the usage of Java Reflection API, which is a powerful way of examining the run time behavior of a class. As such, Java Reflection helps in implementing functionality concerned with capturing the model parameters, name of the experiment, etc. Java Reflection code corresponding to the requirements of the logging and tracing tool are thus introduced to conclude the design of the framework.

3.2 Logging and Tracing Tool Implementation

This section provides details related to the development and implementation of the Logging and Tracing Tool. While providing a brief overview of how the tool fits in the context of models in AnyLogic, the section also discusses the individual components of the tool.

3.2.1 Implementation Overview

The implementation overview of the Logging and Tracing Tool has been illustrated in figure 3.1.

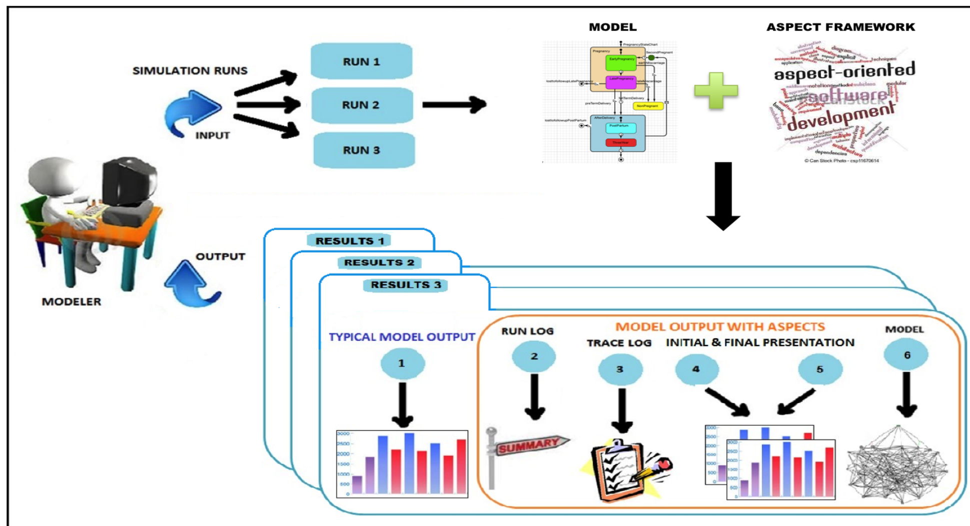


Figure 3.1: Implementation Overview of the Logging and Tracing Tool

The figure shows a modeler executing several experiments or runs for a particular simulation model. Here, “Run 1” through “Run 3” are three different scenarios (termed “experiments” in AnyLogic). When the aspect framework is applied on these experiments, additional output is generated beyond the usual output from the simulation model. In total, for each such scenario run, six different items are output. The aspect-based logging and tracing tool operating during each run creates a new sub-folder corresponding to that run. These sub-folders are named so as to incorporate a user-readable time stamp for the convenience of future reference and are stored in the same directory as the corresponding simulation model. All the outputs generated as a result of applying our tool are stored within this sub-folder. For each of the three such experiments, a set of results is created. In Figure 3.1, Number “1” in the diagram represents the usual output from the simulation model, which is created by the simulation model itself. Application of our tool does not affect the output generated by the simulation models, they get generated in the same unaltered form irrespective of the application of the tool on the model. Numbers “2” through number “6” in Figure 3.1 represent the additional output created beyond the usual output. Number “2” represents the Run Log, while number “3” represents the Trace Log. Numbers “4” and “5” represent snapshots of the output canvas of the model taken at two different times. Whenever a model starts to run in AnyLogic, the canvas window pops up, and a

snapshot of this window is captured in order to obtain the visualization of the initial state of the model. This image is referred to here as the *Initial Presentation*, and is shown in the figure as number “4”. Once the simulation is complete and the model output is generated, a snapshot of the canvas is again captured in order to obtain the visualization of the final state of the model. This image is referred as the *Final Presentation*, and is shown in Figure 3.1 as number “5”. Finally, a copy of the model source file (`.alp` file), represented in the figure as number “6”, is also stored within the same sub-folder. This preserves the current version of the model, ensuring easy re-creation of the output in the future.

3.2.2 Logging and Tracing Tool Framework

The Logging and Tracing Tool is built in the aspect framework. The code made extensive use of java reflection, plus various java packages and AspectJ. The entire code of the tool is encapsulated within the following aspect:

```
/*
 * Aspect to capture the various functionalities of the simulation code in
 * order to enable the actions required for the Logging and Tracing Tool
 */
public aspect LogTrace
{
...
}
```

Listing 3.1: Aspect encapsulating the code for the Logging and Tracing Tool

At a technical level, the main functions of the tool can be categorized into the following:

- Get user response as to whether to enable or disable the tool. The Aspect Code accomplishes this by calling up a JOptionPane from Java’s awt framework to prompt user confirmation as to whether enable or disable the tool. The user confirmation window is represented in figure 3.2.

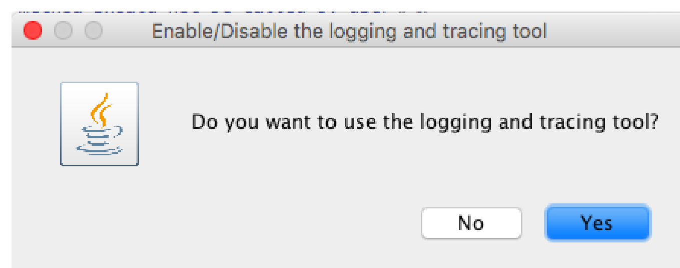


Figure 3.2: Enable/Disable the Logging and Tracing Tool

All other functions are carried on only if the user elects to enable the tool through the tool’s user interface.

- Get the name and location of the simulation model being executed and make a copy of the model (.alp) file.
- Capture the instant when the simulation model starts executing. The presentation window of the model is captured by the tool in the (very) initial phase of the execution.
- Check whether the random number seed used for the current execution has been specified explicitly by the user (using AnyLogic's "Fixed seed" option for the experiment).
- If the random number seed is not specified explicitly by the user, then the tool generates a random number seed, sets the same for the current execution and prints it. This is done to enable recreation of the output. It is inherently difficult to retrieve the random number seed generated by AnyLogic. This is how the reproducibility with the random number seed is handled in our tool.
- Get the name of the AnyLogic experiment being executed. It bears noting that each model will often have multiple simulations or experiments; for documentation, it is important to know which is being executed.
- Get the details of the parameters of the Main class.
- Get the details of a message being sent or received during the execution.
- Get the details of the change of state of an agent during the execution. This mechanism reflects the fact that agent based modeling in AnyLogic uses statecharts composed of various states.
- Capture the instant when the simulation model completes executing. The presentation window of the model is captured by the tool in the final stage of the execution to store the final state of the simulation.

In order to meet each of the above goals, a set of pointcuts and advice have been defined which are enlisted below. Please note that in most of the cases the original code has been replaced by pseudo code within an advice in the code listings.

Run Log and Model Copy

1. Capture the seed

   (a) The following pointcut *fixSeed* is used to capture the calls to the method *setSeed*. This method takes a number and sets the random number seed using this value. In the logging and tracing tool, the presence of this pointcut helps to identify whether a fixed random number seed has been provided by the user or the modeler while executing the simulation.

```
pointcut fixSeed(long seed):
  call(* *.setSeed(..)) &&
  args(seed);
```

Listing 3.2: Pointcut to find the seed

(b) The following after advice advises the join point captured by the pointcut *fixSeed* to capture the seed value. At the beginning of a simulation, the method *setSeed* is called with the number zero as the seed value from the class Engine. Thus the date and time when the simulation starts is recorded here. Moreover, the sub-folder corresponding to an execution is created along with log files because this is the first advice reached. This method is again called by the class simulation when the random number seed needs to be set to a certain value. The boolean field *randomSeedWritten* is false by default and is set to true when the random seed used for the simulation has been written in the run log.

```
after(long seed): fixSeed(seed)
{
  if (seed == 0) {
      Date startDate = new Date();
    createSubfolderAndFiles(startDate);
  }
  else if ((seed != 0) && (randomSeedWritten == false)) {
    randomSeedWritten = true;
    logFixedSeed(seed);
  }
}
```

Listing 3.3: Advice for finding the seed

2. Capture the model name and location

(a) The following pointcut *modelNameAndCopy* is used to capture the execution of the the method *setup*. This method is called from the main method of applications after the experiment is constructed and it handles the initial setup of the simulation. In the logging and tracing tool, this pointcut helps to intercept the underlying join point. At that joinpoint, the tool can locate the name and location of the simulation model being executed.

```
  pointcut modelNameAndCopy():
  execution(* *.setup(..));
```

Listing 3.4: Pointcut to find name and location of the model

(b) The following before advice advises the join point captured by the pointcut *modelNameAndCopy* to track the name and location of the model and then to create a copy of the model file.

```
before(): modelNameAndCopy()
{
  findModel();
  copyModelFile();
}
```

Listing 3.5: Advice for finding name and location of the model

3. Set the seed

   (a) The following pointcut *setSeed* is used to capture the execution of the method *setEngine*. This method takes an instance of a simulation engine and sets the same for the object. In the logging and tracing tool, this pointcut is used to intercept the join point where the tool generates a *random number seed* and sets the same for the current execution. This step is only undertaken if the fixed seed is not provided by the user or the modeler for the current execution.

```
pointcut setSeed(com.anylogic.engine.Engine engine):
  execution(* *.setEngine(..)) &&
  args(engine);
```

Listing 3.6: Pointcut to set the random number seed

   (b) The following after advice advises the join point captured by the pointcut *setSeed* to set the value of the random number seed to be used in the current execution. The random number seed value is set to the number randomly generated by the tool, which is referred to here as *seedNew* and the same is also written in the run log only if the boolean *randomSeedWritten* is false. Note that *randomSeedWritten* will have been set to true if the simulation is run with a fixed seed (see listing 3.3 above), and in that case these lines of code are not executed.

```
after(com.anylogic.engine.Engine engine): setSeed(engine)
{
  if (randomSeedWritten == false) {
    randomSeedWritten = true;
    logRandomSeed(seedNew);
    engine.getDefaultRandomGenerator().setSeed(seedNew);
  }
}
```

Listing 3.7: Advice for setting the random number seed

4. Capture the name of the experiment

   (a) The following pointcut *experimentName* is used to capture the calls to the method *setName*. This method sets the name of the current experiment. In the logging and tracing tool, this pointcut is used to extract the name of the experiment.

```
pointcut experimentName():
  call(* *.setName(..));
```

Listing 3.8: Pointcut to find the name of the experiment

(b) The following before advice advises the join point captured by the pointcut *experimentName* to write the name of the experiment in the run and trace logs.

```
  before(): experimentName()
{
  logExperimentName(thisJoinPoint.getThis().getClass().getSimpleName());
  traceExperimentName(thisJoinPoint.getThis().getClass().getSimpleName());
}
```

Listing 3.9: Advice for finding the name of the experiment

5. Capture the model parameters

(a) The following pointcut *mainParameters* is used to capture the execution of the method *setupRootParameters*. This method is called to setup parameters for the main class. In the logging and tracing tool, this pointcut is used to intercept the join point where the details of the parameters of the main class are sought.

```
pointcut mainParameters(final Agent self, boolean calOnChangeActions):
  execution(* *.setupRootParameters(..)) &&
  args(self, calOnChangeActions);
```

Listing 3.10: Pointcut to find the Main class parameters

(b) The following after advice advises the join point captured by the pointcut *mainParameters* to find the parameters and associated details and subsequently write the same in the run log.

```
  after(final Agent self, boolean calOnChangeActions): mainParameters(self,
  calOnChangeActions)
{
  getParameters(self);
}
```

Listing 3.11: Advice for finding the Main class parameters

Initial and Final Presentation

1. Capture the initial presentation

(a) The following pointcut *experimentStarted* is used to capture the execution of the method *run*. This method executes the model from the current state. In the logging and tracing tool, this pointcut is used to intercept the join points where the presentation window displaying the initial state of the execution is captured.

```
pointcut experimentStarted():
  execution(* com.anylogic.engine.ExperimentSimulation.run(..));
```

Listing 3.12: Pointcut to capture the initial presentation

(b) The following after advice advises the join point captured by the pointcut *experimentStarted* when the experiment starts so that a boolean field *experimentStart* is set to true. This is later used to capture the presentation window when the model starts executing.

```
after(): experimentStarted()
{
  experimentStart = true;
}
```

Listing 3.13: Advice for capturing the initial presentation

2. Capture the final presentation

(a) The following pointcut *experimentCompleted* is used to capture the execution of the method *onEngineFinished*. This method is executed when the engine finishes running and intends to wrap up the simulation run. In the logging and tracing tool, this pointcut is used to intercept the join points where the presentation window displaying the final state of the execution is captured.

```
pointcut experimentCompleted():
  execution(* *.onEngineFinished(..));
```

Listing 3.14: Pointcut to capture the final presentation

(b) The following after advice advises the join point captured by the pointcut *experimentCompleted* when the engine stops when the experiment completes, so that a boolean field *experimentStop* is set to true. This is later used to capture the presentation window when the model completes executing. Here, the time when the simulation completes is recorded and is written in the run log.

```
after(): experimentCompleted()
{
  experimentStop = true;
  Date endDate = new Date();
  logCompletionDetails(endDate);
}
```

Listing 3.15: Advice for capturing the final presentation

3. Capture the presentation object

   (a) The following pointcut *getPresentation* is used to capture the execution of the method *getPresentation*. This method returns the presentation object of the model. In the logging and tracing tool, this pointcut is used to get the presentation object.

```
pointcut getPresentation():
  execution(* *.getPresentation(..));
```

Listing 3.16: Pointcut to get the presentation object

   (b) The following after advice advices the join point captured by the pointcut *getPresentation* to get the presentation window when the experiment starts and when the experiment finishes.

```
  after() returning(Presentation window) : getPresentation()
{
  if (experimentStart == true) {
    experimentStart = false;
    captureInitialPresentation(window);
  }

  if (experimentStop == true) {
    experimentStop = false;
    captureFinalPresentation(window);
  }
}
```

Listing 3.17: Advice for capturing the presentation object

Trace Log

1. Track when a message is sent

   (a) AnyLogic uses messages as a primary means of communicating between agents; the following pointcut *sendMessage* is used to capture the execution of the method *send*. This method is used to send a message to a given agent. In the logging and tracing tool, this pointcut is used to intercept the join points where a message is sent and subsequently to track the details related to the message and its source and destination.

```
pointcut sendMessage(Agent agt, java.lang.Object msg, Agent dest):
  execution(* *.send(..)) &&
  args(msg, dest) &&
  target(agt);
```

Listing 3.18: Pointcut to track when a message is sent

(b) The following after advice advises the join point captured by the pointcut *sendMessage* to write the details associated with sending a message, such as the model time when the message is sent, the sender and the receiver along with the message, in the trace log.

```
after(Agent agt, java.lang.Object msg, Agent dest): sendMessage(agt, msg, dest)
{
  traceSendMessage(agt.time(),agt.getName(),msg, dest.getName());
}
```

Listing 3.19: Advice to track when a message is sent

2. Track when a message is received

(a) The following pointcut *receiveMessage* is used to capture the execution of the method *onReceive*. This method is executed to receive a message. In the logging and tracing tool, this pointcut helps to intercept the join points where a message is received and subsequently to track the details related to the message and its source and destination.

```
pointcut receiveMessage(Agent agt, java.lang.Object msg, Agent sender):
  execution(* *.onReceive(..)) &&
  args(msg, sender) &&
  target(agt);
```

Listing 3.20: Pointcut to track when a message is received

(b) The following after advice advises the join point captured by the pointcut *receiveMessage* to write the details associated with receiving a message, such as the model time when the message is received, the receiver and the sender along with the message, in the trace log.

```
after(Agent agt, java.lang.Object msg, Agent sender): receiveMessage(agt, msg,
  sender)
{
  traceReceiveMessage(agt.time(),agt.getName(),msg, sender.getName());
}
```

Listing 3.21: Advice to track when a message is received

3. Track entry to a state

(a) The following pointcut *enterAState* is used to capture the execution of the method *enterState*. This method executes the entry action of an agent into a state. In the logging and tracing tool, this pointcut is used to intercept the join points where an agent enters a particular state and subsequently the details related to the entry is logged.

```
pointcut enterAState(Agent agt, short state, boolean destination):
  execution(* *.enterState(..)) &&
  args(state, destination) &&
  target(agt);
```

Listing 3.22: Pointcut to track entry to a state

(b) The following after advice advises the join point captured by the pointcut *enterAState* to write the details associated with the entry action of an agent to a state. The model time when an agent enters a state is written in the trace log, along with the entering agent and the new state being entered.

```
  after(Agent agt, short state, boolean destination): enterAState(agt, state,
  destination)
{
  traceEntry(agt.time(),agt.getName(),agt.getNameOfState(state));
}
```

Listing 3.23: Advice to track entry to a state

4. Track exit from a state

(a) The following pointcut *exitAState* is used to capture the execution of the method *exitState*. This method executes the state exit action of an agent. In the logging and tracing tool, this pointcut is used to intercept the join points where an agent is exiting a particular state, and subsequently the details related to the exit is logged.

```
pointcut exitAState(Agent agt, short state, Transition tran, boolean source,
  Statechart statechart):
  execution(* *.exitState(..)) &&
  args(state, tran, source, statechart) &&
  target(agt);
```

Listing 3.24: Pointcut to track exit from a state

(b) The following after advice advises the join point captured by the pointcut *exitAState* to write the details associated with the exit action of an agent from a state. The model time when an agent exits from a state is recorded in the trace log, along with the exiting agent, the state being left, and the transition through which the agent is exiting.

```
  after(Agent agt, short state, Transition tran, boolean source, Statechart
  statechart): exitAState(agt, state, tran, source, statechart)
{
  traceExit(agt.time(),agt.getName(),agt.getNameOfState(state),tran);
}
```

Listing 3.25: Advice to track exit from a state

3.3 Output from Logging and Tracing Tool

The individual elements outputted by the logging and tracing tool are as mentioned below. The next few subsections contain details of each individual element.

- Run Log
- Trace Log
- Images of the initial and final state of the model
- Copy of the present model version

3.3.1 Run Log

The Run Log automatically logs the high-level details or summary level information related to a model execution. This essentially refers to high-level details of the model execution like date and time and the high-level assumptions made for a particular execution of that particular model like the random number seed used for that execution. Figure 3.3 presents a run log generated from a model run.

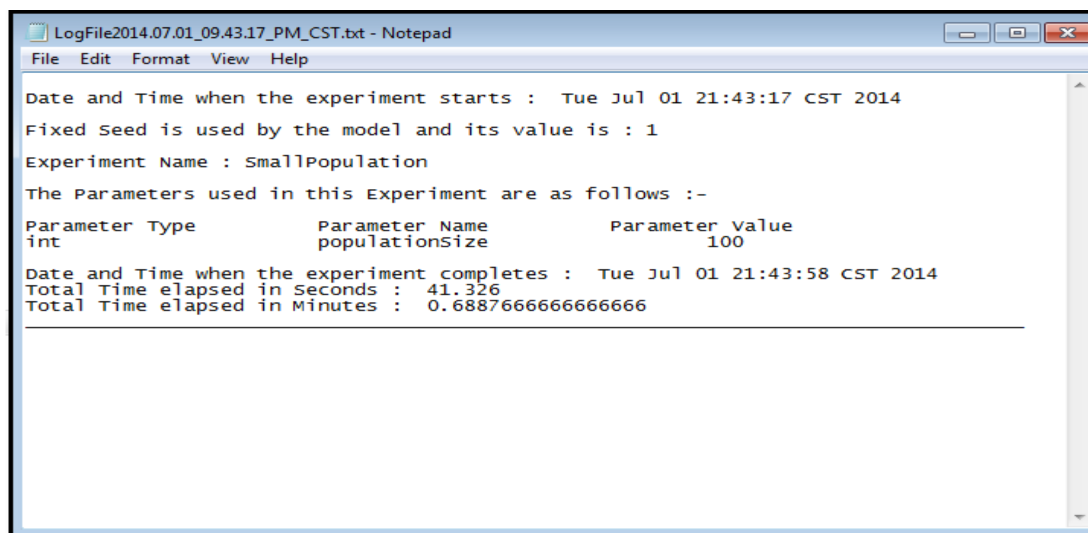


Figure 3.3: The Run Log

It provides a modeler or the user with the following information:

- The *date and time* when the experiment is *started* — this information plays an important role in the run log because modelers tend to seek experiments run on a certain date and time.

- The *random number seed* used by the model — whether fixed or randomly generated by the software framework, along with the corresponding values.

  The AnyLogic framework allows modelers to fix the random number seed value. When a fixed value is used in an experiment, the run log records this fact, and logs the fixed seed value. When the modeler does not choose to fix the random number seed and instead allows AnyLogic to randomly generate it, the run log denotes that by displaying the term "Random seed" instead of "Fixed seed". However, it is extremely difficult to retrieve the random number seed automatically generated by AnyLogic internally. So our tool generates a random number seed. This value is used for the model execution as well, as reflected in the run log for future use. We have discussed this issue with AnyLogic to streamline this process.

- The *name of the experiment* that is executed in that particular model run.

- The *parameter details* like parameter names, data type of the parameters and the value of the parameters used for the experiment. The run log only specifies the "model-level" parameters, which are present in the model's Main class (and not for individual agents of the model). We did not consider storing the agent-level details or the agent parameters as the agent-level assumptions can typically be derived from what we are already storing. We can re-create the exact same output with a copy of the model version, the seed and the model parameters.

- The *date and time* when the experiment *ended*.

- The *total time* consumed by the experiment to run — this information is significant in the run log as it can be a deciding factor for modelers when they wish to re-run a particular scenario but are not sure if they have time (equal to the logged model duration) to re-create it.

One of the goals of the run log is to serve as a reference document which modelers and users can refer to in the future in order to find details of runs executed in the past. This is one of the main reasons of using text files for run log, files being easy to read for humans. If a system requires the logs to be outputted in a database that can be done simply by modifying the aspects. The date and time when the model started and completed has a role to play in the context of future reference. Often, modelers seek to find scenarios that were run on certain dates. The total time taken by the model from start to finish is also significant in many ways, as it hints at the efficiency of the model. If a model takes a substantially greater amount of time than expected, then the modeler can try to figure out any irregularities that caused unexpected delays.

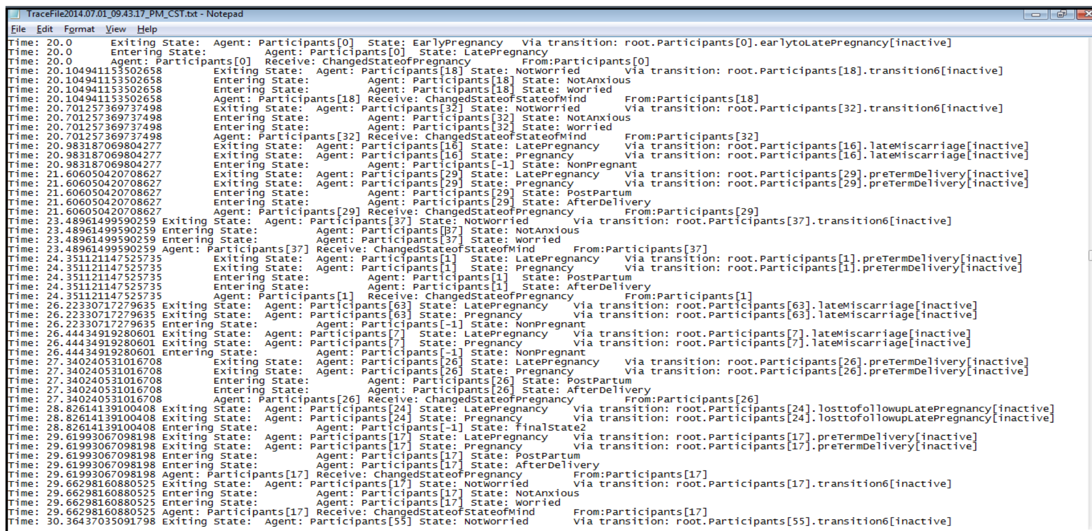
A reference document related to model execution is specially required when a large number of runs are carried out on a particular model to satisfy all test cases. This is also required when an existing model needs to be worked upon after a certain time gap. Details of past runs can prove immensely helpful as they provide understanding of the model assumptions considered previously. Details of the model parameters associated with a scenario provide understanding of the model assumptions for that scenario, hence the model parameters

are captured in the run log. Some of the models contain multiple experiments or simulation scenarios. For this reason, the run log captures the name of the experiment so that the modeler or user referring to the run log knows exactly what scenario of the model was executed in the past. Furthermore, the name of that scenario is often quite informative, and hints at what the scenario might contain and gives a sense of the intention behind running that scenario.

The other purpose of the run log is to enable the re-creation of model output. Knowledge of the random number seed is significantly important for re-creation of simulation output.

3.3.2 Trace Log

The Trace Log automatically logs intricate information related to a particular execution, typically moves or changes taking place in the model over time. In contrast to the run log (which captures parameters only at the level of the model as a whole), the run log captures the information at many different levels of a model (including agent-level) and on the low-level details pertaining to agent movements and the sequence of actions taking place in the model. Figure 3.4 presents a trace log generated from a particular run.



```
TraceFile2014.07.01_09.43.17_PM_CST.txt - Notepad
File Edit Format View Help
Time: 20.0    Exiting State:  Agent: Participants[0]  State: EarlyPregnancy   Via transition: root.Participants[0].earlytoLatePregnancy[inactive]
Time: 20.0    Entering State:      Agent: Participants[0]  State: LatePregnancy
Time: 20.0    Agent: Participants[0]  Receive: ChangedStateofPregnancy      From:Participants[0]
Time: 20.104941153502658      Exiting State:  Agent: Participants[18] State: NotWorried      Via transition: root.Participants[18].transition6[inactive]
Time: 20.104941153502658      Entering State:      Agent: Participants[18] State: NotAnxious
Time: 20.104941153502658      Entering State:      Agent: Participants[18] State: Worried
Time: 20.104941153502658      Agent: Participants[18] Receive: ChangedStateofStateofMind      From:Participants[18]
Time: 20.701257369737498      Exiting State:  Agent: Participants[32] State: NotWorried      Via transition: root.Participants[32].transition6[inactive]
Time: 20.701257369737498      Entering State:      Agent: Participants[32] State: NotAnxious
Time: 20.701257369737498      Entering State:      Agent: Participants[32] State: Worried
Time: 20.701257369737498      Agent: Participants[32] Receive: ChangedStateofStateofMind      From:Participants[32]
Time: 20.983187069804277      Exiting State:  Agent: Participants[16] State: LatePregnancy   Via transition: root.Participants[16].lateMiscarriage[inactive]
Time: 20.983187069804277      Exiting State:  Agent: Participants[16] State: Pregnancy       Via transition: root.Participants[16].lateMiscarriage[inactive]
Time: 20.983187069804277      Entering State:      Agent: Participants[-1] State: NonPregnant
Time: 21.606050420708627      Exiting State:  Agent: Participants[29] State: LatePregnancy   Via transition: root.Participants[29].preTermDelivery[inactive]
Time: 21.606050420708627      Exiting State:  Agent: Participants[29] State: Pregnancy       Via transition: root.Participants[29].preTermDelivery[inactive]
Time: 21.606050420708627      Entering State:      Agent: Participants[29] State: PostPartum
Time: 21.606050420708627      Entering State:      Agent: Participants[29] State: AfterDelivery
Time: 21.606050420708627      Agent: Participants[29] Receive: ChangedStateofPregnancy      From:Participants[29]
Time: 23.48961499590259 Exiting State:  Agent: Participants[37] State: NotWorried      Via transition: root.Participants[37].transition6[inactive]
Time: 23.48961499590259 Entering State:      Agent: Participants[37] State: NotAnxious
Time: 23.48961499590259 Entering State:      Agent: Participants[37] State: Worried
Time: 23.48961499590259 Agent: Participants[37] Receive: ChangedStateofStateofMind      From:Participants[37]
Time: 24.35112114752573 5      Exiting State:  Agent: Participants[1]  State: LatePregnancy   Via transition: root.Participants[1].preTermDelivery[inactive]
Time: 24.351121147525735      Exiting State:  Agent: Participants[1]  State: Pregnancy       Via transition: root.Participants[1].preTermDelivery[inactive]
Time: 24.351121147525735      Entering State:      Agent: Participants[1]  State: PostPartum
Time: 24.351121147525735      Entering State:      Agent: Participants[1]  State: AfterDelivery
Time: 24.351121147525735      Agent: Participants[1]  Receive: ChangedStateofPregnancy      From:Participants[1]
Time: 26.2233071727963 5 Exiting State:  Agent: Participants[63] State: LatePregnancy   Via transition: root.Participants[63].lateMiscarriage[inactive]
Time: 26.22330717279635 Exiting State:  Agent: Participants[63] State: Pregnancy       Via transition: root.Participants[63].lateMiscarriage[inactive]
Time: 26.22330717279635 Entering State:      Agent: Participants[-1] State: NonPregnant
Time: 26.44434919280601 Exiting State:  Agent: Participants[7]  State: LatePregnancy   Via transition: root.Participants[7].lateMiscarriage[inactive]
Time: 26.44434919280601 Exiting State:  Agent: Participants[7]  State: Pregnancy       Via transition: root.Participants[7].lateMiscarriage[inactive]
Time: 26.44434919280601 Entering State:      Agent: Participants[-1] State: NonPregnant
Time: 27.340240531016708      Exiting State:  Agent: Participants[26] State: LatePregnancy   Via transition: root.Participants[26].preTermDelivery[inactive]
Time: 27.340240531016708      Exiting State:  Agent: Participants[26] State: Pregnancy       Via transition: root.Participants[26].preTermDelivery[inactive]
Time: 27.340240531016708      Entering State:      Agent: Participants[26] State: PostPartum
Time: 27.340240531016708      Entering State:      Agent: Participants[26] State: AfterDelivery
Time: 27.340240531016708      Agent: Participants[26] Receive: ChangedStateofPregnancy      From:Participants[26]
Time: 28.82614139100408 Exiting State:  Agent: Participants[24] State: LatePregnancy   Via transition: root.Participants[24].losttofollowupLatePregnancy[inactive]
Time: 28.82614139100408 Exiting State:  Agent: Participants[24] State: Pregnancy       Via transition: root.Participants[24].losttofollowupLatePregnancy[inactive]
Time: 28.82614139100408 Entering State:      Agent: Participants[-1] State: finalState2
Time: 29.61993067098198 Exiting State:  Agent: Participants[17] State: LatePregnancy   Via transition: root.Participants[17].preTermDelivery[inactive]
Time: 29.61993067098198 Exiting State:  Agent: Participants[17] State: Pregnancy       Via transition: root.Participants[17].preTermDelivery[inactive]
Time: 29.61993067098198 Entering State:      Agent: Participants[17] State: PostPartum
Time: 29.61993067098198 Entering State:      Agent: Participants[17] State: AfterDelivery
Time: 29.61993067098198 Agent: Participants[17] Receive: ChangedStateofPregnancy      From:Participants[17]
Time: 29.66298160880525 Exiting State:  Agent: Participants[17] State: NotWorried      Via transition: root.Participants[17].transition6[inactive]
Time: 29.66298160880525 Entering State:      Agent: Participants[17] State: NotAnxious
Time: 29.66298160880525 Entering State:      Agent: Participants[17] State: Worried
Time: 29.66298160880525 Agent: Participants[17] Receive: ChangedStateofStateofMind      From:Participants[17]
Time: 30.36437035091798 Exiting State:  Agent: Participants[55] State: NotWorried      Via transition: root.Participants[55].transition6[inactive]
```

Figure 3.4: The Trace Log

The trace log provides a modeler with the following information:

- The *sequence of actions indexed by model time* — The model scenarios executed simulate the evolution of a system over time, and the trace log not only provides the information about each activity, but also the time within the simulation when the activity takes place.

- The *change of state* of an agent — Agent based models comprises of statecharts that portray the various states or phases experienced by an agent over the span of the execution. The model's "Main" is itself represented as an agent within AnyLogic. Thus throughout the execution, agents enter a particular

state, stay for some time, exit that state and then enter a new state. The trace log captures the entry or exit of an agent from each of the states.

- The *transitions* responsible for the change of state — Agents move from one state to another via a particular transition. This information is also captured by the trace log.
- The *sending and receiving of a message* — Certain actions in agent based modeling are driven by messages. The trace log captures these messages along with the source and recipient of the message.

The table 3.1 provides a mapping of the items in the Trace Log with the information it provides. The table helps in understanding what information each line item in the trace log provides. Primarily, four kinds of line items are being traced in the sequence of the flow of the model.

Example from a Trace Log	**Information provided**
Time: 20.0 Agent: Participants[0] Send: ChangedStateofPregnancy To: Participants[0]	1. **Time:** provides the model time 2. **Agent:** indicates which agent is sending 3. **Send:** provides the particular message being sent 4. **To:** indicates which agent is receiving
Time: 20.0 Agent: Participants[0] Receive: ChangedStateofPregnancy From:Participants[0]	1. **Time:** provides the model time 2. **Agent:** indicates which agent is receiving 3. **Receive:** provides the particular message being received 4. **From:** indicates which agent is sending
Time: 42.0 Entering State: Agent: Participants[0] State: PostPartum	1. **Time:** provides the model time 2. **Entering State:** indicates the act of state entry 3. **Agent:** indicates which agent is entering 4. **State:** indicates the name of the concerned state
Time: 102.0 Exiting State: Agent: Participants[0] State: AfterDelivery Via transition: root.Participants[0].subsequentPregnancy[inactive]	1. **Time:** provides the model time 2. **Exiting State:** indicates the act of state exit 3. **Agent:** indicates which agent is exiting 4. **State:** indicates the name of the concerned state 5. **Via transition:** indicates the name of the transition which facilitated the agent to exit the state

Table 3.1: Mapping of Items in the Trace Log with the information it provides

The aim of the trace log is to provide information concerning behind the scene action of a particular execution. It addresses questions like what, when, how and which with respect to the execution. It logs *what* happened, for instance, a movement by an agent from one state to another; *when* a particular activity took place, that is, the model time at the beginning of each activity; *how* a particular activity took place, that is, transitions that were responsible for the movement of the agent and *which* other agents were affected. The

detail provided by the trace log is important for understanding the flow of the model and to determine if the model flow is per expectation. This helps in better understanding the developed model, as a consequence of which pointing out irregularities or defects becomes easier. Therefore the trace log fulfills its other aim which is to simplify the complex process of model verification and debugging.

3.3.3 Output Canvas

The logging and tracing tool provides two snapshots of the output canvas of the AnyLogic framework, one displaying the model visualization of the model's initial state (i.e., when the model starts executing) and the other displaying the visualization of the final state of the the simulation model. Figure 3.5 shows an example of the initial and the final state of the Output Canvas related to a particular model.

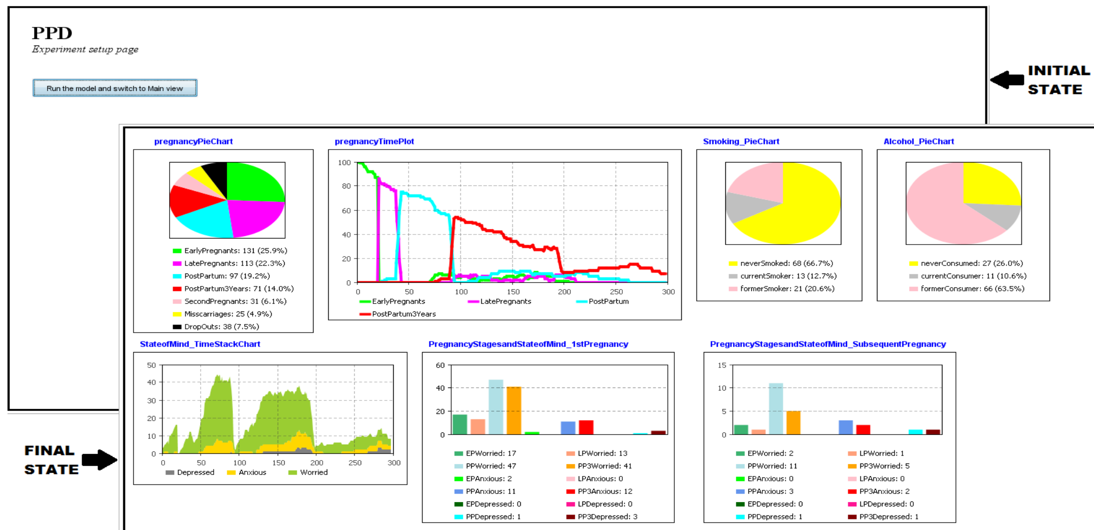


Figure 3.5: Initial and Final state of the Output Canvas.

These two images are automatically captured each time an experiment is run. This information can help modelers and users to revisit visual outcomes from a previous run. This can also avoid unnecessary re-execution when graphical analysis on a previous execution is required. It also helps modelers to compare between the present outcome and the previous one, if necessary.

3.3.4 Copy of the Model

The logging and tracing tool stores a copy of the model (.alp file in our case) that is run in the particular execution. Figure 3.6 shows an example of how it is stored in the system.

Figure 3.6: Copy of the Model

The name of the file consists of the name of the model and the word “copy” to denote that it is a copy of the model named there. A copy of the model is required to preserve the version of the model used in that particular execution. Developing a simulation model is an incremental process, it is also a complex task and as such the modelers keep making slight modifications of the model in quest of better results. Thus, storing the precise version of the model used in a given simulation is extremely important in order to re-create the exact output of the model in the future.

3.4 Logging and Tracing Tool Configuration

The structure of the logging and tracing tool is presented in figure 3.7. The tool actually consists of two folders — one named *lib* and consisting of the library files, and the other named *classes* and containing the compiled AspectJ (“.class”) files.

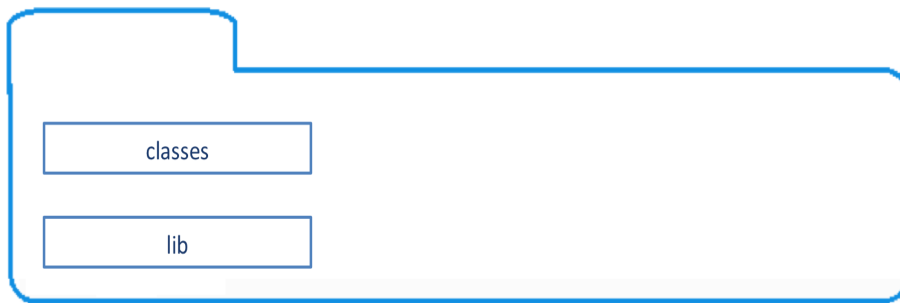


Figure 3.7: The Run Log and the Trace Log.

The configuration process is simple and can be conducted in three steps as follows:

- First, the two folders need to be copied into the project folder containing the model which will utilize this tool.
- Second, the classes folder needs to be added to the model in AnyLogic. AnyLogic allows the addition of jar files and class folders in the “Dependencies” tab of the model.
- Finally the JVM command-line argument for each experiment or model scenario related to the model needs to be updated with the java command — *-javaagent:jarpath[=options]*. More specifically, the *Java Machine Arguments* field found in the *Advanced Java* tab for each experiment of an AnyLogic model needs to be updated with the following java command — *-noverify -javaagent:./lib/aspectjweaver.jar -cp ./lib/aspectjrt.jar:./lib/aspectjtools.jar*. It is important to note that some jar files are accessed from the *lib* folder, so the organization of the folders are important. This command will work fine as long as the model file and the *lib* folder reside in the main project folder. If the user or the modeler change the folder structure, they need to modify these Java Virtual Machine Arguments accordingly.

.

Once the tool is configured for a particular experiment, every execution of that experiment can potentially generate logs, images and model copy if the user or the modeler does not choose to disable the tool in the JOptionPane shown in figure 3.2 that prompts user confirmation. The choice of the user is reflected in the console in the following way:

- When the "No" button is clicked, the message in the console reads as 'NO' button clicked, hence logging and tracing tool is *not* enabled.
- When the "Yes" button is clicked, the message in the console reads as 'YES' button clicked, hence logging and tracing tool is enabled.
- When the window is closed without clicking any button, the message in the console reads as JOptionPane closed, hence logging and tracing tool is enabled by default.

3.5 Logging and Tracing Tool Functionality and Service

In the earlier sections, we focused on the functionality of individual outputs from the tool and discussed how each of these help serve modelers. In this section, we will highlight the overall benefits provided by the logging and tracing tool. The contributions of the logging and the tracing tool in the context of simulation modeling can be summarized as follows:

- It enables the re-creation of the exact same model output in the future.
- It functions as an automated process to generate information important for model verification, testing and debugging.
- It functions as an automated process to preserve and store information in a convenient location.
- It serves as an aid in enhancing the analytical process.
- It serves as an aid in simplifying the debugging process.
- It helps in elimination or reduction of rework (rerun of the same experiment) by providing reference documents.

Chapter 4

Experiments and Evaluations related to the Logging and Tracing Tool

We carried out a series of tests to evaluate our logging and tracing tool. In broader terms, the goal for having such a tool is three-fold — the first being the capability to reproduce simulation results, secured by preserving the seed (random number seed) value to be used for recreating output, the second being the auto-generation of a set of documents that we deem necessary for improving the flexibility and modularity of simulation models, and the third being improvement in the clarity of understanding as well as easy detection of errors. We have carried on our experiments with two agent-based models and a group of modelers and users to informally assess whether all the purposes of creating such a tool are served well. Our evaluations are based upon visual results, concrete output as well as comments from modelers and end-users. In this section we will discuss in detail the models involved and the experiments performed, followed by results and responses obtained.

Experiments

Most of our experiments were carried out with two simulation models. We will discuss about each of these in the following sections. In Table 4.1, we highlight the various scenarios that were associated with our experimental procedures. Several variations among the simulation experiments are required to confirm that the outputs are in accordance with the expected results. The "Distinctive feature" column of Table 4.1 presents the various choices or features that are specific to a particular experiment or model scenario. Most of the experimental results presented here have been published in [9].

4.1 Experiment with the MinimalistSIRNetworkABM Model

The MinimalistSIRNetworkABM is an existing agent based simulation model written in AnyLogic [33]. This model simulates the network-mediated spread of an infection in a population. Agent behaviour is specified by a statechart comprised of three states — Susceptible (exposed to infection), Infective (infected and contagious) and Recovered (having recovered from the illness, and now immune to it). The model contains three simulation scenarios based on population size. In the "UnspecifiedPopSize" experiment, the population

Name of the Model	Name of the Simulation	Distinctive Feature
MinimalistSIRNetworkABM	UnspecifiedPopSize LargePopulation SmallPopulation	Parameter populationSize differs from unspecified through large to small.
Post-Partum Depression	LargePopulation MediumPopulation SmallPopulation	Fixed seed Random seed generated by the model

Table 4.1: List of experiments conducted [9]

size is not specified in the model parameters, the modeler specifies it while executing the experiment. The "LargePopulation" scenario is created to account for a large population; in this model, the population size is statically set to "1000". The "SmallPopulation" scenario is created to account for a small population, and the population size is set statically to "10".

We enabled our logging and tracing tool in the MinimalistSIRNetworkABM model by updating the JVM argument of each of the three experiments. Then each of the experiments were executed to validate results. We found that —

- Each simulation run created a sub-folder related to that run. For example, the name of one such sub-folder is *Run2015.12.22_03.42.29_PM_EST*.
- The sub-folder *Run2015.12.22_03.42.29_PM_EST* in turn contains :
  - The run log — LogFile2015.12.22_03.42.29_PM_EST,
  - The trace log — TraceFile2015.12.22_03.42.29_PM_EST,
  - The initial state image — canvasCaptureInitial1450816949846,
  - The final state image — canvasCaptureFinal1450816949846, and,
  - The model copy — MinimalistSIRNetworkABM_copy.alp.
- We also validated each output to verify the correctness of the results. All the information were as per our expectation.

4.2 Experiment with the Post-Partum Depression Model

The Post-Partum Depression model is a newly created model developed by a group comprising of Computer Science researchers and Public Health researchers. The Post-Partum Depression model deals with pregnancy and childbirth. The model runs for a period of four years and all the agents consist of pregnant women. The model simulates the feeling of a new mother who sometimes suffers from depression due to many factors during her stages of pregnancy, the development of Post-Partum Depression among them, and finally examines

the effect of subsequent pregnancies on mothers already suffering from post-partum depression. The agent behaviour in this model consists of four statecharts. Specifically, the *PregnancyStateChart* depicts several states of pregnancy, such as Early Pregnancy, Late Pregnancy, Subsequent Pregnancy, etc.; the *StateofMind* statechart depicts the mental condition of the mother, including various states of mind during pregnancy, such as worried, not worried, anxious, depressed, etc.; *SmokingStateChart* and *AlcoholStateChart* dealing with smoking and alcohol consumption behaviour of both expectant and new mothers. The model contains three simulation scenarios based on the number of mothers involved in the experiment. In the *LargePopulation* experiment, the model contains "600" agents or new mothers; in the *MediumPopulation* experiment, the model contains "300" agents or new mothers, whereas in the *SmallPopulation* experiment, the model contains "100" agents or new mothers.

In order to verify that our expectations related to the benefits derived from documents generated by the tool are met, the logging and tracing tool was provided to the group developing the Post-Partum Depression model. Their comments and feedback on the logging and tracing tool helped us with our evaluation of the tool. Several vignettes [9] are included in the following sub-sections, where we describe how the components generated by the logging and tracing tool proved beneficial for solving various problems faced by the modelers while developing their simulation model. In addition to this, assessments were also conducted to illustrate how the logging and tracing tool addresses several concerns.

4.3 Experimental Observations

The experiments conducted so far, with several scenarios of two different models, provided evidence for the model independent nature of the tool. These experiments helped us to assess if correct information is being reflected in the Run Log and the Trace Log. They also helped us to observe closely how this framework aids the debugging process. The Run Log and the Trace Log have been instrumental in providing additional facilities to modelers or users. For our evaluations, we compared the results from the new modeling package (consisting of the simulation model along with the tool) with the traditional simulation modeling (without our tool) and found the following advantages of the former over the latter:

- The new modeling package offers enhanced understanding
- It enables re-creation of output
- It improves the insight gained from models
- It helps in improving analysis capacity
- It helps with debugging
- It minimizes rework due to re-runs
- It eliminates cluttering of console due to print statements (for tracking purpose)

In the next few sub-sections, we have discussed how the advantages we mentioned here have been acknowledged by some modelers.

4.3.1 Role of run log, trace log and output canvas as future reference guide

All these three components serve as reference documents for the future. The run log contains important details related to the model and scenarios while the trace trace log contains important details related to the temporal evolution of a scenario. The output canvas displays the final state of the simulation. All these documents are valuable for conducting analysis in the future and hence they serve as future reference documents. Two examples of its usage by the group developing the Post-Partum Depression model are provided below:

- In the testing phase, when different tests were being carried out with the model, the Run Log helped automatically document all of the defining details pertaining to each execution. It relieved the modelers from the burden of manual documentation and also from the risk of important information being lost in the process. They were also equipped with a Run Log to share with their supervisor for review.
- The Run Log, the Trace Log and the screenshot of the canvas were used by the modelers many times to debug and to analyze a particular situation.

4.3.2 Role of the trace log for clear understanding and debugging

The Trace Log is a detailed document containing information about the sequence of actions taking place in the model. As such, it helps improve clarity and understanding. Since the Trace Log has the capacity to clarify intricate details of model execution without making any change in the model, without requiring invasive coding of print statements and without the console from getting cluttered, it can be used as an aid to identify mistakes and find associated errors. Below, we provide two examples of its usage by the group developing the Post-Partum Depression model, firstly as an aid to understanding and, secondly, as an aid to debugging:

- While running the Post-Partum Depression model for a small population of agents, the modelers were confused to find a large number of subsequent pregnant cases, which were unexpected. So they reviewed the *Trace Log* to find out the model time when the subsequent pregnancy event started occurring in the model and also how many times it was occurring for each participant. They found that the restriction on the subsequent pregnancy event to occur just once for each participant was not working.
- The modelers initially tried to solve an issue of an unexpectedly high number of people in the *Depressed* state of the StateofMind statechart conceptually, without the help of *Trace Log.* After spending around 2.5 hours inspecting the code, they realized that they could likely find some clue in the Trace Log. They inspected the Trace Log and uncovered the issue within 15 minutes (this issue is described below). From

then onwards, whenever the modelers found an unexpected output, they used the Trace Log to identify exactly what was happening. They were thus able to figure out the error much more easily. Thus the Trace Log helped simplify the complex job of debugging.

Two instances of debugging are as follows:

1. In the Post-Partum Depression model, the ChangeofStateofPregnancy message was being erroneously sent from the EarlyPregnancy state of the Pregnancy statechart. This message was responsible for triggering a change of state in the StateofMind statechart. This resulted in the incorrect change of state in the StateofMind statechart and increased the number of people in the Depressed state of the StateofMind Statechart. This problem was identified using the Trace Log.

2. Some accumulator variables were used in the model to keep total count of individuals in different states. The individuals were being incremented upon entering the states but were not decremented upon leaving the state. This resulted in incorrect counts and was uncovered after inspecting the Trace Logs.

4.3.3 Role of the model copy and run log and output canvas for recreation of output

These components play an important role for the re-creation of model output. The figure 4.1 shows an example of a model output recreated with the help of the logging and tracing tool.

Figure 4.1: The recreation of output using the logging and tracing tool

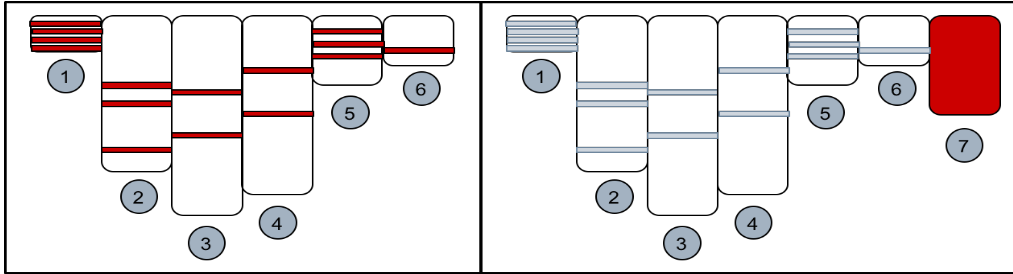


Figure 4.2: Left: Simulation model with manual methods of logging and tracing ; Right: Simulation model with the Logging and Tracing Tool

The top part of the figure 4.1 shows the final output of a model run with a random seed (the seed was randomly generated by AnyLogic). The run log, however, prints the seed used to execute an experiment. This value is later used to fix the seed for the AnyLogic simulation and the model is executed again. The output is exactly the same in both the cases.

An example of such an experience by the group developing the Post-Partum Depression model is also provided below:

- The model team had to re-create output on two occasions, which they did successfully with the help of the seed from the Run Log and the preserved model version as well as the output canvas of the previous run.

4.3.4 Logging and Tracing Tool Modularity Assessment

In this section, we will focus on the benefits of using aspects in the context of modularity. As noted in 1.2, the primary goal of this thesis is to enhance the modularity of simulation frameworks while addressing the issues of auto-documentation and auto-storage. This section is being utilized to assess modularity with aspects vs. that obtaining absent aspects. In order to illustrate the improvement of modularity, two sets of figures, demonstrating the simulation model with aspects (that is, with the Logging and Tracing Tool) vs. the simulation model without aspects (that is, with manual methods of logging and tracing), have been generated.

The first set of modularity assessment figure, figure 4.2 is inspired by the standard aspect visualization tool available with the AspectJ compiler in Eclipse. The figure has been created in the same way with some changes, or rather, additions. The modifications had to be manually incorporated as the figure generated by the visualization tool lacks the complete picture. This is because the codes auto-generated by AnyLogic usually contains the model-specific customized classes like *Main*, *Person* and *Simulation* (for our model). There are other library source codes, viz., *com.anylogic.engine.Engine*, *com.anylogic.engine.Agent*, *com.anylogic.engine.Experiment* and *com.anylogic.engine.ExperimentSimulation* which are used by the model and have been intercepted by the join points of the Logging and Tracing Tool but they do not get reflected in the figure generated by the visualization tool.

The left half of the figure 4.2 represents the simulation model without aspects, or rather, with manual methods of logging and tracing. The simulation model used for this figure has been extensively used for the experiments conducted with the logging and tracing tool. It is the Post-Partum Depression model and is discussed in detail in section 4.2. The left side of the figure 4.2 displays six cylinders, each representing a class. The red stripes represent the different methods of the classes affected by the tool. While assessing modularity for the logging and tracing tool via aspect visualization, we found that the concern (crosscutting concern that can be efficiently managed by aspects) is spread across different locations within 15 methods of 6 classes. In the figure 4.2, number "1" represents the class SmallPopulation (Simulation for this model), number "2" represents the class Person, number "3" represents the class Engine, number "4" represents the class Agent, number "5" represents the class Experiment and number "6" represents the class ExperimentSimulation.

The right half of the figure represents the simulation model with aspects associated with with the logging and tracing tool. The figure displays all the cylinders representing classes present in the left half, but the red stripes replaced by grey stripes here (representing displaced code) and an additional red cylinder represented by number "7" has also appeared. The red cylinder is actually the logging and tracing tool containing the aspect code. It is the single encapsulation containing all the codes concerned with the logging and tracing. It has extracted the logging and tracing concerns from the other six classes, as a result of which they no longer contain red stripes. It is also evident that this approach does not always decrease the lines of code for all cases, since an additional aspect is introduced here. However for our case, or any model with a large number of states and multiple statecharts, this approach saves considerable redundant coding as well. But this approach does improves modularity and maintainability, since now there is a single location which is concerned with the logging and tracing concern, unlike the multiple locations in six different classes. Any change in any of the six classes concerned with other model-specific business logic or the aspect concerned with the logging and tracing will no longer affect one another. For instance, if a new Person class is added in the model, the logging and tracing tool will support the new class in the same way without any changes.

The second set of modularity assessment figure, figure 4.3 depicts the voluminous amount of coding required when the functionalities of the logging and tracing tool are implemented manually in a simulation model in AnyLogic. While the logging and tracing aspects can be reused across different models, the manual coding has to be custom-built for each model. The figure shows the different locations where lines of code need to be inserted.

In the main class (the orange box in the figure), there is a need to represent variables to refer to the output files — the log file and the trace file – and additional purposes, such as the date, need to be added. In the simulation class (the green box in the figure), lines of code need to be inserted in four different locations highlighted in the figure with red boxes. These are :

- *Initial experiment setup* — executes the initial setups of the model when the experiment is created, impacts the method *setup(java.awt.Container container)*.
- *Before each experiment run* — executes before the experiment begins, impacts the method *reset()*.

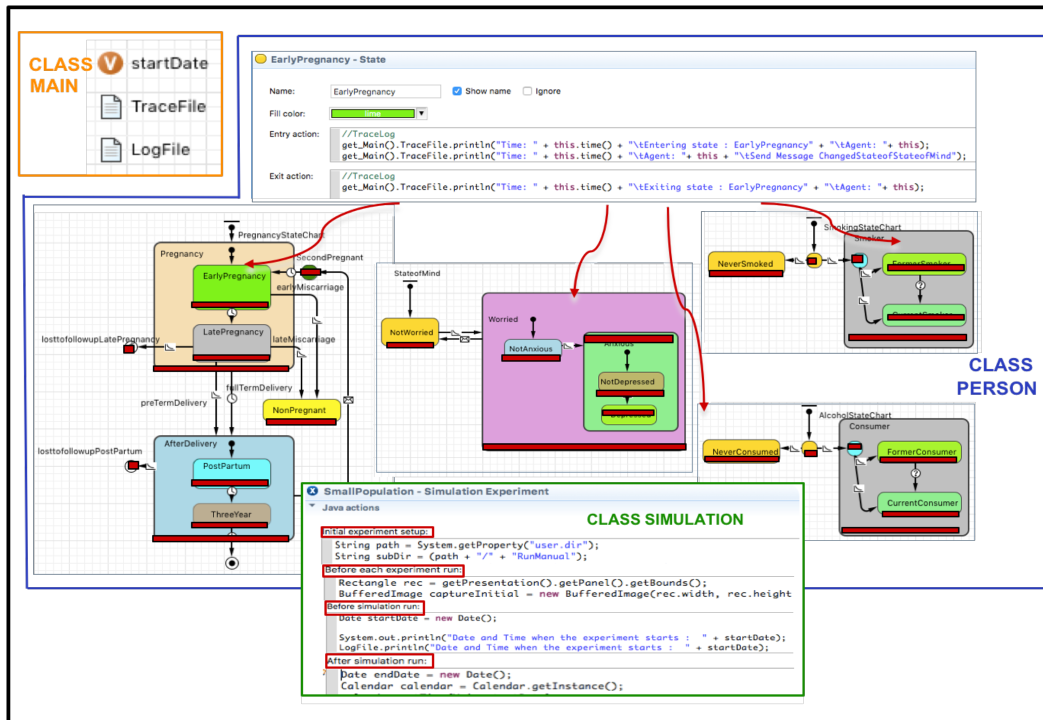


Figure 4.3: Coding required for implementing logging and tracing without aspects

- *Before simulation run* — executes before the model starts running, impacts the method *onBeforeSimulationRun(Main root)*.
- *After simulation run* — executes when the model execution is completed or the engine has stopped, impacts the method *onEngineFinished()*.

The person class (the blue box in the figure) is the most affected class. The same lines of code need to be inserted in the *entry action* and the *exit action* of each and every state of the statecharts present in the model in order to trace the flow of the model. The code in the *entry action* is executed while the agents enter that state, whereas the code in the *exit action* is executed while the agents exit that state. The red stripes and dots represent the locations where these lines of code need to be inserted. If any one location is missed out, then tracing will be incomplete. On the other hand, if a new statechart is introduced, then the same coding needs to be done for the new member as well. In order to have an idea about the extent of affected modules or locations in case of manual tracing, we looked into example models in AnyLogic and found that the agent based models typically contain between 1 to 10 statecharts with 2 to 73 states and 2 to 77 transitions. Our example model contains 4 statecharts with 29 states. Our example model, therefore, is one of the smaller models. The problems with manual process grows with the size of the model, as more states or statecharts mean more affected locations. The manual process requires an individual to write the tracing code in 58 (29 *2) different locations. This is a quite cumbersome and exhaustive process which could lead to errors quite easily. This process is also dependent on individual memory and compliance, as frequent

revisits and revisions are required at the onset of minor changes made in the model. When the code gets scattered across different locations, the governing principles (rules) for doing that are not clear. But if the code is modularized in an aspect, those rules for where to weave it in are explicit. Further, this large and scattered amount of work is model-specific and model dependent. Coding needs to be introduced into each model when adding the logging and tracing functionality to the model.

It needs to be emphasized here that despite following such an exhaustive process, it is difficult to obtain certain information via the manual process. One example is the random number seed. It is difficult to retrieve the value of the random number seed in the manual approach.

Chapter 5

The MCMC Tool

The second tool that we developed also uses the concepts of aspect oriented technology and can be applied easily on a dynamic model built in the AnyLogic framework to generate posterior distributions of parameter(s) applying the MCMC technique. The MCMC tool is capable of running MCMC on any model parameter(s). This consists of a User Interface (UI) that is dynamically customized to the model on which it is run in order to enable the user to select the (unknown) parameter(s) on which he/she wishes to run MCMC. Unlike the tracing and logging tool, the MCMC tool requires running the model in successive iterations. For this reason, the MCMC tool re-purposes the standard Parameter Variation experiment (the scenario that allows the model to run multiple iterations) type of the AnyLogic simulation modeling framework. The JVM argument section of the Parameter Variation experiment for any model needs to be updated in order to use the tool.

Our goal in developing the MCMC tool is to provide information about unknown parameter(s) by sampling from a generated posterior distribution, that can be valuable for modelers dealing with systems comprised of insufficient empirical data. In the 1.2 section, we already discussed how modelers use alternate methods or techniques to gather information about parameter(s) whose values are poorly known, and the challenges they face while working with traditional simulation or parameter estimation methods. Thus, we explored an approach utilizing the combination of dynamic models and Markov Chain Monte Carlo Techniques (MCMC) to try to address the challenges faced by modelers in this context. We will use this section to discuss the approach, design, implementation details and each component of the MCMC Tool. Additionally, we provide details of the output obtained from the tool and the information it contains. Like the underlying MCMC methods, the MCMC tool is designed for use only with deterministic models.

5.1 MCMC Tool Design

We used AspectJ 8 and AnyLogic version 7.1.2 for the development of the MCMC tool, and Java Swing framework to develop the user interface part of the tool. Several Java features like creating the files, writing in a file, have also been utilized. Apart from these, we made strong use of the Java Reflection API in ways mentioned below in order to maintain the generic nature of the tool:

- The UI uses reflection to identify the parameters of a particular dynamic model in order to allow the

user choose parameter(s) among the corresponding names of parameters within the UI.

- The tool uses reflection to find the location of the model file so that the other output files can be placed there.

- The tool uses reflection to invoke functions of the model such as logPosterior.

The design phase of the MCMC tool involved the identification of join points responsible for activities common to all dynamic models created in AnyLogic so that when used with similar dynamic models built in AnyLogic, it behaves in the same manner. With those join points identified, pointcuts were derived in order to select these join points. Finally, a set of advice is defined to carry out the functions required by our tool.

As mentioned earlier, the technique we employ to implement MCMC with dynamic models involves combining a dynamic model with a probabilistic model, specifying a prior distribution defined over parameter(s), as well as a likelihood function that specifies the likelihood of observing empirical data considering the underlying values of the same parameter(s). The MCMC component of the framework relies upon following elements:

- The dynamic model responsible for generating model outputs.

- The probabilistic model, which is statistically Bayesian in nature. This model is responsible for computing the likelihood for the empirical data given the set of dynamic model outcomes. Calculating the likelihood almost always requires a simulation and simulation model results given those parameter values.

The aspect-based MCMC tool comprises of three broad components —

The first component consists of the User Interface which provides some instructions to the user and is mainly used to accept user inputs.

The second component of the tool is concerned with the MCMC implementation in a generic manner. As such it can be applied to any dynamic model (built in AnyLogic) containing the suitable functions required for MCMC. This is the most crucial part of this tool and is discussed in detail in the next sub-section.

The third component of the tool comprises of the trace plot to display the live results of the parameter distributions as iterations progress. In each traceplot, the x-axis indicates the number of MCMC iterations while the y-axis indicates the value of a specific parameter for a particular iteration. The live results of the parameter distribution enable users to see how the parameter varies in parameter space and how equilibrium is reached after a certain time. The capacity to dynamically monitor the sampling using traceplots is a substantial benefit of the approach proposed here, as most other current MCMC tools lack interactive traceplots.

5.2 MCMC Tool Implementation

This section provides details related to the development and the implementation of the MCMC Tool. While offering a brief overview of how the tool fits in the context of a model in AnyLogic, the section goes on to discuss the individual components of the tool.

5.2.1 Implementation Overview

We have developed a tool based on aspect oriented technology that can be used directly with the dynamic models built in AnyLogic. The tool is capable of running MCMC on selected model parameter(s). The pictorial representation of the implementation overview of the aspect-based MCMC tool is presented in figure 5.1. The pictorial representation of the MCMC tool indicates that it is an aspect-based framework containing a user interface and utilizing the concepts of Markov Chain Monte Carlo methods to generate a posterior distribution of the user selected elements (in this case, one or more parameters).

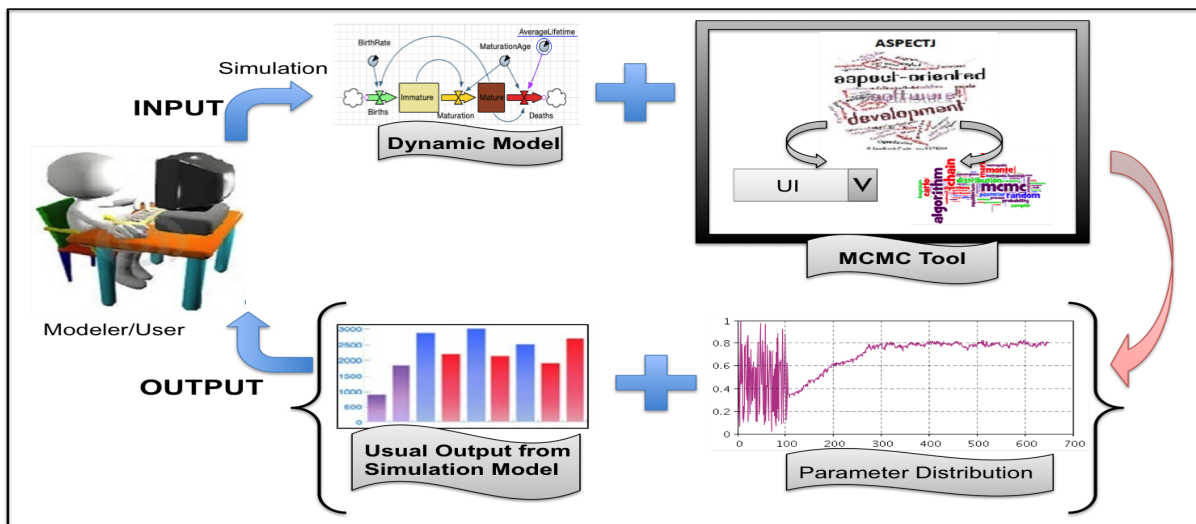


Figure 5.1: Implementation Overview of the Aspect-based MCMC tool

[8]

In this figure we see a modeler executing a dynamic model with our tool. The "simulation" represents a particular experiment of the dynamic model. When the aspect-based MCMC tool is applied to the dynamic model, a new UI window pops up as soon as the experiment is started, and new output is generated in addition to the usual output from the dynamic model. The UI component of the tool allows the user to select parameters over which MCMC should be run. The tool then uses the values entered by the user to run MCMC on the user-selected parameter(s). The MCMC component of the tool samples from the posterior distribution of the selected parameter(s). The graphical component of the tool represents this distribution in trace plots. Finally, a dataset containing the values of the selected parameter(s) is also generated. The

dataset is located is the same folder where the model (.alp file) resides in the computer.

The individual components of the tool are discussed in detail in the next few sub-sections.

5.2.2 GUI for User Input

Figure 5.2 presents the image of the UI which is an integral part of the MCMC tool. Some important points related to the UI are mentioned below.

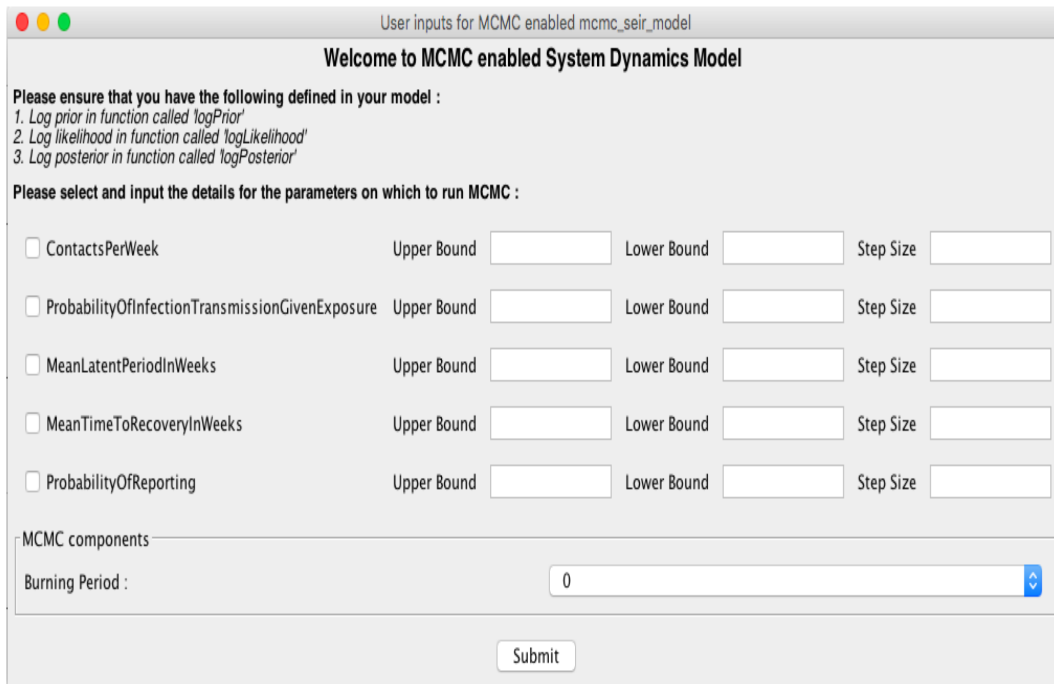


Figure 5.2: User Interface of the Aspect-based MCMC Tool

[8]

- This window pops up as soon as the dynamic model in AnyLogic is executed.
- The name of the model is picked up by our tool via Java reflection mechanisms and is displayed in the window title.
- The MCMC computations have made use of user-defined functions with known names that return components like the log of the prior, and the likelihood and posterior density functions given specified values of parameter(s).Since the dynamic models using this tool need to define these functions within the model, this instruction is provided in the UI window.
- All parameter(s) used in the Main class are displayed in the UI window for users to make a choice about the parameter(s) to be used for sampling.
- The user must also enter the upper and lower bounds to use for the unknown parameter(s) and the standard deviation they prefer to use in the random walk.

- The user needs to enter the *burn-in period*.
- The maximum number of iterations sought can be varied by entering the suitable number in the *Number of runs* field in the Parameter Variation experiment. The experiment can be stopped prior to this limit.

5.2.3 MCMC Tool Model Requirements

The MCMC tool is designed to be used with any dynamic model built in AnyLogic. While the MCMC tool can be used with several types of dynamic models in AnyLogic, once again we stress that it is only meaningful to use it with deterministic models. Specifically, the model must include the following so that the MCMC tool can be used.

- Java Virtual Machine argument section in AnyLogic should be updated to reflect the desire to use this tool.
- The model must include specifically named functions to specify the log prior, log likelihood and log posterior functions. If these functions are not available, the tool throws errors.
  - The log prior function must be defined in a method in the Main class called logPrior declared within the dynamic model. This function reads the parameter values from the current instance of the Main class and returns the log of the prior value evaluated at those parameter values.
  - The log likelihood function must be defined in a method in the Main class called logLikelihood that needs to be declared with the dynamic model. This function also reads the values of the parameter(s) from the Main object and returns the log of the likelihood value evaluated at those parameter values.
  - The function logPosterior must be defined as the posterior function. This function computes the log of the posterior value based on the sum of the log of the prior and the log of the likelihood values returned by the previous two functions.
- The modeler or user also needs to include trace plots in the dynamic model in order to plot parameter values.

5.2.4 MCMC Tool Framework

The MCMC Tool is built in the AspectJ framework. The code makes extensive use of java reflection, Java Swing and various other java packages and AspectJ. The entire code of the tool is encapsulated within the following aspect:

```
/*
 * Aspect to capture the various functionalities of the simulation code in
 * order to enable the actions required for the MCMC Tool
 */
public aspect MCMCTool
{
...
}
```

Listing 5.1: Aspect encapsulating the code for the MCMC Tool

At a technical level, the main functions of the tool can be categorized into the following:

- Get user response as to whether to enable or disable the tool. This is accomplished using an aspect-created JOptionPane from Java's awt to prompt user confirmation as to whether to enable or disable the tool. The user confirmation window is represented in figure 5.3.

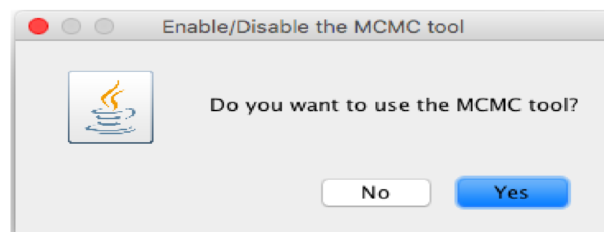


Figure 5.3: Enable/Disable the MCMC Tool

All other functions are carried on only if the user elects to enable the tool.

- Get the name and location of the simulation model being executed.
- Get the names of the parameters of the main class.
- Generate the User Interface window of the tool during the initial setup phase of the model, that is, just before the execution starts.
- Ask for user inputs via the aspect-created user interface (as described in section 5.2.2) and store the same to be used for the MCMC computation.
- Capture each iteration of the simulation to perform the following actions:
  - Generate new candidate value(s) by perturbing the current parameter value(s) for the chosen parameter(s) to be sampled.
  - Set the model parameter(s) with the new candidate value(s).
  - Run the model and generate model output.

– Compute the posterior for the new candidate value(s) by invoking the posterior function provided within the model. This further uses the prior and the likelihood functions, also provided within the model.

– Determine if the new candidate value(s) is (are) accepted (Acceptance has been discussed in greater detail in 6.1.2). If so, set the new parameter value(s) as the corresponding candidate value(s); if not, set the new parameter value(s) as the old parameter value(s) from the previous iteration.

– Write the new parameter value(s) in the output file. The output file thus contains sampled values of the chosen parameter(s) over several iterations.

- Capture the instant when the simulation model completes executing. The mean, standard deviation and credibility intervals are computed and written at this point.

In order to achieve each of the above, a set of pointcuts and advice have been defined, and are listed below. Please note that in most of the cases original code has been replaced by pseudo code within an advice in the code listings.

1. Capture the parameters and generate the UI window

   (a) The following pointcut *initialSetUpUI* is used to capture the execution of the method *setup*. This method is called from the main method of applications after the experiment is constructed, and handles the initial setup of the simulation. In the MCMC tool, this pointcut helps to intercept the join point where the tool locates the location of the simulation model being executed. Moreover, the tool finds the names of the parameters via Java reflection and generates the UI window displaying the parameter names at this juncture.

```
pointcut initialSetUpUI(java.awt.Container container):
2  execution(* *.setup(..)) &&
   args(container);
```

Listing 5.2: Pointcut to generate the UI window

   (b) The following after advice advises the join point captured by the pointcut *initialSetUpUI* to grab the initial setup of the model. At this point, the pointcut tracks the location of the model and generates the UI window so that the user inputs can be entered there. The pointcut further finds the names of the parameters and displays the same in the UI window generated, so as to allow the user to indicate what parameters should be sampled.

```
1 after(java.awt.Container container): initialSetUpUI(container)
  {
```

```
        findModelLocation();
    getParameters();
    generateFrame();
}
```

Listing 5.3: Advice for capturing the UI window

2. Set the parameters

   (a) The following pointcut *setParameter* is used to capture the execution of the method *setupRootParameters*. This method is called to setup the parameters for the main class and is called for each iteration. In the MCMC tool, this pointcut is used to intercept the join point where the parameters of the main class are assigned the values generated by the tool.

```
pointcut setParameter(final Agent self, int index, boolean callOnChangeActions)
:
execution(* *.setupRootParameters(..)) &&
args(self, index, callOnChangeActions);
```

Listing 5.4: Pointcut to set the parameters

   (b) The following after advice advises the join point captured by the pointcut *setParameter* to assign the model parameters with the values generated by the MCMC tool. Until a nonzero posterior or a defined posterior is found (as we know the log of zero is undefined and we are considering the log of posterior in the tool), the process of generating the initial parameter value is repeated in each iteration. The method *generateInitialTheta* generates these initial values. The method *setParameters* assigns these values to the parameters of the model to be used in the next iteration.

   6.1.2

```
after(final Agent self, int index, boolean callOnChangeActions): setParameter(
  self, index, callOnChangeActions)
{
  if (logPosteriorInitialTheta == Double.NEGATIVE_INFINITY) {
    ArrayList<Double> initialTheta = generateInitialTheta();
    setParameters(self, initialTheta.get(i));
  }
  else {
    ArrayList<Double> newTheta = generatePerturbed(oldTheta);
    setParameters(self, newTheta.get(i));
  }
}
```

Listing 5.5: Advice for setting the parameters

3. Capture each iteration of the simulation

   (a) The following pointcut *afterEachIteration* is used to capture the execution of the method *afterEachIteration*. This method is called after the experiment code for each iteration. In the MCMC tool, this pointcut is used to execute the Metropolis Hastings Algorithm to generate the posterior for each iteration.

```
1  pointcut afterEachIteration():
     execution(* *.onAfterIteration(..));
```

Listing 5.6: Pointcut to capture each iteration

   (b) The following after advice advises the join point captured by the pointcut *afterEachIteration* to compute the posterior for the selected parameters in the method *computeLogPosterior*. Once the posterior is computed, the *writeToFile* method is called which evaluates whether to accept the computed posterior or not and, accordingly, write it to the output file.

```
     after(): afterEachIteration()
2      {
         computeLogPosterior();
4        writeToFile(theta);
       }
```

Listing 5.7: Advice for capturing each iteration

4. Capture the completion of the simulation

   (a) The following pointcut *experimentCompleted* is used to capture the execution of the method **ExperimentRunFast.stop*. This method is executed when the scenario is terminating the model execution. In the MCMC tool, this pointcut is used to intercept the join points where the final results of the execution can be computed and displayed.

```
1  pointcut experimentCompleted():
     execution(* *ExperimentRunFast.stop(..));
```

Listing 5.8: Pointcut to track the completion of the simulation

   (b) The following after advice advises the join point captured by the pointcut *experimentCompleted* when all the iterations of the simulation completes, so that the final computations can be performed. The acceptance rate is computed here along with basic statistics comprising the mean, standard deviation and the credibility interval.

```
   after(): experimentCompleted()
2  {
       computeAcceptanceRate();
```

```
4     computeStatistics();
  }
```

Listing 5.9: Advice for tracking the completion of the simulation

5.3 Output from the MCMC Tool

The component that forms the backbone of the output of the MCMC tool is the MCMC results or the reported samples from the parameter distribution. The output from the MCMC Tool generates the following individual elements each time the MCMC process is applied to the dynamic model:

- Trace plots for the parameters.
- Simulation dataset capturing the successive samples from each chosen parameter.
- Output dataset containing the basic statistical computations like "Mean", "Standard deviation" and "95 % Credibility Interval" to verify the correctness of the MCMC tool.

The next few sub-sections contain the details about each output component.

5.3.1 MCMC Results

The output from the MCMC technique is the list of samples from the posterior distributions of the selected parameter(s), along with derivative elements such as plots. The computation of the posterior is based on the prior and likelihood functions defined by the modeler or the user within the dynamic model itself. The MCMC results are further represented in different forms of documents as mentioned below.

MCMC Plots

An important part of the results reported by the MCMC tool are trace plots displaying samples from the posterior parameter distribution for successive iterations. Figure 5.4 presents an examples of trace plots generated for a dynamic model using the MCMC Tool.

The trace plots help in easy visualization of MCMC results. Each trace plot displays live results as the MCMC operation progresses. One of the major benefits that our MCMC tool offers is the ability to visualize such distributions live as the results are generated. The other packages used to carry out MCMC operations (that we are aware of) do not offer such visualization of live results. Usually users need to wait for the MCMC operation to complete before inspecting the trace plots – and before securing confidence as to whether the MCMC iterations have converged. Since the MCMC tool enables the users or the modelers to see the MCMC results as and when generated, they can observe whether the results suggest convergence. If the user finds that the value of the

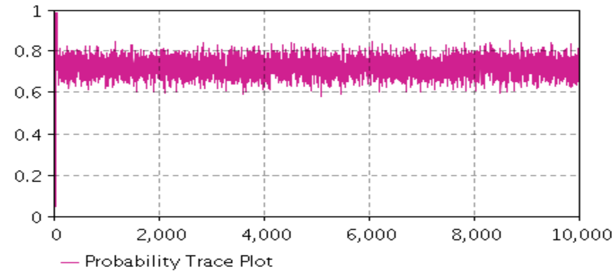


Figure 5.4: Plots generated by the MCMC Tool

unknown parameter(s) do not converge, then he or she has the option to continue the iterations to their specified upper bound. By contrast, if the user finds that the value converged much earlier than the total number of iterations specified in the experiment, he or she can terminate sampling. This is the other benefit offered by our MCMC tool. Thus considerable processing time and resources can be saved from being unnecessarily used, or re-run, and learning can take place more quickly. The current trace plots, however, do have a flaw: they include an initial sequence of iterations in which the algorithm explores the parameter space to find the initial value of parameters with non-zero posterior probability.

MCMC Simulation Data Set

An output simulation data set is generated by the MCMC tool, using the MCMC results, to document successively sampled values of the parameter(s) that are selected by the user using the tool UI. The data set is output as a csv file which resides in the project folder of the model. The name of the file consists of a combination of the name of the package followed by the string "Output_SimulationDataset". For example, the file generated for our SEIR model is *mcmc_seir_model_Output_SimulationDataset*. Every time the dynamic model using the MCMC tool is executed, this csv file is overridden to reflect the latest MCMC results. Figure 5.5 presents an example of a simulation data set generated for the SEIR model using the MCMC Tool.

The following are evident from the figure:

- Each column in this data set represents a parameter. For this particular case, two unknown parameter(s) were selected, so the data set contains two columns. Each column contains the name of a parameter along with the the successive samples from that parameter.
- The first row in the data set is the heading row containing the name of the parameters. Each successive row of the data set represents an MCMC iteration. So there are as many rows as iterations made excluding the burn-in period. For instance, if the number of iterations are 10,000 and the burn-in period is 2000, then the number of total rows in the data set is 8001 (10,000 - 2000 for the samples and 1 for the heading).

	A	B
1	ContactsPerWeek	ProbabilityOfReporting
2	300.833957	0.733269645
3	300.833957	0.733269645
4	300.833957	0.733269645
5	300.833957	0.733269645
6	300.833957	0.733269645
7	300.833957	0.733269645
8	299.5134512	0.728958167
9	299.5134512	0.728958167
10	299.5134512	0.728958167
11	299.5134512	0.728958167
12	299.5134512	0.728958167
13	299.5134512	0.728958167
14	299.5134512	0.728958167
15	299.5134512	0.728958167
16	299.5134512	0.728958167
17	299.5134512	0.728958167
18	299.5134512	0.728958167
19	296.2324724	0.738652657
20	299.9247787	0.711650129
21	299.9247787	0.711650129
22	299.9247787	0.711650129

Figure 5.5: MCMC simulation data set generated by the MCMC Tool

- It bears emphasis that the samples from the joint posterior (parameter values from the same iteration) stored in the data set can be used to run the simulation model for future simulations that can sample other values (e.g., model output values)

The csv file serves two-fold purpose in this context —

First, it serves as a data set that can be used for additional statistical analysis using software packages such as “R”, which is typically required.

Second, it helps in storing information that can be used later on for analysis or study.

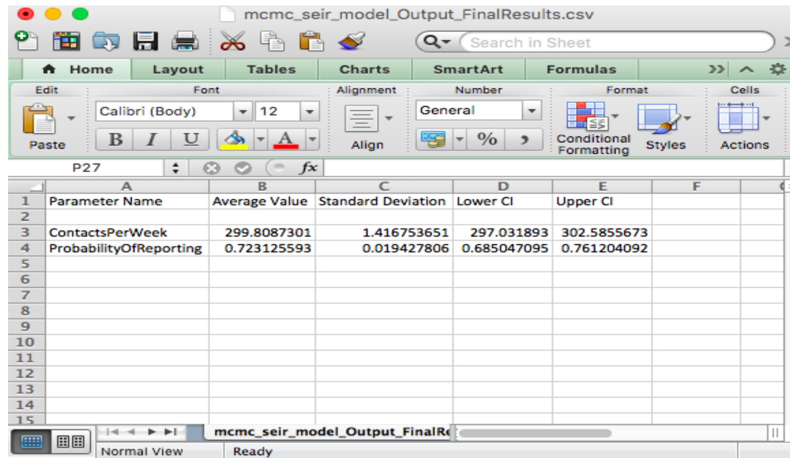
MCMC Output Data Set

An output data set containing some basic computations on the MCMC results is generated by the MCMC tool. This data set contains information useful for analyzing whether the generated MCMC results are acceptable or not. The name of the file consists of a combination of the name of the package followed by the string “_Output_FinalResults”. For example the file generated for our model is *mcmc_seir_model_Output_FinalResults*. This file also resides in the same location as the model. Figure 5.6 presents examples of such an output data set generated for the SEIR model using the MCMC Tool.

The data set contains information about the mean, the standard deviation and 95% credibility intervals for each unknown parameter. This information helps the modeler or user evaluate the correctness of the parameter distribution.

5.4 MCMC Tool Configuration

The configuration process of the MCMC tool is very similar to the configuration process of the logging and tracing tool. All the procedures followed to configure the logging and tracing tool need



Parameter Name	Average Value	Standard Deviation	Lower CI	Upper CI
ContactsPerWeek	299.8087301	1.416753651	297.031893	302.5855673
ProbabilityOfReporting	0.723125593	0.019427806	0.685047095	0.761204092

Figure 5.6: MCMC output data set generated by the MCMC Tool

to be followed here as well. It is important to emphasize again that in order to use the MCMC tool one must define the log prior, log likelihood and log posterior functions with their proper names.

5.5 MCMC Tool Functionality and Service

We have already discussed the functionality and benefits of each component of the MCMC tool in previous sections. In this section, we will focus on the overall benefits provided by the MCMC tool. The MCMC tool helps dynamic modelers or users in the following ways:

- The MCMC tool provides an easy and efficient way to run MCMC on dynamic models.
- The MCMC tool samples from the posterior distribution of unknown parameter(s), thus helping the dynamic modelers to simulate systems for which empirical data is not readily available.
- The UI component of the MCMC tool adds flexibility to the approach and empowers modelers, as they can select any compatible parameter(s) of their choice to generate the MCMC results.
- The visualizations of the live results provided by the MCMC tool helps in ongoing notification of the results to the modeler, thus saving time and resources as well as enabling the users to stop the process when they deem fit, or to continue directly when necessary to secure convergence (rather than having to re-start the process over following incomplete convergence).
- The storage of the MCMC results facilitated by the MCMC tool helps in documentation of the model run and in carrying out other statistical analysis, as is generally required.

Chapter 6

Experiments and Evaluations related to the MCMC Tool

This chapter discusses the experiments performed with the MCMC tool, results obtained and evaluations conducted on the MCMC results to assess the correctness of the MCMC results generated by our new tool. We will also discuss the workings of the MCMC tool. Both the dynamic model as well as the MCMC technique need to be explained for clear understanding of the functionality of the tool. In this section, we will discuss in detail the specifics of the dynamic model we chose for our experiment, along with how we handled the various attributes of the MCMC algorithm.

6.1 Experimental Set up

The goal of the MCMC tool is to sample from the (joint) probability distribution of a set of unknown parameters from a dynamic simulation model using the Markov Chain Monte Carlo (MCMC) technique. Our approach employs combination of a dynamic simulation model with a probabilistic model specifying a prior distribution defined over parameters, along with a likelihood function that specifies the probability of observing empirical data, given specified parameter values. We have chosen a preexisting dynamic model to illustrate the application of the MCMC tool, demonstrate experiments, validate and evaluate the MCMC operation. The model we selected is a standard SEIR model [35, 4] using the System Dynamics features of AnyLogic. In order to cross-verify the accuracy of the output of the tool, we used a model for which all parameter values were known.

6.1.1 Analyzing the SEIR Model

The SEIR model is concerned with the flow of an infection in a population divided into four categories:

- Susceptible (denoted as S), referring to healthy people within the population who can get infected.
- Exposed (denoted as E), referring to people who are latently infected but not yet infectious.

– Infective (denoted as I), referring to people who are infectious and who exert a risk of transmitting the infection to susceptibles in the population.

– Recovered (denoted as R), referring to people who have recovered from the infection.

Figure 6.1 depicts the SEIR model that we are discussing here. The left half of the figure presents the simulation model in AnyLogic (version 7.1.2), while the right half contains the pictorial depiction of the simplified version of the model containing only the important elements that are significant for our experiment.

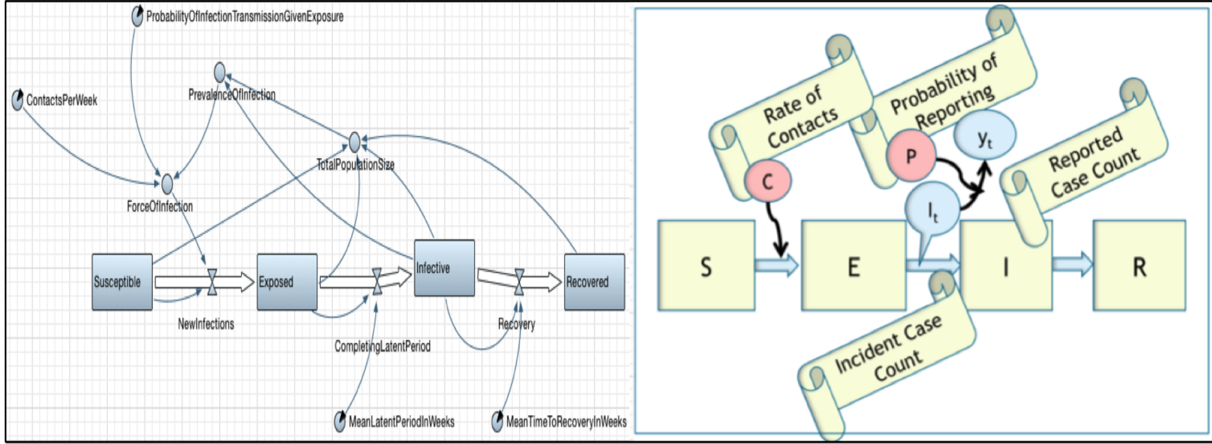


Figure 6.1: Left: SEIR model in AnyLogic ; Right:Pictorial depiction of simplified SEIR Model significant for our project

The system can be mathematically represented as a set of following state equations:

$$\begin{aligned}
\dot{S} &= -cS\left(\frac{I}{S+E+I+R}\right)\beta \\
\dot{E} &= cS\left(\frac{I}{S+E+I+R}\right)\beta - \frac{E}{\tau} \\
\dot{I} = i &= \frac{E}{\tau} - \frac{I}{\mu} \\
\dot{R} &= \frac{I}{\mu}
\end{aligned}$$

As is typical in aggregate infection transmission models, random mixing assumptions have been adopted for the model. In this model, it is assumed that each susceptible comes into contact with c individuals per week, that each contact between a susceptible and an infective results in β probability of infection transmission, thereby latently infecting the susceptible individual, that each exposed individual proceeds to the infective state following a first-order delay associated with a mean incubation period of τ, and that each infected individual recovers with a mean recovery time of μ, represented as a simple first-order delay. The model time unit is weeks.

As such, at any given point of time,

— the fraction of the population infected is given by $\frac{I}{S+E+I+R}$.

— the mean count of infectious individuals each susceptible comes across per day is given by $c\frac{I}{S+E+I+R}$.

— In a common approximation, the probability of a susceptible getting infected is treated as $c\frac{I}{S+E+I+R}\beta$.

Once a susceptible gets the infection, the individual becomes latently infected and carries the infection but is not yet capable of transmitting it. The individual will next proceed to the Infective state following a first-order delay. The rate (in persons per week) of flow from Exposed to Infective, also referred to as the weekly incident case count, is the focus of data collection, and as such is denoted as i in the model.

We have focused on the four important factors of the SEIR model that play a significant role in our experiments:

- Parameter c or "Contacts Per Week" — Parameter c denotes the number of contacts that an individual come across in a week, or in other words, the weekly rate of contacts. This is considered as the only unknown parameter from the simulation model which directly influences the number of individuals getting infected per week, as we are considering a case where infection is transmitted when a healthy person comes in contact of an infectious person.
- i_t or "Completing Latent Period" — i_t denotes the number of individuals getting infected per week, which is also referred to as the (weekly) incident case count.
- Parameter p or "Probability Of Reporting" — Parameter p denotes the probability that a particular incident case will be reported. Although parameter p is conceptually a parameter of the probabilistic model and not of the dynamic model, it is still a significant factor since it is a known fact that for many communicable pathogens, not all incident case counts are detected and reported; in many cases, the gap between the two is large. Therefore, this parameter needs to be defined in the dynamic model along with the prior, likelihood and posterior functions since this parameter has a role to play in the statistical computations of MCMC.
- y_t or "Empirical Data" — y_t is the number of reported cases, and is also referred to as the reported case count. In this model, this data is generated using binomial samples from the incident case counts over time.

The purpose of the experiments conducted on the SEIR model after applying the MCMC tool is to sample from the (joint) probability distribution of parameters c from the dynamic model and p from the probabilistic model using the Markov Chain Monte Carlo technique. Below is the corresponding parameter vector :

$$\vec{\theta} = \begin{bmatrix} c \\ p \end{bmatrix}$$

We now proceed to discuss how we handled each attribute of the random walk Metropolis Hastings algorithm. The next subsection lays down the details of the approach undertaken to implement the MCMC algorithm within the MCMC tool.

6.1.2 Handling the attributes of the Random Walk Metropolis Hastings

Initial value of parameters

We need to define an upper bound and a lower bound for all the unknown parameters, parameter p and c in this case, from prior knowledge or domain expertise. The upper and lower bounds for unknown parameters need to be specified by the user using the UI window of the MCMC tool. Before the MCMC operation commences, an initial algorithm explores the parameter space, randomly chooses an initial parameter value $\vec{\theta}_0$ and checks its posterior probability. This is repeated until a point with nonzero posterior probability is found. This is done to accelerate convergence of the algorithm.

Step size and perturbation

At each MCMC iteration, we need to perturb the current parameter value to generate a new candidate. This perturbation is implemented via a random walk using a Gaussian kernel (a draw from a Gaussian distribution) with mean 0 and standard deviation (σ) equal to step size. Since the Gaussian kernel is symmetric, we obtain $q(x|x') = q(x'|x)$. Thus, the acceptance probability of the Metropolis Hastings Algorithm reduces to :

$$\alpha = \frac{\tilde{p}(x')}{\tilde{p}(x)}$$

In our model σ is a free vector parameter. Similar to upper and lower bounds, the step size of a parameter to be sampled also needs to be specified by the user using the UI window. Since we have access to the upper and lower bounds, we can conduct an additional check of whether the new candidate is within these bounds or not. If not, then a new candidate is re-generated. This resampling continues until a candidate within the user-specified bounds is found. This reflects the fact that the posterior value for a candidate not within these bounds will definitely be zero and, hence, the proposal will not be accepted. For our experiments, we fine tune this value manually so as to obtain an *acceptance rate* (defined later) of 25% or higher [13]. For our experiments, we have used a step size of 10 for the parameter c and a step size of 0.001 for the parameter p.

Prior

We fix a prior distribution $p(\vec{\theta})$ for the unknown parameters from prior knowledge, domain expertise or from "best guesses" about the distribution of the parameter values. For simplicity, we

consider a uniform distribution on the parameters, i.e.,

$$p(\vec{\theta}) = \begin{cases} 1, & \text{if } c \text{ and } p \text{ are within bounds} \\ 0, & \text{otherwise.} \end{cases}$$

Likelihood

One of the assumptions of our model is that only a fraction of the underlying number of individuals who are moving from Exposed to Infective (i.e., who are finishing their latent phase) will be reported. We consider two further simplifications:

There are no false positives, and

The chance of each individual being reported is a constant p.

Considering these simplifications, the empirical reported count of individuals y_t for a week is binomially distributed with probability p of success and $1-p$ of failure over infected individuals i_t at time t, i.e., $y_t \sim binom(i_t, p)$. Therefore, the likelihood function is

$$p(\vec{y}|\vec{\theta}) = \prod_{t=0}^{T-1} BinP(y_t; i_t, p) = \prod_{t=0}^{T-1} \binom{i_t}{y_t} p^{y_t} (1-p)^{i_t - y_t}.$$

For the SEIR model, a time span of $T = 32$ weeks is considered.

Posterior

The posterior $p(\vec{\theta}|\vec{y})$ is computed from the prior $p(\vec{\theta})$ and the likelihood $p(\vec{y}|\vec{\theta})$ functions by using Bayes' Rule as discussed in 2.7.3.

Acceptance Criterion

The acceptance or rejection criterion of the new candidate is given by the Metropolis-Hastings algorithm as follows:

- If the posterior probability of the new candidate $\vec{\theta}_{t+1}$ is greater than that of the current candidate $\vec{\theta}_t$, then $\vec{\theta}_{t+1}$ is always accepted.
- If, however, the posterior probability of $\vec{\theta}_{t+1}$ is less than that of $\vec{\theta}_t$, then $\vec{\theta}_{t+1}$ is still accepted with a certain probability. This is done in order to explore the whole space and to avoid the issue of getting stuck at a local probability hill or maxima.

6.1.3 Overview of the empirical data

The empirical data for our experiments consists of synthetic datasets that have been generated using known parameter values and that have been subject to binomial sampling error. Two datasets

were synthetically generated by making slight modifications in parameters p and c while fixing the other model parameters. This process was conducted so as to facilitate the cross-validation of the parameter distribution values generated by our aspect-based MCMC tool. Because the datasets consists of the rate of reported infectives per week for 32 weeks, we have *32* data points in each dataset.

We use two sets of such synthetically generated empirical data :

- The first set has been generated with parameter c as 300 and parameter p as 0.7.
- The second set has been generated with parameter c as 350 and parameter p as 0.8.

6.2 Experimental Observation and Analysis

We applied our aspect-based MCMC tool on the SEIR model and carried out a number of experiments with the two sets of empirical data, assuming a uniform prior for the unknown parameters. The MCMC tool generated a joint posterior distribution of the parameters p and c as an outcome of these experiments. We conducted some statistical analysis on these distributions to confirm that the MCMC results are satisfactory. In turn, we used the software package "R" to render additional plots with the results using the simulation dataset containing parameter sample values created by the MCMC tool, in order to evaluate further the correctness of our tool. Some of the experimental results presented here have been taken from [8].

6.2.1 Statistical Analysis

The following plots are obtained from the MCMC samples:

- Trace plots of parameters p and c. These plots depict the convergence of the MCMC algorithm. In other words, these plots demonstrate that the space is well explored and that there is no situation of getting stuck at a local maxima or overly slow mixing. Another important fact about these plots is that they display the live results directly on the AnyLogic output window.
  * Figure 6.2 provides trace plots when $p = 0.8$, $c = 350$ and the prior of p is $U(500, 0)$, c is $U(1, 0)$ and the number of iterations is 100,000.
  * Figure 6.3 provides trace plots when $p = 0.7$, $c = 300$ and the prior of p is $U(500, 0)$, c is $U(1, 0)$ and the number of iterations is 100,000.
- 2-D Histogram plots for the joint distribution of parameters p and c. This has been plotted in R. The red spots in these plots denote the most dense regions, which coincide closely with the actual parameter values.
  * Figure 6.4 provides a 2-D histogram plot when $p = 0.8$, $c = 350$ and the prior of p is $U(500, 0)$, c is $U(1, 0)$.

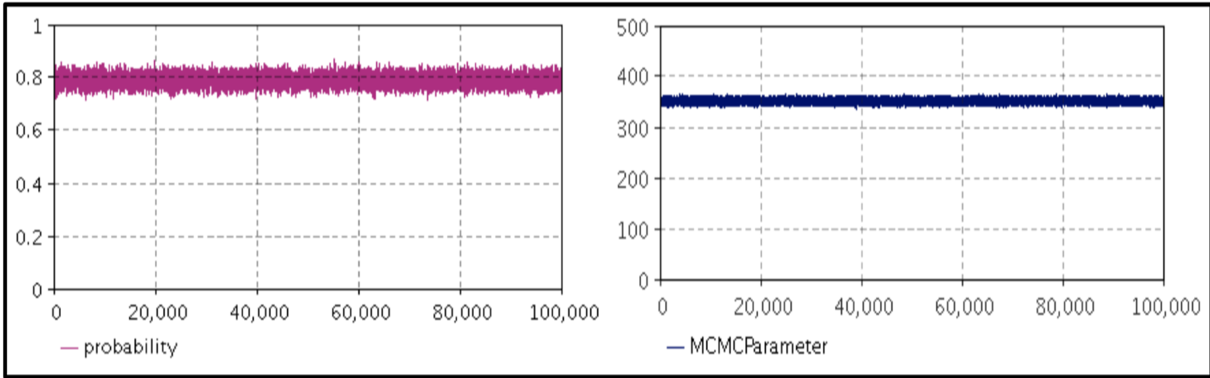


Figure 6.2: Trace Plot of of parameter p and c when the actual values are 0.8 and 350, respectively.

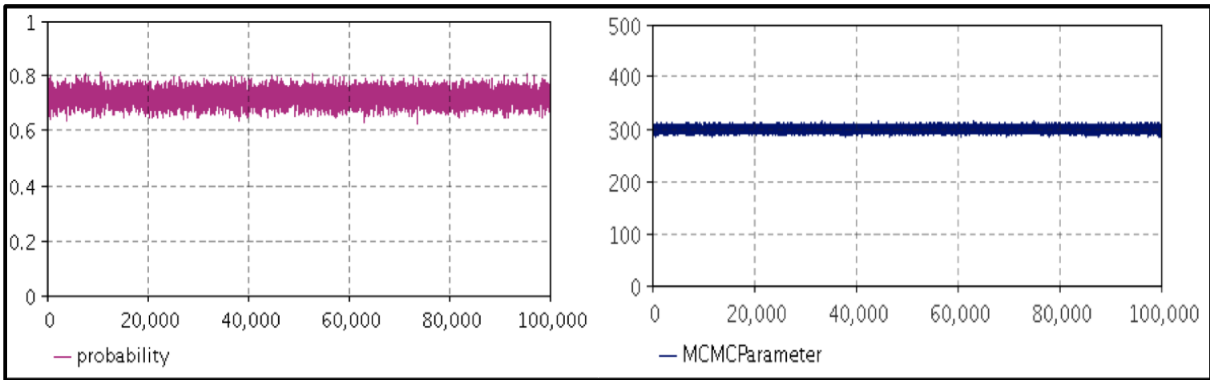


Figure 6.3: Trace Plot of of parameter p and c when the actual values are 0.7 and 300, respectively.

    * Figure 6.5 provides a 2-D histogram plot when $p = 0.7$, $c = 300$ and the prior of p is $U(500, 0)$, c is $U(1, 0)$.

  – Density plots of the marginal distribution of the parameters p and c. This has also been plotted in "R". These plots show that the distribution of c is multi-modal, while p has a broad and smooth extent.

    * Figure 6.6 provides the marginal density plots when $p = 0.8$, $c = 350$ and the prior of p is $U(500, 0)$, c is $U(1, 0)$.

    * Figure 6.7 provides the marginal density plots when $p = 0.7$, $c = 300$ and the prior of p is $U(500, 0)$, c is $U(1, 0)$.

Some statistical analysis associated with the MCMC samples of parameters p and c are presented in Tables 6.1 and 6.2. Table 6.1 provides the quantile values of the parameters, as summarized by the statistical package "R". Table 6.2 provides the mean value of the parameters, associated standard deviations and the 95% Credibility Interval for the parameters.

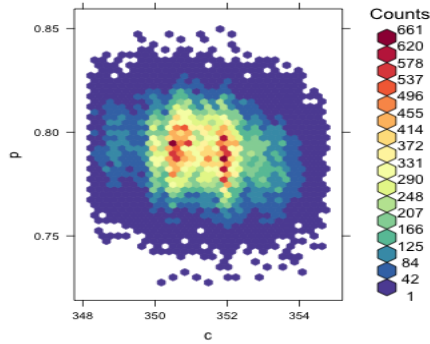


Figure 6.4: Joint distribution of parameters p and c when the actual values are 0.8 and 350, respectively.

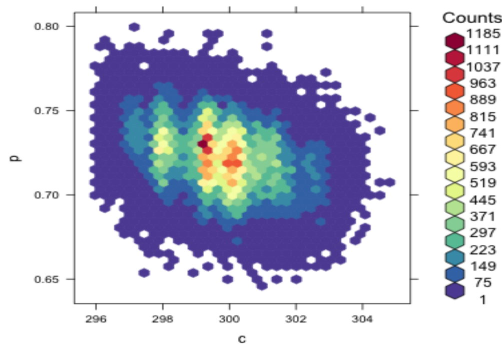


Figure 6.5: Joint distribution of parameters p and c when the actual values are 0.7 and 300, respectively.

6.3 Advantages of the MCMC Tool

In this section we discuss the benefits of using the MCMC tool for generating the joint posterior distribution of unknown parameter(s). Specifically, we emphasize on three factors:

– How the MCMC tool improves modularity of the system as compared to implementing the MCMC functionality in a model on ad-hoc basis.

– From a software engineering point of view, how challenging it is to implement the MCMC functionality manually in a model and how the MCMC tool simplifies it.

– The correctness of the results provided by the tool.

6.3.1 MCMC Tool Modularity Assessment

This section presents the discussion as to how aspects can help in the enhancement of modularity in the simulation modeling architecture when additional functionality such as MCMC needs to be introduced for a simulation. To that end, two versions of a model with the MCMC functionality

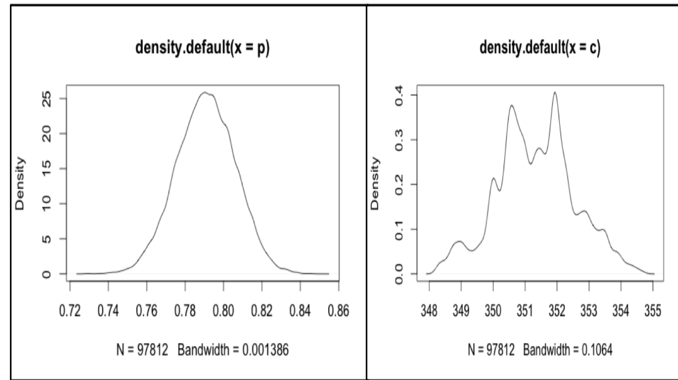


Figure 6.6: Marginal Density plots of parameters p and c when the actual values are 0.8 and 350, respectively

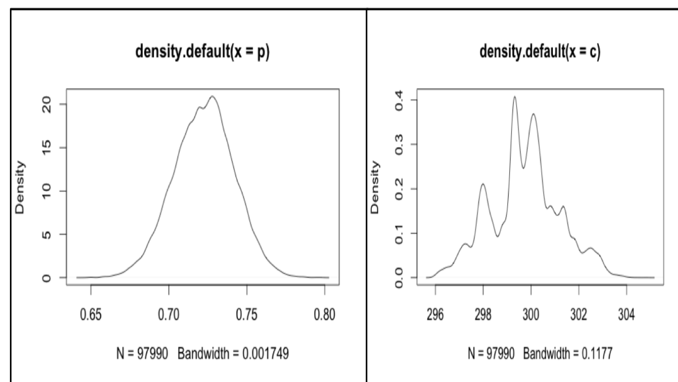


Figure 6.7: Marginal Density plots of parameters p and c when the actual values are 0.7 and 300, respectively

have been developed for comparison and contrast. In one version, the model directly contains the code required for introducing the MCMC functionality within the model. This is like a traditional (manual) method of writing one's own code to add new functionality. The other version utilizes the aspect-based MCMC tool with the model. Thus the comparative study is conducted between the version without aspects with the one with aspects. A figure demonstrating the improvement of modularity has been generated to strengthen our argument.

The modularity assessment figure, figure 6.8 was inspired by the standard aspect visualization tool available with the AspectJ compiler in Eclipse. Similar to the figure 4.2, this figure has been created with minor additions. As explained in the section 4.3.4, the modifications had to be manually incorporated because the model code auto-generated by AnyLogic does not provide the complete source code. In this case, the missing class is *com.anylogic.engine.ExperimentRunFast*. It is used both by the model and the MCMC Tool, but it does not get reflected in the figure generated by the visualization tool because it lies in a library called by that code.

The left half of the figure 6.8 represents the simulation model with code written manually to incorporate the MCMC functionality, or in other words, the simulation model without aspects.

Par	Emp	2.5%	25%	50%	75%	97.5%
p	0.8	0.7605	0.7804	0.7909	0.8014	0.8202
c	350	348.8229	350.5272	351.3359	352.1038	353.7728
p	0.7	0.6843	0.7094	0.7230	0.7356	0.7594
c	300	296.9731	298.8563	299.7823	300.6012	302.7186

Table 6.1: Quantiles for parameters p & c (Here Par stands for Parameter and Emp for Empirical Value)

Par	Emp	Mean	Stdev	Credibility Interval
p	0.8	0.7908	0.01533	[0.7656, 0.8160]
c	350	351.3381	1.2164	[349.3373, 353.3388]
p	0.7	0.7225	0.01935	[0.6907, 0.7543]
c	300	299.7444	1.4425	[297.3717, 302.1171]

Table 6.2: Mean, Standard Deviation and Credibility Intervals for parameters p & c (Here Par stands for Parameter, Emp for Empirical Value and Stdev for Standard Deviation)

The simulation model used for this figure has been extensively used for the experiments conducted with the MCMC tool. It is a common and popular epidemiological model and is discussed in detail in section 6.1.1. The left side of the figure 6.8 displays two cylinders, each representing a class. The red stripes represent the different methods of the classes affected by the tool. While assessing modularity for the MCMC tool via aspect visualization, we found that the concern (crosscutting concern that can be efficiently managed by aspects) is spread across different locations within 4 methods of 2 classes. In the figure :MCMCToolModularity1, number "1" represents the class ParameterVariation (Simulation scenario used for this model) and number "2" represents the class ExperimentRunFast.

The right half of the figure represents the simulation model with the MCMC tool – in other words, the simulation model with aspects. The figure displays all the cylinders representing classes present in the left half with a change, wherein the red stripes have changed to grey stripes and an additional

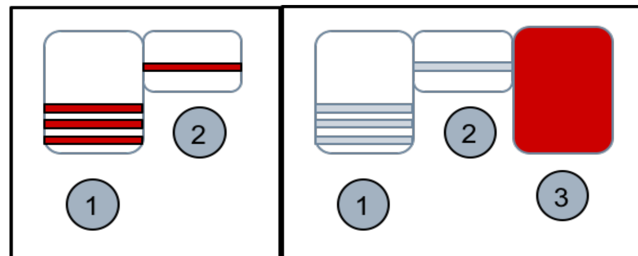


Figure 6.8: Left: Simulation model with code written manually to incorporate the MCMC functionality; Right: Simulation model with the MCMC Tool

red cylinder represented by number "3" has appeared. The red cylinder is actually the MCMC tool containing the aspect code. It is the single encapsulation containing all the code concerned with the MCMC algorithm along with the UI window generation code. It has extracted the MCMC related concerns from the two affected classes, as a result of which they no longer contain the red stripes. It has been established earlier that this approach does not necessarily decrease the lines of code, since an additional aspect is introduced here. But certainly this approach improves modularity, comprehensibility and maintainability since now there is a single location which is affected with the MCMC concern, unlike multiple locations in two different classes. Any change in either of these classes concerned with other model-specific business logic will no longer impact the MCMC code.

6.3.2 MCMC Tool Simplification Assessment

In this section we focus on the problems introduced in the system with the manual implementation of the MCMC functionality and how the MCMC tool helps in resolving such software engineering challenges. Introducing the MCMC code manually in a model incurs the following difficulties:

- First, the MCMC algorithm requires iterative runs of the simulations with the generated parameter values. As a result of this, one needs to control the run of the model such that each run is carried forward with the new parameter value which is eventually used to generate the model output, which in turn, is used to compute the posterior. The only way to control the model runs is to use the *Custom Experiment* provided by AnyLogic. A "Custom Experiment" runs the simulation with a custom scenario entirely written by the modeler. This experiment type is available only in the professional version of AnyLogic, which is much more expensive than the educational version, with the latter being commonly used among students and researchers. Hence, the manual version of MCMC requires a substantial amount of coding (code to run the model beyond the MCMC code) along with the extra price for the professional version of AnyLogic.
- Second, the custom experiment has no built-in graphical interface. Hence the interactive plots cannot be readily supported in this version.
- Third, the manual version of MCMC code lacks the UI present in the MCMC tool to easily accept the user inputs. Hence, many changes in parameters can entail code changes.

Figure 6.9 depicts the architecture of the implementation of the MCMC functionality in a simulation model in AnyLogic. The figure presents the custom experiment that needs to be created to simulate the model iteratively along with the MCMC code and the classes that need to be developed to execute the MCMC code.

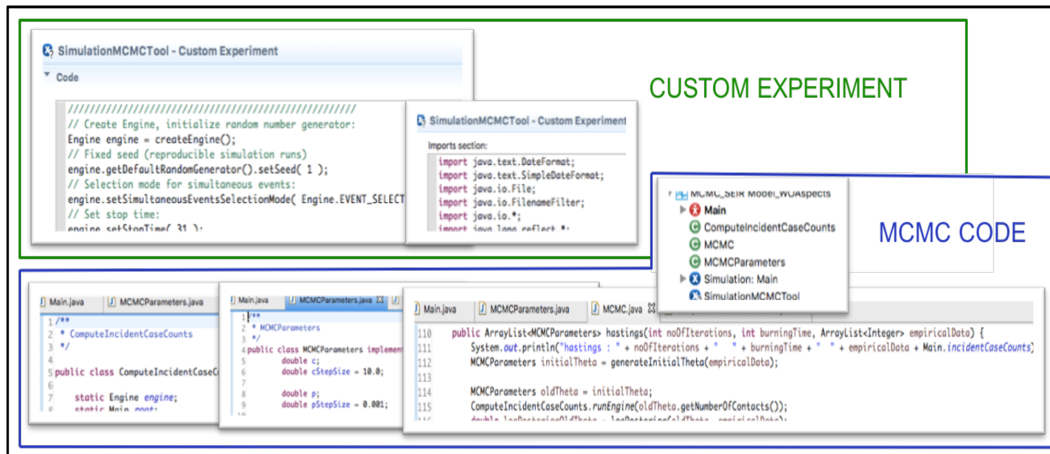


Figure 6.9: Coding required for incorporate the MCMC functionality without aspects

The custom experiment class is displayed within the green box in the figure 6.9. It needs to be developed to customize execution of the simulation engine which is required for MCMC. One has to keep in mind that this experiment lacks the defined and readily parameterized behaviour which is characteristic of other experiments of Anylogic. As such, the modeler needs to take care of all fundamentals using manual coding, including but not limited to assigning values to parameters, fixing the random number seed if required, setting the time period for the model, starting the simulation engine and stopping the simulation engine.

The MCMC code-base is displayed within the blue box in the figure 6.9. The project view of the AnyLogic model is provided for understanding the structure of the model, as is useful when seeking to add functionalities such as MCMC. Three classes are created here within the Anylogic model to implement MCMC. Once the posterior samples are generated, the modeler needs to use "R" or some other plot generating software to plot the posterior samples. In this version, the modelers or the users do not get the privilege of experiencing the interactive plots. As such, certain parameters like the number of iterations and upper and lower bounds may require frequent changes and these can lead to multiple changes in the MCMC code (as the parameters are hardcoded) and thus slower learning.

The implementation of the MCMC algorithm requires a substantial amount of coding in Java. The coding in this case is not solely a set of print clauses – which characterize much of the functionality of the tracing and logging tool – but requires more than that. As such many modelers, specially if not from a Computer Science background, might find this challenging. Even for others, for example at crunch times for stringent deadlines, the time required to develop such an application might be a challenge. The manual process is a quite cumbersome and exhaustive process which is more prone to errors. This process is also dependent on individual memory and compliance as frequent revisits and revisions are required at the onset of minor changes made in the model.

The main challenge revolves around creating the interfaces between the MCMC algorithm and the AnyLogic model. In this approach, the MCMC algorithm is executed on a simulation model to sample from posterior distribution(s) of certain parameter(s). Technically, this means that the MCMC algorithm depends upon the posterior probability computed by the simulation model. Thus a relationship needs to be developed between the AnyLogic model and the MCMC algorithm such that the AnyLogic model is executed after the concerned parameter(s) of the model is(are) updated with a candidate value generated by the MCMC algorithm, and the output of the model is subsequently used to assess the posterior probability used to determine acceptance by the MCMC algorithm. This action must occur iteratively to support MCMC iterations. Thus, creating MCMC functionality for a dynamic model in AnyLogic involves the complete architecture as featured in figure 6.9. Although the MCMC algorithm is readily available, the connections with AnyLogic need to be developed by a modeler or user. This requires a certain amount of time and knowledge about the software engineering side of AnyLogic, which may be beyond the modeling knowledge typically required for AnyLogic projects

6.3.3 MCMC Tool Correctness Assessment

MCMC performed reasonably well with our example model and the empirical data. In our approach, we have cross-leveraged the MCMC technique with a widely used dynamic simulation model to deduce posterior distributions for unknown model parameters, assuming a prior based on existing knowledge of the system and a sampling function determining the likelihood of observing a given set of observations and considering a set of specific parameters. This mechanism has several advantages as follows:

- MCMC is a generic and powerful Machine Learning technique that helps to sample from the posterior distributions of unknown parameters.
- The joint distribution of parameters c and p illustrates that the most dense regions in the parameter space closely coincides with the actual parameter values. The same is true for both the datasets used for experiments. This further denotes that this approach is highly efficient and generic in nature.
- The trace plots of parameters c and p illustrates convergence. This implies that the entire parameter space is explored efficiently which, in turn, suggests that the issue of getting stuck at local maxima is not a problem for the particular model examined here.
- The density plots of parameters c indicate a multi-modal distribution. This helps re-emphasize the fact that this approach can also handle the multi-modal systems.
- Good results are achieved for all the plots and statistics. Although we deal with uniform prior here but assuming a different, highly biased prior does not change the results [35]. This

suggests that this approach is powerful and robust to different priors. This is because the Metropolis Hastings Algorithm implemented to compute the posterior is robust and does not depend on the prior. It must be emphasized here that the modeler has the liberty to adopt any prior in the model.

- The mean value of the sampled values of parameters c and p are very close to their actual values. This further suggests the correctness of the implementation.
- The actual values of the parameters c and p always lie within the range of the 95% credibility intervals, lending further confidence in the accuracy of the implementation.

6.3.4 Learning

The work described in this chapter concerns the implementation of a technique that combines a broad class of dynamic models with one of the most powerful and efficient Bayesian Machine Learning techniques that exist today - the Markov Chain Monte Carlo method, or MCMC. MCMC is a highly generic technique and can be combined with dynamic simulation models to overcome a number of limitations :

- This technique provides us with results traditionally achieved using a combination of different methods, namely Calibration and Sensitivity Analysis.
- Through re-running the MCMC algorithm, this technique can be reused for newly arrived data points.
- One of the key factors of this approach is drawing samples from a posterior distribution. This lacks closed form expression mainly because the posterior results depend upon the results of the simulation models, which themselves are not amenable to closed form description based on parameter values. The simulation model plays an important role here, as it helps in mapping parameter values to model output, making it easier to combine with empirical data in a probabilistic model.

Chapter 7

Conclusion

7.1 Summary

One of the primary reasons for the emergence of an aspect oriented programming paradigm is to enhance modularity of applications. Aspects play an important role in eliminating crosscutting concerns. If the logging and tracing functionality are introduced within simulation models themselves, it will lead to repetition of the same code fragments as they need to be inserted into all the classes of every model, and thus lead to coupling among several classes and within distinct models. The aspect oriented programming paradigm helps with decoupling in situations like these by providing a single platform, viz. aspects, for handling inter-connected and inter-dependent functionalities. According to this demonstration, AspectJ helps with the addition of new features in a model in an efficient, generalized and modularized manner. The modularity improves the code by eliminating dependency on several methods and classes, loosening coupling, and decreasing dependency, thereby increasing the flexibility of the system. The system becomes more flexible as it can support modifications without affecting other dependent classes and can add new functionality more efficiently and in a simplified manner. Our Aspect-based logging and tracing framework enables documentation of the details related to model execution and the details related to the activities occurring in the model execution, without affecting the existing model code. It can in fact be applied to diverse models with minimal changes. This further leads to reduction of code changes, thereby reducing the risk of changing the model and its semantics. The inclusion of aspects in simulation modeling provides a number of new facilities to modelers as well as to end-users and other stakeholders. It results in more modular and flexible design by helping minimize new lines of code within a model for the purpose of tracing and decreasing the mingling of concerns.

Using MCMC techniques with dynamic models can help to reduce overhead and complexity in the context of dynamic modeling by providing an powerful and general means of parameter estimation. The aspect oriented approach of this implementation enhances the flexibility and modularity of the process. MCMC is one of the most powerful and widely used (Bayesian) Machine Learning techniques that aids in resolving a number of problems associated with the lack of data availability.

We have made an attempt to utilize this technique with dynamic models to reduce the challenges emerging from a lack of sufficient data. The central feature of this approach is to draw samples from a posterior distribution. But in this case, the posterior distribution (and the likelihood function on which it depends) lacks a closed form expression, mainly because the posterior results depend upon the results of the dynamic model under analysis, which in general lacks a closed form solution. Thus dynamic models help in mapping the parameter values to model output, making it easier to compare to empirical data.

7.2 Contributions from this research

The use of aspects improves the flexibility of the simulation modeling framework and enhances the efficiency of the entire process. The contributions of this research can be summarized as below:

- An aspect-based framework referred to as the logging and tracing tool. This tool generates the run log, the trace log, images portraying the initial and final states of the model and a copy of the present version of the model.
    * The run log documents some vital information which can be later used for reproduction of the output and can serve as a reference document. It helps ensure reproducibility of a model and Run Logs also aid reviewers and knowledge-users to secure a glimpse of the summary for different executions.
    * The trace log carefully documents intricate details related to a simulation run. It provides a vivid description of all behind the scenes activities, so that it can be used for better understanding, analysis and debugging purposes.
    * The images of the output canvas capturing the initial and final state of the model provides us with the information required for analysis and re-creation of output.
    * A copy of the present version of the model that is required for re-creation of output.
- Another aspect-based framework referred to as the MCMC tool. This tool samples from the posterior distribution for unknown parameters of the model as specified by the model users, which is a general way of handling unknown parameters in systems with insufficient ground-truth data.

7.3 Limitation of our tools

So far we have mainly discussed the benefits of using our tools, including motivations behind developing such tools. In this section, we will discuss the limitations of both our tools.

7.3.1 Limitations of the logging and tracing tool

The logging and tracing tool provides considerable benefits to users in the form of documentation, storage, reproduction and efficiency. However there are certain limitations associated with it as well:

* One of the major drawbacks of our approach is dependency on the AnyLogic API. The aspects use join points, and method names from the implementation of AnyLogic. Thus with minor changes of the AnyLogic API, there is a chance that our tools will cease to function correctly. AnyLogic undergoes version updates at least once a year. As a consequence of the above, the aspect-based application working now might not work in the future. Thus, in order to make the tool work successfully and reliably, some sort of guarantee is required from the AnyLogic company.
* The tracing and logging system is not able to read input data from files or databases dynamically. As a result of this, the parameter values might be flawed if the corresponding model uses files or databases as its source of data, and also the model may not be reproducible.
* The tracing and logging system records the primary random number seed only. So if a simulation model uses more than one random number seed, then the model may not be reproducible.
* The trace log files created by the logging and tracing tool are usually large, depending on the model size, the number of factors and the model population size. But each detail present in the trace file could be important; economies could be secured by compressing the output files. Another way to deal with this would be to generate a user interface providing the users with a menu of different kinds of information to be recorded and allow them to make their choice. That way, only required output would be generated.

7.3.2 Limitation of the MCMC tool

The MCMC tool also provides a lot of benefits to users by sampling from the parameter distribution, generating appropriate statistical results and related data. However there are certain limitations, including the following:

* The MCMC tool requires modelers or users to create the plots within the simulation. This might seem to be a little bit of extra work. We will try to eliminate this process and create graphs in the next version of the tool.
* The trace plots that are generated to display the distribution of the parameter(s) include the initial phase when the algorithm explores the parameter space to find the initial value of parameters with non-zero posterior probability. In the future implementation, we will

exclude this phase from being displayed in the graph.

7.3.3 Future Work

We have achieved some success in improving the flexibility, modularity and efficiency of simulation modeling technique via AnyLogic. Now we look forward to extending the utilization of aspects to decouple the model and view of simulation models, using the MVC architecture. The decoupling of view from the governing processes of a simulation model allows attaching multiple views to that model to provide different presentations. It further changes the way a view responds to user input (that is, without changing its visual presentation affront). With this approach, the user is able to store away a particular ”lens” for re-use in the future. Figure 7.1 presents a design pattern for the separation of user interface logic from business logic.

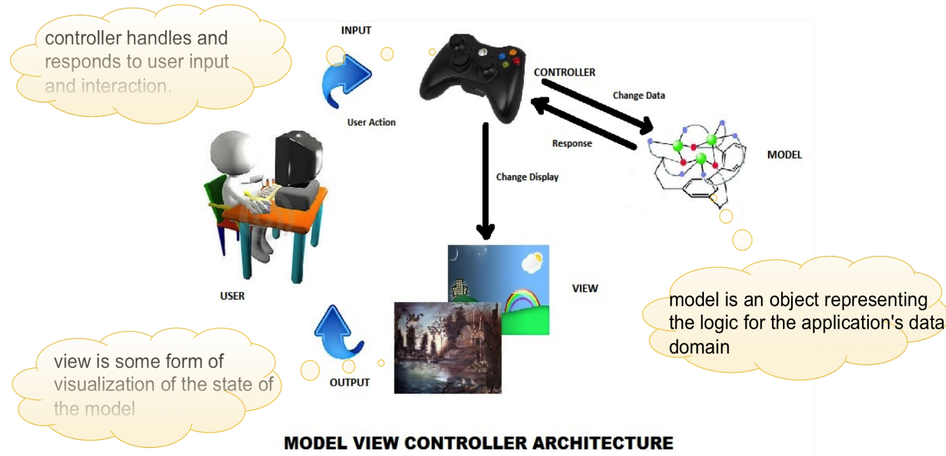


Figure 7.1: Model View Controller Architecture for decoupling view from model

AspectJ will be used with the MVC pattern to:

* enable alternative visualizations, i.e., different representations for different sets of users, such as a spreadsheet view of data, or a pie chart of the same data, or — for yet another example — a stacked bar.
* reduce the complexity of a model by decoupling the user interface logic from the business logic of the model. The modular approach of the proposed framework will further minimize complexity.
* display aggregate data summaries which are otherwise not easy to obtain from the model.
* improve maintenance of the application by redesign of the simulation modeling technique in a modular layout to separate the concerns.
* decrease the number of changes required for any modification, since modifying the business logic of the model will not impact its user interface logic and vice versa.

References

[1] *Markov Chain Monte Carlo for Computer Vision.* Available at http://vcla.stat.ucla.edu/old/MCMC/MCMC_tutorial/Lect1_MCMC_Intro.pdf, last accessed Feb 28, 2015.

[2] AnyLogic features: Overview. http://www.anylogic.com/features, (last accessed Nov 26, 2015).

[3] Fatima Alawami. An Aspect Refactoring Tool for The Observer Pattern. Master's thesis, Department of Computer Science, University of Saskatchewan, May 2012.

[4] Roy M. Anderson and Robert M. May. *Infectious Diseases of Humans Dynamics and Control*. Oxford University Press, 1992.

[5] AOP with Spring Framework. http://www.tutorialspoint.com/spring/aop$_{w}ith_{s}pring.htm$, $(lastaccessedApr30, 2016)$.

[6] The AspectJ Programming Guide. https://eclipse.org/aspectj/doc/released/progguide/, (last accessed Nov 26, 2015).

[7] Scott Bateman, Carl Gutwin, Nathaniel Osgood, and Gordon McCalla. Interactive usability instrumentation. In *Proc. EICS 2009*, pages 45–54. ACM Press, 2009.

[8] Priyasree Bhowmik, Christopher Dutchyn, and Nathaniel Osgood. An aspect oriented framework to applying markov chain monte carlo methods with dynamic models. In *Proc. Symposium on Theory of Modeling and Simulation (TMS/DEVS 2016)*, Pasadena, CA, April 2016.

[9] Priyasree Bhowmik, Nathaniel Osgood, and Christopher Dutchyn. Improving the flexibility of simulation modeling with aspects. In *Proc. Symposium on Theory of Modeling and Simulation (TMS/DEVS 2015)*, Alexandria, Virginia, April 2015.

[10] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

[11] Meriem Chibani, Brahim Belattar, and Abdelhabib Bourouis. The use of the aspect oriented programming (aop) paradigm in discrete event simulation domain: Overview and perspectives. In *Proc. The Third International Conference on Digital Information Processing and Communications (ICDIPC2013)*, page 653660, Dubai 2013.

[12] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004.

[13] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis, Second Edition*. Chapman and Hall/CRC, 2004.

[14] Nigel Gilbert. *Agent-based models*. Sage Publications Inc., 2007.

[15] Chapter 3 General Principles. http://www.mi.fu-berlin.de/inf/groups/ag-tech/teaching/2012_SS/L_19540_Modeling_and_Performance_Analysis_with_Simulation/03.pdf, (last accessed Nov 26, 2015).

[16] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *In AOSD 04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM, 2004.

[17] Java for AnyLogic users. http://www.xjtek.com/files/book/Java$_{f}or_{A}nyLogic_{u}sers.pdf$, $(lastaccessedApr20, 2014)$.

[18] Java Reflection Tutorial. http://tutorials.jenkov.com/java-reflection/index.html, (last accessed Dec 15, 2015).

[19] Yuri G. Karpov. Anylogic a new generation professional simulation tool. In *Proc. VI International Congress on Mathematical Modeling*, Nizni Novgorog, Russia, 20-26 September 2004.

[20] G Kiczales. Aop the fun has just begun. In *New Visions for Software Design & Productivity Workshop, Vanderbilt University, Nashville, TN*, 2001.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *In ECOOP '01 Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–354. Springer-Verlag, 2001.

[22] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.

[23] Takis Konstantopoulos. Markov Chains and Random Walks, 2009. http://www2.math.uu.se/ takis/L/McRw/mcrw.pdf, (last accessed Dec 4, 2015).

[24] Uira Kulesza, Claudio Sant'Anna, and Carlos Lucena. Refactoring the junit framework using aspect-oriented programming. In *In OOPSLA '05 Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 136–137. ACM, 2005.

[25] Ramnivas Laddad. *AspectJ in Action*. Manning Publications Co., 2003.

[26] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. In *In OOPSLA 03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12. ACM Press, 2003.

[27] Kwanwoo Lee, K. C. Kang, Minseong Kim, and Sooyong Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In *Proc. 10th International Software Product Line Conference (SPLC'06)*, pages 10 pp.–112. IEEE, Baltimore, MD 2006.

[28] Jani Metsa, Shahar Maoz, Mika Katara, and Tommi Mikkonen. Using aspects for testing of embedded software: experiences from two industrial case studies. *Software Quality Journal*, 22(2):185–213, June 2014.

[29] Russ Miles. *AspectJ Cookbook*. O'Reilly Media Inc., 2005.

[30] An AgentBased Approach for Modeling Population Behavior and Health with Application to Tobacco Use. http://www.sandia.gov/CasosEngineering/docs/PSM_9_18_2012_PLOS.pdf, (last accessed Nov 26, 2015).

[31] Modeling Simulation. http://www.systems-thinking.org/modsim/modsim.htm, (last accessed Nov 25, 2015).

[32] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, Cambridge, MA, 2012.

[33] AnyLogic Example Health Model Inventory. http://www.cs.usask.ca/faculty/ndo885/Classes/ConsensusA (last accessed Apr 20, 2014).

[34] Nathaniel Osgood. Silver: Software in support of the system dynamics modeling process. In *Proc. The 27th International Conference of the System Dynamics Society*, 2009.

[35] Nathaniel Osgood and Juxin Liu. *Analytical Methods for Dynamic Modelers*, chapter Combining Markov Chain Monte Carlo Approaches and Dynamic Modeling. The MIT Press, Cambridge, MA, Forthcoming (2015).

[36] Nathaniel Osgood and Yuan Tian. 15 things system dynamics can learn from software development. In *Proc. The 29th International Conference of the System Dynamics Society*, 2012.

[37] Jeeva S. Paudel. Aspect Structure of Compilers. Master's thesis, Department of Computer Science, University of Saskatchewan, August 2009.

[38] Steven F. Railsback and Volker Grimm. *Agent-based and individual-based modeling: a practical introduction.* Princeton University Press, 2011.

[39] Trail:The Reflection API. https://docs.oracle.com/javase/tutorial/reflect/, (last accessed Dec 15, 2015).

[40] Oliva Rogelio. Model calibration as a testing strategy for system dynamics models. *European Journal of Operational Research*, 151(3):552–568, 2003.

[41] Vladimir O. Safonov. *Using Aspect-Oriented Programming for Trustworthy Software Development.* Wiley-Interscience, New York, NY, 2008.

[42] Suliza Sumari, Roliana Ibrahim, Nor Hawaniah Zakaria, and Amy Hamijah Ab Hamid. Comparing Three Simulation Model Using Taxonomy: System Dynamic Simulation, Discrete Event Simulation and Agent Based Simulation. *Internal Journal of Management Excellence*, 1(3):54–58, 2013.

[43] Maarika Teose, Kiyan Ahmadizadeh, Eoin OMahony, Zhao Smith, Rebecca L.and Lu, Stephen P. Ellner, Carla Gomes, and Yrjo Grohn. Embedding system dynamics in agent-based models for complex adaptive systems. In *Proc. Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[44] Aleksandra Tesanovic, Ke Sheng, and Jorgen Hansson. Application-tailored database systems: A case of aspects in an embedded database. In *Proc. Int'l Database Engineering and Applications Symposium*, pages 26–35. ACM, 2004.

[45] John Viega, JT Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

[46] Wayne W. Wakeland, Edward J. Gallaher, Louis M. Macovsky, and C. Athena Aktipis. A comparison of system dynamics and agent-based simulation applied to the study of cellular receptor dynamics. In *Proc. 37th Hawaii International Conference on System Sciences*, 5-8 Jan., 2004.