Data Provenance for Multi-Agent Models in a Distributed Memory

Delmar Bryan Davis II

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science and Software Engineering

University of Washington

2017

Committee:

Hazeline U. Asuncion, Chair

Munehiro Fukuda

Michael Stiber

Program Authorized to Offer Degree:

Computing & Software Systems

University of Washington

Abstract

Data Provenance for Multi-Agent Models in a Distributed Memory

Delmar B. Davis II

Chair of the Supervisory Committee:
Associate Professor Hazeline U. Asuncion, Ph.D.
Computing & Software Systems

Multi-agent systems (MAS) assist with studying emergent collective behavior of individual entities in social, biological, economic, network, and physical systems. Applied to MAS, data provenance can support agent-based modeling by explaining individual agent behavior. However, there is no provenance support for MAS in a distributed setting. The Multi-Agent Spatial Simulation (MASS) library provides a framework for simulating agent-based models (ABM) at fine granularity, as multi-agent models (MAM), where agents and spatial data are shared application resources in a distributed memory. This Master's thesis evaluates ProvMASS, a novel approach to capture MAM provenance in a distributed memory. Queries and performance measures indicate that an adaptive approach can generate provenance that explains coordination of distributed shared resources, simulation logic, and agent behavior while

limiting performance overhead for long-running simulation (several hours) of large models (including hundreds of thousands of agents).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am profoundly grateful to my advisor, Prof. Hazeline U. Asuncion, for outstanding mentorship, collaboration and support. She supported my entrance to the graduate program, helped fund my education, and guided me on a journey from programmer to research leader. I would not have pursued this degree without the chance meeting, and subsequent discussion, that led to my undergraduate capstone research. I thank God for placing us in each other's paths and for perseverance in my studies and our research.

I would also like to thank my committee member, Prof. Munehiro Fukuda, for enlightening me with patient explanation of intricacies in distributed and parallel computing; without which, I would not have pursued this topic; and Prof. Michael Stiber, for stretching my thinking with discussion about models, system-state and the contexts in which researchers use software.

I would like to show appreciation to the Software and Traceability Research Group for conversations and input into various software development endeavors. Every project that I have supervised has provided insight and the continued evolution of these projects was made possible by the hard work of my fellow students. I would especially like to acknowledge the help of Jonathan Featherston, Jason Woodring, Morteza Chini and Wei Xu for working closely with me on ProvMASS and related research. The following students' collaboration also helped to shape my understanding of provenance and traceability through related projects: Mohammed Daubal, Namita Dave, Nathan Duncan, Kriti Gupta, Qi Zhang, Sadia Suhail, and Thomas Helms.

This work is based in part upon work supported by the US National Science Foundation under Grant No. ACI 1350724 and the UW Bothell CSS Graduate Research fund.

DEDICATION

To Amy… your patience and loving support sustain me.

To Dave… the provided perspective, means and confidence aided in completing this degree.

Chapter 1. INTRODUCTION

Data provenance describes the creation, manipulation and consumption of data. The suitability of a provenance capture technique is strongly influenced by the context in which data operations are executed. For example, consistent identification of distributed data (e.g., data copied from one computer to another via network system calls) is necessary to reason about data operations that span multiple locations. Techniques specific to capturing provenance of shared resources in distributed memory have yet to be discussed in provenance literature. Meanwhile, there are few discussions about data provenance for agent-based models (ABM), and none that address distributed, parallel, or distributed-parallel execution contexts. This is unfortunate, as it has been noted that provenance can be used to explain individual agent behavior [7], which agent-based models should strive to focus on [2].

Multi-agent systems (MAS) are software systems used to simulate phenomena in real-world systems by coordinating interactions between entities of fine-grain ABMs, referred to as multi-agent models (MAM). Performance requirements for MAS are intensified by the large number of agents (ranging in orders of thousands to millions) necessary to compose these fine-grain systems. Distributed and parallel computing are viable means to increase simulation performance, if orchestrated to enable unimpaired agent interaction; such as in a distributed memory. Consequently, there is a need for new provenance techniques to bridge the gap between automatic parallelization and model understanding.

This thesis details and evaluates ProvMASS, a novel approach to capture provenance of fine-grain ABMs as multi-agent models (MAM) in a distributed memory. This approach is evaluated with queries to demonstrate the ability to (1) *understand simulation logic with respect to execution*,

(2) *explain agent behavior*, and (3) *describe coordination of distributed data*. Each support scenario is also evaluated with performance measures. This thesis shows that an adaptive approach enables support for each scenario while limiting overhead.

1.1 Multi-Agent Models in MASS

Agent-based models (ABM) describe systems of entities that interact with each other and their environment [17]. Multi-agent systems (MAS) allow researchers to implement ABMs consisting of many fine-grain, coordinated agents. This thesis refers to these types of ABMs as multi-agent models (MAM). Fine-grain coordination makes MAMs well suited to investigate emergent collective behavior of systems composed of individual interactions, but such a focus may detract from understanding intermediate phenomena. Provenance can provide an understanding of intermediate phenomena with details of individual agent interactions, which combine to form agent behaviors.

In many ABM implementations, interactions occur within procedure calls, which occur more frequently in MAM implementations. Consequently, provenance capture for MAMs naturally imposes greater simulation overhead. Provenance in NetLogo [19] captures provenance for sequential MAM implementations. However, operations on fine-grain agents can be processing intensive, especially in large-scale simulations. Consequently, a key goal of successfully simulating MAMs lies in increasing agent scale while maintaining performance. The Multi-Agent Spatial Simulation (MASS) library distributes MAM resources (e.g., entities and spatial data) over the memory of a cluster of computers to achieve scale, while parallelizing operations over resources through shared memory programming.

MASS handles coordination of agents and spatial data, allowing the developer to focus on model design. A byproduct of this feature is split abstraction of modeled resources. Agent

interactions are still dependent on framework operations, but are of interest to the developer in the context of the application logic. Bridging the two representations is an important consideration for capturing provenance of MAMs. However, more fundamental problems hinge on generating coherent provenance data within the context of MASS framework execution. While techniques to capture provenance in related execution contexts have been discussed, a new approach is required to capture provenance of shared resources in a distributed memory.

1.2 Distributed and Parallel Provenance Capture

Many provenance capture techniques handle execution properties of distributed and parallel computing. While, these properties are included in handling distributed shared resources, assumptions about data representation and use are further constricted. Consequently, application of parts of these techniques to capture MAM provenance is impossible. In some cases, however, they may be integrated with slight modification. Meanwhile, others may require wholesale replacement. Still, existing techniques originating from these execution contexts illuminate some of the technical challenges associated with capturing distributed memory operations.

Common to distributed computing scenarios is the ability to send and receive data between processes. Operations that span physical resources can be reasoned about with consistent identification of transmitted data. In other words, one must trace data generation from one machine to data use on another. Provenance capture for distributed computing has been discussed for grid and data-intensive scalable computing systems, but concurrent tasks are assumed to be independent. Techniques also exist for capturing provenance for distributed file systems and network system calls, but focus on data at coarse granularity or at a single level of abstraction (i.e. at the file-level).

Shared memory programming enables concurrent data access between threads of execution. Consequently, related provenance techniques must differentiate procedure invocations. Techniques for capturing memory operations in this context represent data with respect to memory locations and require explicit synchronization mechanisms. An approach to capture distributed system calls within applications accounts for the corresponding thread of execution when capturing procedures associated with network system calls. However, the technique does not account for intra-procedure activities and only captures references to data that is sent and received (e.g., file names instead of file data).

In addition to contending with properties of distributed and parallel execution, provenance capture should avoid inhibiting non-functional properties of the MASS framework. High inter- and intra-process parallelism are applied to efficiently scale operations to many agents, with little explicit synchronization. Preservation of this property precludes use of external provenance capture mechanisms or databases, as requests to these systems can interrupt multiple threads of execution. MASS also increases programmability by hiding execution details. Consequently, the approach cannot rely on these execution details or leak them to the user.

1.3 Provenance Support for Multi-Agent Models

MAM provenance centers around model design and development, to promote model evolution over time. A simple outline of the MAM development cycle includes specification, execution and analysis. Without provenance, analysis centers on simulation output, alone. With provenance, developers can verify that changes to a simulation workflow or agent behavior affect output as expected.

MAMs in MASS consist of simulation logic, agent specification and definition of spatial data. The simplest provenance support provides a high-level understanding of agent behavior. More

difficult to reason about and capture are coordination activities over agents and spatial data to help verify data representation and coordination. Arguably the most complex, yet useful, MAM provenance is that which bridges simulation logic with agent behaviors via framework operations to support full reproducibility. These support areas consist of three types of MAM-related provenance (provenance about the simulation history, execution environment, and social aspect of model development [28]) within two levels of abstraction (application- and framework-levels).

In this thesis, the adequacy of each type of MAM-related provenance data is evaluated with queries to support corresponding use cases. The ability to bridge all three types of provenance is supported by queries on relationships between levels of abstraction. Meanwhile, overhead associated with capturing various types of provenance, at different levels of granularity, is measured with respect to the scale of agents, spatial data and simulation length. Performance overhead associated with capturing each type of provenance data is also measured to put adaptive provenance capture in perspective with full provenance capture.

1.4 Thesis Statement

The ProvMASS approach can be adapted to capture provenance of shared resources in distributed memory that explains multi-agent models in execution with limited performance overhead.

1.5 Research Questions

The following research questions were investigated to support the statement of this thesis:

- **RQ1** Can provenance of shared resources be captured in a distributed memory?
- **RQ2** Can the full context of individual agent behaviors (e.g., coordination and state of agents and spatial data) be traced to simulation logic?

- **RQ3** Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?

1.6 THESIS CONTRIBUTIONS

This thesis explores the research questions with an extension to the MASS framework, called ProvMASS.

Exploration of RQ1 focuses on methods to relate data of various execution contexts with provenance that appropriately expresses corresponding data use semantics. Technical challenges linked to capturing provenance of MASS distributed memory operations are presented in the first three sections of Chapter 4. The principal concerns in investigating RQ1 include resolving data identity in distributed message passing, disambiguating concurrent intra-process procedure invocation contexts and storing provenance while coping with high resource constraints.

Description of relationships between passed and received messages hinge on consistent message identification across network boundaries, as temporal ordering cannot be relied upon, due to network latency and inconsistent processor clocks. Procedure call semantics also raise a barrier to resource identification because of data scoping (i.e. local variables are only accessible within the local procedure scope – not the called procedure scope). Resources are highly constrained in the MASS environment; a factor that imposes restrictions on the types of solutions that may be employed. For example, the MASS memory model employs weak consistency through barrier synchronization. Within the barrier, data structure is relied upon to avoid collision in writing shared resources, rather than locks. Meanwhile, highly parallelized operations over those resources keep all processing cores busy. Consequently, provenance capture mechanisms based on locking or dedicated threads must be avoided.

Novel solutions corresponding to each problem are also presented in Chapter 4; they include techniques to consistently identifying distributed resources, causally order shared memory operations and store provenance while minimally impacting concurrency. ProvMASS combines these techniques to generate provenance described in RQ1, by representing causally ordered concurrent events surrounding consistently identified distributed data in a directed provenance graph. Two such graphs were captured in execution of the SugarScape and RandomWalk multi-agent models to evaluate these methods. Queries over these graphs are presented in Chapter 6 and indicate that ProvMASS can capture provenance of shared resourced in a distributed memory.

Knowledge of relationships between simulation logic and changes in agent state help to answer RQ2. Two basic observations provide the foundation to capture provenance that connects agent behavior to a simulation workflow. First, simulation logic is embodied in source code, whereas changes in agent state are embodied in execution. Second, simulation logic instructs the MASS framework to perform operations over agents, which lead agents to exhibit behaviors. Therefore, three types of activities must be bridged to generate provenance related to RQ2: simulation driver instruction, framework resource coordination operations and agent interactions.

Multi-agent model activities in MASS relate to data represented at two levels of abstraction: the logical-level (i.e. data represented with respect to the investigated model) and the framework-level (i.e. data represented with respect to the underlying distributed memory). An approach to bridge these activities while reconciling the two data abstractions is presented in Section 4.5. ProvMASS combines the previously discussed techniques explored in RQ1 with granularity control described in Section 4.4 to generate provenance postured in RQ2. Queries tying the simulation logic of the SugarScape model to agent behaviors through framework operations are presented in Chapter 6. Results of these queries indicate that, given proper configuration of

provenance granularity, ProvMASS can generate provenance that describes the full context of individual agent behaviors and trace them to respective simulation logic.

The primary insight that prompted RQ3 is that large overhead is an obvious consequence of documenting procedure calls in a high-performance computing setting. High-performance computing results in many data operations. Meanwhile, data provenance is articulated with text and string operations can be expensive. Therefore, tracing procedure calls in a well-factored, highly parallelized library can introduce considerable overhead. Even capture of minimalistic MAM provenance (e.g., just agent procedure invocations) is considerably expensive when captured for all agents during the entire simulation. Hence, some form of control must be provided regarding what provenance is captured. Further, such control mechanisms must accommodate common patterns of provenance use.

Methods to restrict provenance-related performance overhead as simulated models scale, are presented in Section 4.4. These methods restrict provenance capture to specific activities (instructions in simulation drivers), data (number of agents), and data relationships (types and granularity of provenance data). In Section 6.2, queries for several use cases are presented along with the types of provenance required to support those queries. Later, in Section 6.3, performance measures are provided to compare execution for various models, model sizes and simulation lengths. To answer RQ3, these comparisons are supplemented with performance measures for identical executions, but with combinations of the restricted provenance capture. Results of the performance comparisons indicate that provenance capture overhead can be limited as the number of agents, space and simulation iterations scale.

Chapter 2. BACKGROUND AND RELATED WORK

2.1 Spatial Simulation and Agent-Based Models

2.1.1 *Spatial Simulation*

Systems that are defined by the proximity and state of their elements can be investigated with spatial simulations [26]. Spatial models can represent entities that interact and have simple state. For example, stochastic models such as influenza transfer or cell automata, rely on state between neighboring regions. These models represent individuals with counts of static entities that fluctuate from location to location. Mathematical models such as computational fluid dynamics, Schrödinger's wave diffusion, and Fourier's heat diffusion are also well represented by spatial mechanisms. Models from these two categories may even be combined to investigate more complicated entity-environment interactions. However, it is difficult to extend this utility to complex migratory entities, such as those found in biological or social systems. In these systems, behavioral pattern observation is better served with multiagent models (MAM).

2.1.2 *Agent-based and Multi-Agent Models*

Agent-based models (ABM) describe systems of possibly complex migratory entities that decide how to interact with each other and their environment based on internal rules [17]. In coarse-grain ABMs, agents may be highly individualized and operate as completely autonomous objects. In such models, the agents provide an interaction medium, imposing top-down development methodologies. The corresponding simulation and development paradigms are ill-suited to describe collective behavior.

Simulation of global, interactive phenomena requires more fine-grain agent definition and often, a larger number of agents. Consequently, individual entities are allotted less memory.

Extraction of interaction mechanisms between agents can help resolve memory-related issues. Additionally, or alternatively, agent complexity may be reduced by shifting some model-specific agent properties (e.g., group affiliations) to collective behavioral patterns, rather than internal state. For example, agents representing certain types of fish may decide to swim together in colder water, rather than migrating to a predestined locale. These alterations not only reduce the agent memory footprint, but also serve to facilitate bottom-up development by promoting a separation of concerns. These types of ABMs is referred to as a multi-agent models (MAMs) in this thesis.

2.1.3 *Multi-Agent Model Support in Multi-Agent Systems*

Multi-agent systems (MAS) facilitate changes in ABMs to implement multi-agent models (MAMs). In MAMs, agents are represented at finer granularity and rely on the MAS to manage underlying operations. In turn, model developers can focus on agent behavior. If the MAS also extracts environment detail from agent state, MAMs may be used to predict collective patterns in greater detail. For example, agents that represent cars in a traffic simulation may predict congestion to detail larger routes of travel. In another example, growth rate of biological entities may be tied to static properties of congregation areas, such as water or food supply. In the first example, inclusion of the environment property (i.e. cars on each street throughout the driver's route) reduces the communication necessary to make decisions, allowing support for more drivers, each with their own route. In the second example, expression of the food or water supply levels need not be communicated between agents. Instead, interactions are limited to specific agent locations, allowing areas to be represented in finer detail.

2.2 MULTI-AGENT SPATIAL SIMULATION (MASS)

The Multi-Agent Spatial Simulation (MASS) library provides a parallel/distributed framework to automatically parallelize simulations with symmetric multiprocessing over a cluster of computers. While other parallelization technologies such as OpenMP, MPI and Hadoop support scientific computing in grid and cloud environments, the MASS library was created with the intention of bridging the semantic gap between underlying parallelization constructs and entity-based model analysis [14]. Parallelization of spatial simulations with MAMs affords the performance required to analyze detailed changes in large systems. In addition to handling low-level object manipulation, both distributed execution details (e.g., locale) and concurrency context (e.g., thread of execution) are also hidden from the developer. Consequently, coordination of underlying agent operations facilitates a focus on modeling agent behaviors.

2.2.1 *Coordination of Distributed Memory*

The MASS library organizes a distributed memory of agents and spatial data, through distributed computing, that are operated with symmetric multi-processing. Coordination of distributed computing resources proceeds in the master-slave paradigm, wherein distributed processes are connected through secure shell (SSH) tunneling. Meanwhile, parallel coordination is extended to intra-process operations at each host through multithreading.

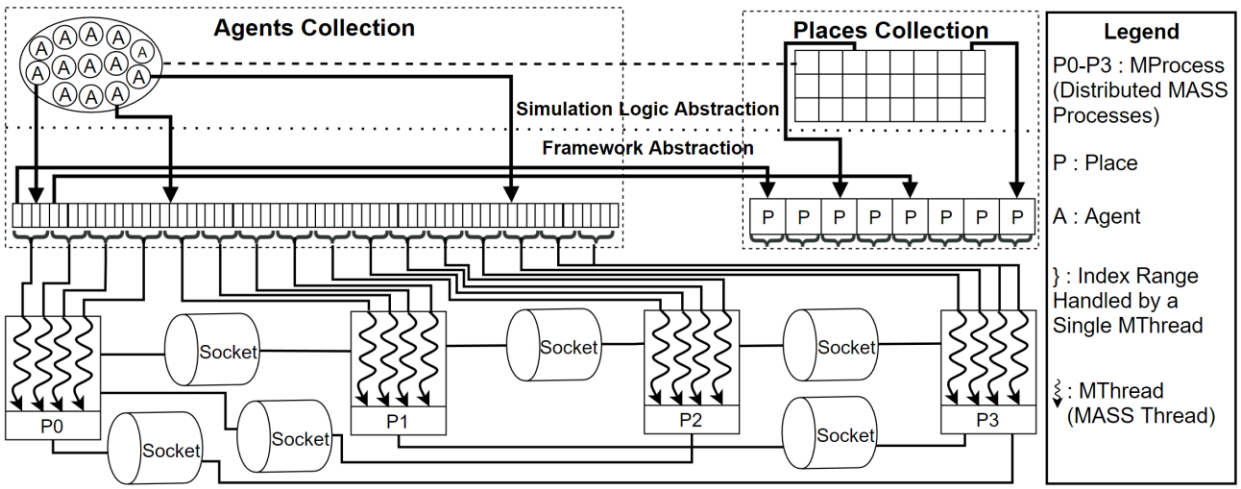

Figure 1. Coordination of shared resource operations in MASS distributed memory

MASS stores simulation space and agents as shared resources in a distributed memory, managed at the object-level (shown in Figure 1). Host processes are referred to by class name, as *MProcess*; each of which maintains several thread objects, of the *MThread* class, which operate shared resources (e.g., *agents* and spatial elements). A distributed array containing spatial elements, called *places*, is stored in sections that span multiple processes in physically disparate hardware. Meanwhile, *agents* are mapped to those *places* and are contained in an additional global array.

Relationships between MThreads and ranges of application resources (e.g., *agents* and *places*), are also shown in Figure 1. Coordination of management operations is first accomplished by calculating the storage range for each host process, based on an integer identifier, then by calculating the subrange with respect to threads of execution. For instance, the sixth *place* (in the range of zero through fifteen elements) is stored on a host with the process identifier of one (where identifiers range from zero to three). Since *places* with indices four through seven are handled by this host, which manages four threads of execution, the sixth *place* is operated on by the third thread (with an identifier of two, in the range of zero to three).

Exchange of data occurs between pairs of *MThreads*, across all processes, ignoring *MThreads* within the same host. Exchange occurs concurrently between *MThread*-pairs via separately spawned communication threads. In the case of the configuration shown in Figure 1, every *MThread* spawns up to 24 exchange threads (12 to send and 12 to receive) because there are four threads for each of four processes. Each process maintains one socket connection to each other process. Communication threads paired to threads residing on the same remote host share a socket. The communication operation is barrier synchronized by the *MThreads*, but communication threads concurrently read and write in an unsynchronized manner. Recall that each thread handles a specific range of resources in the global arrays to ensure that concurrent element-read and -write operations avoid data collision despite lack of locking mechanisms. Consequently, full network and processor utilization ensues. Note, however, that message delivery is unordered; a factor in capturing the provenance of messages.

2.2.2 *Shared Resources at Multiple Levels of Abstraction*

The MASS library provides distributed shared resource management specific to spatial and agent models, while hiding not only the associated communication and concurrency mechanisms, but also data locale. The top half of Figure 1 describes the relationship between agent and place resources in their logical collections, as provided by the user interface, and within the context of the framework memory and parallel coordination. Above the abstraction boundary, logical collections of *Places* and *Agents* are shown. A *Places* collection represents an n-dimensional array of spatial elements, where each element addresses an instance of the *Place* class. The collection provides an interface for coordination of procedure invocation and data exchange. More specifically, ranges of the collection can be instructed to invoke exchange procedures that provide neighboring places with data. These procedures are written by the user and constitute part of the

spatial model. Similarly, ranges of places can be instructed to invoke a specific procedure that represents an activity of the model. Meanwhile, an *Agents* collection represents all the agents that reside within a *Places* collection. Like *Places*, *Agents* can be instructed to invoke a procedure that models an activity. Distributed coordination features such as termination, propagation, and migration are also provided at the application level [9]. Meanwhile, the mechanisms required to orchestrate such distributed manipulation are handled within the framework.

From the developer perspective, coordinated behavior is encoded within procedures of the agents and places. Application specific agent classes inherit from the corresponding framework parent abstraction; a similar inheritance relationship exists for application specific place classes. The *Agents* and *Places* collections provide a method to call a single procedure on a range of application specific agent instances via an inherited *callMethod* procedure, which takes as a parameter an identifier associated with the procedure that should be called. The developer provides the conditional logic required to call the appropriate method based on the provided procedure identifier.

Increased programmability has been shown for simulation models and logic in MASS versus combining lower-level parallelization frameworks like Open-MP and MPI [3]. This feature is partly a consequence of focusing the programming paradigm on model description and simulation coordination; partly a consequence of providing coordination mechanisms while hiding underlying resource manipulation. While affording developers the ability to focus on modeling, semantics of these hidden operations play a role in anticipating simulation state, which in turn, plays a role in modeling. For instance, in a model that should prevent agent collision at a single spatial element, what outcome is anticipated when two agents attempt to migrate to the same place? The answer is obvious if there is no *guarded agent-migration* mechanism; a logical error occurs, in which

multiple agents are associated to the same place. However, with *guarded agent-migration*, which place can the user expect each agent to end up at? The confusion is compounded when *ghost-space management* (i.e. sharing border places between neighboring hosts) is added to the example. The user may not know the outcome of migrating two agents from different hosts to the same place, when the place is represented in *ghost-space* on one host and in normal space on the other. While automated data coordination mechanisms circumvent the need for a deep understanding of distributed and parallel computing, details of data exchange between places and agent migration remain important considerations in model design and verification.

2.3 Provenance Support for Multi-Agent Systems

Provenance about the social aspect of model development, execution environment and simulation history are three categories of provenance that support agent-based simulations [28]. A closely related work presents an approach to explain agent behavior with provenance [7] captured in a tool called Provenance in NetLogo (PIN) [19]. This approach combines a data slice with the program slice to form a provenance slice that explains how an agent arrived at a specific state. The provenance slice is most closely categorized as provenance about the simulation history. This type of provenance is based on discrete event modeling in [28] and source code instrumentation in both [7] and ProvMASS.

The ProvMASS approach is distinctive in that it also considers provenance about model development with respect to reproducibility, and execution environment with respect to low-level details such as locale and execution context (i.e. differentiating procedure invocation with respect to thread of execution). Further, these types of provenance are integrated into provenance about the simulation history. Related provenance capture approaches do not consider distributed and shared execution context. The core distinction, however, revolves around not only capturing, but

also relating procedures, arguments, return values, field access and assignment *in situ*; a necessity of capturing the semantics of shared-resource operations within a distributed memory without reliance on temporal ordering. Provision of such relationships affords the ability to bridge all three types of ABM provenance.

2.4 Provenance in Distributed and Parallel Computing

2.4.1 *Provenance for Data Intensive Scalable Computing*

Provenance capture techniques for distributed and parallel computing environments have been discussed in the context of data intensive scalable computing (DISC) systems such as Hadoop [1][10][27][38] and Spark [20]. Several techniques add provenance-related identifiers to augment transformation input and propagate provenance through the shuffle stage, pairing map and reduce tasks [10][27][38]. The benefit of this approach is that data is consistently identified while traversing various execution contexts. However, this approach results in a larger memory footprint and corresponding performance degradation, as the data size grows with each mapping. Through framework instrumentation, rather than task wrapping, performance degradation can be avoided by leveraging access to data mappings between tasks [1][20]. While data is no longer consistently identified *in situ*, this only results in delayed graph construction, as DISC systems assume independence between sibling tasks.

Table 1. Operating overhead and provenance collected by other techniques

Provenance Collected	Overhead Reported	Category
Data slice that includes procedure invocations; program slice consisting of conditional statements, read operations and write operations [7]	250% (includes message passing and persistence in addition to collection)	Multi-Agent Systems
Relationships between users and files, System calls (open, close, read, write, exec, fork, exit, clone truncate, and rename [18]	9-12% over normal execution of the BLAST genome sequencing tool	Distributed System Calls
System calls (from [18] aliased by procedure invocations, primitive parameters and return values [36]	5-13.8% additional execution time of four tinyhttpd procedures	Client-Server Applications
Relationships between provenance-aware applications with respect to file transactions (e.g., open, close, read, and write operations) [25]	As low as 1.3% average overhead for processing intensive cases; as high as 14% average overhead for system-call intensive test-cases	Layered Systems
Read and write instructions with corresponding thread identifier as 64-bit word [22]	As high as 153% increase for Communication Traps (CTraps) and up to 50% increase for Last Writer Slices (LWS)	Symmetric Multiprocessing
Read and write memory operations by PThreads [37]	250-3500% execution time	Symmetric Multiprocessing
Output records (of a stage, shuffle step, etc.), input records and the resilient distributed dataset (RDD) transformations that relate them [20]	110-129% execution time	DISC Systems
Record-level derivations [1]	Under 10% on a typical MapReduce workload	DISC Systems
Input and output of MapReduce jobs [27]	20-76% increase for Wordcount and Terasort jobs	DISC Systems
Dependencies between data and actor executions [10]	Under 100% increase in WordCount workflow execution	DISC Systems

A consequence of assuming sibling task independence is that provenance captured in DISC systems [1][10][20][27] does not describe the relationship between tasks, subtasks and intermediate data. The types of provenance collected by these and other techniques are shown in Table 1. While DISC systems coordinate fine-grain processing of distributed data, approaches to capture provenance in these systems are not as closely related to capturing provenance of multi-agent models in distributed memory as those in the shaded rows of Table 1. Not only do constraints on task independence inhibit direct application of the DISC approaches, but also compel task-level provenance granularity. Concurrent procedure call semantics imply the possibility of interleaving data derivations. Consequently, fine-grain provenance is necessary to describe shared resource operations in a distributed memory, as agent behaviors may be interconnected through shared data operations. ProvMASS handles this challenge by employing parallel provenance storage to enable pairing of shared resource operations with their thread of execution. This solution also negates the bottleneck introduced by external storage that is noted in [20]; of even greater importance in MASS as such bottlenecks subvert parallelism. At the same time, parallel provenance storage supports the cross-context resource identification required to reason about interdependent tasks, while avoiding wrappers and growing data sizes.

2.4.2 *Provenance for Shared Memory Programming*

Provenance capture techniques for shared resource operations have been investigated in the context of synchronization constructs in shared memory programming [22][37]. Here, intra-process memory operations are captured at the instruction level and can be paired with core-dumps to determine the cause of an error. Debugging of general shared-memory multithreaded programs is supported, but such low-level provenance can be difficult to tie to models in a user-meaningful way.

These techniques apply provenance capture in stricter memory models than those provided by the MASS library. For instance, one approach [37] replaces PThreads, providing provenance capture at release consistency. Meanwhile, MASS uses barrier synchronization to implement shared memory operations at weak consistency. Generally, stricter memory models enable relaxation of assumptions about correctness, but at the expense of performance. ProvMASS does not solve this issue from the perspective of correctness (e.g., reconciling race conditions) for application-level shared memory operations. Instead, it relies on synchronization or locking implemented by the developer. However, it does preserve increased performance offered by the weak memory consistency model by automatically segregating provenance storage for concurrent capture of resource coordination at the framework-level (i.e. uncoordinated concurrency within barrier synchronizations).

While inconvenient provenance granularity and performance degradation simply impede application of these approaches to capture distributed shared resource operations; the central concern is that they do not account for procedure invocation, which is key in consistently identifying distributed communication. Meanwhile, ProvMASS handles these issues in a holistic manner, allowing queries in closer proximity to natural language. For example, it is possible to find an aggressively driven sub-compact that started its journey from a road physically located on my.laptop.host.name and arrived at a destination physically located at university.host.name. While it is necessary to join queries, it is not necessary that any of those queries reconcile low-level memory operations with high-level resources.

2.4.3 *Distributed Application-Level Provenance*

A technique to automatically instrument application-level provenance capture has been evaluated in a distributed setting [36]. While capture of remote communication is only evaluated in server-

side procedures of a distributed application, integration with a SPADE [18] reporter service implies the ability to alias network system calls with procedure invocations. At query-time, distributed system calls can be paired [23]. Concurrent procedure invocations are differentiated based on the procedure name, call count, and thread name in [36].

Provenance is captured externally at the system-call level, with an LLVM Reporter component that assumes provenance generation does not exceed the ability to consume it. These calls are matched with ordered procedure invocations. While procedure invocation contexts are differentiated, the restriction on ordering makes this technique incompatible with unordered data exchanges in MASS. Further, such data exchange would result in provenance generation far exceeding the ability to capture it. This is partly a consequence of high system resource use (i.e. near-constant concurrent execution) and partly a consequence of the eventual need to persist the provenance data. While adaptation of this approach may be possible with modification, ProvMASS avoids these restrictions by leveraging parallel in-memory provenance storage and consistent inter-process resource identification.

2.4.4 *Connecting Layered Provenance*

Integration of provenance across multiple layers of abstraction has been investigated with a provenance-aware storage system [25]. Applications disclose provenance through an API that interfaces with PASSv2 (a provenance aware storage system). Consequently, data from various layers of abstraction are connected at the system level.

A benefit of this approach is the ability to provide user-meaningful names to identify file provenance. However, provenance at system-level granularity falls short of bridging distributed and procedure-level operations at multiple layers of abstraction within a cohesive distributed memory. The approach taken in ProvMASS, on the other hand, bridges these layers in the parlance

of specific MAMs. Meanwhile, interoperability with other provenance systems is promoted by using the Prov Ontology [29] to express these relationships with W3C Prov [30] concepts.

Chapter 3. MOTIVATION AND USE CASES

Multi-agent model (MAM) research centers on analyzing patterns of collective behavior, exacerbating difficulty in model validation. It has been noted that the core agent-based model (ABM) philosophy warns against focusing on emergent outcomes, as they may be misleading, and instead promotes emphasis on interactions and uniqueness of the individual [7][2]. The impact of individual interactions is especially difficult to ascertain in the results of MAM simulations, as agents are represented at fine-granularity, necessitating provision of fine-grain provenance. This shortcoming motivates several use cases based on the types of ABM provenance described by Pignotti et al. [28], discussed in the previous chapter (see Section 2.3); which combine to strengthen model validation through verification of individual agent interaction, with respect to model specification and the execution environment. This thesis considers three use cases, in which:

1) A researcher analyzes agent decisions to understand individual agent behavior;
2) A colleague interprets the corresponding simulation logic and configuration to understand simulation execution;
3) An analyst debugs execution of distributed operations to verify the agent behavior exhibited in first case.

These use cases highlight novelty of the ProvMASS approach while also forming the basis of evaluation in Chapter 7. Two simulations illustrate requirements for adequate use case support while emphasizing overhead limitations: SugarScape [15] and RandomWalk [9].

3.1 THE SUGARSCAPE MODEL

SugarScape is a simulation in which agents move in a two-dimensional space, consuming predetermined amounts of "sugar" from destinations upon arrival. In each simulation step, the agent decides where to move next based on the ratio of sugar to agents in neighboring locations. When the agent arrives with empty sugar stores at a destination that is also devoid of sugar, it sets a termination flag for the next agent update operation. Computationally intensive neighbor update operations and limited host traversal help illustrate a specific overhead profile while providing the opportunity to examine detailed agent behavior, including field access and assignment.

3.2 THE RANDOMWALK MODEL

RandomWalk also provides a simulation of agent movement in a two-dimensional space. Primarily a test simulation, RandomWalk offers the ability to analyze simple properties of the physical network. In the original simulation [9], agents migrate to one of four randomly selected neighboring locations. A modified version of RandomWalk is used to provide some variety in network communication while evaluating scalable capture of framework operations. Specifically, agents migrate to random locations to apply network saturation.

3.3 USE CASE 1: UNDERSTAND AGENT BEHAVIOR

In the first case, a researcher would like to understand the circumstances surrounding an agent's behavior, specific to a group of interactions with its environment. Understanding of individual agent behavior assists in validation and as a result, model evolution. However, social science research tends to "filter out" various agent behavioral effects [16]. The same can be said of biological system research with MAMs. For example, a voracious foreign species of fish may be

introduced to a model of marine habitat with the intention of measuring predator population. In the initial model, predators simply consume prey. A new behavior may be introduced that includes attacks on predators. Meanwhile, the simulation logic remains unchanged; outcomes are still measured with respect to prey populations. Such consequences are accounted for in individual-based models [13][32], but simply lead to correlation of changes in modeled outcomes.

While scientists are trained to be cautious of false attribution, in isolation within a simple model, correlation of behavioral modification to simulation outcome essentially implies causation. However, introduction of many similar behavioral modifications over time increases model complexity. Consequently, it becomes more and more difficult to anticipate the impact of these changes, and in turn, to validate the model. Understanding of *how* an agent behavioral modification led to a new simulation outcome is confined to static model analysis. Meanwhile, verification of agent behavior is not only hampered by increased interaction complexity, but obstructed by non-determinism.

In addition to supporting model validation and verification, provenance of individual agent behaviors can also add new modeling capabilities such as agent migration tracing. For example, a researcher may wish to trace the routes taken by a specific vehicle in icy weather conditions. While it might be possible to achieve a similar affect by gathering agent locations in every simulation step, such provenance eliminates the need to accommodate corresponding modifications to simulation logic.

3.4 Use Case 2: Understand Simulation Execution

In the second case, a colleague would like to reproduce the findings of an experiment run by the researcher mentioned in the previous section. It has been argued that replicability is an impoverished version of reproducibility; that the context in which an experiment is carried out

invariably changes with the researcher, the lab, and the equipment, despite all efforts (e.g., using same source code) and that such differences matter [12]. The more robust version of replicability, reproducibility, is supported with the context of the results, which includes not only the software, but the details of the computing environment [11]. Alone, however, this information is insufficient to support full reproducibility of MAMs in MASS, as execution context is complex and models are handled at multiple levels of abstraction. Further, opacity of framework operations hinders an understanding of source code changes with respect to the execution environment.

Provenance to understand individual behaviors is extended to the execution environment in the next use case, but even this amount of contextual information is insufficient to bridge overarching logic (e.g., data staging, iterative directives, and result gathering) to individual interactions (e.g., spatial and agent operations), when considered in isolation. Provenance of these dynamic relationships with respect to prospective provenance (e.g., simulation logic embodied by a source code) can help fellow researchers understand simulation execution and promote reproducibility. In turn, both model evolution and experiment derivation are supported.

3.5 Use Case 3: Debug Distributed Execution

In the third case, the researcher mentioned from case one could verify the conditional logic of an agent behavior, but finds that the corresponding agent state is unexpected. Consequently, the researcher would like to investigate assumptions about framework operations. A traditional debugger may be provided by the development environment, but such debuggers only monitor local execution. Meanwhile, debugging mechanisms explored for the MASS library introduce features to monitor framework operations [34] and agent state [21] separately, but those features do not express how changes in state depend on the operations.

The MASS distributed execution environment, discussed in Chapter 2, hides a complex landscape of distributed concurrency, characterized by nondeterministic data handling that is guarded by barrier synchronization. Such system properties promote efficient coordination of distributed memory while enabling system modeling defined by concurrent agent interaction (i.e. mimicking real systems), but can also obscure understanding of corresponding agent interactions. Simulation orchestration features hide these resource coordination operations at the framework-level of abstraction, allowing developers to concentrate on modeling resources at the application-level of abstraction. However, execution details of these operations may impact agent behavior, yet are hidden from the developer. Meanwhile, split resource abstraction obstructs methods to relate framework-level operations to model-specific activities.

Identification of intermediate data leading to exceptions and trial-and-error interrogation of associated tasks are common debugging patterns that motivate provenance capture in DISC systems [20]. In the case of distributed execution in MASS, where sibling task independence cannot be assumed, these debugging patterns extend to subtasks and internal agent state. Further, framework operations may be started by these subtasks (e.g., coordination of the next migration or data exchange) and task coordination is unordered. Provenance of these data operations, including relationships between them, can help to further verify the agent behaviors described in the first use case.

Chapter 4. APPROACH

Tracing agent behavior to simulation logic through distributed memory operations within MASS presents various technical challenges. Chief among these challenges is the ability to connect data generated and used at multiple levels of abstraction. This is especially challenging in the context of high-performance distributed and parallel computing, where distributed communication introduces difficulty in identifying data, shared memory programming precludes traditional methods of connecting procedure invocations through temporal ordering, and high resource constraints impose limitations on provenance collection and storage. Meanwhile, the properties of the original MASS system design must be upheld. The remainder of this chapter provides details on each of these challenges, along with corresponding novel solutions offered by the ProvMASS approach.

4.1 Challenge: Consistently Identifying Distributed Resources

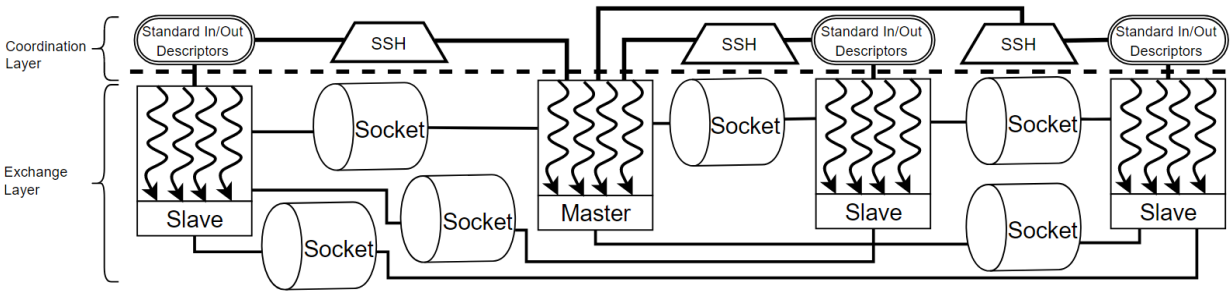

Figure 2. Communication layers for host coordination and data exchange in MASS

MASS coordinates distributed resources over a computing cluster as outlined in Chapter 2; a master host process coordinates remote slave processes with message-passing, via secure-shell (SSH) tunneling, as shown in the coordination layer of Figure 2. While messages passed in the coordination layer (i.e. messages delivered between master and slave processes) are delivered at

low frequency, those passed in the exchange layer (i.e. messages containing neighboring place data, possibly including agents) involve many concurrent socket operations. These coordination mechanisms make it difficult to identify distributed resources within applications. Other techniques only solve some of these problems and only in systems marked by ordered, low frequency communication. Meanwhile, ProvMASS addresses this issue in the rapid, unordered delivery that characterizes high-performance distributed shared memory operations.

One method of identifying distributed resources is to pair read and write operations between hosts via provenance sketches [23]. At the application level, this involves aliasing system calls to corresponding procedure invocations [36], but aliasing relies on ordering. Consequently, application of this approach to unordered concurrent communication poses risk in generating many false-positive relationships between read and write operations on different hosts. While many simultaneous communications can be ordered within a given host, accuracy cannot be guaranteed between hosts, as system clock counts differ.

This challenge is addressed by consistently identifying shared resources rather than matching read and write activities across network boundaries. Thereafter, those identifiers can be used to relate activities through the shared data. Consistent inter-process resource identification is made possible by augmenting each resource with a universally unique identifier (UUID) during instantiation. Later, the identifier is accessed via an interface shared by all distributed shared resources (e.g., agents, complex exchanged spatial data, and messages). ProvMASS constrains message use to make this technique effective. More specifically, messages may not be reused; otherwise, causal ordering cannot be guaranteed without read-write ordering, due to the possibility of message delays. Following the reuse constraint implies unique pairing of message passing

operations in the provenance record, since those operations are paired with respect to the messages they use or generate, rather than timestamps.

In addition to message passing relationships, identifier augmentation of shared resources is extended to the migratory entities packaged in messages; namely, agents and place data. This allows query complexity to be reduced for data relationships that can be specified based on event ordering. For example, consider two queries that join to explain one behavior that depends on multiple interactions, each of which occurs on a different host. If those interactions occur within adjacent iterations of the simulation logic, traversal of all message-passing operations is not necessary to determine inter-process dependencies on shared resources (e.g., interactions started by the same agent, but on different hosts). Instead, activities that are started by those agents or that use exchanged place data, can be temporally ordered to relate consistently identified agents and place data to inter-host activities that make up a specific intermediate behavior. Meanwhile, behaviors that depend on activities concurrently executing on multiple hosts can still be related via direct message-passing relationships (e.g., chains of procedure invocation, bridged through uniquely identified shared messages).

The ProvMASS approach to consistently identify distributed resources, described above, is like some techniques for capturing provenance of MapReduce tasks [10][27][38], in that some provenance accompanies transmitted data. However, the key difference is that the accompanying provenance also identifies tasks that generated or used the data. These task identifiers build up over time, continually enlarging the transmitted data as it is passed to the next task, or through mapping in the MapReduce shuffle stage. This is a caveat that is inapplicable to the ProvMASS approach, as such relationships are recorded *in situ*, rather than formulated at the end of computation. The resultant relationships represent two vertices connected by an edge, that can later

be added to a provenance graph without additional interpretation. Meanwhile, the MapReduce task provenance, mentioned above, is parsed into task input, task, and output (which becomes input to the next task). That approach reduces the overall size of the raw provenance and the coordination required to collect the provenance data, making it well-suited to data intensive scalable computing, which places a higher importance on horizontal scale rather than performance. However, MASS employs a high degree of parallelism to increase performance and the master-slave paradigm supports only a medium amount of scalability (i.e. cluster rather than grid/cloud computing), as discussed in Chapter 2. Meanwhile, inter-dependencies exist between shared resource operations, making consistent distributed resource identification a better fit for MASS execution.

4.2 Challenge: Ordering Operations in Shared Memory Programming

Capture of procedure invocations with parameter and return values is sufficient to describe call and return semantics in sequential systems. Dependency analysis can proceed directly from the sequence of recorded data operations and procedure invocations, as they occur within a single context of execution. However, such assumptions break down in the face of concurrent execution. Even after adjacent invocation contexts are differentiated, shared resource operations impose limitations on how provenance may be captured and stored. ProvMASS solves these challenges with concurrent *in situ* capture of procedure invocation relationships and shared data operations. A novel resource matching technique helps to specify these relationships, despite differing scopes of execution. High resource constraints and desired performance properties of the execution environment preclude use of external capture, such as databases; a stipulation discussed further in the next section. Meanwhile, resources are not only shared by differing threads of execution, but by physically disparate processes, as described in Chapter 2. A simple example comparing procedure-level provenance capture in sequential versus concurrent invocation patterns serves as

a good starting point to illuminate basic requirements of this approach. Thereafter, provenance requirements imposed by the MASS execution context can be investigated in further detail.

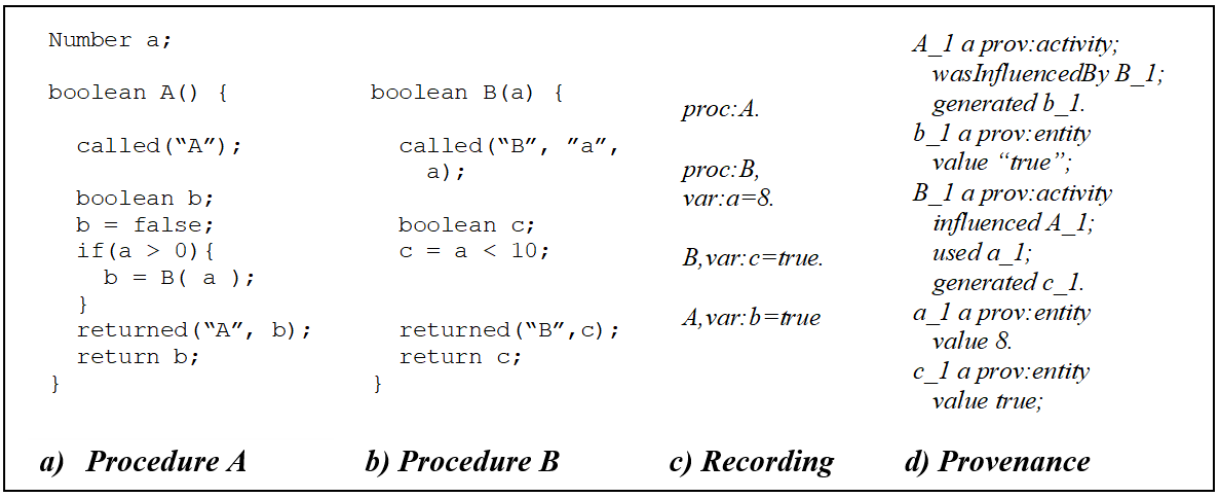

Figure 3. Provenance from recording sequential procedure invocation

MASS execution context necessitates a new technique to order operations in shared memory. Consider the set of procedures shown in Figure 3, chained by sequential invocation. Procedure A calls procedure B with parameter a, receives a true Boolean value in return, and returns that value to the caller of A. The sequence of events is shown in the recording: procedure A was invoked; then procedure B was invoked, with a parameter, a, that had a value of 8; procedure B generated a variable, c, that had a true Boolean value; procedure A generated a variable, b, that had a true Boolean value. Assume that implicit returns are recorded with an indication that nothing was returned to mark the end of an invocation (e.g., void). By splitting the recording on delimiters in an algorithm to model procedure call semantics, the sequential execution that led to this recording can be constructed, as shown in the provenance record at the far right of Figure 3. The provenance suggests that the invocation of procedure B was a part of the invocation of procedure A (i.e.

procedure B influenced procedure A). However, this assumption is only accurate due to sequential procession of procedure invocation.

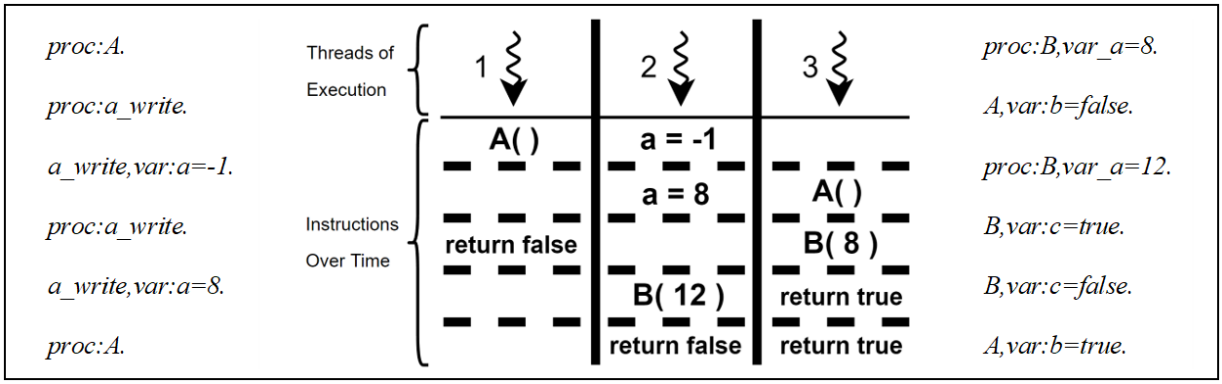

Figure 4. Recording concurrent invocation in sequence

Consider the set of procedures of Figure 3, chained in various threads of execution, as shown in Figure 4. Also shown is a sequential recording of these events, ordered from top to bottom and left to right. Note that it is now impossible to infer involvement of procedure B in procedure A based on the order of invocation and return. The context of execution for each event (e.g., thread identifier, procedure name and invocation count) may be included to differentiate procedure invocations. This post-processing provenance technique (i.e. determining causation from ordered events) is applied in [36] to pair system calls with corresponding procedure invocations.

While post-processing execution context may also apply to constructing a chain of procedure invocation, the technique ignores several issues in high-performance computing environments. The most obvious is the amount of invocations that may lead back to the entry-point of the process (e.g., the first procedure invocation in the program execution). Note, in Figure 3 and 4, that to determine involvement of procedure B in the invocation of procedure A requires both invocation and return events. This implies that any sufficiently complex operation will either require a large

amount of memory to store all the identifiers (e.g., hashing invocation identifiers) or involve many traversal operations (e.g., repeatedly examining nested identifiers). Further, the mechanism used to count invocation must either be synchronized or distributed to prevent race conditions during increment and decrement operations. Relatedly, the technique does not automatically reconcile operations on shared resources. Further, it cannot be used to differentiate operations on distributed shared resources, since it relies on temporal ordering and system clock counts vary on different hosts.

To order operations over shared resources in a distributed memory, ProvMASS introduces a novel resource matching technique that accommodates parallel intra-process capture and storage through a component called a *ResourceMatcher*. Putting aside the necessity of parallel intra-process capture and storage, discussed in the next section, this technique avoids the pitfalls of postprocessing execution context. The *ResourceMatcher* tracks procedure invocation identifiers and related variable identifiers with respect to thread of execution. As an invocation scope opens, an identifier for the procedure invocation is added to the *ResourceMatcher*. As the invocation scope closes, the corresponding identifier is released. This allows the caller (i.e. the procedure invocation leading to another procedure invocation) to be paired with the callee (i.e. the procedure invocation that resulted from another procedure invocation), and *vice versa*, while retaining only the identifiers necessary to build concurrent procedure call chains. While the resulting provenance data does not yet form a graph, it contains all necessary relationships to directly construct a causal graph, including those between field access and assignment and the procedure invocation in which those operations occur.

```
Integer a = new Integer( 1 );

// record generation of "a" by this procedure and get its resource identifier
String aID = documentEntity( "a", a, … );

// identifier for local variable "a" is staged for matching
matcher.addEntity( aID );

// aID is matched to the identifier of the local parameter in procedure C and
// the matcher is provided a return variable identifier at the end of procedure C
Boolean B = C( A );

// retrieve the identifier of the returned variable that is local to procedure C
String cReturnID = matcher.removeEntity( );

// capture that B is an alternate of the returned resource
referenceReturn( B, cReturnID );
```

Figure 5. Relating inter-procedure variables through the ResourceMatcher component

Matching resource identifiers in memory also enables the specification of relationships between data from different procedure scopes. The *ResourceMatcher* can be used to explicitly generate such relationships, as shown in Figure 5. For instance, the arguments provided for a function call can be tied to the local parameter variables in the called procedure. Similarly, the local data returned by the callee may be associated with an assignment to a local variable of the caller. Caller-callee relationships between procedure invocations and between procedure invocations and shared data operations are provided by provenance recording procedures. These recording procedures are automatically inserted into each source code file prior to compilation. Conversely, the data-matching operations shown in Figure 5 and described above are optional features provided to the developer. This is necessary to reduce memory and performance overhead. Note the assumption that the provenance resource identifier corresponding to the return of procedure C was staged through the *ResourceMatcher* before returning to the caller. Automatic instrumentation of the extended resource matching technique would require that all procedures add

identifiers of returned variables to the *ResourceMatcher*. Instead, developers can use the extended resource matching features to investigate procedures of interest.

4.3 Challenge: Persisting Provenance in a Highly Resource Constrained Environment

Applications that utilize a high degree of parallelism demand the same level of responsiveness from a provenance collection infrastructure. In scenarios marked by low-throughput provenance generation, communication of provenance data and subsequent storage may be dealt with asynchronously. For instance, provenance of a procedure invocation may be sent to a database and the database may acknowledge the request immediately, instead of waiting for the data to be persisted. Thereafter, the procedure that enacted the provenance capture may continue execution. Next, that procedure may call another procedure that makes the same type of request to the database. However, the request may be received before finishing the previous storage operation. Depending on the configuration of the database, buildup of these operations over time may result in discarded provenance, ignored requests, or in the best case, a blocked call (delivering the acknowledgement after persistence is completed).

When provenance is delivered to another application or service, with separate memory, that provenance capture is typified as being “inter-process” (e.g., requests to a database, as described above). Such provenance capture comes with the drawback of having to dedicate resources to the external process and being less responsive, by relying on system calls for communication (e.g., Sockets). Conversely, “intra-process” provenance capture occurs within the same application and memory space. A drawback of intra-process capture is that it necessitates modification and/or extension of the existing system and is by definition, detached from other applications, and thereby introduces extra steps to integrate provenance with that of other systems. However, performance

is a key benefit of intra-process capture. The MASS processing environment, described in Chapter 2, puts all processing cores and memory to use for most operations. Consequently, it is important to avoid hampering parallelism with unwarranted thread interruption. Several related provenance capture techniques make use of external provenance capture infrastructure that can reduce or interrupt concurrency. Others capture execution relationships internally to avoid transmission overhead, but do not reconcile operations over shared resources. Meanwhile, ProvMASS combines in-memory parallel storage and lightweight provenance management to capture these relationships *in situ* without inhibiting parallelism.

Parallel in-memory provenance storage prevents thread stalls by eliminating the need for additional synchronization and by isolating interruption to the requesting thread. Interruption of the requesting thread is unavoidable for high-throughput provenance generation. However, external storage through inter-process provenance capture may also interrupt other threads in addition to the requesting thread, since at least one thread is dedicated to fielding incoming requests. This caveat may be avoided by dedicating processing resources to the provenance capture infrastructure, but the application is then denied those resources. Thread interruptions are reduced with intra-process provenance capture as in-memory storage can be directly accessed by thread requesting provenance capture.

Lightweight management of parallel provenance storage can help to maintain concurrency. While intra-process provenance capture addresses reduction of processing resource availability, it does not resolve contention over write access to in-memory provenance storage. As described in the previous section, *in situ* capture of procedure call relationships takes place at the beginning and end of each procedure invocation. Consequently, synchronization of provenance capture operations would severely impede concurrency, and in turn, preservation of increased performance

offered by the MASS framework. To combat these impediments, ProvMASS introduces a novel technique that isolates capture and storage of provenance data to a single thread of execution, through a provenance storage component. Each instance of the storage component is mapped to an individual thread of execution and maintains an isolated provenance storage buffer to simultaneously avoid the use of locks and write collision. These mappings are handled by the provenance management infrastructure; which creates new mappings for a single requesting thread at one time, but can concurrently provide multiple stores from those mappings, thereafter.

Persistence is also managed independently by each provenance store. While the operating system may become a focal point of performance overhead, due to concurrent persistence to secondary storage, several properties of the provenance store help to distribute persistence. Provenance stores are double-buffered, writing when the store buffer is full, but only flushing to disk when the secondary buffer is full. Persistence is staggered with respect to the frequency of procedure invocation and size of procedure input and output. Such frequency is somewhat uniform, at least between MThread executions (see Section 2.2.1). When multiple provenance stores simultaneously wait for access to secondary storage, procedure invocation is automatically staggered. Finally, buffer dimensions are also configurable, helping to support such staggered persistence, in a bid to relieve contention. Regardless of the buffer size, secondary storage access is orders of magnitude slower than main memory access and nothing short of restricting provenance capture to memory will serve to overcome this bottleneck. However, this technique serves to retain some concurrency without unneeded reduction in processing resources while supporting concurrent capture of shared resource operations. More importantly, the design of the provenance store supports future work (e.g., *in situ* query over provenance in memory) by layering the storage buffer to allow simultaneous reading and writing.

Data is added to provenance stores as independent triples that are later used to construct a provenance graph. Triples consist of a subject, predicate and object, that represent a source node, a directed edge and a destination node, respectively. Deferred graph construction reduces provenance collection overhead at the expense of delayed query, as discussed in related literature [1][20]. While details of the ProvMASS provenance model are discussed further in Section 6.1, triple isolation is mentioned here because it not only supports delayed graph construction, but also enables provenance stores to be detached from dormant threads and recycled. In turn, less provenance stores need to be provisioned, reducing the in-memory storage requirements.

While persistence and storage of provenance data is delegated to the provenance stores, residual responsibilities are handled with a lightweight provenance store manager. Most global operations are freed of bottlenecks by offloading data management to the provenance stores. However, thread-mapping and provision of parallel provenance storage is centralized. Such initialization operations are isolated to a single event, since synchronization is required to ensure the stability of the corresponding collections. Thereafter, read operations proceed in an unrestricted manner to support concurrency. Dynamic modification of global configuration settings is also supported. However, such operations simply utilize the existing MASS coordination mechanisms. While the manager references the overall storage-space utilized by the provenance stores of a single host, the space is not manipulated by the manager, avoiding a coordination bottleneck. This design decision also enables future work in reducing the performance impact of persistence, by introducing a means to monitor assigned buffers and tag some stores for premature persistence.

4.4 Challenge: Achieving Acceptable Performance Overhead

The ProvMASS approach predicates design philosophy on two simple observations about processing overhead. The first is that levels of provenance capture overhead correspond to

processing density (i.e. the rate at which tasks are executed and data is moved per unit of time) and provenance granularity (i.e. The types of data and relationships that are being recorded). The second is that these overhead levels are related to degradation of preexisting performance properties. Significant overhead is attributed to persistence, but directly relates to the provenance size. Acceptable performance is achieved by manipulating these factors. In part, this is accomplished by maintaining MASS performance properties, as discussed in the previous section (i.e. concurrency and thread availability). This section, however, focuses on reducing execution overhead through a set of adaptive mechanisms that tailor provenance capture to specific support scenarios.

Table 2. Provenance captured at various levels of granularity

Granularity	**Provenance Captured**
Process Provenance (PP)	Application, user, application started by user, time started, parameters, time ended, and agents' and places' provenance (e.g., procedures, parameters, and returns specific to agents and places)
Simulation Provenance (SP)	PP, simulation logic (e.g., loops, conditionals, procedure calls, field access and assignment with start time, end time, and value)
Procedure Provenance (PrP)	SP, procedure name, calling procedure, time started, time ended, agent started the procedure or procedure was associated with place, hostname
Return Provenance (RP)	PrP, return values, procedure generated the return
Parameter Provenance	RP, parameters, parameter values (immutable data only), procedure that used a parameter

An adaptive variation of the ProvMASS approach allows users to specify the type and granularity of captured provenance (see Table 2), as well as operations and data of interest. Pervasive provenance recording is instrumented into framework operations, many of which involve low-level data manipulation. While these operations may be of little interest to the user, the provenance of some of these operations may be used to relate disconnected model procedures. Only the provenance of procedure call relationships is necessary to provide this connection, thus

demonstrating the need to adjust provenance granularity. To reduce provenance of framework-level operations, provenance granularity (process, simulation, procedure, return, parameter) can be adjusted to the levels shown in Table 2.

To verify specific behaviors of an agent model, users only require provenance of the interactions that make up those behaviors. Such interactions may be tied to specific iterations of the simulation logic or specific operations within one or more of those iterations. In addition to filtering through granularity adjustment, provenance capture can be further focused to these operations via a pause feature that starts and stops provenance capture, across all hosts. Contrastingly, support for more general uses of data provenance, such as boosting data integrity, require more thorough provenance capture.

To support pervasive provenance use cases (i.e. those that require a record of all activity) while still reducing performance overhead, a user can first capture the full provenance of a small simulation (e.g., simulations containing less iterations, agents, and space). Then, provenance filters for specific places and agents can be applied, while granularity is reduced to the simulation-level for framework operations. Such an approach supports experiment reproducibility by providing a sample MAM profile, fine-grain samples of agent interaction, and a coarse-grain record of configuration and simulation outcomes. Meanwhile, performance overhead is drastically reduced.

4.5 Challenge: Supporting Machine Unawareness

Machine unawareness is a MASS feature that allows developers to concentrate on modeling agents and places. Execution details of distributed memory management are hidden from the developer and kept from impacting the modeled behavior. Consequently, model execution can scale to include more physical resources, increasing performance without affecting the model specification. While low-level execution detail is abstracted away to support machine

unawareness, corresponding framework operations become opaque. However, representing resources at multiple layers of abstraction introduces difficulty in understanding model execution.

Developers may not realize the impact of these hidden execution details on model specification. For example, it may be difficult to tell exactly how agent management operations such as spawning and termination affect agents in a model edge case. For example, an agent located in a place on the global array border (i.e. a place that lacks a neighbor in one or more directions) may be modeled to migrate to a neighboring place to simulate propagation of biological entities. In this case, it may be difficult for the developer to anticipate the side effects of a failed migration.

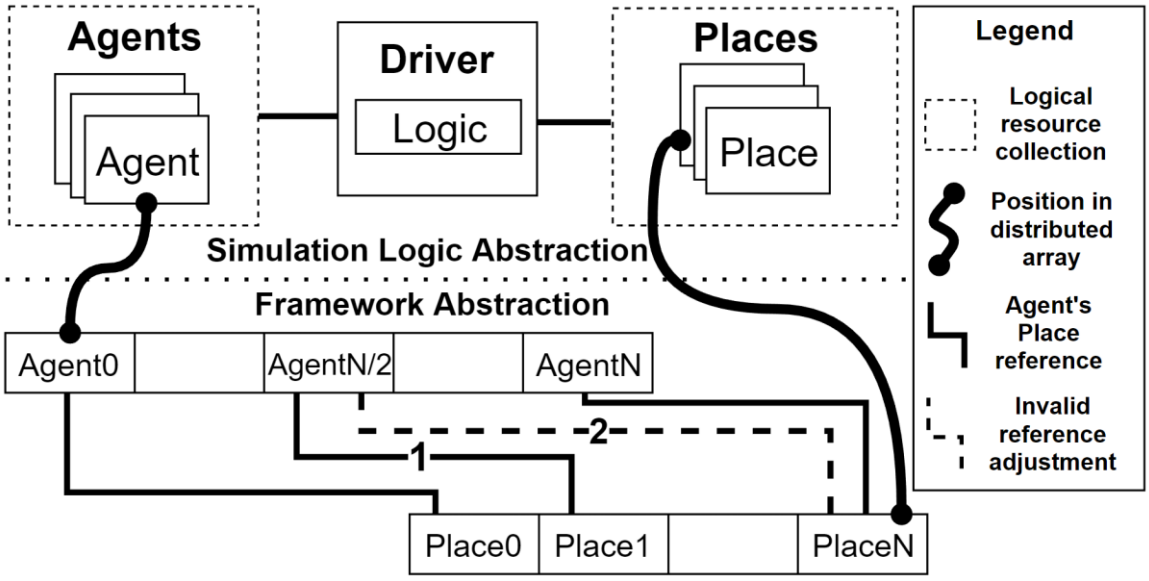

Figure 6. Cause of agent migration failure hidden from logical abstraction

To further illustrate the disconnect between layers of abstraction, consider a migration operation from the simulation logic abstraction corresponding to a collection of agents versus the same operation at the framework-level of abstraction, shown in Figure 6. In the figure, provenance from either layer of abstraction, alone, is insufficient to explain the success or failure of the

operation. First, the middle agent succeeds in migrating to place labeled as Place1 in the underlying array. Then, the agent fails to migrate to the place at the end of the array. The framework abstraction layer shows that the agent occupying the last element of the global agent array has already occupied the place at the last element of the global place array. However, this information is unavailable to procedures at the application-level of abstraction. Consequently, this information is also absent from the provenance record generated in those procedures.

ProvMASS provides a novel solution to bridge operations in multiple layers of abstraction while maintaining machine unawareness. As mentioned in previous sections, application resources pertaining to models (e.g., agents and places) are consistently identified and resource matching enables procedure call chaining (e.g., caller-callee invocation relationships). Together, these techniques produce provenance about framework operations with respect to model resources. For example, the following framework operations on a collection of agents combine to trace execution from the simulation logic to a migration activity: the framework procedure *callAll* calls another framework procedure, *callMethod*, which calls the procedure, *decideNewPosition*, at the application-level of abstraction, which calls the framework-level procedure, *migrate*. Provenance of this call chain is supported by the resource matching technique described in Section 4.2. Meanwhile, the provenance record also indicates that these procedures were started by the same agent, due to the consistent distributed resource identification technique described in Section 4.1. The *migrate* procedure in the example attempts to adjust the place index associated with the agent, based on parameters corresponding to the n-dimensional array of the places collection at the logical level of abstraction. Meanwhile, a subsequent agent management operation, *manageAll*, performs the manipulation required to see this operation through in the underlying distributed memory. These operations too, are described in the provenance record with respect to the same consistently

identified agent. Consequently, with the appropriate provenance granularity setting, it is possible to track any model resource at the application-level of abstraction with respect to all framework-level operations that handle that same resource.

Chapter 5. ARCHITECTURE

ProvMASS hybridizes the main-program and subroutines architectural style to work in a cluster (i.e. master-slave). Cluster systems gained popularity in the mid to late 1990s, when symmetric multi-processing started being mainstreamed to commodity hardware [4][5]. Such systems invert the host relationship of a layered client-server architectural style, to parallelize processing over multiple servers, coordinated to handle requests that originate from a single client application. While processing coordination resembles that of a grid, control is thoroughly centralized and tightly coupled, as participating hosts are typically more homogenous and connected through a local area network.

The main application class and core communicating components (i.e. master-slave coordination) of the MASS library serve as the main program of control-flow in the ProvMASS architecture. This aligns provenance capture with the functional properties of the MASS library and application driver while also helping to maintain existing nonfunctional system properties. Consequently, ProvMASS can support capture of distributed memory operations, while maintaining scalability, performance and usability of the MASS environment.

In this chapter, the ProvMASS architecture is presented from the perspectives of initial coordination, distributed provenance management, and provenance capture and storage. Relevant components and connectors shared by the MASS architecture are included to provide context for ProvMASS deployment and global coordination, but these mechanisms are described more fully in Chapter 2. Then, key component implementations are discussed in more detail. The Chapter closes with a discussion of the non-functional system properties; both those maintained within the MASS environment and those resulting from constraints on the organization of ProvMASS-specific components and connectors.

5.1 Distributed Provenance Capture Architecture

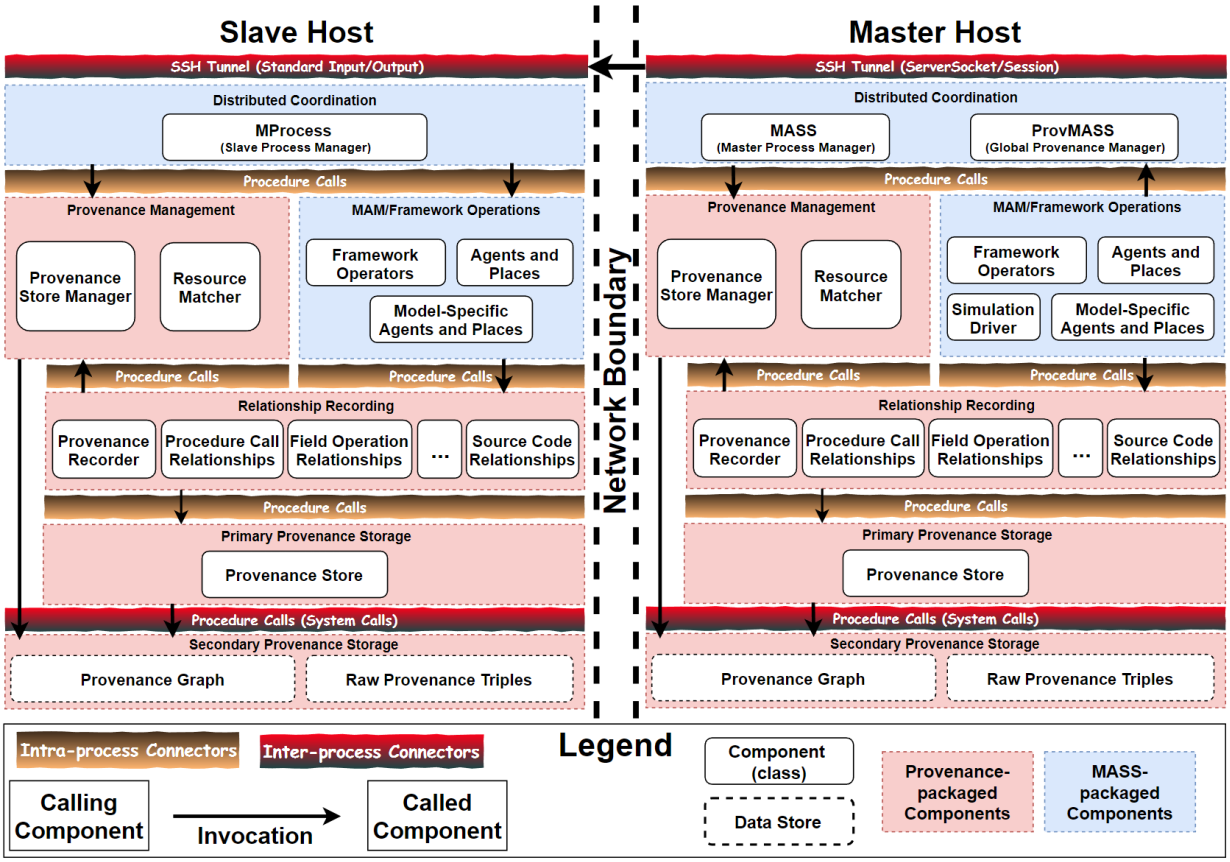

Figure 7. Architecture to distribute parallel provenance capture and storage

5.1.1 *Initialization and Distributed Coordination*

ProvMASS components and connectors are distributed as shown in Figure 7. For brevity, and to focus on structural considerations of provenance capture and storage, MASS coordination mechanisms are generalized into the MProcess and MASS components shown in the top portion of the slave and master hosts. All white boxes in the figure correspond to MASS and ProvMASS classes, apart from flat-files shown in secondary storage. The darker colored connectors (i.e. SSH tunnels and system calls) indicate the process boundary. Deployment is straightforward and includes one process per host. The process on the slave host corresponds to an MProcess (starting

in the MProcess component), whereas the process on the master host corresponds to the simulation application process (starting in the simulation driver component).

Processes are coordinated in the master-slave paradigm via SSH-tunneling. The master process on the right connects to one or more slave processes shown on the left through an SSH server socket. Future communication occurs through this connection on the master process, but is tunneled to the slave processes through standard input and output descriptors.

Prior to host coordination, ProvMASS is configured through the simulation driver (the entry point to the main process of the application). After the master and participating hosts are initialized, distributed provenance capture facilities are also coordinated through the MASS component. This enables utilization of existing barrier synchronization mechanisms, allowing changes in global provenance configuration to be applied consistently in the capture of MASS framework operations.

ProvMASS coordination begins by sending an initialization message from the MASS component to the MProcess components, triggering setup of the provenance infrastructure on remote hosts. In turn, ProvenanceStoreManager initialization is triggered. Configuration settings such as provenance buffer dimensions and provenance granularity settings are packaged within the ProvMASS initialization message. After the message is received by MProcess and the ProvenanceStoreManager is initialized, an instance of the ProvenanceStore component is provisioned for the main thread of the slave process.

5.1.2 *Management of Provenance Capture and Storage*

Provenance capture and storage is parallelized through the ProvenanceStore component shown in Figure 7. ProvenanceStores are issued by the ProvenanceStoreManager, which handles their instantiation, thread-mapping, and buffer provision. The ProvenanceStoreManager maps each thread to individual ProvenanceStore instances. Mapping occurs upon the first provenance capture

request from an unmapped thread of execution. Agent and Place operations occur through the MASS framework via MThreads. The main thread of the process may also request provenance capture. Meanwhile, many exchange requests utilize threads to send messages among host pairs, including both master and slave processes. These threads of execution invoke procedures on framework operator classes, agents, places, and the simulation driver.

Each ProvenanceStore generates its own file in secondary storage (i.e. the raw provenance triples file in Figure 7) which it writes to through an additional buffer whenever primary buffer space is full. However, all ProvenanceStores may be directed to collectively flush their buffers by the store manager. Distributed persistence of provenance storage and adjustment to global configuration settings is performed by ProvenanceStoreManager components across all hosts via the MASS component's communication facilities. However, these global operations are limited to initialization, finalization and configuration to maintain MASS performance and scalability.

Provenance capture and storage is insulated in the ProvenanceStore since each instance references an isolated portion of the provenance buffer. Meanwhile, writes to that buffer space are dedicated to specific threads of execution via thread mappings described above. Thus, provenance capture about the same shared resource can occur concurrently without resulting in interleaving provenance data. This helps to maintain efficiency of MASS framework operations. Provenance capture is enacted on provenance stores through procedures instrumented into operations of the original code. These relationships are discussed in the next section.

5.1.3 *Provenance Capture and Storage*

In contrast to infrequent coordination and distribution operations, most ProvMASS activity corresponds to provenance capture and subsequent storage. Provenance capture operations progress through layers of procedure calls, and eventually system calls to secondary storage shown

in Figure 7. For the most part, these layers are traversed sequentially. However, this is done concurrently as the provenance storage layer is parallelized as described in the previous section. This results in straightforward component relationships while supporting symmetric multiprocessing operations, which increase performance.

Provenance capture starts in pre-existing components (see MAM/framework operations in Figure 7) that are instrumented with calls to the ProvenanceRecorder class. These procedures invoke high-level provenance recording operations based on relationship generating procedures. Meanwhile, provenance recording operations within the ProvenanceRecorder class utilize configuration from the ProvenanceStoreManager and stage and use resource identifiers through the ResourceMatcher. These procedures also retrieve the appropriate ProvenanceStore for their current thread of execution from the ProvenanceStoreManager. Provenance relationships are built through combination of the appropriate sub procedures, also shown in the relationship operations section of Figure 7. These relationships are written to the ProvenanceStore. Provenance stores eventually persist this raw data to a file as provenance triples in the secondary storage layer. Finally, when all provenance capture operations have completed, each store manager uses the triples from these files to generate a directed provenance graph.

5.2 Key Component Implementations

Provenance Capture Instrumentation – Provenance capture instrumentation is provided by a pre-compiler that utilizes an open source Java parser called QDox [31], to gather information about source files, classes, fields, procedure signatures, parameters. Fields are used to generate corresponding provenance identifiers that are added to the source code of the respective classes. The remainder of these details are used to formulate two procedure calls to the ProvenanceRecorder class, discussed next, to capture provenance of the corresponding procedure.

Instructions within the procedure are also provided, together, as a single string of text. This text is further parsed to find the returned variable or expression. If the return is not associated with a local variable, the declaration and assignment of one is formulated based on the name of the procedure, the expression and the return type of the procedure. Within the procedure text, field accesses and assignments are identified and used to formulate capture procedures for the corresponding activities. Provenance capture procedure calls mentioned above, are inserted at the opening of the procedure scope and just before the procedure return in the procedure instruction text. Similarly, provenance capture procedure calls associated with the field accesses and assignments are injected just after the corresponding activities in the procedure text. The text is then used to replace the original procedure call.

While the component is usable in the current implementation, manual intervention is required in some instances. More specifically, fields and procedures of nested classes are accessed differently from outer classes in QDox. A related instrumentation bug prevents procedure- and field-injection into these classes, and is currently under investigation.

ProvenanceRecorder – The ProvenanceRecorder provides procedures utilized by the instrumented code described above and by the developer for adding custom provenance capture code (See Appendix A). Thus, it includes procedures with both lengthy and shorthand signatures. In addition to the ProvenanceRecorder interface, an example of the instrumented capture for procedure invocation is shown in Appendix A. Procedures with lengthy signatures wrap simpler procedures to filter agents and conditionalize the capture of provenance in accordance with the provenance granularity setting (see Section 4.4). Internal procedures are conditionally invoked based upon Boolean parameters. These internal procedures can be used directly by the developer and include

Boolean parameters that indicate whether to stage entity identifiers corresponding to returned values, for the extended resource matching features described in Section 4.4.

Capture of procedure call relationships is based on lower level relationships between agents and activities, between activities, and between activities and entities. These relationships are provided to the user, in addition to other common MAM related relationships. For example, field access generates an activity representing the operation and pairs it with the field entity.

Regardless of whether a provenance capture operation of the ProvenanceRecorder stems from instrumentation or developer use, each operation starts by retrieving the ProvenanceStore of the current executing thread from the ProvenanceStoreManager. The ProvenanceStoreManager is also contacted for granularity configuration settings and to check the status of provenance capture (i.e. whether the provenance capture is active, paused, or uninitialized). The ProvenanceStore and ProvenanceStoreManager are described in detail, next.

ProvenanceStore and ProvenanceStoreManager – At the lowest level, provenance capture occurs through instances of the ProvenanceStore component, each of which is managed by the ProvenanceStoreManager. Very few ProvenanceStore operations are initiated by the ProvenanceStoreManager, to boost concurrency. Aside from global persistence, which occurs when a simulation completes, initialization is the most substantial operation.

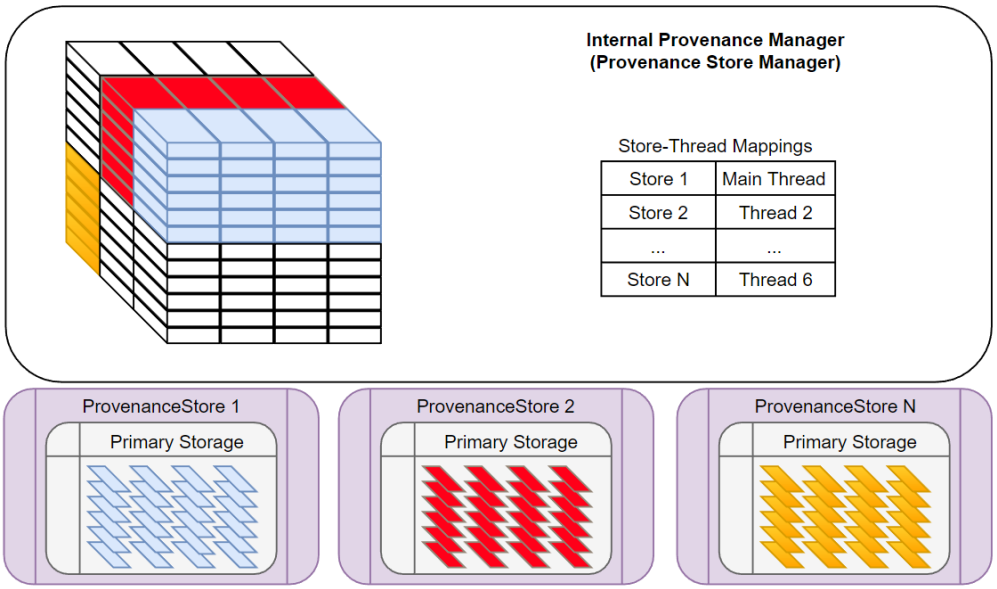

Figure 8. Buffering and thread safety of managed provenance storage

During initialization, a large buffer is provisioned, as shown within the top portion of Figure 8. When the ProvenanceRecorder requests the ProvenanceStore of the currently executing thread, the ProvenanceStoreManager checks the store-thread mapping shown in the figure, in an unsynchronized manner (i.e. read-only, without locking the resource). If the mapping is not found, the StoreManager first looks for previously allocated, released ProvenanceStores; those previously mapped to threads that have finished running. Otherwise, the ProvenanceStoreManager attempts to provision a new ProvenanceStore instance using the remaining buffer space. Thereafter, a new entity is added to the Store-Thread map, in a synchronized manner. Isolated buffer space and association of ProvenanceStore instances to specific threads of execution promote concurrent provenance recording.

5.3 NON-FUNCTIONAL PROPERTIES

Requirements of ProvMASS can be broken down into two parts: those that maintain existing requirements of the MASS environment and those that support provenance capture and use. First, this section presents ProvMASS feature attributes and architectural constraints that maintain performance, scalability, and usability of the MASS environment. This is followed with discussion of additional qualities corresponding to user and developer requirements, including provenance understandability and portability and that improve the quality of the provenance data, as well as software reusability, extensibility and adaptability that promote continued use of ProvMASS as the MASS library evolves over time.

5.3.1 *Maintaining MASS Environment Requirements*

Performance – The purpose of parallelizing MAMs with MASS is to ensure that long-running simulations (e.g., those requiring hours or days to complete) finish in a reasonable amount of time. To accomplish this, MASS employs symmetric multiprocessing to split the work among multiple processors (generally, processing cores) within each host process. Consequently, ProvMASS must provide the same concurrency for provenance capture and storage. This is solved for primary provenance storage with instances of thread-mapped ProvenanceStore components. The primary storage buffer size is configurable in multiple dimensions. This allows the developer to cause more data to be written to secondary storage together and more infrequently by dedicating more memory to provenance buffering, when possible. It also ensures that individual provenance triples fit into contiguous memory by allowing the line and character sizes to be configured independently. Constraining the global coordination of this primary provenance storage (e.g., to initialization and

system-wide persistence) increases the availability of communication facilities (e.g., Server Sockets and SSH tunneling between hosts).

Scalability – MASS provides horizontal scale (i.e. involving more hardware as opposed to more powerful and capacious hardware) for MAMs by coordinating a distributed memory over several hosts. ProvMASS supports this type of scalability by distributing provenance storage. ProvenanceStore instances located on each host maintain individual memory space. While they share the secondary storage access on each host, this access is split among hosts to reduce the overall impact of persisting provenance data.

Usability – MASS components exhibit location transparency in that they are shielded from the details of their deployment. This improves usability by sparing the developer from making implementation considerations based on those details. Like MASS, ProvMASS assumes little about the specific environment in which the provenance is captured. Instead, it interfaces with the MASS coordination components to initiate distributed provenance tasks. While opacity of interactions with the hardware environment allows MASS to scale without imposing the need to modify simulated models, it also introduces the need to write specialized agent and place procedures to gather this information (e.g., in network system models or for debugging). It may also impede investigation of unexpected errors associated with an execution environment (e.g., hardware failure in a specific host). ProvMASS circumvents such caveats and increases usability by providing the user with provenance that relates framework operations to the location of their execution.

5.3.2 *Provenance-Related Properties*

Reusability – Component reuse is an important property of the ProvMASS system, as high-level model relationships vary. For example, spatial models have a concept of a neighbor, while agent models may only have a concept of themselves (i.e. state and behaviors) and the space they occupy (i.e. place instance). Provenance to describe an operation involving read-only neighbor data (i.e. shadow-space) must somehow differentiate that data from that of the actual data of the neighboring place. Meanwhile, provenance capture for spatial operations initiated by an agent must include the ability to discern the agent, individually. While these relationships may be nuanced in the model output, the underlying relationships in the provenance model are straightforward and constant. Further, these examples are generalizations, whereas higher level relationships specific to the domain of the model may be unique. Consequently, the methods of constructing low-level relationships must be reusable.

ProvMASS provides reusable provenance generation components allowing developers to utilize easy to compose generic provenance relationships that are analogous to the relationships observed in real-world systems. Additionally, generic provenance constructs to describe common program execution relationships are also available, but these too are composed of components that generate provenance of intermediate relationships. Consequently, developers need not create one-off solutions (e.g., at the framework-level of abstraction) that require an understanding of the inner workings of MASS distributed execution and related distributed memory management. Nor must they directly use the provenance stores to generate relational provenance themselves; which would require a detailed understanding of the provenance model (e.g., knowledge of domain and range of classes and relationships of provenance resources) and how the model relates to intermediate phenomena.

Extensibility – Several interfaces are provided to shield developers from underlying ProvMASS implementation. First, the ProvMASS component allows developers to make configuration changes without directly contacting the ProvenanceStoreManager. Such decoupling prevents modification of the underlying management implementation from affecting user applications. Second, the provenance recorder provides an interface for recording common high-level relationships in corresponding source code. This interface also provides access to low-level relationships that can be used in isolation or combined to form more complex provenance descriptions. These interfaces remove the need for the developer to handle direct provenance capture. Finally, secondary provenance storage can easily be extended to different mediums as primary storage is buffered.

Adaptability – Simple coordination, and a lack of assumptions about each host, allows ProvMASS to adapt to changing requirements that may be introduced by future versions of the MASS library. Intra-host provenance capture mechanisms are disconnected from those of the MASS framework because of source code instrumentation. All provenance capture occurs through recording procedures that are injected into procedures of specified classes prior to compilation. Consequently, only fundamental changes to inter-host coordination break this adaptability.

Portable Provenance – While ProvMASS is written in Java, making it portable to different environments, portability of the provenance data is also important. For systems that generate heterogeneous provenance data, such as in data lakes, standard representations can help reduce the complexity of integrating such provenance into a common model [35]. MAM relationships

captured by ProvMASS are specified with concepts from the W3C PROV of family of documents [30]. More specifically, these relationships are specified with ontological concepts from the PROV Ontology [29]. Further, the provenance graph is formatted as a turtle file, making it easily interpretable on various operating systems.

Understandable Provenance – Well established provenance representations form a dichotomy between inversion and annotation in which inversions provide a compact mechanism to determine data derivation relationships, while annotations are richer, including additional context [33]. Provenance generated by ProvMASS provides such relationships and contextual information in a semantically rich representation. MAM relationships that correspond to users' understanding of the model and underlying execution environments can help in supporting analysis. This understandability is provided through fine provenance granularity paired with actual procedure and variable names, as well as activity locale corresponding to host-names. Not only does understandable provenance support use cases described in Chapter 3, but it also supports usability of MASS, as described above.

Chapter 6. EVALUATION

Adequacy of the ProvMASS approach is related, in part, to supporting use cases described in Chapter 3. This chapter provides an assessment of support for these use cases with respect to queries over actual provenance data generated for the SugarScape and RandomWalk models. Since approach adoption is highly dependent on upholding key properties of the MASS environment, scalability and performance also play a role in assessing the adequacy of this support. Consequently, performance measures are provided and interpreted with respect to capture of the types of provenance that support these use cases. These evaluation mechanisms answer research questions presented in Section 1.6, contributing to the current body of provenance capture research. First, details of the provenance model are presented to provide a useful discussion and interpretation of queries and related performance measures.

6.1 PROVENANCE MODEL

The provenance model used to describe multi-agent model (MAM) execution and specification is expressed with concepts from the W3C PROV family of documents [30]. Provenance relationships correspond to procedure call semantics and object-oriented resource relationships and are specified with resource description framework (RDF) triples. The triples express the relationships according to semantics defined in the PROV ontology [29].

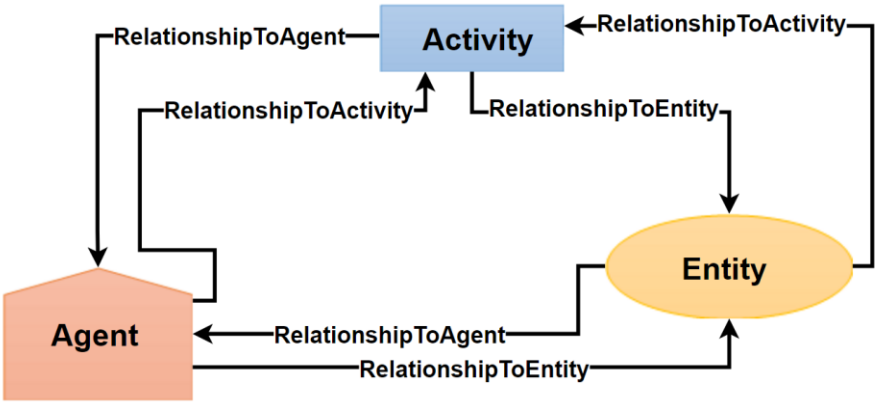

Figure 9. Directed provenance graph

The directed graph, shown in Figure 9, is composed of vertices representing agents, activities, and entities (provenance classes) and edges representing the relationships between them (provenance properties). Specialized provenance classes exist, but generally do not modify these relationships. An exception is the collection class, which is composed of entities that may have direct edges, in addition to edges between the collection and other vertices. Triples are added to provenance stores and periodically flushed to secondary storage. In this form, each triple represents two vertices connected by an edge; made possible by capturing data and activity relationships *in situ* via resource matching and consistent inter-process resource identification, as discussed in Section 4.3. Later, the triple resources are connected to form a graph of provenance resources representing MAM application concepts.

Table 3. Mapping between MAM concepts and PROV Ontology starting point classes

Provenance starting point classes	**Concepts in MAM applications**
Agent	User, simulator, agents (i.e. multi-agent entities)
Activity	Procedure invocations, field operations (e.g., access and assignment)
Entity	Fields, parameters, places, returns, messages; instructions, arrays, conditionals and loops (in specification of simulation logic)
value	Description of the content or state of any above classes; identifiers tying instructions to corresponding activities (in simulation logic and specification)

The classes, restrictions and properties of the PROV Ontology can be used to define systems, including relationships within the system. ProvMASS applies these concepts to MAM execution and specification. The vertices in Figure 9 correspond to starting point classes outlined in Table 3. These classes form the bases for describing resources related to MAMs and their execution. An agent is an individual that carries out activities. Activities are dynamic occurrences that involve the use of, and/or result in generation of, entities. Entities are things that may be attributed to an agent who carried out an activity. MAM-related concepts are categorized to provide the closest analogy to corresponding PROV concepts. The purpose of this categorization centers on the range and domain of relationships between classes. For instance, the *used* relationship has a domain that includes activities, but not entities, and a range that includes entities, not activities. In other words, an activity uses an entity, but an entity does not use an activity.

PROV Ontology concepts are mapped to MAM software specification and execution in Table 3. An agent may define the user who started the application, the application itself, or an individual in the modeled system. Activities also correspond solely to execution concepts, including field access and assignment operations, as well as procedure invocations. Entities, on the other hand are either tied to data in memory, or to instructions and other expressions in specification of the

application driver (i.e. the procedures that start the MASS library and orchestrate the workflow of a simulation). In general, the value property describes the state of the respective class. However, the value relationship is used to describe the connection between a static simulation instruction in the application driver and the corresponding activity in execution. More specifically, the value of the instruction corresponds exactly to the resource identifier of the activity, allowing joined queries to bridge the software to corresponding execution.

Table 4. Mapping between MAM concepts and specific PROV Ontology classes

Provenance classes	**MAM concepts**
Agent	User
SoftwareAgent	Simulator, agents (i.e. multi-agent entities)
Entity	Fields, parameters, places, returns; source code instruction (in specification of simulation logic)
Collection	Places, messages; arrays, conditionals and loops (in specification of simulation logic)
Activity	Procedure invocations, field operations (e.g., access and assignment)
value	Description of the content or state of any above classes; identifiers tying instructions to corresponding activities (in specification of simulation logic)

Some MAM concepts are better defined by expanded classes. These specializations are shown in Table 4. While the user is still an agent, some of the agents are running software and are therefore specified as SoftwareAgents. Meanwhile, some entities are composed of other entities. As previously mentioned, these collections can have relationships to other provenance resources while their members can also have these relationships. These relationships are defined by properties of the PROV ontology. While the specification differentiates starting point properties from expanded

properties, they are described together here, as the difference in specificity is not necessarily analogous in corresponding MAM relationships.

Table 5. Mapping between MAM relationships and PROV Ontology properties

Provenance property	**MAM relationship**
influenced	A called procedure was part of the calling procedure; A field access or assignment operation was part of a procedure
wasInfluencedBy	A portion of a calling procedure is represented by a called procedure; A procedure involved a field access or assignment operation
generated	A procedure provided a return; a field assignment generated a field value
wasStartedBy	A procedure was started by an agent
value	Any parameter, return, field accessed, or field assignment had a value (if immutable) or reference hashCode (if mutable)
startedAtTime	A procedure was started at a system-time (in nanoseconds)
endedAtTime	A procedure ended (just before return) at a system-time (in nanoseconds)
alternateOf	Procedure parameters and returns reference distributed application resources (e.g., agents, places, and messages); procedure parameters reference procedure call arguments; local variables reference returns
atLocation	A procedure was invoked at a host with the specified name (or internet protocol address)
wasAttributedTo	When a new datum is generated through local variable assignment or field assignment, it was attributed to the agent

Many of the relationships outlined in Table 5 correspond to procedure call semantics. The influenced and wasInfluencedBy relationships correspond to chains of procedure invocation. More specifically, the caller was influenced by the callee, while the callee influenced the caller. Meanwhile, entity generation and use correspond to procedure input and output, respectively. Parameters are used in the invocation of procedures, while the invocation results in the generation of a returned entity. Location transparency is also important; thus, every procedure invocation is tied to a hostname through the atLocation property.

Since individuals of the multi-agent models are treated as autonomous decision makers, invocation of any agent procedure, or stemming from an agent procedure (e.g., a place procedure invoked by the agent) is specified as being started by the agent. This enables simple queries indicating all procedures a given agent was involved in or all the agents that started a given procedure. Similarly, the alternateOf property bridges related data. This includes methods to consistently identify distributed shared resources described in Section 4.1 and methods to match references across procedure boundaries described in Section 4.2. Consistently identified model resources (e.g., spatial data and agents) and other distributed data (i.e. messages) are described as alternates of the corresponding local variables (e.g., parameters and returns) and *vice versa*. Similarly, parameters of a callee invocation are alternates of arguments specified in the caller and *vice versa.* The same relationships exist between references returned from a callee and local variables assigned that reference in the caller.

The ProvMASS approach generally relies on causal ordering, rather than temporal, the procedure start time and end time are important considerations when relating disconnected operations via consistent entity identifiers. Such relationships can be applied directly in queries about data relationships. An example of such a direct relationship involves finding all procedures invoked by a specific agent, possibly invoked on many hosts and stemming from various instructions in the simulation driver. Conversely, composite relationships are expressed by joining queries. An example of queries indicating composite relationships is one that finds all procedures that directly preceded migration activities of all agents. The most complex queries rely on the full causal relationship from an instruction in the application driver to chains of activity influence that essentially result in a full stack trace; these are referred to as forward chains, but backward chaining is also possible.

6.2 PROVENANCE QUERIES

Three types of provenance to support MAM research and development, introduced in Section 2.3, form the basis for use cases presented in Chapter 3. These types correspond to provenance about the simulation history, model development, and execution environment. This section presents queries that support corresponding use cases. Overlap exists between support for these use cases. For example, support for reproducing phenomena exhibited in a system modeled by a fellow researcher depends upon an understanding of agent interactions. Meanwhile, a full understanding of agent behavior requires context related to the execution environment.

The remainder of this section presents queries and results, from actual MAM execution, that demonstrate the adequacy of the provenance collected with the ProvMASS approach to support these use cases, answering two of the research questions described in Section 1.6. Results are displayed in related figures depicting the relevant provenance graph, but 128-bit provenance resource identifiers have been truncated to save space.

Answers to two research questions, from Section 1.5, are provided in this study:

- **RQ1** Can provenance of shared resources be captured in a distributed memory?
- **RQ2** Can the full context of individual agent behaviors (e.g., coordination and state of agents and spatial data) be traced to simulation logic?

6.2.1 *UC1: Individual Agent Behavior in SugarScape*

ProvMASS captures provenance within procedures of the agent and place instances that compose a multi-agent model. These procedures are specified at the logical level of abstraction and combine to form agent behaviors. Meanwhile, interactions between agents or between agents and their environment may be carried out through invocation of these procedures, field operations, or a

combination of the two. Provenance of these activities can help explain individual decisions, which make up agent behaviors.

Agent behavior in SugarScape revolves around survival. Intuitively, activities resulting in an agent's death may play a role in survival. The context surrounding these activities can be used to describe the agent behavior more clearly. Specifically, answers to questions in this use case can assist in understanding why an agent exhibited a specific behavior. As agents are shared resources in MASS distributed memory, provenance of these behaviors can help answer the first research question presented in Section 1.5:

RQ1 Can provenance of shared resources be captured in a distributed memory?

a) What activity resulted in an agent's death?

This question can be answered using only agent-related provenance. Simulation-level provenance granularity is sufficient to produce this type of provenance (see Table 3). The kill procedure stages an agent's termination, which takes place during the following management operation on the collection of agents. The activity of interest is not the procedure invocation that directly invoked kill. Rather, the focus is on the cause of the agent's death. SugarScape agents are terminated when they run out of food (i.e. energy to keep moving). Thus, the activity of interest is that which led the agent to a state in which the kill procedure was invoked. This question can be answered with a series of filtered queries.

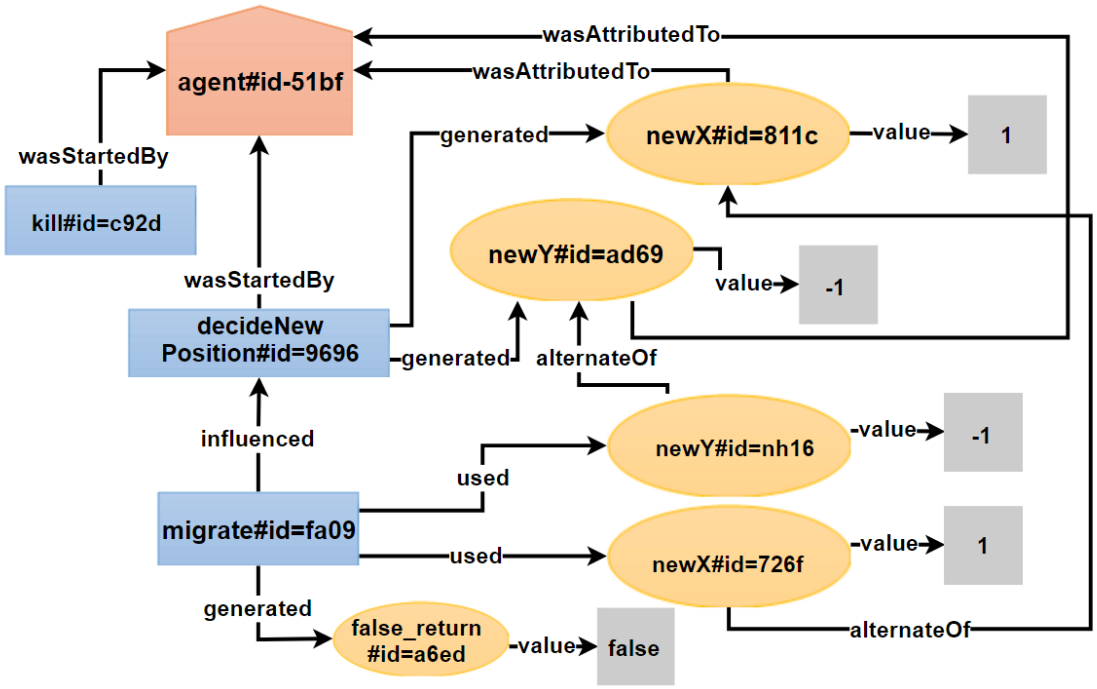

Figure 10. Condensed provenance of individual agent behavior

In the following queries, references to specific provenance resources correspond to Figure 10. First, a query on all agent activities is filtered to identify an invocation of the kill procedure (kill#id=c92d). From these results, an individual agent can be singled out by filtering the first result of a query on the wasStartedBy relationship between agents and activities (see Table 3). Since agents move around looking for food, migration decisions are of interest. As discussed in the previous section, procedure invocation relationships include the time at which the procedure activity was started. Leveraging the wasStartedBy relationship between procedure invocations and agents, all migrate activities can be queried. Thereafter, the results can be filtered to startedAtTime values less than that of the kill procedure invocation.

There are two options to determine the migrate procedure invocation that immediately proceeded invocation of the kill procedure. The first option is to rely on temporal ordering. Since temporal ordering can be relied upon within the same host process when there is sufficient delay

in between invocations, the migrate activities can be ordered based on their startedAtTime. Since these activities occurred prior to the kill activity, the most recent activity is the migrate procedure invocation of interest. The second option is to backward chain activity influence. When such ordering cannot be relied upon, it is still possible to chain queries on the influence relationship between activities, to follow the procedure call chain back to the simulation driver and then back to the last migrate activity. However, it is preferable to avoid forward and backward chaining, as it can be slow or otherwise, requires substantial memory. Either option results in the migrate procedure invocation (migrate#id=fa09) that directly preceded the kill procedure invocation of interest. Finally, a query can be made on the *influenced* relationship between the migrate activity and its calling procedure (decideNewPosition#id=9696). This is the activity that led to the agent's death.

b) What conditions (activities and entities) cause an agent (agent#id=51bf) to die?

The queries from the previous question resulted in a decideNewPosition activity (decideNewPosition#id=9696) that preceded the agent's death. At this point in the query process, it is unknown whether this behavioral activity had direct influence over the agent's death. However, learning the conditions that led to the migrate activity will resolve this question.

Once again queries are issued about transpired events in reverse chronological order. The migrate activity represents a procedure invocation and conditions of the activity correspond to parameters. A query on entities used by the migrate activity results in two destination coordinates (newX#id=726f and newY#id=nh16). A joined query reveals that the coordinate values were 1 and -1. This result implies that the agent failed to migrate, as the destination (1, -1) lies outside of the logical coordinate boundaries of the places collection. This is confirmed with a query to determine the return generated by the migrate activity; a false Boolean value. Further investigation

reveals that out of the three possible calls to the migrate procedure, random neighboring coordinates were chosen. Further, the process of choosing neighboring coordinates resulted in a logical error.

The queries presented in this case demonstrate the ability to single out an agent to relate one among many concurrent chains of procedure invocation. In turn, query results indicate that ProvMASS can capture provenance of shared resourced, partially answering the second research question presented in section 1.5:

RQ1 Can provenance of shared resources be captured in a distributed memory?

Partial Answer: This study shows that provenance of shared resources can be captured.

6.2.2 *UC2: Simulation Specification and Execution in SugarScape*

This use case demonstrates the ability to tie execution provenance to simulation source code, reconciling segregated layers of model abstraction in MASS applications. Connections between these layers and intermediate framework operations serve to answer the second research question presented in Section 1.5:

RQ2 Can the full context of individual agent behaviors (e.g., coordination and state of agents and spatial data) be traced to simulation logic?

a) What activities and entities were involved in generating a given simulation output?

This question can be answered by querying simulation-related provenance; produced at simulation-level granularity (see Table 3). An answer to this question involves provenance resources that correspond to instructions in the simulation logic, activities that correspond to execution of those instructions, and the corresponding framework activities. Control structures of the simulation logic are represented as collections, with members represented as static entities that

correspond to procedure calls. These entities correspond to instructions and have a value property that matches the identifier of the analogous execution activity (i.e. procedure invocation).

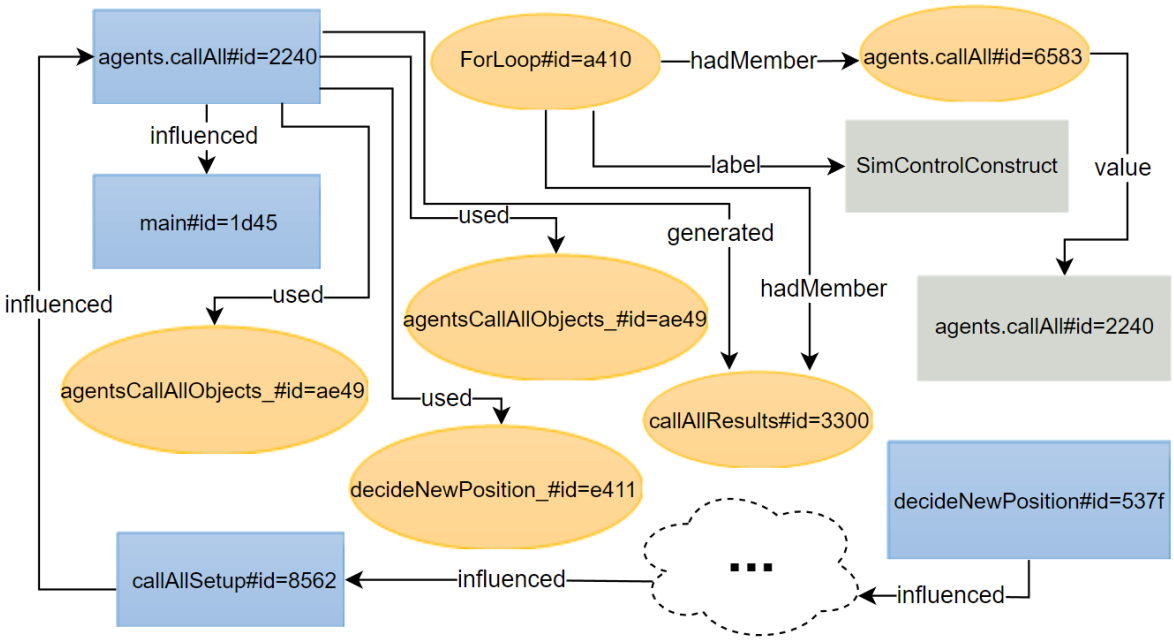

Figure 11. Provenance of simulation logic tied to corresponding execution

The following queries on provenance resources shown in Figure 11, can be used to investigate the activities and entities associated with simulation output. The first query involves a control structure to coordinate simulation iterations, represented by a for-loop collection (ForLoop#id=a410). Joined to the first query, a query on the hadMember property reveals an instruction that corresponds to a procedure call with assignment of simulation output (agents.callAll#id=6583). Another query obtains the member's value which identifies the procedure invocation (agents.callAll#id=2240). This query can be joined with two other queries. The first join reveals the entities (i.e. parameters) used in the invocation (decideNewPosition_#id=e411 and agentsCallAllObjects#id=ae49). The second join operation

reveals the simulation output that was generated by the invocation (callAllResult#id=3300). This resource corresponds with an entity member of the for-loop collection. Consequently, it is possible to form a backward trace from the simulation output in the source code, in addition to the forward trace described above. Together, the query results and join-relationships indicate all the activities and entities that were directly involved in generating the simulation output.

b) What agent-interactions were involved in generating a given simulation output?

The simulation output and corresponding framework activities, found in the previous queries, can also be tied to specific agent interactions. However, these relationships depend on more detailed provenance about framework activities (i.e. all framework operations – not only those accessible to the simulation driver through public interfaces). This question can be answered by querying provenance about framework operations; produced at procedure-level granularity (see Table 3). An answer to this question begins with a query to find procedures that influenced the framework operation found while answering the previous question (agents.callAll#id=2240). This results in a chain of activity dependencies (callAllSetup#id=8562, etc.) ending with an agent activity (decideNewPosition#id=537f). Provision of these procedure- and dependency-relationships indicates an answer to the second research question discussed in Section 1.6:

> **RQ2** Can the full context of individual agent behaviors (e.g., coordination and state of agents and spatial data) be traced to simulation logic?
>
> **Answer: Yes,** this study shows that agent interactions can be traced to simulation source code through framework operations.

6.2.3 *UC3: Distributed Execution in RandomWalk*

The ability to bridge distributed MAM execution facilitates restoration of disconnected logic. While this can often be accomplished by simply ordering disconnected operations, direct dependency relationships are sometimes required. Queries in the first use case demonstrated the ability to inspect agent behavior by leveraging procedure call relationships. Those in the previous use-case demonstrate the ability to bridge simulation logic to the interactions that make up those agent behaviors through framework operations. Answers to questions in this use case fill the causal gap between the previous two by connecting agent procedure invocation to corresponding simulation instructions through framework procedure invocations across all hosts, demonstrating the ability to provide holistic model provenance.

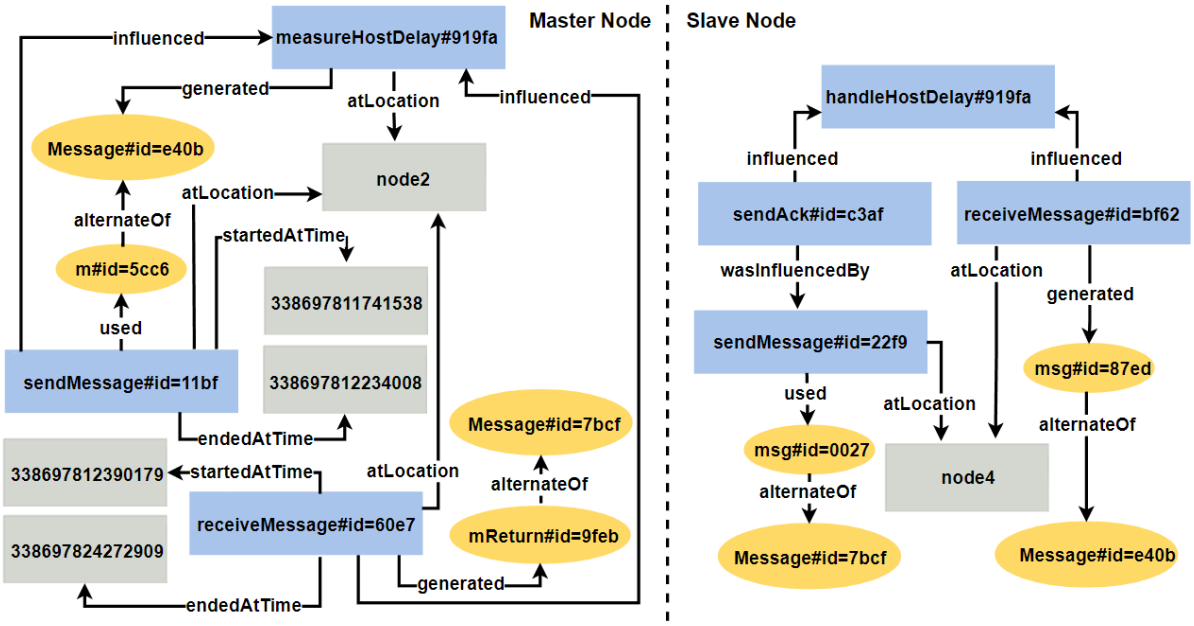

Figure 12. Provenance of an operation to measure message delivery delay

a) What related operations across various hosts took place during a simulation?

This question can be answered by querying provenance about parameters, returns and related procedures; produced at parameter-level provenance granularity (see Table 3). Partial provenance of an operation that measures message delivery delay between the MASS master host process and remote slave processes is shown in Figure 12. The figure shows the provenance of message passing operations between the master node and just one slave node; in fact, the operation involves 12 host pairs, but provenance of one pair suffices to illustrate the ability to answer this question.

Normal MASS communication to coordinate remote hosts occurs asynchronously, first sending messages to all hosts, then awaiting reply from all hosts in barrier synchronization. However, this procedure communicates with remote hosts in sequence to enable comparison of inter-host message passing delay. More specifically, the procedure measures the elapsed time between message passing from the master host to a slave and receiving the corresponding

acknowledgment from that slave, then compares results to find the least latent connection. Without directly using provenance of these results, consistent inter-process resource identification can be used to calculate the same measurement. The first step in this process is to identify the distributed operations related to this task.

For this operation, each remote host is sent a request message. Upon receiving the request, the host sends an acknowledgment message. A query on sendMessage activities that influenced the measureHostDelay activity (measureHostDelay#id=919fa) results in all communication procedure invocations originating from the local host (e.g., sendMessage#id=11bf and receiveMessage#id=60e7). Joined with this, a query on entities used by these activities is joined with a query on the alternateOf property, resulting in all sent messages (e.g., Message#id=e40b). A subsequent query inverts the alternateOf relationship to also retrieve all associated procedure returns (e.g., msg#id=87ed). The first alternateOf query results in messages (consistently identified inter-process entities), while the second alternateOf query results in local variables (entities specifically tied to procedure invocation). These returns correspond with invocations of receiveMessage procedures on the remote hosts (e.g., receiveMessage#id=bf62). Together with invocations of the sendMessage procedure, these activities constitute all message passing operations originating from the measure host delay operation, across various hosts.

To obtain activities related to the measure host delay operation that originate from remote hosts, queries can be made on the receiveMessage activities to obtain subsequent procedure invocations (e.g., handleHostDelay#919fa). The remaining queries match those described above to obtain procedures originating from the measure host delay activity. Similar queries can be made to match acknowledgment messages (e.g., Message#id=7bcf) between the remote host and the local hosts using the alternativeOf property. Inversion of these entity-entity queries results in

returns (e.g., mReturn#id=9feb) of receiveMessage invocations (e.g., receiveMessage#id=60e7) to the measure host delay activity. The generated property can be utilized on a query for related receiveMessage activities. Queries to answer this question result in four sets of message-passing activities for each slave process, representing all operations related to the measure host delay activity across various hosts.

b) What hosts are experiencing the most lag?

In addition to the message passing operations related to the measureHostDelay activity, queries to answer this question also provide all related message entities. By utilizing consistent message identifiers and the alternateOf relationship, matching pairs of send and receive activities can be found. Following the influence property and the acknowledgment message back to the measure host delay activity provides pairs of sendMessage and receiveMessage invocations. These pairs directly influence the measureHostDelay activity and represent a round-trip message passing operation. Queries on the startedAtTime property and endedAtTime property can be joined to find the duration of each round-trip operation. The fastest and slowest connections between remote hosts and the local host are indicated by ordering query results.

The queries presented in this use case demonstrate the ability to capture coherent provenance about remote operations in distributed memory. Since messages are used to transmit agents and spatial data from one host to another, this demonstrates the ability to capture provenance about distributed operations over shared resources. These query results indicate that provenance of distributed memory operations can be captured. Together with use case 1, these results provide an answer to the first research question presented in Section 1.5:

RQ1 Can provenance of shared resources be captured in a distributed memory?

Partial Answer: Study shows that provenance of distributed resources can be captured.

Answer: Yes, in addition to capturing shared resource provenance, that provenance can be related across various hosts.

6.3 PERFORMANCE MEASURES

To assess the performance overhead of the ProvMASS approach and adaptive capture features, the run time of SugarScape and RandomWalk simulations are compared for executions with and without provenance capture. Several factors are assessed, including simulation profile (i.e. number of places, agents and simulation iterations), provenance granularity, and provenance capture filtering. These studies serve to answer the third research question from Section 1.5:

RQ3 Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?

Table 6. Computing environment configuration for performance measures

Environment Property	**Environment Configuration**
Central processing unit	1.6 GHz 4-core Intel i7
Random access memory	16 gigabytes
Network type and speed	1 gigabit per second shared local area network
Operating system	Ubuntu version 16.04.1
Java virtual machine configuration	Version 1.8.0, 64-bit data model, 9 gigabyte heap size

In the following performance comparisons, simulations are conducted in a distributed setting, characterized by similarly configured hosts outlined in Table 6; each with a 1.6 GHz 4-core Intel i7 CPUs, 16 GB of RAM, all connected by a 1 Gbps LAN, and running version 16.04.1 of the Ubuntu operating system. The Java Virtual Machine was configured in 64-bit mode with a 9GB heap size for all host processes. The connection time from the master to other hosts is not measured

as it does not reflect on efficiency of the approach and unnecessarily increases execution time variation.

6.3.1 *Agents-Scale Comparisons*

The following comparisons involve an increasing number of places and agents to assess the factor of increased execution duration while applying an adaptive capture approach. Provenance of a subset of all agent's interactions is combined with that of the simulation driver, ignoring agent management and other detailed framework operations. This comparison measures the outcome of limiting provenance capture to answer questions posed in use case 1, of the previous evaluation, except from a smaller pool of agents and filtering irrelevant operations.

a) Study Design

Table 7. Model and provenance configurations for agents-scale comparisons

Configuration Property	Configuration Setting
Provenance granularity	Simulation-level (includes agent-procedures, corresponding parameters and returns; see Table 2 for more)
Agent filter	Range by ID filter 10 agents per host (reported results for execution on 12 host)
Pause feature	Capture is paused for all but 4 simulation operations (2 operations per iteration on all agents, for 2 iterations)
Provenance buffer	~2 GB total reserved / ~256 MB used; 2048 maximum stores per host (8 used), 2^{19} characters per store (2^{22} characters used out of 2^{30} total buffered characters)
Number of places	Scale from 16,384 to 65,536 to 262,144
Number of agents (RandomWalk)	Scale from 16,384 to 65,536 to 262,144
Number of agents (SugarScape)	Scale from 640 to 1280 to 2560
Simulation iterations	25
Total Simulation Operations	77 in RandomWalk / 103 in SugarScape
Runs averaged	15

The provenance capture configuration outlined in Table 7 for these comparisons, is as follows. Provenance granularity was set to the simulation-level (i.e. no procedure invocations were captured in the framework). Procedures, parameters and returns were gathered for 10 agents per host process. The additional intra-procedure provenance described in use case 1 was also captured for SugarScape, including field access and assignment operations. The pause feature was also applied to narrow provenance capture to two operations in each of two iterations.

Configuration of multi-agent models for this comparison is as follows. In simulation runs for RandomWalk the number of agents was matched to the number of places which scaled from 16,384, to 65,536 and finally, 262,144. SugarScape agent mapping is somewhat random. While the places configuration matches RandomWalk, the number of agents scaled from 640, to 1280

and finally, 2560. The remainder of the configuration parameters were identical for both models. The number of simulation iterations was set to 25. With 3 simulation operations per iteration and instantiation of agent and place collections, a total of 77 simulation operations were executed from the application driver on the master host. Meanwhile, SugarScape also includes initialization for neighbor exchange and one exchange operation per iteration, bringing the total count of simulation operations to 103. Comparisons for each model were averaged over 15 runs with provenance capture turn on and then off.

b) Study Results

Figure 13. RandomWalk simulations scaling number of places and agents on 12 hosts

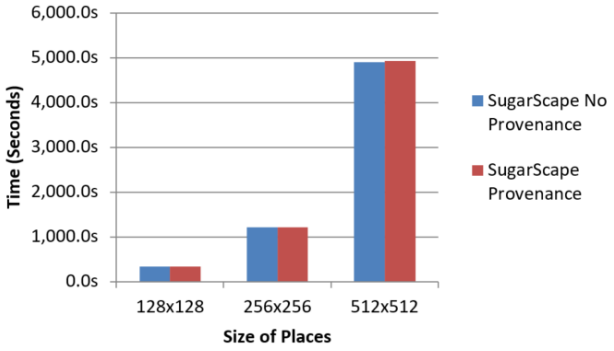

Figure 14. SugarScape simulations scaling number of places and agents on 12 hosts

Figures 13 and 14 show increased duration of execution attributed to provenance capture for the RandomWalk and SugarScape models, respectively. The factor of increased execution for SugarScape was minimal (under 4.1%, 3.2%, and 2% decreasing as the model size increases in the x-axis). RandomWalk, on the other hand, presents more variation and higher factor of increase in duration. Trends in these comparisons indicate execution overhead increases tapered as the model size increased; an expected result, as the amount of provenance captured remains the same. This demonstrates capability to throttle performance overhead through adaptive capture.

c) Study Discussion

The RandomWalk model is not as complex as SugarScape. Moreover, it is less computationally intensive and more network intensive. While more time spent waiting for acknowledgment messages translates to a lower percentage of time spent waiting for provenance capture, RandomWalk is simple, involving only agent procedure invocation and management. Conversely, SugarScape involves more framework-supported operations such as data exchange between neighbors; more operations that are not captured due to adjustment of provenance granularity and application of the pause feature. Consequently, the overhead of capturing agent behavior in SugarScape is relatively low compared to RandomWalk, for this comparison.

6.3.2 *Pause Provenance Comparison*

The following is a combined comparison which assesses the basic profile of overhead across all granularity settings while also comparing the same simulation runs with application of the pause feature. The comparison was conducted based upon the presumption that full provenance capture at fine granularity results in excessive performance cost. This cost also corresponds with the complexity and duration of the simulation. The purpose of applying the pause feature is to minimize overhead increase as granularity becomes finer.

a) Study Design

Table 8. Model and provenance configurations for pause provenance comparisons

Configuration Property	Configuration Setting
Provenance granularity	Range from process-level (coarse) to parameter-level (fine) (all granularity levels include agent-procedures, corresponding parameters and returns; see Table 2 for more)
Agent filter	All Filter (10 agents per host on 16 hosts)
Pause feature (when used)	Capture is paused for all but 4 simulation operations in runs that include use of the pause feature (2 operations per iteration on all agents, for 2 iterations)
Provenance buffer	2 GB total reserved / 256 MB used; 2048 maximum stores per host (8 used), 2^{19} characters per store (2^{22} characters used out of 2^{30} total buffered characters)
Number of places	1024
Number of agents	160
Simulation iterations	10
Total Simulation Operations	32 in RandomWalk / 43 in SugarScape
Runs averaged	20

Configuration of the multi-agent models and provenance capture for these comparisons is outlined in Table 8. Simulation runs for both models were configured as follows: the number of places was set to 1024, number of agents was set to 160 and the number of iterations was set to 10. The lowest number of agents for SugarScape was 158 and the highest was 161, despite randomness in agent mapping, making simulation configurations very similar for all runs. Note that this is not the case in the previous comparison, where variation in model configuration was intended to emphasize differences in processing profile. Conversely, similarity in model configuration is intended to emphasize limitations of adaptive capture, building on model differences noted in the previous comparison. Another substantial difference is the amount of iterations.

In the previous comparison, the pause feature made a large impact, reducing capture of simulation operations (including place exchanges and agent management) from 103 to just 4 in SugarScape. Meanwhile, this comparison only narrowed total simulation operations from 43 to 4 in SugarScape. While one comparison filtered out ranges of agent instances from capture, the number of agents with captured procedures on each host remains the same in both comparisons (10 per host, as shown in Tables 7 and 8). In effect, the previous comparison provided measures of a more practical profile of adaptive capture configuration, while this comparison hints at upper-boundaries of capture expense for various support scenarios (e.g., analysis tasks that require fine granularity, but few agents and operations or vice-versa).

b) Study Results

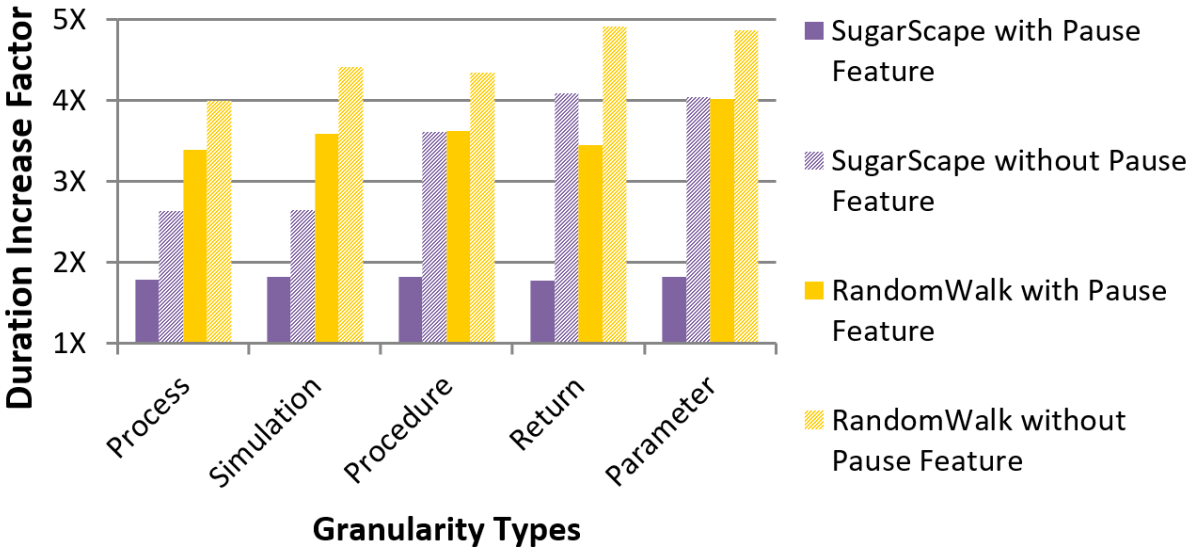

Figure 15. Factors of duration increase, with and without pause feature, on 16 nodes

Granularity level for this comparison varied from process (coarse) to parameter (fine), along the x-axis of Figure 15. Meanwhile, across all configurations and models, the total execution

duration of simulation runs (compared with corresponding runs without provenance capture) ranged from 177% to 491%, as shown along the y-axis of Figure 15. Provenance capture was turned on for the entire simulation in the first set of simulation runs (without pause feature), shown in light purple and light gold. In other words, while the granularity configuration may have prevented some provenance capture, the pause feature was not used for these runs. Meanwhile, the simulation runs, shown in solid purple and solid gold in Figure 15, capture two operations within two iterations of the simulation logic, as described in the agents-scale comparison.

c) Study Discussion

As in the previous study, these measurements are based on total execution time. At finer granularity and with larger numbers of agents, more resources (i.e. memory) must be dedicated to the simulation. Meanwhile, the provenance size can also become quite large. These factors influence the number of iterations executed in the main loop of simulation logic. This comparison involved just 10 iterations. Counterintuitively, the factor of increased run duration was quite higher than in the last comparison for runs with the same granularity-level (Simulation-level). This is primarily a result of total execution time. The previous comparison took around 160 minutes to complete for the largest model size. Meanwhile, these simulation runs never exceeded 10 minutes. Consequently, time for buffer allocation and setup has a higher impact on the results. While the main concern is provenance capture overhead, initialization is an essential part of the expense considerations that this study was meant to present.

At the finest granularity levels (i.e. return-level and parameter-level), the factor of increased execution has been reduced by over 50% for SugarScape. RandomWalk results indicate less reduction. However, for both models, there is less variation between results for different

granularity levels. This indicates a reduction in overhead increase as provenance becomes more fine-grained. Consequently, the pause feature is promising, where applicable.

6.3.3 *Discussion*

Both SugarScape and RandomWalk were selected for their simplicity, to clarify the relationship between agent operations and performance overhead. RandomWalk provides the most straightforward migration activity to reason about, wherein an agent simply migrates during every simulation iteration after reporting its location. Meanwhile, SugarScape adds minimal logic to this migration activity to form unique behaviors. While these simulations do not demonstrate practical provenance use, they do eliminate confounding operations through reduced model complexity.

The studies in this section assess performance overhead of the provenance capture approach compared to a complete lack of provenance support, but it is important to keep the cost of alternatives in mind. An approach to support reproducibility and verification in model development may still be necessary, despite a lack of provenance support. These comparisons do not include more traditional measures for supporting such needs. For example, a researcher might attempt to gain a rough insight into intermediate phenomena, periodically gathering agent state by calling a utility procedure on all agents, at regular intervals. The cost of such an approach is not included in these performance comparisons. Instead, these comparisons baseline performance overhead considerations for adopting the ProvMASS approach.

While performance overhead is an important consideration for configuring adaptive capture, memory and disk space requirements are also important. Comparison of duration increase related to various levels of provenance granularity, shown in Figure 15, indicates that full provenance capture comes with excessive performance cost. Simple observation that there is a relationship between the generated provenance size and increased execution time implies that unfiltered

provenance at fine granularity may also result in excessive space requirements, both in terms of memory and disk space. Consequently, in-depth investigation of space requirements across various models, model configurations and provenance configurations is needed.

Results in these studies show that, depending on the use case, provenance capture related performance overhead may be reduced to viable levels. Even in the most demanding cases, minimal provenance is still an option to consider (e.g., simulation-level granularity, filtering out all agent's and place's provenance). These results indicate an answer to the third research question presented in Section 1.5:

> **RQ3** Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?
>
> **Answer: Yes,** irrespective of model size and iterations, provenance capture can be fixed, limiting overhead. Further, overhead *increase* can be limited when capture cannot be fixed (i.e. when fine granularity is required).

6.4 LIMITATIONS

The ProvMASS approach is limited by constraints placed on the processing environment and corresponding design trade-offs. Among these trade-offs, the greatest limitation corresponds to the need for in-memory storage to maintain concurrency (i.e. the ProvenanceStore component discussed in Chapter 5). Consequently, available memory is diverted from the application to store provenance. Memory allotment and corresponding persistence frequency are important considerations when tuning configuration of provenance management. Lowering the size of in-memory provenance storage increases the memory for the application, but may also increase persistence frequency. While double-buffering in-memory provenance storage provides an upper-

bound on persistence frequency, it might be further limited by increasing the primary buffer space. Yet, some models may have substantial memory requirements, necessitating a minimal provenance memory configuration.

To aid in making configuration decisions, heuristics are collected by the provenance management system (i.e. distributed ProvenanceStoreManager component discussed in Chapter 5) and presented to the user in the form of logs. All active provenance storage components track the highest character length added. During global persistence operations, the managers determine the highest length reported by all provenance storage. This value is logged along with notification of the persistence operation. The manager also keeps track of the names of threads that were denied provenance storage and the name of the last thread to be assigned provenance storage. This information is also logged during global operations, aiding the user in selecting the amount (e.g., number of ProvenanceStore instances) and size of provenance storage (e.g., number of buffers, lines, and characters) to provision during initialization. Unfortunately, this provides limited aid to users, as the profile of the model (e.g., requirements for supporting exchange operations, large number of agents and places, etc.) also plays a role in tuning these provenance configuration parameters. Instead, such heuristics simply serve to indicate the minimum provenance storage requirements.

Another limitation pertains to applying the adaptive approach in the master-slave paradigm. Most global provenance management operations are restricted to initialization and finalization. However, the granularity adjustment and pause features require systemwide changes in manager configuration. As MASS coordination is distributed, this entails a message broadcast to all participating hosts, involving barrier synchronization (i.e. the master awaits reply from all slaves before continuing). Using either of these operations multiple times in the main simulation loop can

result in performance degradation that outweighs the feature benefits. Thus, the user is encouraged to reduce these operations. This can be accomplished by narrowing the query focus to provenance generated in a specific operation or specific iterations. Alternatively, regrouping operations within the simulation loop can also reduce the necessary broadcasting to pause and resume provenance capture. For example, it may be possible to move an exchangeAll operation (i.e. spatial data exchange among neighboring places) to the beginning or end of a loop rather than in between callAll and manageAll operations (i.e. instruction for all agents to invoke a specific procedure and instruction to update all agent's positions). By doing this, the global coordination messages to pause and resume provenance can be reduced from four or more per iteration, to just two.

Another limitation stems from postponement of graph construction. Internal provenance storage and high provenance throughput necessitate delayed graph construction. Further, to keep provenance management lightweight, construction of the provenance graph is postponed until MASS execution finishes. Triples stored in raw provenance files are independent and can be used to form a partial provenance graph during execution. Consequently, users can query provenance of long-running simulations. However, unflushed provenance (i.e. buffered, yet not persisted) will not be represented in the graph, making the resulting graph temporally jagged. The application developer may optionally broadcast systemwide persistence at regular intervals, but such operations impede performance, as described above.

Finally, these limitations imply an inability to support *in situ* query. While raw provenance triples directly represent graph vertices and edges, alleviating the need to post-process relationships between resources, graph structure is necessary to enable relational queries. Not only is secondary storage required, but queries must be executed outside of the MASS application. Mentioned in Chapter 3, MASS execution constrains system resources. Meanwhile, complex

queries may require a great deal of memory and the mechanisms to execute them will require free processing cores. Adaptive capture provides a workaround in which unnecessary provenance is discarded, resulting in improved query performance. If the provenance is stored in a network-accessible location, it can be copied to other hardware, where free memory and processing resources can be applied to bolster query performance. These workarounds, however, do not eliminate possible bottlenecks introduced by the need for secondary storage.

Chapter 7. CONCLUSION AND FUTURE WORK

ProvMASS represents a novel approach to capture provenance of multi-agent models in a distributed memory. While techniques exist to capture multi-agent model provenance, shared memory operations and distributed system calls, these techniques have been investigated in isolation. Even combined, they are inadequate to reconcile operations over distributed shared resources.

Evaluation of the ProvMASS approach indicates that sufficient provenance is captured to explain individual agent behaviors, relate those behaviors via framework execution traces, and tie them to source code of respective simulation logic. Further, performance measures indicate the ability to adapt provenance capture to fit specific use cases and limit impact on simulation efficiency.

This thesis represents an initial investigation in solving provenance capture challenges corresponding to distributed memory. While the provenance generated by the current approach is sufficient to answer research questions presented in Chapter 1, the following work remains. Investigation into removing the persistence bottleneck is necessary to support queries that require the full provenance record, yet are specific to a group of data. For example, the user may wish to observe all the activities of an agent that has a specific quality or certain interaction with the framework. In this case, the query does not depend on the entire provenance record, but a part that in not known in advance and involves provenance at various granularity and simulation operations. Techniques to support these scenarios with *in situ* queries over a sliding window of provenance (e.g., graph in which older vertices are removed to make room for newer vertices) are under consideration, as are advanced filtering techniques.

In situ queries can be evaluated as elements of a provenance stream are generated and subsequently discarded [6]. This reduces space requirements for provenance-based big data analysis. However, stream processing techniques preclude investigative probing of agent behavior, illustrated by the series of joined queries described in Section 6.2.1. Study of advanced filtering (e.g., dynamic adjustment of provenance granularity) to probe an agent's lineage may compensate for this deficiency while adding more practical provenance support.

Techniques to constrain provenance to memory while tracing individual resource lineage (e.g., that of an agent or place) may provide a powerful simulation analysis tool. For example, consider a traffic simulation to measure congestion. Combined, these techniques could be used to first identify a vehicle that avoided a congested route, then retain only that vehicle's provenance that includes the route and travel duration. Note, however, that retention of the relevant agent's provenance is necessary and that, until the agent is identified, this includes similar provenance of all agents. Techniques to abbreviate the provenance graph (i.e. apply transitive activity influence, where appropriate) and filter provenance granularity more precisely (e.g., retaining parameters and returns only where necessary to describe the route) will be explored to cope with investigative portions of *in situ* analysis.

Space requirements of the ProvMASS approach have yet to be evaluated. In addition to performance, these are not only important considerations for choosing an adaptive capture configuration, but in removing the persistence bottleneck described in Section 4.3. Future study of these considerations is supported by the design of current provenance storage and management components described in Chapter 5. First, the primary provenance storage mechanism (i.e. the ProvenanceStore component) supports this with offset pointers into dimensions of the buffer structure, which can be used to directly calculate the number of bytes consumed. Meanwhile, the

map between provenance storage and threads of execution can be used in the same way that it is used to coordinate global persistence; a procedure can be carried out on each storage instance to indicate the total amount of stored provenance. These studies will require that final persistence is disabled to ensure that the size of all recorded provenance data is being measured. While storage overhead varies with the procedures used, which are specific to the multi-agent model, the space requirements for various model operations can be measured starting and ending in the procedure that coordinates the corresponding simulation. These measurements could also provide useful information about the space requirements at various granularity, and the memory profile of various framework operations (e.g., comparing the provenance space requirements of an exchangeAll operation on a collection of places to a manageAll operation on a similarly sized collection of agents).

Pursuit of methods to eliminate persistence also provide the opportunity to apply practical provenance support to more complex simulations. Capture of the agent's provenance in the traffic simulation example, described above, may provide an interesting overhead profile as the agent migrates from host to host. Meanwhile, it is also interesting to consider elements of the spatial environment with respect to all agents. Consider an evacuation simulation, where agents attempt to escape a burning or collapsing building. The advanced filtering and graph abbreviation techniques, described above, may help compare rates of agent egress through the most congested doorways. Useful multi-agent modeling hinges on the ability to analyze real-world systems. Therefore, development and evaluation of analysis techniques to answer questions in more practical scenarios will further validate the ProvMASS approach to reasoning about multi-agent models in a distributed memory.

BIBLIOGRAPHY

[1] Sherif Akoush, Ripduman Sohan, and Andy Hopper. HadoopProv: towards provenance as a first class citizen in mapreduce. In Proc of the Theory and Practice of Provenance (TaPP), 2013.

[2] Li An. Modeling human decisions in coupled human and natural systems: Review of agent-based models. Ecological Modelling, 229:25 – 36, 2012. Modeling Human Decisions.

[3] Zachary J. Brownell. Critical MASS: Performance and programmability evaluation of MASS (multi-agent spatial simulation) and hybrid OpenMP/MPI. Master's thesis, University of Washington, 2015.

[4] Rajkumar Buyya, High Performance Cluster Computing: Architectures and Systems, Vol. 1 (Prentice-Hall, 1999).

[5] Rajkumar Buyya, High Performance Cluster Computing: Architectures and Systems, Vol. 2 (Prentice-Hall, 1999).

[6] Peng Chen, Tom Evans, and Beth Plale. Analysis of Memory Constrained Live Provenance. In Proc of Int'l Provenance and Annotation Workshop (IPAW), pages 42–54. Springer International Publishing, 2016.

[7] Peng Chen, Beth Plale, and Tom Evans. Dependency provenance in agent based modeling. In Proc of eScience, pages 180–187, Oct 2013.

[8] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases: Why, how, and where. Foundations and Trends in Databases, 1(4):379–474, 2009.

[9] Timothy Chuang and Munehiro Fukuda. A parallel multi-agent spatial simulation environment for cluster systems. In Proc. of Int'l Conf on Computational Science and Engineering, 2013.

[10] Daniel Crawl, Jianwu Wang, and Ilkay Altintas. Provenance for mapreduce-based data-intensive workflows. In Proc of Workshop on Workflows in Support of Large-scale Science, 2011.

[11] Andrew Davison. Automated capture of experiment context for easier reproducibility in computational research. Computing in Science & Engineering, 14(4):48–56, 2012.

[12] Chris Drummond. Replicability is not reproducibility: nor is it good science. Proc of Eval Methods for Machine Learning Workshop, 2009.

[13] John B. Dunning, David J. Stewart, and Jianguo Liu. Individual-Based Modeling, pages 228–245. Springer New York, New York, NY, 2002.

[14] John Emau, Timothy Chuang, Munehiro Fukuda, A Multi-Process Library for Multi-Agent and Spatial Simulation. In Proc. of Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), 2011.

[15] Joshua M. Epstein. Agent-based computational models and generative social science. Complexity, 4(5):41–60, 1999.

[16] Joshua M Epstein and Robert Axtell. Growing artificial societies: social science from the bottom up. Brookings Institution Press, 1996.

[17] Jacques Ferber. Multi-agent systems: an introduction to distributed artificial intelligence, volume 1. Addison-Wesley Reading, 1999.

[18] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. In Proc of Int'l Middleware Conf, 2012.

[19] Provenance in NetLogo. Provenance in netlogo. https://sourceforge.net/projects/pin/, 2017.

[20] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. Titian: Data provenance support in spark. Proc of the VLDB Endowment, 9(3):216–227, Nov 2015.

[21] Hongbin Li. A Debugger of Parallel Multi-Agent Spatial Simulation https://depts.washington.edu/dslab/MASS/reports/HongbinLi_final_au14.ppt, August 2017.

[22] Brandon Lucia and Luis Ceze. Data provenance tracking for concurrent programs. In Proc of Int'l Symposium on Code Generation and Optimization (CGO), pages 146–156, Feb 2015.

[23] Tanu Malik, Ashish Gehani, Dawood Tariq, and Fareed Zaffar. Data Provenance and Data Management in eScience, chapter Sketching Distributed Data Provenance, pages 85–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[24] Paolo Missier, Khalid Belhajjame, and James Cheney. The W3C PROV family of specifications for modelling provenance metadata. In Proc of Int'l Conf on Extending Database Technology, 2013.

[25] K-K Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In Proc USENIX Annual Tech Conf, 2009.

[26] David O'Sullivan and George LW Perry. Spatial simulation: exploring pattern and process. John Wiley & Sons, 2013.

[27] Hyunjung Park, Robert Ikeda, and Jennifer Widom. RAMP: a system for capturing and tracing provenance in MapReduce workflows. In Proc of Int'l Conf on Very Large Data Bases (VLDB), 2011.

[28] Edoardo Pignotti, Gary Polhill, and Peter Edwards. Using provenance to analyse agent-based simulations. In Proc of the Joint EDBT/ICDT Workshops, 2013.

[29] PROV-O. Prov-o. http://www.w3.org/TR/prov-o/, August 2017.

[30] PROV-Overview. Prov-overview. http://www.w3.org/TR/2013/NOTE-provoverview-20130430/, August 2017.

[31] QDox. GitHub - paul-hammant/qdox: QDox - full extractor of Java class/interface/method definitions (including annotations, parameters, param names). https://github.com/paul-hammant/qdox, August 2017.

[32] Steven F. Railsback. Concepts from complex adaptive systems as a framework for individual-based modelling. Ecological Modelling, 139(1):47 – 62, 2001.

[33] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance in e-Science. SIGMOD Record 34(3): 31-36, 2005.

[34] Niko Simonson, Sean Wessels, and Munehiro Fukuda, Language and Debugging Support for Multi-Agent and Spatial Simulation. In Proc. of the Int'l Conf on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2012.

[35] Isuru Suriarachchi, and Beth Plale. Crossing analytics systems: A case for integrated provenance in data lakes. In Proc of eScience. pages 349 – 354, 2016.

[36] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. In Proc of the Theory and Practice of Provenance (TaPP), 2012.

[37] J. Thalheim, P. Bhatotia, and C. Fetzer. Inspector: Data provenance using intel processor trace. In Proc Int’l Conf on Distrib Comp Systems, 2016.

[38] Jianwu Wang, Daniel Crawl, and Ilkay Altintas. Kepler + Hadoop: A general architecture facilitating data-intensive applications in scientific workflow systems. In Proc of Workshop on Workflows in Support of Large-Scale Science, 2009.

APPENDIX A

```
/**
 * High-level Provenance Capture Sourcecode Instrumentation Logic
 *
 * Instruments source code files with provenance capture statements and field IDs.
 *
 * Partially un-factored to show logic
 */
public class SourceCodeInstrumentor {
    public static void generateInstrumentedFiles(List<String> folders, List<String> ignored)
            throws IOException {
        List<File> sourceFiles = getSourceFiles(folders, ignored);
        removeIgnoredFiles(sourceFiles);
        JavaProjectBuilder builder = new JavaProjectBuilder();
        for (File sourceFile : sourceFiles) {
            builder.addSource(sourceFile);
        }
        for (JavaSource javaSource : builder.getSources()) {
            for (JavaClass javaClass : javaSource.getClasses()) {
                String sourceCode = javaClass.getCodeBlock();
                for (JavaMethod javaMethod : javaClass.getMethods()) {
                    /* get signature parts */
                    String name = javaMethod.getName();
                    List<JavaParameter> params = javaMethod.getParameters();
                    JavaClass returnClass = javaMethod.getReturns();
                    // get entire method block
                    String instructions = javaMethod.getSourceCode();
                    // parse instructions, manipulate them with sig parts,
                    // and update source
                    sourceCode = updateProcedure(sourceCode, name, params, returnClass,
                            instructions);
                }
                for (JavaClass nestedClass : javaClass.getNestedClasses()) {
                    for (JavaMethod javaMethod : javaClass.getMethods()) {
                        /* get signature parts */
                        String name = javaMethod.getName();
                        List<JavaParameter> params = javaMethod.getParameters();
                        JavaClass returnClass = javaMethod.getReturns();
                        // get entire method block
                        String instructions = javaMethod.getSourceCode();
                        // parse instructions, manipulate them with sig parts,
                        // and update source
                        sourceCode = updateProcedure(sourceCode, name, params, returnClass,
                                instructions);
```

```
                    }
                }
                // insert field IDs and calls to retrieve them upon initialization
                sourceCode = enableFieldOpCapture(javaClass, sourceCode);
                // make an instrumented copy of the original code
                createInstrumentedClassCopy(javaSource, javaClass, sourceCode);
                break;
            }
        }
    }
    // ...
} // end source code

/* Sample of Prexisting Procedure, Instrumented with Provenance Capture /*
/**
 * Returns the number of agents to initially instantiate on a place indexed
 * with coordinates[]. The maxAgents parameter indicates the number of
 * agents to create over the entire application. The argument size[] defines
 * the size of the "Place" matrix to which a given "Agent" class belongs.
 * The system-provided (thus default) map( ) method distributes agents over
 * places uniformly as in: maxAgents / size.length The map( ) method may be
 * overloaded by an application-specific method. A user-provided map( )
 * method may ignore maxAgents when creating agents.
 *
 * @param initPopulation - initial agent population
 * @param size - dimensions of the places collection where agent resides
 * @param index - coordinates (in places collection) of place being
 * evaluated for agent mapping
 * @return number of agents that should be instantiated and mapped to the
 * place
 */
public int map(int initPopulation, int[] size, int[] index) {
    /* CAPTURE PROCEDURE INVOCATION */
    StringBuffer procRID = ProvenanceRecorder.documentProcedure(provOn,
            ProvUtils.getStoreOfCurrentThread(provOn), this, new StringBuffer("map"),
            new StringBuffer("label"), true, new String[]{"initPopulation", "size", "index"},
            new Object[]{initPopulation, size, index});

    // compute the total # places
    int placeTotal = 1;
    for (int x = 0; x < size.length; x++) {
        placeTotal *= size[x];
    }

    // compute the global linear index
    int linearIndex = 0;
    for (int i = 0; i < index.length; i++) {
        if (index[i] >= 0 && size[i] > 0 && index[i] < size[i]) {
            linearIndex = linearIndex * size[i];
```

```
            linearIndex += index[i];
        }
    }

    // compute #agents per place a.k.a. colonists
    int colonists = initPopulation / placeTotal;
    int remainders = initPopulation % placeTotal;
    if (linearIndex < remainders) {
        colonists++; // add a remainder
    }

    /* CAPTURE PROCEDURE RETURN */
    ProvenanceRecorder.endProcedureDocumentation(provOn,
            ProvUtils.getStoreOfCurrentThread(provOn), this,
            new StringBuffer("map"), procRID, new StringBuffer("colonists"),
            colonists, null, new StringBuffer(""), true, false, false);
    return colonists;
}

/**
 * Instrumented procedure wraps general procedure capture, to automatically control
 * granularity.
 * (see bold 1st line in the map method, just above)
 */
public static String documentProcedure(boolean provOn, ProvenanceStore store,
        Object caller, String procName, StringBuffer label,
        boolean alwaysPushActivityID, String[] paramNames, Object[] params) {
    alwaysPushActivityID = false;
    if (provOn) {   // CALLS GENERAL PROCEDURE CAPTURE HERE
        return documentProcedure(store, caller, procName, label, alwaysPushActivityID,
                paramNames, params, false);
    }
    return null;
}

/**
 * Captures provenance of a procedure invocation, including caller-callee
 * relationship with ResourceMatcher and parameters used by the newly
 * generated "activity"
 *
 * Called from corresponding wrapper, shown above.
 *
 * @return The ID of the newly generated/stored procedure invocation
 * activity (to pair with procedure return capture)
 */
 public static String documentProcedure(ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, String[] paramNames, Object[] params,
        boolean ignoreGranularity) {
```

```
alwaysPushActivityID = false;
StringBuffer procRID = null;
if (MASSProv.provOn) {
    try {
        procRID = ProvenanceRecorder.documentProcedure(store, caller,
                procName, label, alwaysPushActivityID, ignoreGranularity);
        if (procRID != null && paramNames != null && params != null
                && (ignoreGranularity || granularityLevel()
                >= Granularity.PARAMS.getValue())) {
            ResourceMatcher matcher = ResourceMatcher.getMatcher();
            StringBuffer paramRID, stackedArgID;
            StringBuffer paramValue = null;
            for (int i = 0, im = ProvUtils.getLowest(
                    paramNames.length, params.length); i < im; i++) {
                paramRID = ProvUtils.getUniversalResourceID(
                        new StringBuffer(paramNames[i]));
                stackedArgID = null;
                /* param is an entity */
                store.addRelationalProv(paramRID, provOntology.getRDFTypeFullURIBuffer(),
                        ProvOntology.getEntityStartingPointClassFullURIBuffer());
                /* param has value (remember id is unique as references are
            passed by value... this will not overwrite the value outside
            of the procedure's scope) */
                if (params[i] != null) { // value available
                    StringBuffer value = isImmutable(params[i])
                            || params[i] instanceof Date
                                    ? new StringBuffer(params[i].toString())
                                    : new StringBuffer(
                                            String.valueOf(params[i].hashCode()));
                    paramValue = new StringBuffer("\"").append(value).
                            append("\"");
                } else { // sensitive data
                    paramValue = new StringBuffer("\"sensitiveValue\"");
                }
                store.addRelationalProv(paramRID,
                        ProvOntology.getValueExpandedPropertyFullURIBuffer(),
                        paramValue);
                /* proc used param */
                store.addRelationalProv(procRID,
                        ProvOntology.getUsedStartingPointPropertyFullURIBuffer(),
                        paramRID);
                if (!isImmutable(params[i])) {
                    stackedArgID = matcher.popEntityIDBuffer();
                    if (stackedArgID != null) { // was stacked
                        /* param is alternate version of the corresponding argument */
                        store.addRelationalProv(paramRID,
                                ProvOntology.getAlternateOfExpandedPropertyFullURIBuffer(),
                                stackedArgID);
                    }
```

```
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace(IO.getLogWriter());
        }
    }
    return procRID;
   }


/**
 * Several Functions for ProvenanceRecorder Operations (mostly called from instrumented
 * procedures, like the map function, shown above) follow. ProvenanceStores are retrieved
 * from the Manager in instrumented procedures, inline with the procedure calls.
 *
 * Most procedures return a value corresponding to a newly generated
 * identifier for the main provenance resource that was generated (e.g.,
 * procedure invocation identifier), to use in future recording.
 *
 * Most parameters are optional. For example, any combination of the following can be used to
 * formulate identification of an entity representing a variable: name (make new ID using
 * name), provenance identifier (ID ready), actual variable object (use type to generate ID)
 */

public static String documentProcedure(ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, boolean ignoreGranularity);

public static String documentProcedure(boolean provOn, ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, boolean ignoreGranularity);

public static String documentProcedure(ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID);

public static String documentProcedure(boolean provOn, ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID);

public static String documentProcedure(ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, String[] paramNames, Object[] params,
        boolean ignoreGranularity);
```

```
public static String documentProcedure(boolean provOn, ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, String[] paramNames, Object[] params,
        boolean ignoreGranularity);

public static String documentProcedure(ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, String[] paramNames, Object[] params);

public static String documentProcedure(boolean provOn, ProvenanceStore store,
        Object caller, String procName, String label,
        boolean alwaysPushActivityID, String[] paramNames, Object[] params);

public static void endProcedureDocumentation(ProvenanceStore store,
        Object caller, String procName, String procRID, String returnObjectName,
        Object returnObject, String returnObjectRID, String label,
        boolean procGeneratedReturn, boolean alwaysStackReturnID,
        boolean dontPopActivityID, boolean ignoreGranularity);

public static void endProcedureDocumentation(boolean provOn, ProvenanceStore store,
        Object caller, String procName, String procRID, String returnObjectName,
        Object returnObject, String returnObjectRID, String label,
        boolean procGeneratedReturn, boolean alwaysStackReturnID,
        boolean dontPopActivityID, boolean ignoreGranularity);

public static void endProcedureDocumentation(ProvenanceStore store,
        Object caller, String procName, String procRID, String returnObjectName,
        Object returnObject, String returnObjectRID, String label,
        boolean procGeneratedReturn, boolean alwaysStackReturnID,
        boolean dontPopActivityID);

public static void endProcedureDocumentation(boolean provOn, ProvenanceStore store,
        Object caller, String procName, String procRID, String returnObjectName,
        Object returnObject, String returnObjectRID, String label,
        boolean procGeneratedReturn, boolean alwaysStackReturnID,
        boolean dontPopActivityID);

public static void documentProcReturn(String procName, String procRID,
        Object returnObject, String returnObjectRID, String returnObjectName,
        ProvenanceStore store, boolean documentGenerationByProc,
        boolean alwaysStackReturnID, boolean ignoreGranularity);

public static void documentProcReturn(boolean provOn, String procName, String procRID,
        Object returnObject, String returnObjectRID, String returnObjectName,
        ProvenanceStore store, boolean documentGenerationByProc,
        boolean alwaysStackReturnID, boolean ignoreGranularity);
```

```
public static void documentProcReturn(String procName, String procRID,
        Object returnObject, String returnObjectRID, String returnObjectName,
        ProvenanceStore store, boolean documentGenerationByProc,
        boolean alwaysStackReturnID);

public static void documentProcReturn(boolean provOn, String procName, String procRID,
        Object returnObject, String returnObjectRID, String returnObjectName,
        ProvenanceStore store, boolean documentGenerationByProc,
        boolean alwaysStackReturnID);

public static String documentEntity(ProvenanceStore store, Object caller,
        String callerRID, String procName, String procRID, String entityName,
        Object entity, String entityRID, String alternativeOfRID,
        boolean stackRID);

public static String documentEntity(boolean provOn, ProvenanceStore store, Object caller,
        String callerRID, String procName, String procRID, String entityName,
        Object entity, String entityRID, String alternativeOfRID,
        boolean stackRID);

public static String documentAgent(ProvenanceStore store, Object agent,
        String agentRID, String agentName, String label);

public static String documentAgent(boolean provOn, ProvenanceStore store, Object agent,
        String agentRID, String agentName, String label);

public static String documentProvEnabledObject(ProvenanceStore store,
        Object provEnabledObject, String provEnabledObjectRID,
        String provEnabledObjectName, boolean objIsProvEnabled,
        String label);

public static String documentProvEnabledObject(boolean provOn, ProvenanceStore store,
        Object provEnabledObject, String provEnabledObjectRID,
        String provEnabledObjectName, boolean objIsProvEnabled,
        String label);

public static String documentAgent(ProvenanceStore store, Object agent,
        String agentRID, String agentName);

public static String documentAgent(boolean provOn, ProvenanceStore store, Object agent,
        String agentRID, String agentName);

private static String documentEntity(ProvenanceStore store,
        String entityRID);

private static String documentEntity(boolean provOn, ProvenanceStore store,
        String entityRID);
```

```
private static void documentGeneration(ProvenanceStore store,
        String entityRID, String activityRID);

private static void documentGeneration(boolean provOn, ProvenanceStore store,
        String entityRID, String activityRID);

private static void documentAlternativity(ProvenanceStore store,
        String entityRID, String alternativeOfRID);

private static void documentAlternativity(boolean provOn, ProvenanceStore store,
        String entityRID, String alternativeOfRID);

private static void documentEntity(ProvenanceStore store, String entityRID,
        String value);

private static void documentEntity(boolean provOn, ProvenanceStore store,
        String entityRID, String value);

public static String documentSimDriver(ProvenanceStore store,
        String simulatorRID, String driverRID, String driverMethodName,
        String label);

public static String documentSimDriver(boolean provOn, ProvenanceStore store,
        String simulatorRID, String driverRID, String driverMethodName,
        String label);

public static String[] documentCommandLineArguments(ProvenanceStore store,
        String simulatorRID, String[] args, String argsCollectionRID);

public static String[] documentCommandLineArguments(boolean provOn, ProvenanceStore store,
        String simulatorRID, String[] args, String argsCollectionRID);

public static void documentFieldAccess(ProvenanceStore store,
        String fieldName, String fieldRID, String procRID, Object accessed);

public static void documentFieldAccess(boolean provOn, ProvenanceStore store,
        String fieldName, String fieldRID, String procRID, Object accessed);

public static void documentFieldAccess(ProvenanceStore store,
        String fieldName, String fieldRID, String procRID, Object accessed,
        boolean ignoreGranularity);

public static void documentFieldAccess(boolean provOn, ProvenanceStore store,
        String fieldName, String fieldRID, String procRID, Object accessed,
        boolean ignoreGranularity);

public static void documentFieldAssignment(ProvenanceStore store,
        String fieldName, String fieldRID, Object field, String procRID);
```

```
public static void documentFieldAssignment(boolean provOn, ProvenanceStore store,
        String fieldName, String fieldRID, Object field, String procRID);

public static void documentFieldAssignment(ProvenanceStore store,
        String fieldName, String fieldRID, Object field, String procRID,
        boolean ignoreGranularity);

public static void documentFieldAssignment(boolean provOn, ProvenanceStore store,
        String fieldName, String fieldRID, Object field, String procRID,
        boolean ignoreGranularity);
```

VITA

Delmar Bryan Davis

EDUCATION

Master of Science **2017**
Computer Science and Software Engineering
University of Washington, Bothell *Bothell, Washington*

Bachelor of Science **2012**
Computer Science and Software Engineering
University of Washington, Bothell *Bothell, Washington*

RESEARCH EXPERIENCE

Graduate Research Assistant **2013 – 2017**
School of Science, Technology, Engineering & Mathematics
University of Washington, Bothell *Bothell, Washington*

Undergraduate Research Assistant **2012 – 2013**
Computing & Software Systems
University of Washington, Bothell *Bothell, Washington*

TEACHING EXPERIENCE

Adjunct Faculty **2013**
Cascadia Community College *Bothell, Washington*

REFEREED CONFERENCE AND WORKSHOP PUBLICATIONS

Data Provenance for Multi-Agent Models **Oct. 2017**
Delmar B. Davis, Jonathan Featherston, Munehiro Fukuda, and Hazeline U. Asuncion. *(to appear)*
13th Int'l Conference on eScience

BrainGrid+Workbench: High-Performance/High-Quality Neural Simulation **May 2017**
Michael Stiber, Fumitaka Kawasaki, Delmar Davis, Hazeline Asuncion, Jewel Lee, and Destiny Boyer
Int'l Joint Conference in Neural Networks

Improving Data Provenance Reconstruction via a Multi-Level Funneling Approach **Oct. 2016**
Subha Vasudevan, William Pfeffer, Delmar Davis, and Hazeline Asuncion
12th Int'l Conference on eScience

A Multi-Level Funneling Approach to Data Provenance Reconstruction **Oct. 2014**
Ailifan Aierken, Delmar B. Davis, Qi Zhang, Kriti Gupta, Alex Wong, and Hazeline U. Asuncion
e-Science Workshop of Works in Progress

Tracing Domain Data Concepts in Layered Applications **Jul. 2014**
Mohammed Daubal, Nathan Duncan, Delmar B. Davis, and Hazeline U. Asuncion
26th Int'l Conference on Software Engineering and Knowledge Engineering

Uncovering File Relationships using Association Mining and Topic Modeling **Mar. 2014**
Namita Dave, Delmar B. Davis, Karen Potts, and Hazeline U. Asuncion
Sixth Int'l Conference on Information, Process, and Knowledge Management

Towards Recovering Provenance with Experiment Explorer **Feb. 2013**
Delmar B. Davis, Hazeline U. Asuncion, Ghaleb M. Abdulla, and Christopher W. Carr
Fifth Int'l Conference on Information, Process, and Knowledge Management

Experiment Explorer: Lightweight Provenance Search over Metadata **Jun. 2012**
Delmar B. Davis, Hazeline U. Asuncion, and Ghaleb Abdulla
USENIX Workshop on the Theory and Practice of Provenance

AFFILIATION

Upsilon Pi Epsilon **2012-2017**
Honor Society of the Computing and Information Disciplines
Delta Chapter of Washington, University of Washington, Bothell
President **2014-2017**