Using Actors to Implement Sequential Simulations

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
Ryan Harrison

Permission to Use

Abstract

This thesis investigates using an approach based on the Actors paradigm for implementing a discrete event simulation system and comparing the results with more traditional approaches. The goal of this work is to determine if using Actors for sequential programming is viable. If Actors are viable for this type of programming, then it follows that they would be usable for general programming. One potential advantage of using Actors instead of traditional paradigms for general programming would be the elimination of a distinction between designing for a sequential environment and a concurrent/distributed one. Using Actors for general programming may also allow for a single implementation that can be deployed on both single core and multiple core systems.

Most of the existing discussions about the Actors model focus on its strengths in distributed environments and its ability to scale with the amount of available computing resources. The chosen system for implementation is intentionally sequential to allow for examination of the behaviour of existing Actors implementations where managing concurrency complexity is not the primary task. Multiple implementations of the simulation system were built using different languages (C++, Erlang, and Java) and different paradigms, including traditional ones and Actors. These different implementations were compared quantitatively, based on their execution time, memory usage, and code complexity.

The analysis of these comparisons indicates that for certain existing development environments, Erlang/OTP, following the Actors paradigm, produces a comparable or better implementation than traditional paradigms. Further research is suggested to solidify the validity of the results presented in this research and to extend their applicability.

Acknowledgements

I would like to thank my graduate supervisor, Nadeem Jamali, for his support and guidance throughout my graduate studies. Thanks are also in order my my thesis committee members, Chanchal Roy and Nathaniel Osgood, for their feedback and advice as I worked through my research.

Google Canada has employed me for the last four years of my research and provided me with the opportunity to work on my research as part of my regular duties, for which I am grateful. All of my managers during my time at Google — Alex Nicolao, Neil Thomas and Derek Phillips — have been supportive of my work. Finally I would like to acknowledge the efforts of Chris Wolfe, my fellow employee at Google, who has been subjected to my research over the years, and retaliated by making helpful suggestions.

This work is dedicated to my friends and family who have supported me through out the process of working on my thesis and my earlier education. This especially includes my parents, Adeline and James Harrison, who provided me with every opportunity to achieve everything that I have and will achieve.

I would like to specially acknowledge the efforts of my wife, Sarah Gritzfeldt, who has been supportive in an innumerable number of ways including, but not limited to, copy editing my work and keeping me sane for all this time.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

List of Abbreviations

API	Application Programming Interface
BEAM	Bogdan/Bjrn's Erlang Abstract Machine
CPU	Central Processing Unit
DES	Discrete Event Simulation
DTS	Discrete Time Simulation
FSM	Finite State Machine
GCC	GNU Compiler Collection
GNU	GNU's not UNIX
HiPE	High Performance Erlang
IDE	Integrated Development Environment
ISO	International Standards Organization
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OO	Object Orientated
OS	Operating System
OTP	Open Telecom Platform
PID	Process Identitifier
RAM	Random Access Memory
RSS	Resident Set Size
SEIR	Susceptible, Exposed, Infectious, Recovered/Removed
SEIRS	Susceptible, Exposed, Infectious, Recovered/Removed, Susceptible
SMP	Symmetric Multiprocessing
SQL	Structured Query Language
VM	Virtual Machine
VSZ	Virtual Memory Size

Chapter 1
Introduction

1.1 Overview

In the development of modern computers, focus has shifted from performing more work on a single core to systems that use multiple cores to process multiple streams of data and instructions at once. This has led to evolution in the paradigms used to program these systems. As a consequence the focus of developers has moved to algorithms and frameworks that handle concurrency.

Actors[1] is a model of concurrency that has had success both as an active choice, i.e. Akka for Java/Scala, and as an convergent design, i.e. Erlang/OTP, for programming multi-core systems. This model divides computational operations up into discrete atomic units called actors. Each actor encapsulates part of a computation's state and implements its own thread of control. Actors communicate with each other using an asynchronous mailbox based messaging protocol. This means that there is no shared state between threads of control, which eliminates the need for sophisticated locking mechanisms. There are still issues like livelock and deadlock that need to be addressed, but Actors offers significant advantages for implementing concurrent systems over traditional paradigms.

One of the key characteristics of Actors is that it abstracts away the underlying system details, such as the number of cores available. This is achieved by implementations having a management layer that handles message passing and thread coordination. This offers benefits, by allowing application programmers to not be concerned with system programming details and producing portable code. It does come with costs though, since it introduces overhead and another level of indirection. The management layers need to be implemented in a generic manner and thus cannot take advantage of some optimizations, such as mapping the number of active actors created to the number of cores available.

An alternate and complementary model for concurrency is that of threading. This is a fundamentally lower level abstraction of concurrency and often Actor systems are built on top of it. It is common in threaded systems to have one main thread of control for the application that is performing the core of the algorithm. At points where it is appropriate this main thread will create worker threads, using a thread pool, that will perform computations concurrently, eventually returning the result to the main thread or into a shared memory location. This allows for a more optimal usage of resources since the developer can tune their system to only create as many threads as there are cores available, and only use them when multiple operations can occur. In contrast, Actor systems need to have the data structures and logic related to concurrency active at all times. As a low level abstraction, threading leads to a significant implementation burden, since the developer needs to address issues related to coordinating access to shared resources and the code being less portable.

Both of these concurrency models embed the concept that some operations can be performed at the same time, concurrently, and others need to happen in a specific order, sequentially. This idea of there being concurrent and sequential operations in a computation and how it influences the performance of multi-core processors is formalized in Amdahl's law [2]. Amdahl's law looks at parallelizing computations and predicting how much speed up can be expected to be achieved by adding more cores based on how much of the operation can be performed concurrently and how much must be sequential. There are many systems that are very amendable to parallelization, i.e. raster graphics. There are many more systems in computer science that are not exceptionally amendable to concurrent computation, which makes highly parallel problems the minority.

Implementations of concurrent systems are often evaluated by how well they scale up to many threads of control, which assumes a highly parallelizable problem is being solved. The reverse question of how well these systems handle very sequential problems is not as often discussed, since it is assumed that implementers would use a traditional paradigm if the problem was highly sequential. This methodology ignores that models like Actors, which are nominally for concurrent problems, are often useful for describing non-concurrent ones. Actors describes problems in terms of discrete modules of logic and encapsulated state, which paradigms like object orientation (OO) have found to be very powerful. It possible to implement a solution following the Actors model, even if it is sequential, and if the problem is amendable to concurrent speed up, then the management layer could be used to take advantage of this without significant changes to the code.

This design pattern of implementing one Actor based system, that was used in both the single threaded and multi-threaded case, has one significant drawback: it depends on the ability of the management layer to limit the amount of overhead in the sequential case. For concurrent systems the management layer has to manage threads of control and mailboxes which, in comparison to calling a method on an object, is relatively expensive. If the Actor system has been implemented with down scaling in mind, in the sequential case message passing should reduce to using direct function calls and there would be no thread handling.

In this thesis the viability of having a single Actor based implementation that scales both up and down effectively was studied. The specific system that was implemented is a discrete event simulation (DES) of a SEIRS model. This problem is relatively sequential due to the strict ordering of operations that needs to be enforced, but there is some opportunity on each timeslice for multiple operations to be occurring at once.

There do exist algorithms for implementing more concurrent algorithms to solve to this problem, but they do not exhibit the sequential nature of interest, so were not investigated.

Two frameworks were used for implementing the system following the Actor model: Akka, which is Java/Scala based, and Erlang/OTP. In addition to the Actor implementations, there were sequential implementations in the base languages to give a baseline to compare with. For absolute comparison, a C++ sequential implementation was developed, since it was expected to have optimal run time characteristics. Finally, a thread pool based Java implementation was used to compare Actors to other models of concurrency.

The implementations were compared based on run time characteristics and code quality. The run time parameters look at the CPU and memory usage. The OS reported CPU time for running a model on the implementation was recorded as well as the maximum resident set size. The measurements of code quality focus on the complexity and size of the implementations, specifically measuring the cyclomatic complexity of the implementation; the number of statements used; and the number of lines of functional code. Additional metrics were derived from these values to examine properties like complexity density.

1.2 Hypothesis

Due to the sequential nature of the system being run, it was expected that the Actor based implementations of the simulation system would have a significant amount of run time overhead associated with them and the speed up parallelizing part of the operation would not be sufficient to overcome this. There would be a speed up as more cores are made available, but it would be substantially sub-linear relative to the number of cores. It was expected that the data from this research would show that running multiple instances of the sequential implementation in parallel would have significantly better run times than running the Actor based implementation serially for the same number of iterations. It was expected that there will be similar results for the thread pool style implementation, though not as pronounced. For comparison between languages, it was expected that the C++ implementation would be faster by orders of magnitude, and the Erlang ones would be generally faster than the Java ones.

For memory usage it was expected that there would be a similar pattern, where the sequential implementations are the most efficient, the actor ones least, and the thread pool version falling somewhere in between. Additionally, it was expected that the sequential implementations would have relatively static memory usage relative to the number of cores available, while a significant portion of the Actors based implementation memory usage would scale linearly with the number of cores available. For language specific implementations; it was expected that C++ will be lowest memory user, followed by Erlang, then Java.

The overall complexity of the code was not expected to vary significantly between the implementations in a single language. It was expected that there would be some slight differentiation between simpler sequential and more complex concurrent implementations. The major source of variance in complexity was expected to be between languages. Specifically, Erlang, a functional language, was expected to be more terse, with a higher complexity per line or statement, but overall lower complexity than the other implementations. For the object orientation (OO) based implementations, C++ and Java, it was expected that the Java code would be less complex but longer, though the difference between these languages would be smaller than between them and Erlang.

1.3 Organization of Thesis

There are six chapters in this thesis; as well as a bibliography of cited references and an appendix for additional data. Chapter 1 is the current introductory chapter. Chapter 2 of this thesis details the background knowledge that forms the foundation for the research presented. Chapter 3 details the methodology of the research: the experiments being run, the parameters being controlled, and an outline of the analysis process. Following this, in Chapter 4, there is a discussion of the implementation details of the various simulation kernels and a qualitative comparison of them. Chapter 5 contains the results, which includes the most relevant tables and graphs for comparing the different implementations. Figures for expanding and elaborating the results appear in an appendix at the end for reference to avoid breaking up the flow of the text. The final chapter details the conclusions reached based on the results and relates them to the hypothesis proposed in this introduction. This chapter will also include suggestions for future research to either extend or reinforce the results of this research.

Chapter 2
Related Work

2.1 Simulation

Simulation is the process of representing a real world system as a simplified abstract model. This representation allows researchers to create experiments to study these systems. This is done due to the difficulty of performing these experiments on the real world system, be it from physical, economic, or moral constraints. Often randomness is introduced into various elements of the model being simulated. This allows the researchers to use statistical observations to fill in details of the model that they either do not have a mathematical model for or are not directly interested in. This randomness, combined with the ability to vary initial conditions of the model, allows researchers to generate results by observing a series of runs of the model and performing statistical analysis.

Simulations of interest for this thesis are those implemented in digital computers, though there is a long history of simulation that exists prior to the development of computer systems. Mathematical modelling is a related and complementary discipline that is often confounded with simulation. In a mathematical model the entire system under investigation is expressed in analytic equations that are solved for a given a set of initial conditions. Simulation, in comparison, focuses on developing executable models for systems that cannot be solved completely using analytic techniques, and then running the models for possible initial conditions to understand their behaviour.

Computer based simulations fall into two broad categories. The first are simulations in which there are people or other real world elements interacting with the system as it runs the simulation. These systems are often referred to as “in-the-loop” systems, with computer games being the most well known examples of “human-in-the-loop” simulations. Other common “in-the-loop” systems include military training systems and industrial machinery testing systems. The second category are simulations in which the entire system is implemented in the computer. There are a variety of simulation techniques that fall into this category including discrete event simulation, discrete time simulation, and statistical/Monte Carlo simulation. Discrete event simulation (DES) is the most heavily researched of these topics and is the type of simulation studied in this research.

One of the defining differences between “in-the-loop” techniques and other techniques is the need for real time responsiveness. Non-“in-the-loop” techniques can spend as much time as needed to achieve the required precision for each unit of time being simulated. “in-the-loop” systems need to receive a response by a hard deadline so that the external observer has appropriate feedback, so are restricted in the fidelity that they can provide.

Simulation is an experimental science, so the building and utilizing of simulations often follows a similar procedure to that of a chemistry or physics laboratory. The first step is to define the system is of interest to the investigation and its significant properties. If possible, the system is defined in terms of answering a falsifiable hypothesis, e.g. selecting a specific workshop to determine which of a set of layouts maximizes productivity. Following the specification of the scope, observations are taken of the real world systems. This can include static measurements of the systems and gathering of statistical data regarding dynamic elements. This step is similar to the theory/pre-reading section of a physical experiment, since it is used to define the domain and parameters of the experiment. Next a model is built to simulate the system. This model must go through a validation and verification process to confirm that it models the real world system within the desired accuracy [32] for the properties of interest. The building and validation of the model are the equivalent of experimental theory in a physical lab setting. Once a valid model is established, simulation runs are performed, which is the equivalent of performing an experiment in a laboratory setting. For stochastic models, randomized runs are performed for each set of initial conditions of interest to generate statistically significant results. This data is then analyzed and conclusions are drawn from it.

2.1.1 Discrete Event Simulation

In a DES, the system being modelled is represented as a state and series of events that affect it. Events have a partial ordering defined for them which is determined by the time that they occur in the model. Events are held in an ordered structure, often a queue, and processed by the simulation algorithm. The DES algorithm implements an event loop that processes the events until either they are exhausted or a termination condition is reached. Common terminations conditions include: a number of events being processed, the model time passing a limit, or a certain criteria regarding the state being satisfied. The model time is defined as the timestamp of the last event that was processed. On each iteration of the event loop. the earliest occurring event is removed from the queue for processing. In the event that there is a tie for earliest, then either a mechanism for tie breaking or atomically processing multiple events is needed. To process the event, the state of the model is mutated in a manner determined by the content of the event and new events may be created. In DES, only the times where events occur are simulated, so the intervening spaces can be skipped over, thus reducing the computation required to simulate sparsely populated periods of time.

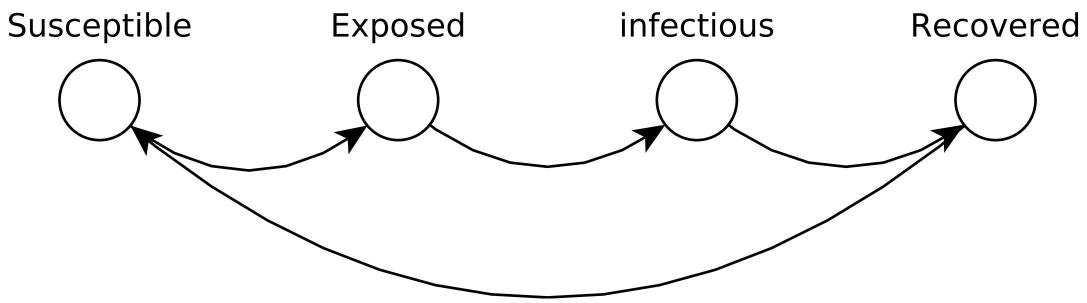

Figure 2.1: SEIRS state diagram

Discrete Time Simulation (DTS) is an alternative discretization paradigm for simulation. Instead of advancing time in the jumps dictated by the time of events, DTS advances the time of the model in fixed time slices. Any events that have occurred during the time slice are taken to have occurred simultaneously. Since events do not need to be stored in a centralized ordering structure, there are two major advantages of this algorithm. In models where events are dense, DES cannot skip over gaps of time, so the cost of maintaining an ordered queue becomes a liability. In a DTS, the events can be stored in an unordered structure like a hash map, which has better run time properties. Additionally, in very dense models, DTSs can be parallelized easily by having workers that perform a scatter and gather on each timeslice. It is possible to create a hybrid between DTS and DES that uses DES style jumping between interesting times and DTS scatter-gather for processing the specific time. This design is the approach taken by the implementations in this thesis.

2.1.2 SEIRS Model

The SEIR (Susceptible, Exposed, Infectious, Recovered/Removed) model is a classic model for epidemiological systems [5]. In this model the population is divided into four groups: susceptible, exposed, infectious, or recovered/removed. Each member of the population can be modelled as an agent encapsulating a finite state machine (FSM). If an agent can return to the susceptible state from the recovered/removed state, then the model is called SEIRS (Susceptible, Exposed, Infectious, Recovered/Removed, Susceptible). All of the models used in this research are SEIRS model.The states and their possible transitions for the agents in a SEIRS model are shown in Figure 2.1. In addition to the current state of the FSM, an agent needs information about what other agents it is in contact with. When the agent is in the infectious state, this contact list is used to determine who is exposed to the infection from the agent.

The transition from susceptible to exposed is driven by an exposure event being sent from an infectious agent that is connected to the susceptible agent. Subsequent state transitions are dictated by stochastic process associated with each transition. In the SEIRS model there is a transition from the recovered/removed state to the susceptible state, which models either immunity fading over time or the infection mutating. From a simulation perspective this means that the states can form a cycle and lead to long running systems.

2.2 Programming Languages

Simula [9], a language which introduced foundational concepts of OO programming, places a heavy emphasis on simulation and has been influential on the development of simulation systems. This entwined development between simulations and OO programming occurred since OO is a natural paradigm to express simulations in. This relationship has continued into the current era, with many modern simulation systems/frameworks are implemented using popular OO languages, such as Java and C++. Some research [15] has also been performed looking into the viability of using functional languages like Erlang for implementing simulations.

2.2.1 C++

C++ [36] is an multi-paradigm general purpose language that was developed in the 80s as a systems programming language. It has OO features for developing large scale programs, i.e. classes, and the performance benefits of low level C style programming. Due to a variety of factors, C++, has been a popular language for many years and continues to be [6, 28, 30, 10, 34]. Its longevity and active standards committee has led to a number of revisions of the language over the years [24, 8, 7]. The C++ code for this research does not use C++11 and later language features, since significant portions of the code base was written before the standard was fully implemented in the GCC [37] and LLVM [25] tool chains.

Syntactically, C++ is a superset of C [23], with some exceptions related to new keywords and features developed after the initial versions of C++ being implemented in different ways in both languages. The C like syntax, combined with removing implicit support for casting `void *` variables, makes C++ a strongly statically typed language, though there are still some exceptions due to unions and other types of pointer casting. Due to its C foundation C++, is able to be programmed following a traditional procedural paradigm. A significant portion of the new language features originally implemented in C++, on top of C, are related to OO programming, so C++ contains support for classes, objects, polymorphism, inheritance, and other features that are expected in a modern OO language. Beyond these OO language features C++, includes support for templates, which allows for types parameterization based programming.

The C++ core language defines the syntax and semantics of programs in the language, but a significant portion of the functionality expected in a modern programming environment is not provided directly. Instead, a standard library is provided that supplies many of the expected building blocks, i.e. abstract containers, generic algorithms, I/O, etc. This allows for the grammar of the language to be as small as possible and for features like data structure APIs to evolve over time without causing backwards compatibility issues in the core language. As the language has evolved, the number and complexity of the features in the standard library has increased. Many of the features that appear in the standard library, i.e. threads and smart pointers, gained popularity with programmers as part of the Boost C++ libraries.

Boost

The Boost C++ libraries [11] are a collection of open source C++ libraries that are designed to integrate easily with the existing standard libraries and codify existing practices in the programming community. Members of the C++ Standards Committee are active in the development and maintenance of the Boost libraries. The libraries operate as a staging ground for functionality to be included in the standard library, with a number of libraries having been directly integrated into the TR1 and C++11 standards.

2.2.2 Java

Java [18], which was developed in the early 90s, is another popular language in the C syntax family that supports OO and other programming paradigms. Java supports a wide variety of programming paradigms through various language features though. However, unlike C++, objects and classes are found at the core of the language design. Conceptually, Java is more of a OO language than C++, with other paradigms being added on top of its OO core. Java implements strong static safe typing. Like C++, it has maintained long time popularity and an active development community, which has led to a series of standards for the language being released. Java is not managed by an ISO standards committee, but instead has de-facto standards released through the Java Community Process. The Java code in this research is compliant with the Java SE6 standard, which was the current standard when the code base was started and the version most familiar to the investigator. Later versions of Java have introduced features like streams and lambda expressions, which are not used. If used, these feature would be expected to decrease the size of the implementation by a small amount.

Though both are a part of the same family of languages, there are substantial differences between C++ and Java deriving from the different goals of their creators [35]. C++ has a heavy emphasis on run time performance and puts the responsibility on the programmer to ensure correctness. This had led to a language with support for directly accessing the hardware, orthogonal support for many programming paradigms/styles, and compilation into machine code. Java, in comparison, has focused on the OO paradigm, secure/safe operations, and portability from its inception. This has led to a language that has garbage collection, runs in a VM, and is generally prescriptive in how tasks are performed [19]. This shifts some of the responsibility for correctness and security from the developer to the language implementer.

In Java the standard library is more tightly coupled with the core language than what is seen in C++. Due to this, it is common to see discussions about Java make no distinction between the language and the standard library. It also means that implementations of the Java tool chain tend to be pinned to a specific version without flags to select what version of the standard to run and there are no partial implementations of specific versions. The standard library in Java is a very feature rich with many modules included (i.e., threading) before they were in C++. Though relatively feature rich, the Java standard library does not include native support for Actors, so a third party library is needed to support them.

Akka

Akka [21] is a third party toolkit for Scala and Java that implements the Actor programming paradigm. The toolkit is written in Scala [29] and has become part of the Scala standard library [22], but also provides API bindings for Java. Scala is explicitly designed to compile to the same byte code as Java for the JVM and thus is easy to integrate into Java programs. The design of Akka's implementation of the Actor paradigm is based off of Erlang's OTP library.

2.2.3 Erlang

Erlang [12, 3] is a concurrent functional programming language that Ericsson started developing for telecommunication systems in the mid 80's [4]. In the late 90's Ericsson decided to open source the language as part of a move away from proprietary technologies. This significantly raised awareness of the language in the larger computer science community and has led to a small, but active, following for the language.

Syntactically, the language is influenced by Prolog, since the original implementation was written in Prolog. Programs written in Erlang run in a VM that supports one time assignment and garbage collection called BEAM. The language implements a strong dynamic typing system that is difficult to statically check [26]. The language was designed with an aim of solving practical problems that Ericsson are essential for telecommunications, so focuses on robustness instead of computational performance. Through its VM, Erlang is able to support code hot swapping and supervision of running processes to this end. Erlang implements concurrency as a first class language feature.

Erlang's concurrency model is based around threads of control and message passing, which looks very similar to the Actors model discussed later in this thesis. It was not the explicit intent of the designers of the language to implement Actors, but given that they were addressing a similar set of problems, they reached the same end state via convergent evolution. With the addition of the Open Telecom Platform (OTP) system libraries, which implements encapsulation through behaviours, the language implements all of the features expected in an Actor framework.

2.3 Concurrency

2.3.1 Threading

Threading is one of the earliest models of concurrency and parallel execution that appears in computer science, being seen in the 60's as part of IBM's System/360 [16] and the THE multiprogramming system [13]. In the threading model of concurrency, the computation is broken up into sequences of instructions, called threads, that can be scheduled in arbitrary order. There exists a scheduling algorithm that is responsible for determining when each of the threads run. In the case of a traditional single core computer, this means that the threads are interleaved as the scheduling algorithm dictates, whereas on a multi-core system more than one thread may be running at a time. The scheduling algorithm may be a separate module that operates as a mediator, but for simple systems it is implicitly implemented in the operating semantics of the threads by having them block and yield control of the processor.

Threads are closely related to processes as a mechanism for achieving multitasking or concurrency. From a scheduling algorithm perspective, there is not a huge difference between them: thread libraries tend to prefer ad-hoc scheduling solutions, whereas processes have an external mediator in the OS. Threading is often implemented within the context of a process, so a program may have one operating system level process allocated to it, but internally it may multiplex its timeslice through threading. This is often done to support both synchronous and asynchronous operations at the same time without having multiple processes. Processes have separate address spaces managed by the operating system and need to use specialized mechanisms to share memory and transfer data, whereas threads share an address space and can communicate directly with each other.

This shared memory model in threads allows for significant potential performance benefits over processes. In process based concurrency, for cross process communication, there needs to either exist a shared section of memory between address spaces or data needs to be shuttled back and forth by the OS. Both of these mechanisms have significant overhead associated with them. In the threading model, two threads have access to the same data by default. Beyond the difference in communication costs, since threads do not have their own separate address spaces, the overhead for switching between them is lower since less processor state needs to be flushed when switching. This also allows threads to be implemented solely within the context of a user land application without support from the OS or hardware. Most modern computer systems supply infrastructure to support threads and improve performance.

There are two major drawbacks to using threading for implementing concurrency. First, since the memory is shared between threads, there needs to be a synchronization mechanism. This guarantees that access to the memory be sufficiently serialized for the consistency requirements of the computations. Synchronization for complex systems can be difficult to implement and even harder to prove correct. The other major issue is that threads share the same process space, so in the event of a crash, all of the threads are lost, since the OS level process is killed. The trade off between threads and processes can be summarized as finer grained control at the cost of higher complexity and penalties for errors.

2.3.2 Actors

The Actors model is a paradigm for programming for concurrent systems that is an alternate to threading, though it is often built on top of it. Actors' theoretical foundation was laid during an active period of research from the early 70's [20] to the mid 80's [1]. Practical implementations of Actors as a framework for concurrency has been and continues to be an active area of development. Some languages, e.g., Erlang and Scala, support Actors semantics in the core language and standard libraries, while others, e.g. Java and C++, depend on third party library implementations.

Actors differs from threading by choosing a different fundamental unit in computation. For threads, computations are divided up at the level of control flow paths running independently, without consideration of what data they are operating on, allowing it to be shared. Actors, in comparison, divides up the computation into units of encapsulated state bundled with behaviour logic. This is very similar to how OO languages divide up computations, though Actors fundamentally considers computations to be concurrent in nature and OO considers them to be sequential.

In the Actors model, the computation is divided into separate elements called actors. Each of these actors encapsulates part of the system state and has its own thread of control. Actors communicate via messages and their behaviour is driven by receiving and reacting to messages. Messages are sent asynchronously and the receiver has a mailbox that queues up messages for the actor to process. In the formal model of Actors, the order of message delivery is not guaranteed, though in implementations it is often the case that a message sent from actor A to actor B is guaranteed to be delivered to the mailbox of B before any subsequent sent messages. This does not guarantee processing order, since B can process its mailbox out of order. It also makes no guarantees about the ordering of messages from B to A or for any other pairings. This is done for practical implementation reasons and to ease reasoning about message passing.

As mentioned in the previous paragraph, an actor is able to process the messages in its mailbox in any order that it desires, though some implementations have performance penalties for out of order processing. The mailbox and the actor associated with it are separate entities, so it is possible for the actor processing a mailbox to change over time. In Erlang, this is part of the robustness mechanism, since it means that an actor can crash and a new instance started to take over operation without needing to inform actors with knowledge of the address that there is a new actor. For addressing messages, this means that the message is not technically going from actor A to actor B, but from actor A to mailbox M, which is processed by actor B. Often this distinction is not material, so discussions speak of a message going from A to B with no mention of M.

The addresses that an actor knows are part of its state; on creation it knows no addresses other than its own. When creating a new actor, the creator learns the address of the new actor. Addresses can also be passed as part of messages, so an actor can communicate to its children the actors it knows of. In implementations of the Actors model, this strict requirement about address knowledge is sometimes broken by having the ability to register globally known addresses for specific services. This, from a theoretical standpoint, can be modelled as there being a root actor that operates as a broker for these global addresses that is known to all of the other actors in the system via always having its address propagated when a new actor is created.

The scope of the addressing mechanism of Actors is another major factor that differentiates it from threads and other concurrency paradigms. At a theoretical level, there is no distinction between addresses that are local to a sender's machine versus ones that are located on other machines. From a practical standpoint, this requires that addresses be a tuple of machine name and unique local address, but it allows for the same syntax for sending locally as remotely. There are significant infrastructure elements that needed to be implemented in the framework to support this, but from a user's perspective there is no difference between concurrent and distributed computing. Threading frameworks only support threads local to the same machine and do not support remote/distributed communication.

In the theory of Actors, when an actor processes a message from its mailbox, there are three types of actions that it may take as part of its response. It may send out messages to addresses, including its own. It may create new actors, which it knows the address of and can send messages to. Finally, it may assign the behaviour for itself for the next message that it receives, though this may just be the same one that it is currently using. For sending messages and creating actors, these operations need to be bounded in the time it takes to operate, which means only a finite number can occur. This requirement is designed to prevent an actor from locking up on receipt of a message and not processing future messages.

In implementations of the Actors model, there is often a significant amount of syntactic sugar present to ease programming and reduce the amount of boiler plate code written. When defining an actor, there are two major components that need to be written, which are inserted into the framework to define an actor. First, some sort of constructor/initializer, which is responsible for setting up the state of the actor, must be defined. This code normally takes in a payload from the creator which may contain addresses and data. The other portion of code that needs to be defined is a message processing loop. Some frameworks implement the loop and the developer just needs to implement the callbacks for specific message types. The callbacks send messages, create new actors, and mutate the state of the actor, e.g. the three operations in the theoretical model. Other frameworks do not provide the looping structure, so the developer must implement fetching the message from the mailbox and filtering it to the correct handling code.

The benefits that Actors provide over threads from a concurrency standpoint mostly involve easing development, not adding expressiveness. The one expressive difference between threads and Actors is not well covered from a pure concurrency perspective, since Actors can also naturally express distributed systems, which threading cannot. In local execution situations, everything implementable with Actors is implementable using threads, though it may not be as simple. This difference in complexity in implementation often arises from the shared memory in threading. This requires that the programmer actively implement synchronization to prevent incorrect access to memory. This shared memory model leads to issues with robustness, which are not encountered in Actors systems.

From a theoretical standpoint Actors, and threads are at different levels of abstraction, which is shown by the fact that implementations of the Actors model are often implemented using threading frameworks. This higher level of abstraction in Actors leads to a significant downside in terms of run time cost of the system. The ease in implementation, due to this abstraction, means that the increased run time cost may be mitigated by additional efforts in optimization. Actor systems need to have a runtime environment or VM to implement their infrastructure to support their abstractions, so they pay a higher overhead for the same operations as threads. An example of this is can be seen in sending data between two threads or actors.

In the threading model, to send data, the code needs to perform the algorithm shown in Algorithm 2.1. The signalling can be achieved through a variety of simple mechanisms depending on the specific system. If the update itself can be done as an atomic operation, then the lock/unlock steps are not needed. This is a very low cost sequence of operations, but requires the programmer to understand the low level details of the concurrency model and reason about synchronization.

```
$D \leftarrow$ data to be sent
$T \leftarrow$ thread to receive data
function SEND_DATA($D, T$)
    $S \leftarrow$ shared memory location
    ACQUIRE_LOCK($S$)
    UPDATE_DATA($S$, $D$)
    RELEASE_LOCK($S$)
    SIGNAL_THREAD($T$)
end function
```

Algorithm 2.1: Algorithm for sending data between threads

For the Actor model, multiple of these low level transfers need to occur to achieve transferring data, which is show in Algorithm 2.2. Depending on the implementation of the runtime, especially if the message is going to a remote address, there may be additional transfers within the runtime. In addition to the minimum two extra transfers, more data needs to be transferred due to the addressing information needing to be sent as well. Beyond the bandwidth cost associated with this, the heterogeneous nature of the data makes it very unlikely that the system implements a low level atomic copy operation that can be used, so each transfer requires explicit synchronization.

2.4 Cyclomatic Complexity

Cyclomatic complexity [27], sometimes called McCabe's complexity, is a measure of complexity in software that quantifies the number of independent paths through an implementation. The complexity value for a piece of code under this metric, also known as cyclomatic number, is the number of independent paths through it. Cyclomatic complexity is closely related to structured testing [38], with the number of test cases to be tested being equal to the cyclomatic complexity of the source code.

To calculate the cyclomatic number of a piece of source code, it first needs be divided up into the fewest number of possible basic blocks, where a basic block has no branching structures in it, but is connected to other blocks by branching structures. This can then be transformed into a graph of the flow control for the code, where the discrete units are the nodes of the graph and the different branching structures are directed edges. For a graph, G, of a program, the cyclomatic number, $M(G)$, is calculated by the formula: $M(G) = E - N + 2$, where E is the number of edges in G and N is the number of nodes in G. When analyzing a program it is common to calculate the value of $M(G)$ for each of the modules of the program (i.e. functions or classes) and sum them up to determine a total cyclomatic number for the program.

```
$D \leftarrow$ data to be sent
$A \leftarrow$ actor to receive data
function SEND_DATA($D, A$)
    $S \leftarrow$ shared memory
    ACQUIRE_LOCK($S$)
    UPDATE_DATA_WITH_MESSAGE($S$, $D$)
    RELEASE_LOCK($S$)
    $T \leftarrow$ runtime message handler thread
    HANDLE_MESSAGE_TX($T$)
end function

$T \leftarrow$ runtime message handler thread
function HANDLE_MESSAGE_TX($T$)
    $S \leftarrow$ shared memory
    ACQUIRE_LOCK($S$)
    $D \leftarrow$ data being sent, from $S$
    RELEASE_LOCK($S$)
    $B \leftarrow$ mailbox of addressee of $D$
    ACQUIRE_LOCK($B$)
    INSERT_MESSAGE($B$, $D$)
    RELEASE_LOCK($B$)
    $A \leftarrow$ actor for $B$
    HANDLE_MESSAGE_TX($A$, $B$)
end function

$B \leftarrow$ mailbox of addressee
$A \leftarrow$ actor for $B$
function HANDLE_MESSAGE_RX($A, B$)
    ACQUIRE_LOCK($B$)
    $M \leftarrow$ REMOVE_MESSAGE($B$)
    RELEASE_LOCK($B$)
    PROCESS_MESSAGE($A$, $M$)
end function
```

Algorithm 2.2: Algorithm for sending data between actors

Chapter 3

Methodology

To test the hypothesis, multiple versions of a simulation system, called kernels, were implemented. These kernels were compared by executing a variety of instances of the SEIRS model on them and analyzing their run time characteristics. The experimental framework which schedules these runs collects information related to amount of time taken and the amount of memory used by each of the kernels. This data was filtered and analyzed by a custom written analysis script. The analyzed results are used in Chapter 5 for discussion and comparison of the different implementations.

3.1 Experimental Process

The details of the simulation kernel implementations are discussed in Chapter 4. This chapter focuses on the process of running experiments using them. The structure of the software modules for this process are depicted in Figure 3.1. This process has three stages: experiment generation, experiment execution, and data analysis. From an implementation standpoint, these stages are three modules in sequence that feed their output into the subsequent one's input. They are implemented in two Python scripts, `experiment_framework` and `results_analyzer`. `experiment_framework` contains both the generation and execution modules and `results_analyzer` contains the analysis module.

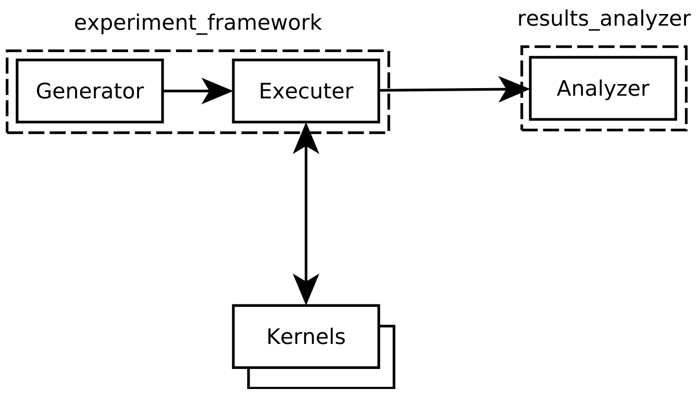

Figure 3.1: Structure of experiment process modules

3.1.1 Experiment Generation

Very early in development, it was found that hand generating all of the JSON specification files for the experiment runs was not feasible, so an automated mechanism was developed. This mechanism is embodied in the experiment generation module and has its own specification JSON files. One specification file contains contains a list of generator definitions. These generator definitions are used to configure the code that creates the specification files for the different experiment runs. Experiment runs are differentiated by the topology of the underlying model, the number of agents in the model, and the duration of the simulation. The other specification file for the generation process contains a listing of all the possible probability distributions that may appear in the generated models. The details of these two specification files appears below.

Generator Specification

The generator specification JSON file contains a single entry `generators`, which contains an array of generator specifications. The full generator specifications JSON file used in the experiments appears in Listing B.2. A snippet of an example of this specification file appears in Listing 3.1. A full example of an element from the `generators` array appears in the previously mentioned listing. The meaning of each of the fields in the entries appears below.

`base_name`
: String that is used as a prefix in values that refer to and/or identify specification files created by this generator.

`generator_type`
: Generation algorithm to be used to generate a graph of agents and connections for the model.

`generator_params`
: Array of floats that are passed in as parameters to the generator. The required arity of this array and semantic meaning are defined by the value of `generator_type`.

`looping`
: Boolean value that indicates whether or not agents in the models should have a transition from the `recovered` state back to `susceptible`. False makes the model a SEIR model and true a SEIRS model.

`agents_start`
: The lower bound for the range of the agent counts used in model generation.

`agents_end`
: The upper bound for the range of the agent counts used in model generation.

The valid values for `generator_type` are listed below. These graph generators were selected to exercise a wide range of behaviours of the implementations and not meant to be representative of empirical observations from real world systems.

`complete` *Complete connected graph*

Every vertex in the graph has an edge to every other vertex. Takes no parameters.

`complete-bipartite` *Complete bipartite graph*

Graph is divided into two partitions, A and B. Every vertex in partition A has edges connecting it to every vertex in partition B. Takes 2 parameters that define the ratio of sizes of A and B.

`Watts-Strogatz` *Small world graph*

Graph is generated using the Watts-Strogatz algorithm [39] and has the property that distances between two vertices increase logarithmically with the number of vertices. This property is known as the small world property. Takes two parameters, the first specifying the number of neighbours each vertex has an edge to in the initial ring graph and a second value specifying the probability that an edge is rewired.

`circular-ladder` *Circular ladder graph*

Graph is composed of two rings, A and B, of the same number of vertices. The *ith* vertex of A has an edge connecting it to the *ith* vertex of B and vice versa. Takes no parameters.

`Cycle` *Simple ring graph*

Graph is composed of a cycle of vertices, where each vertex has edges connecting it to its nearest neighbours. This is also known as a ring. Takes no parameters.

`periodic-2grid` *2D grid with wrapping along the borders*

Graph is composed of a two dimensional grid of connected vertices. Vertices on the border of the grid have edges connecting them with the far side of the grid to create cycles. Takes in two parameters that define the ratio of the length of the dimensions.

`nonperiodic-2grid` *2D grid with no wrapping*

Graph is composed of a two dimensional grid of connected vertices. Vertices on the edges of the grid do not have a edges connecting them with the far side of the grid. Takes in two parameters that define the ratio of the length of the dimensions.

`hypercube` *N-dimensional hypercube graph*

Graph vertices are mapped onto the corners of a n-dimensional hypercube with edges connecting vertices on corners that share a side. Takes no parameters.

`star` *Simple hub and spoke graph*

Graph has one central vertex with every other vertex connected to it via an edge. Takes no parameters.

`wheel` *Star graph with ring connecting ends of the spokes*

Graph has a ring of vertices all connected to a single central hub vertex by spoke edges. Takes no parameters.

`erdos-reyni` *Random graph*

Graph is a random graph generated using the Erdős-Rényi binomial algorithm [14, 17]. Takes in a parameter to control the likelihood of an edge connecting two vertices appearing in the generated graph.

```
{
    "generators" : [
        {
            "base_name": "CompleteGenerator",
            "generator_type" : "complete",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 20,
            "agents_end": 2000
        },
        ...
    ],
}
```

Listing 3.1: Example experiment generator JSON specification

Distribution Specification

The distribution specifications JSON file contains a single field `distributions`. `distributions` contains an array of all the potential distribution specifications. Like the graph generators above these distributions are meant to exercise the implementations and not represent real world systems. The full distribution specifications JSON file used in the experiments appears in Listing B.3. An example of the distribution specification appears in Listing 3.2. The meaning of the fields in elements of the `distributions` are as follows.

`label`

Unique string identifying this distribution. The agents use this string to refer to this distribution and this appears in logging to identify the distribution being referred to.

`type`

The type of random number distribution to be used to generate values for this distribution.

`params`

An array of parameters for the random number distribution. The arity and semantics of this array are determined by the value of `type`.

`scale`

Scaling factor to apply to random numbers generated by this distribution. This value is optional, with a default value of 1.0

The valid values for the `type` field appear below.

`gaussiantail`

Generates a random number from the upper half of a Gaussian distribution. Takes in two values, the first is a lower bound on the value returned and the second is the standard deviation of the Gaussian distribution.

`exponential`

Generates a random number using an distribution that displays exponential decay. Takes in a single parameter which is the mean of the distribution.

`flat`

Generates a random number using a distribution such that all values in the range are equally likely. Takes in two parameters which define the lower and upper inclusive boundaries of the range.

`lognormal`

Generates a random number using a distribution that returns numbers whose logarithm follows the Gaussian distribution. Take in two parameters, the mean and standard deviation of the Gaussian distribution.

`poisson`

Generates a random number following the Poisson distribution. Takes one parameter which defines the mean of the distribution.

`bernoulli`

Generates a random number of either 0 or 1 depending on a single sample, called a Bernoulli experiment. Takes in single parameter defining the probability that 1 is returned.

`binomial`

Generate a random number by performing a series of Bernoulli experiments and counting the successes. Takes in two parameters, the probability value of success in an individual experiment and the number of experiments to run.

`negativebinomial`

Generate a random number by performing a series of Bernoulli experiments and counting the number of attempts required to reach a specified number of failures. Takes in two parameters, the probability value of success in an individual experiment and the number of failures attempting to be reached.

`geometric`

Generate a random number by performing a series of Bernoulli experiments and counting the number of attempts required to reach a single success. Takes in one parameter, the probability value of success in an individual experiment.

```
{
    "distributions" : [
        {
            "label": "DistributionPoisson10x20",
            "type": "poisson",
            "params": [
                10
            ],
            "scale": 20.0,
        },
        ...
    ],
}
```

Listing 3.2: Example distributions JSON specification

For executing simulations, a kernel takes in a JSON file with an experiment specification, which defines a single experimental run. As part of this experiment specification, there is a file name for a model specification, which is another JSON file. A single model specification may be referenced by many experiment specifications. The experiment generation module is responsible for generating these files during the run of the `experiment_framework`.

The high level algorithm used for generating simulation specifications appears in Algorithm 3.1. This algorithm is augmented in the following sections with simulation execution to define the algorithm used in `experiment_framework`.

3.1.2 Experiment Execution

Execution of the experiments requires three inputs, one of which is supplied by the caller of `experiment_framework` and the remaining two are the outputs of the experiment generation module. From the caller, it receives a JSON specification file, which contains a set of kernel binaries to run the experiments on. From the gen-

```
$G \leftarrow$ generator specifications
$D \leftarrow$ distribution specifications
**function** EXPERIMENT_GENERATOR($G, D$)
    $M \leftarrow$ model files
    $E \leftarrow$ experiment files
    **for each** generator $g$ **in** $G$ **do**
        **for each** agent count $c$ **in** GENERATE_RANGE($g$[agent_start], $g$[agent_end]) **do**
            $M[m] \leftarrow$ GENERATE_MODEL_FILE($g$, $c$)
            **for each** event limit $l$ in GENERATE_RANGE($g$[event_start], $g$[event_end]) **do**
                $E[e] \leftarrow$ GENERATE_EXPERIMENT_FILE($g$, $D$, $c$, $l$, $M[m]$)
            **end for**
        **end for**
    **end for**
    **return** $E$, $M$
**end function**
```

Algorithm 3.1: High level algorithm for experiment generator module

eration module, the execution module receives a set of experiment specifications and their related model specifications. The details of these specification files are listed below.

Kernel Specification

The kernel specification JSON file contains a listing of all the kernel binaries to be tested. It contains a single entry `kernels` which contains an array of strings. Each of the strings is a relative or absolute path to a kernel binary. The full kernel specification JSON file used in the experiments appears in Listing B.1. Listing 3.3 contains a snippet of an example of the kernel specifications.

```
{
    "kernels": [
        "kernels/rysim_des_actor_erlang/rysim_des_actor",
        ...
    ]
}
```

Listing 3.3: Example of experiment framework kernel JSON specification

Experiment Specification

The experiment specification JSON file contains values that define a single experiment run. An example of this type of specification file appears in Listing 3.4. The meaning of each of the entries in this specification are as follows.

`experiment_name`

The base of the tag that is be applied to this run. This base is composed with other information like the number of agents and the event limit to generate a unique identifier for this run.

`model_filename`

Relative path from this file to the model specification to be used in this experiment. Often these files are in the same directory.

`type`

The type of generator used to create the model's graph. This information is used for categorizing the results of this experiment for further analysis.

`event_limit`

The number of events to run the simulation to. This is a soft limit, so the current timeslice is still completed after this limit is reached.

```
{
    "experiment_name": "CompleteGenerator",
    "model_filename": "CompleteGenerator_model_20.json",
    "type": "complete",
    "event_limit": 20000
}
```

Listing 3.4: Example of experiment JSON specification

Model Specification

The model specification JSON file includes the initial state for all of the agents in the model, a listing of the distributions that are used, and a preamble containing general information about the model. Listing 3.5 contains an abbreviated example of a model specification. The meaning of the fields in this specification appear below.

`model_name`

A string which is used to identify all of the experiment runs that contain this specific model in them.

`total_connections`

The total number of connections in the model between the agents. This translates to the number of edges in the underlying graph. This value is one of the independent variables in the analysis of the results.

`agents`

An array containing all of the agent specifications for the model.

`distributions`

An array containing all of the distribution specifications referenced by the agents. The meaning of the fields for elements in this array are given earlier in this chapter.

```
{
    "model_name": "CompleteGenerator_20",
    "total_connections": 380,
    "agents": [
        {
            "label": "Agent_0",
            "state": "susceptible",
            "s2e": "DistributionPoisson10x20",
            "e2i": "DistributionBinomial0.7x20",
            "i2r": "DistributionGaussianTail0x25",
            "r2s": "DistributionGaussianTail10x25",
            "connections": [
                "Agent_1",
                ...
            ]
        },
        ...
    ],
    "distributions": [
        {
            "params": [
                0.8
            ],
            "scale": 1.0,
            "type": "Geometric",
            "label": "DistributionGeometric0.8x1"
        },
        ...
    ],
}
```

Listing 3.5: Example model JSON specification

An example agent specification is given in Listing 3.6. The meaning of the fields in the agent specification appear below

`label`

Unique string identifying this agent. Other agent specification use this string to refer to this agent and this appears in logging to identify the agent being referred to.

`state`

Initial state of the agent at the beginning of the simulation, may be one of the following values: `susceptible`, `exposed`, `infectious`, or `recovered`.

`s2e`

Identifier for the distribution to be used for calculating transitions between the `susceptible` and `exposed` states.

`e2i`

Identifier for the distribution to be used for calculating transitions between the `exposed` and `infectious` states.

`i2r`

Identifier for the distribution to be used for calculating transitions between the `infectious` and `recovered` states.

`r2s`

Identifier for the distribution to be used for calculating transitions between the `recovered` and `susceptible` states. This value may not be present in models that don't have state looping, i.e. SEIR models.

`connections`

Array of agent label strings that is the set of agents in the simulation that this agent may potential infect. This is a directed relationship, so agents in this set are not guaranteed to be able to infect the agent being defined.

```
{
   "label": "Agent_0",
   "state": "susceptible",
   "s2e": "DistributionPoisson10x20",
   "e2i": "DistributionBinomial0.7x20",
   "i2r": "DistributionGaussianTail0x25",
   "r2s": "DistributionGaussianTail10x25",
   "connections": [
       "Agent_1",
       "Agent_2",
       "Agent_3",
       "Agent_4",
       "Agent_5",
       ...
   ]
},
```

Listing 3.6: Example agent JSON specification

Given the previous discussion of the modules involved, there is a naive structure in which `experiment_framework` could be organized. This would have a monolithic generator implementation that takes in the generator specification and produces model and experiment specifications. These specifications, with the kernel specifications, would then be used by a monolithic execution implementation that is responsible for running the experiments and collecting the results. The high-level algorithm for this is shown in Algorithm 3.2. This design is amendable to having the modules as separate binaries, which allows for the possibility of running generation once and having many different executions in different environments.

```
$G \leftarrow$ generator specifications
$D \leftarrow$ distribution specifications
$K \leftarrow$ kernel specifications
function NAIVE_FRAMEWORK($G, D, K$)
    $M \leftarrow$ model files
    $E \leftarrow$ experiment files
    for each generator $g$ in $G$ do
        for each agent count $c$ in GENERATE_RANGE($g$[agent_start], $g$[agent_end]) do
            $M[m] \leftarrow$ GENERATE_MODEL_FILE($g$, $c$)
            for each event limit $l$ in GENERATE_RANGE($g$[event_start], $g$[event_end]) do
                $E[e] \leftarrow$ GENERATE_EXPERIMENT_FILE($g$, $D$, $c$, $l$, $m$)
            end for
        end for
    end for
    $R \leftarrow$ results
    for each experiment $e$ in $E$ do
        for each kernel $k$ in $K$ do
            $R[e, k] \leftarrow$ PERFORM_EXPERIMENT($e$, $k$)
        end for
    end for
    return $R$
end function
```

Algorithm 3.2: High level algorithm for naive `experiment_framework`

This design was originally implemented for this thesis and found to have some significant drawbacks that led to a redesign of the `experiment_framework` implementation. The naive design requires all of the outputs of the generation module to be created and stored before the execution module can consume them. This was an issue since the storage capacity in the executing environment was limited and storing of these results led to a negative effect on performance and stability. Reducing the amount of data stored during execution mitigated these problems.

The algorithm for the version of `experiment_framework` that was used appears in Algorithm 3.3. In this design each model is generated, used, then discarded. While a model is present a series of experiments that use it are created, run, then discarded. During the lifetime of an experiment it is executed by each of the kernels and results recorded. This design requires only one instance of the experiment and model specifications to be stored at a time. The major drawback to this design is that the execution of the entire simulation set can take longer, since experiment generation cannot be performed a prior.

```
$G \leftarrow$ generator specifications
$D \leftarrow$ distribution specifications
$K \leftarrow$ kernel specifications
function EXPERIMENT_FRAMEWORK($G, D, K$)
    $R \leftarrow$ results
    for each generator $g$ in $G$ do
        for each agent count $c$ in GENERATE_RANGE($g$[agent_start], $g$[agent_end]) do
            $m \leftarrow$ GENERATE_MODEL_FILE(g, c)
            for each event limit $l$ in GENERATE_RANGE($g$[event_start], $g$[event_end]) do
                $e \leftarrow$ GENERATE_EXPERIMENTFILE($g$, $D$, $c$, $l$, $m$)
                for each kernel $k$ in $K$ do
                    $R[e, k] \leftarrow$ PERFORM_EXPERIMENT($e$, $k$)
                end for
            end for
        end for
    end for
    return $R$
end function
```

Algorithm 3.3: High level algorithm for `experiment_framework`

The environments that `experiment_framework` were run in are virtualized general computing systems, so there are variances in the load on the physical machines beyond the control of experimenters. Within a virtualized instance, there are other randomized sources of load related to the OS and other system processes that occur in most modern general computing environments. These two factors are major contributors of noise in the data, since the duration of the execution and sampling times are less deterministic. To combat this each experiment was run multiple times, 50, and the results averaged as part of the analysis phase.

There are three sets of information that appear in the results for a simulation run. The first of these is identification/classification information. This includes entries about the model being run, the machine being run on, the kernel being used. This information is derived from the specification file and from the output of the kernel itself. The next set of information is the values of the independent variables for the experiment, which includes the number of events run, the number of agents in the model, the number of connections in the model. Most of this information is derived from the generated specification, with the exception of event count related information. Since when the event limit is reached the current time slice is completed the number of events actually run and the limit differ. The intended event limit is used to bucket the results and the actual event count is used as an independent variable. The final set of data in the result is the dependent variables, which the CPU time taken to run the simulation and the maximum memory usage.

The dependent variable information was collected during a simulation run by having the framework execute the kernel in the context of a script that is tracking the values of the variable. This script takes in the kernel to be run with the specification files and runs it as a child process. Since it knows the identifier of this child, the script is able to walk the process information provided by the OS to sum up the CPU time and resident memory set allocated to the kernel and its child processes. It samples these value at a rate of 120 Hz and reports the values to the framework at the end of execution. This script also supports killing long running, over 1 minute, runs to limit the total time spent collecting data.

This sampling based mechanism was designed to minimize the impact of the instrumentation on the results. It also has the advantage of not requiring instrumentation within the kernel, thus preserving the validity of the code complexity analysis. However, this does come at a cost related to fidelity of the data. If the kernel allocates and deallocates a large block of memory between samples, then the max memory usage value does not reflect this. This was most noticeable in some of the short lived C++ runs where the kernel was able to perform all of the operations in a very short interval, so a memory usage of 0 was reported. These results were not usable and thus filtered out of the data.

The choice of CPU time is a standard parameter to measure on multi-tasking systems like modern OSs, since the wall time is heavily influenced by the load on the system. There is still some effect of load on the CPU time since, though the time spent loading and unloading a process is not counted against the CPU time and memory caches may be cold after a context switch. Since the experiments are being running in a virtualized environment there is be additional layer of variability introduced during periods of high load as the entire virtual machine instance may be being switched in and out by the underlying physical machine.

For reporting memory usage the maximum resident set size (RSS) was tracked. The RSS was chosen over the virtual memory size (VSZ), since the RSS tracks the memory held in RAM and is a better reflection of the actual working set size. Increased RSS values generally have a more pronounced effect on systems performance than similar increases to VSZ. For many of the VM based kernels the VSZ balloons on start to a fixed value, but the RSS tracks a value more in line with the size of operation being performed. The choice of maximum usage was done since this defines a lower bound for the amount of RAM that is needed to execute the model efficiently.

The results from each simulation run were written out to a SQLite database for consumption latter by `results_analyzer`. A database was chosen as the output format since it allowed for `experiment_framework` to write and commit results instead of queuing them up until the end of execution. This was needed since on some of the smaller machines being used it was common for the both the kernel and `experiment_framework` to be killed by the OS on large models. When this occurred the framework script would need to be manually restarted for the remaining experiments, but any cached results would have been lost. The `result_analyzer` also originally saved its work into a database, so it was natural to take input in this format. The final version of these scripts don't heavily use the database and it could probably be replaced with a more lightweight solution like a log file.

3.1.3 Data Analysis

The `results_analyzer` script takes multiple databases produced by `experiment_frameworks`, filters and merges the entries, processes the results, and produces a variety of artifacts. The high-level algorithm of this is shown in Algorithm 3.4.

When reading in the results from the databases the `results_analyzer` does some filtering to remove results with unusable values like zero length run times that were missed by the framework scripts. Since the bucket information for the event limit is not included in the entry it also attempts to determine the proper bucket for entries. Each entry is tested to find if there is a an event count bucket within 10% of its value. If so it is added to that bucket, otherwise it is discarded. This means that runs that exited early or had a very long final time slices are removed from the data, which reduces the variability of the data. This comes at the cost of throwing out potentially significant results if a large number of the runs that were supposed to be in a specific bucket fall outside of this threshold range, which is likely if the system is exhibiting non-linear/chaotic behaviours. The bucketing does also has the potential for other errors due to a run being so far outside its intended bucket that it falls into another bucket.

The results were grouped using three different sets of keys to create three different sets of analysis and artifacts. For each grouping the values are filtered into groups based on the specific key associated with them and then have their bucketed values averaged. The first grouping with be based purely on the kernel

```
$D \leftarrow$ experiment_framework databases
function RESULTS_ANALYZER($D$)
    $R \leftarrow$ results database
    for each database $d$ in $D$ do
        for each result $r$ in $R$ do
            if RESULT_USABLE($r$) then
                ADD_BUCKET_INFORMATION($r$)
                INSERT($R$, $r$)
            end if
        end for
    end for
    $K \leftarrow$ CREATE_RESULT_TABLE((kernel), $R$)
    GENERATE_MULTI_LINE_PLOTS($K$)
    GENERATE_FITS_TABLES($K$)
    GENERATE_SCORES_TABLES($K$)
    $KM \leftarrow$ CREATE_RESULT_TABLE((kernel, machine), $R$)
    GENERATE_MULTI_LINE_PLOTS($KM$)
    GENERATE_FITS_TABLES($KM$)
    GENERATE_SCORES_TABLES($KM$)
    $KMT \leftarrow$ CREATE_RESULT_TABLE((kernel, machine,model type), $R$)
    GENERATE_FITS_TABLES($KMT$)
    GENERATE_SCORES_TABLES($KMT$)
end function
```

Algorithm 3.4: High level algorithm for `results_analyzer`

used to run the experiment. This data was expected to be the noisiest, though easiest to analyze. The next key groups the results based on the kernel and the machine that the experiments were run on. This was expected to remove the largest source of noise, since the performance characteristics of the different execution environments are significantly different. The final grouping that was analyzed is keyed based on kernel, machine and model type. This was expected to produce the highest fidelity results, but take the most computation effort to analyze.

Linear Regression

For all of the groupings there were per kernel single and multivariate regressions run on the data. These regressions were performed generate a representative fit of data that could be used in further analysis. The single variate regressions compared the maximum memory usage and CPU time to agent, connection, and event counts. It is unlikely that all of these values are linear independent, since the number of connections has an upper limit based on the number of agents in the system. Additionally, lengths of message queues for each time slice is influenced by the underlying topology, so the resource usage per event is expected to have some dependence on the other variables not examined in this thesis.

To address this dependency issue, the results underwent multi-variate regression of all of the possible independent variable combinations. It is possible that either agent or connection count dominates in the regression, so using both would lead to over-fitting. All of the multi-variate fits were performed for both for CPU time and maximum memory usages. The two dimension fits that were performed are event and agent count, event and connection count, and agent and connection count. The three dimensional fit that was performed is agent, connection and event count.

3.1.4 Ranking

As discussed previously, there was a set of linear regressions calculated that are attempting to predict the CPU time and the maximum memory usage. For each of the these fits R^2, the coefficient of determination, was calculated. This value was averaged across all of the regressions for a specific grouping and type of fit to produce a score for how well that choice of regression fits the data. The combination of grouping key and linear regression with the best R^2 value was used for further analysis of the data.

For both of the dependent variables, CPU time and max memory usage, the desire of a developer is to minimize them. Using the calculated fit a kernel, a score value for a specific selection key was calculated by integrating the function defined by the fit over the range of the independent variables used to perform the fit. Kernels with a lower number from this calculation perform better than those with higher values. Since the analysis being performed is a linear regression, the fits are lines and their higher dimensional equivalents, meaning the integral was calculated using the trapezoidal rule.

The above process was used for generating scores for the kernels for each of the possible key values. Then for each key value the kernels was stack ranked based on the value of this score, first being the lowest value and last being the highest value. These rankings were tabulated across all of the keys to determine a global ranking of the kernels with respect to CPU time and memory usage.

3.1.5 Machine Comparison

The previous ranking analysis process produced an optimal fitting selection for CPU time and maximum memory usage. These two fitting selections were used to perform further analysis of the data. Each kernel had its results grouped by the optimal selection key and the definite integral for each of the unique fits calculated over the range of the independent variables. These values were partitioned again by the specific machine that the experiment was run on. These per kernel per machine partitions were then averaged to give a single value for each unique combination of machine and kernel. Each machine type has a specific number of cores, so for these fits the number of cores does not vary within a single data set.

Each of the machines used in the experiment had a unique number of processing cores available. Thus the partitioning and averaging in the previous paragraph had produced a set of data with the CPU time or maximum memory per kernel vs the number of cores available. Depending on the independent variables used this data have units of mS $\cdot$ event $\cdot$ agent $\cdot$ connection, $mS \cdot$ event $\cdot$ agent, or $mS \cdot$ event $\cdot$ connection and $kB \cdot$ event $\cdot$ agent $\cdot$ connection, $kB \cdot$ event $\cdot$ agent, or $kB \cdot$ event $\cdot$ connection. The dimensional analysis for CPU time versus all three independent variable is shown in Figure 3.2. The analysis for the other cases follows a similar pattern. These units are not particularly useful from a predictive standpoint, but they do allow for tracking of trends in response to changing the number of core and seeing how the kernels compare with each other. This data was plotted on a multi-line plot with all of the kernel values to look for trends and see how the kernels compare. To examine variability in the data single kernel plots with boxes and whiskers were also produced.

3.2 Controlled Parameters

There are five parameters that were controlled through out the experiments to attempt to exercise the dependent variables. Two of these parameters, underlying graph type and execution environment, were used to categorize the runs into groups. The other three parameters; agent count, connection count, and event count, were used as the independent variable in the fitting process.

The graph types used for the connection topology in the model influence the dependence between the number of agents and connections that appear in a specific model. To attempt to limit the influence of this on the results, a wide variety of graph generators are used in the model generation process. This set of graph generators is not exhaustive, so there may still exist biases in the results related to this.

$$
\begin{aligned}
x, y, z &= \text{Independent variables for fit (events, agents, and connections)} \\
A, B, C &= \text{Calculated fitting parameters} \\
f(x, y, z) &= Ax + By + Cz
\end{aligned}
$$

$$
\begin{aligned}
[f(x, y, z] &= \text{mS} \\
&= [Ax + By + Cz] \\
&= [A]\text{event} + [B]\text{agent} + [C]\text{connection}
\end{aligned}
$$

$$
\begin{aligned}
[A] &= \frac{\text{event}}{\text{mS}} \\
[B] &= \frac{\text{agent}}{\text{mS}} \\
[C] &= \frac{\text{connection}}{\text{mS}}
\end{aligned}
$$

$$
\begin{aligned}
F(x, y, z) &= \iiint f(x, y, z)\, dx\, dy\, dz \\
&= \iiint (Ax + By + Cz)\, dx\, dy\, dz \\
&= \iiint Ax\, dx\, dy\, dz + \iiint By\, dx\, dy\, dz + \iiint Cz\, dx\, dy\, dz \\
&= \frac{A}{2}x^2yz + \frac{B}{2}xy^2z + \frac{C}{2}xyz^2
\end{aligned}
$$

$$
\begin{aligned}
[F(x, y, z)] &= [\frac{A}{2}x^2yz + \frac{B}{2}xy^2z + \frac{C}{2}xyz^2] \\
&= [A] \cdot \text{event}^2 \cdot \text{agent} \cdot \text{connection} + [B] \cdot \text{event} \cdot \text{agent}\dot{}\text{connection} + [C] \cdot \text{event} \cdot \text{agent} \cdot \text{connection}^2 \\
&= \frac{\text{mS}}{\text{event}} \cdot \text{event}^2 \cdot \text{agent} \cdot \text{connection} + \frac{\text{mS}}{\text{agent}} \cdot \text{event} \cdot \text{agent}\dot{}\text{connection} \\
&+ \frac{\text{mS}}{\text{connection}} \cdot \text{event} \cdot \text{agent} \cdot \text{connection}^2 \\
&= \text{mS} \cdot \text{event} \cdot \text{agent} \cdot \text{connection}
\end{aligned}
$$

Figure 3.2: Dimensional Analysis for CPU time vs Events, Agents, and Connections

The execution environments for the experiment were all Amazon EC2 instances running the same base system image. They were all in the same region and running around the same time. These instances are virtualized, which introduces the potential for significant error if the load on the underlying physical machine varies across the experiment run. To attempt to alleviate this, the working set of the experiments to run for any specific environment was broken up and run across multiple instances at the same time.

The process for generating experiments discussed previously involves specifying the number of agents and the event limit in addition the model type. The agent and event values were not all manually specified in the configuration files. Instead the start and end values of their ranges are specified. The generator module calculates a set of values to use following a simple doubling algorithm. Doubling was chosen instead of fixed stride for the interval length, since for fixed stride either the range of values would have to be very small or the stride size very large to keep the number of experiments to a reasonable amount.

3.2.1 Complexity Analysis

The majority of the computation for calculating the cyclomatic complexity and other code metrics was performed using a third party tool, SonarQube. SonarQube [31] is an open source project for managing code quality, which includes generating code metrics. It supports Java directly and has community plugin support for C++ and Erlang. For the purposes of analysis in this thesis, it provides the number of statements, number of functional lines of code, and the cyclomatic number for the program. It does not supply any of the compound metrics like complexity per statement or lines of code per function, which were calculated by hand.

Chapter 4

Implementation

All of the kernels used in this thesis implement the same high level system using different paradigms and languages. The different kernels follow the same core pattern, but vary significantly in their overall structure and implementation details. In spite of this, there is a significant amount of shared code between the implementations. An example of this can be see in the code for sampling from statistical distributions for random number generation [33]. Each kernel in the same languages used the same implementation and wrapper for implementing these algorithms. The first section this chapter contains a high level description of how to implement a DES SEIRS system and each of the subsequent sections describe how each kernel implements this, focusing on how they differ from the original pattern.

The source code for the implementations can be found at https://github.com/google/rysim.

4.1 DES

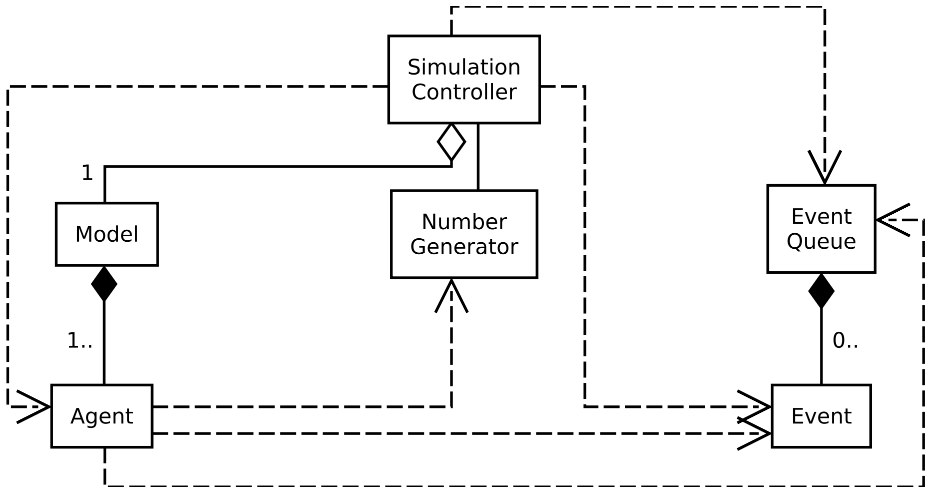

Figure 4.1: Structure of simulation modules

The high level structure of a generic DES implementation appears in Figure 4.1. In this design the `Simulation Controller` module is the centre point of control in the system, being responsible for initializing

```
$ES \leftarrow$ experiment specification
**function** SIMULATION_INITIALIZATION($E$)
    $S \leftarrow$ seed for random number generation, from command line
    $N \leftarrow$ CREATE_NUMBER_GENERATOR($S$)
    $L \leftarrow$ PARSE_EVENT_LIMIT($ES$)
    $F \leftarrow$ PARSE_MODEL_SPECIFICATION_LOCATION($ES$)
    $MS \leftarrow$ READ_MODEL_SPECIFICATION($F$)
    $D \leftarrow$ PARSE_DISTRIBUTIONS($MS$)
    **for each** distribution $d$ **in** $D$ **do**
        ADD_DISTRIBUTION($N$, $d$)
    **end for**
    $Q \leftarrow$ CREATE_EVENT_QUEUE
    $M \leftarrow$ CREATE_MODEL($Q$)
    $A \leftarrow$ PARSE_AGENTS($MS$)
    **for each** agent $a$ **in** $A$ **do**
        ADD_AGENT($M$, $a$)
    **end for**
**end function**
```

Algorithm 4.1: High level algorithm for simulation initialization

the system and running the event loop. The high level algorithm for initializing the system appears in Algorithm 4.1. During initialization the controller creates the `Number Generator`, `Model`, and `Event Queue` modules. The `Number Generator` module is initialized with a random seed from the user and distribution specifications from the model specification file. It operates as a singleton that encapsulates the random number generation logic for all of the `Agent` modules. The `Model` is a container of for instances of the `Agent` module that is owned by the `Simulation Controller` and initialized using the agent specifications from the `agents` field of the model specification file. The `Event Queue` is an ordered container of `Event` instances and is initially empty. It is a singleton accessible from multiple locations in the system,, with `Event` instances from each `Agent` in the `Model` being enqueued to it and the `Simulation Controller` dequeueing from it. As part of creating the `Agent` instances for `Model`, the `Event Queue` is seeded with the initial `Events` for the experiment.

After completion of initialization, the simulation system proceeds into the main body of operation, which is implemented in the simulation loop. The high level algorithm for the simulation loop appears in Algorithm 4.2. The simulation loop is designed to continue running until either the `Event Queue` has been exhausted or the event limit has been reached. Each timestamp processed by the loop is handled as a two phase operation.

```
$Q \leftarrow$ event queue
$M \leftarrow$ model
$L \leftarrow$ event limit
function SIMULATION_LOOP($Q$, $M$, $L$)
    $C \leftarrow$ number of events processed, initially 0
    $E \leftarrow$ next event to be processed, initially unset
    while $C < L$ && $|Q| > 0$ do
        $E \leftarrow$POP_EVENT($Q$)
        PROCESS_EVENT($M$, $E$)
        $C \leftarrow C + 1$
        $T \leftarrow$ GET_EVENT_TIMESTAMP($E$)
        while $|Q| > 0$ && NEXT_TIMESTAMP($Q$, $T$) do
            $E \leftarrow$POP_EVENT($Q$)
            PROCESS_EVENT($M$, $E$)
            $C \leftarrow C + 1$
        end while
        ADVANCE_STATE($M, T$)
    end while
end function
```

Algorithm 4.2: High level algorithm for simulation loop

First, all of the events for the current timestamp are batched and sent to the `Model`. The `Model` fans out the `Events` to the appropriate receiving `Agent` instances. Once this is completed, the `Model` is told to advance the state. This command is fanned out to the `Agent` instances, which causes them to determine the state transition that should occur on this timeslice and emit the appropriate `Event` instances.

This two phase operation is needed since an `Agent` may receive multiple `Event` instances on the same timestamp, but should only undergo a single state transition. This becomes an issue since other `Agent` instances send exposed `Event` instances, so it is possible for an `Agent` to receive a susceptible and an exposed `Event` on the same timestamp. Without the two phase operation, the `Event` instances would have to be processed in the order they were received, which is not guaranteed to be consistent. Thus, a pair of `Event` instances may be run two different times and have different effects on the model. New `Event` instance generated during the advance state phase must have a timestamp in the future, new timestamp $\geq$ current timestamp + 1, to make sure that all of the `Event` instances for the current timestamp were processed in the initial phase.

4.2 C++ Implementation

4.2.1 Sequential Implementation

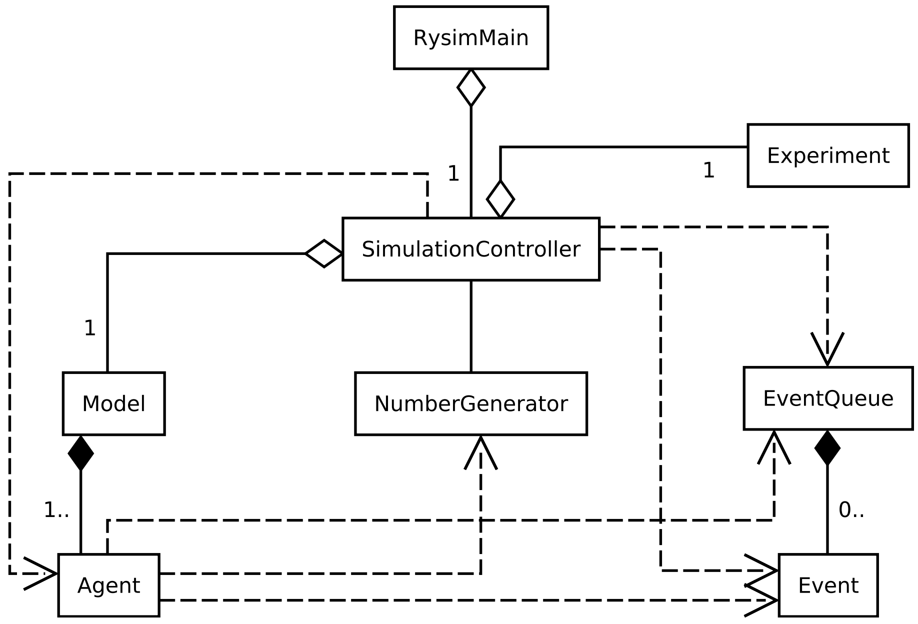

Figure 4.2: Structure of C++ sequential implementation classes

For C++ there is a single implementation, which follows a traditional sequential DES design. The structure of this code is similar to the design discussed above and appears in Figure 4.2. The modules of the previous

```
bool SimulationController::run() {
    timestamp_ = 0;
    while (eventCount_ < eventLimit_ && !EventQueue::get()->isEmpty()) {
        Event event = EventQueue::get()->pop();
        model_.processEvent(event);
        eventCount_++;
        timestamp_ = event.getTimestamp();
        while (EventQueue::get()->isNextEventAt(timestamp_)) {
            event = EventQueue::get()->pop();
            model_.processEvent(event);
            eventCount_++;
        }
        model_.advanceState(timestamp_);
    }
    return true;
}
```

Listing 4.1: Implementation of simulation loop in C++ sequential kernel

design, Figure 4.1, have been converted into classes in this implementations. The changes in the structure relate to how the implementation interacts with user input. A class, `RysimMain`, has been added as an entry point to the program, containing code related to command line argument parsing and drives the `SimulationController` through the stages of operation. The `Experiment` class is an intermediary form of the experiment specification data which is used to make operations more idiomatic. The `SimulationController` class is also augmented to track and report information about the experiment run, which was not discussed in the high level design.

Listing 4.1 contains the implementation of the simulation loop that is used in this kernel. As expected, it is very similar to what was described above in Algorithm 4.2. The only significant difference is that the inner while loop's test has been bundled up into a utility method, `EventQueue::isNextEventAt` instead of having a complex conditional.

4.3 Java Implementations

4.3.1 Sequential Implementation

The Java sequential implementation is very similar to the C++ sequential implementation due to both languages being OO and having similar features. The structure of the classes in the implementation appear in Figure 4.3. The modules from the initial high level design in Figure 4.1 have been converted to Java classes. `Main` and `Experiment` classes have been added to isolate the kernel implementation away from the I/O specific details. The details of these classes are similar to `RysimMain` and `Experiment` in the C++ implementation. The `SimulationController` class again has been augmented to support tracking details of the experiment run for later use in the analysis process.

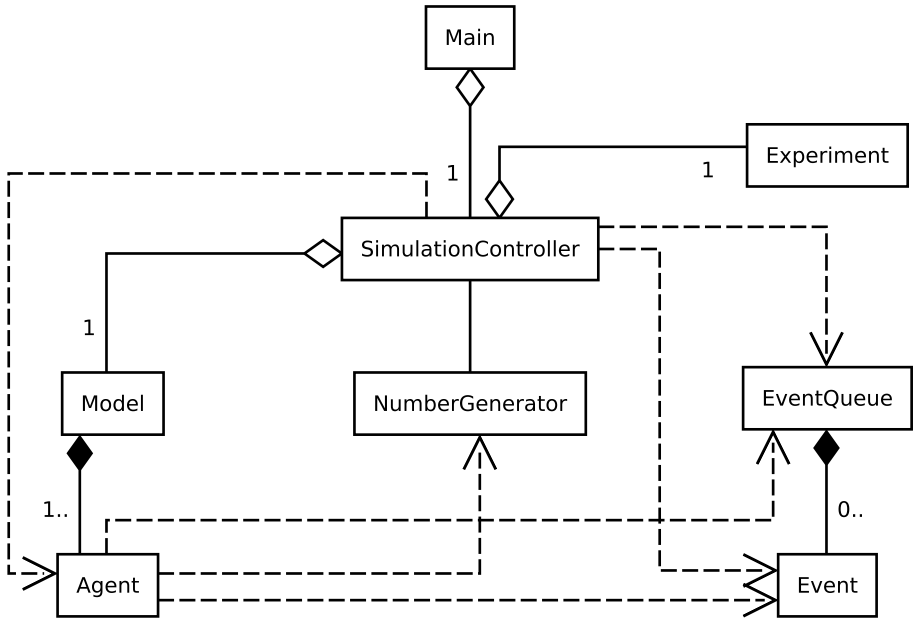

Figure 4.3: Structure of Java sequential implementation classes

The implementation of the simulation loop for this kernel appears in Listing 4.2. It is very similar to what is described in Algorithm 4.2, as expected. The only significant change is the encapsulation of the inner while loop's conditional in a helper function, which was also appears in the C++ implementation.

4.3.2 Threaded Implementation

The structure of the threaded implementation, shown in Figure 4.4, is an incremental evolution of the design that appears in the sequential implementation, Figure 4.3. It contains the previously discussed `Main` and `Experiment` classes as well as using the standard library `ThreadPool` class. `ThreadPool` is used to acquire thread instances in the simulation loop when parallelizing operations.

Listing 4.3 contains the method that implements the simulation loop for the threaded implementation. This implementation follows the two phase pattern, but with substantial semantic differences. For the first phase, instead of sending each `Event` to the model for processing, they are added to a map using the receiving `Agent` as the key and chaining multiple `Event` instances for the same `Agent`. For the second phase a thread is retrieved from the `ThreadPool` for each `Agent` in the map. A thread processes the chain of `Event` instances for the associated `Agent` and generates the appropriate new `Event` instances. There is locking implemented in the `EventQueue` to serialize access. The main thread in `SimulationController` awaits the completion of the threads and then moves onto processing the next timestamp.

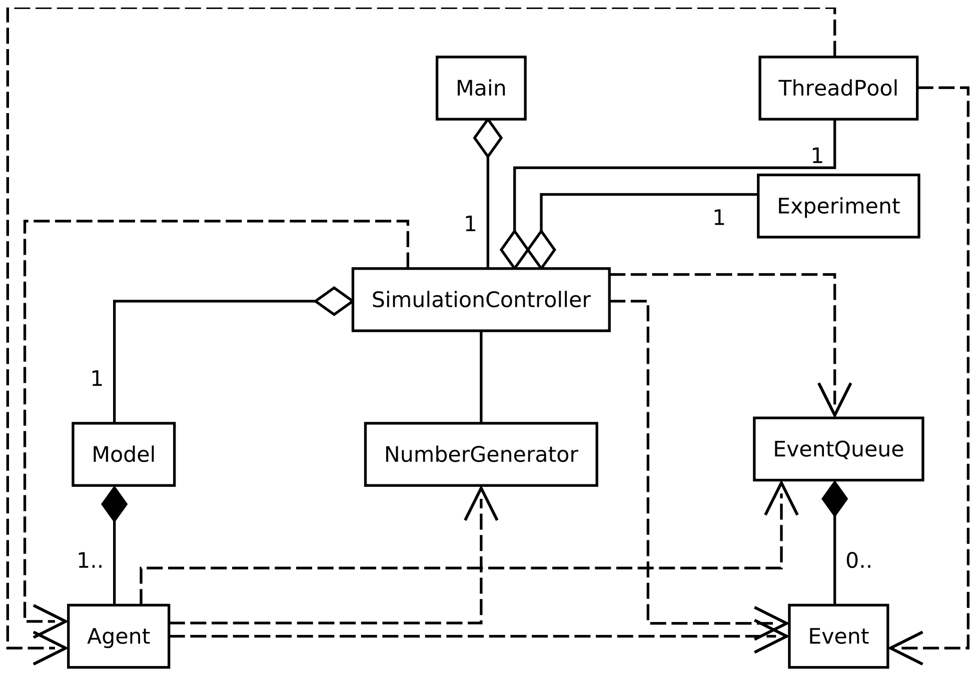

Figure 4.4: Structure of Java threaded implementation classes

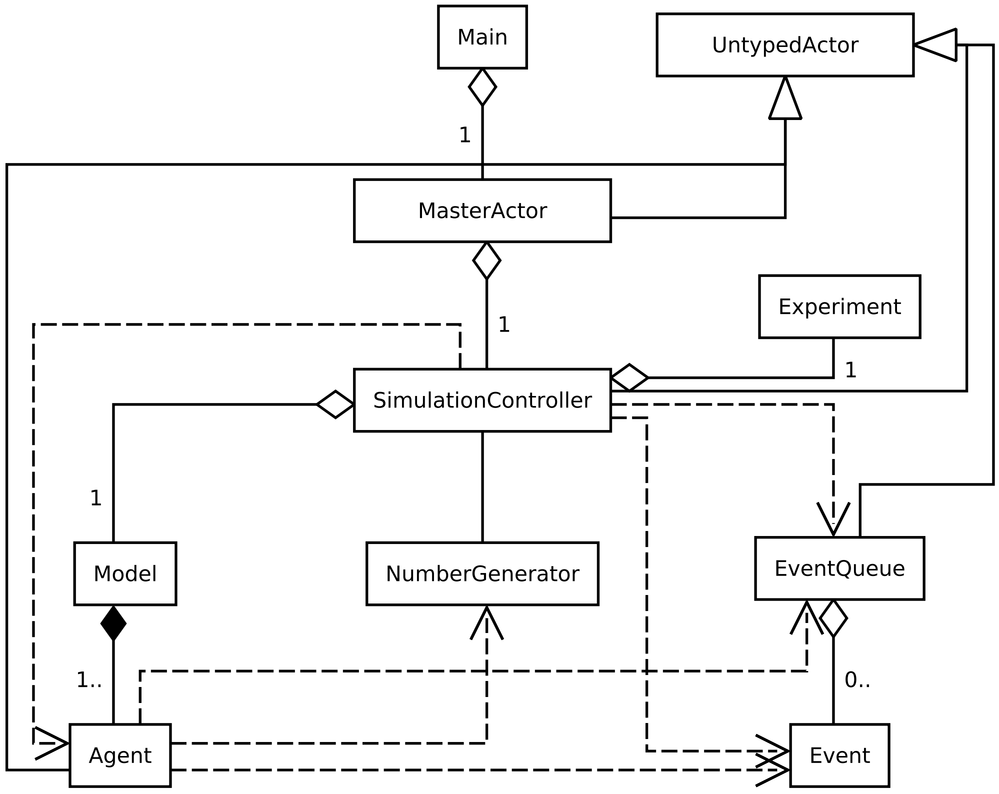

Figure 4.5: Structure of Java Actor implementation classes

```
public boolean run() {
    while (mEventCount < mEventLimit && !EventQueue.getInstance().isEmpty()) {
        Event event = EventQueue.getInstance().pop();
        mModel.processEvent(event);
        mEventCount++;
        mTimestamp = event.getTimestamp();
        while (EventQueue.getInstance().isNextEventAt(mTimestamp)) {
            event = EventQueue.getInstance().pop();
            mModel.processEvent(event);
            mEventCount++;
        }
        mModel.advanceState(mTimestamp);
    }

    return true;
}
```

Listing 4.2: Implementation of simulation loop in Java sequential kernel

4.3.3 Actor Implementation

The Actor based implementation's structure is also based off of the model presented by the sequential implementation, Figure 4.3, and is presented in Figure 4.5. In this implementation `SimulationController`, `Agent`, and `EventQueue` all subclass the Akka framework class `UntypedActor`. `UntypedActor` contains the generic parts of the Actor behaviour in Akka and sub classes only have to implement a callback for message handling. An additional class `MasterActor` is added to the system, which also inherits from `UntypedActor` and operates as the root of the Actor system. `NumberGenerator` doesn't change state during execution of the simulation, since the raw random number generation and seed control are implementated by the VM, so it does not need to be an Actor and thus remains a singleton.

The implementation of the simulation loop, Listing 4.4, is structured in manner more similar to the high level algorithm, Algorithm 4.2, than the threaded implementation, Listing 4.3. The first phase is implemented in a very similar manner, sending each of the popped `Event` instances to the `Model`, which forwards them to the correct `Agent`. For the second phase, an advance state command is issued to the dirty `Agent` instances via the `Model` to generate new `Event` instances. The communication with the `EventQueue` is performed synchronously since processing cannot proceed until a response received. The communication with the `Agent` instances is asynchronous, which is acceptable since the order of the messages sent from Actor A to Actor B is guaranteed to arrive in their sending order, so there doesn't need to be a sync step before the advance state command. After sending the advance state messages, `SimulationController` leaves this method and awaits messages from all of the `Agent` instances indicating that their processing is performed before starting the next timestamp.

```
public boolean run() {
    HashMap<String, List<Event>> timestampHash = new HashMap<>();
    while (mEventCount < mEventLimit && !EventQueue.getInstance().isEmpty()) {
        Event event = EventQueue.getInstance().pop();
        addEventToHash(timestampHash, event);
        mTimestamp = event.getTimestamp();
        while (EventQueue.getInstance().isNextEventAt(mTimestamp)) {
            event = EventQueue.getInstance().pop();
            addEventToHash(timestampHash, event);
        }

        for (Map.Entry<String, List<Event>> entry : timestampHash.entrySet())
            mThreadPool.execute(createProcessTimestampRunnable(entry.getKey(),  \
                mTimestamp, entry.getValue()));

        mLock.lock();
        try {
            while (mOutstandingAgents > 0)
                mAgentsOutstanding.await();
        } catch (Exception e) {
            SimpleLogger.fatal(TAG, ``While awaiting for outstanding agents  \
                received exception, `` + e);
        } finally {
            mLock.unlock();
            timestampHash.clear();
        }
    }
    mThreadPool.shutdown();
    return true;
}
```

Listing 4.3: Implementation of simulation loop in Java threaded kernel

4.4 Erlang Implementations

Since Erlang is a functional language, the data and code elements of the implementations are separated from each other. Thus, it is not appropriate to discuss the structure of the source code in terms of classes, but instead the code is discussed in terms of modules. Most of the concepts that one uses for describing the relationships between classes do not translate well, since most modules just use each other and have a dependency relationship. Through how the implementations have been structured, with an interface module and multiple implementation modules, there are equivalents to composition and inheritance occurring. This becomes significantly more prevalent in code that is using behaviours from the Erlang OTP, since the relationship created in this case is similar to an abstract generic class with concrete specific implementations.

4.4.1 Sequential Implementation

The structure of the sequential implementation appears in Figure 4.6. This structure follows the basic pattern of the initial high level description, Figure 4.1, with significant modifications. First, the parsing and validation code appears in its own modules instead of being part of an experiment or model module.

```
private void run() {
    Boolean booleanResult = null;
    if (mEventCount < mEventLimit
            && !MessagingUtils.askSynchronous(mEventQueue, EventQueue. \
                IsEmptyMessage.create(), booleanResult)) {
        Event event = null;
        event = MessagingUtils.askSynchronous(mEventQueue, EventQueue.PopMessage. \
            create(), event);
        mModel.processEvent(event);
        mEventCount++;
        mTimestamp = event.getTimestamp();
        while (MessagingUtils.askSynchronous(mEventQueue, EventQueue. \
            IsNextEventAtMessage.create(mTimestamp), booleanResult)) {
            event = MessagingUtils.askSynchronous(mEventQueue, EventQueue. \
                PopMessage.create(), event);
            mModel.processEvent(event);
            mEventCount++;
        }
        mDirtyCount = mModel.advanceState(mTimestamp);
    } else {
        mMasterActor.tell(true, getSelf());
    }
}
```

Listing 4.4: Implementation of simulation loop in Java Actor based kernel

The `parsers` module is just an interface for accessing the specific parser, either `parsers_experiment` or `parsers_model`. `validators` contains validation code that checks the sanity of the inputs. This code has been factored out to limit the nesting required and remove duplicated code in the calling functions. There is no separate model module at all in the structure. This is because the model is just a language provided tree structure and the logic for operating on it has been subsumed into the `simulation_controller` module.

The code for implementing the simulation loop appears in Listing 4.5. This code is divided into two different functions, `simulation_loop` and `run_timestamp`, to avoid having one overly complex function that handles all cases. The entry point to this code is the `simulation_loop` function, which tests if another timestamp should be processed. If not, then the result of the simulation is returned to the caller, else `run_timestamp` is called for the next timestamp. `simulation_loop` is structured to make the end of each logic branch either be a return or a call to itself to take advantage of tail recursion optimization and prevent a new stack frame being generated for each timestamp.

`run_timestamp` processes all events for the specified timestamp in the manner of the first phase from the original algorithm. Once all of the events for the current timestamp have been processed, then the state is advanced to generate new events. This code is also structured to take advantage of tail recursion optimization to avoid memory usage proportional with the number of events on the timestamp.

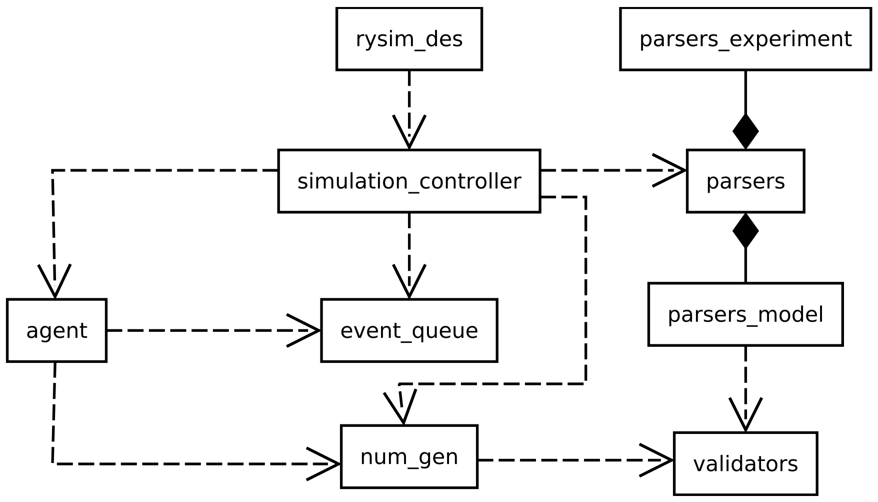

Figure 4.6: Structure of Erlang sequential implementation modules

4.4.2 Actor Implementation

The structure of the Actor implementation in Erlang is presented in Figure 4.7. The structure of this code is substantially more complex than what was used for the sequential implementation, Figure 4.6, because of the heavy use of indirection. All of the modules that now implement the `gen_server` behaviour have a wrapper module that implements the same interface that the original sequential code used. This allows for the interfaces between the different logical modules to remain simple and significant amounts of code to be reused. It does lead to two modules where there was previously one, in a number of cases.

The `simulation_controller`, `agent`, and `event_queue` modules have been refactored to use `gen_server` to implement Actor like behaviours. The module with the `server` suffix is the actual implementation and the non-suffixed version is the interface.

The code listing for the simulation loop in Listing 4.6 demonstrates the advantage of this approach, with the implementation being almost identical to the sequential version. The major change is that a message needs to be sent to `simulation_controller_server` at the end of the simulation to finalize the results. In the sequential implementation, a common pattern for calling another module is to pass in the related piece of state as part of the function call and receive a mutated form of it back. In the Actor version the state does not need to be passed around, since it is encapsulated in the `gen_server` state. Instead, just the related process identifier is passed in as part of the function call to let the wrapper route the call correctly.

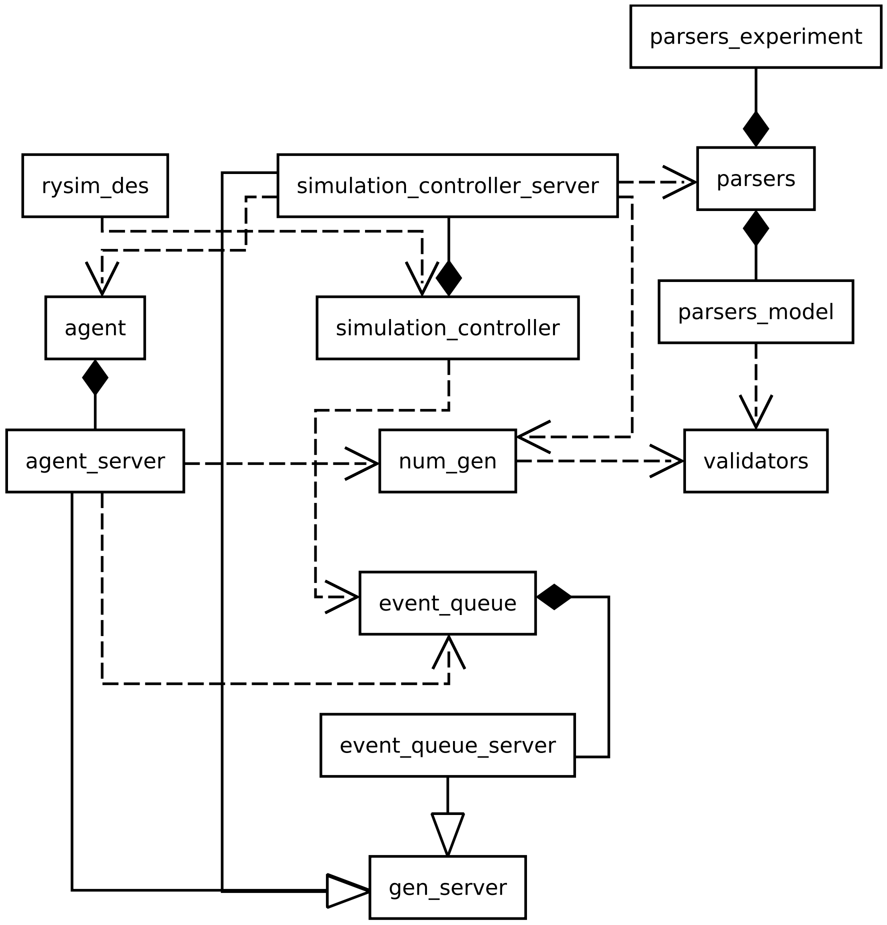

Figure 4.7: Structure of Erlang Actor implementation modules

4.5 Discussion

Validity

There are two major concerns about validity that arise from the above implementations. The first deals with the correctness of each implementation and second is concerned with ability to fairly compare the implementations. Though it is impossible to guarantee perfect implementations, reasonable efforts have been taken to address both of these concerns throughout the implementation process and are discussed below.

The concern about correctness of the implementations can be stated as the question, 'How can we be sure that these implementations are doing what is intended?'. During the development process a small suite of test inputs was used to help verify that the implementations were behaving as expected. This suite included some trivial models to sanity check the system and some longer running models that have known results. These models were run under profiling to make sure that all of the core logic was being exercised by them. In addition to these high level tests some unit tests were implemented for specific implementations to help investigate bugs that were discovered by the high level tests.

The other major concern relating to validity can be stated as, 'Is it fair to compare these implementations?'. This concern is focused on if the implementations are of similar quality, so that comparing their performance and complexity is reasonable. The overall process used for implementing the systems was the same for all the implementations. There was an initial high level design stage that planned out the general structure of the implementation and worked out some of the low level details like the simulation loop. A rough implementation would then be performed based on this design. Once the initial implementation was completed the test suite would be implemented, and used to refine the initial implementation and correct any bugs. Once a feature complete version of the implementation was achieved, performance profiling was performed and any unexpected hot spots addressed.

The above described implementation process was used to guarantee consistency of code quality across the implementations. The same developer, using the same base tools, implemented all of these systems and had comparable levels of experience with all of the languages being used. The profiling that was performed at the end of the development process was performed using standard coverage tools and focused on addressing time spent in specific functions and the number of times they were called.

Qualitative Analysis

This section contains qualitative observations about the different implementations and the ease of working with the various languages. The quantitative analysis of the implementations discussed above appear later in the thesis in Section 5.4. For all of the implementations, a similar development environment was used.

Development was performed using the Intellij family of IDEs on a Linux laptop. This was done due to the preferences of the investigator and to make sure that issues related to poorer language support in the development tools did not effect any of the languages.

The first significant difference between the three languages being used related to the build systems involved. C++ doesn't have a standard tool for performing builds, so customs Makefiles were written. This allowed for specification of the build rules, but does not support higher level concepts like dependency management.

For build management, Erlang has Rebar, which has become the standard tool among developers. Unlike Make, it doesn't require as much detail in specifying the system as long as the source code directories are laid out in a pre-defined manner. It also handles third party dependency management, so additional libraries did not need to be manually installed. Rebar also supports the standard Erlang unit and integration testing frameworks. A Make based wrapper was written for Rebar to make it easier to use from the command line.

For Java, there are a number of very similar build systems available, i.e. Maven, Ant, and Gradle. Gradle was chosen since Akka's documentation has instructions for building using Gradle, and Intellij has plugins for integrating Gradle into the IDE.

From an implementation perspective, the only language and combination that had any issues was the sequential Erlang kernel. This is due to the simulations having a relatively large amount of state that mutates over time and some elements of it, i.e. the event queue, need to be globally available. Erlang being a single assignment functional language does not handle this well, since it means that there needs to be a large state structure that is passed around, unpacked, repacked and returned. Compounding this issue is that Erlang's syntax for data structures, records, is relatively verbose to work with. This led to a lot of boilerplate being written for passing state around and generally making the code much harder to read and reason about. There are language feature like DTS tables for allowing global state, but they include locking/synchronization, so come with a significant performance hit in the single threaded case.

The Actors based Erlang implementation was able to address many of these issues through encapsulation. The servers that implement each of the actors have their own local state that is managed by the OTP framework. Instead of having to pass in the relevant part of the simulation state into each function, just the identifier of the hosting process for that actor is needed. Additionally, Erlang allows for globally named processes, so all of the actors could easily make calls to the event queue without state being passed around.

Both the C++ and Java implementations already had this level of encapsulation and mechanisms for sharing state, so it was easier to visualize and implement the system in them. They are similar languages, so there wasn't huge differences between them from an implementation perspective. C++ was a slightly more difficult language to work with for two reasons. The code was written to the pre-C++11 standards, so convenience features like auto typing were not available. This led to more boilerplate code, i.e. writing `std::set<std::string, std::vector>::iterator` instead of `auto`. Typos in this type of code can lead to compile time issues. The other issue with C++ was due to the declarations and definitions of classes being in separate files. In spite of having significant experience writing C++ code, this was still an issue at times. Either it requires a developer to be very attentive to make sure when they change the implementation that the declaration is updated or run into sometimes cryptic compiler time failures. Both of these issues were minimized in the development process through the use of tooling in the IDE.

Dynamic Connection Graphs

In the design presented in the preceding discussions the models have a static connection graph between the agents. Additionally, due to the transition from the `recovered` state to the `susceptible` state no agent is ever removed from the model. This design was chosen to restrict the scope of the implementation. Allowing for a more dynamic connection graph in the model would allow the system to simulate situations that are more realistic and generate real world usable results.

The current design has the edges of the graph embedded in the agents themselves as the connections lists and acting as directed edges, so it is difficult to rewire the graph efficiently. To remove an agent in the graph currently would require walking the entire set of agents to find any that have a link to the agent to be removed. To make the graph dynamic, the first step would be to extract the graph information from the agents into an adjacency matrix that contains the current graph state. Agents would then have a reference into this structure that they would use to look up who they were connected to at various stages. Depending on the density of connections in the graph, this may actually be some sort of sparse matrix structure.

In this design, removing an agent would be achieved by looking up who the agent sends to and receives from in the matrix and removing the related connections. Agents in the matrix could be allocated a reusable identifier when created and deallocate it on removal. These identifiers would map into the matrix and maintained in a pool, so the matrix would not need to be resized on each allocation/deallocation. Using something like a doubling/halving algorithm for reallocating the matrix when it does need to grow or shrink would minimize the amount of memory shuffling even further. There are other optimizations that could be done for allocating and reallocating the matrix or using another data structure, but that would require knowledge of the specific model being run. Moving to an externalized connection graph design would allow for efficient agent birth and death in the system.

This externalization could also be used for performing dynamic rewiring of the connection graph. This rewiring could be performed after committing the changes on a specific timeslice in the model or could be implemented as another event type that is handled through the event queue. This would allow for finer grained modelling of the evolution of human interactions. The connection graph is modelling social and physical interactions of people during an outbreak, but when it is static there is an assumption that people continue behaving in the exact same way. Thus it does not help with examining aspects like the effectiveness of quarantine policies or effects of presenteeism.

```
simulation_loop(EventCount, Time, ControllerData) when is_integer(EventCount),
    is_integer(Time),
    is_record(ControllerData, controller_data) ->
    Limit = ControllerData#controller_data.event_limit,
    case EventCount >= Limit of
        true ->
            {ok, ControllerData};
        _ ->
            {ok, NextEvent} = next_event(ControllerData),
            case NextEvent == undefined of
                true ->
                    {ok, ControllerData};
                _ ->
                    NewTime = NextEvent#sim_event.timestamp,
                    {ok, NewEventCount, NewControllerData} = run_timestamp( \
                        EventCount, NewTime, ControllerData),
                    Results = NewControllerData#controller_data.results,
                    NewResults = Results#sim_results{event_count = NewEventCount,
                                                     final_time = NewTime},
                    simulation_loop(NewEventCount, NewTime, NewControllerData# \
                        controller_data{results = NewResults})
            end
    end.

run_timestamp(EventCount, Timestamp, ControllerData) when is_integer(EventCount),
                                                           is_integer(Timestamp),
                                                           is_record( \
                                                               ControllerData,  \
                                                               controller_data) ->
    {ok, NextEvent} = next_event(ControllerData),
    case NextEvent == undefined of
        true ->
            {ok, NewControllerData} = advance_state(Timestamp, ControllerData),
            {ok, EventCount, NewControllerData};
        _ ->
            case NextEvent#sim_event.timestamp =/= Timestamp of
                true ->
                    {ok, NewControllerData} = advance_state(Timestamp,  \
                        ControllerData),
                    {ok, EventCount, NewControllerData};
                _ ->
                    {ok, NewControllerData} = process_event(NextEvent,  \
                        ControllerData),
                    run_timestamp(EventCount + 1, Timestamp, NewControllerData)
            end
    end.
```

Listing 4.5: Implementation of simulation loop in Erlang sequential kernel

```
simulation_loop(ControllerData) when is_record(ControllerData, controller_data)  \
  ->
    EventCount = ControllerData#controller_data.results#sim_results.event_count,
    Limit = ControllerData#controller_data.event_limit,
    case EventCount >= Limit of
        true ->
            gen_server:cast(controller, run_experiment_finalize),
            {ok, ControllerData};
        _ ->
            {ok, NextEvent} = next_event(ControllerData),
            case NextEvent == undefined of
                true ->
                    gen_server:cast(controller, run_experiment_finalize),
                    {ok, ControllerData};
                _ ->
                    Timestamp = NextEvent#sim_event.timestamp,
                    {ok, NewEventCount, NewControllerData} = run_timestamp( \
                        EventCount, Timestamp, ControllerData),
                    Results = NewControllerData#controller_data.results,
                    NewResults = Results#sim_results{event_count = NewEventCount,
                                                     final_time = Timestamp},
                    {ok, NewControllerData#controller_data{results = NewResults}}
            end
    end.

run_timestamp(EventCount, Timestamp, ControllerData) when is_integer(EventCount),
                                                          is_integer(Timestamp),
                                                          is_record( \
                                                              ControllerData,  \
                                                              controller_data) ->
    {ok, NextEvent} = next_event(ControllerData),
    case NextEvent == undefined of
        true ->
            {ok, NewControllerData} = advance_state(Timestamp, ControllerData),
            {ok, EventCount, NewControllerData};
        _ ->
            case NextEvent#sim_event.timestamp =/= Timestamp of
                true ->
                    {ok, NewControllerData} = advance_state(Timestamp,  \
                        ControllerData),
                    {ok, EventCount, NewControllerData};
                _ ->
                    {ok, NewControllerData} = process_event(NextEvent,  \
                        ControllerData),
                    run_timestamp(EventCount + 1, Timestamp, NewControllerData)
            end
    end.
```

Listing 4.6: Implementation of simulation loop in Erlang Actor based kernel

Chapter 5

Results

As discussed in Chapter 3, it was unclear from the outset of the experimentation what the optimal grouping of results and independent variables for fitting would be. The first phase of the analysis of the results was performed to determine this. In choosing the groupings there is a significant trade off being made that is controlled by the size of the resulting groups. Using a more general selection key, i.e. just the kernel type, produces larger groups that are less uniform in origin. Increasing the group size makes under-fitting less likely by increasing the sample pool, but the data is more variable due to the lower cohesion. In selecting a precise key that considers the kernel, machine, and model type the variability of the data decreases, but so does the sample pool. Looking at the following results it is clear that the more precise selection keys is the correct choice and produced more useful results.

For many of the fits, the intercepts are a significant portion of the value being reported in entries from the data. This indicates that for the range of the independent variables that were experimented on there was a dominating factor for the dependent variables which was not accounted for in these variables. Given that this was more pronounced for experiments running on Erlang and Java based kernels, it is likely that one of the unaccounted for factors was the cost of initializing and running in a VM. The performance variability of the virtualized environment that was used likely also played a significant factor.

One of the sides effects of this occurs when considering the effectiveness of a fit using R^2, since the values do not significantly vary between different choices of independent variables for the same selection key. Looking at Tables 5.1 and 5.5 this effect can be seen very clearly. The groupings are distinguishable from each other when looking at a selection key, but within a selection key the different independent variables are very similar. The R^2 values do vary slightly between different choices of independent variables for a selection key's value in less significant digits than shown. These small differences were used to create an ordering between the choice of independent variables for a specific selection key, so that one set of independent variables could be selected to do further analysis.

5.1 Run Time

Looking at Table 5.1, fitting the CPU results after grouping them by kernel, machine, and model type created the best fits. In comparison, using only the kernel type produced the worst fits, and using the kernel and machine type fell in between the other options. Within the set of kernel, machine, and model type fits, using the event count and the number of connections as independent variables produced the optimal fits. This implies that for CPU time, the number of connections dominates the number of agents for predictive power.

Selection Keys	Independent Variables	R^2
Kernel and Machine and Type	Events and Connections	0.4125
Kernel and Machine and Type	Events and Agents and Connections	0.4125
Kernel and Machine and Type	Events and Agents	0.4125
Kernel and Machine and Type	Events	0.4125
Kernel and Machine and Type	Connections	0.4125
Kernel and Machine and Type	Agents and Connections	0.4125
Kernel and Machine and Type	Agents	0.4125
Kernel and Machine	Events and Connections	0.1322
Kernel and Machine	Events and Agents and Connections	0.1322
Kernel and Machine	Events and Agents	0.1322
Kernel and Machine	Agents and Connections	0.1322
Kernel and Machine	Events	0.1322
Kernel and Machine	Connections	0.1322
Kernel and Machine	Agents	0.1322
Kernel	Events and Connections	0.0733
Kernel	Events and Agents	0.0733
Kernel	Agents and Connections	0.0733
Kernel	Events	0.0733
Kernel	Connections	0.0733
Kernel	Agents	0.0733
Kernel	Events and Agents and Connections	0.0733

Table 5.1: Comparisons for CPU fits

Tables 5.2 and 5.3 show the results of scoring the algorithms following the method described in Chapter 3 using the above described optimal fits for CPU time. Table 5.2 contains an additional column of total scores for each kernel that is calculated by giving each kernel a number of points depending on how many times they ranked first, second, etc. In this system, first positions receives seven points and last position, seventh, receives one point, so higher total scores are better. Table 5.3 shows the percentages for the occurrences of each for each ranking.

Kernel	1st	2nd	3rd	4th	5th	6th	7th	Total Score
ErlangDESActor	—	5	10	18	10	1	—	184
ErlangDES	—	6	4	5	5	21	3	136
JavaDES	—	33	3	6	2	—	—	243
C++DES	44	—	—	—	—	—	—	308
ErlangDESActorSMP	—	—	3	7	20	14	—	131
JavaDESThread	—	—	24	8	6	6	—	182
JavaDESActor	—	—	—	—	1	2	41	48

Table 5.2: Scores based on Events and Connections vs CPU for Kernel and Machine and Type fits

Kernel	1st	2nd	3rd	4th	5th	6th	7th
ErlangDESActor	—	0.1136	0.2273	0.4091	0.2273	0.0227	—
ErlangDES	—	0.1364	0.0909	0.1136	0.1136	0.4773	0.0682
JavaDES	—	0.7500	0.0682	0.1364	0.0455	—	—
C++DES	1.0000	—	—	—	—	—	—
ErlangDESActorSMP	—	—	0.0682	0.1591	0.4545	0.3182	—
JavaDESThread	—	—	0.5455	0.1818	0.1364	0.1364	—
JavaDESActor	—	—	—	—	0.0227	0.0455	0.9318

Table 5.3: Score percentages based on Events and Connections vs CPU for Kernel and Machine and Type fits

From Tables 5.2 and 5.3, it is clear that the C++ kernel had the best CPU time behaviour. It ranked first for every comparison performed. From these tables, it is also obvious that the Actor based Java implementation had the worst CPU time behaviour, since it is ranked last in over 90% of the comparisons and its total score is nearly a hundred points behind the next closest kernel.

The sequential Java implementation was second overall, with three quarters of the comparisons ranking it there, though it does have a long tail of lower rankings. It is not clear which kernel came in third and fourth, since the threaded Java and non-SMP Actor based Erlang implementations scored very close together. The Erlang kernel had a wider variance in rankings that was centred on fourth with no skew, while the Java kernel was heavily skewed towards third, but with a heavy long tail in the lower rankings. There was a similar conflict for fifth and sixth between the Erlang sequential kernel and the SMP Erlang Actor kernel.

The sequential kernel had a wide range of values with a lower centre, while the Actors kernel was centred slightly higher, but without high ranking outliers. Table 5.4 gives a summary of the ordering discussed.

Rank	Kernel
1st	C++
2nd	Java Sequential
3rd and 4th	Java Thread and Erlang Actor Non-SMP
5th and 6th	Erlang Sequential and Erlang Actor SMP
7th	Java Actor

Table 5.4: Relative rankings of kernels based on CPU

This ordering makes it clear that adding in concurrent processing did not help with the execution speed of the Java implementations, since both the threaded and actor based versions scored substantially worse than the sequential implementation in CPU time. For Erlang, the story is less clear, since using an Actor paradigm without SMP support ran faster than the sequential code, but turning on SMP support made it run slower. This suggests that the bytecode generated using the Actor paradigm was more efficient than the sequential version, but the overhead of SMP support defeats this speed up.

Comparing the languages it is clear that C++ was superior for execution efficiency. With regards to Java and Erlang, it is not clear if either language was superior, since Java had kernels that score both better and worse than all of the Erlang kernels.

5.2 Memory Usage

Looking at Table 5.5, fitting the memory results by grouping by kernel, machine, and model type created the best fits. In comparison, using only the kernel type produced the worst fits and using the kernel and machine type was middling. These fits in general were of worse quality than what was seen for CPU times and more tightly clustered in their scores. Within the kernel, machine, and model type fits, groupings using the event, agent, and connection counts as independent variables produced the best fits. This implies that for maximum memory, neither the number of agents or connections was dominant.

Tables 5.6 and 5.7 are the maximum memory scoring tables for the optimal grouping and independent variables. These tables have the same semantics as Tables 5.2 and 5.3, but for maximum memory fits.

From Tables 5.6 and 5.7, there is a clear pattern to the rankings of the kernels, which conveniently correlates with the alphabetic sorting of them. The C++ kernel used the least memory, since it again scored first in all

Selection Keys	Independent Variables	R^2
Kernel and Machine and Type	Events and Agents and Connections	0.1684
Kernel and Machine and Type	Events and Connections	0.1684
Kernel and Machine and Type	Events and Agents	0.1684
Kernel and Machine and Type	Agents and Connections	0.1684
Kernel and Machine and Type	Events	0.1684
Kernel and Machine and Type	Connections	0.1684
Kernel and Machine and Type	Agents	0.1684
Kernel and Machine	Events and Agents and Connections	0.0744
Kernel and Machine	Events	0.0744
Kernel and Machine	Connections	0.0744
Kernel and Machine	Agents	0.0744
Kernel and Machine	Events and Connections	0.0744
Kernel and Machine	Events and Agents	0.0744
Kernel and Machine	Agents and Connections	0.0744
Kernel	Events	0.0194
Kernel	Connections	0.0194
Kernel	Agents	0.0194
Kernel	Events and Connections	0.0194
Kernel	Events and Agents	0.0194
Kernel	Agents and Connections	0.0194
Kernel	Events and Agents and Connections	0.0194

Table 5.5: Comparisons for Memory fits

of the comparisons. Second and third are a toss up between the sequential and the non-SMP Actor Erlang kernels. The Actor kernel distribution was skewed slightly higher, but the two kernels are very similar in the rankings. The SMP Actor Erlang implementation was clearly fourth, with 75% of the comparisons ranking it there. The remaining three spots are very clear, with the sequential, Actor, and threaded Java kernels being fifth, sixth, and seventh. This ordering is summarized in Table 5.8.

For the Java based kernels, it is clear that using a concurrency based paradigm comes at a significant cost, with both the thread and Actor based kernels have higher memory usage than the sequential kernel. For the Erlang kernels there is a similar pattern to what was seen for CPU time, where the paradigm choice does not appear to matter, but turning on SMP support does come at a cost. There is a clear stratification between the languages in the results, which is probably explained by the overheads of the languages being the dominant factor in determining the amount of memory used. The languages for memory usage are ordered

Kernel	1st	2nd	3rd	4th	5th	6th	7th	Total Score
ErlangDESActor	—	22	20	2	—	—	—	240
ErlangDES	—	20	17	7	—	—	—	233
JavaDES	—	—	—	—	43	1	—	131
C++DES	44	—	—	—	—	—	—	308
ErlangDESActorSMP	—	2	7	35	—	—	—	187
JavaDESThread	—	—	—	—	—	9	35	53
JavaDESActor	—	—	—	—	1	34	9	80

Table 5.6: Scores based on Events and Agents and Connections vs Max Memory for Kernel and Machine and Type fits

Kernel	1st	2nd	3rd	4th	5th	6th	7th
ErlangDESActor	—	0.5000	0.4545	0.0455	—	—	—
ErlangDES	—	0.4545	0.3864	0.1591	—	—	—
JavaDES	—	—	—	—	0.9773	0.0227	—
C++DES	1.0000	—	—	—	—	—	—
ErlangDESActorSMP	—	0.0455	0.1591	0.7955	—	—	—
JavaDESThread	—	—	—	—	—	0.2045	0.7955
JavaDESActor	—	—	—	—	0.0227	0.7727	0.2045

Table 5.7: Score percentages based on Events and Agents and Connections vs Max Memory for Kernel and Machine and Type fits

C++, Erlang, then Java.

5.3 Machine Comparison

The results up to this point have been comparing kernels across all of the machines. They have not been addressing the changes in the dependent variables that are seen in relation to the machine. Each of the machines used in the experimentation has a different number of cores; 1, 2, 4, and 8 cores. Figures 5.1 and 5.2 contain information about how the kernels performed when the number of cores was varied for the optimal fits described above. Appendix A contains per kernel detailed plots for additional reference.

Figure 5.1 contains a multi-line plot with each kernel's CPU time response to varying the number of cores. For clarification, the individual plots with boxes and whiskers appear in Figures A.1 through A.7. The details of how the data for these plots was generated is discussed in Chapter 3. In these plots, lower scores are better, so for six of the seven kernels the optimal number of cores was two. These kernels saw an initial drop in run time from one core to two cores and a gradual increase in run time as more than two cores were used. The

Rank	Kernel
1st	C++
2nd and 3rd	Erlang Sequential and Erlang Actor Non-SMP
4th	Erlang Actor SMP
5th	Java Sequential
6th	Java Actor
7th	Java Thread

Table 5.8: Relative rankings of kernels based on maximum memory

outlier was the Erlang sequential kernel, which actually saw a run time increase at two cores, then a drop at 4 and a gradual increase up to 8 cores.

The kernels mostly stayed in their own tracks over the range of the experimentation, though the slopes suggest that there would be some crossing over at higher numbers of cores. The only kernels that traded relative positions were the SMP and non-SMP Actor based Erlang kernels. The SMP version initially started with a slight edge, but this was lost at two cores, with the gap becoming progressively larger. The stratification seen in these results is what one would expect to see from the earlier results, with the C++ kernel outperforming all of the other kernels, the Java Actor kernel being the worst performing and there being some muddying between the other kernels.

Figure 5.2 contains a multi-line plot comparing the kernels, memory usage vs the number of cores using the optimal fits discussed earlier in this chapter. For clarification, the individual plots with boxes and whiskers appear in Figures A.8 through A.14. As with the previously discussed figure, the semantics of this plot are discussed in Chapter 3. For these plots, lower values are considered better. From this plot, there are two distinct groups. The first is composed of the C++ and Erlang kernels, which have a substantially lower memory footprint and have relatively constant usage for the range of cores used. The Erlang SMP Actor kernel starts in this group, but appears to have a very slight upwards slope. The other group is the three Java based kernels, which have a close to linear increase with a significant slope. The kernels are ordered how the previous results would suggest they would be.

5.4 Code Complexity

Tables 5.9 and 5.10 contain the tabulated complexity data derived from SonarQube. Table 5.9 contains the metrics calculated in terms of number of statements, while 5.10 has them in terms of number of lines of code. Both metrics for code length were included since the different languages have significantly different styles

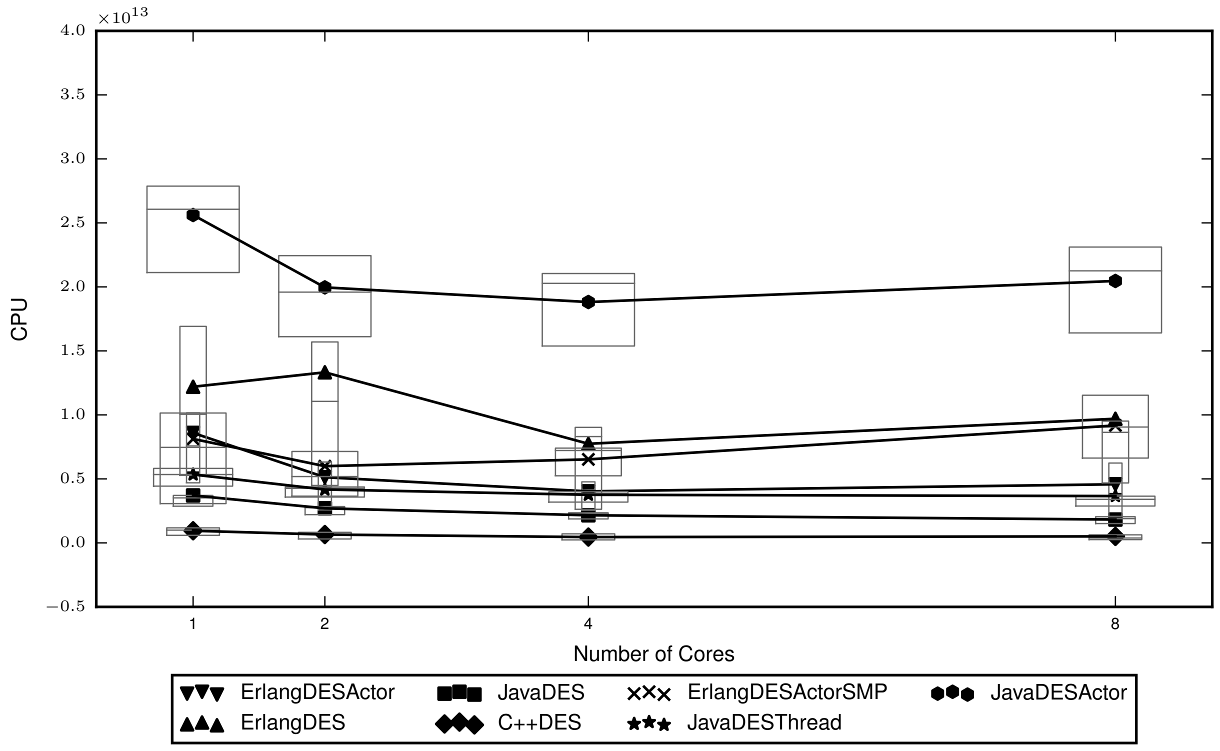

Figure 5.1: Multi-line Box and Whisker Plot of Machine Comparison for Events and Connections vs CPU

when it comes to vertical spacing. This means that the average lines of code per statement isn't consistent across the different languages, so neither metric encapsulates the information provided by the other. When talking about the length of programs, both the number of statements and physical size of the functional code are of interest.

In terms of numbers of statements, the C++ implementation was the longest implementation, while the sequential Java was the shortest. The threaded Java implementation was also quite short, being below 500 statements, while the two Erlang implementations and the Java Actor implementation were grouped in the middle. For both the Java and Erlang implementations, the number of statements increased with the amount of concurrency in the implementation.

The number of statements per function in the implementations is very stable across all of the languages and implementations paradigms, being slightly more than 5 statements per function. The complexity per statement was highest in the Erlang implementations and lowest in the C++ implementation. The Java implementations fell about midway between the other languages in terms of complexity per statement.

Looking at the lengths in terms of lines of code, the Erlang Actor based implementation is the longest and the Java sequential is the shortest. There is significant spread between most of the line counts, with

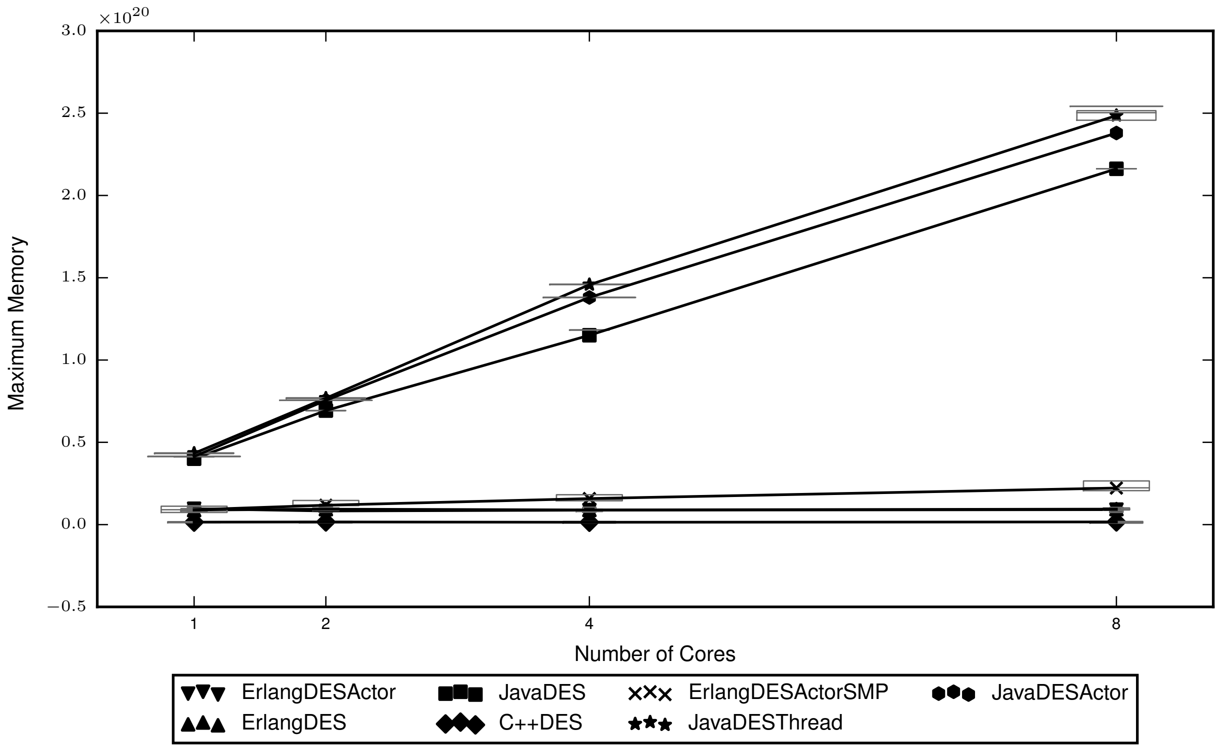

Figure 5.2: Multi-line Box and Whisker Plot of Machine Comparison for Events and Agents and Connections vs Maximum Memory

the remaining implementations being ordered in ascending order as follows: Java thread, C++, Java Actor, then Erlang sequential. The increase in length in response to adding more concurrency is also seen for this metric. The position of Erlang as the longest implementation by this metric isn't surprising when looking at the source code, since Erlang style favours short lines with large amounts of vertical spacing.

The Erlang implementations had the highest number of lines of code per functions, while C++ had the fewest. The Java implementations were close to the C++ implementations. Looking at the Erlang code, this positioning is not unexpected ,since it is common in Erlang to write branching logic in terms of functions using multiple, which require restating the function header multiple times. The amount of complexity per line of code is very consistent across all of the languages, being very close to 0.25.

The Erlang implementations have the highest total complexity, with some of the Java implementations being less complex than the C++ implementation, but the Java Actor based implementation being more complex. The complexity per function is similar in ranking, but with the distinction between C++ and Java being clearer. The C++ implementation has less complexity per function than the Java implementations.

Kernel	Statements	Functions	Complexity	Comp/Func	Stmnt/Func	Comp/Stmnt
C++	619	111	246	2.2	5.6	0.40
Erlang Sequential	520	96	299	3.1	5.3	0.59
Erlang Actor	555	104	333	3.2	5.3	0.60
Java Sequential	455	86	232	2.7	5.3	0.51
Java Actor	561	113	278	2.5	5.0	0.50
Java Thread	483	89	240	2.7	5.4	0.50

Table 5.9: Statement Based Complexity Metrics

Kernel	LoC	Functions	Complexity	Comp/Func	LoC/Func	Comp/LoC
C++	1060	111	246	2.2	9.5	0.23
Erlang Sequential	1190	96	299	3.1	12.4	0.25
Erlang Actor	1291	104	333	3.2	12.4	0.26
Java Sequential	877	86	232	2.7	10.2	0.26
Java Actor	1133	113	278	2.5	10.0	0.25
Java Thread	933	89	240	2.7	10.5	0.26

Table 5.10: LoC Based Complexity Metrics

Chapter 6
Conclusions

The goal of this thesis was to determine if it is viable to implement general purpose programs following the Actors paradigm. The specific area of interest was their suitability for implementing sequential systems and comparing their performance to traditional paradigms. Sequential performance was focused on since the existing research has thoroughly investigated concurrent and distributed performance. To understand the differences between the Actors paradigm and other paradigms, multiple implementations of a discrete event simulation were written in Erlang, C++, and Java. These implementations were compared on a variety of quantitative metrics and qualitative observations.

The initial claims from the hypothesis relate to the run time behaviour of the various implementations. It was expected that sequential kernels would be superior to the concurrent kernels in terms of run time behaviour. The overall ranking of the kernels with respect to running time can be found in Table 5.4. For Java, the claim was validated by the sequential kernel outperforming the other kernels. For Erlang, this claim was only validated when SMP support in the VM was enabled for the Actor kernel. When SMP support was not enabled, the Actor based Erlang kernel was actually faster than the sequential one. The expected stratification of run time performance between the languages only partially occurred. C++, as expected, has the best run time characteristics, but there is not a clear distinction between Java and Erlang, which was counter to the expectations.

It was expected that the behaviour of the kernels in terms of execution time, when the number of cores was varied, would fall into two distinct groups. The first group is the sequential kernels, which should not be affected by using more cores. The other group was the concurrent kernels which should experience sub-linear improvement correlated with the number of the cores. The data related to validating these claims is presented in Figure 5.1 and Appendix A. The expected behaviours did not happen for most of the kernels. The exception was the C++ kernel, which had a flat trend line, indicating no effect from the additional cores.

The trend line for the Java sequential kernel is almost flat, but there is a slight curve downwards indicating improvement when using higher numbers of cores. Since the kernel is sequential, it is something other than application code that is receiving a benefit from using multiple cores, which could be the VM itself or the library for JSON parsing. The Erlang sequential kernel trend line is very out of line with the expectations, having a decrease in performance initially when introducing another core, and gaining performance with the addition of more cores. The wide variance would suggest that something more than a slight parallelism in a library is occurring, so it is likely that some of the critical code paths in the VM change depending on if there are multiple cores available.

All of the concurrent kernels have a similar shape to their trend lines, which is in agreement with expectations. They all have an initial performance boost from adding one core, but as more cores are added, their performance actually degraded. This indicates that there was a small parallelizable section in the implementation that using the second core assisted in. This parallelizable section is not that large, so adding more cores receives diminishing returns, which are in turn eliminated by the cost of accounting for the additional cores.

In the hypothesis it was asserted that the data would show that selecting a sequential kernel of the same language and running multiple instances of it, one per core, would be faster than running a concurrent kernel across those cores. In falsifiable terms, it was asserted that the run time of a sequential kernel would never be n times that of a concurrent kernel running on n cores. The graph presented in Fig 5.2 shows this assertion is true for Java, since at all times the sequential kernel has a lower run time value than the concurrent kernels. For Erlang, this assertion holds true for most cases, with one exception. For the case of the non-SMP Actor based kernel running on two cores, this kernel is in the range of having a low enough value. This is an odd case, because this kernel is implementing a concurrent paradigm, but executing in a sequential manner since SMP support has been turned off. The optimal selection for Erlang from a run time perspective is multiple instances of the non-SMP Actor kernel running on a single core each.

For memory usage, it was posited that kernels would be ranked sequential, threaded, then Actor in terms of memory efficiency. Table 5.8 shows that this claim is partially correct. The sequential kernels used the least memory, but the threaded kernel was actually less memory efficient than the Actor kernel in Java. It is also notable that the sequential and non-SMP Actor kernels for Erlang were tied for memory usage. For language grouping, it was asserted that they would be ranked C++, Erlang, then Java in terms of memory usage, which was validated by the data.

For the scaling of memory usage with respect to the number of cores, there were two main hypotheses advances. The first was that the sequential kernels would have flat trends, and the second was that the concurrent kernels would have memory usage that scaled proportionally with the number of cores available. These claims partition the kernels by the paradigm that they embodied. The data in Figure 5.2 and Appendix A indicates the appropriate partitioning would have been along both language and paradigm of the kernel. The C++ and Erlang sequential kernels have the flat memory usage that was expected, but the Java one increases proportionally with the number of cores. For Erlang, the non-SMP Actor kernel also has a flat memory usage behaviour, which is expected given its sequential nature. The SMP version of the Erlang Actor kernel has a slight increase in memory usage with the number of cores. For Java, there is a separation between the different paradigms, but all of them increased proportionally with the number of cores.

For the code complexity metrics, it was expected that most of the variation would be seen across languages. Erlang was expected to have the highest complexity per statement and lines of code, but to have a lower total complexity. C++ was expected to be the second densest language and to have the highest total complexity. Java was expected to have the lowest density and to be between the others for total complexity. It was expected that within a specific language the sequential implementations would be less complex than the concurrent ones.

The results relevant for evaluating these assertions appear in Tables 5.9 and 5.10. For comparing languages, the claim that Erlang was going to be the densest was validated in terms of statements, but not lines of code. Erlang was also shown to have the highest total complexity. The claims for C++ were invalidated, it has the lowest density in terms of both statements and lines of code, and one of the lowest total complexities, which was unexpected. Java was in the middle for density, and had a very wide range for total complexity. The kernels within a single language become more complex as concurrency was added, though in different proportions depending on the specific language.

From the results presented in this research, there was a significant differences in execution performance and a minor differences in implementation difficulty, the choice for implementing a simulation system would be C++. It produced by far the most efficient kernel and the difficulties of working with the language were minor and manageable with proper tooling. When picking a language for a software project there are many considerations beyond the two discussed above, i.e., integration with existing systems and team tool preference.

An unexpected result of this research was that implementing the kernel in Erlang using an Actor paradigm with SMP support disabled performed better than the sequential implementation. When considering the background of Erlang, this is not as surprising, since the Actor like behaviour in the language comes from the OTP standard library which is designed to be a robust and production ready piece of software. This result makes it clear that it is possible to implement an Actor framework that can be used for sequential systems and not have a significant performance loss when scaled down to running on a single core.

6.1 Discussion

The possible directions for work to follow this research fall into two major camps. The first would be to address identified sources of error in the current experiment to reinforce the validity of the results. The other direction would be to extend the applicability of the results and explore the application of the conclusions.

6.1.1 Potential Sources of Error

There were a variety of error sources highlighted in this research along with the efforts made to minimize their impact. Given the variability of the data, there is significant room for improvement. The change that would likely cause the largest improvement in the data would be to run the experiments on dedicated hardware instead of virtualized environments.

During the development process attempts were made to make sure that the code being produced was of similar quality between languages and that any obvious bottlenecks were removed. This process was not of a sufficient standard for development of a production quality system. A more careful review of the code could lead to stylistic and performances improvements. Using updated versions of C++ and Java would likely produce more concise code since both languages have introduced features that reduce the amount of boilerplate needed. The code as it exists also has limited tests for its correctness, so revising the code should involve implementing a complete test suite.

The measurement of memory usage using the maximum value is of limited utility and — due to the sample methodology employed — rather noisy. The experimental framework could be modified to add tracking the average memory usage of the kernel. In addition, one could add to the kernels a signalling mechanism to indicate when different stages had been reached, i.e. model parse, simulation started, etc., and this could be used to partition the metrics. These changes would significantly increase the fidelity and usefulness of the data generated.

The analysis techniques that were applied to the data are linear in nature, due to an assumption that the dependent variables of interest respond in a linear fashion to changes in the independent variables. Given the relatively low scores for R^2 even in the optimal fits, there is reason to suspect that the response function might not be linear. This suggests that future research in this area should investigate using a wider toolbox of traditional analysis techniques, including polynomial and non-polynomial fitting, to make sure the relationship between the variables is properly categorized.

The raw data from the experiment has the appearance of being very variable and had filtering applied to it to make it usable for the analysis. The assumption made here is that all of the noise is due to variability in the execution environment and other factors. There is another possibly that was not addressed in the research for this noisiness. It is possible that the the system being investigated behaves chaotically for some, if not, all of the experiments. This would make the results generated very sensitive to the initial state of the model or the parameters for transitioning state, so require a different set of tools than those that were used for analysis. It is recommended that reducing the environmental variability be performed before attempting to locate complex behaviours in the system to avoid confounding sources of variability.

6.1.2 Extension of Results

The results presented in this thesis were derived from implementing a specific model in the simulation system, which limits the application of the conclusions to more general simulation systems. The conceptually simplest way to resolve this would be to add more model types to the test set. This could be achieved by adding additional model input schema and implementing support for them in the kernels. This would not be optimal for extending the system, since it would lead to a set of special case code for each model type with limited common code. It would make more sense to generalize as much of the existing code and write a general specification language that the kernels could parse. This would separate the semantics of the model out of the kernel and be more representative of how most simulation frameworks operate.

The kernels implemented in this research were all based on a simple sequential simulation algorithm. There exists a large body of research, in the area of distributed simulation, related to simulation algorithms that take advantage of many cores and computers. It would be valuable to implement the common algorithms from this area. This would allow comparison of the different implementations and an understanding of how distributed algorithms scale using the results from this research as a baseline.

The fact that the Erlang Actor implementation outperformed the sequential implementation suggests an avenue of research for Actor frameworks. It appears reasonable that an Actor framework could be designed to have two modes of operation, depending on whether there is one or multiple cores available. This design could be implemented as a new framework or an existing framework could be modified to use it. From a pure performance standpoint it would make the most sense to implement this in C++ to improve the developer experience. For extending this research it would be most useful to have an implementation in both C++ and Java to allow for comparison between all three of the languages.

References

[1] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[2] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] Joe Armstrong. Erlanga survey of the language and its industrial applications. In *Proc. INAP*, volume 96, 1996.

[4] Joe Armstrong. The development of erlang. In *ACM SIGPLAN Notices*, volume 32, pages 196–203. ACM, 1997.

[5] Ottar Bjørnstad. Seir models. 2005.

[6] P. Carbonnelle. Pypl popularity of programming language index. https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language, 2014.

[7] C++ Standards Committee et al. Iso/iec 14882: 2011, standard for programming language c++. Technical report, Tech. rep., ISO/IEC, 2011. http://www. open-std. org/jtc1/sc22/wg21.

[8] C++ Standards Committee et al. Iso/iec 14882: 2003 programming languagesc++. Technical report, Technical report, ISO, 2003.

[9] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. Simula 67 common base language. 1967.

[10] LLC DedaSys. Programming language popularity. http://www.langpop.com, 2013.

[11] Boost Developers. Boost c++ libraries. http://www.boost.org/, 2014.

[12] Erlang Developers. Erlang programming language. http://www.erlang.org/, 2014.

[13] Edsger W Dijkstra. The structure of the the multiprogramming system. In *Classic operating systems*, pages 223–236. Springer, 2001.

[14] P Erdős and A Rényi. On random graphs. i. *Publicationes mathematicae*, 6:290–297, 1959.

[15] A. Ermedahl et al. Discrete event simulation in erlang. *UU/CSD*, 1994.

[16] Charles T. Gibson. Time-sharing in the ibm system/360: Model 67. In *Proceedings of the April 26-28, 1966, Spring Joint Computer Conference*, AFIPS '66 (Spring), pages 61–78, New York, NY, USA, 1966. ACM.

[17] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, pages 1141–1144, 1959.

[18] James Gosling. *The Java language specification.* Addison-Wesley Professional, 2000.

[19] James Gosling and Henry McGilton. *The Java language environment*, volume 2550. Sun Microsystems Computer Company, 1995.

[20] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[21] Typesafe Inc. Akka. http://akka.io/, 2014.

[22] Vojin Jovanovic and Philipp Haller. The scala actors migration guide. http://docs.scala-lang.org/overviews/core/actors-migration-guide.html, 2014.

[23] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[24] A Koenig. Standard–the c++ language. report iso/iec 14882: 1998. *Information Technology Council (NCTIS)*, 1998.

[25] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[26] Simon Marlow and Philip Wadler. A practical subtyping system for erlang. *ACM SIGPLAN Notices*, 32(8):136–149, 1997.

[27] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[28] G. Montmollin. The transparent language popularity index. http://lang-index.sourceforge.net/, 2013.

[29] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[30] S. O'Grady. The redmonk programming language rankings: June 2014. http://redmonk.com/sogrady/2014/06/13/language-rankings-6-14/, 2014.

[31] SonarSource S.A. Sonarqube. http://www.sonarqube.org/, 2014.

[32] R.G. Sargent. Verification and validation of simulation models. In *Proceedings of the 37th conference on Winter simulation*, pages 130–143. Winter Simulation Conference, 2005.

[33] Richard Saucier. Computer generation of statistical distributions. Technical report, DTIC Document, 2000.

[34] TIOBE Software. Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2014.

[35] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1. ACM, 2007.

[36] Bjarne Stroustrup et al. *The C++ programming language*. Pearson Education India, 1995.

[37] GCC Team. Gcc, the gnu compiler collection. https://gcc.gnu.org/, 2014.

[38] Arthur H Watson, Thomas J McCabe, and Dolores R Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST special Publication*, 500(235):1–114, 1996.

[39] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-worldnetworks. *nature*, 393(6684):440–442, 1998.

Appendix A

Machine Comparison Graphs

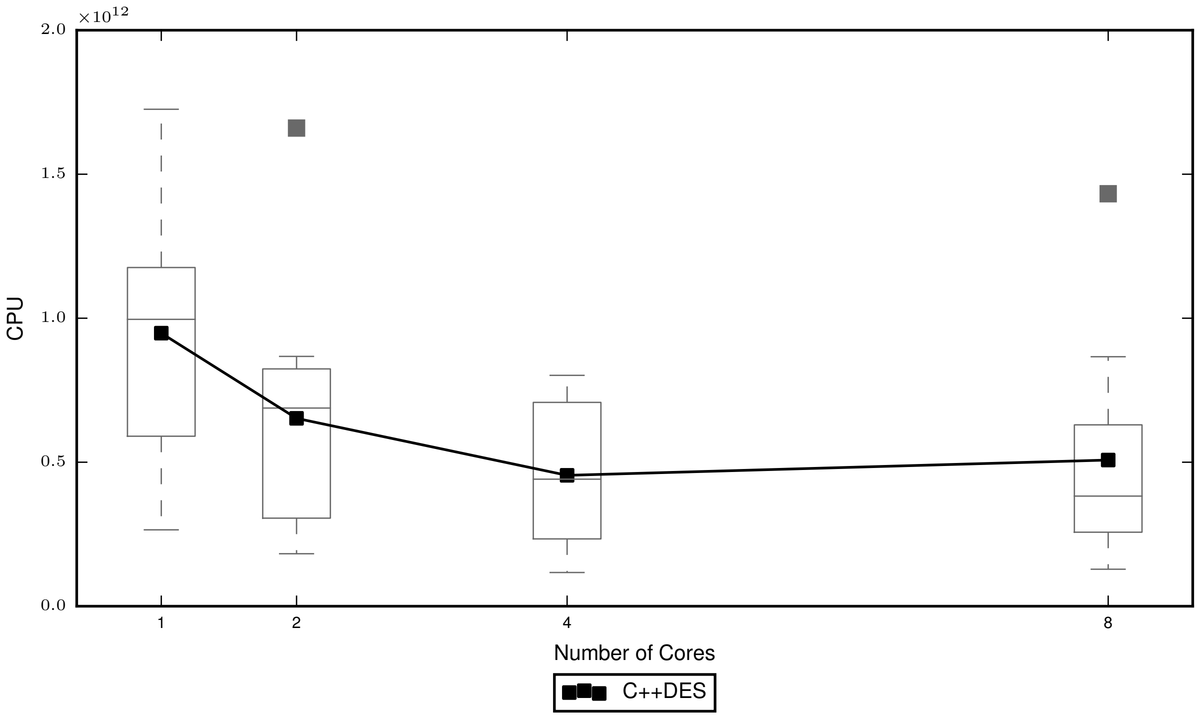

Figure A.1: Box and Whisker Plot of Machine Comparison of C++DES for Events and Connections vs CPU

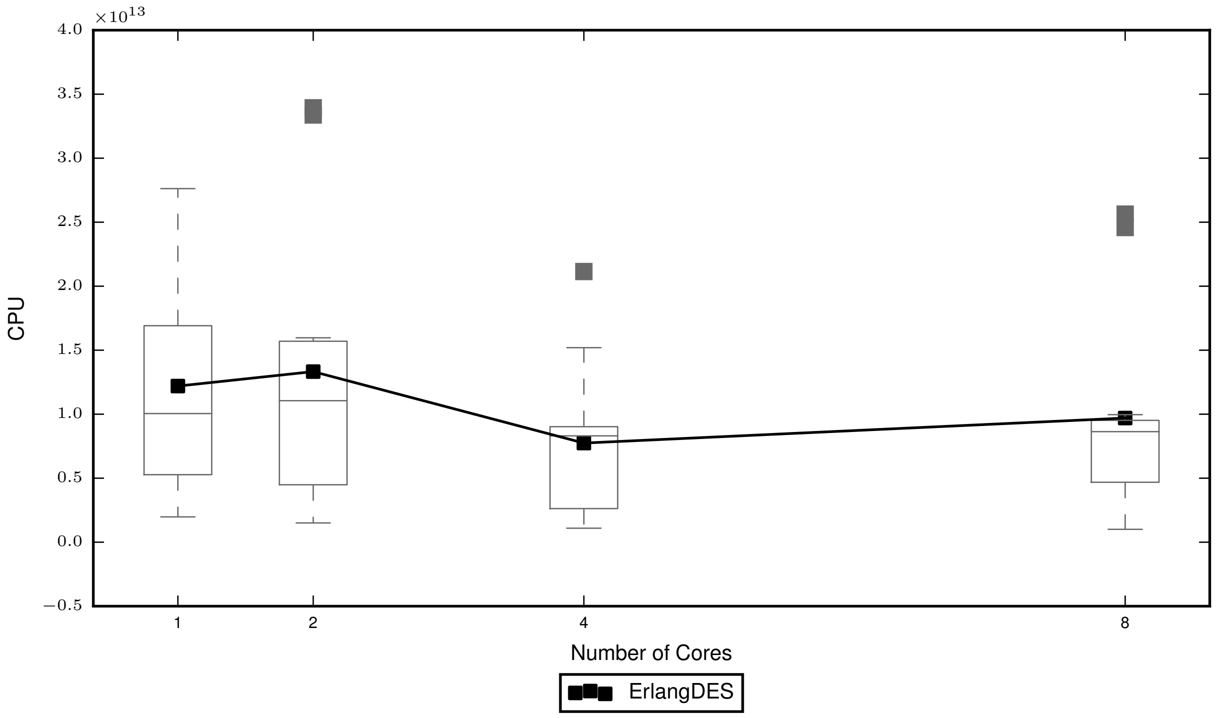

Figure A.2: Box and Whisker Plot of Machine Comparison of ErlangDES for Events and Connections vs CPU

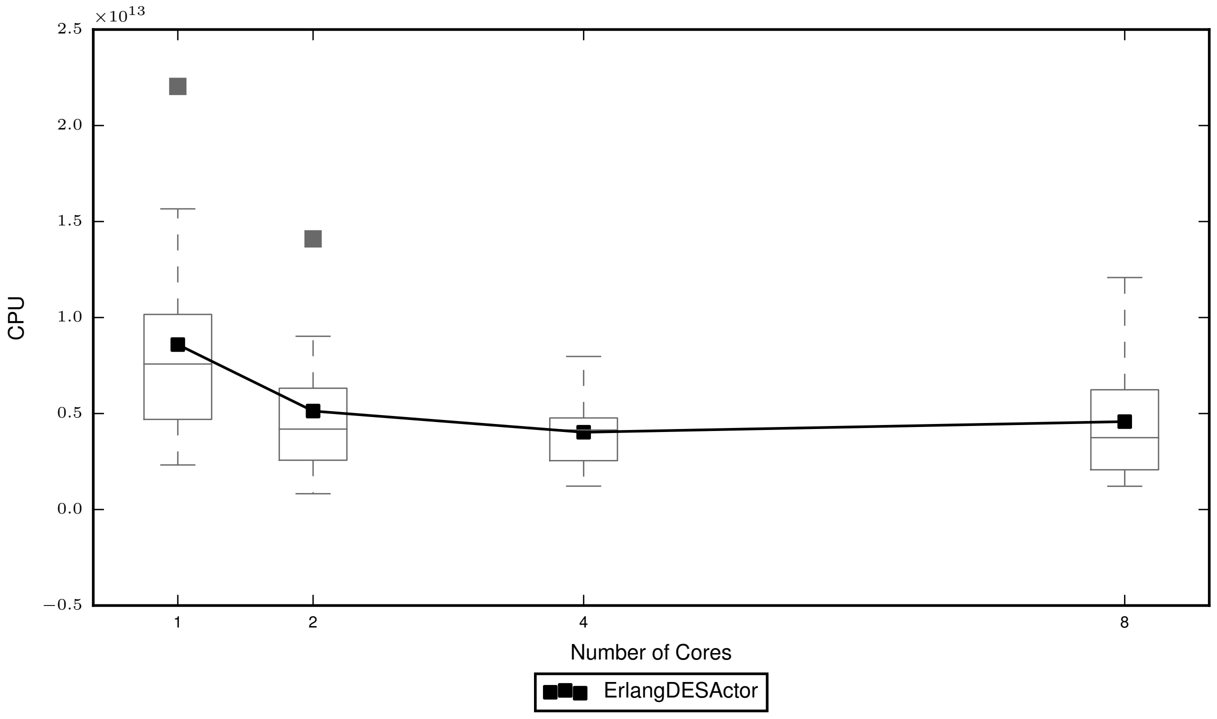

Figure A.3: Box and Whisker Plot of Machine Comparison of ErlangDESActor for Events and Connections vs CPU

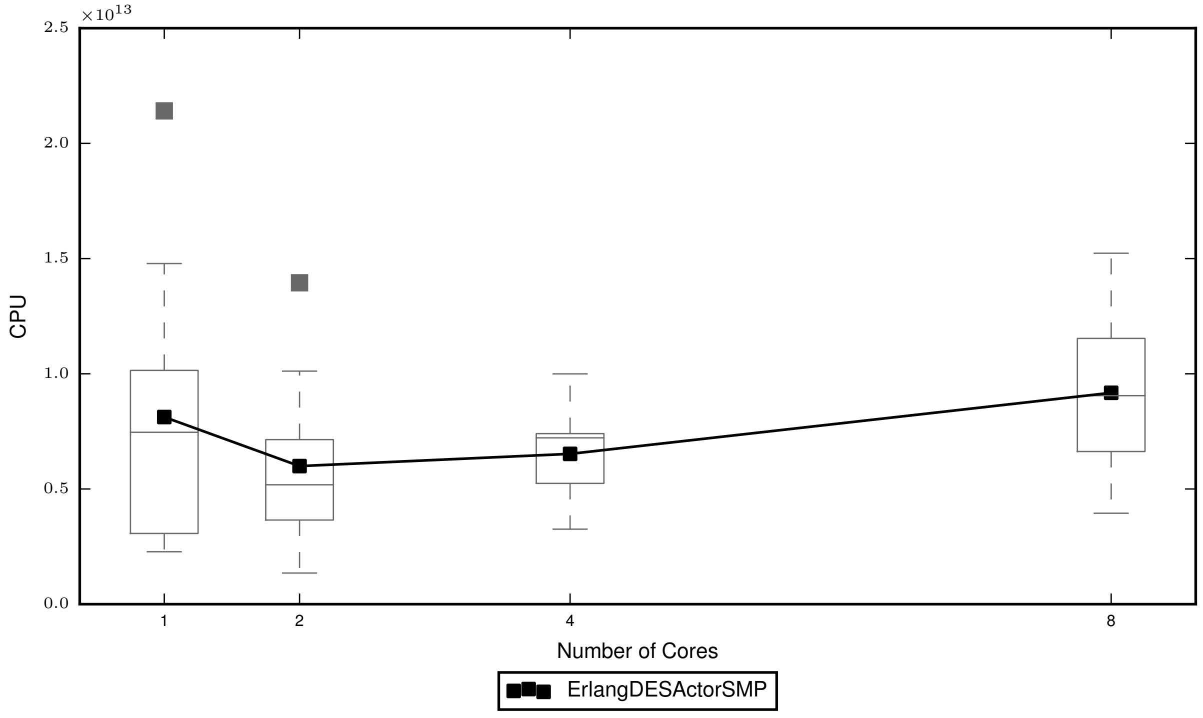

Figure A.4: Box and Whisker Plot of Machine Comparison of ErlangDESActorSMP for Events and Connections vs CPU

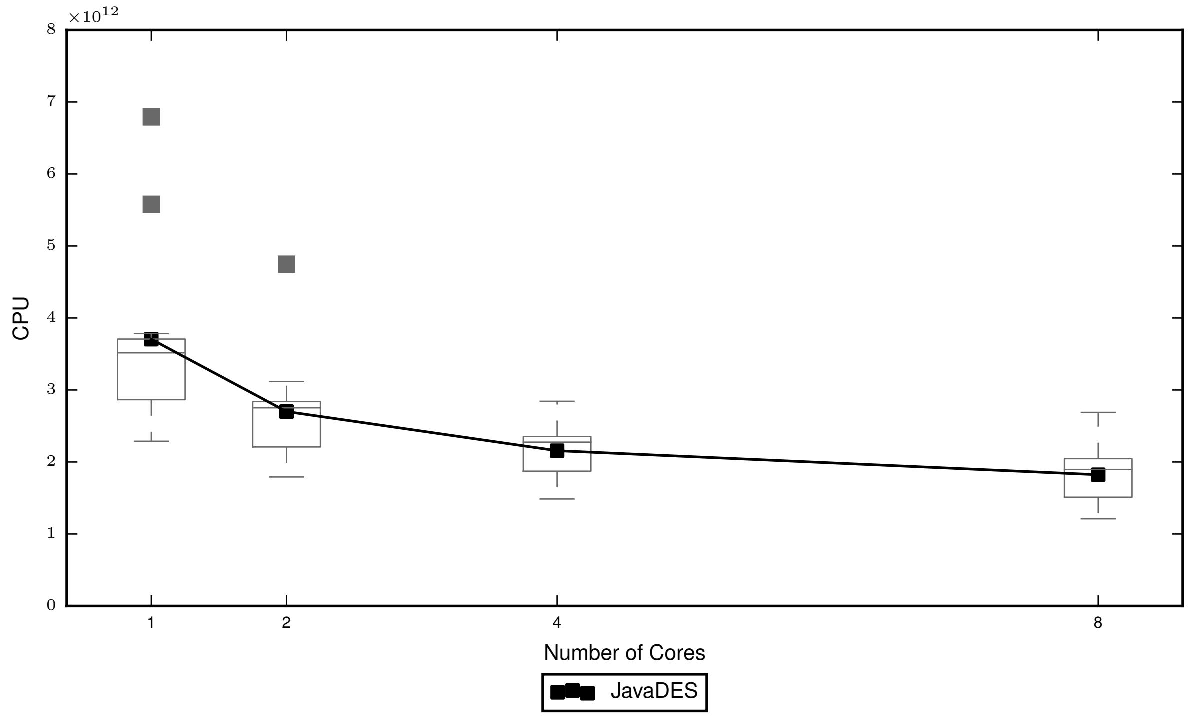

Figure A.5: Box and Whisker Plot of Machine Comparison of JavaDES for Events and Connections vs CPU

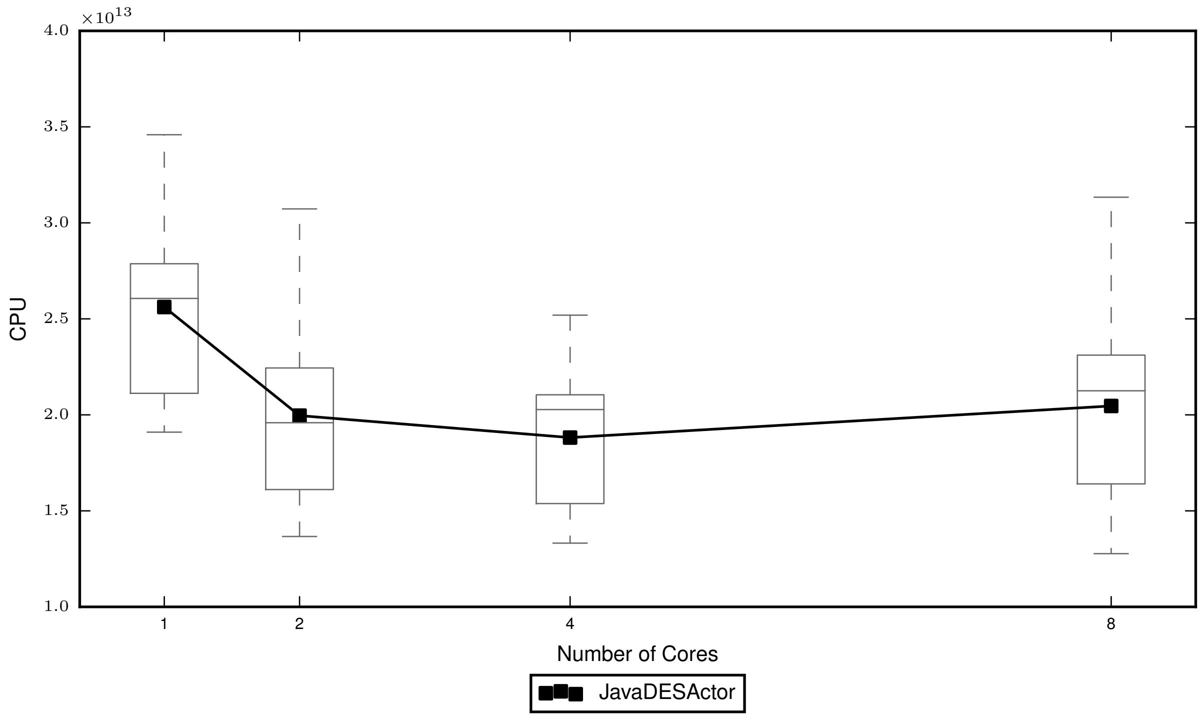

Figure A.6: Box and Whisker Plot of Machine Comparison of JavaDESActor for Events and Connections vs CPU

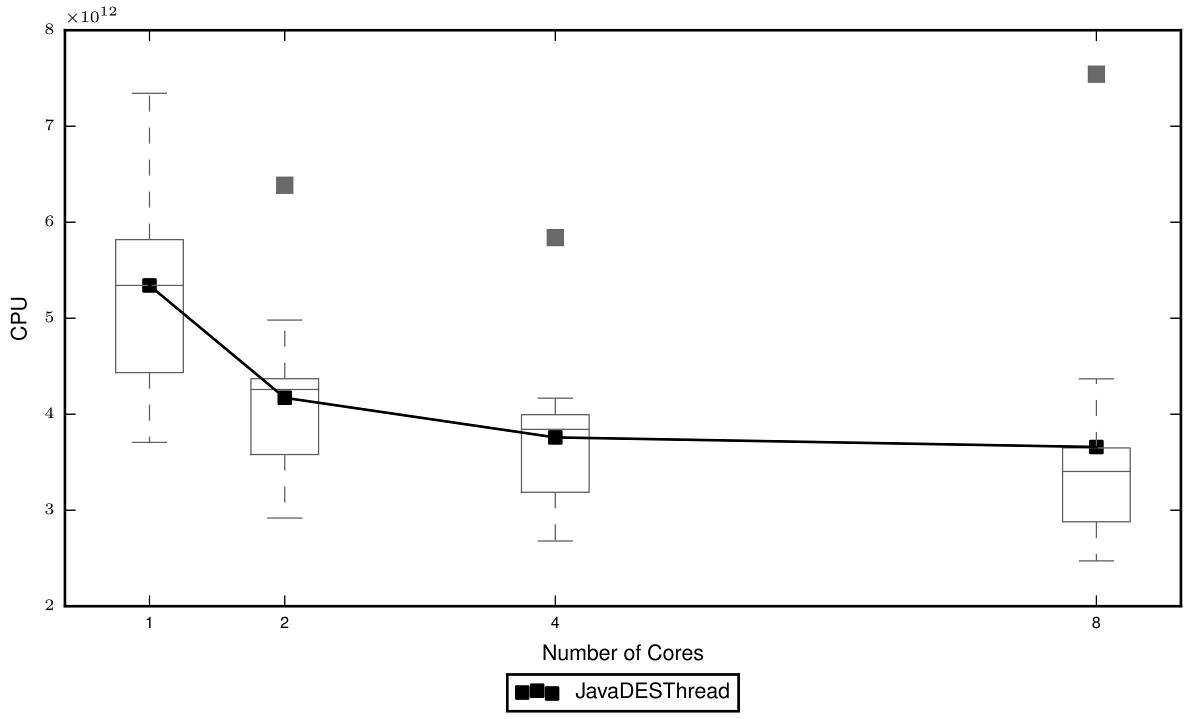

Figure A.7: Box and Whisker Plot of Machine Comparison of JavaDESThread for Events and Connections vs CPU

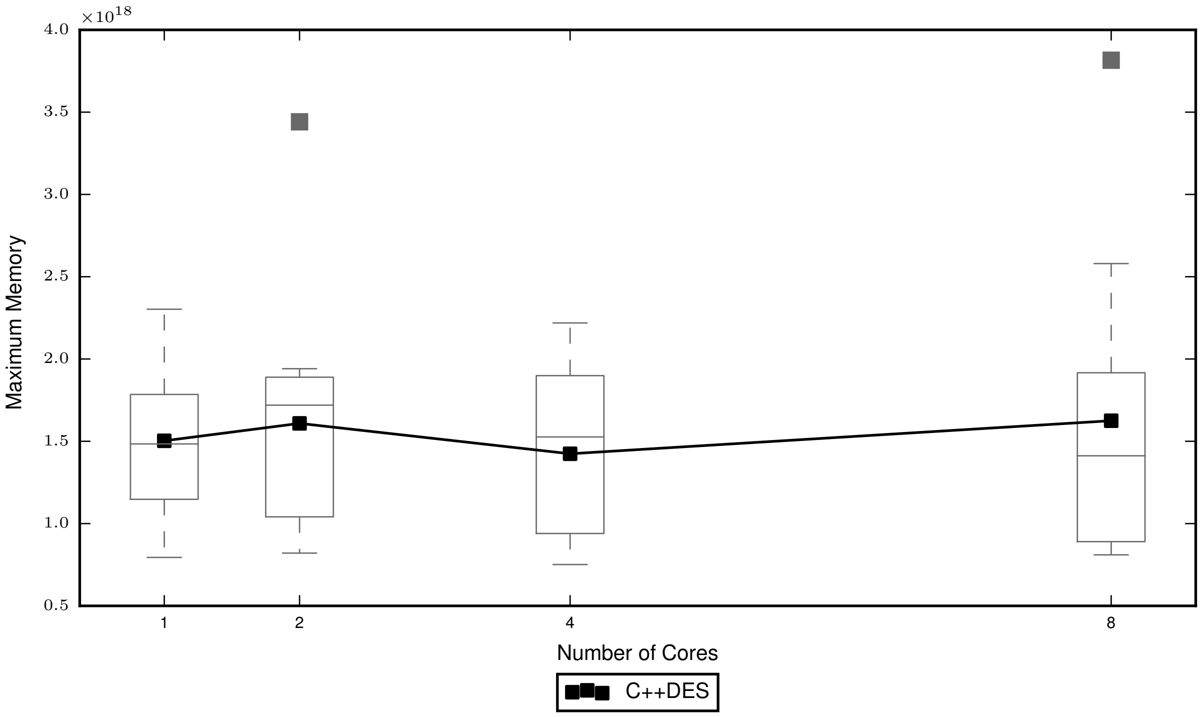

Figure A.8: Box and Whisker Plot of Machine Comparison of C++DES for Events and Agents and Connections vs Maximum Memory

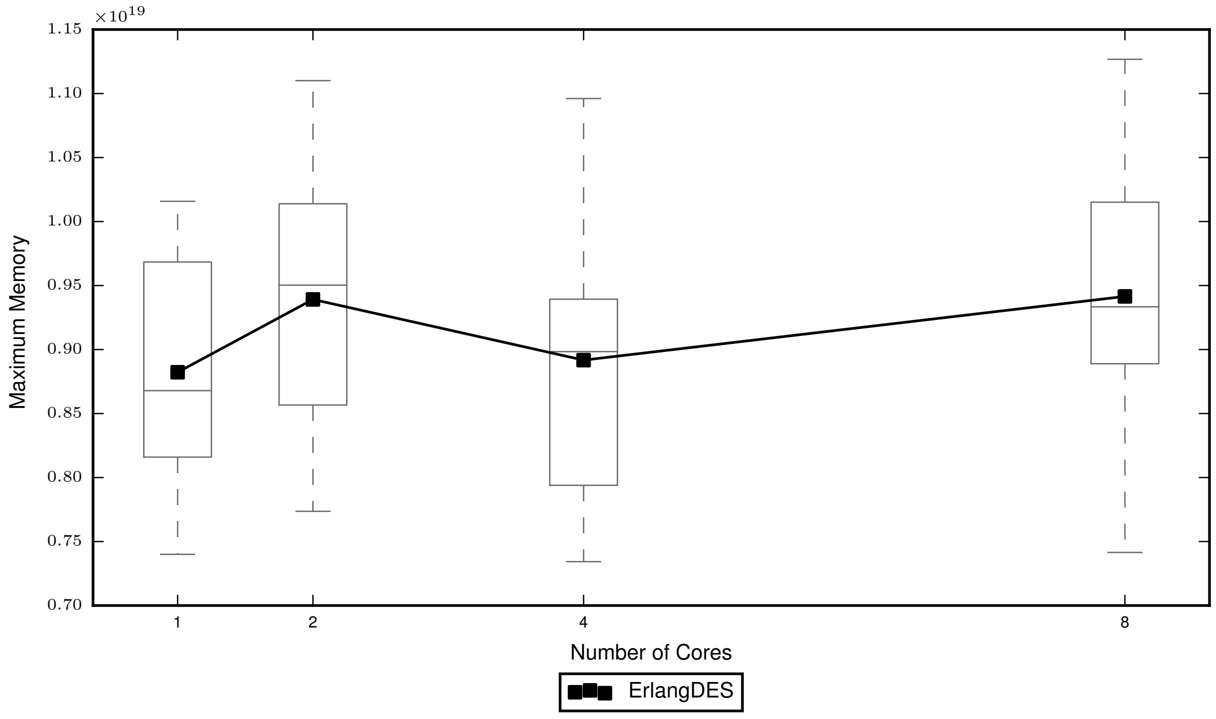

Figure A.9: Box and Whisker Plot of Machine Comparison of ErlangDES for Events and Agents and Connections vs Maximum Memory

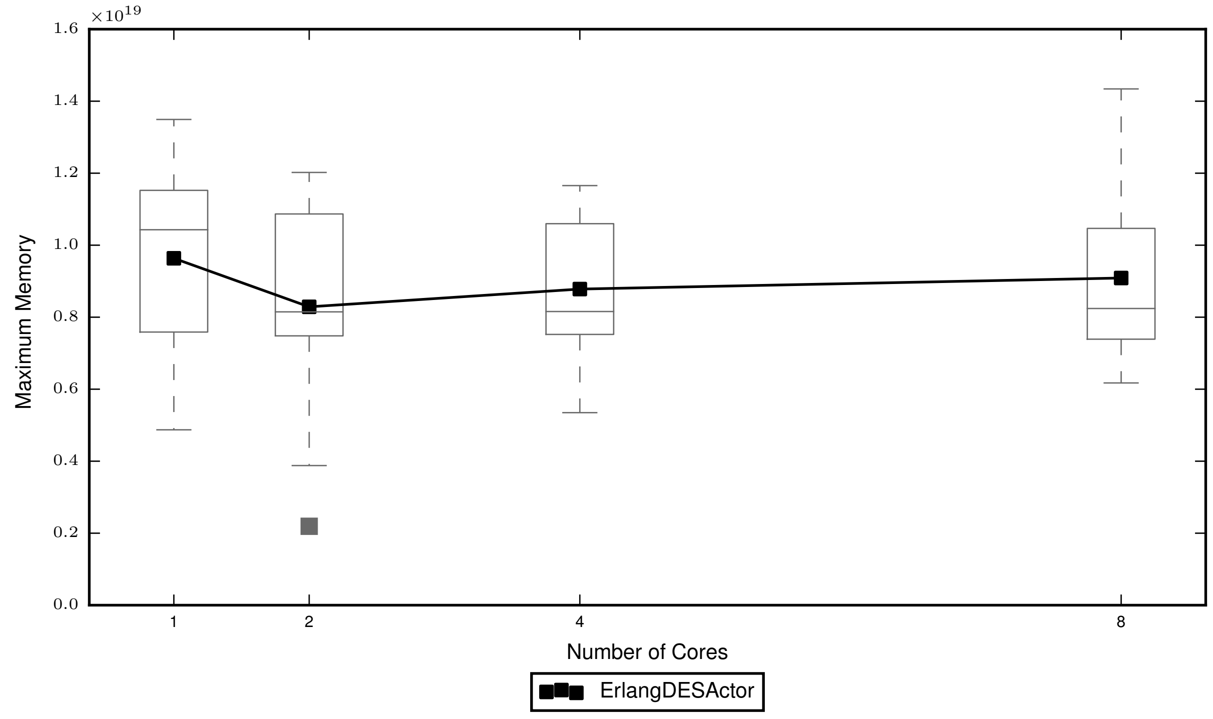

Figure A.10: Box and Whisker Plot of Machine Comparison of ErlangDESActor for Events and Agents and Connections vs Maximum Memory

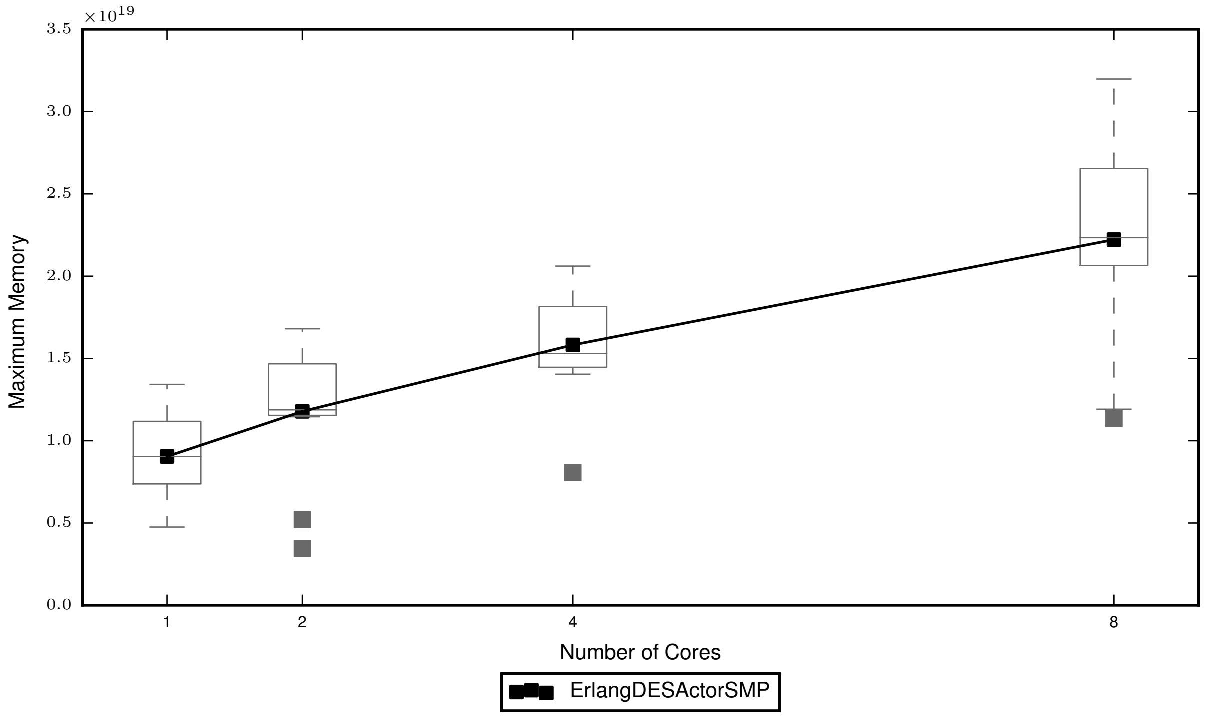

Figure A.11: Box and Whisker Plot of Machine Comparison of ErlangDESActorSMP for Events and Agents and Connections vs Maximum Memory

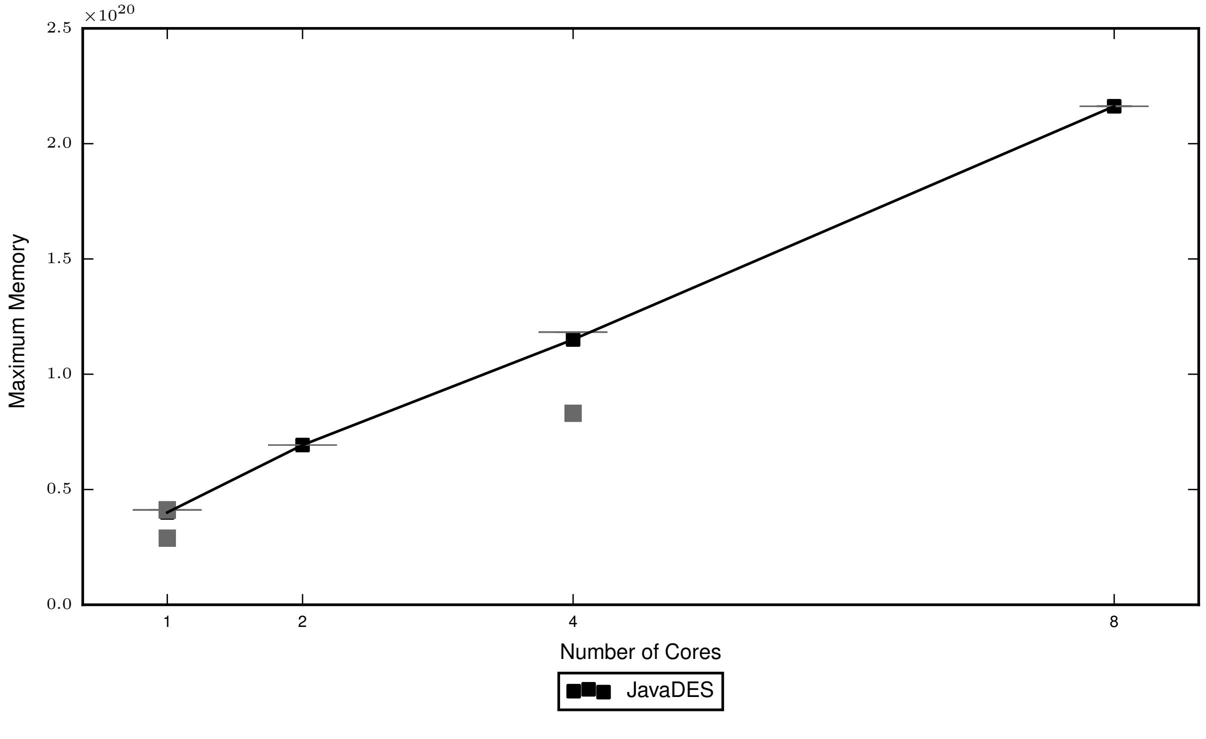

Figure A.12: Box and Whisker Plot of Machine Comparison of JavaDES for Events and Agents and Connections vs Maximum Memory

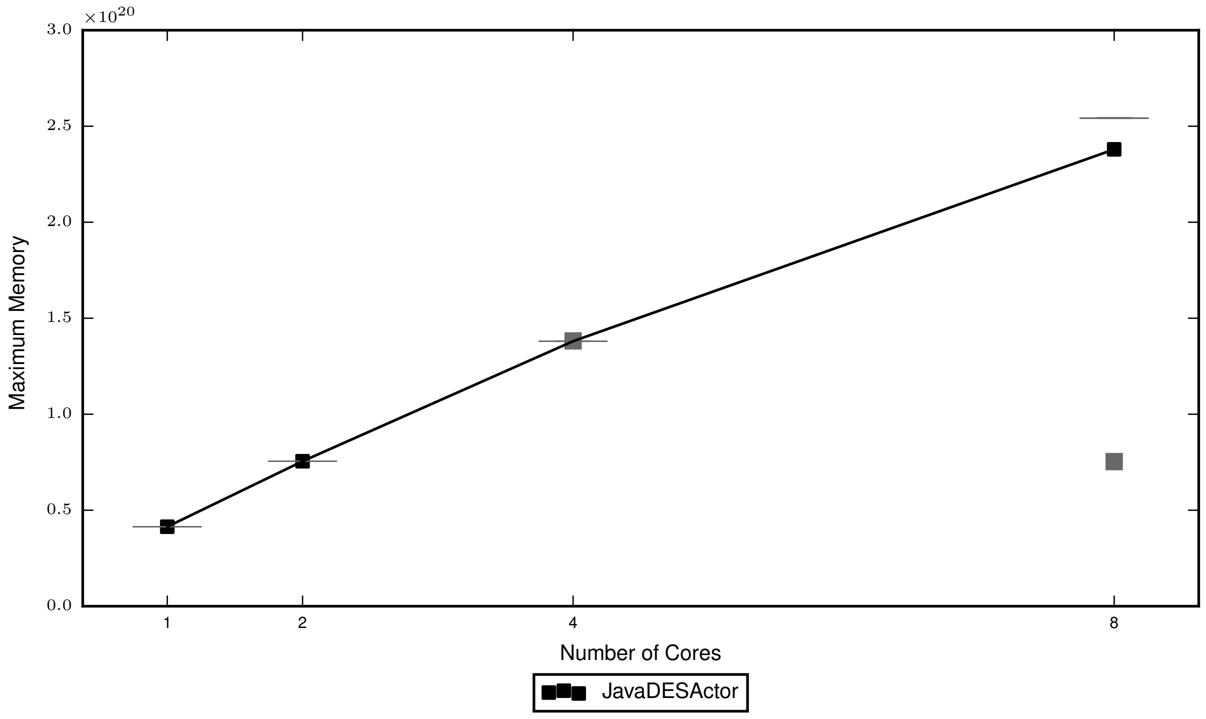

Figure A.13: Box and Whisker Plot of Machine Comparison of JavaDESActor for Events and Agents and Connections vs Maximum Memory

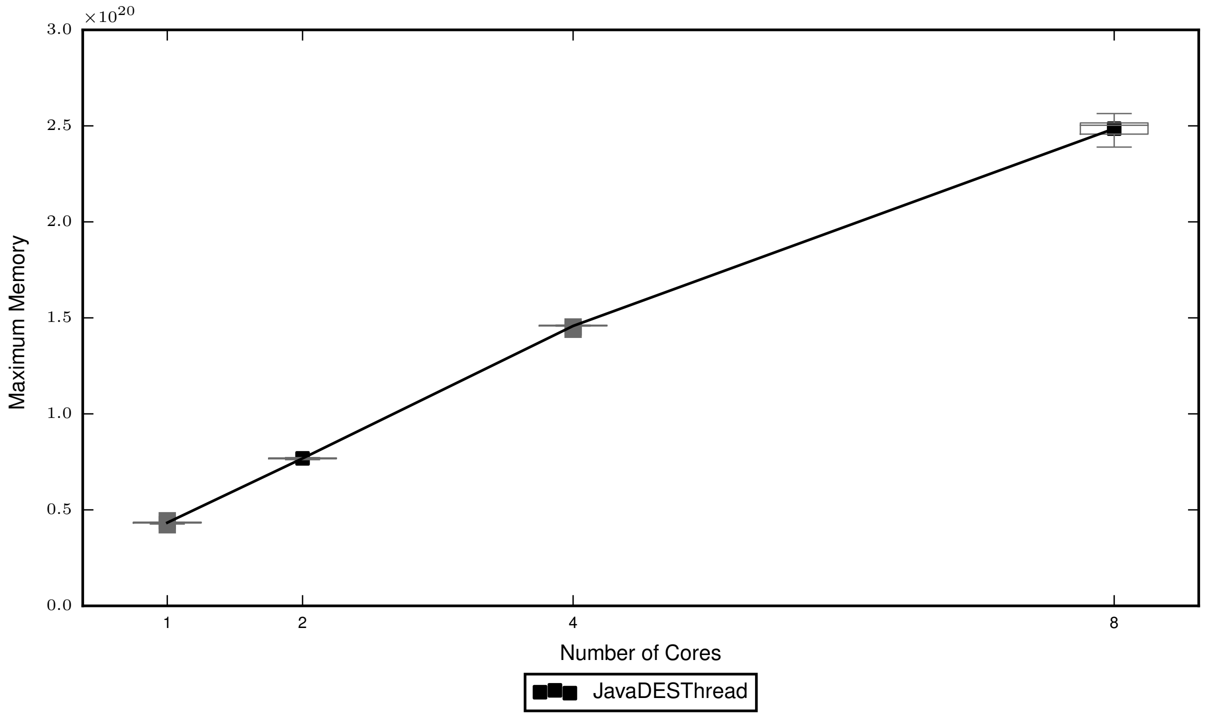

Figure A.14: Box and Whisker Plot of Machine Comparison of JavaDESThread for Events and Agents and Connections vs Maximum Memory

Appendix B

Experiment Configuration Files

```
{
    "kernels": [
        "kernels/rysim_des_actor_erlang/rysim_des_actor",
        "kernels/rysim_des_actor_erlang_smp/rysim_des_actor_smp",
        "kernels/rysim_des_actor_java/rysim_des_actor-1.0/bin/rysim_des_actor",
        "kernels/rysim_des_c++/rysim",
        "kernels/rysim_des_erlang/rysim_des",
        "kernels/rysim_des_java/rysim_des-1.0/bin/rysim_des",
        "kernels/rysim_des_thread_java/rysim_des_thread-1.0/bin/rysim_des_thread"
    ]
}
```

Listing B.1: Kernel JSON specifications used in experiments

```
{
    "generators": [
        {
            "base_name": "CompleteGenerator",
            "generator_type" : "complete",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 20,
            "agents_end": 2000
        },
        {
            "base_name": "CompleteBiparite1x2Generator",
            "generator_type" : "complete-bipartite",
            "generator_params" : [ 1, 2 ],
            "looping" : true,
            "agents_start": 20,
            "agents_end": 2000
        },
        {
            "base_name": "SmallModelGenerator4x0.9",
            "generator_type" : "Watts-Strogatz",
            "generator_params" : [ 4, 0.9 ],
            "looping" : true,
            "state_weightings" : [ 0.8, 0.1, 0.05, 0.05 ],
            "agents_start": 1000,
            "agents_end": 100000
        },
        {
            "base_name": "CircularLadder",
            "generator_type" : "circular-ladder",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 1000,
            "agents_end": 100000
        },
        {
            "base_name": "Cycle",
            "generator_type" : "cycle",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 5000,
            "agents_end": 500000
        },
        {
            "base_name": "Periodic2Grid1x10",
            "generator_type" : "periodic-2grid",
            "generator_params" : [ 1, 10 ],
            "looping" : true,
            "agents_start": 1000,
            "agents_end": 100000
        },
        {
            "base_name": "NonPeriodic2Grid1x1",
            "generator_type" : "nonperiodic-2grid",
            "generator_params" : [ 1, 1 ],
            "looping" : true,
            "agents_start": 1000,
            "agents_end": 100000
        },
        {
            "base_name": "Hypercube",
            "generator_type" : "hypercube",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 1000,
            "agents_end": 100000
        },
        {
            "base_name": "Star",
            "generator_type" : "star",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 5000,
            "agents_end": 500000
        },
        {
            "base_name": "Wheel",
            "generator_type" : "wheel",
            "generator_params" : [ ],
            "looping" : true,
            "agents_start": 1000,
            "agents_end": 100000
        },
        {
            "base_name": "ErdosReyni0.5",
            "generator_type" : "erdos-reyni",
            "generator_params" : [ 0.5 ],
            "looping" : true,
            "agents_start": 60,
            "agents_end": 6000
        }
    ]
}
```

Listing B.2: Generator JSON specifications used in experiments

```
{
    "distributions" : [
        {
            "label" : "DistributionFlat100",
            "type" : "Flat",
            "params" : [ 1, 100 ]
        },
        {
            "label" : "DistributionFlat5",
            "type" : "Flat",
            "params" : [ 1, 5 ]
        },
        {
            "label" : "DistributionFlat10",
            "type" : "Flat",
            "params" : [ 1, 10 ]
        },
        {
            "label" : "DistributionFlat25",
            "type" : "Flat",
            "params" : [ 1, 25 ]
        },
        {
            "label" : "DistributionFlat50",
            "type" : "Flat",
            "params" : [ 1, 50 ]
        },
        {
            "label" : "DistributionPoisson1x5",
            "type" : "Poisson",
            "scale" : 5.0,
            "params" : [ 1 ]
        },
        {
            "label" : "DistributionPoisson1x20",
            "type" : "Poisson",
            "scale" : 20.0,
            "params" : [ 1 ]
        },
        {
            "label" : "DistributionPoisson1x1",
            "type" : "Poisson",
            "scale" : 1.0,
            "params" : [ 1 ]
        },
        {
            "label" : "DistributionPoisson4x5",
            "type" : "Poisson",
            "scale" : 5.0,
            "params" : [ 4 ]
        },
        {
            "label" : "DistributionPoisson4x20",
            "type" : "Poisson",
            "scale" : 20.0,
            "params" : [ 4 ]
        },
        {
            "label" : "DistributionPoisson4x1",
            "type" : "Poisson",
            "scale" : 1.0,
            "params" : [ 4 ]
        },
        {
            "label" : "DistributionPoisson10x5",
            "type" : "Poisson",
            "scale" : 5.0,
            "params" : [ 10 ]
        },
        {
            "label" : "DistributionPoisson10x20",
            "type" : "Poisson",
            "scale" : 20.0,
            "params" : [ 10 ]
        },
        {
            "label" : "DistributionPoisson10x1",
            "type" : "Poisson",
            "scale" : 1.0,
            "params" : [ 10 ]
        },
        {
            "label": "DistributionGaussianTail0x10",
            "type": "GaussianTail",
            "params": [0, 10]
        },
        {
            "label": "DistributionGaussianTail0x25",
            "type": "GaussianTail",
            "params": [0, 25]
        },
        {
            "label": "DistributionGaussianTail0x100",
            "type": "GaussianTail",
            "params": [0, 100]
        },
        {
            "label": "DistributionGaussianTail5x10",
            "type": "GaussianTail",
            "params": [5, 10]
        },
        {
            "label": "DistributionGaussianTail5x25",
            "type": "GaussianTail",
            "params": [5, 25]
        },
        {
            "label": "DistributionGaussianTail5x100",
            "type": "GaussianTail",
            "params": [5, 100]
        },
        {
            "label": "DistributionGaussianTail10x10",
            "type": "GaussianTail",
            "params": [10, 10]
        },
        {
            "label": "DistributionGaussianTail10x25",
            "type": "GaussianTail",
            "params": [10, 25]
```

```
},
{
    "label": "DistributionGaussianTail10x100",
    "type": "GaussianTail",
    "params": [10, 100]
},
{
    "label": "DistributionExponential1x1",
    "type": "Exponential",
    "scale": 1.0,
    "params": [ 1.0 ]
},
{
    "label": "DistributionExponential1x5",
    "type": "Exponential",
    "scale": 5.0,
    "params": [ 1.0 ]
},
{
    "label": "DistributionExponential1x10",
    "type": "Exponential",
    "scale": 10.0,
    "params": [ 1.0 ]
},
{
    "label": "DistributionExponential0.5x1",
    "type": "Exponential",
    "scale": 1.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionExponential0.5x5",
    "type": "Exponential",
    "scale": 5.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionExponential0.5x10",
    "type": "Exponential",
    "scale": 10.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionExponential0.15x1",
    "type": "Exponential",
    "scale": 1.0,
    "params": [ 0.15 ]
},
{
    "label": "DistributionExponential0.15x5",
    "type": "Exponential",
    "scale": 5.0,
    "params": [ 0.15 ]
},
{
    "label": "DistributionExponential0.15x10",
    "type": "Exponential",
    "scale": 10.0,
    "params": [ 0.15 ]
```

```
},
{
    "label": "DistributionLognormal5x0.5x1",
    "type": "Lognormal",
    "scale": 1.0,
    "params": [ 5, 0.5 ]
},
{
    "label": "DistributionLognormal5x1.5x1",
    "type": "Lognormal",
    "scale": 1.0,
    "params": [ 5, 1.5 ]
},
{
    "label": "DistributionLognormal0x1x1",
    "type": "Lognormal",
    "scale": 1.0,
    "params": [ 0.0, 1.0 ]
},
{
    "label": "DistributionLognormal0x1x5",
    "type": "Lognormal",
    "scale": 5.0,
    "params": [ 0.0, 1.0 ]
},
{
    "label": "DistributionLognormal0x1x10",
    "type": "Lognormal",
    "scale": 10.0,
    "params": [ 0.0, 1.0 ]
},
{
    "label": "DistributionLognormal0x0.5x1",
    "type": "Lognormal",
    "scale": 1.0,
    "params": [ 0.0, 0.5 ]
},
{
    "label": "DistributionLognormal0x0.5x5",
    "type": "Lognormal",
    "scale": 5.0,
    "params": [ 0.0, 0.5 ]
},
{
    "label": "DistributionLognormal0x0.5x10",
    "type": "Lognormal",
    "scale": 10.0,
    "params": [ 0.0, 0.5 ]
},
{
    "label": "DistributionLognormal0x0.25x1",
    "type": "Lognormal",
    "scale": 1.0,
    "params": [ 0.0, 0.25 ]
},
{
    "label": "DistributionLognormal0x0.25x5",
    "type": "Lognormal",
    "scale": 5.0,
    "params": [ 0.0, 0.25 ]
},
{
    "label": "DistributionLognormal0x0.25x10",
    "type": "Lognormal",
    "scale": 10.0,
    "params": [ 0.0, 0.25 ]
},
{
    "label": "DistributionBernoulli0.1x1",
    "type": "Bernoulli",
    "scale": 1.0,
    "params": [ 0.1 ]
},
{
    "label": "DistributionBernoulli0.5x1",
    "type": "Bernoulli",
    "scale": 1.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionBernoulli0.9x1",
    "type": "Bernoulli",
    "scale": 1.0,
    "params": [ 0.9 ]
},
{
    "label": "DistributionBernoulli0.1x5",
    "type": "Bernoulli",
    "scale": 5.0,
    "params": [ 0.1 ]
},
{
    "label": "DistributionBernoulli0.5x5",
    "type": "Bernoulli",
    "scale": 5.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionBernoulli0.9x5",
    "type": "Bernoulli",
    "scale": 5.0,
    "params": [ 0.9 ]
},
{
    "label": "DistributionBernoulli0.1x10",
    "type": "Bernoulli",
    "scale": 10.0,
    "params": [ 0.1 ]
},
{
    "label": "DistributionBernoulli0.5x10",
    "type": "Bernoulli",
    "scale": 10.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionBernoulli0.9x10",
    "type": "Bernoulli",
    "scale": 10.0,
    "params": [ 0.9 ]
},
{
    "label": "DistributionBinomial0.5x20",
    "type": "Binomial",
    "params": [ 0.5, 20 ]
},
{
    "label": "DistributionBinomial0.7x20",
    "type": "Binomial",
    "params": [ 0.7, 20 ]
},
{
    "label": "DistributionBinomial0.5x40",
    "type": "Binomial",
    "params": [ 0.5, 40 ]
},
{
    "label": "DistributionNegativeBinomial0.3x7",
    "type": "NegativeBinomial",
    "params": [ 0.3, 7 ]
},
{
    "label": "DistributionNegativeBinomial0.2x1",
    "type": "NegativeBinomial",
    "params": [ 0.2, 1 ]
},
{
    "label": "DistributionNegativeBinomial0.6x10",
    "type": "NegativeBinomial",
    "params": [ 0.6, 10 ]
},
{
    "label": "DistributionGeometric0.2x1",
    "type": "Geometric",
    "scale": 1.0,
    "params": [ 0.2 ]
},
{
    "label": "DistributionGeometric0.2x5",
    "type": "Geometric",
    "scale": 5.0,
    "params": [ 0.2 ]
},
{
    "label": "DistributionGeometric0.2x10",
    "type": "Geometric",
    "scale": 10.0,
    "params": [ 0.2 ]
},
{
    "label": "DistributionGeometric0.5x1",
    "type": "Geometric",
    "scale": 1.0,
    "params": [ 0.5 ]
},
{
    "label": "DistributionGeometric0.5x5",
        "type": "Geometric",
        "scale": 5.0,
        "params": [ 0.5 ]
    },
    {
        "label": "DistributionGeometric0.5x10",
        "type": "Geometric",
        "scale": 10.0,
        "params": [ 0.5 ]
    },
    {
        "label": "DistributionGeometric0.8x1",
        "type": "Geometric",
        "scale": 1.0,
        "params": [ 0.8 ]
    },
    {
        "label": "DistributionGeometric0.8x5",
        "type": "Geometric",
        "scale": 5.0,
        "params": [ 0.8 ]
    },
    {
        "label": "DistributionGeometric0.8x10",
        "type": "Geometric",
        "scale": 10.0,
        "params": [ 0.8 ]
    }
]
}
```

Listing B.3: Distribution JSON specifications used in experiments