Supporting Interactive Computing Features for MASS Library: Rollback and Monitoring System

Nasser Alghamdi

A Capstone Project Proposal
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington | Bothell

2020

Reading Committee:

Munehiro Fukuda, Chair

Arnie Lund

Kelvin Sung

Program Authorized to Offer Degree:
School of Science, Technology, Engineering & Mathematics

University of Washington | Bothell

Abstract

Supporting Interactive Computing Features
for MASS Library: Rollback and Monitoring System

Nasser Alghamdi

Chair of the Supervisory Committee:
Ph.D. Munehiro Fukuda
Computing & Software Systems

M**ulti-**A**gent **S**patial **S**imulation* (MASS) library provides parallel execution over multiple computing-nodes for a wide range of agent-based simulations and data science applications. It conceals the complexity of parallel execution, dynamic and static entities allocation, and management process behind a set of APIs. Developing MASS applications for non-computing specialists or novices is a challenging and time-consuming task. Applications that depend on the library have to be compiled, distributed, executed for every change that is introduced by the user. Additionally, the user has to use distributed log files or additional library calls for probing the application state. Thus, the user spends more time when experimenting on re-compiling, re-distributing, re-executing the application executable, and gathering information from distributed logs or results of additional calls. Though the library provides an intuitive programming model, its rigidity and the lack of convenient inspection tools can draw users away from using the library. In this project, we introduce InMASS (In**teractive computing feature for **MASS** library*) with two supporting features, namely: monitoring tool and rollback. We design computer experiment to emulate user-changes and we found that interactive version performs 9.2 times faster than non-interactive version when experimenting in ABM settings. Also, We compare and demonstrate how the rollback and monitoring tool adds flexibility and observability, respectively, to ABM systems by comparing InMASS

against Repast Simphony, a well-known ABM framework.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
INTRODUCTION

This paper presents interactive features and backtrackable computation for ABM systems, provides concrete implementation on an ABM library (MASS), quantitatively measures potential speedup of interactivity in ABM settings, and qualitatively compares the architecture to a well-known framework (Repast Simphony) that has interactive capabilities.

1.1 Background

Agent-Based Modeling (ABM) is a multidisciplinary technique for simulating systems in bottom-up fashion where up-down techniques are hard or even impossible to implement. In ABM, the user starts by defining individual's (agent) rules which dictate the interactions within the system's environment and maybe with different agents in multi-agents systems (MAS). The ending result is an emergent behaviors or even a continuous-change of the system's state that is based on the pre-defined rules and the initial state of its system. The resultant behaviors or state, hopefully, explains the system as a whole [9]. ABM's applications span across several disciplines including, but is not limited to, social science, data science, bioinformatics, economic, physics, and computation.

1.1.1 MASS

MASS[1] facilitates a programming model for parallelizing a wide range of agent-based micro-simulation programs including physical simulation (e.g., wave dissemination and molecular

[1]MASS is an abbreviation for for ***M**ulti-**A**gent **S**patial **S**imulation* library which is developed by Distributed Systems Laboratory (DSLab) at Computing and Software Systems Department, University of Washington | Bothell.

dynamics) as well as social, behavioral, and economic simulation (e.g., social network, artificial life, and bank/investor/firm). It also facilitates agent-based data sciences and optimization such as ant colonial optimization and particle swarm optimization. The parallelization process in MASS is built around two main classes: Places and Agents. Places are elements that are statically allocated to different threads across multiple computing nodes. On the other hand, agents are execution entities that autonomously move over different processes. The library exposes a set of APIs that conceal the complexity of thread and process spawning, mapping, execution, communication, and termination under the hood, yet the end-user must learn how to use them [3].

The MASS paper [3] and the MASS User Manual [19] follow theroitical and technical approaches for explaining MASS and its programming model. The MASS library exposes five classes: (1) MASS, (2) Place, (3) Agent, (4) Places, and (5) Agents. MASS class has two distinct functions: one for initilizing the cluster and the other for terminating. Internally, it maintains connections between the node that user interacting with and the remote daemon process or processes. Place and Agent are intended to be extended by the user to define the place and agent models. Extention classes maintains data and actions which both represent the application state. Places and Agents are controling units in which they provide a set of functions that operate on place and agent instances respectively. Operations preformed by Places class include creating elements, exchanging messages, and invoking methods on Place instances. While Agents class operations include creating, spawning, killing, migrating entities, and invoking methods on Agent instances.

1.1.2 InMASS

An **In**teractive computing feature was implemented for the **MASS** library (InMASS). It is a wrapper around JShell Tool which allows the MASS APIs to be executed interactively. It benefits from both the MASS communication model and JShell Tool[2] functionalities.

[2]JShell Tool is an interactive shell for Java programming language which is shipped with JDK, Java Development Kit, version 9 or higher [14, 6].

First, it injects the MASS initialization code into JShell. Next, a JShell prompt is returned where the user can interact with the MASS components: MASS, Places, Agents, Place, and Agent classes. Finally, cleanup is automatically called on an exit event. It uses the MASS communication functionalities to disseminate any class that is defined by the user at runtime.

1.2 Motivation and Project Goals

The MASS library has a smooth learning curve and intuitive programming model. However, the lack of observability tools and flexibility makes tracking the application state and tuning the code a tiresome process. Figure 1.1 illustrates evolvement process of an application when using MASS library. The users start with writing initial code. Next, they compile the source code with its dependencies. Before running the application and when experimenting on a cluster, the executable must be distributed to each computing node in the cluster. After execution, the user collects the results. Result can be written in log files or it can be collected via an additional library calls and printed on the console window. Next, the users evaluate whether the output meets the expectations or not. If not, they go back to the first step and make adjustments, modifying the code. This cycle keeps going until they reach the expected outcome.

Besides the interactive computing feature, we introduce monitoring and rollback features. InMASS eliminates repeatitive tasks such as code compilation, executable distribution, and library initialization. While, monitoring provides state observability without having to collect the state manually from multiple computing nodes. And rollback feature allows the users to revert the state to a previous point in time. For example, when users have multiple steps of computations, rollback feature can fix the state at a specific point in time, and from that point users can experiment different computations with no need to re-create the state for each experimentation. Collectively, they should make agent-based modeling approachable by beginners. To conclude, interactive computing allows users to experiment much faster than non-interactive environment while rollback and monitoring facilitate computation flexibility and state observability, respectively.

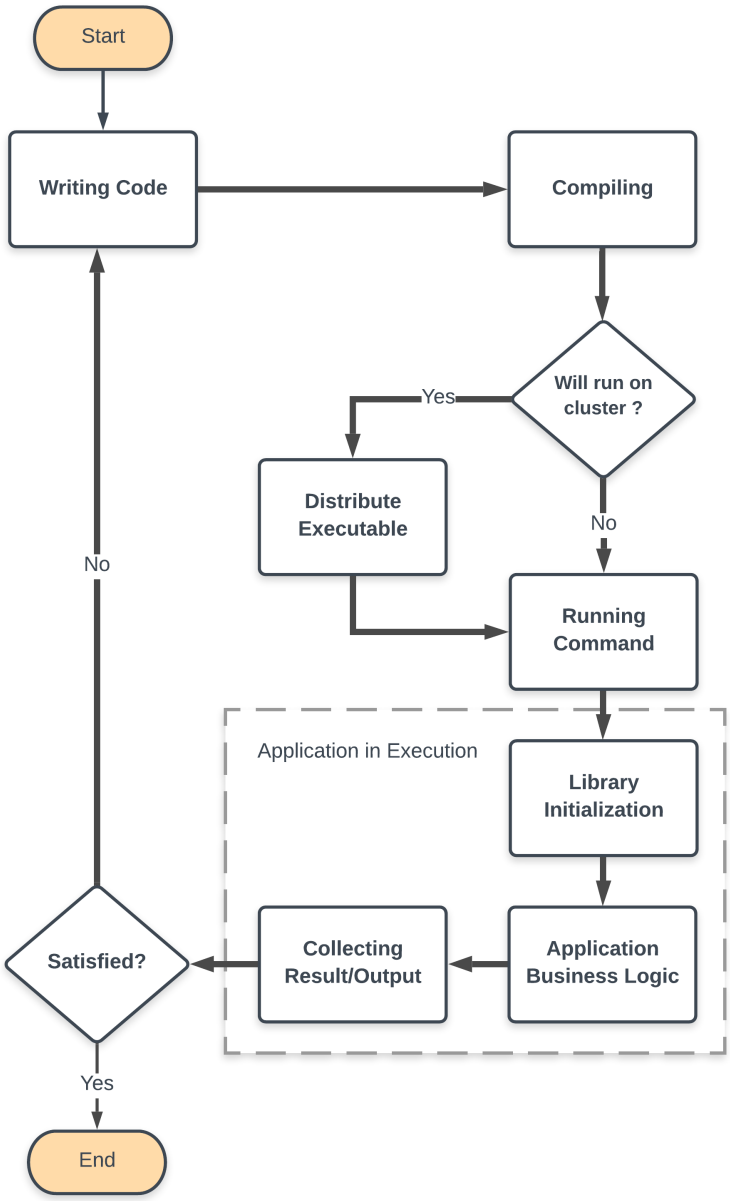


Figure 1.1: Experimentation in Non-Interactive Environment

The project goals are: (1) speedup the experimentation process by providing results to novices and non-computing specialists much faster than the compile-then-run approach; (2) a monitoring tool for observing MASS application state at runtime; (3) and a rollback feature that allows the user to checkpoint a program's state and retrieves it as needed.

The rest of the paper contains the following four sections: section 2 discusses Repast Simphony, monitoring in distributed systems, previous work within DSLab; section 3 discusses MASS programing model, rollback and monitoring architectures, and support for non-interactive applications; section 4 discusses computer experiment for emulating user-changes, development speedup, comparison to an existing solution, and it has demonstration for both monitoring and rollback features; section 5 discusses limitations and future work.

Chapter 2
RELATED WORK

There are many software developed to make ABM reachable by novices or even people who have no computing expertise.

2.1 Repast Symphony

REcursive Porous Agent Simulation Toolkit Simphony (Repast S) is an open source software for agent-based modeling and simulation. It uses Eclipse, an open source Integrated Development Environment (IDE), as its primary IDE. Repast S uses a plug-in style for combining its components that makes the framework highly modular. Plug-ins can be integrated at runtime for monitoring, visualization, or analysis.

In Repast S, modeling objects (agents) are held by *context*. Relationship between agents and the context is defined by *projection*. Both the context and projection constitutes the simulation space. It allows simulation time management and behavior activation using discrete event time *scheduler* and *watchers*. Repast S uses Graphics User Interface (GUI) to run the simulations. Users can interactively manipulate the simulations when models are parameterized and properly linked to the presentation layer.

2.2 Monitoring in Distributed Systems

Distributed systems are complex and it is much complex to reason about software and libraries that are running on top of a distributed infrastructure. A wide variety of tools exist to ease the complexity of such systems and to pave the way for monitoring distributed systems [8].

Some monitoring tools such as *Ganglia* [10] provide a resource-based approach for mon-

itoring a cluster state in which they collect resources' metrics such as CPU, memory, and network usages across the cluster. Such solutions can be integrated with existing libraries. However, they do not reflect the state of client applications, that run on top of such libraries, for two reasons. First, the application state is a library-dependent in which the state composition is governed by the library model. Therefore, it is hard to provide a generic solution for all different kinds of libraries. Second, it is against the architectural design decisions of such tools since they are high-level monitoring tools. Distributed libraries use such monitoring tools to drive management decisions rather than watching the application state.

Other libraries such as *Spark* [20], large-scale data processing library, take a step further by collecting metrics related to their predefined execution path. Collected metrics in this approach give insights related to the utilization, execution status, or/and bottlenecks issues. Yet, it does not reflect the application state and it is the users' responsibility to find out what is the state of their applications via what is appropriate for that library, e.g. logging the state into multiple files or adding additional calls for probing the state.

2.3 Previous MASS Java Debugger

MASS Java Debugger is a previous work done by Niko Simonson and Sean Wessels [16] within the DSLab group related to debuging and monitoring. We deviate from that work for two reasons: (1) it is geared toward visualization rather than monitoring and (2) its design poses a limitation when large sets of data are aggregated from all cluster nodes into a single process, the visualization process in that work. That being said, we follow a similar approach to incorporate support for non-interactive applications.

MASS follows a unique agent-based approach leveraging various parallelization techniques [3]. It has two models known as Place and Agent classes. Client application extends these classes which in turn defines its state. However, the library does not provide a support for state monitoring neither its execution path. Only, one can check the MASS application state via multiple logs files (one for each process spawned by the system) or by adding additional library calls for probing the state.

In this work, we follow the later approach where a generic status about various library calls is collected. However, our work eases probing the state via a selection mechanism presented in this paper. In addition to monitoring, we introduce supporting for interactive computing feature that allows checkpointing of the state at a point of time and rollbacking to it as needed. We believe that MASS is the first multi-agent simulation library to allow selective and efficient monitoring for the system components.

Chapter 3

ARCHITECTURE OF MASS JAVA INTERACTIVE COMPUTATION

This section shows the MASS model and the architecture of MASS Java's new interactive features: (1) rollback features and (2) monitoring tool, and thereafter explains about their implementation.

3.1 MASS Library

The MASS model as shown in Figure 3.1 has two fundamental classes: *Agent* and *Place*. *Agent* represents dynamic execution entity that has unique ID and can migrate from a *Place* to another. *Place* represents a static element that has network-independent index, can exchange information and host *Agent* or a set of agents. Each *Agent* must be associated with *Place*. That does not mean all applications have to have both *Agent* and *Place*. Some applications might need only to define *Place*, others have to define both classes.

Other classes are for setup and controlling purposes. The model uses a star topology in which all processes are individually interconnected. Connections between processes are TCP sockets. The model follows master-worker pattern for processes management. There is only one master process that has an ID set to zero, *MASS*. *MProcess* class initilizes the worker process. To avoid ambiguities throughout the paper, master and worker processes will be referred to as daemon processes. The model maintaines collection of agents using *Agents* class and it maintaines collection of places using *Places* class. *Agents* and *Places* each have APIs that operate on their *Agent* and *Place* instances, respectively.

The MASS library has intuitive programming model. Listing 3.1 shows QuickStart application for MASS library. Any MASS application has to start with calling *MASS.init()*,

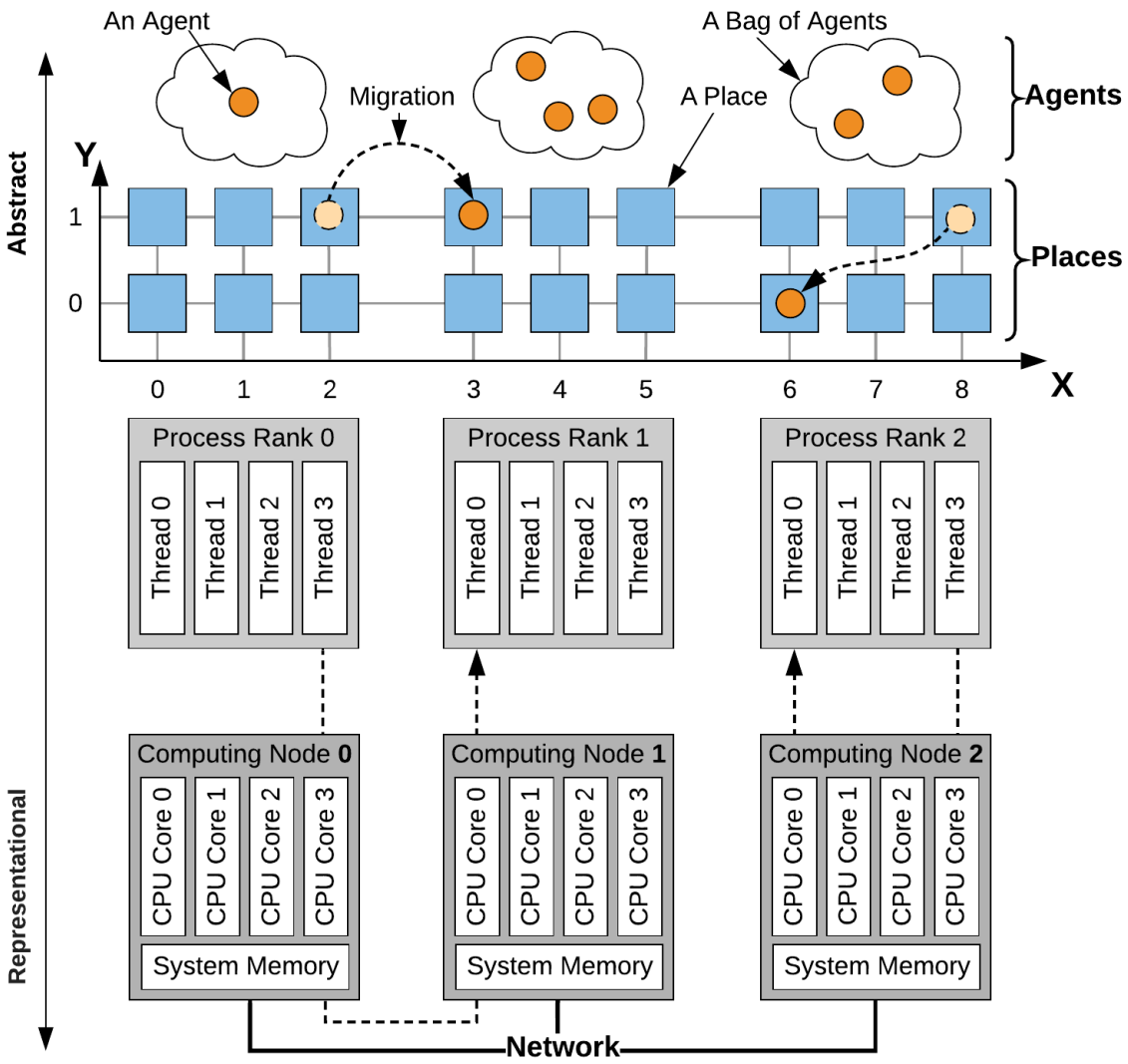


Figure 3.1: MASS Model

line 3, for initialization, e.g. spawning execution threads, spawning daemon processes, and establishing network connections. Also, it has to end with *MASS.finish()*, line 10, for releasing all resources, e.g killing spawned threads, terminating daemon processes and closing the network connections. Note the two previous conditions are not applicable for InMASS since the functions are automatically called. *Places* and *Agents* constructors, lines 4 and 5, are used to create *Place* and *Agent* instances, respectively. The library facilitates behaviors activation (function calls) in parallel for all instances using *Places.callAll(int functionId)* for *Place* instances or using *Agents.callAll(int functionId)* for *Agent* instances as shown in lines 6 and 8, respectively. Information can be exchnaged between *Place* instances using *Places.exchangeAll(int functionId)*, line 7. The *Agents.callAll(int functionId)* can trigger migration, termination, or spawning behaviors for *Agent* instances. Such behaviors can be committed using *Agents.manageAll()* as shown in line 9.

Listing 3.1: MASS Programming Model: QuickStart Application

```
1   public class QuickStart {
2       public static void main( String[] args ) {
3           MASS.init();
4           Places<PlaceModel> places = new Places<>(...);
5           Agents<AgentModel> agents = new Agents<>(...);
6           places.callAll(PlaceModel.FUNC_ID_1);
7           places.exchangeAll(PlaceModel.FUNC_ID_2);
8           agents.callAll(AgentModel.FUNC_ID_1);
9           agents.manageAll();
10          MASS.finish();
11      }
12  }
```

3.2 Rollback Feature

A MASS application's state is distributed across multiple computing nodes, and it is changing according to various APIs calls. Automatic state preserving, a snapshot before each change, will increasingly consume available memory and might not be feasible for all kind of simulations and data science programs. Thus, an explicit checkpointing provides less memory usage, and it preserves only relevant snapshot for the user.

Rollback feature as shown in Figure 3.2 has two distinct APIs: checkpoint and rollback. There are five tasks when calling checkpoint action: (1) interactive shell delegates the checkpoint command to master process; (2) master holds the execution control and propagates checkpoint command to all workers; (3) all daemon processes –master and workers– write their states locally; (4) when workers finish state writing, they send acknowledgment to the master; (5) after receiving all acknowledgments from workers, master release the execution control back to the user.

Likewise, the rollback action has five tasks. However, it reads the preserved snapshot to be the current state for the system. The rollback command can specify which point in histroy using integer number. In addition to checkpoint and rollback actions, four extra APIs can be used to ease storage configuration and history navigation.

3.2.1 Storage

The user can select from three storing mechanisms for different purposes. There are three APIs for specifying the storing mechanism: (1) *StoreInMemory()*; (2) *StoreInFile()*; and (3)*StoreInDisk()*. First mechanism, the default, is storing the state in the memory. Each cluster node will preserve its protion of the state in its memory. Storing in memory space is the fastest approach among the three since it does not require disk I/O which are relativly slow. However, it consumes the available memory space almost twice than the others do, and it can cause out-of-memory for data intensive applications. The second is storing in sequenced files writen in the disk. It creates sequenced files, one per each computing node.

For example, when calling *StoreInFile('state')*, the files will be created in the current working directory for each daemon process, and the naming of the files will be *state.0.ser* for the first computing node, *state.1.ser* for the second computing node and so on. Since the state is written to files, this mechanism allows restoring the state even when the application craches. Third is similar to the second but it stores in a temporary files that are created and deleted by the operating system. Both (1) and (2) mechanisms are useful to reduce memory usage by writing the state into the disk rather than the memory.

3.2.2 History

Rollback feature keeps track of computation once the state has checkpointed. In checkpoint action, each computing node preserves its local state, which represents a portion of the application state. For further library calls, the master will keep track of each call as an incremental stepping, considering that the checkpoint is the first step. In rollback action, each node restores its state from the preserved snapshot. Next, the master will automatically re-execute the required steps to bring the state back to the user-selected point in history.

3.3 Monitoring Tool

The Monitoring Tool architecture is a mix and match of three-tier server-client and publish-subscribe architecture styles. We consider the following principal design decisions while developing the system:

1. Library calls are the only source that introduces side effects on the state.

2. Data –cluster metrics and the application state represented by Places and Agents data– are collected once per library call and only the necessary portion of the data is streamed to the presentation layer.

3. Data must not be aggregated holistically in a single node. In other words, each cluster node maintains its local portion of the data and the presentation layer queries what is

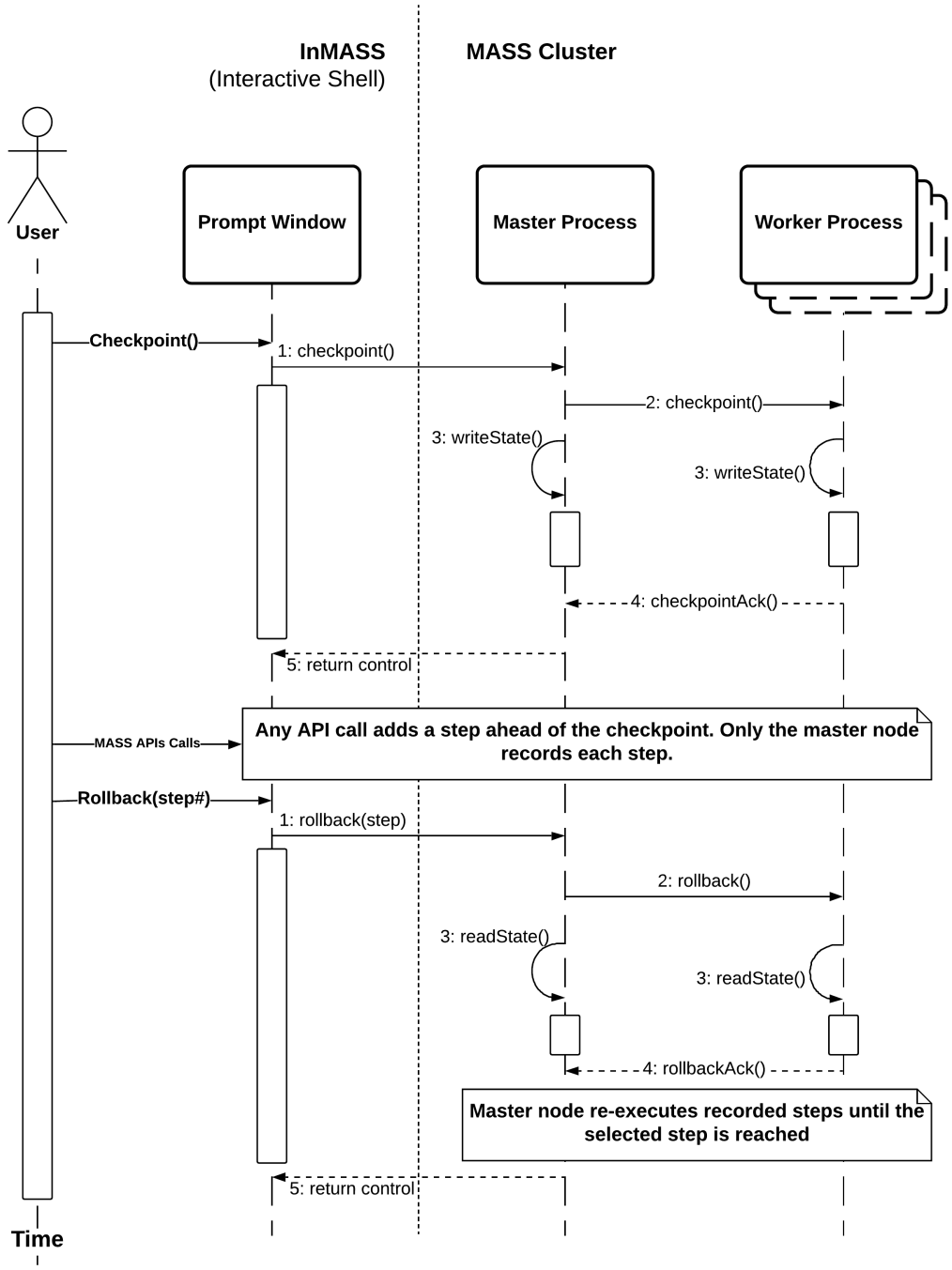


Figure 3.2: Rollback Feature: Architecture

needed for the user.

4. Representation layer must reflect the most recent version of data.

Figure 3.3 depicts the architecture and its components. Each computing node in the MASS cluster maintains *Node, Monitor,* and *Repository* components. The *Node* component is embedded by daemon process (*MASS* or *MProcess* process). *Monitor* and *Repository* are contained within a separate process, which will be referred to as the *monitoring process* in this paper, that resides locally to the daemon process. *UI* is a client (browser) logic that manages both data querying and visualization for the user.

The following are the purposes of each component. *Node* component has three distinct tasks: (1) launching the monitoring process when monitoring flag is set; (2) listening for resume signal when the user explicitly pauses the application; (3) and streaming updated state to the monitoring process whenever the state has changed. *Monitor* component has the following responsibilities: (1) managing the monitoring process; (2) redirect system messages and signals to their appropriate destinations. For example, when user sends resume signal, *Monitor* component will receive the signal from *UI* component then pass it to the *Node* component. When the *Node* streams updated state, *Monitor* component will forward the content to the *Repository* component; (3) serving static HTML/JavaScript code to the *UI* component; (4) and maintaining *UI* connections. *Repository* component responsibilities are: (1) serving *UI* requests such as fetching or subscribing for specific agent; (2) and comparing updated state against the old state in order to notify subscribed clients with the new changes. *UI* component has three tasks: (1) initiating connection per each computing node. The MASS process serves static HTML files with a JavaScript code. These files are provided by the master node only. JavaScript contains necessary logic to setup communication to all computing nodes; (2) visualizing cluster status at runtime; (3) and sending user queries and resume signal to the monitoring process.

The architecture has two types of connectors: Pipes and WebSocket/HTTP[1]. Pipes con-

[1]WebSocket protocol in its current implementation requires HTTP during the connection establishment.

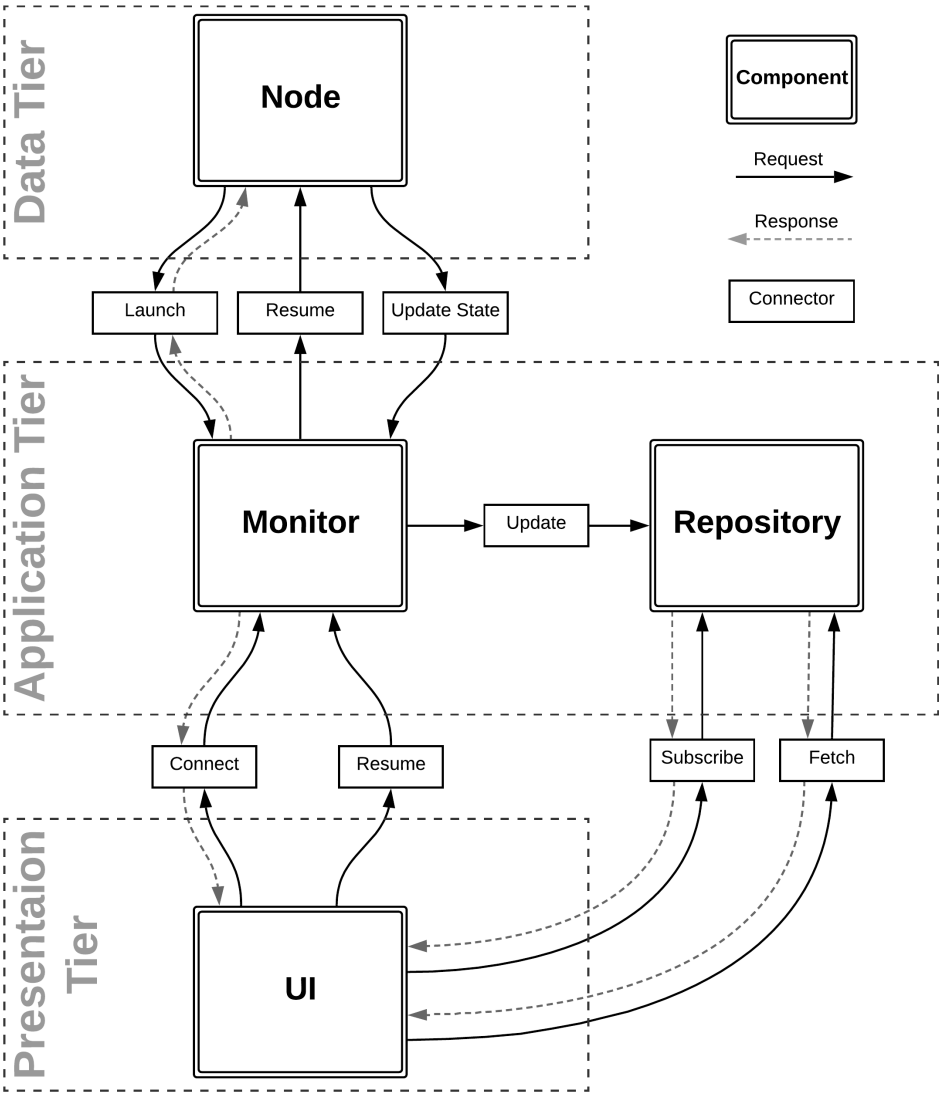


Figure 3.3: Monitoring Tool Architecture

nect daemon process with the monitoring process. Data flow from daemon process to the monitoring process except for the resume signal. Resume signal is triggered from the *UI* then it is passed to the daemon process by monitoring process via another pipe. WebSocket/HTTP connect *UI* with the monitoring process.

The monitoring process depends on Vert.x library [5]. Vert.x provides a lightweight HTTP server that can be easily embedded in the monitoring process. While the *UI* is writen using Angular framework [2]. We decided to go with Angular framework for two reasons: (1) support for type-safe language, TypeScript [17], and (2) consistency thanks to its simplified MVC pattern. However, the architecture is agnostic of any front-end framework.

Collected data from *Node* component are metrics about the cluster status and a user-selected state. For efficiency purposes, the monitoring tool must have an explicit selection mechanism for the application state. Otherways, the system will collect large sets of data from multiple computing nodes into the *UI* component which will cause huge delays. The same for the *UI* when it queries the data, the architecture must provide a selective querying mechanism. Otherways, aggregating all data in the *UI* is not practical for data-intensive application. The architecture is agnostic of these mechanisms yet they are captured by its principal design decisions.

We use Java Annotation and Reflection for enabling state selelcetion mechanism. In our implementation we have two custom annotations: *Inspect* and *Watch* annotations. 3.2 illustrates how the *Node* component utilizes *Java Annotation*. ***Inspect()***, a class annotation, indicates that the annotated class has at least one field to monitor while ***Watch()***, a field annotation, indicates which fields to monitor. By doing so, *Node* component will know which data to look for when collecting the state. It will only collect selected (annotated) fields by the user.

Listing 3.2: QuickStart Application: Agent and Place Models

```
1 @Inspect() // <- Indicate class intended to inspect
2 public class AgentModel extends Agent {
3     ...
```

```
4     @Watch() // <- Indicate property selection
5     private String data;
6     ...
7  }
```

The UI component does not collect the whole state rather it queries only the required state for the active views. Thus, the memory bottleneck at the UI component is avoided. Users can explore the state from the UI component using places or agents identifiers. Users do not have to know about the underlying querying mechanism, only developers might do (See Appendix A for more details)

3.4 Support for Non-Interactive Application

Monitoring Tool and Rollback are built for InMASS, interactive application, in mind. However, we added a few changes to port theirs benefits for non-interactive applications. For that purpose, the monitoring tool operates in two modes: AUTO or ONPAUSE. AUTO mode is intended for the interactive shell, InMASS. When running in AUTO mode, data will be automatically collected per each API call.

While ONPAUSE mode is intended for non-interactive applications. When running in ONPAUSE mode, the cluster must be, explicitly, instructed when to collect the state. Therefore, we add two functions to the MASS APIs. (1) *MASS.collect()*, it instructs each node to forcelly collect the state. (2) *MASS.pause()*, it collects the state much the same as the MASS.collect() does and it holds control right after collecting the state. Execution control will be released upon a resume signal that is triggered by the front-end, UI. Figures 3.4 and 3.5 depict these functions and their behaviours.

It is worthwhile to mention that AUTO mode can be used in non-interactive and the ONPAUSE can be used in interactive applications, too. However, in a non-interactive application, one must hold the execution control in order to successfully observe the state. Otherways, the application might run faster than the user opening a browser window.

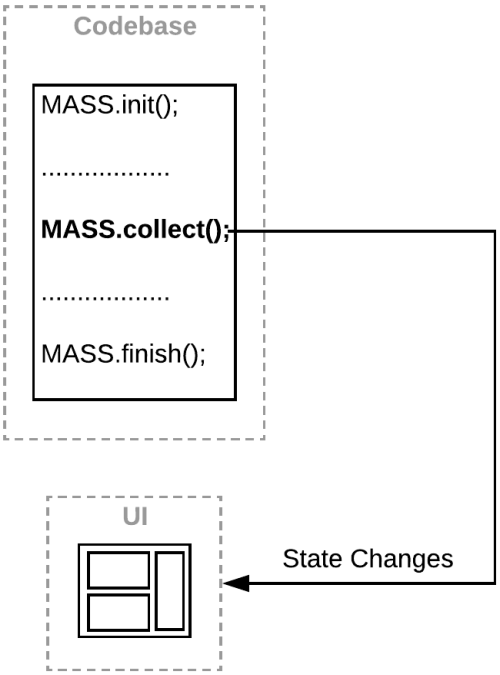


Figure 3.4: Collect Function

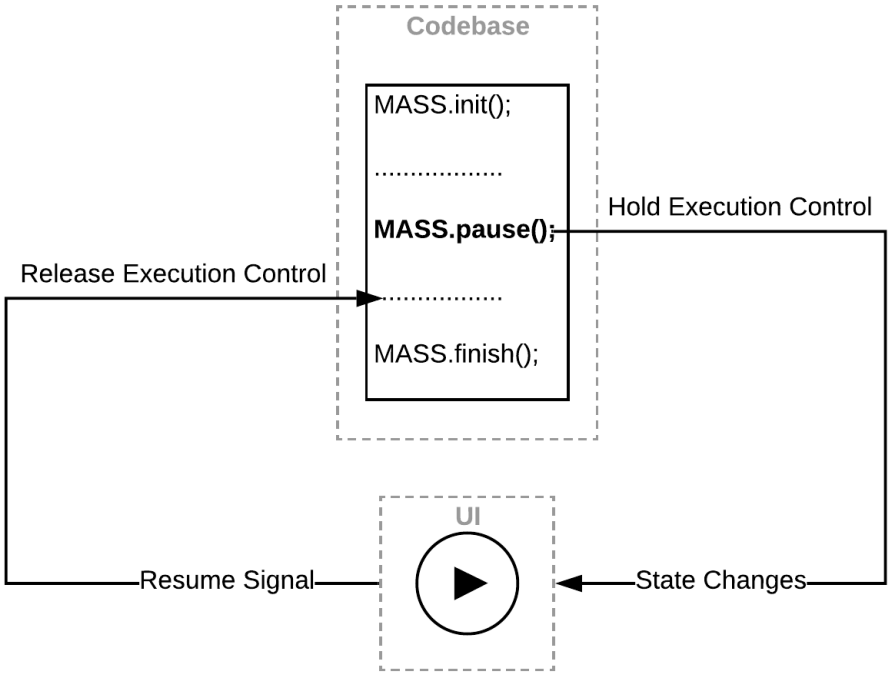


Figure 3.5: Pause Function

Chapter 4

VERIFICATION

In this section we show (I) potential speedup gain having interactive environment in ABM systems, (II) comparison of InMASS versus Repast Simphony, and (III) demonstration on how the monitoring tool and rollback can be used in MASS library.

4.1 Speedup of Software Development Process

The following experiment is designed to emulate user changes when experimenting in interactive and non-interactive environment. The objective is to measure potential development-speedup can be gained having an interactive environment. In software development, an application starts with a lot of changes, experimentation runs, at early phases of the development cycles and the magnitude of changes become less and less as the application approaches its final expectations. In this experiment, an ABM application (Agent Walker) which moves agents in a definite direction across the simulation space is used to mimic code changes that are introduced by the user. User-change is emulated by moving agents in a different direction. For instance, moving agents x steps in i dimension is a change for a previous behavior where agents were moved y steps in j dimension, where $x \neq y$ or $i \neq j$.

In reality, it is hard to mimic all possible code changes, much less their combinations. In this experiment, two types of changes are emulated: (1) environmental change and (2) behavioral change since these are the two level of abstraction that ABM libraries generally expose to the users. The former affects the simulation environment such as the number of agents and the size of the simulation space. The latter affects the behavior of entities in the simulation space such as an entity attributes, functions, or function arguments.

Interactive and non-interactive, standalone, versions of Agent Walker are compared in

three different scenarios: (1) environmental changes only, (2) behavioral changes only, (3) both environmental and behavioral changes, (4) consecutive changes. Though the first three scenarios are not measuring the effectiveness of interactivity because they capture only one run for changes, they can show the effect of particular type of change on the speedup. In contrast, the fourth scenario captures the case where the user introduces multiple changes consecutively on multiple runs. Therefore, it is more representative of real experimentation in interactive settings than the other scenarios.

For each scenario, time measurement is recorded for compilation, library initialization and execution processes as shown in Figure 4.1. Time measurement excludes tasks such as code writing, executable distribution, and log-in into remote server to run the executable. Code writing time should have no effect since it is the same for both versions. The other tasks cannot be measured accurately, and some are not applicable when running in a single mode.

Our emulation experiment can be expressed mathematically. If t_w is the code-writing time, t_c is the compilation time, t_i is the library-initialization time, t_e is the execution time, and c is the time for additional tasks such as running, distributing executable or deleting previous logs, then the time for standalone version can be express as:

$$T_S = t_w + t_c + t_i + t_e + c$$

And the time for interactive version can be express as:

$$T_I = t_w + t_e$$

Hence, the speedup:

$$S = 1 + \frac{t_c + t_i + c}{t_w + t_e}$$

In the emulation experiment, t_w and c have no effect. Therefore, the speedup can be rewritten as follows:

Type of Change	# Runs	Time Measurement (in seconds)					Difference	Speedup
		Interactive (InMASS)	Standalone (MASS)					
			Compilation	Initialization	Execution	Total		
Environmental	1	2.93	8.87	0.70	3.31	12.88	9.96	4.40
Behavioral	1	1.01	8.63	0.62	1.21	10.46	9.45	10.38
Environmental & Behavioral	1	3.10	8.35	0.75	3.39	12.49	9.39	4.03
Consecutive	2	1.76	17.96	1.23	2.67	21.86	20.10	12.44
Consecutive	4	3.68	33.99	2.27	5.26	41.52	37.84	11.29
Consecutive	8	8.21	68.74	4.51	11.25	84.49	76.28	10.29
Consecutive	16	15.23	142.74	9.27	25.04	177.05	161.82	11.63

Experiment was conducted using MacBook Pro (16-inch, 2019):
CPU: 2.3 GHz 8-Core Intel Core i9
Memory: 16 GB 2667 MHz DDR4

Figure 4.1: InMASS vs MASS: Time Difference and Speedup

$$S = 1 + \frac{t_c + t_i}{t_e}$$

There are three points worth noting about the experiment. First, the interactive version requires no explicit compiling since the compilation of the code is managed internally by JShell while standalone, non-interactive, version always requires an explicit compiling by the user. Second, initialization process is performed only once for interactive version; thus, there are no reinitialization when new changes are introduced. Users can re-create the simulation space for new experimentations or they can use rollback feature. Third, in standalone version the initialization and other additional tasks are performed for every experimentation. Additional tasks include running the executable and might include transferring executable to every computing node when running on a cluster mode, accessing remote terminal, and/or cleaning previous logs.

The result shows that the total time difference between interactive and standalone version grows linearly with the number of runs, as shown in Figure 4.2. The number of runs correspond to number of experimentations in real settings. The average speedup for interactive version across all scenario is 9.2 times the standalone version.

4.2 Qualitative Comparison between Repast Symphony and InMASS

In this section we compare Repast Simphony and InMASS in four characteristics as shown in Table 4.1. Repast S is a widely used ABM framework. It uses GUI that allows users to run, step, and visualize simulations interactively. It adopts plugin architecture which allows users to integrate custom-build plugins into the framework. Repast S depends on the IDE for plugin integration, with Eclipse as its primary IDE. On the other hand, InMASS has a smaller community than the former. InMASS is less opinionated and has no dependency on a specific IDE. InMASS uses CLI to create models and interact with MASS APIs.

Repast S requires models (Agent, Context, and Projection classes) to be ready before starting the GUI window whereas InMASS does the startup before writing models (Agent and

Figure 4.2: InMASS vs MASS: Time Difference per Consecutive Experimentations

Place extension classes). In Repast S, behaviors cannot be modified at runtime unless they are parametrized to be controlled via IDE wizard while in InMASS not only the behaviors can be modified but the models can be modified and reinitialized. Repast S simulations can only be stepped forward while InMASS can be stepped forward and backward thanks to rollback feature.

Repast S excels in state monitoring and visualization. It has dynamic charts for analytics purposes and Statecharts for state monitoring. Both are updated at runtime which gives the user analytic and state observability. As it is an opinionated framework, users have to add Statechart using the IDE wizards, set appropriate scheduling for updating the them, and add necessary annotation on the agent class. On the other hand, InMASS has UI for state monitoring whereas visualization is left to the user to implement. All needed is that the fields of interest in the agent (or place) class have to be marked with appropriate annotation for monitoring.

Table 4.1: Repast Simphony versus InMASS

	Repast Simphony	InMASS
User Interface	GUI	CLI
Computation Trackability	Forward Stepping only	Backward and Forward Stepping
State Monitoring	IDE Wizards: Dynamic charts and Statecharts	Web UI
Visualization	IDE Wizards: 2D and 3D	–

4.3 Verification of Monitoring and Rollback Tools

In this section, we demonstrate how users can select, query an application state, and interact with the Web UI component. We use the QuickStart application [12] for demonstrating the Monitoring Tool. It is MASS application that moves agents across y-axis and it is used as a starter template. Next, we demonstrate rollback feature using the Agent Walker application.

Monitoring Tool: State Selection

Place and Agent models are represented as shown in Listing 4.1 lines 3- 15 and 17- 23 respectively. Inspect annotations at line 3 and 17 indicate that the user want to watch the state of each model. At the place model, lines 6, 8, 10, and 12 demonstrate the use of Watch annotation as well as in the Agent model line 20.

Listing 4.1: QuickStart Application: Agent and Place Models

```
1
2  // place model
3  @Inspect()
4   public class Matrix extends Place {
5      @Watch()
6      public int touches = 0;
7      @Watch(key = "nullArray2")
8      public Object[] nullArray;
9      @Watch(key = "generic")
10     public Object genericObject = new Object[] {"Hi", 123, 7L};
11     @Watch(key = "private")
12     private final Object secureValue = 1234567890L;
13     public Matrix(Object obj) { ... } // place constructor
14     ... // place functions
15 }
16 // agent model
```

```
17 @Inspect()
18 public class Nomad extends Agent {
19     @Watch()
20     public int touches = 0;
21     public Nomad(Object obj) { ... } // agent constructor
22     ... // agent functions
23 }
24 // creating simulation space (Places and Agents)
25 Places places = new Places( 1, Matrix.class.getName(), null, x, y, z );
26 Agents agents = new Agents( 1, Nomad.class.getName(), null, places, x * y );
27 ... // Library Calls (Computations)
28 // explicit pause (this will hold execution, useful for non-interactive)
29 MASS.pause();
30 // exit interactive shell
31 \exit
```

The UI webpage consists of multiple views: Cluster Status, Timers, Query Dialogs, Nodes, Logging, and Queries views. Cluster Status view as shown in Figure 4.3 contains the status of last call and two lists of models. One list for places' models and the other is for agents' models. When the execution reaches Listing 4.1 line 29, the Cluster Status view will have a button to return control back to the application. When the user clicks on the timer icon (at the top-right), a pop-up view will be presented that shows calls stack each with a time elapsed as shown in Figure 4.4. When the user click on the search icon (at the right side for each model), a pop-up view will be presented as shown in Figure 4.5.

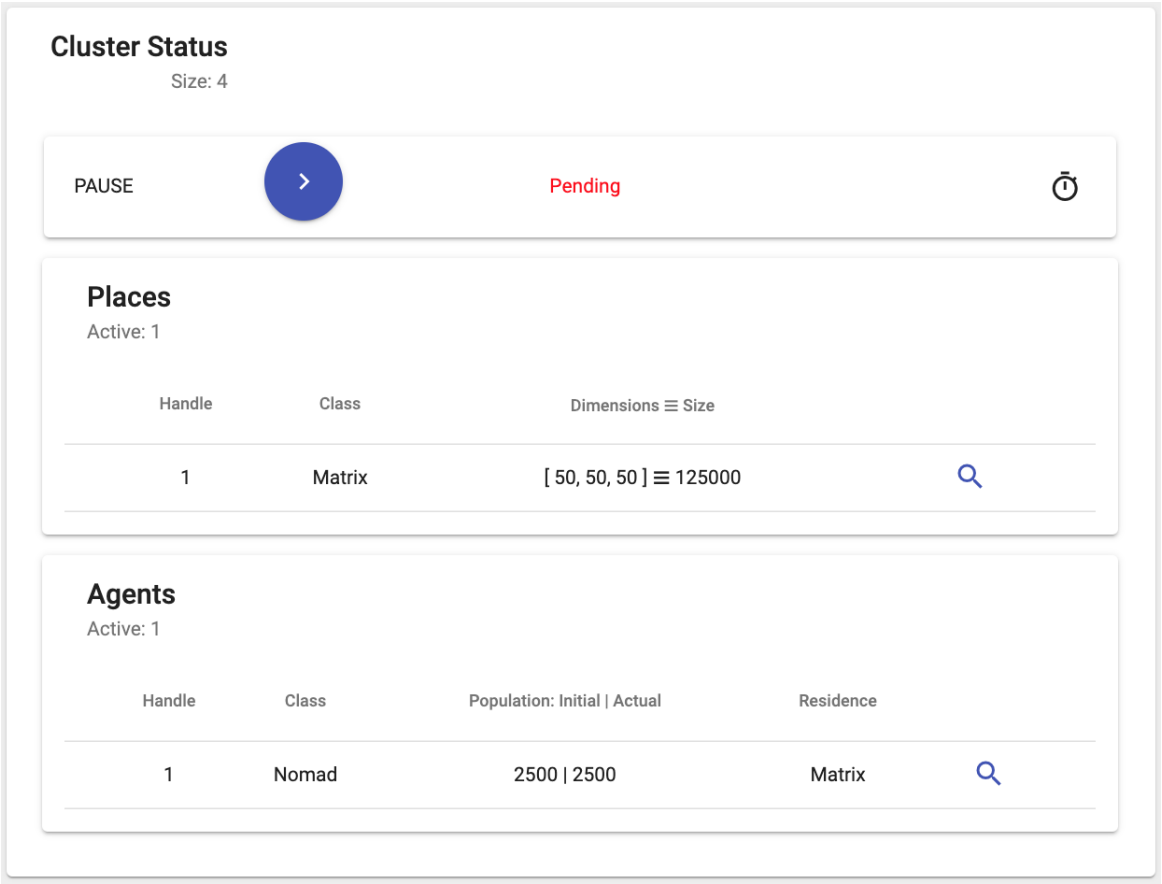


Figure 4.3: Cluster Status View: Places and Agents Representation

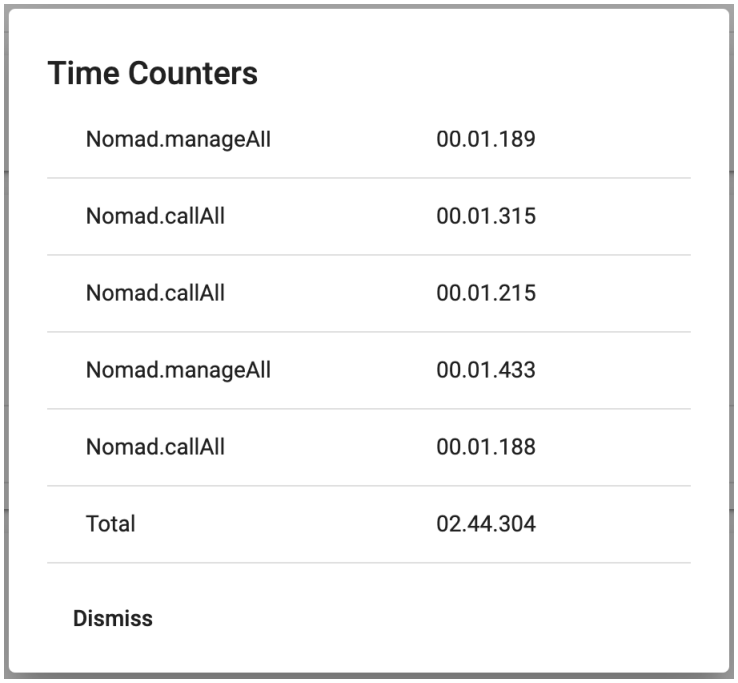


Figure 4.4: Timers View

The Query Dialog has two variations, Figure 4.5.. One is for querying from place models and the second is for agent models. Place query dialog allows dimensional or linear index for querying a place while the Agent query dialog except linear id only.

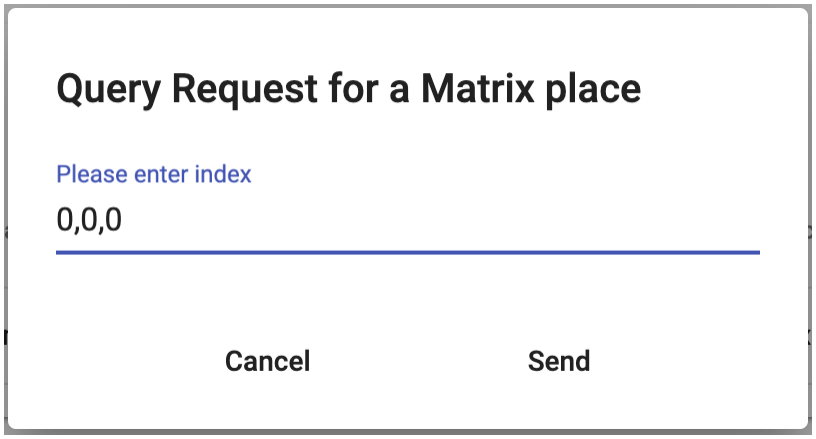


Figure 4.5: Place Query Dialog

Nodes view illustrates places mapping per each node as well as the agent population as shown in Figure 4.6. Also, it has a status view that shows the last function call, similar to the Cluster Status view. Each node is labeled whether it is a master or worker node.

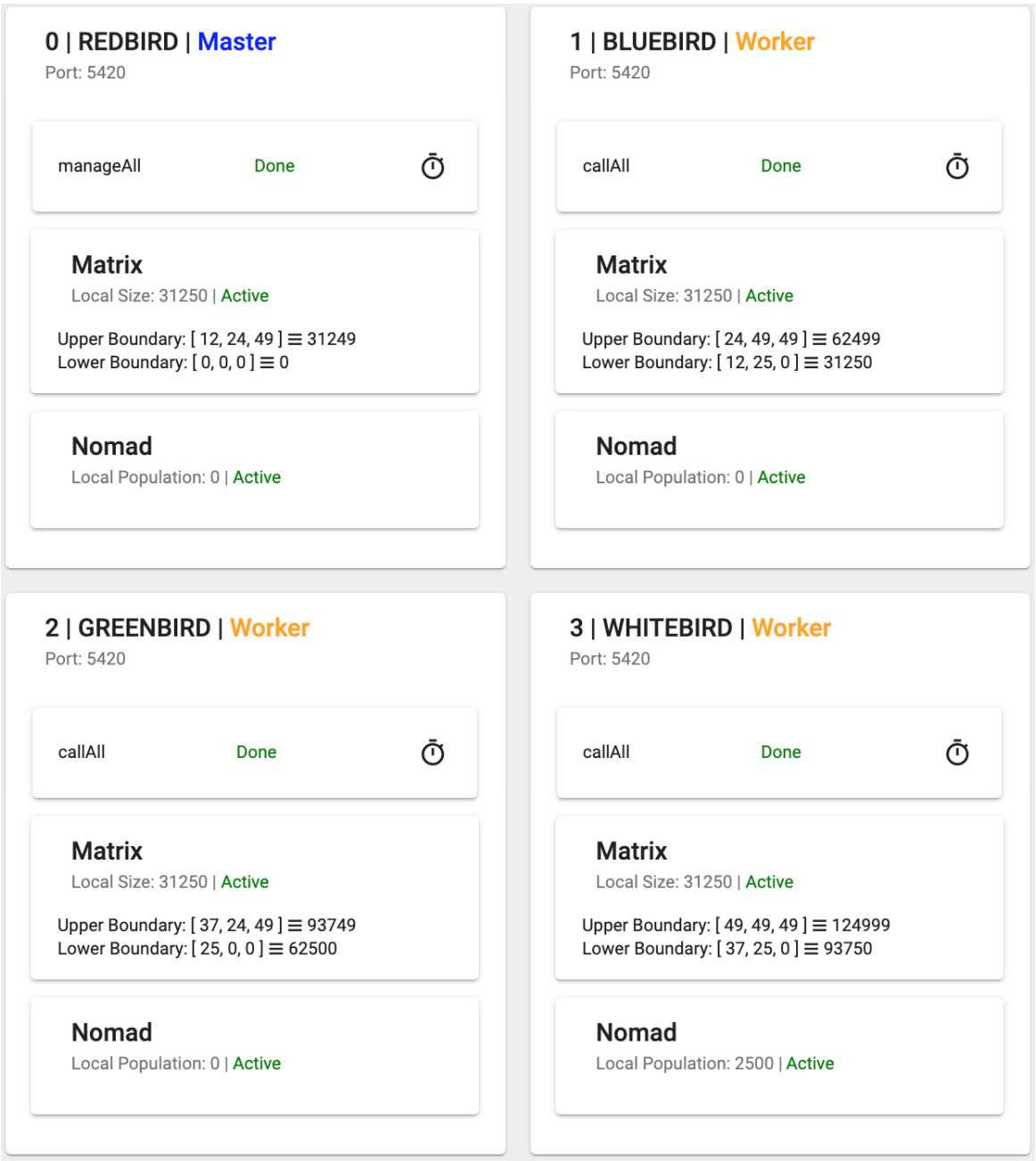


Figure 4.6: Nodes View

Figure 4.7 shows the Logging view. All error pipes from the MASS daemon processes are collected in this window.

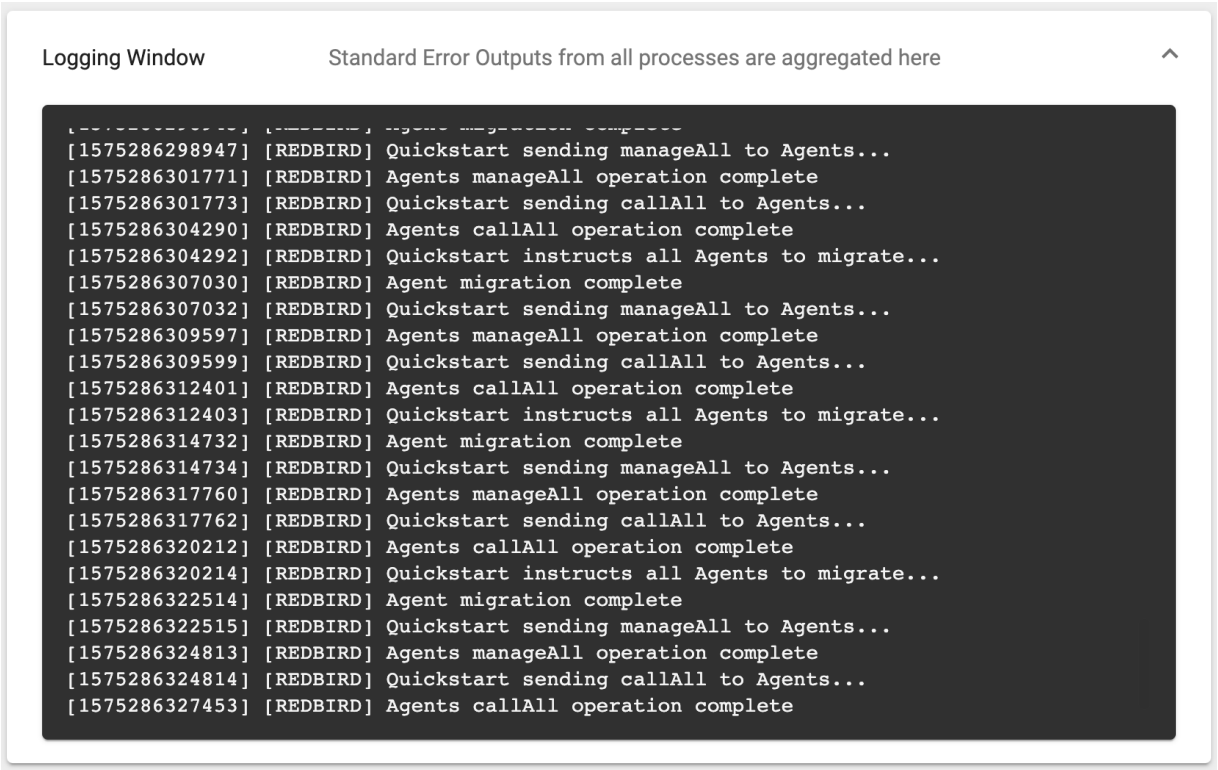


Figure 4.7: Logging View

When a place or an agent model is fetched from the monitoring process, the UI will add a new view (we call it Query view) that shows query data. Figure 4.8 illustrates a place query view. It has three tabs: inspection, neighbors, and occupants. The annotated state can be inspected at the inspection tab as shown in Figure 4.9. The neighbor places can be inspected at the neighbors' tab as shown in Figure 4.10.

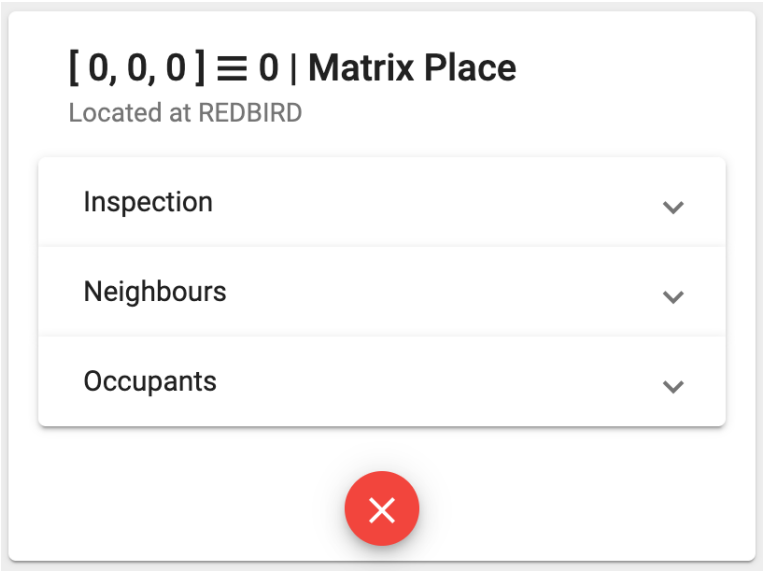


Figure 4.8: Place View

All selected fields in the model will be collected and shown under the inspection tab, illustrated in Figure 4.9.

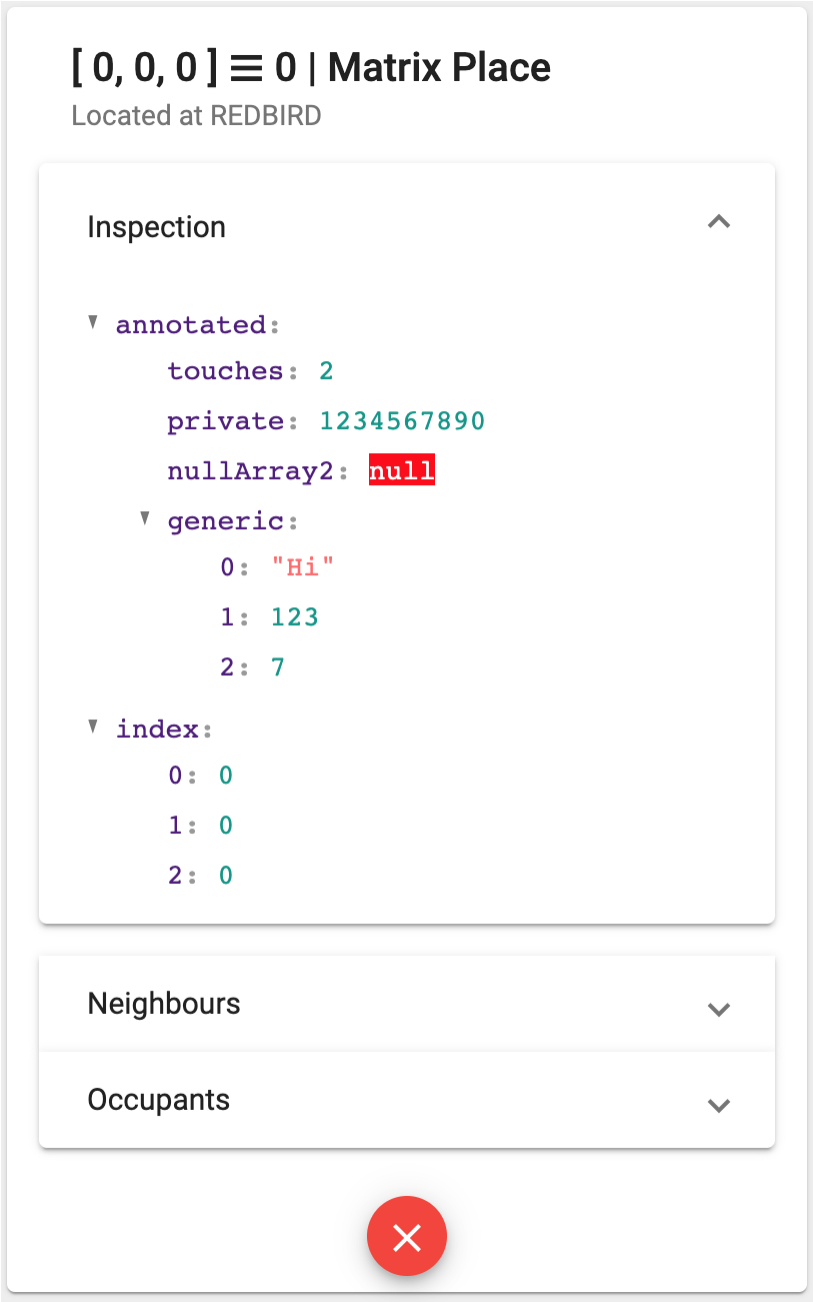


Figure 4.9: Place View: Inspection Tab Expanded

In neighbors tab Figure 4.10, the user can navigate to a neighbor by clicking on its index. The UI will add new view for the neighbor if it is not queried yet.

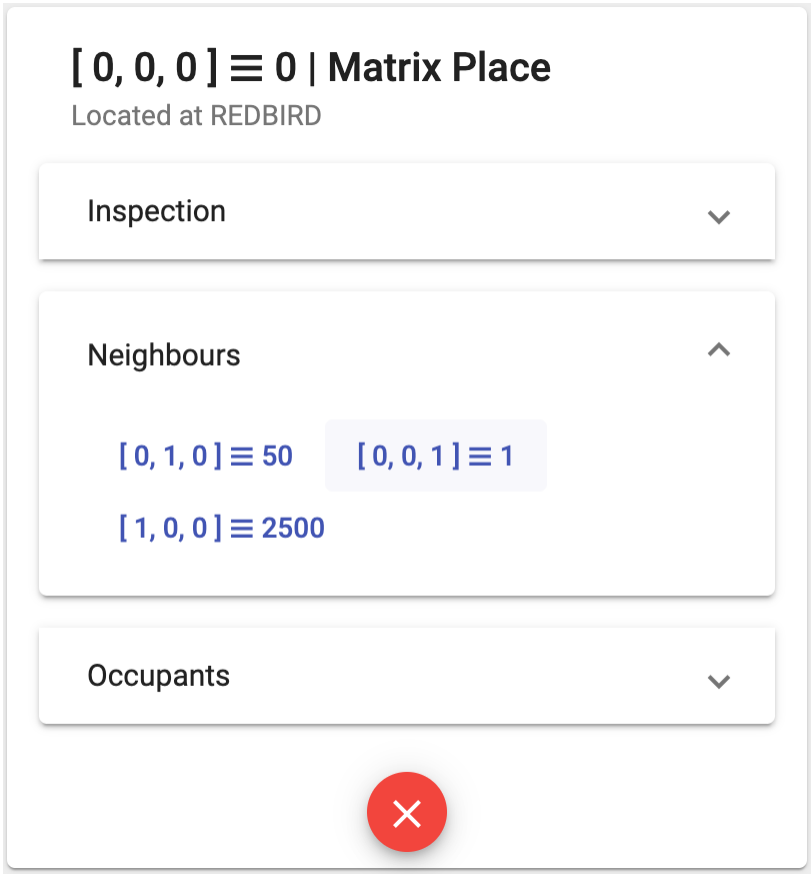


Figure 4.10: Place View: Neighbors Tab Expanded

Rollback Feature

For simplicity we use the following example to demonstrate how users can use rollback feature. Agent Walker is a straightforward application that moves agents into definite direction. In Listing 4.2, we create four agents each in its own place. Then, we checkpoint the state as shown in line 11. This will serialize the current state of the application into the memory or disk. Next, agents are move four steps line 13. This will change the state by moving agents to the last place. Finally, the state is rolled back using rollback function line 18. All agents

should return to their original places.

Users can specify where to store the snapshots using *StoreInDisk()* for storing in temporary files, *StoreInFile()* for storing in explicit file, or *StoreInMemory()* for storing in the memeory. History of previous computations, library calls, can be listed using *History()* function. Users can jump to specific step in history by calling *Rollback()* function with the number of step as argument, as shown in line 19.

Listing 4.2: Agent Walker: Rollback

```
1  // Agent model
2  public class Walker extends Agent {
3      public Walker() { ... }
4      // migration function
5      public void move(int... dir) { ... } }
6  // Creating 4 agents with 4 places using builder pattern
7  Agents<Walker> walkers = AgentsBuilder.builder(Walker.class, 4)
8                                  .homeSize(4).handles(0,0).build();
9  // Check UI, each agent will reside in its own place (1:1 mapping)
10 // Checkpoint the state
11 Checkpoint();
12 // moving agents (this will moves all agents to the last place = place[3])
13 walkers.doAll( a -> { a.move(1); }, 4);
14 // Check UI, all agent should be moved into the last place
15 // List previous steps (function calls)
16 History();
17 // restore the state to checkpoint
18 Rollback();
19 // Rollback(0);
20 // Check UI, all agents must return to their original places
```

Discussion

The emulation experiment measures how much time difference between experimentation in interactive and non-interactive versions of MASS library. That does not mean the interactive version (InMASS) is more performant than non-intearctive (MASS). Actually, InMASS has compromise on performance because of the addtional computations needed for collecting the state, compiling user-code on the fly, maintaining *UI* connections. Also, it adds addtional usage to the network for serving *UI* content and updates. Therefore, our solution is not suitable for performant and long-running applications. It is useful when users want to exploring their ideas in ABM settings.

Chapter 5
CONCLUSION

In ABM systems, interactivity can speed up development and prototyping relative to the number of experimentational runs. In this paper we emulate consecutive-experimentational changes on both interactive and non-interactive versions of the MASS library, interactive version performs 9.2 times faster than non-interactive version.

Interactive computing in ABM settings necessitates observability tools. In this paper we present architecture for state monitoring and computation backtrackability. We demonstrate the architecture by implementing a montoring tool that works on both interactive and non-interactive versions of the MASS library. Also, we introduce computation backtrackability support for the InMASS (interactive version of the library) using checkpointing and rollback-recovery technique. Despite the lack of analysis and visualization tools in the MASS library, the state can be observed at finer granularity and less opinionated manner than Repast Simphony. Moreover, our solution allows users not only to advance the computation of applications/simulations incrementally but it can roll back the state to a previous point in computations history.

Limitations

Users might find the Command-Line Interface (CLI) counterintuitive for editing the code. However, there are two other options for editing the code when using JShell: (1) using *edit* and/or (2) *open* commands. The *edit* command will open text editor of user's choice while *open* command uses pre-written code from text files. We recommend using *edit* command when writing extention classes, CLI when calling the APIs. If the user has code in text files, the *open* command is preferable. Overall, JShell has its own learning curve. Once users learn

how to use its command and shortcuts, they can navigate and edit their code easily.

The monitoring architecture puts the responsabiltiy of initiating and managing multiple connections at the front-end teir. The number of WebSocket connections is limited by the browser implementation used at the client device. Also, connections management for multipe computing nodes requires addtional CPU and network usage. Thus, the user experience will be degraded when monitoring on cluster, especially for low-end devices.

Future Work

Rollback feature can be used to add fault-tolerance capability. In our design, state checkpoint and rollback are user-driven actions. Adding fault-tolerance capability requires periodic checkpointing and it should automatically roll back to the last preserved state when the system crashes.

In addition, monitoring architecture provides querying mechanism. Next steps would be developing visualization and data analysis tools by interfacing with the monitoring process. However, our implementation is limited to WebSocket protocol for communication and JSON for message representation. The architecture can be modified to accommodate various type of communication protocols such as the TCP/IP protocol.

BIBLIOGRAPHY

[1] Java reflection docs. `https://docs.oracle.com/javase/9/docs/api/java/lang/reflect/package-summary.html`.

[2] Angular. Introduction to the angular docs. `https://angular.io/docs`.

[3] Timothy Chuang and Munehiro Fukuda. A parallel multi-agent spatial simulation environment for cluster systems. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 143–150. IEEE, 2013.

[4] George Coulouris. *Distributed systems: concepts and design*. Addison-Wesley, 2012.

[5] Eclipse. Eclipse vert.x is a tool-kit for building reactive applications on the jvm. `https://vertx.io`.

[6] Robert Field. Jshell: An interactive shell for the java platform, 2017. Available at `https://my-conference.ml/talks/2350`.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] Łukasz Kufel. Tools for Distributed Systems Monitoring. *Foundations of Computing and Decision Sciences*, 41(4):237–260, November 2016.

[9] C. M. Macal. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation*, 10(2):144–156, May 2016.

[10] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[11] Alexey Melnikov and Ian Fette. The WebSocket Protocol. RFC 6455, December 2011.

[12] University of Washington Bothell. Mass java: Application. `https://bitbucket.org/mass_application_developers/mass_java_appl/src/master/`, 2018.

[13] University of Washington Bothell. Mass: Interactive computing feature. https://bitbucket.org/mass_library_developers/mass_java_core/src/interactive-feature/, 2019.

[14] Oracle. *Java Platform, Standard Edition Java Shell User's Guide*. Oracle, 2018. Available at https://docs.oracle.com/javase/10/jshell.

[15] Jonathan Ozik, Nicholson Collier, Todd Combs, Charles M. Macal, and Michael North. Repast simphony statecharts. *Journal of Artificial Societies and Social Simulation*, 18(3):11, 2015.

[16] Niko Simonson, Sean Wessels, and Munehiro Fukuda. Language and Debugging Support for Multi-Agent and Spatial Simulation. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 373–379, Las Vegas, July 2012.

[17] TypeScript. Documentation. https://www.typescriptlang.org/docs/home.html.

[18] University of Washington. *MASS Java Developers Guide*, 2016. Available at https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Developer%20Guide.pdf.

[19] University of Washington. *MASS Java Manual*, 2016. Available at https://depts.washington.edu/dslab/MASS/docs/MASS%20Java%20Technical%20Manual.pdf.

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

Appendix A

DEVELOPER GUIDE

In this section we provide implementation detailes that can help developers to understand relation between the architecture components and their modules in the code.

The tool is implemented for MASS Java version. We consider three independent processes when implemented the tool as shown in Figure A.1. One is the MASS/MProcess which represents the MASS daemon process. Second is the monitoring process which manages data and the querying mechanism. The other is the UI client which is a browser process that executes the JavaScript code.

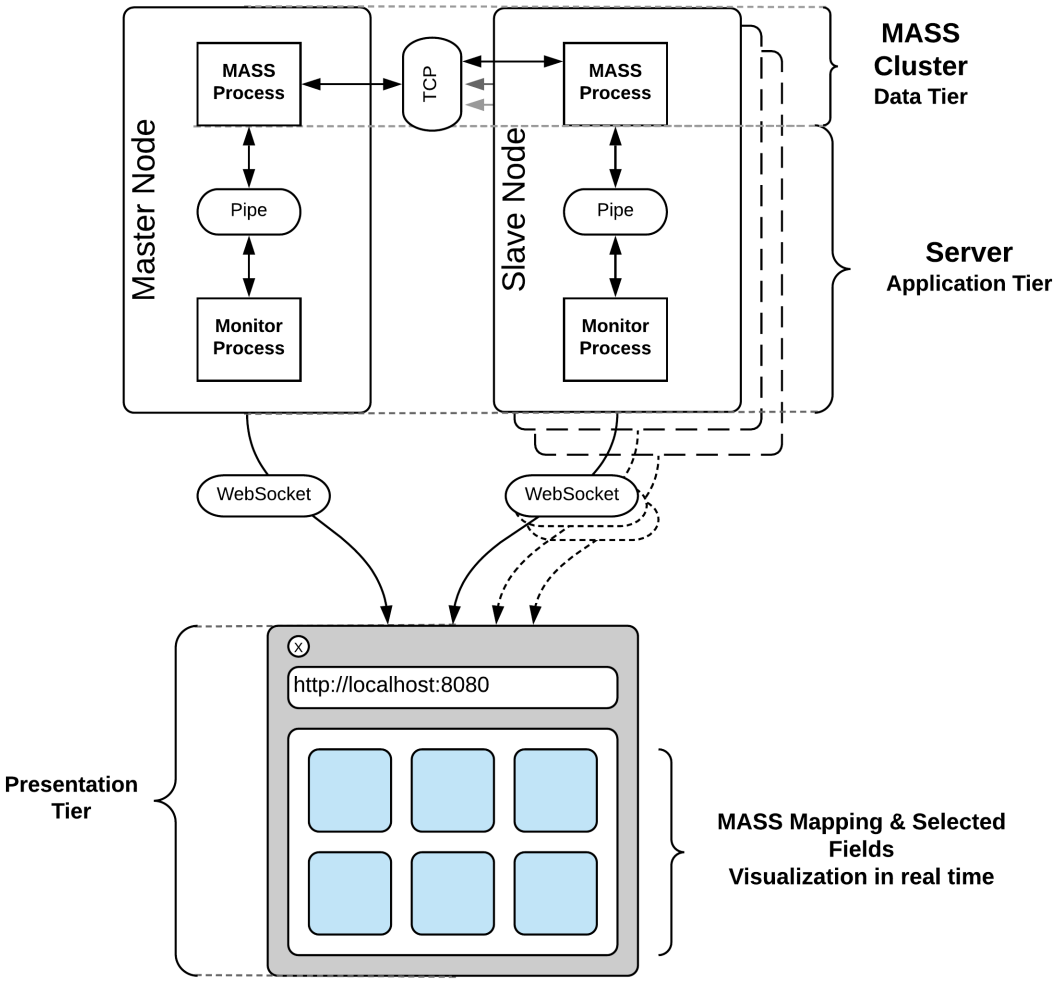


Figure A.1: Implementation View

We added code for probing the application state and the library calls. The probing does not conflict with MASS execution and it is a synchronized logic that occurs before and after most of the exposed library functions.

The probing code is written in Java as well as the Monitor process. While the UI is a combination of HTML, JavaScript, and CSS that is executed by a browser.

The interactive feature code is available on BitBucket [13]. Implementation code for the Monitoring Tool resides under the following path:

```
// Monitoring Tool
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring
```

Handlers and MonitorConnector Classes

The probing code consists of a set of classes that are called *Handlers*. A Handler class is responsible for transforming MASS component into JSON representation. Handlers collectively produce a one-line string (a JSON object) that represents the node (MASS/MProcess) state. The tool has five handlers: MASSHandler, PlacesHandler, AgentsHandler, PlaceHandler, and AgentHandler.

MASSHandler, PlacesHandler, and AgentsHandler collects metrics for MASS, Places, and Agents, respectively. While PlaceHandler and AgentHandler collect metrics and the state for Place and Agent, respectively.

Also, it has MonitorConnector and MsgUti classes which are responsible for interfacing with the monitoring process. The implementation codes for the Handlers, MonitorConnector, and MsgUti classes reside under the following paths:

```
// Handlers classes
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/handlers

// MonitorConnector class
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/MonitorConnector.java

// MsgUti class
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/MsgUti.java
```

Monitoring Process

The monitor process consists of two types of classes. The first type is classes that manage the data: Repository and Resources classes. Repository forwards requests (read or write) to the appropriate Resources instance. Resource handle requests such as updating the data (write) and fetching the data (read). Also, it manages client subscriptions on the data.

The second type is classes that manage the monitor process: Launcher, HTTPServer, WebSocketServer, WebSocketHandler, and RequestHandler class. The Launcher is an entry point of the monitoring process which keeps listening for MASS/MProcess messages. The HTTPServer manages an HTTP/WebSocket server. The WebSocketServer manages a WebSocket server only. The WebSocketHandler handles WebSocket events such as opening and closing client connection. The RequestHandler handles client messages that are sent via the WebSocket protocol.

The implementation code for the monitoring process resides on the following paths:

```
// Data classes
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/repository

// Management classes
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/server
```

Implementation codes for annotations and reflections resides under the following path:

```
// Annotations
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/annotations

// Reflection
src/main/java/edu/uw/bothell/css/dsl/MASS/monitoring/serializers
```

Monitoring Messages

Monitoring messages take place between the Node component and the monitoring process. A monitoring message is a string that starts with one of the predefined headers and might contain data. Headers are defined via a configuration file that is used by both processes. The header helps the monitoring process deciding which action to take.

A monitoring message might contain data associated with a given header. While some messages might have no header which are considered to be logging or error messages. Also, some might have only a header which are considered to be signal messages: pause and resume signals. Nine headers are considered to define all monitoring messages types.

Six headers are reserved for signals: (1) one is for starting an HTTP with WebSocket server, (2) another is for starting a WebSocket server only, (3) one is for an acknowledgment signal, two for (4) resume and (5) pause signals, (6) the last one is to stop the server. The other three headers are reserved for: (7) hosts, (8) data, and (9) status messages.

Querying Mechanism

After the *Handlers* produce a local state representation, it will be streamed to the monitoring process. The application state is represented in a large JSON object which might be scattered across multiple monitoring processes. Collecting such an object in the UI component (the browser process) is an inefficient solution and it might be invisible for applications that have a large state.

We provide a querying mechanism to avoid the memory bottleneck at the UI component. It tackles this issue by querying the necessary data for the UI component rather than collecting the whole state. As illustrated in Figure A.2, a querying message has a field called query. This field is used to specify a small set or even a single field from the state.

For instance, the query field can be used to query data for a specify Agent. All needed is to provide the Agents' handle and the id for the specified Agent. To make this example concrete, let us assume the Agents handle is 1234 and the Agent id is 5678. The query field

will be set to 1234.5678. The querying message can be constructed as follows:

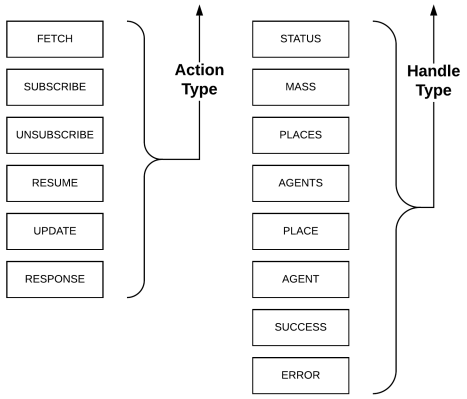
```
// This will fetch the agent data only once
{ action: "FETCH", handle: "AGENT", query: "1234.5678" }

// This will fetch the agent data once and every time it is changed
{ action: "SUBSCRIBE", handle: "AGENT", query: "1234.5678" }
```

UI Code

We used Angular framework for the developing the UI component. Front-end developers can pick any framework for implementing the UI component as long as the binding to the querying mechanism is respected. The output of Angular framework is HTML, JavaScript, and CSS files. The monitoring process must have these files in order to serve them via an HTTP protocol thanks to Vert.x library. We designated **webroot** folder (inside the resources folder) to hold these files.

Query Message Structure:

```
{   "action": "ACTION",  "handle": "HANDLE", "query": "text", "msg": {}  }
```



Examples:

Request:
```
{ "action": "FETCH",     "handle": "STATUS",  "query": null, "msg": null   }
```
Request:
```
{ "action": "SUBSCRIBE", "handle": "PLACES",  "query": "id", "msg": null   }
```
Response:
```
{ "action": "RESPONSE",  "handle": "SUCCESS", "query": "id", "msg": {...}  }
```
Response:
```
{ "action": "UPDATE",    "handle": "SUCCESS", "query": "id", "msg": {...}  }
```

Figure A.2: Query Message Structure

Appendix B

VERIFICATION CODE

B.1 Complete Code for Monitoring Demo

The following is the complete interactive code for monitoring

Listing B.1: Complete Code for Monitoring Demo: QuickStart Application

```
@Inspect()
public class Matrix extends Place {

public static final int GET_HOSTNAME = 0;

@Watch()
public int touches = 0;

@Watch(key = "nullArray2")
public Object[] nullArray;

@Watch(key = "generic")
public Object genericObject = new Object[] {"Hi", 123, 7L};

@Watch(key = "private")
private final Object secureValue = 1234567890L;

public Matrix(Object obj) {
touches++;
Vector<int[]> placeNeighbors = new Vector<int[]>();
```

```
21      placeNeighbors.add( new int[] { 0, -1, 0 } );
22      placeNeighbors.add( new int[] { 0, 1, 0 } );
23      placeNeighbors.add( new int[] { 0, 0, 1 } );
24      placeNeighbors.add( new int[] { 0, 0, -1 } );
25      placeNeighbors.add( new int[] { 1, 0, 0 } );
26      placeNeighbors.add( new int[] { -1, 0, 0 } );
27      setNeighbors( placeNeighbors );
28      }
29
30      public Object callMethod(int method, Object o) {
31      touches++;
32      switch (method) {
33
34      case GET_HOSTNAME:
35      return findHostName(o);
36
37      default:
38      return new String("Unknown Method Number: " + method);
39      }
40      }
41
42      public Object findHostName(Object o){
43      String result;
44      try{
45      result = (String) "Place located at: "
46      + InetAddress.getLocalHost().getCanonicalHostName()
47      +" " + Integer.toString(getIndex()[0])
48      + ":" + Integer.toString(getIndex()[1])
49      + ":" + Integer.toString(getIndex()[2]);
```

```
50      }catch (Exception e) {
51      result = "Error : " + e.getLocalizedMessage() + e.getStackTrace();
52      }
53      return result;
54      }
55      }
56
57      @Inspect()
58      public class Nomad extends Agent {
59      public static final int GET_HOSTNAME = 0;
60      public static final int MIGRATE = 1;
61      public static final int MIGRATE_REVERSE = 2;
62      public static final int MIGRATE_RANDOM = 3;
63      public static final int KILL_RANDOM = 4;
64      public static final int KILL = 5;
65
66      @Watch()
67      public int touches = 0;
68
69      Random generator;
70
71      public Nomad(Object obj) {
72      this.touches++;
73      this.generator = new Random();
74      }
75
76      public Object callMethod(int method, Object o) {
77      this.touches++;
78      switch (method) {
```

```
79
80      case GET_HOSTNAME:
81      return findHostName(o);
82
83      case MIGRATE:
84      return move(o);
85
86      case MIGRATE_REVERSE:
87      return moveBack(o);
88
89      case MIGRATE_RANDOM:
90      return randomMove(o);
91
92      case KILL_RANDOM:
93      return this.randomKill(o);
94
95      case KILL:
96      return this.kill(o);
97
98      default:
99      return new String("Unknown Method Number: " + method);
100     }
101     }
102
103     public Object findHostName(Object o){
104     String result;
105     try{
106     result = (String) "Agent located at: "
107     + InetAddress.getLocalHost().getCanonicalHostName()
```

```
108     + " " + Integer.toString(getIndex()[0])
109     + ":" + Integer.toString(getIndex()[1])
110     + ":" + Integer.toString(getIndex()[2]);
111     }catch(Exception e) {
112     result = "Error : " + e.getLocalizedMessage() + e.getStackTrace();
113     }
114     return result;
115     }
116
117     public Object move(Object o) {
118     int xModifier = this.getPlace().getIndex()[0];
119     int yModifier = this.getPlace().getIndex()[1];
120     int zModifier = this.getPlace().getIndex()[2];
121
122     xModifier++;
123     migrate(xModifier, yModifier, zModifier);
124     return o;
125     }
126
127     public Object moveBack(Object o) {
128     int xModifier = this.getPlace().getIndex()[0];
129     int yModifier = this.getPlace().getIndex()[1];
130     int zModifier = this.getPlace().getIndex()[2];
131
132     xModifier--;
133     migrate(xModifier, yModifier, zModifier);
134     return o;
135     }
136
```

```
137     public Object randomMove(Object o) {
138     int[] randomDest =
139     IntStream.of(this.getPlace().getSize())
140     .map(x -> this.generator.nextInt(x)).toArray();
141     migrate(randomDest);
142     return o;
143     }
144
145     public Object randomKill(Object o) {
146     if(generator.nextFloat() < 0.5)
147     this.kill();
148     return o;
149     }
150
151     public Object kill(Object o) {
152     this.kill();
153     return o;
154     }
155     }
156
157     int x = 50;
158     int y = 50;
159     int z = 50;
160
161     System.err.println( "Quickstart creating Places..." );
162     Places places = new Places( 1, Matrix.class.getName(),
163     ( Object ) new Integer( 0 ), x, y, z );
164     System.err.println( "Places created" );
165
```

```
166     Object[] placeCallAllObjs = new Object[ x * y * z];
167     System.err.println( "Quickstart sending callAll to Places..." );
168     Object[] calledPlacesResults = ( Object[] )
169     places.callAll( Matrix.GET_HOSTNAME, placeCallAllObjs );
170     System.err.println( "Places callAll operation complete" );
171
172     System.err.println( "Quickstart creating Agents..." );
173     Agents agents = new Agents( 1, Nomad.class.getName(), null, places, x * y );
174     System.err.println( "Agents created" );
175
176     Object[] agentsCallAllObjs = new Object[ x * y ];
177     System.err.println( "Quickstart sending callAll to Agents..." );
178     Object[] calledAgentsResults = ( Object[] )
179     agents.callAll( Nomad.GET_HOSTNAME, agentsCallAllObjs );
180     System.err.println( "Agents callAll operation complete" );
181
182     // can be used instead of the loop function
183     // calledAgentsResults = ( Object[] )
184     // agents.doAll( Nomad.MIGRATE, agentsCallAllObjs, z );
185
186     for (int i = 0; i < z; i ++) {
187
188     // tell Agents to move
189     System.err.println( "Quickstart instructs all Agents to migrate..." );
190     agents.callAll(Nomad.MIGRATE);
191     System.err.println( "Agent migration complete" );
192
193     // sync all Agent status
194     System.err.println( "Quickstart sending manageAll to Agents..." );
```

```
agents.manageAll();
System.err.println( "Agents manageAll operation complete" );

}

MASS.pause();

// return all agents back to where they start
// calledAgentsResults = ( Object[] )
// agents.doAll( Nomad.MIGRATE_REVERSE, agentsCallAllObjs, z );

System.err.println(
"Quickstart sending callAll to Agents to get final landing spot..." );
calledAgentsResults = ( Object[] )
agents.callAll(Nomad.GET_HOSTNAME, agentsCallAllObjs );
System.err.println( "Agents callAll operation complete" );
```

B.2 Complete Code for Rollback Demo

The following is the complete interactive code for rollback verification:

Listing B.2: Agent Walker: Rollback Complete Code

```
// Agent model
public class Walker extends Agent {

    public Walker() {}

    // migration function
    public void move(int... dir) {
        if(dir.length != this.getIndex().length) return;
```

```
9            migrate(IntStream.range(0, dir.length)
10               .map(i -> dir[i] + getPlace()
11               .getIndex()[i])
12               .toArray());
13        }
14
15    }
16
17    // Creating 4 agents with 4 places using builder pattern
18    Agents<Walker> walkers = AgentsBuilder.builder(Walker.class, 4)
19                                    .homeSize(4).handles(0,0).build();
20    // Check UI, each agent will reside in its own place (1:1 mapping)
21
22    // Checkpoint the state
23    Checkpoint();
24
25    // moving agents (this will moves all agents to the last place = place[3])
26    walkers.doAll( a -> { a.move(1); }, 4);
27
28    // Check UI, all agent should be moved into the last place
29
30    // List previous steps (function calls)
31    History();
32
33    // restore the state to checkpoint
34    Rollback();
35    // Rollback(0);
36
37    // Check UI, all agents must return to their original places
```