informatics

Article

Data Provenance for Agent-Based Models in a Distributed Memory

Delmar B. Davis *, Jonathan Featherston, Hoa N. Vo, Munehiro Fukuda and Hazeline U. Asuncion *

School of Science, Technology, Engineering & Mathematics, University of Washington Bothell, Bothell, WA 98011, USA; luxmundi@uw.edu (J.F.); hoav@uw.edu (H.N.V.); mfukuda@uw.edu (M.F.)
* Correspondence: davisdb1@uw.edu (D.B.D.); hazeline@uw.edu (H.U.A.); Tel.: +1-425-352-5403 (H.U.A.)

Received: 12 December 2017; Accepted: 4 April 2018; Published: 9 April 2018

Abstract: Agent-Based Models (ABMs) assist with studying emergent collective behavior of individual entities in social, biological, economic, network, and physical systems. Data provenance can support ABM by explaining individual agent behavior. However, there is no provenance support for ABMs in a distributed setting. The Multi-Agent Spatial Simulation (MASS) library provides a framework for simulating ABMs at fine granularity, where agents and spatial data are shared application resources in a distributed memory. We introduce a novel approach to capture ABM provenance in a distributed memory, called ProvMASS. We evaluate our technique with traditional data provenance queries and performance measures. Our results indicate that a configurable approach can capture provenance that explains coordination of distributed shared resources, simulation logic, and agent behavior while limiting performance overhead. We also show the ability to support practical analyses (e.g., agent tracking) and storage requirements for different capture configurations.

Keywords: data provenance; agent-based systems; distributed parallel computing

1. Introduction

Agent-based models (ABMs) describe systems of entities capable of acting in and perceiving their environment [1]. The goal of an ABM is to search for explanatory insight into the collective behavior of agents (which may not be "intelligent" but obey simple rules). These agents typically simulate natural systems, rather than solving specific practical or engineering problems. In this context, ABMs model different reactive agents [2], which is the focus of this paper. ABMs facilitate a micro-simulation that models microscopic events (e.g., traffic simulation involving traffic lights, constructions, and pedestrian movements) and observe an emergent collective group behavior of many agents. Such events may be modeled at fine-granularity, yet centrally coordinated at large-scale (e.g., hundreds of thousands or even millions of agents). However, performance requirements intensify as the number of agents grows. For example, to simulate nationwide or worldwide globalization, urbanization, and population increases, ABMs' size must scale up both processing power and memory size. In general, a cluster system such as Amazon Web Service cluster compute instances (Amazon, 2018. "Amazon Web Services (AWS)-Cloud Computing Services" <https://aws.amazon.com> (accessed 24 February 2018)) can help address such computing requirements by coordinating a distributed memory (i.e., a single memory space that collects all memory from compute instances connected to the cluster). Unfortunately, distributed parallel processing introduces difficulty in reasoning about and debugging agent interactions, especially concurrent distributed memory operations that may be interdependent. Consequently, there is a need for new provenance techniques to bridge the gap between automatic parallelization and model understanding.

Data provenance is defined as the "origin and history of data" [3]. One goal of data provenance is to support the understanding of a given dataset, to determine whether the results of an experiment or simulation are valid [4]. The context in which data operations occur largely determines the suitability of a provenance capture technique. 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. In general, data provenance support is provided within scientific workflows (e.g., [3]) or databases (e.g., [5]), which do not support agent-based modeling [6]. There is support for data provenance for ABMs, but none address distributed, parallel, or distributed-parallel execution contexts. Provenance support in these contexts is needed to explain individual agent behavior [6], which agent-based models aim to provide [7].

Traditional data provenance aims to answer questions such as find the process(es) that led to the current state of data X, find all the invocations of a procedure that ran at a given time, find the differences between two experiment or simulation outputs [8]. We take these traditional provenance queries and apply them to the context of agent-based systems. For example, the query "find the process(es) that led to the current state of data X" is applied to understand agent behavior (UC1, Section 3.2.1) and understand simulation execution (UC2, Section 3.2.2). The related queries are "What conditions (activities and entities) caused an agent to die" (UC1, Section 5.1.1) and "What activities and entities were involved in generating a given simulation output" (UC2, Section 5.1.2). Another example, i.e., the query "find all the invocations of a procedure that ran at a given time", is applied to understand distributed execution (UC3, Section 3.2.3). The related query is "What operations across various hosts took place during a simulation" (UC3, Section 5.1.3). This last query can only be answered with ABM provenance support in a distributed setting, which ProvMASS provides.

ProvMASS is a novel approach to capture provenance of fine-grained ABMs in a distributed memory. This approach is evaluated in two ways. First, we demonstrate the capability to capture provenance in two simple simulations to answer questions related to: (1) agent behavior; (2) simulation logic with respect to distributed coordination of those interactions; and (3) common ABM analysis tasks (see use cases in Section 3.2). This capability aims to answer research question RQ1 presented in Section 3.1; Second, we evaluate performance overhead of provenance capture for various provenance configurations (research question RQ2). We previously introduced these studies with the ProvMASS approach [9,10]. Here, we extend these investigations to: (1) study support for more practical ABM analyses; (2) measure the capability to maintain preexisting system scalability; and (3) measure the storage requirements associated with configurable capture mechanisms (described in Section 4.4).

2. Background

2.1. Spatial Simulation

We consider space as an environment where agents behave and interact with each other [1]. Space can also be a collection of static agents or cellular automata. Therefore, spatial simulation is simulating agents on a given environment or simulating a space itself as a collection of static agents or cellular automata. One goal of our provenance support is to describe agent interactions in terms of these spaces. However, we focus on agent-based modeling rather than spatial simulation.

Systems that are defined by the proximity and state of their elements can be investigated with spatial simulations [11]. One category of spatial simulations can represent entities that interact and have simple state. Examples of spatial simulation includes stochastic models such as influenza transfer or cellular automata. These models rely on state between neighboring regions and can represent individuals with counts of static entities that fluctuate from location to location. Another category of spatial simulations is based on mathematics: wave simulation using Schrodinger's equation, heat diffusion based on Euler's equation, and computational fluid dynamics modeled from Navier–Stokes' equations. Although they partition a space into many small cells, each maintaining a different state (possibly represented by a static agent which does not move over a space), they all use differential formulae derived from these equations. Models from these two categories may even be combined to investigate more complicated entity–environment interactions. However, it is difficult to extend this type of modeling to complex migratory entities, such as those found in biological or social systems. ABMs are better suited to simulate these types of systems.

2.2. Agent-Based Models

In contrast to spatial simulations, ABMs do not generally rely on any mathematics, because model designers want to observe an emergent collective group behavior of many agents. For instance, in Wa-Tor, a school of fish does not assign a mathematical formula to each fish, yet each fish still uses a simple algorithm to escape from a shark [12]. SugarScape, an artificial life simulation over a two-dimensional space of sugar (see Section 3.2), does not assign any mathematical formula to each creature. Instead, these creatures simply move to a cell with sugar, consume it, metabolize it, and pollute the cell [13]. This sequence of behavior is too simple and intuitive to be modeled with mathematical formulae. A well-known proprietary traffic simulator, EMME/3 (INRO, 2018. “INRO | Emme” <http://www.inro.ca/en/products/emme/index.php> (accessed 24 February 2018)), is a macroscopic simulator that observes freeway traffic flow using differential equations. It can be considered as a spatial simulator. The current trend in traffic simulators is ABM. Examples include MATSim [14] and PEDSIM [15]. These simulators give simple mathematical formulas to each agent (i.e., a rule to follow the previous car or person and a formula to avoid collision), but unlike spatial simulation, there are no mathematical formulae to model the entire simulation.

Multi-agent system modeling, with intelligent agents, may center on the individual at the micro-level (i.e., on the internal state of the individual and interactions with its environment) or on organizational concepts [16]. Multi-level simulations offer a means to integrate heterogeneous models, possibly based on different modeling paradigms (e.g., differential equations and cellular automata) [17]. Holonic agents can be used to solve problems that exhibit a recursive structure, breaking the problem down into substructures that can be solved by individual agents that form a holon (i.e., a single larger representative agent) [18]. While intelligent agents can be modeled organizationally to solve complex problems, high resource requirements preclude extremely fine-grained models (hundreds of thousands to millions of concurrently acting agents). Our technique does not support these types of agents (i.e., intelligent agents).

Our technique focuses on reactive agents, which are used for simulations focused on general system modeling (i.e., those concerned only with collective behavior) rather than specific problem solving. These agents may be modeled to simply react to their environment. Simulation of global, interactive phenomena requires more fine-grained 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, 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 (or globally calculated) locale. These alterations not only reduce the agent memory footprint, but also serve to facilitate bottom-up development by promoting a separation of concerns.

2.3. Multi-Agent Spatial Simulation (MASS)

The Multi-Agent Spatial Simulation (MASS) library [19] provides a parallel/distributed framework to automatically parallelize simulations with symmetric multiprocessing over a cluster of computers. Simulation is distributed to overcome two limitations of multi-threaded computing. First, shared-memory computing (or a single computing node with multicores) cannot extend the memory space. MASS resolves this issue by connecting the memory space of instances in the cluster to form a high-level memory coordinated as several distributed arrays. Second, shared memory computing introduces difficulty in handling race conditions. This limitation is resolved by enforcing weak consistency in handling concurrent operations. Without weak consistency, multi-threading needs system-supported stricter synchronizations or manually operated wait/notify synchronization. Therefore, we decided to use MASS as our testbed.

While other parallelization technologies such as OpenMP, MPI, and MapReduce 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 [20]. Parallelization of ABMs 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. While the MASS library supports spatial simulation [21], it also supports broader applications including ABMs [22,23].

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 connect through secure shell (SSH) tunneling. Meanwhile, parallel coordination is extended to intra-process operations at each host through multithreading.

MASS stores simulation space and agents as shared resources in a distributed memory, managed at the object-level (shown in Figure 1). A host process, known as MProcess, maintains several thread objects, known as MThread, 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. Agents are mapped to Places and are contained in another global array.

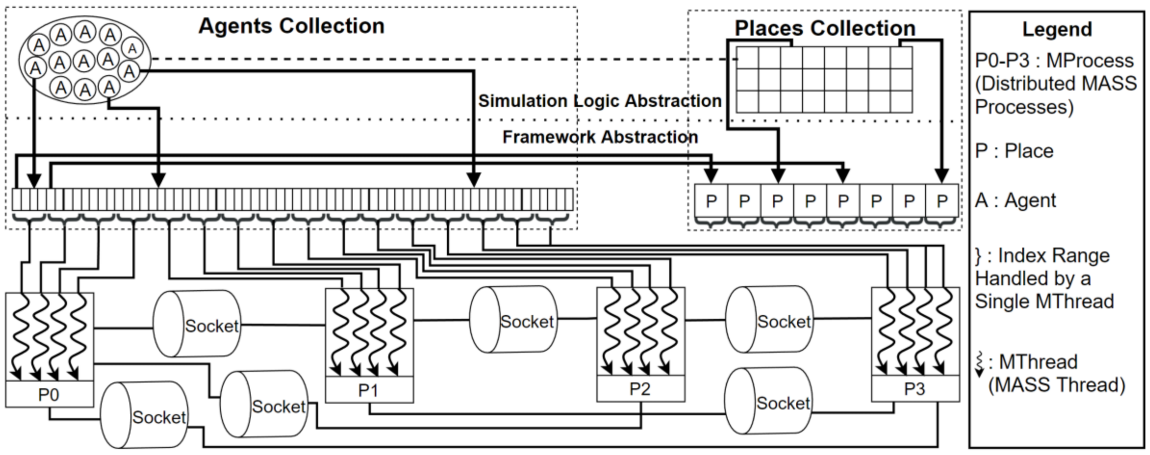

Agents Collection
Places Collection
Legend
Simulation Logic Abstraction
Framework Abstraction
P0-P3 : MProcess (Distributed MASS Processes)
P : Place
A : Agent
} : Index Range Handled by a Single MThread
: MThread (MASS Thread)
Socket
P0
P1
P2
P3

Figure 1. Coordination of shared resource operations in MASS distributed memory [10].

Exchange of data occurs between pairs of MThreads, across all processes. Remote exchange requests are delivered 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), communicating with each remote thread. Each process maintains one socket connection to other processes. 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.

As a result, full network and processor utilization is possible. Note, however, that message delivery is unordered, a challenge in capturing the provenance of messages.

The MASS library provides distributed shared resource management, while hiding not only the associated communication and concurrency mechanisms, but also data locale. The top section of Figure 1 describes the relationship between agent and place resources in their logical collections (i.e., Places and Agents), as provided by the user interface (i.e., within the simulation logic abstraction), and how these resources map to distributed memory management (i.e., within the framework abstraction) in the bottom section. 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. LiSimilar toke 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 simulation logic level [24]. Meanwhile, the mechanisms required to orchestrate such distributed manipulation are handled within the framework.

3. Motivation

3.1. Research Questions

We investigate the following research questions:

- **RQ1** Can provenance be captured to support ABM in a distributed setting?
- **RQ2** Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?

As provenance support for ABM in a distributed setting is currently not supported, we are interested in identifying provenance queries that can be supported for RQ1. More specifically, we investigated if the following types of provenance (PT) can be supported:

PT1 Can provenance of shared resources be captured in a distributed memory?
PT2 Can the full context of individual agent behaviors (e.g., coordination and state of agents and spatial data) be traced to simulation logic?
PT3 Can agent spatial relationships be obtained directly from framework provenance?
PT1 focuses on relating data of various execution contexts with provenance that expresses how the data were used. PT1 also aims to resolve data identity in distributed message passing and disambiguate concurrent intra-process procedure invocation contexts.

Knowledge of relationships between simulation logic and changes in agent state are provided by PT2. 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 type PT2: simulation driver instruction, framework resource coordination operations, and agent interactions.

Unlike the other provenance types, PT3 is concerned with analyzing simulation outcomes without relying on procedures or provenance from user defined models. Different models require different types of analysis, but agent spatial relationships are key to understanding collective agent behavior. Meanwhile, a balance exists between model knowledge and analysis. Without some model knowledge, it can be difficult to formulate analyses. However, analyses provide an understanding of the model. The same balance applies to analyzing provenance of model-specific agent behaviors. If these relationships can be obtained from framework provenance (i.e., provenance of framework operations), an understanding of agent migration patterns can inform prospective analyses.

RQ2 is concerned with the large overhead associated with documenting procedure calls in a high-performance computing setting. High-performance computing results in many data operations. Meanwhile, data provenance capture using text and string operations can be expensive. Therefore, tracing procedure calls in a highly parallelized library can introduce considerable overhead. Even capture of minimalistic ABM provenance (e.g., only agent procedure invocations) is considerably expensive when captured for all agents during an entire simulation. Hence, some form of control is needed to limit the amount of provenance captured. Further, such control mechanisms must accommodate common patterns of provenance use.

3.2. Use Cases

It has been noted that the core ABM philosophy warns against focusing on emergent outcomes, as they may be misleading, and instead promotes emphasis on interactions and uniqueness of the individual [6,7]. The impact of individual interactions is especially difficult to ascertain from fine-grained ABM simulation results, in turn necessitating fine-grained provenance. This shortcoming motivates several use cases based on the types of ABM provenance described by Pignotti et al. [25], including provenance about the social aspect of model development, execution environment, and simulation history. We consider four use cases:

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 the first case.
4. A model designer would like to perform some detailed spatial analysis of a model in development.

These use cases highlight the novelty of the ProvMASS approach and are used in the evaluation in Section 5. Two simple simulations illustrate the basic requirements for use case support while providing a basis for performance overhead measurement: SugarScape [26] and RandomWalk [24]. While these simulations do not serve to illustrate the capability to scale to complex reasoning, they help to form a baseline for model understanding and to show challenges imposed by the processing environment.

SugarScape is a simulation in which agents move in a two-dimensional space, consuming predetermined amounts of "sugar" 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 without sugar at a destination without sugar, it sets a termination flag for the next agent update operation. SugarScape uses computationally intensive neighbor update operations and has limited host traversal by agents, while enabling agent behavior to be examined.

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 [24], 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.

3.2.1. 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 individual agent behavior assists in validation and as a result, model evolution. However, social science research tends to "filter out" various agent behavioral effects [26]. The same can be said of biological systems research.

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 in individual-based models [27,28], but simply lead to correlation of changes in modeled outcomes. As model complexity increases, it becomes more difficult to verify these connections, since it incurs a tremendous amount of work to compare model interactions and the reality of the system [29] (p. 48). Dynamic model testing of ABM simulations requires model instrumentation, model execution and model evaluation [30]. However, evaluation of model execution from isolated observation points can be insufficient where cause and effect is necessary to understand the observation, such as in understanding concurrent interactions in a distributed environment. Provenance that explains exactly how simulation outcomes are affected by changes in the model can supplant the need to add explicit verification to the simulation logic (e.g., in specialized state-checking procedures).

In addition, 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 effect by gathering agent locations in every simulation step, these changes in the simulation logic are unnecessary with ProvMASS.

3.2.2. Use Case 2: Understand Simulation Execution

In the second case, a colleague would like to reproduce the output 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 for running an experiment invariably changes with the researcher, the lab, and the equipment, despite all efforts (e.g., using the same source code) and that such differences matter [31]. The more robust version of replicability, reproducibility, is supported by the context of the results, which includes not only the software, but the details of the computing environment [32]. Information about the computing environment is insufficient to support full simulation reproducibility in MASS, as execution context is complex, and models are handled at multiple levels of abstraction. Reproducibility can be supported when provenance of simulation execution is paired with that of individual agent behaviors (see next use case).

3.2.3. Use Case 3: Debug Distributed Execution

In the third case, the researcher mentioned from use case one could verify the conditional logic of an agent's behavior but finds that the corresponding agent's 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 provided by the MASS library monitor framework operations [33] and agent state [34] separately, and dependencies between the two are not captured.

The MASS distributed execution environment, discussed in Section 2.3, hides complex distributed concurrency. Simulation orchestration features hide these resource coordination operations at the framework-level of abstraction, allowing developers to concentrate on modeling resources at the simulation logic abstraction. However, execution details at the framework level may impact agent behavior. In addition, split resource abstraction obstructs methods from relating framework-level operations to model-specific activities.

Common debugging activities such as identifying intermediate data that lead to an exception, and trial-and-error task rerun with varying inputs, motivate provenance capture in DISC systems [35]. However, in the case of distributed execution in MASS, sibling task independence cannot be assumed, and it is necessary to track subtasks and internal agent state. Further, framework operations may be started by 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 verify the agent behaviors described in the first use case.

3.2.4. Use Case 4: Analyze Agent-Spatial Relationships

In the fourth case, a model developer would like to determine which areas agents visit most often. The general rules of the model are clear, as well as the characteristics of places that agents should visit most often. However, the specific place ranges are unknown. To gain a better understanding of required model modifications, the developer would like to reduce the result set to make manual inspection easier. By determining the coordinates of places with the highest amount of agent congregation, the developer can configure provenance capture with place filters to reduce a range of monitored places (explained further in Section 4.4), reducing performance and memory overhead. In turn, finer grained provenance can be captured for the spatial environment and agents.

4. Method

Tracing agent behavior to simulation logic through distributed memory operations within MASS presents various technical challenges. The main challenge is the ability to connect data generated and used at multiple levels of abstraction. This is especially difficult 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 section details these challenges, along with corresponding novel solutions offered by the ProvMASS technique.

4.1. Challenge: Consistently Identifying Distributed Resources

MASS coordinates distributed resources (i.e., agents and place data) over a computing cluster as outlined in Section 2.3; 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., between master and slave processes) are delivered at low frequency, those passed in the exchange layer (i.e., messages containing place data or agents) involve many concurrent socket operations. These coordination mechanisms make it difficult to identify distributed resources within applications.

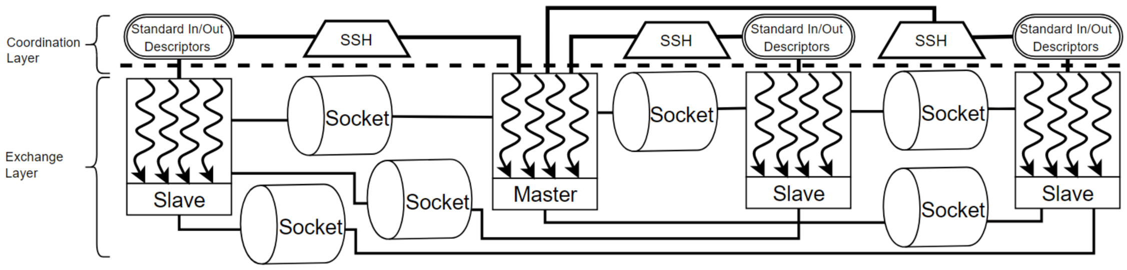


Figure 2. Communication layers for host coordination and data exchange in MASS.

One method to reason about distributed data is to pair read and write operations between hosts via provenance sketches [36]. At the simulation logic level, this involves aliasing system calls to corresponding procedure invocations [37], but aliasing relies on ordering. Consequently, using this method in unordered concurrent communication will likely generate 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 clocks differ.

The ProvMASS approach to address this challenge is unorthodox with respect to other provenance techniques. Rather than using unique identifiers for each instance of the transmitted data, we use the same identifier to label that data in the provenance of both the read operation and the write operation. This 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). To make this technique effective, messages may not be reused; otherwise, causal ordering cannot be guaranteed, due to the possibility of message delays. Following the no message reuse constraint ensures 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 capturing relationships between messages passed, contents of the messages (agents and place data) are also uniquely identified. 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, it is not necessary to check all message-passing operations to determine inter-process dependencies on shared resources (e.g., interactions started by the same agent, but on different hosts). Instead, activities can be temporally ordered to relate them to agents that started them. 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, possible with consistently identified messages).

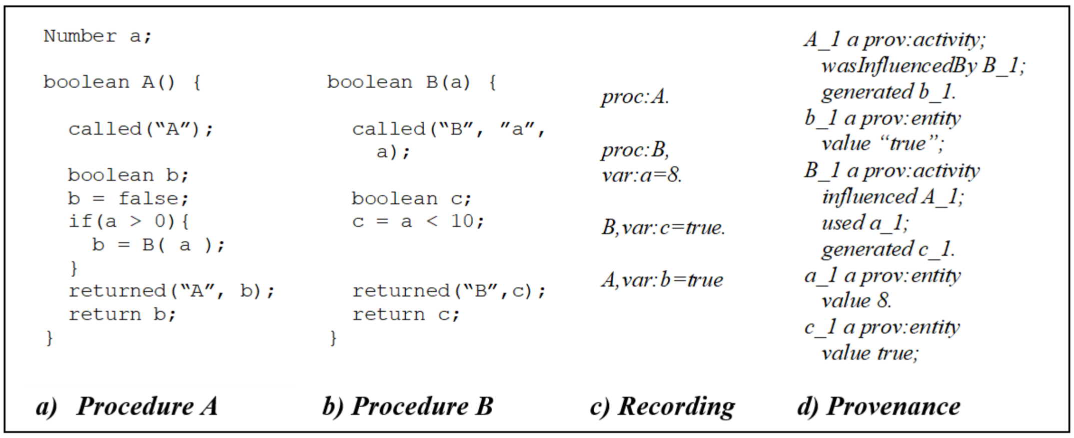
Like some techniques to capture provenance of MapReduce tasks [38–40], the ProvMASS technique transmits provenance with corresponding data to provide consistent identification across network boundaries. However, in these other techniques, the accompanying provenance also identifies tasks that generated or used the data. 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 does not apply to the ProvMASS technique, as such relationships are recorded in situ, rather than formulated at the end of the computation. The resulting 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 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, our approach for consistent distributed resource identification is a better fit for distributed memory environments such as MASS, where inter-dependencies exist between shared resource operations.

4.2. Challenge: Ordering Operations in Shared Memory Programming

Capturing 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. We use a novel resource matching technique to specify these relationships by bridging differing scopes of execution. Resources are not only shared by differing threads of execution, but by physically disparate processes. We first cover a simple example to explain the basic requirements of this approach. Then, we discuss in detail the provenance requirements of distributed memory environments such as MASS.

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 Procedure A. The sequence of events is shown in the recording: Procedure A was invoked; then Procedure B was invoked, with Parameter a that had a value of 8; Procedure B generated Variable c that had a true Boolean value; and Procedure A generated 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 part of the invocation of Procedure A (i.e., Procedure B influenced Procedure A). However, this assumption is only accurate due to sequential procedure invocation.



```
Number a;

boolean A() {

  called("A");

  boolean b;
  b = false;
  if(a > 0){
    b = B( a );
  }
  returned("A", b);
  return b;
}
```

a) Procedure A

```
boolean B(a) {

  called("B", "a",
    a);

  boolean c;
  c = a < 10;

  returned("B",c);
  return c;
}
```

b) Procedure B

```
proc:A.

proc:B,
var:a=8.

B,var:c=true.

A,var:b=true
```

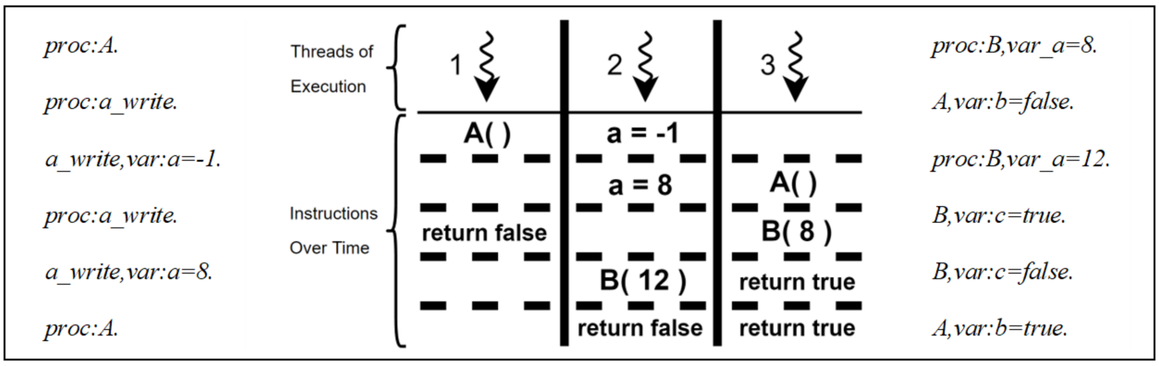
c) Recording

```
A_1 a prov:activity;
  wasInfluencedBy B_1;
  generated b_1.
b_1 a prov:entity
  value "true";
B_1 a prov:activity
  influenced A_1;
  used a_1;
  generated c_1.
a_1 a prov:entity
  value 8.
c_1 a prov:entity
  value true;
```

d) Provenance

Figure 3. Provenance from recording sequential procedure invocation.

Consider the set of procedures in Figure 4, chained in various threads of execution. A sequential recording of these events, ordered from top to bottom and left to right, is also shown. 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 of 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) has been used to pair system calls with corresponding procedure invocations [37].



```
proc:A.
proc:a_write.
a_write,var:a=-1.
proc:a_write.
a_write,var:a=8.
proc:A.
```

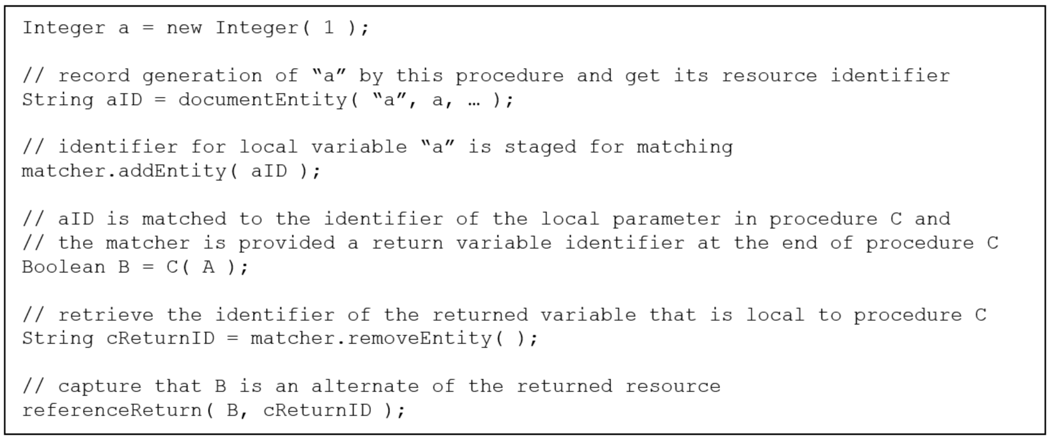
```
proc:B,var_a=8.
A,var:b=false.
proc:B,var_a=12.
B,var:c=true.
B,var:c=false.
A,var:b=true.
```

Figure 4. Recording concurrent invocation in sequence.

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 that, in Figures 3 and 4, determining 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. In addition, 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 clocks vary on different hosts.

To order operations over shared resources in a distributed memory, ProvMASS uses a ResourceMatcher component to aid parallel intra-process capture and storage, avoiding the pitfalls of post-processing 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 do not yet form a graph, threy contain all necessary relationships to construct a provenance graph, including those between field access and assignment and the procedure invocation in which those operations occur.

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 in a step we refer to as automatic source code instrumentation. This process involves gathering identifiers to formulate a provenance capture procedure call for each procedure (i.e., any function within a framework or model class) and field operation (i.e., class-level field access and assignment).



```
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.

The data-matching operations shown in Figure 5 are optional features provided to the developer. Leaving these operations out of automatic source code instrumentation reduces memory and performance overhead by removing the requirement 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. For example, to pair a return to a local variable assignment, the developer can specify within a called procedure (e.g., within Procedure `C`, in the line: `Boolean B = C(A)` shown in Figure 5) that the identifier of the return should be staged in the resource matcher (e.g., `matcher.addEntity (cReturnID)`, similar to `matcher.addEntity(aID)` of the calling procedure shown in Figure 5). After a procedure executes (e.g., after this line executes: `Boolean B = C(A)`), specify within the calling procedure that the staged identifier, `cReturnID`, should be retrieved from the resource matcher (e.g., `matcher.removeEntity()`). This staged identifier is then related to the local variable (e.g., `cReturnID` in the scope of Procedure `C)`. Unfortunately, it is possible that staged and retrieved identifiers may be mismatched, due to user error. For this reason, we plan to partially automate this caller–callee data pairing.

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, sending provenance data and storing it may occur 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 sent to another application or service, with separate memory, that communication is known as "inter-process" (e.g., requests to a database, as described above). Such provenance transmission 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. This detachment introduces extra steps to integrate provenance with other systems. However, performance is a key benefit of intra-process capture. Resource constrained environments, such as MASS, use all processing cores and memory for most operations. Consequently, it is important to avoid hampering parallelism with unwarranted thread interruption. 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 handling incoming requests. This caveat may be avoided by dedicating processing resources to the provenance capture infrastructure. Thread interruptions are reduced with intra-process provenance storage as in-memory storage can be directly accessed by the provenance thread.

Lightweight management of parallel provenance storage can help to maintain concurrency. While intra-process provenance storage 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 performance offered by the MASS framework.

To combat these impediments, ProvMASS uses a 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 2.3). 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 and 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 storage 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 are added to provenance stores as independent triples that are later used to construct a provenance graph. Triples consist of a subject, predicate and object, which 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 [35,41]. While details of the ProvMASS provenance model are discussed further in Section 4.8, independent triples are mentioned here because they not only support delayed graph construction, but also enable 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 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

To achieve acceptable overhead, the design of ProvMASS is based on two simple observations. The first is that levels of provenance capture overhead correspond to processing density (i.e., the rate at which tasks are executed and data are 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 and is directly related 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 configuration mechanisms that tailor provenance capture to specific support scenarios.

The ProvMASS approach allows users to specify the type and granularity of captured provenance (see Figure 6), as well as operations and data of interest. Provenance recording is instrumented into framework operations, many of which involve low-level data manipulation. To limit the amount of provenance captured, the provenance granularity can be adjusted to the following levels: parameter (PaP), return (RP), procedure (PrP), simulation (SP), and process (PP). Meanwhile, procedures, parameters, and returns of agents and places are captured irrespective of the granularity setting. The amount of required provenance is dependent on the analysis task. Generally, procedures of the agents and places are not directly connected within the model definition but by framework operations. However, together, they may be considered part of an agent behavior. PrP granularity is necessary to make these connections. However, to simply verify a specific procedure (i.e., part of the behavior) in an agent model, users only need SP provenance that constitutes the simulation logic leading to the interaction and the interaction itself (i.e., the procedure invocation, parameters, and return). Granularity is specified at runtime to keep the configuration simple but may also be adjusted at any point in the model source code.

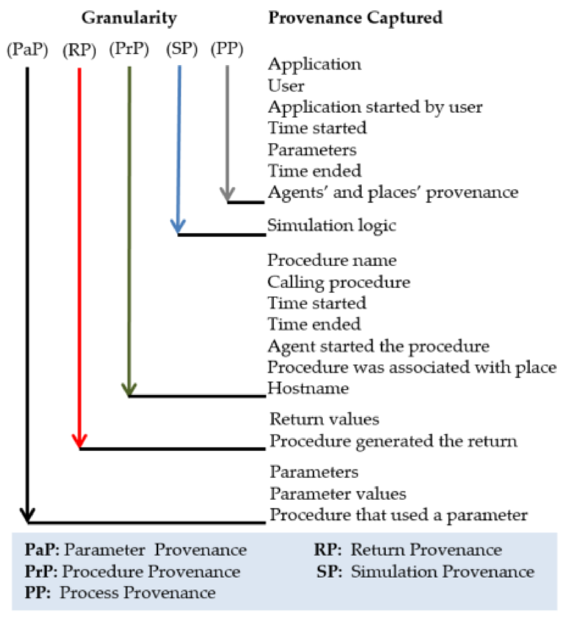


Figure 6. Provenance captured at various levels of granularity.

Provenance can be further limited via a pause feature that stops and starts provenance capture, across all hosts. This allows users to fix the size of the captured provenance. Unlike the other provenance configuration features, the pause feature is added prior to compilation, rather than at run time since it is closely tied to the simulation logic abstraction. Provenance is paused by adding an instruction before the operation(s) of interest and resumed by adding an instruction after.

We provide agent and place filters configured at runtime to focus provenance capture to limit the size of the provenance. Some analyses can be carried out on a subset of agents and/or places. The filters are highly configurable and closely tied to agent mapping. These filters include logical identifier, range of places where agents start, fraction of total population, and several other common mapping patterns. In addition, just as developers can specify a mapping procedure to distribute agents during initialization, they may also specify a custom filter class to extend these default filter options.

Unlike granularity adjustments, filter configurations must be specified at runtime otherwise defaulting to capture the full range of agents' and places' provenance.

To support use cases 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). In subsequent analysis, provenance filters for specific places and agents can be applied, while granularity is reduced to the SP level for framework operations. Total simulation replicability is impossible but such an approach supports reproducibility [31] by providing a sample ABM profile, fine-grained samples of agent interaction, and a coarse-grained 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 focus on modeling agents and places rather than distributed memory management. This feature also allows simulations to scale up to include more physical resources, increasing performance without affecting the model specification. While low-level execution detail is abstracted away to support machine unawareness, operations at the framework abstraction become opaque, making it difficult to understand 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. In such a case, 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 reason for a failed migration.

To further illustrate the disconnect between layers of abstraction, consider a guarded 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 7. Guarded agent migration is a MASS feature that allows developers to model systems in which one or fewer agents may occupy a place at one time. 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 AgentN has already occupied PlaceN in the global places array. This failure may stem from a misconception about the initial mapping of the agents. Another cause of the failure may be that the logic used by agents in making migration decisions is outdated (i.e., after the dimensions of the places collection have changed in the simulation logic). However, this information is unavailable to procedures at the application-level of abstraction. Consequently, this information is also absent from the provenance record generated from those procedures.

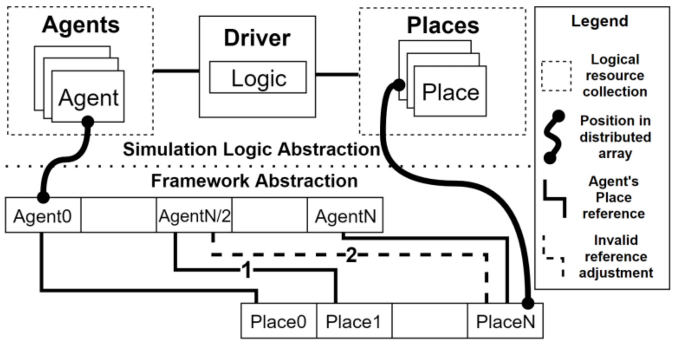


Figure 7. Cause of agent migration failure hidden from simulation logic abstraction [10].

In a broader sense, bridging abstraction layers provide the opportunity to overcome many issues related to the differences that arise in a distributed environment. For example, in a fault tolerant traffic simulation, consider how a failed host might be handled to continue a simulation. In this situation, locations stored on the failed host may be treated as traffic accidents, road closures, or natural disasters. Results of migration attempts may indicate the failure, accounted in the provenance record by tracing a call by the agent (e.g., AgentN/2 in Figure 7) to the framework-level migration operation. The logical obstruction (e.g., a road closure at PlaceN) can be determined, as well as the physical failed host (based on the index of the location as stored in the distributed environment, i.e., index N in the global array shown at the very bottom of Figure 7). Once the underlying representation is obtained, the problem becomes one of distributed debugging, using the message identification techniques described in Section 4.1. Hence, the provenance can be used to distinguish between a technical issue (failed node) and a simulated failure.

It should be noted that the need for the analyses described above must be anticipated. Identification of a failed host can be made at PrP granularity, while a thorough investigation of the remote error tying the parameters to the intended migration location requires PaP granularity (see Section 4.4). Selection of provenance granularity is a requirement but is set simply, at runtime. Other optional configuration features (i.e., filtering and pausing discussed in Section 4.4) may be used to mitigate some performance overhead added by capturing finer grained provenance but require additional setup. While some additional effort is required to setup ProvMASS for this purpose, it provides information that is otherwise unavailable (e.g., message passing delay or failure, insufficient memory, and other errors on the remote hosts).

ProvMASS provides a technique 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, in Figure 8, 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`, possibly a call across host boundary. This procedure then calls another procedure, `decideNewPosition`, at the simulation-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. In addition, the provenance 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 are also 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.

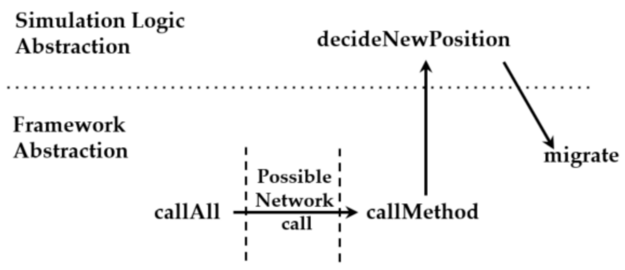


Figure 8. Operations at different levels of abstraction.

4.6. Distributed Provenance Capture Architecture

ProvMASS hybridizes the main-program and subroutines architectural style with client–server 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 [42,43]. 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 the 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.

The ProvMASS architecture is shown in Figure 9, with the components and connectors distributed between 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.

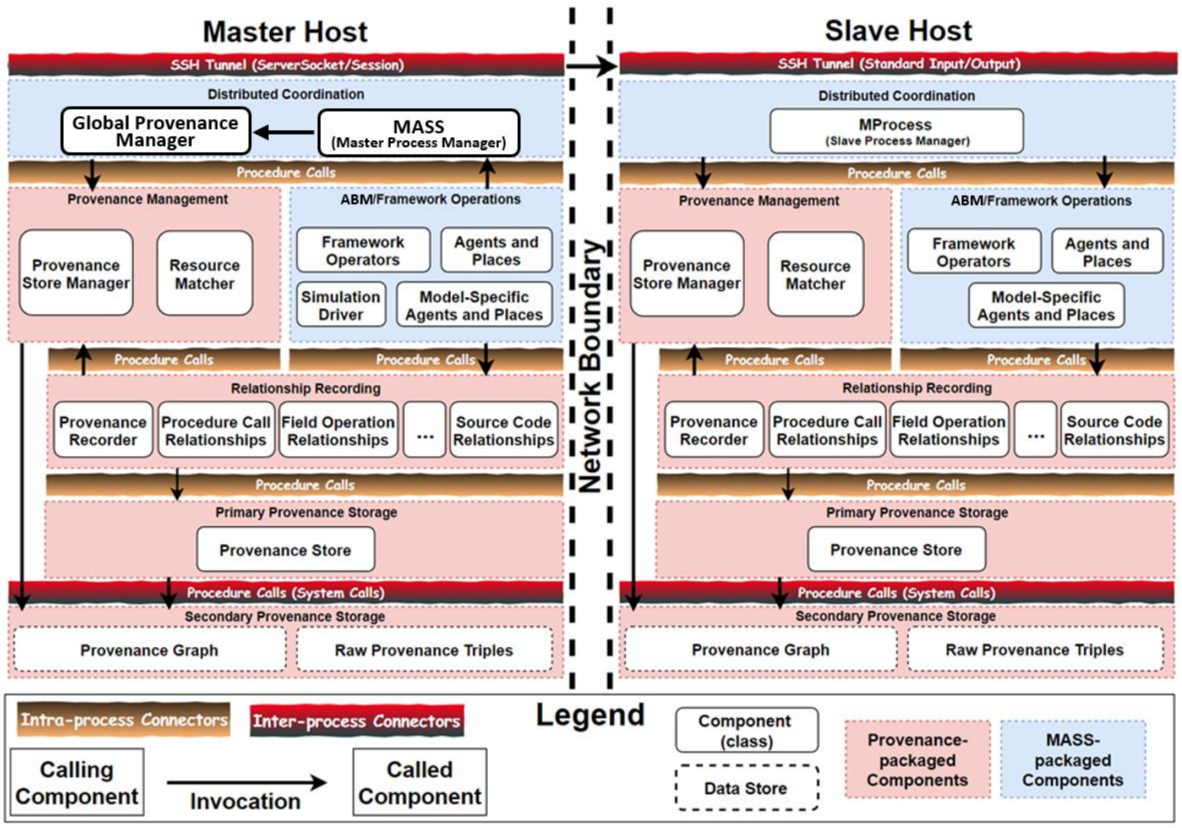


Figure 9. Architecture to distribute parallel provenance capture and storage.

Processes are coordinated in the master–slave paradigm via SSH-tunneling. The master process on the left connects to one or more slave processes shown on the right through an SSH server socket.

Future communication occurs through this connection in the master process but is tunneled to the slave processes through standard input and output descriptors.

ProvMASS components include the Global Provenance Manager, ProvenanceStores, ProvenanceStoreManager, Provenance Recorder, and ResourceMatcher. ProvMASS is initialized during MASS initialization through the Global Provenance Manager. Provenance capture is accomplished via calls to the ProvenanceRecorder. These calls are made in parallel by storing provenance in a thread-safe ProvenanceStore component shown in Figure 9. ProvenanceStores are issued by the ProvenanceStoreManager, which handles their instantiation, thread-mapping, and buffer provision. The ProvenanceRecorder provides procedures utilized by the instrumented code described above and by the developer for adding custom provenance capture code. The ResourceMatcher handles staging and provisioning of activity (see Section 4.8) and data identifiers to relate cross-context resources, as described in Section 4.2.

At the lowest level, provenance storage 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.

During initialization, a shared buffer is provisioned, as shown within the top portion of Figure 10. The shared buffer is allocated and apportioned according to user configuration, provided at runtime. 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 ProvenanceStoreManager first looks for previously allocated, released ProvenanceStores (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.

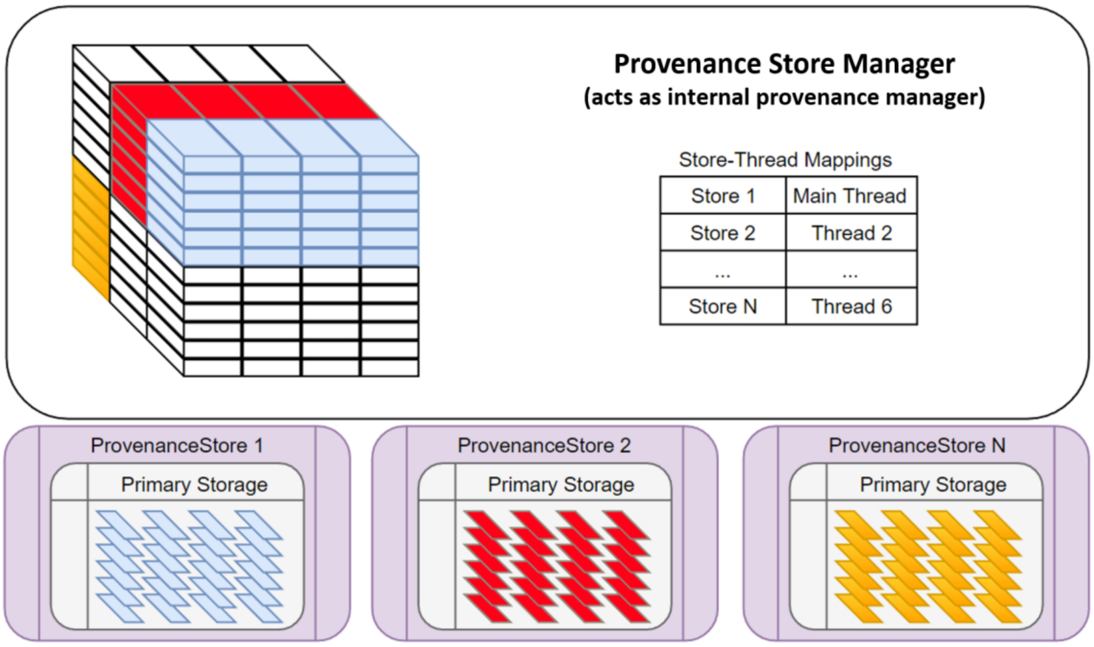


Figure 10. Buffering and thread safety of managed provenance storage.

4.7. ProvMASS Requirements

Requirements of ProvMASS are covered in two parts: those that maintain existing requirements of the MASS environment and those that support provenance capture and use. First, this section presents how ProvMASS maintains performance, scalability, and usability of the MASS environment. This is followed with a discussion of provenance-related properties such as understandability and portability that improve the quality of the provenance data, as well as software-related properties such as reusability, extensibility and adaptability that promote continued use of ProvMASS as the MASS library evolves over time.

4.7.1. MASS-Related Properties

Performance: The purpose of parallelizing ABMs with MASS is to ensure that long-running simulations 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 handled in primary provenance storage with instances of thread-mapped ProvenanceStore components. The primary storage buffer size is configurable in the following ways. First, more memory can be dedicated to buffer size to minimize access to secondary storage. Second, the amount of lines and characters for provenance triples can be configured independently. Global coordination of the primary provenance storage is minimized (e.g., to initialization and system-wide persistence) to prevent interruption of normal host coordination in MASS.

Scalability: ProvMASS supports MASS scalability by distributing provenance storage, while avoiding coordination in its management. This is possible for two reasons. First, data transferred between hosts are labeled with the same identifier on each host (as described in Section 4.1). Second, provenance statements are independent of one another (i.e., as triples consisting of two provenance resources and the relationship between them), making it possible to merge statements at query time (as discussed in Section 4.3). Consequently, the addition of secondary storage (i.e., with each added host) reduces the overall impact of persisting provenance.

Usability: As mentioned before, MASS components exhibit location transparency in that they are shielded from the details of their deployment. Similar to 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 location 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.

4.7.2. 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 a local copy of a neighboring place (i.e., shadow-space) must differentiate that copy from the actual neighboring place. Meanwhile, individual agents must be distinguished in provenance capture for the spatial operations they initiate. While these relationships may be nuanced in the simulation 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, ProvMASS provides provenance capture operations that build model relationships (e.g., agent migration, interaction with a place, spawning, termination, etc.) and source code relationships (e.g., agent started a procedure at a host location) from common composite provenance relationships (e.g., an activity generated data derived from other data), all of which are available to model developers.

ProvMASS provides reusable provenance generation components. This allows 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. 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 execution environment nor an understanding of the provenance model.

Extensibility: Several interfaces are provided to support extensibility and to shield developers from underlying ProvMASS implementation. First, the Global Provenance Manager allows developers to make configuration changes without directly contacting the ProvenanceStoreManager. Such decoupling shields user applications from any internal ProvMASS changes. Second, the ProvenanceRecorder provides an interface for recording common high-level relationships in corresponding source code. This interface provides access to low-level relationships that can be used in isolation or combined to form more complex provenance descriptions. This becomes important when describing higher level relationships that vary between ABM classifications (e.g., simple migration in traffic simulation vs. migration with derivation in microorganism reproduction). These interfaces remove the need for the developer to handle direct provenance capture; instead, developers build on top of these interfaces to extend existing idioms to describe composite relationships. 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 [44,45], standard representations can help reduce the complexity of integrating such provenance into a common model [46]. ABM relationships captured by ProvMASS are specified with concepts from the W3C PROV [47]. More specifically, these relationships are specified with ontological concepts from the PROV Ontology [48]. Further, the provenance graph is formatted as a turtle file, making it easily interpretable by various provenance systems.

Understandable Provenance: Provenance generated by ProvMASS provides ABM relationships and contextual information in a semantically rich representation. These ABM relationships correspond to users' understanding of the model and underlying execution environments, and assist with analysis. Understandability is provided through fine provenance granularity which contains actual procedure and variable names, as well as activity locale corresponding to host-names. Not only does understandable provenance support use cases described in Section 3.2, it also supports usability of MASS, as described above.

4.8. Provenance Model

The provenance model used to describe ABM execution and specification is expressed with concepts from the W3C PROV family of documents [47]. Provenance relationships correspond to procedure call semantics and object-oriented resource relationships and are specified with resource description framework (RDF) [49] triples. The triples express the relationships according to semantics defined in the PROV Ontology [48].

4.8.1. Directed Graph

The directed graph, shown in Figure 11, is composed of vertices representing agents, activities, and entities (provenance classes) and edges representing the relationships between them (provenance properties). Triples are added to provenance stores and periodically flushed to secondary storage. In this form, each triple represents two vertices connected by an edge, as comma-delimited plain text, which is made possible by capturing data and activity relationships in situ via resource matching (see Section 4.2) and consistent inter-process resource identification (see Section 4.1). For example, the relationship between a procedure that receives a message sent by another host (`receiveMessage_#id=22a3`) and its return value (`message#id=7y2b`) will be recorded as `receiveMessage_#id=22a3, prov:generated, message#id=7y2b`. Meanwhile the relationship between a procedure (`sendMessage#id=3109`) that sends the previously mentioned message (`message#id=7y2b`) is recorded as `sendMessage#id=3109, prov:used, message#id=7y2b`. Although the messages are located on different hosts, this provenance can be used to form a graph representing ABM application concepts without any additional inference.

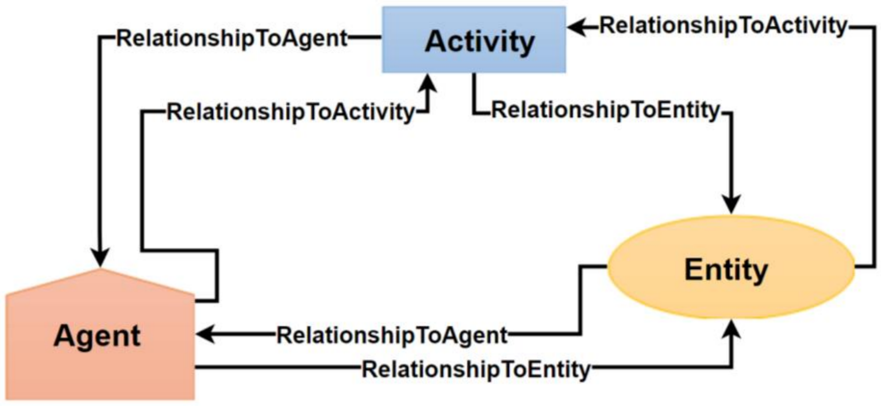


Figure 11. Directed provenance graph.

4.8.2. ABM Concept Mapping with the PROV Ontology

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 ABM execution and specification. The vertices in Figure 11 correspond to starting point classes outlined in Table 1. These classes form the bases for describing resources related to ABMs 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. ABM-related concepts are categorized to provide the closest analogy to corresponding PROV concepts. For example, agents are categorized as SoftwareAgents, rather than Entities despite being objects with methods in execution, because they carry out simulation activities.

PROV Ontology concepts are mapped to ABM software specification and execution in Table 1. 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. Some ABM concepts are better defined by expanded classes. 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, and specified as collections. 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 ABM relationships.

Table 1. Mapping between ABM concepts and PROV Ontology classes.

Provenance Classes	ABM Concepts
Agent	User
Software Agent	Simulator, agents (i.e., ABM entities)
Entity	Fields, parameters, places, returns; source code instructions
Collection	Places, messages; arrays, conditionals and loops (in specification of simulation logic)
Activity	Procedure invocations, field operations (e.g., access and assignment)
Value	Descriptions of the content or state of any above classes; identifiers tying instructions to corresponding activities

Many of the relationships outlined in Table 2 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. Field operations follow these procedure semantics. More specifically, entity generation and use may also correspond to field access and assignment, respectively. However, field operations do not influence another field operation directly. Instead, they influence the procedure activity in which they occur. Location transparency is also important; thus, every procedure invocation is tied to a hostname through the atLocation property. In addition to describing the state of a resource, the value property is also used to describe the connection between a simulation instruction 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.

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. Since individuals of the models are treated as autonomous decision makers, invocation of any agent procedure, or invocation 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 invoked by a given agent or all the agents that invoked a given procedure.

The ProvMASS approach generally relies on causal ordering, rather than temporal ordering. The procedure start time and end time in Table 2 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.

Table 2. Mapping between ABM relationships and PROV Ontology properties.

Provenance Property	**ABM Relationships**
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
used	A procedure used a parameter; A field assignment used a field value
generated	A procedure provided a return; a field access provided a field value
wasStartedBy	A procedure or field operation was started by an agent
value	Any parameter, return, field accessed, or field assignment had a value (if immutable) or reference hashCode (if mutable); Identifiers tying instructions to corresponding activities
startedAtTime	A procedure was started at a system-time (in nanoseconds)
endedAtTime	A procedure ended (just before the 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

5. Results

We evaluate the ProvMASS technique using provenance queries and performance measures. We describe provenance queries for use cases described in Section 3.2, over captured provenance for the SugarScape and RandomWalk simulations, to investigate research question RQ1. The queries have been made available with sample data (see Data S2 in Supplementary materials), as well as the simulations and provenance capture software (see Source Code S1 in Supplementary materials). We also measure performance overhead associated with different provenance capture configurations to answer research question RQ2.

5.1. Provenance Queries

The three types of provenance to support ABM research and development, namely simulation history, model development, and execution environment (refer to Section 3.2), form the basis of our use cases. This section presents provenance queries that support these 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. Results of these queries are displayed in related figures depicting the relevant provenance graph, but 128-bit provenance resource identifiers have been truncated to save space.

5.1.1. UC1: Individual Agent Behavior in SugarScape

Agent activities involved in a single agent behavior may occur in different places (e.g., agent interactions with a neighboring place), possibly on different hosts. ProvMASS simplifies querying the captured provenance, as it does not require users to specify the physical location of agents, as shown in our example. This simple example (following activities of a single agent) serves to demonstrate the capability to reason about agent migration, without the need for specialized queries over distributed data. We provide an example that focuses on querying distributed operations in further detail in Section 5.1.3.

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 agent behaviors can help answer the first type of provenance presented in Section 3.1:

PT1 Can provenance of shared resources be captured in a distributed memory?

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

SugarScape agents are terminated when they run out of food (i.e., energy to keep moving). Thus, we are interested in activities that led the agent to a state in which the kill procedure was invoked. These activities can be determined by querying provenance captured using the SP granularity (see Figure 6).

A condensed view of the captured provenance is shown in Figure 12. To answer the question, the captured provenance is analyzed. 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 identified by filtering the first result of a query on the `wasStartedBy` relationship between agents and activities (see Table 2). Since agents move around looking for food, migration decisions are of interest. Leveraging the `wasStartedBy` relationship between procedure invocations and agents, all migrate activities can be queried. Then, the results can be filtered to `startedAtTime` values (not shown in Figure 12) less than that of the `kill` procedure invocation.

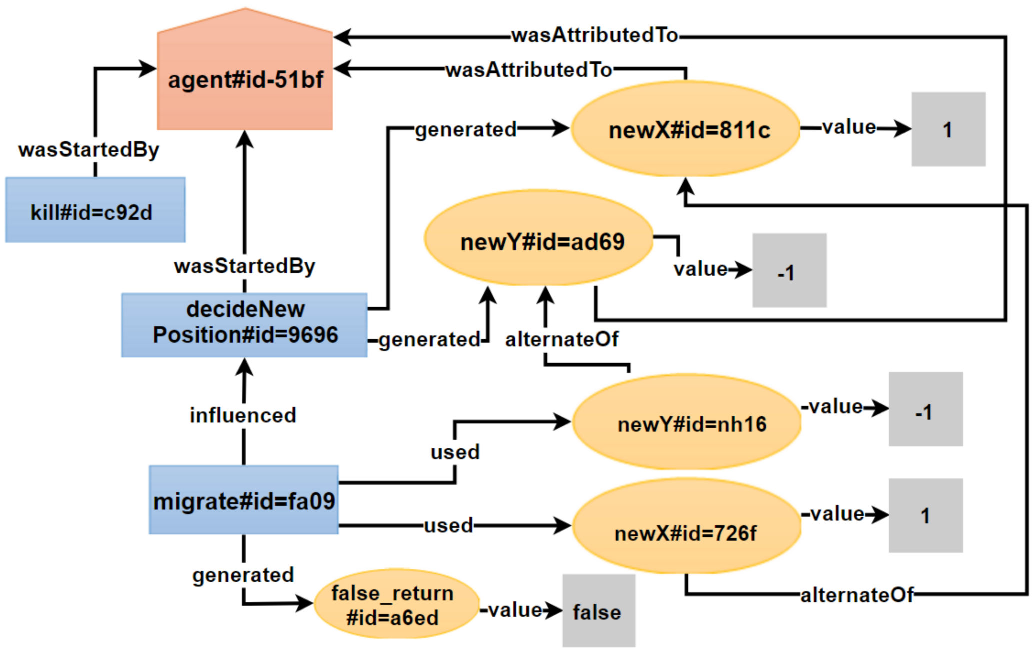


Figure 12. Condensed provenance of individual agent behavior [10].

There are two options to determine the migrate procedure invocation that immediately preceded the invocation of the kill procedure. The first option is to rely on temporal ordering. Since temporal ordering can be used 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. The second option is to backward chain activity influence. When such ordering cannot be relied upon, it is still possible to chain queries of the `influenced` 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 and 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`). At this point in the provenance analysis, it is unknown whether this activity had direct influence over the agent's death. This question can be resolved by examining the conditions that led to the migrate activity.

Once again, the captured provenance is queried to obtain activities in reverse chronological order. The conditions of the `migrate` activity are represented with the `used` relationship as parameters (`newX#id=726f` and `newY#id=nh16`). An examination of the values of these entities 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 by an entity generated by the migrate activity, which has a false Boolean value.

The queries presented in this use case demonstrate the ability to identify a single agent and its related activities and entities, amid many concurrent chains of procedure invocation between many agents migrating across different hosts. The results of these queries indicate that ProvMASS can capture provenance of shared resources (i.e., agents), partially answering provenance type PT1.

PT1 Can provenance of shared resources be captured in a distributed memory?
Partial Answer: This study shows that provenance of shared resources can be captured.

5.1.2. UC2: Simulation Specification and Execution in SugarScape

In this use case, we demonstrate the ability of ProvMASS to connect execution provenance (below the framework abstraction in Figure 7) to simulation source code (at the simulation logic abstraction), different layers of abstraction in MASS applications. These connections support the ability to provide the second type of provenance:

PT2 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 provenance query can be answered by capturing SP provenance granularity. In the instructions that carry out simulation logic, most activities can be traced back to the main simulation loop, including the simulation output. We start our analysis with this control structure to narrow results to simulation instructions, as most of the provenance describes entities and activities from execution. Control structures such as loops are captured as entity collections, with entity members that refer to instructions. Instructions that lead to procedure invocation are given a value that corresponds to the identifier of the activity in execution. Meanwhile, instructions that include variable assignments are given the same identifier as the entity returned by a procedure where available (i.e., when the instruction includes a procedure with a return value).

Figure 13 shows a partial view of the provenance captured for a simulation output. The following analysis illustrates how we answer the provenance query. We first identify the simulation output in the main loop of the source code. In this case, the instruction assigns the return of a procedure, `callAll`, to a variable `callAllResults`. To identify this instruction, we first issue a query to a control structure represented by a for-loop collection (`ForLoop#id=a410`). We query these results to find the `hadMember` property, revealing an instruction that corresponds to a procedure call with assignment of simulation output (`agents.callAll#id=6583`), which has a value that identifies the procedure invocation (`agents.callAll#id=2240`). Two queries on this procedure invocation provide the following types of provenance. The first query reveals the entities (i.e., parameters) used in the invocation (`decideNewPosition_#id=e411` and `agentsCallAllObjects_#id=ae49`). The second query reveals the simulation output that was generated by the invocation (`callAllResult#id=3300`). This output corresponds to the variable assignment. Another way to answer our provenance query in this case is to start with the output (`callAllResult#id=3300`). Then we could reverse these queries to trace the activities and other entities from this variable. However, some simulations may provide output without returning it to the source of the operations (i.e., the simulation driver class). In either case, we make this determination by inspecting the main simulation loop in the source code.

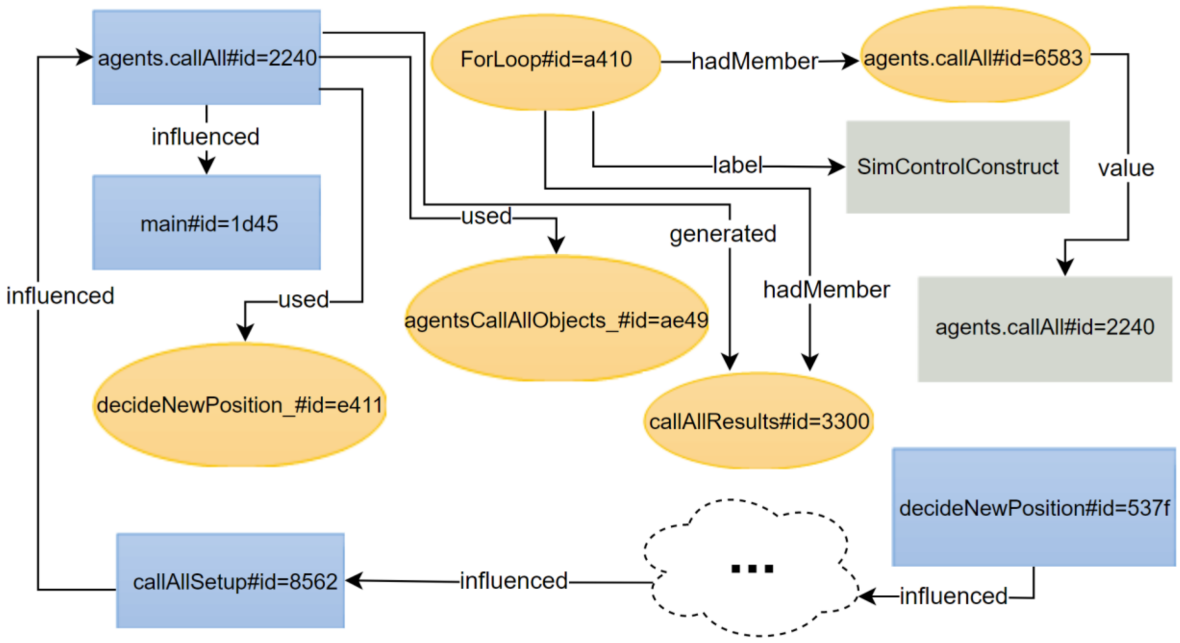


Figure 13. Provenance of simulation logic tied to corresponding execution.

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

The simulation output and corresponding framework activities, answered in the previous provenance query, can also be tied to specific agent interactions. Such interactions between agents, between an agent and its environment, or between the agent and itself take the form of procedure invocations. These invocations start with instructions in the simulation logic which broadcast directives to many agents simultaneously (e.g., `Agents.callAll`) and end with the procedure being invoked by many agents (e.g., `decideNewPosition`). These connections 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 capturing provenance at the PrP granularity.

The captured provenance can be analyzed in the following way. First, one can find procedures that influenced the framework operation, `agents.callAll#id=2240`. This results in a chain of activity dependencies (`callAllSetup#id=8562`, etc.) ending with an agent activity (`decideNewPosition#id=537f`). Obtaining this chain of activity dependencies provides an answer to the second type of provenance:

PT2 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.

5.1.3. UC3: Distributed Execution in RandomWalk

The ability to bridge distributed ABM execution makes specialized queries to reason about distributed operations unnecessary. For queries where ordering is important (e.g., those focused on concurrent activities involving similar entities and agents), direct dependency relationships may be required. In ABM execution, such relationships take the form of procedure call chains, which may be distributed. Queries in the first use case demonstrated the ability to inspect agent behavior by leveraging procedure call relationships. Those in the second 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 third use case fill the gap between the previous two. Agent procedure invocation is connected to corresponding simulation instructions through framework procedure invocations across all hosts. This use case illustrates how provenance type PT1 is addressed with respect to distributed resources (e.g., migrating agents or interactions with remote places):

PT1 Can provenance of shared resources be captured in a distributed memory?

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

This provenance query can be answered by capturing provenance at the PaP granularity. Partial provenance of an operation that measures message delivery delay between the MASS master host process and remote slave processes is shown in Figure 14. In fact, the operation involves 12 host pairs, but provenance of one pair suffices to illustrate the ability to answer this question.

To demonstrate the ability to reason about concurrent distributed events, we analyzed the provenance of a ping-style utility method that measures host delay. This operation simultaneously pings all hosts and awaits responses. The first step is to identify all distributed activities related to this task (e.g., those that begin on one host and end on another). For this operation, each remote host is sent a request message. Upon receiving the request, the host sends an acknowledgment message.

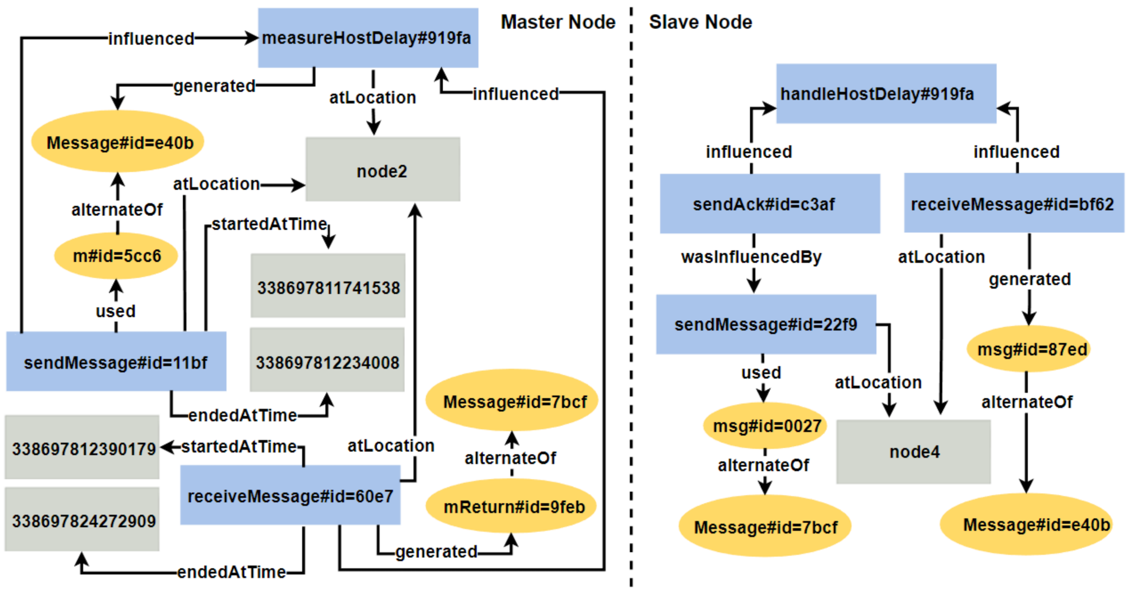


Figure 14. Provenance of an operation to measure message delivery delay [10].

To obtain the related operations across hosts, we examined the `sendMessage` activities that influenced the `measureHostDelay` activity (`measureHostDelay#id=919fa`) in the master host. This examination results in all communication procedure invocations originating from the local host (e.g., `sendMessage#id=11bf` and `receiveMessage#id=60e7`). Using these results, a query on entities used by these activities and a query on the `alternateOf` property results in all sent messages (e.g., `Message#id=e40b`). We can then follow these messages to the slave host by querying their `alternateOf` relationships (an example result is `msg#id=87ed`). The first `alternateOf` query (in the master host) results in messages (consistently identified inter-process entities e.g., `Message#id=e40b` and `Message#id=7bcf`), while the second `alternateOf` query results in local variables (entities specifically tied to procedure invocation e.g., `m#id=87ed` and `mReturn#id=9feb`, respectively). 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`). Similar queries can be made to match acknowledgment messages (e.g., `Message#id=7bcf`) between the slave hosts and the master 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 property of the related `receiveMessage` activities can be obtained. Queries to answer this question result in four sets of message-passing activities for each slave process (e.g., `sendMessage#id=11bf`, `receiveMessage#id=bf62`, `sendMessage#id=22f9`, and `receiveMessage#id=60e7`), 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, we can also answer this provenance query by examining 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 `influenced` 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. Examining the `startedAtTime` and `endedAtTime` properties enables one to find the duration of each round-trip operation. Once the round-trip times are ordered, we can answer Question *b*.

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 regarding the first type of provenance:

PT1 Can provenance of shared resources be captured in a distributed memory?
Partial Answer from UC3: Study shows that provenance of distributed resources can be captured.
Answer: Yes, in addition to capturing shared resource provenance (from UC1), that provenance can be related across various hosts.

5.1.4. UC4: Agent Spatial Relationships in RandomWalk

Where previous use cases demonstrated capability to describe causal data relationships across network boundaries and abstraction layers, this use case demonstrates utility of the provenance data at a higher level. For this use case, we assess support for practical simulation analyses. Specifically, we analyze agents' relationships with the spatial environment. This analysis is also verified through custom control links (provenance captured to count the number of agent visits) added to construction and migration procedures, helping to answer the provenance type PT3:

PT3 Can agent spatial relationships be obtained directly from framework provenance?

Agents are assigned spatial locations when they are initialized and when they migrate. These operations involve both Agents' and Places' procedures. However, relationships between the two occur only through parameter use. In initialization of the Agents collection, for example, a new Agent is provided a reference to its Place through the `setPlace` procedure (which takes a reference to a place as a parameter). Thereafter, the Set of Agents is retrieved from the Place using the `getAgents` procedure and the Agent is added to the Set. Consequently, capturing provenance at the granularity of PaP is required to answer this question directly from framework provenance (i.e., without requiring the control links, mentioned above).

(a) *What Places has an Agent visited?*

Provenance to answer this question is shown in Figure 15. We begin by selecting an Agent (e.g., `Agent#id=0878`). Then, we find `setPlace` activities (e.g., `setPlace#id=a2c1`) with a `wasStartedBy` relationship to the Agent. We then find the parameters used in the `setPlace` activities, and the `alternateOf` relationship. This results in all the places that were visited by an Agent.

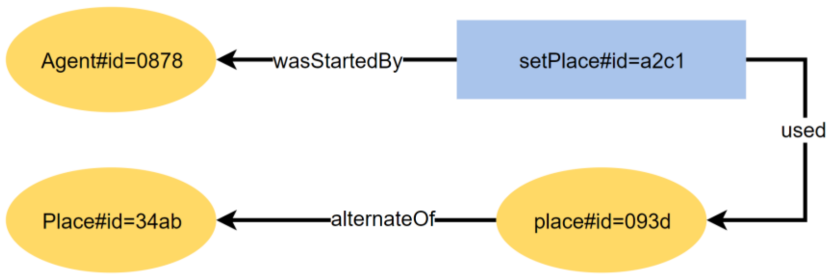


Figure 15. Provenance of the setPlace operation relating an agent to the place it visited.

(b) *How many places did an agent visit?*

Adding to the queries described in Question *a*, we aggregate occurrences of the `setPlace` activity. This provides a count of the places visited (see Figure A3 in Appendix A for the query used and Figure A6 for sample results).

(c) *Which agents visited a place?*

To answer this provenance query, we can also use the provenance shown in Figure 15 by reversing the queries presented in Question *a*. We begin by selecting a `Place` (e.g., `Place#id=34ab`). Then, parameter variables (e.g., `place#id=093d`) used to reference the Place can be found using the `alternateOf` relationship, in the reverse order (i.e., where the Place is the object, not the subject, of the query). These results are further narrowed by focusing on parameters that were used by `setPlace` activities (e.g., `setPlace#id=a2c1`). Last, we look at the `startedBy` relationship to find the Agents (e.g., `Agent#id=0878`) that started the activities.

(d) *How many agents visited a place?*

Building on top of the results in Question *c*, we aggregate occurrences of the `startedBy` relationships between the `setPlace` activities and Agents. This provides a count of the Agents that visited the Place (see Figure A4 for the query used and Figure A8 for sample results).

Provenance of the control links, added to verify the arrival of an agent at a new place, is shown in Figure 16. The `visit` procedure was added just after `Agent` construction and just after the procedures to handle completion of `Agent` migration requests. This helps to indicate any false positive relationships stemming from errant `setPlace` procedure invocations. The `visit` procedure takes references to one `Agent` and one `Place` as parameters. We apply the same automatic provenance capture instrumentation to the visit procedure as all other framework procedures. While these empty visit procedures require manual insertion and knowledge of the framework code, the corresponding provenance is more straightforward to query than corresponding `setPlace` activities (used to answer Questions *a*–*d*). To determine the `Places` an `Agent` visited, we start with an `Agent` (e.g., `Agent#id=0878`) and query for `visit` activities (e.g., `visit#id=733d`). The results from this query is matched with a query to find each `Place` entity (e.g., `Place#id=34ab`) used by the `visit` activity. The queries can be reversed to find the Agents that visited a Place and the results can be aggregated to count the visit occurrences. We compared the results from this analysis with another set of queries and we obtained the same visit counts (see Figures A1 and A2, respectively, for queries used and Figures A5 and A7, respectively, for sample results).

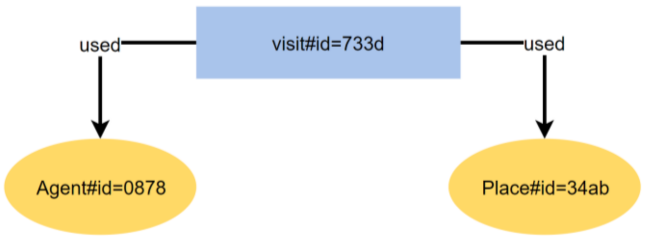


Figure 16. Provenance of an operation to link an agent and the place it visited.

The provenance queries presented in this use case demonstrate the ability to analyze agent migration patterns without relying on provenance of agent behavior (i.e., from execution of a specific agent model). As a result, the captured provenance is simpler; otherwise, source code to extract agent location must be included in the model implementations (e.g., making all agents report a place index at each iteration of the simulation loop). We can extend these queries to include the `atLocation` property of the `setPlace` activity to replace the `getHostName` procedure of RandomWalk agents. In more practical simulations, provenance queries can serve as first-class migration pattern analysis. For example, we can determine how many people moved through an exit point in an evacuation simulation, or the most frequented intersections in a traffic simulation.

Finally, provenance queries in this use case address provenance type PT3:

PT3: Can agent spatial relationships be obtained directly from framework provenance?
Answer: Yes, provenance of agent locale changes from framework execution eliminates the need for corresponding analysis procedures in agent models or provenance of those procedure invocations.

5.2. Performance Overhead Assessment

To assess the performance overhead of the ProvMASS approach and configurable provenance capture features, the run time of SugarScape and RandomWalk simulations are compared for executions with and without provenance capture. We assess the following: agent-scale, pause provenance, and storage size. These studies serve to answer the second research question:

RQ2 Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?

In the following performance comparisons, simulations are conducted in a distributed setting, characterized by similarly configured hosts outlined in Table 3; each with a 1.6 GHz 4-core Intel i7 CPU, 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.

Table 3. 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

5.2.1. Agent-Scale Comparisons

The following comparisons involve increasing the number of places and agents to assess the factor of increased execution duration while applying a configurable provenance capture approach to limit the amount of captured provenance. Provenance of a subset of all agents' interactions is combined with that of the simulation driver, ignoring agent management and other detailed framework operations. This comparison uses the provenance queries in Use Case 1 (Section 5.1.1).

(a) Study Design

The provenance capture configuration used in these comparisons is outlined in Table 4, as follows. Granularity was set to SP (i.e., no procedure invocations were captured in the framework). Procedures, parameters and returns were gathered for 10 agents per host. The additional intra-procedure provenance described in Use Case 1 (i.e., class member read and write operations) was also captured for SugarScape. The pause feature was also applied to limit provenance capture to two operations in each of the two iterations.

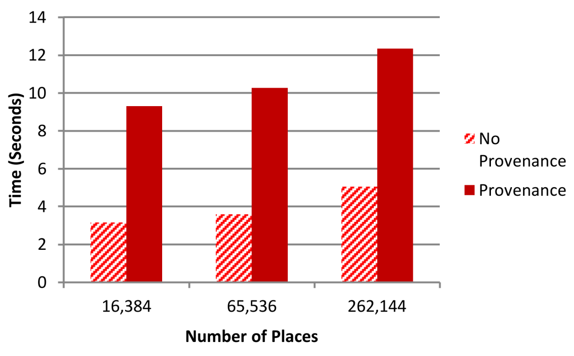
Table 4. Model and provenance configurations for agent-scale comparisons.

Configuration Property	Configuration Setting
Provenance granularity	Simulation-level (SP)
Agent filter	Range by ID filter 10 agents per host (reported results for execution on 12 and 14 hosts)
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), 219 characters per store (222 characters used out of 230 total buffered characters)
Number of places	Scale from 16,384 to 65,536 to 262,144
Number of agents (RandomWalk at Small-Scale)	Scale from 640 to 1280 to 2560
Number of agents (SugarScape)	Scale from 640 to 1280 to 2560
Number of agents (RandomWalk at Full-Scale)	Scale from 16,384 to 65,536 to 262,144
Simulation iterations	25
Total Simulation Operations	77 in RandomWalk/103 in SugarScape
Runs averaged	15

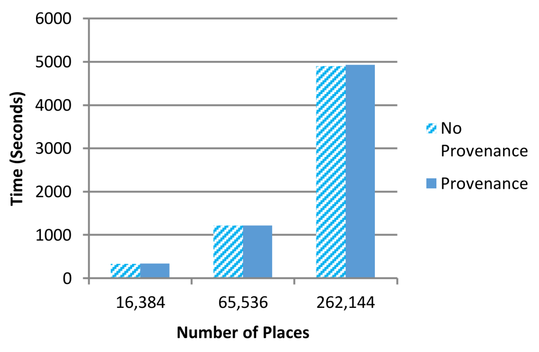
In the full-scale simulation runs for RandomWalk, the number of agents match the number of places, which scaled from 16,384, to 65,536 and finally, 262,144. In these runs, we also compare increased execution duration with and without application of the pause feature. In the small-scale simulations, the number of agents is just a fraction of the number of places, scaling from 640 to 1280 to 2560. Agent mapping to places in the SugarScape simulation is somewhat random. The places configuration matches both RandomWalk comparisons. The number of agents matches those stated for the small-scale RandomWalk runs. The remainder of the configuration parameters were identical for both simulations. The number of simulation iterations was set to 25. With three 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 runs additionally include initialization for neighbor exchange and one exchange operation per iteration, bringing the total count of simulation operations to 103. Runs that utilized the pause feature focused on four simulation instructions. Comparisons for each model were averaged over 15 runs with provenance capture turned on.

(b) Study Results

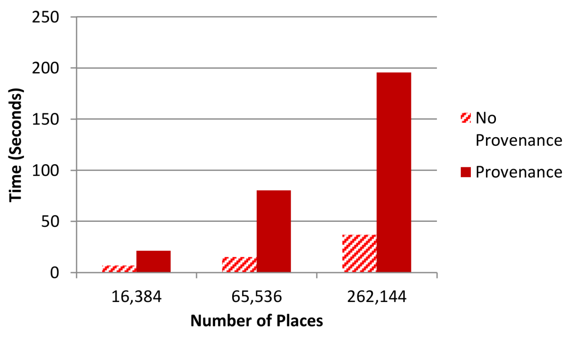
Figure 17 shows increased duration of execution attributed to provenance capture for: (a) the small-scale RandomWalk; (c) full-scale RandomWalk without the pause feature; (d) full-scale RandomWalk with the pause feature; and (b) the SugarScape simulations. The factors of increased execution for SugarScape were minimal (under 4.1%, 3.2% and 2%, decreasing as the number of places increases in the *x*-axis). RandomWalk, on the other hand, presents more variation and higher factor of increase in duration for both types of agent scaling. We analyze the difference between these four result sets further in the next section. For the small-scale RandomWalk the duration of execution attributed to provenance ranges from double to triple the normal execution as the number of places increased. For the full-scale RandomWalk the duration attributed to provenance ranges from triple to a little over quintuple as the number of places increased. Trends in these comparisons indicate execution overhead increases tapered as the model size increased, which is an expected result, as the amount of provenance captured remains the same. This demonstrates capability to throttle performance overhead through configurable provenance capture.



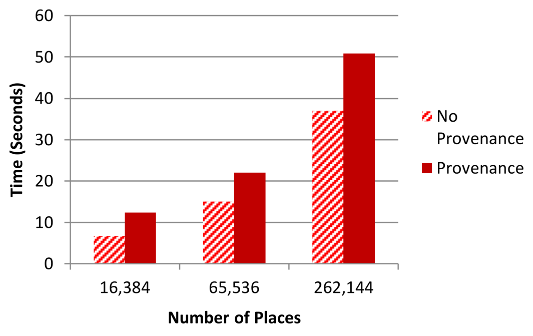
(**a**)



(**b**)



(**c**)



(**d**)

Figure 17. Duration of increased execution as number of places and agents scale: (**a**) RandomWalk simulations scaling to 2560 agents [10]; (**b**) SugarScape simulations scaling to 2560 agents [10]; (**c**) RandomWalk simulations scaling to 262,144 agents without pause feature (on 14 nodes); (**d**) RandomWalk simulations scaling to 262,144 agents with pause feature (on 14 nodes).

(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. This explains why the measured duration of execution escalates at such a rapid pace as the number of places and agents increase. Comparing the two types of scaled RandomWalk simulations shows that for short simulations, the time to allocate the provenance buffer has a larger impact on overhead. It should be noted that everything stated for the smaller number of agents, the small-scale, also holds true for the full-scale comparison, despite the increase in the number of hosts involved in the simulation.

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. This explains why we see such a de-escalation in the increase factor of the execution times. With more agents and places assigned to each host, the host spends more time performing operations than communicating with the other hosts. Consequently, the overhead of capturing agent behavior in SugarScape is relatively low compared to RandomWalk, for this comparison.

RQ2 Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?
Partial Answer: This study shows that overhead can be limited as the number of agents scale, given the nature of each simulation.

5.2.2. Pause Provenance Comparisons

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 presuming 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

Configuration of the models and provenance capture for these comparisons is outlined in Table 5. Simulation runs for both RandomWalk and SugarScape 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. Despite the randomness in agent mapping in SugarScape, the lowest number of agents was 158 and the highest was 161, making simulation configurations very similar with RandomWalk for all runs. Note that this is not the case in the previous comparison (i.e., agent-scale comparison, Section 5.2.1), where variation in model configuration was intended to emphasize differences in processing profile. In this comparison, similarity in model setup is intended to emphasize limitations of our configurable provenance capture, building on model differences noted in the previous comparison. Another substantial difference from the previous comparison 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, the current comparison only narrowed total simulation operations from 43 to 4 in SugarScape. While provenance for the full set of agents is captured in this comparison, this number of agents matches the number of agents used in the agents-scale comparison (10 per host). In effect, the previous comparison provided measures of a more practical profile of configurable provenance capture, while the current comparison hints at upper-boundaries of performance costs for various support scenarios (e.g., analysis tasks that require fine granularity, but few agents and operations or vice versa).

Table 5. Model and provenance configurations for pause provenance comparisons.

Configuration Property *	Configuration Setting
Granularity	Range from process provenance or PP (coarse) to parameter provenance or PaP (fine)
Number of places	1024
Number of agents	160
Simulation iterations	10
Total Simulation Operations	32 in RandomWalk/43 in SugarScape
Runs averaged	20

* Unless shown, configuration properties match those of the previous study outlined in Table 4.

(b) Study Results

Granularity of captured provenance for this comparison varied from process provenance (coarse) to parameter provenance (fine), along the x-axis of Figure 18. Meanwhile, across all configurations and simulations, 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 18. Provenance capture was turned on for the entire simulation without pause feature, for simulations shown in dark blue and dark red. 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 light blue and light red in Figure 18, used the pause feature and captured provenance for two operations within two iterations of the simulation logic, as described in the agent-scale comparison.

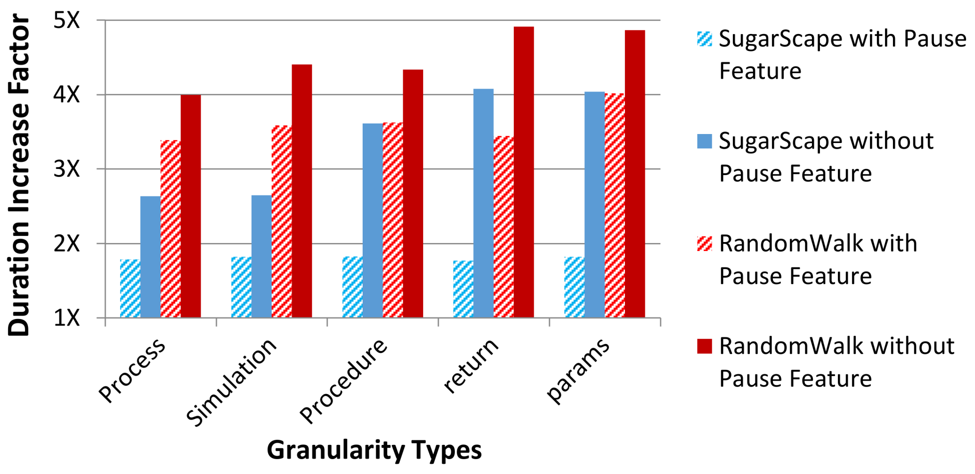


Figure 18. Factors of duration increase, with and without pause feature, on 16 nodes [10].

(c) Study Discussion

As in the previous comparison, these measurements report 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. For example, the simulation runs in this comparison produced up to 136 MB despite running for only 10 iterations (storage sizes are discussed further in the next section). Counterintuitively, the factor of increased run duration was quite higher than in the last comparison for runs with the same granularity-level (simulation provenance). This is primarily a result of total execution time. The previous comparison took around

160 min to complete for the longest runs. Meanwhile, the simulation runs in the current comparison never exceeded 10 min. 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 performance cost considerations, as this study shows.

At the finest granularity levels (i.e., return provenance and parameter provenance), the factor of increased execution has been reduced by over 50% for SugarScape. RandomWalk results indicate less reduction. However, for both simulations, there is less variation between results for different granularities. This indicates a reduction in overhead increase as provenance becomes more fine-grained. Consequently, the pause feature is promising in reducing overhead, where applicable.

5.2.3. Storage Comparisons

We now assess the storage profiles for various provenance capture configurations, to understand how storage size affects the selection of a provenance capture configuration for different models and types of analyses. These studies align with the previous performance measures. For each performance result set, we report the corresponding storage requirements. This study helps to provide more context in answering the following research question:

RQ2 Can provenance capture overhead be limited as the number of agents, space and simulation iterations scale?

(a) Study Design

Configuration of the models and provenance capture for these comparisons is outlined in Table 6. First, we compare the storage size of simulation runs at the granularity of simulation provenance. All simulations runs were conducted with and without the pause feature. The number of places scaled from 16,384 to 65,536 to 262,144 and the provenance data are gathered for 10 agents per host process as shown in Table 6. For both SugarScape and small-scale RandomWalk, the number of agents scaled from 640 to 1280 to 2560.

Next, we compare storage requirements across various levels of granularity. For these simulation runs, the number of places scaled from 16,384 to 65,536 to 262,144 and agents scaled from 640 to 1280 to 2560.

Table 6. Model and provenance configurations for storage comparisons.

Configuration Property	Configuration Setting
Granularity	Simulation provenance (SP); Range from process provenance or PP (coarse) to parameter provenance or PaP (fine)
Agent filter	Range by ID filter 10 agents per host (reported results for execution on 12 and 14 hosts); None
Pause feature	On and off
Provenance buffer	~2 GB total reserved/~256 MB used; 2048 maximum stores per host (8 used);
Number of places	Scale from 1024 and 16,384 to 65,536 to 262,144
Number of agents (RandomWalk Small-Scale)	Scale from 640 to 1280 to 2560
Number of agents (RandomWalk Full-Scale)	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

Finally, we compare the duration of execution time for various storage sizes. Only RandomWalk was used for this comparison. The configuration of these runs match those of the first storage size comparison, but we also extend this to include full-scale RandomWalk, with the number of agents scaling from 16,384 to 65,536 to 262,144.

(b) Study Results

Figure 19 shows the storage requirements for the runs at the granularity of simulation (SP) provenance. In Figure 19, we see the factor of decrease for RandomWalk ranges about 3–3.5 times. For SugarScape, it is about 10–11 times.

Figure 20 shows the storage requirements across various provenance granularities. In Figure 19, we see a dramatic decrease in provenance output when using the pause feature. For RandomWalk, the storage requirement for runs without the pause feature increase by approximately 19 times compared to runs with the pause feature. For SugarScape, storage requirements increase by approximately 118 times compared to runs with the pause feature. Both RandomWalk and SugarScape display a dramatic decrease in provenance output as the granularity of provenance capture becomes coarse.

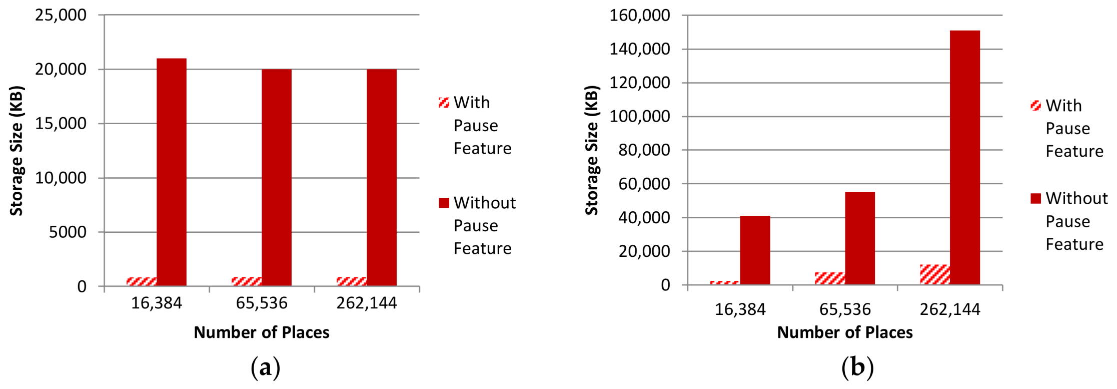


Figure 19. Storage size requirements for simulation runs with agent filtering and toggling the pause feature: (**a**) storage requirements for RandomWalk run with 12 hosts; and (**b**) storage requirements for SugarScape run with 12 hosts.

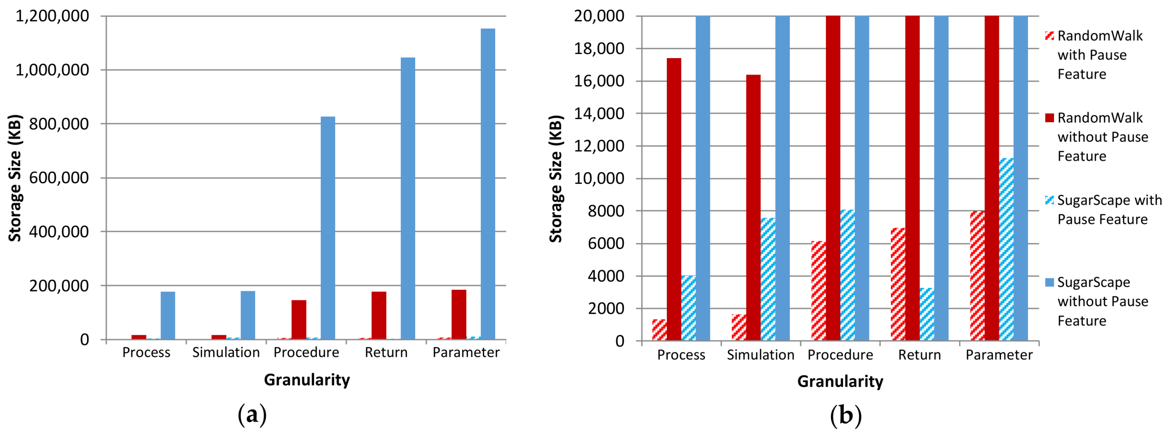
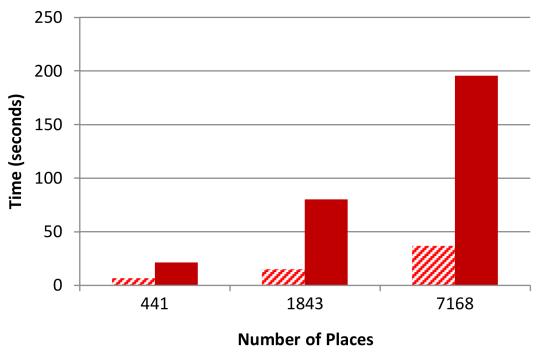


Figure 20. Storage size requirements for simulation runs with various levels of granularity, without agent filtering, and toggling the pause feature for 1024 places: (**a**) high range values, showing all storage requirements; and (**b**) low range values, zooming in on configurations with low storage requirements.

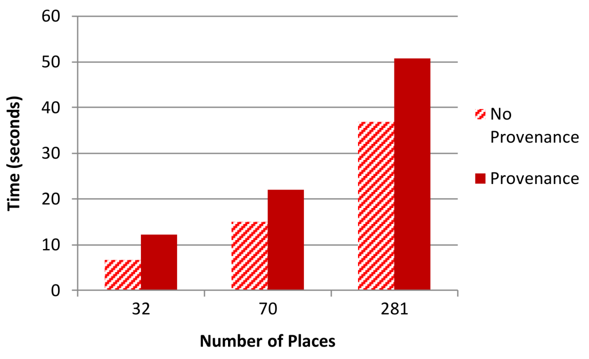
Figure 21 shows the execution for the RandomWalk simulation (Figure 21a) without the pause feature and (Figure 21b) with the pause feature. It also shows (Figure 21c) the storage requirements as the size of places increasing (see Figure A9 in Appendix B for details) and (Figure 21d) the execution time as the size of places increases, both with and without the pause feature. A summary of results is also shown in Table 7. In all cases, execution time increases as storage requirements increase. As the provenance size increases, the execution time increases by an approximate 1.28 times with runs that include the pause feature. Meanwhile, without the pause feature, we see an increase factor of approximately 4.6 times. As the place size increases, storage sizes without the pause feature increase by an average of approximately 21.9 times over runs with the pause feature.

Table 7. RandomWalk simulation on 14 hosts with pause and without pause feature.

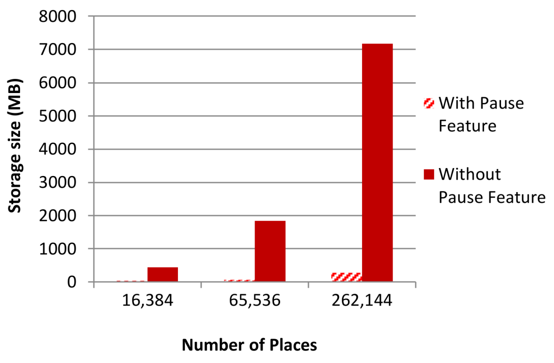
	With pause			**Without Pause**		
Number of places	16,385	65,636	262,144	16,385	65,636	262,144
Number of Agents Vs.	640;	1240;	2560;	640;	1240;	2560;
Storage Size (MB)	32	70	281	441	1483	7168
No Provenance (seconds)	6.68	14.98	36.88	6.68	14.98	36.88
Provenance (seconds)	6.68	22.02	50.72	21.28	80.30	195.59



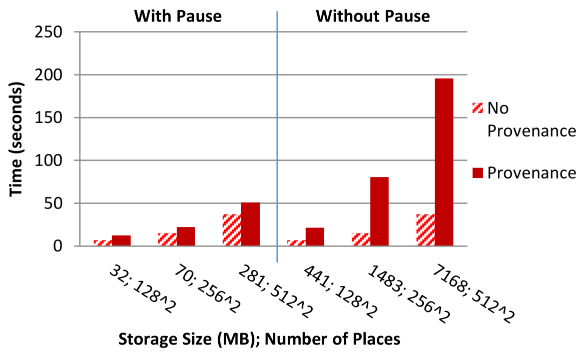
(**a**)



(**b**)



(**c**)



(**d**)

Figure 21. Execution time and storage requirements for RandomWalk runs at the granularity of simulation (SP) provenance: (**a**) execution time for runs with 14 hosts and no pause; (**b**) execution time for runs with 14 hosts with pause; (**c**) storage requirements on 14 hosts with and without pause; and (**d**) execution time comparison of runs with and without pause at specified size and number of places.

(c) Study Discussion

Storage requirements at various levels of provenance granularity can be tempered by applying agent filters and the pause feature. Agent filters reduce provenance related to agent behavior but limit the agent sample population for queries. The pause feature limits provenance to points of interest and fixes the provenance size but eliminates traces that rely on spanning activities between points of interest. The analysis requirements of a simulation determine the provenance capture configuration.

Provenance Storage size may be fixed by focusing on a specific simulation iteration and applying these provenance capture configurations. However, this is not possible for all types of analyses. For example, consider provenance queries to inspect a specific agent state that is only explained at fine granularity with framework provenance. Inspection of these operations requires them to be present in the provenance record. It may be possible to combine traditional analysis with dynamic provenance configuration adjustments (e.g., setting rules to start capturing provenance at fine granularity when a specific state change is detected). However, this is still just a trial and error method. Required provenance may still be absent, especially since we have yet to implement dynamic application of agent filters (i.e., they are applied when agents are initially mapped to places). While capturing provenance for all agents increases the required storage, query support is another issue. Queries on all agents' provenance require a great deal of memory and/or time (depending on the use of stream processing) when applied to many agents. Further, pattern discovery may become cumbersome (i.e., queries become difficult to understand). These factors indicate some limits when answering the research question:

RQ2 Can provenance capture overhead be limited as the number of storage size and size of places scale? **Answer:** This study shows that performance overhead is effectively limited if the provenance storage is also limited. Further, this applies to many, but not all, provenance capture configurations.

5.2.4. 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. 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 scenarios. For example, a researcher might attempt to gain a rough insight into intermediate phenomena by periodically gathering agent state through a utility procedure invoked by 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.

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., granularity at simulation (SP) provenance, and filtering out all agent's and place's provenance). However, utility of such provenance may be reduced.

6. Limitations and Future Work

While key elements of the ProvMASS approach have been demonstrated with queries in Section 5.1, the demonstration does not correspond to practical simulation. The SugarScape and RandomWalk models are simple ABM simulations, to illustrate the basic use cases and constraints of the processing environment. While this helps affirm that the provenance properly describes operations in distributed memory, it does little to attest to scalable reasoning (i.e., as model complexity increases). In other words, we have shown that each part of the relevant questions can be answered but have yet to measure how well this is facilitated. We plan to assess these capabilities in future work with thorough analysis of configuration requirements in a running example (e.g., traffic simulation). Our performance measures and initial investigation into reasoning capabilities indicate that several limitations need to be overcome first, including restrictions on configuration features (i.e., filtering, granularity, and pausing) and query processing.

The ProvMASS approach is limited by constraints placed on the processing environment and corresponding design trade-offs. The greatest limitation corresponds to the need for in-memory storage to maintain concurrency (i.e., the ProvenanceStore component discussed in Section 4.6). 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 (e.g., decreasing the buffer space by half will roughly double the persistence frequency since provenance stores are fully flushed when filled). 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. However, some models may have substantial memory requirements, necessitating a minimal provenance memory configuration. Despite its simplicity, SugarScape required around 5 of 9 GB on each of four hosts, leaving less than half the provisioned application memory for provenance support.

To aid with configuration decisions, heuristics are collected by the provenance management system (i.e., distributed ProvenanceStoreManager component discussed in Section 4.6) 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 configurable provenance capture 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 system-wide 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 minimize using these operations. This can be accomplished by narrowing the query to focus on 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 system-wide 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 Section 2.3, MASS execution constrains system resources. Complex queries may require much memory and the mechanisms to execute them will require free processing cores. Configurable capture provides a workaround in which unnecessary provenance is discarded, resulting in improved query performance.

This paper 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 Section 3.1, the following work remains. Investigation into removing the persistence bottleneck is necessary to support queries that are specific to a group of data but require the full provenance record. 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 is 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 [50]. 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 5.1.1. 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.

7. Related Work

7.1. Provenance Support for Multi-Agent Platforms

A closely related work presented an approach to explain agent behavior with provenance [6] captured in a tool called Provenance in NetLogo (PIN) [51]. 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 [25]. Similar to ProvMASS, this type of provenance is captured through procedure calls inserted through source code instrumentation [6].

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. Other provenance capture techniques do not consider distributed and shared execution context. The core distinction 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.

7.2. Provenance Support 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 [38–41] and Spark [35]. Several techniques add provenance-related identifiers to augment transformation input and propagate provenance through the shuffle stage, pairing map and reduce tasks [38,40]. The benefit of this approach is that data are consistently identified while traversing various execution contexts. However, this approach results in a larger memory footprint and corresponding performance degradation, as the data provenance size grows with each mapping. To reduce this performance degradation, some DISC-related provenance capture techniques employ framework instrumentation (i.e., adding provenance capture statements to framework source code), leveraging access to data mappings between tasks [35,41]. However, data are no longer consistently identified in situ, delaying graph construction and queries. Source code instrumentation also introduces the caveat of delayed query to ProvMASS. However, it does not prevent consistent in situ data identification. Instead of propagating past provenance through current processing stages, ProvMASS employs a resource identifier matching technique (discussed in Section 4.2).

In addition, provenance in DISC systems assumes sibling task independence [35,38,39,41], alleviating the need to account for relationships between tasks, subtasks and intermediate data in the provenance record. Meanwhile, reasoning about operations on distributed shared resources (e.g., agents and spatial data) in the MASS environment requires provenance of these relationships. While DISC systems coordinate fine-grained processing of distributed data, approaches to capture provenance in these systems are not as closely related to capturing provenance of models in distributed memory as those in the shaded rows of Table 8.

Table 8. 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 [6]	250% (includes message passing and persistence in addition to collection)	Multi-Agent Modeling Platforms
Relationships between users and files, System calls (open, close, read, write, exec, fork, exit, clone truncate, and rename) [52]	9–12% over normal execution of the BLAST genome sequencing tool	Distributed System Calls
System calls (from [52] aliased by procedure invocations, primitive parameters and return values [37])	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) [53]	Processing intensive: As low as 1.3% average overhead; system-call intensive: as high as 14% average overhead	Layered Systems
Read and write instructions with corresponding thread identifier as 64-bit word [54]	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 [55]	250–3500% execution time	Symmetric Multiprocessing
Output records (of a stage, shuffle step, etc.), input records, resilient distributed dataset (RDD) transformations that relate them [35]	110–129% execution time	DISC Systems
Record-level derivations [41]	Under 10% on a typical MapReduce workload	DISC Systems
Input and output of MapReduce jobs [39]	20–76% increase for Wordcount and Terasort jobs	DISC Systems
Dependencies between data and actor executions [38]	Under 100% increase in WordCount workflow execution	DISC Systems

In distributed memory operations, concurrent procedures called semantics imply the possibility of interleaving data derivations. Consequently, fine-grained provenance is necessary to describe shared resource operations in a distributed memory, as agent behaviors may be interconnected through shared data operations. Provenance in DISC Systems are not able to support this type of provenance. ProvMASS handles this challenge by employing parallel provenance storage to enable pairing of shared resource operations with their thread of execution. This solution also avoids the bottleneck introduced by external storage that is noted in [35]; 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.

7.3. Provenance for Shared Memory Programming

Provenance capture techniques for shared memory programming (SMP) exist in the context of synchronization constructs, but they do not provide the level of granularity needed in fine grained ABMs [54,55]. In SMPs, intra-process memory operations are captured at the instruction level and can be paired with core-dumps to determine the cause of an error but can be difficult to tie to models in a user-meaningful way. In addition, they do not account for procedure invocation, unlike ProvMASS, which is key in consistently identifying distributed communication.

Provenance in SMPs also do not provide the level of performance needed in MASS. SMP techniques apply provenance capture in stricter memory models than those provided by the MASS library. For instance, one approach replaces PThreads, providing provenance capture at release consistency [55]. 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. Provenance storage is automatically segregated for concurrent capture of resource coordination at the framework-level (i.e., uncoordinated concurrency within barrier synchronizations).

7.4. Distributed Application-Level Provenance

There is also a technique to automatically instrument application-level provenance capture in a distributed setting [37]. While capture of remote communication is only evaluated in server-side procedures of a distributed application, integration with a SPADE [52] reporter service implies the ability to alias network system calls with procedure invocations. At query-time, distributed system calls can be paired [36]. Concurrent procedure invocations are differentiated based on the procedure name, call count, and thread name in [37].

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.

7.5. Connecting Layered Provenance

Integration of provenance across multiple layers of abstraction has been studied in a provenance-aware storage system [53]. 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. ProvMASS, on the other hand, bridges these layers in the parlance of specific ABMs.

8. Conclusions

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

We have demonstrated the ability of ProvMASS to capture provenance during simulation runs of SugarScape and RandomWalk and connect distributed and parallel activities at different layers of abstraction. In addition, ProvMASS can answer specific provenance queries related to three use cases we presented for these models. Finally, we presented the implications of using our configurable provenance collection via various performance measures.

Further study is required to assess the capabilities for reasoning about complex simulations, especially with respect to tradeoffs involved in using different provenance configurations. For example, the amount of instructions in the simulation logic that are required to reason about a collective behavior varies with the model, impacting overhead efficiency affected by the pause feature. However, future work is promising since we have already performed some study of the performance impact of these features. Future studies of in-memory provenance analysis may also illuminate additional reasoning requirements.

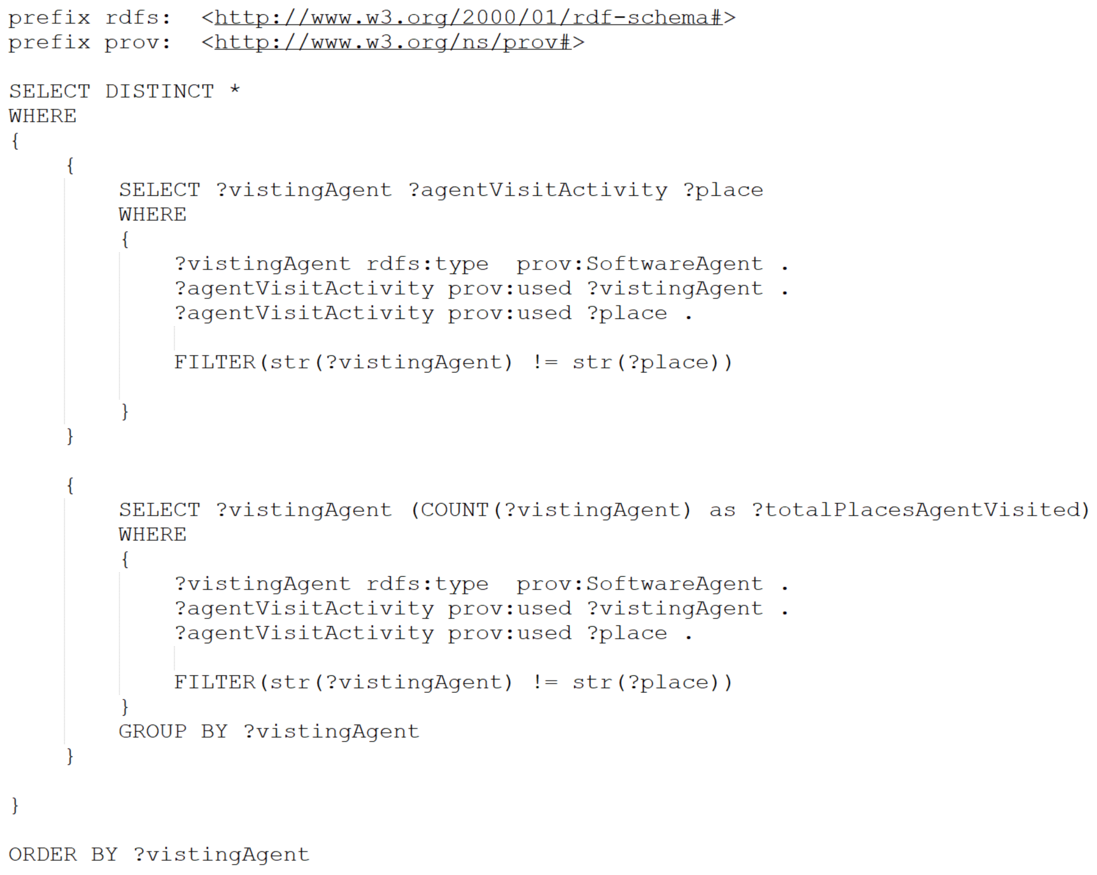
Supplementary Materials: The following is available online at https://bitbucket.org/davisdb1/provmass, Source Code S1: ProvMASS software, http://depts.washington.edu/ptrg/download/608/, Data S2: Queries, Sample Provenance, Sample Results.

Acknowledgments: 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.

Author Contributions: D.B.D. conceived and designed the experiments; D.B.D. and J.F. performed the experiments; J.F., D.B.D, and H.N.V. analyzed the data; M.F. contributed reagents/materials/analysis tools; D.B.D., H.U.A., and J.F. wrote the paper; and M.F. and H.U.A. provided feedback on technical soundness with respect to agent-based systems/distributed parallel computing and data provenance, respectively.

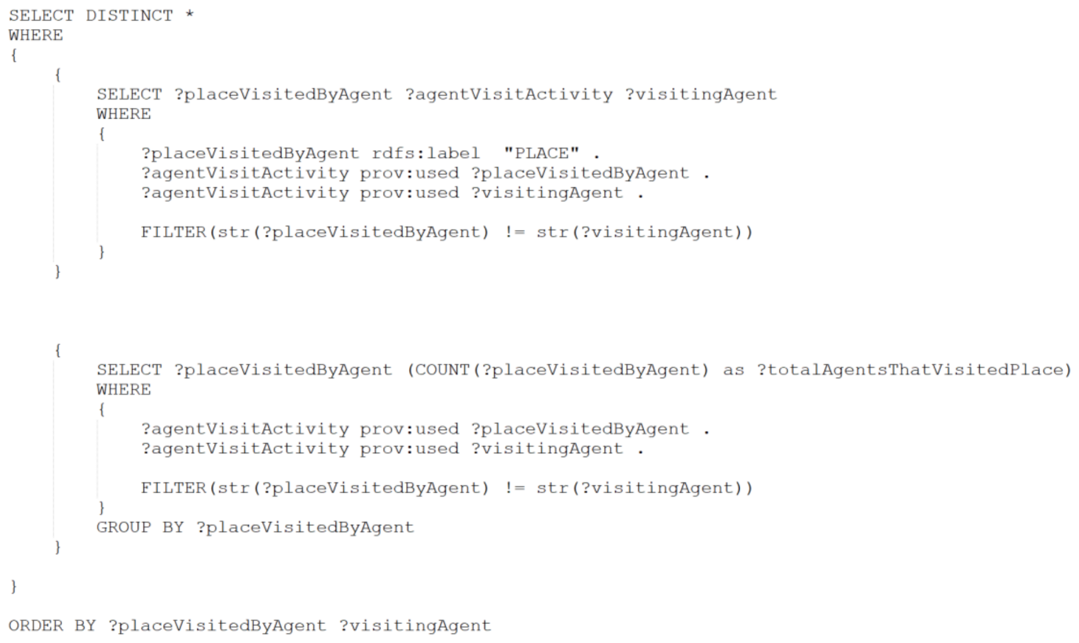
Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Appendix A



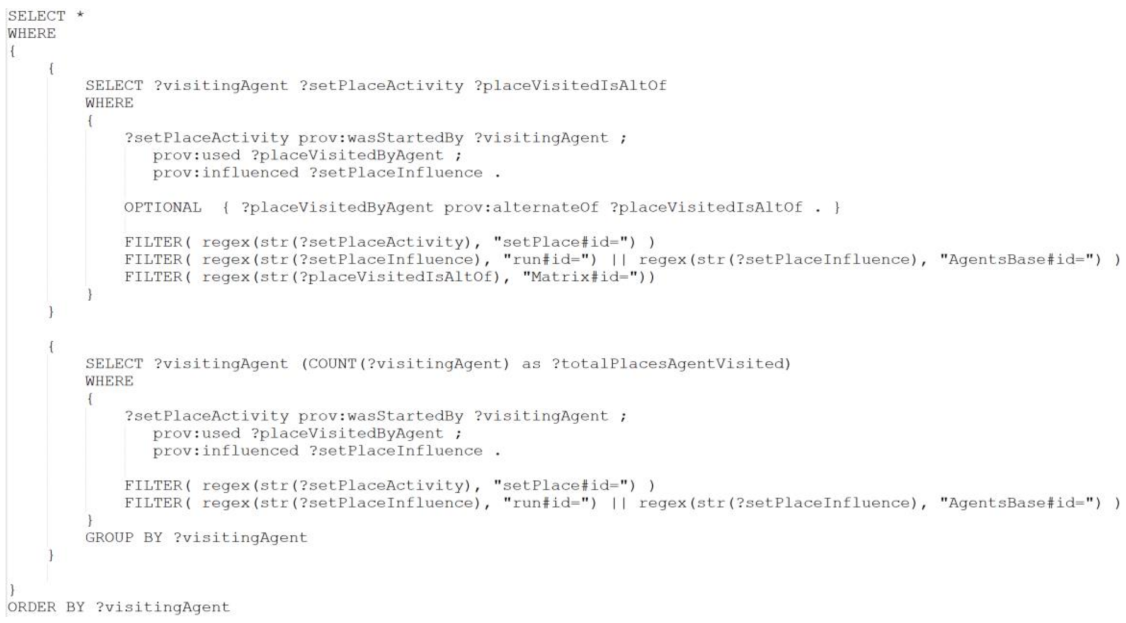
```
prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
prefix prov:  <http://www.w3.org/ns/prov#>

SELECT DISTINCT *
WHERE
{
    {
        SELECT ?vistingAgent ?agentVisitActivity ?place
        WHERE
        {
            ?vistingAgent rdfs:type  prov:SoftwareAgent .
            ?agentVisitActivity prov:used ?vistingAgent .
            ?agentVisitActivity prov:used ?place .

            FILTER(str(?vistingAgent) != str(?place))

        }
    }

    {
        SELECT ?vistingAgent (COUNT(?vistingAgent) as ?totalPlacesAgentVisited)
        WHERE
        {
            ?vistingAgent rdfs:type  prov:SoftwareAgent .
            ?agentVisitActivity prov:used ?vistingAgent .
            ?agentVisitActivity prov:used ?place .

            FILTER(str(?vistingAgent) != str(?place))
        }
        GROUP BY ?vistingAgent
    }

}

ORDER BY ?vistingAgent
```

Figure A1. Query to count the number of places an agent has been, using custom control link.



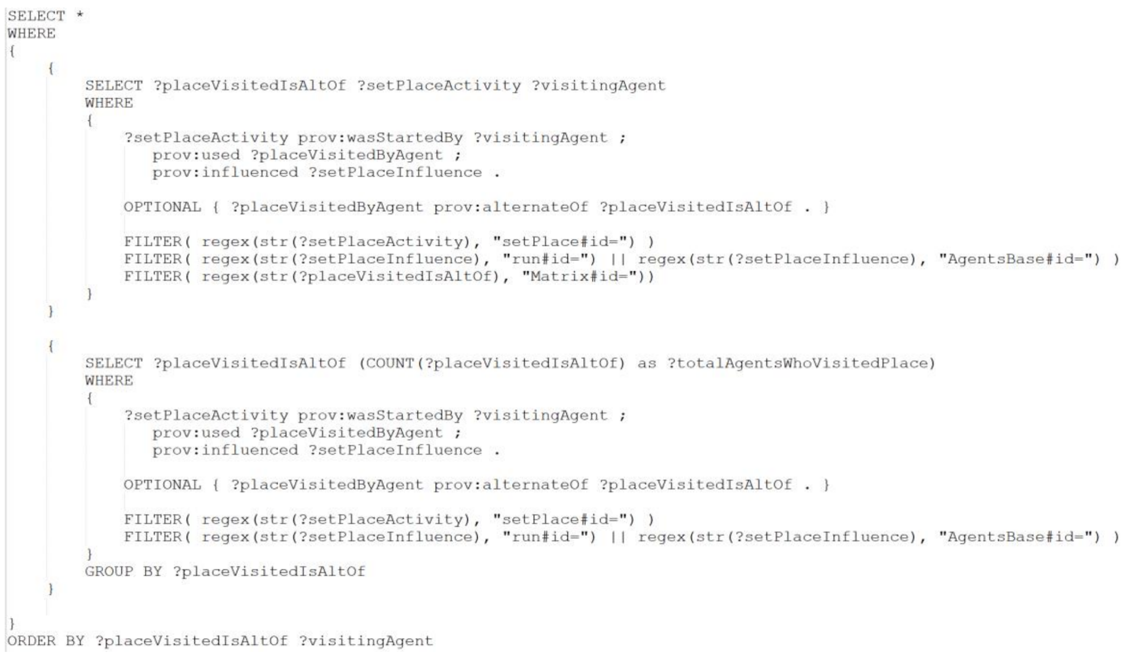
```
SELECT DISTINCT *
WHERE
{
    {
        SELECT ?placeVisitedByAgent ?agentVisitActivity ?visitingAgent
        WHERE
        {
            ?placeVisitedByAgent rdfs:label  "PLACE" .
            ?agentVisitActivity prov:used ?placeVisitedByAgent .
            ?agentVisitActivity prov:used ?visitingAgent .

            FILTER(str(?placeVisitedByAgent) != str(?visitingAgent))
        }
    }


    {
        SELECT ?placeVisitedByAgent (COUNT(?placeVisitedByAgent) as ?totalAgentsThatVisitedPlace)
        WHERE
        {
            ?agentVisitActivity prov:used ?placeVisitedByAgent .
            ?agentVisitActivity prov:used ?visitingAgent .

            FILTER(str(?placeVisitedByAgent) != str(?visitingAgent))
        }
        GROUP BY ?placeVisitedByAgent
    }

}

ORDER BY ?placeVisitedByAgent ?visitingAgent
```

Figure A2. Query to count the number of agents that have visited a place using, custom control link.



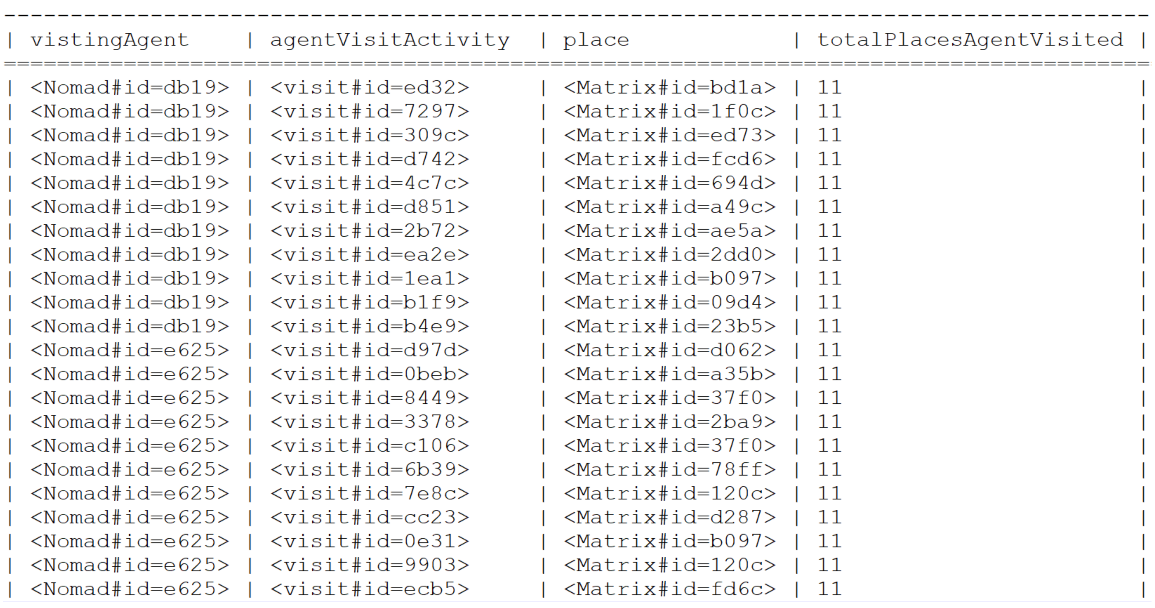
```
SELECT *
WHERE
{
    {
        SELECT ?visitingAgent ?setPlaceActivity ?placeVisitedIsAltOf
        WHERE
        {
            ?setPlaceActivity prov:wasStartedBy ?visitingAgent ;
               prov:used ?placeVisitedByAgent ;
               prov:influenced ?setPlaceInfluence .

            OPTIONAL  { ?placeVisitedByAgent prov:alternateOf ?placeVisitedIsAltOf . }

            FILTER( regex(str(?setPlaceActivity), "setPlace#id=") )
            FILTER( regex(str(?setPlaceInfluence), "run#id=") || regex(str(?setPlaceInfluence), "AgentsBase#id=") )
            FILTER( regex(str(?placeVisitedIsAltOf), "Matrix#id="))
        }
    }

    {
        SELECT ?visitingAgent (COUNT(?visitingAgent) as ?totalPlacesAgentVisited)
        WHERE
        {
            ?setPlaceActivity prov:wasStartedBy ?visitingAgent ;
               prov:used ?placeVisitedByAgent ;
               prov:influenced ?setPlaceInfluence .

            FILTER( regex(str(?setPlaceActivity), "setPlace#id=") )
            FILTER( regex(str(?setPlaceInfluence), "run#id=") || regex(str(?setPlaceInfluence), "AgentsBase#id=") )
        }
        GROUP BY ?visitingAgent
    }

}
ORDER BY ?visitingAgent
```

Figure A3. Query that seeks to count the number of places an agent visited without custom control link.



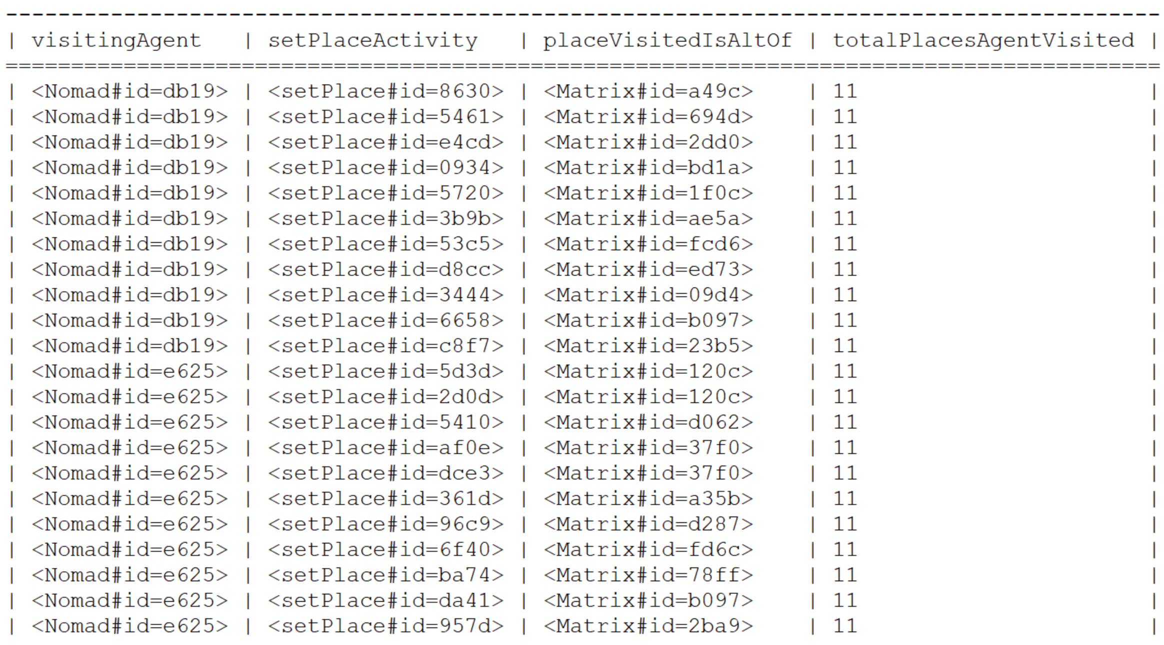
```
SELECT *
WHERE
{
    {
        SELECT ?placeVisitedIsAltOf ?setPlaceActivity ?visitingAgent
        WHERE
        {
            ?setPlaceActivity prov:wasStartedBy ?visitingAgent ;
               prov:used ?placeVisitedByAgent ;
               prov:influenced ?setPlaceInfluence .

            OPTIONAL { ?placeVisitedByAgent prov:alternateOf ?placeVisitedIsAltOf . }

            FILTER( regex(str(?setPlaceActivity), "setPlace#id=") )
            FILTER( regex(str(?setPlaceInfluence), "run#id=") || regex(str(?setPlaceInfluence), "AgentsBase#id=") )
            FILTER( regex(str(?placeVisitedIsAltOf), "Matrix#id="))
        }
    }

    {
        SELECT ?placeVisitedIsAltOf (COUNT(?placeVisitedIsAltOf) as ?totalAgentsWhoVisitedPlace)
        WHERE
        {
            ?setPlaceActivity prov:wasStartedBy ?visitingAgent ;
               prov:used ?placeVisitedByAgent ;
               prov:influenced ?setPlaceInfluence .

            OPTIONAL { ?placeVisitedByAgent prov:alternateOf ?placeVisitedIsAltOf . }

            FILTER( regex(str(?setPlaceActivity), "setPlace#id=") )
            FILTER( regex(str(?setPlaceInfluence), "run#id=") || regex(str(?setPlaceInfluence), "AgentsBase#id=") )
        }
        GROUP BY ?placeVisitedIsAltOf
    }

}
ORDER BY ?placeVisitedIsAltOf ?visitingAgent
```

Figure A4. Query that seeks to count the number of agents that have visited a particular place without custom control link.



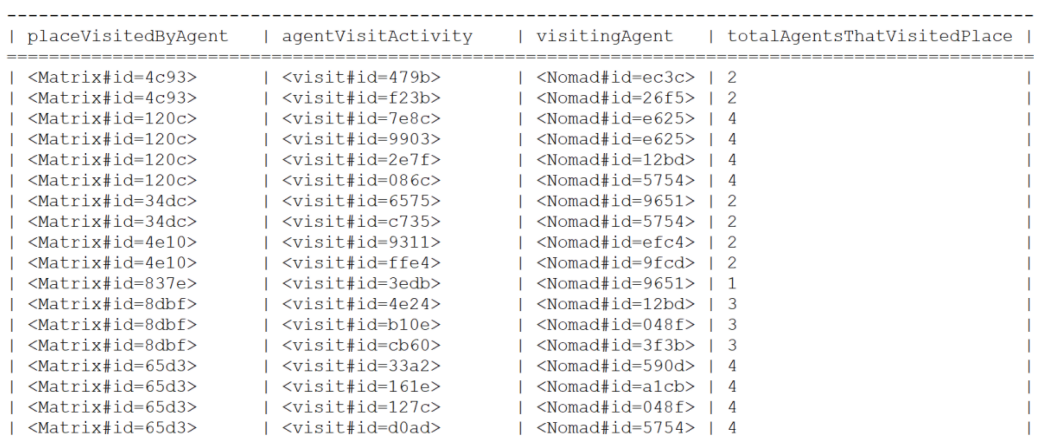
```
------------------------------------------------------------------------------------------
| vistingAgent      | agentVisitActivity   | place              | totalPlacesAgentVisited |
==========================================================================================
| <Nomad#id=db19> | <visit#id=ed32>      | <Matrix#id=bd1a> | 11                        |
| <Nomad#id=db19> | <visit#id=7297>      | <Matrix#id=1f0c> | 11                        |
| <Nomad#id=db19> | <visit#id=309c>      | <Matrix#id=ed73> | 11                        |
| <Nomad#id=db19> | <visit#id=d742>      | <Matrix#id=fcd6> | 11                        |
| <Nomad#id=db19> | <visit#id=4c7c>      | <Matrix#id=694d> | 11                        |
| <Nomad#id=db19> | <visit#id=d851>      | <Matrix#id=a49c> | 11                        |
| <Nomad#id=db19> | <visit#id=2b72>      | <Matrix#id=ae5a> | 11                        |
| <Nomad#id=db19> | <visit#id=ea2e>      | <Matrix#id=2dd0> | 11                        |
| <Nomad#id=db19> | <visit#id=1ea1>      | <Matrix#id=b097> | 11                        |
| <Nomad#id=db19> | <visit#id=b1f9>      | <Matrix#id=09d4> | 11                        |
| <Nomad#id=db19> | <visit#id=b4e9>      | <Matrix#id=23b5> | 11                        |
| <Nomad#id=e625> | <visit#id=d97d>      | <Matrix#id=d062> | 11                        |
| <Nomad#id=e625> | <visit#id=0beb>      | <Matrix#id=a35b> | 11                        |
| <Nomad#id=e625> | <visit#id=8449>      | <Matrix#id=37f0> | 11                        |
| <Nomad#id=e625> | <visit#id=3378>      | <Matrix#id=2ba9> | 11                        |
| <Nomad#id=e625> | <visit#id=c106>      | <Matrix#id=37f0> | 11                        |
| <Nomad#id=e625> | <visit#id=6b39>      | <Matrix#id=78ff> | 11                        |
| <Nomad#id=e625> | <visit#id=7e8c>      | <Matrix#id=120c> | 11                        |
| <Nomad#id=e625> | <visit#id=cc23>      | <Matrix#id=d287> | 11                        |
| <Nomad#id=e625> | <visit#id=0e31>      | <Matrix#id=b097> | 11                        |
| <Nomad#id=e625> | <visit#id=9903>      | <Matrix#id=120c> | 11                        |
| <Nomad#id=e625> | <visit#id=ecb5>      | <Matrix#id=fd6c> | 11                        |
```

Figure A5. Snippet of truncated query results using custom control link data to query the total amount of places an agent has visited.



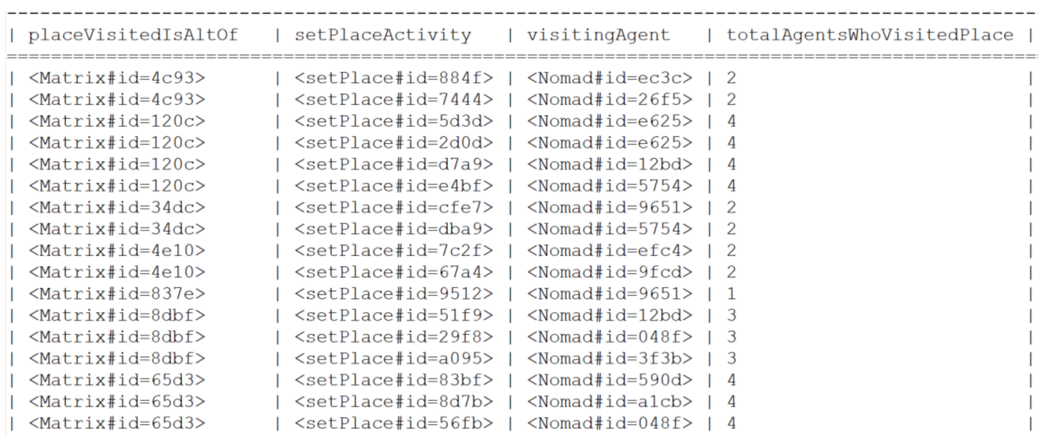
```
------------------------------------------------------------------------------------------
| visitingAgent     | setPlaceActivity     | placeVisitedIsAltOf | totalPlacesAgentVisited |
==========================================================================================
| <Nomad#id=db19> | <setPlace#id=8630> | <Matrix#id=a49c>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=5461> | <Matrix#id=694d>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=e4cd> | <Matrix#id=2dd0>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=0934> | <Matrix#id=bd1a>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=5720> | <Matrix#id=1f0c>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=3b9b> | <Matrix#id=ae5a>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=53c5> | <Matrix#id=fcd6>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=d8cc> | <Matrix#id=ed73>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=3444> | <Matrix#id=09d4>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=6658> | <Matrix#id=b097>    | 11                      |
| <Nomad#id=db19> | <setPlace#id=c8f7> | <Matrix#id=23b5>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=5d3d> | <Matrix#id=120c>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=2d0d> | <Matrix#id=120c>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=5410> | <Matrix#id=d062>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=af0e> | <Matrix#id=37f0>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=dce3> | <Matrix#id=37f0>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=361d> | <Matrix#id=a35b>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=96c9> | <Matrix#id=d287>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=6f40> | <Matrix#id=fd6c>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=ba74> | <Matrix#id=78ff>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=da41> | <Matrix#id=b097>    | 11                      |
| <Nomad#id=e625> | <setPlace#id=957d> | <Matrix#id=2ba9>    | 11                      |
```

Figure A6. Snippet of truncated query results showing the total amount of places an agent has visited without using custom control link.



```
-----------------------------------------------------------------------------------------------
| placeVisitedByAgent    | agentVisitActivity    | visitingAgent    | totalAgentsThatVisitedPlace |
===============================================================================================
| <Matrix#id=4c93>       | <visit#id=479b>       | <Nomad#id=ec3c> | 2                           |
| <Matrix#id=4c93>       | <visit#id=f23b>       | <Nomad#id=26f5> | 2                           |
| <Matrix#id=120c>       | <visit#id=7e8c>       | <Nomad#id=e625> | 4                           |
| <Matrix#id=120c>       | <visit#id=9903>       | <Nomad#id=e625> | 4                           |
| <Matrix#id=120c>       | <visit#id=2e7f>       | <Nomad#id=12bd> | 4                           |
| <Matrix#id=120c>       | <visit#id=086c>       | <Nomad#id=5754> | 4                           |
| <Matrix#id=34dc>       | <visit#id=6575>       | <Nomad#id=9651> | 2                           |
| <Matrix#id=34dc>       | <visit#id=c735>       | <Nomad#id=5754> | 2                           |
| <Matrix#id=4e10>       | <visit#id=9311>       | <Nomad#id=efc4> | 2                           |
| <Matrix#id=4e10>       | <visit#id=ffe4>       | <Nomad#id=9fcd> | 2                           |
| <Matrix#id=837e>       | <visit#id=3edb>       | <Nomad#id=9651> | 1                           |
| <Matrix#id=8dbf>       | <visit#id=4e24>       | <Nomad#id=12bd> | 3                           |
| <Matrix#id=8dbf>       | <visit#id=b10e>       | <Nomad#id=048f> | 3                           |
| <Matrix#id=8dbf>       | <visit#id=cb60>       | <Nomad#id=3f3b> | 3                           |
| <Matrix#id=65d3>       | <visit#id=33a2>       | <Nomad#id=590d> | 4                           |
| <Matrix#id=65d3>       | <visit#id=161e>       | <Nomad#id=a1cb> | 4                           |
| <Matrix#id=65d3>       | <visit#id=127c>       | <Nomad#id=048f> | 4                           |
| <Matrix#id=65d3>       | <visit#id=d0ad>       | <Nomad#id=5754> | 4                           |
```

Figure A7. Snippet of truncated query results using custom control link to obtain the total number of agents that visited a place.



```
-----------------------------------------------------------------------------------------------
| placeVisitedIsAltOf    | setPlaceActivity     | visitingAgent    | totalAgentsWhoVisitedPlace |
===============================================================================================
| <Matrix#id=4c93>       | <setPlace#id=884f> | <Nomad#id=ec3c> | 2                            |
| <Matrix#id=4c93>       | <setPlace#id=7444> | <Nomad#id=26f5> | 2                            |
| <Matrix#id=120c>       | <setPlace#id=5d3d> | <Nomad#id=e625> | 4                            |
| <Matrix#id=120c>       | <setPlace#id=2d0d> | <Nomad#id=e625> | 4                            |
| <Matrix#id=120c>       | <setPlace#id=d7a9> | <Nomad#id=12bd> | 4                            |
| <Matrix#id=120c>       | <setPlace#id=e4bf> | <Nomad#id=5754> | 4                            |
| <Matrix#id=34dc>       | <setPlace#id=cfe7> | <Nomad#id=9651> | 2                            |
| <Matrix#id=34dc>       | <setPlace#id=dba9> | <Nomad#id=5754> | 2                            |
| <Matrix#id=4e10>       | <setPlace#id=7c2f> | <Nomad#id=efc4> | 2                            |
| <Matrix#id=4e10>       | <setPlace#id=67a4> | <Nomad#id=9fcd> | 2                            |
| <Matrix#id=837e>       | <setPlace#id=9512> | <Nomad#id=9651> | 1                            |
| <Matrix#id=8dbf>       | <setPlace#id=51f9> | <Nomad#id=12bd> | 3                            |
| <Matrix#id=8dbf>       | <setPlace#id=29f8> | <Nomad#id=048f> | 3                            |
| <Matrix#id=8dbf>       | <setPlace#id=a095> | <Nomad#id=3f3b> | 3                            |
| <Matrix#id=65d3>       | <setPlace#id=83bf> | <Nomad#id=590d> | 4                            |
| <Matrix#id=65d3>       | <setPlace#id=8d7b> | <Nomad#id=a1cb> | 4                            |
| <Matrix#id=65d3>       | <setPlace#id=56fb> | <Nomad#id=048f> | 4                            |
```

Figure A8. Snippet of truncated query results showing the total number of agents that visited a place without using custom control link.

Appendix B

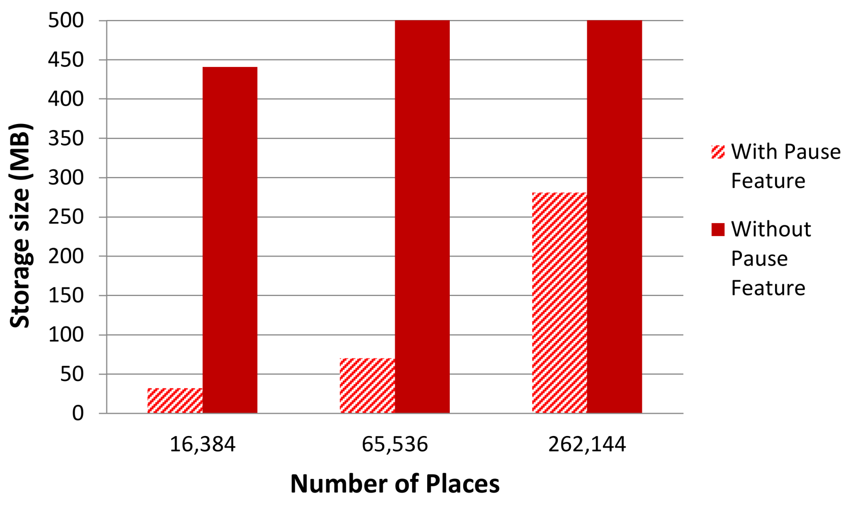


Figure A9. Low *y*-axis version for chart in Figure 21c.

References

1. Ferber, J. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, 1st ed.; Addison-Wesley Reading: Boston, MA, USA, 1999; ISBN 978-0-201-36048-6.
2. Ferber, J. Simulating with reactive agents. In *Many-Agent Simulation and Artificial Life*; Hillebrand, E., Stender, J., Eds.; IOS Press: Amsterdam, The Netherlands, 1994; pp. 8–30.
3. Stevens, R.; Zhao, J.; Goble, C. Using provenance to manage knowledge of in silico experiments. *Brief. Bioinform.* **2007**, *8*, 183–194. [CrossRef] [PubMed]
4. Bose, R. A conceptual framework for composing and managing scientific data lineage. In Proceedings of the 4th International Conference on Scientific and Statistical Database Management, Melbourne, Australia, 21–24 January 2003; IEEE: Edinburgh, UK, 2002.
5. Buneman, P.; Khanna, S.; Tan, W. Why and Where: A Characterization of Data Provenance. In *Database Theory—ICDT 2001, Proceedings of the 8th International Conference on Database Theory*; Springer: Berlin/Heidelberg, Germany; London, UK, 2001; pp. 316–330.
6. Chen, P.; Plale, B.; Evans, T. Dependency Provenance in Agent Based Modeling. In Proceedings of the 9th International Conference on eScience, Beijing, China, 22–25 October 2013; pp. 180–187. [CrossRef]
7. An, L. Modeling human decisions in coupled human and natural systems: Review of agent-based models. *Ecol. Model.* **2012**, *229*, 25–36. [CrossRef]
8. Moreau, L.; Ludascher, B.; Bertram, A.; Ilkay, B.; Roger, S.; Bowers, S.; Callahan, S.; Chin, G.; Clifford, B.; Cohen, S.; et al. Special Issue: The First Provenance Challenge. *Concurr. Comput. Pract. Exp.* **2008**, *20*, 409–418. [CrossRef]
9. Davis, B.D. Data Provenance for Multi-Agent Models in a Distributed Memory. Master's Thesis, University of Washington, Bothell, WA, USA, 2017.
10. Davis, B.D.; Featherston, J.; Fukada, M.; Asuncion, H.U. Data Provenance for Multi-Agent Models. In Proceedings of the 13th International Conference on eScience, Auckland, New Zealand, 24–27 October 2017; pp. 39–48. [CrossRef]
11. O'Sullivan, D.; Perry, G.L. *Spatial Simulation: Exploring Pattern and Process*, 1st ed.; John Wiley & Sons: Hoboken, NJ, USA, 2013; ISBN 978-1-119-97079-8.
12. Dewdney, A.K. Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Sci. Am.* **1984**, *251*, 14–22. [CrossRef]
13. Epstein, J.M. Agent-based computational models and generative social science. *Complexity* **2011**, *4*, 41–60. [CrossRef]
14. Balmer, M.; Rieser, M.; Meister, K.; Charypar, D.; Lefebvre, N.; Nagel, K. MATSim-T: Architecture and simulation times. In *Multi-Agent Systems for Traffic and Transportation Engineering*; IGI Global: Hershey, PA, USA, 2009; pp. 57–78.
15. Okal, B.; Arras, K.O. Towards group-level social activity recognition for mobile robots. In *IROS Assistance and Service Robotics in a Human Environments Workshop*; International Society of Biomechanics: Chicago, IL, USA, 2014.
16. Ferber, J.; Gutknecht, O.; Michel, F. From Agents to Organizations: An Organizational View of Multi-Agent Systems. In *Proceedings of the International Workshop on Agent-Oriented Software Engineering, Melbourne, Australia, 15 July 2003*; Lecture Notes in Computer Science; Giorgini, P., Miiler, J.P., Odell, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2004.
17. Morvan, G. Multi-level agent-based modeling-a literature survey. *arXiv* **2012**, arXiv:1205.0561.
18. Gerber, C.; Siekmam, J.; Vierke, G. *Holonic Multi-Agent Systems*; Springer: Berlin/Heidelberg, Germany, 1999.
19. Mass_Java_Core—BitBucket. Available online: https://bitbucket.org/mass_library_developers/mass_java_core (accessed on 26 February 2018).
20. Emau, J.; Chuang, T.; Fukuda, M. A Multi-Process Library for Multi-Agent and Spatial Simulation. In Proceedings of the 2011 Pacific Rim Conference on Communications and Computers and Signal Processing (PACRIM), Victoria, BC, Canada, 23–26 August 2011.
21. Mistry, B.; Fukuda, M. Dynamic load balancing in multi-agent spatial simulation. In Proceedings of the 2015 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), Victoria, BC, Canada, 24–26 August 2015.

22. Ma, Z.; Fukuda, M. A multi-agent spatial simulation library for parallelizing transport simulations. In Proceedings of the 2015 IEEE Winter Simulation Conference (WSC), Huntington Beach, CA, USA, 6–9 December 2015.
23. Bowzer, C.; Phan, B.; Cohen, K.; Fukuda, M. Collision-Free Agent Migration in Spatial Simulation. In Proceedings of the Communication Papers of the 2017 Federated Conference on Computer Science and Information Systems (FedCSIS 2017), Prague, Czech Republic, 3–6 September 2017.
24. Chuang, T.; Fukuda, M. A Parallel Multi-Agent Spatial Simulation Environment for Cluster Systems. In Proceedings of the 16th International Conference on Computational Science and Engineering, Sydney, NSW, Australia, 3–5 December 2013.
25. Pignotti, E.; Polhill, G.; Edwards, P. Using Provenance to Analyse Agent-based Simulations. In Proceedings of the EDBT/ICDT 2013 Joint Workshops, Genoa, Italy, 18–22 March 2013; ACM: New York, NY, USA, 2013; pp. 319–322. [CrossRef]
26. Epstein, J.M.; Axtell, R. *Growing Artificial Societies: Social Science from the Bottom Up*, 1st ed.; Brookings Institution Press: Washington, DC, USA, 1996; ISBN 978-0-262-55025-3.
27. Dunning, J.B.; Stewart, D.J.; Liu, J. Individual-Based Modeling. In *Learning Landscape Ecology: A Practical Guide to Concepts and Techniques*; Gergel, S.E., Turner, M.G., Eds.; Springer: New York, NY, USA, 2002; pp. 228–245, ISBN 978-0-387-21613-3.
28. Railsback, S.F. Concepts from complex adaptive systems as a framework for individual-based modelling. *Ecol. Model.* **2011**, *139*, 47–62. [CrossRef]
29. Batty, M. A Generic Framework for Computational Spatial Modeling. In *Agent-Based Models of Geographical Systems*; Springer, Dordrecht: Dordrecht, The Netherlands, 2012; pp. 19–50, ISBN 978-90-481-8927-4.
30. Gurcan, O.; Dikenelli, O.; Bernon, C. A generic testing framework for agent-based simulation models. *J. Simul.* **2013**, *7*, 183–201. [CrossRef]
31. Drummond, C. Replicability is not reproducibility: Nor is it good science. In Proceedings of the Evaluation Methods for Machine Learning Workshop, Montreal, QC, Canada, 14–18 June 2009.
32. Davison, A. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Comput. Sci. Eng.* **2012**, *14*, 48–56. [CrossRef]
33. Simonson, N.; Wessels, S.; Fukada, M. Language and Debugging Support for Multi-Agent and Spatial Simulation. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, NV, USA, 16–18 July 2012; pp. 373–379.
34. Li, H.A. A Debugger of Parallel Multi-Agent Spatial Simulation. Available online: https://depts.washington.edu/dslab/MASS/reports/HongbinLi_final_au14.ppt (accessed on 26 February 2018).
35. Interlandi, M.; Shah, K.; Tetali, S.D.; Gulzar, M.A.; Yoo, S.; Kim, M.; Millstein, T.; Condie, T. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* **2015**, *9*, 216–227. [CrossRef]
36. Malik, T.; Gehani, A.; Tariq, D.; Zaffar, F. Sketching Distributed Data Provenance. In *Data Provenance and Data Management in eScience*; Liu, Q., Bai, Q., Giugni, S., Williamson, D., Taylor, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 85–107, ISBN 978-3-642-29931-5.
37. Tariq, D.; Ali, M.; Gehani, A. Towards Automated Collection of Application-Level Data Provenance. In Proceedings of the Theory and Practice of Provenance (TaPP), Boston, MA, USA, 14–15 June 2012; USENIX Association: Berkeley, CA, USA, 2012.
38. Crawl, D.; Wang, J.; Altintas, I. Provenance for MapReduce-based Data-intensive Workflows. In Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science, Seattle, WA, USA, 12–18 November 2011; ACM: New York, NY, USA, 2011; pp. 21–30. [CrossRef]
39. Park, H.; Ikeda, R.; Widom, J. RAMP: A System for Capturing and Tracing Provenance in MapReduce Workflows. In Proceedings of the International Conference on Very Large Data Bases (VLDB), Seattle, WA, USA, 29 August–3 September 2011; VLDB: Seattle, WA, USA, 2011; pp. 1351–1354.
40. Wang, J.; Crawl, D.; Altintas, I. Kepler + Hadoop: A General Architecture Facilitating Data-intensive Applications in Scientific Workflow Systems. In Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science, Portland, LA, USA; ACM: New York, NY, USA, 2009; pp. 12:1–12:8. [CrossRef]
41. Akoush, S.; Sohan, R.; Hopper, A. HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In Proceedings of the 5th International Workshop on the Theory and Practice of Provenance (TaPP), Edinburgh, Scotland, 8–9 July 2015; USENIX Association: Lombard, IL, USA, April 2013.

42. Buyya, R. *High Performance Cluster Computing: Architectures and Systems Vol. 1*, 1st ed.; Prentice-Hall: Upper Saddle River, NJ, USA, 1999; ISBN 978-0-13-013785-2.
43. Buyya, R. *High Performance Cluster Computing: Architectures and Systems Vol. 2*, 1st ed.; Prentice-Hall: Upper Saddle River, NJ, USA, 1999; ISBN 978-0-13-013784-5.
44. Terrizzano, T.; Schwarz, P.; Roth, M.; Colino, J.E. Data Wrangling: The Challenging Journey from the Wild to the Lake. In Proceedings of the Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, 4–7 January 2015.
45. Stein, B.; Morrison, A. The Enterprise Data Lake: Better Integration and Deeper Analytics. *PwC Technol. Forecast Rethink. Integr.* **2014**, *1*, 1–9.
46. Suriarachchi, I.; Plale, B. Crossing analytic systems: A case for integrated provenance in data lakes. In Proceedings of the eScience, Baltimore, MD, USA, 23–27 October 2016; pp. 349–354. [CrossRef]
47. PROV-Overview. Available online: https://www.w3.org/TR/2013/NOTE-prov-overview-20130430/ (accessed on 26 February 2018).
48. PROV-O. Available online: https://www.w3.org/TR/prov-o/ (accessed on 26 February 2018).
49. RDF—Semantic Web Standards. Available online: https://www.w3.org/RDF/ (accessed on 26 February 2018).
50. Chen, P.; Evans, T.; Plale, B. Analysis of Memory Constrained Live Provenance. In Proceedings of the International Provenance and Annotation Workshop (IPAW), McLean, VA, USA, 7 June 2016; Springer: Cham, Switzerland, 2016; pp. 42–45. [CrossRef]
51. Provenance in Netlogo. Available online: https://sourceforge.net/projects/pin/ (accessed on 26 February 2018).
52. Gehani, A.; Tariq, D. SPADE: Support for Provenance Auditing in Distributed Environments. In Proceedings of the International Middleware Conference, Montreal, QC, Canada, 3–7 December 2012; Springer: New York, NY, USA, 2012.
53. Muniswamy-Reddy, K.; Braun, U.; Holland, D.A.; Macko, P.; Maclean, D.; Margo, D.; Seltzer, M.; Smogor, R. Layering in Provenance Systems. In Proceedings of the USENIX Annual Technical Conference, San Diego, CA, USA, 14 June 2009.
54. Lucia, B.; Ceze, L. Data provenance tracking for concurrent programs. In Proceedings of the International Symposium on Code Generation and Optimization (CGO), San Francisco, CA, USA, 7–11 February 2015; IEEE: Washington, DC, USA, 2015; pp. 146–156.
55. Thalheim, J.; Bhatotia, P.; Fetzer, C. INSPECTOR: Data Provenance Using Intel Processor Trace. In Proceedings of the International Conference on Distributed Computing Systems, Nara, Japan, 27–30 June 2016; pp. 25–34. [CrossRef]