

# Dependency Provenance in Agent Based Modeling

Preprint Version

Forthcoming in IEEE eScience 2013

Peng Chen

School of Informatics and Computing  
Indiana University  
chenpeng@cs.indiana.edu

Beth Plale

School of Informatics and Computing  
Indiana University  
plale@cs.indiana.edu

Tom Evans

Department of Geography  
Indiana University  
evans@indiana.edu

**Abstract**—Researchers who use agent-based models (ABM) to model social patterns often focus on the model’s aggregate phenomena. However, aggregation of individuals complicates the understanding of agent interactions and the uniqueness of individuals. We develop a method for tracing and capturing the provenance of individuals and their interactions in the NetLogo ABM, and from this create a “dependency provenance slice”, which combines a data slice and a program slice to yield insights into the cause-effect relations among system behaviors. To cope with the large volume of fine-grained provenance traces, we propose use-inspired filters to reduce the amount of provenance, and a provenance slicing technique called “non-preprocessing provenance slicing” that directly queries over provenance traces without recovering all provenance entities and dependencies beforehand. We evaluate performance and utility using a well known ecological NetLogo model called “wolf-sheep-predation”.

## I. INTRODUCTION

An agent-based model (ABM) is a computerized simulation of distributed decision-makers (agents) who interact through prescribed rules. ABM has been effective in studies such as complex adaptive spatial system (CASS) [6] and ecological modeling [4] because of its ability to represent heterogeneous individuals and the interactions among them. Researchers using agent-based models often focus on the emergent outcomes of aggregated behaviors, however, the fundamental philosophy in ABMs of methodological individualism warns that this may yield misleading results, and advocates a focus on the uniqueness of individuals and interactions [4]. In addition, the interactions among agents are inherently associated with cause-effect relations in the transition of system states through time, and these cause-effect relations are important and necessary for an understanding of complex process, but elucidating these relations poses a significant challenge to the research community because of the complex dynamics in ABM [6].

The utility of provenance information has been established in many scientific domains, of most relevance to this paper are computer science [13] and geographic information system [14]. Provenance, which is a type of metadata, is the lineage of a data product or process: it’s creator, contributing processes, interactions, and data sources. In the context of agent based modeling, provenance captures state changes in individual agents and interactions through time. These state changes reveal every unique individual behavior, but produces provenance of a volume that is computationally and analytically challenging for models with large numbers of agents. Cheney et al. [10][9] argues that dependency analysis techniques used in program slicing can be a formal foundation for provenance that is

intended to show how (part of) the output of a query depends on (parts of) its input. This so called *dependency provenance* is different from *where-provenance* and *data lineage*, but similar to *how-provenance* or *why-provenance* [7] in that it identifies a data slice showing the input data relevant to the output data. However, we argue that it is also important to consider the part of program execution that is relevant to the data dependency—program slice. We demonstrate that the combination of data slice and program slice—what we call provenance slice—can explain how and why the output data depends on the input data, and can yield insight into cause-effect relations among system behaviors.

NetLogo [16] is an agent-based modeling platform in wide use in research and education worldwide. NetLogo has its own Logo programming language, containing high level primitives for performing batch operations over a group of agents. Capturing the provenance of complex system behaviors in NetLogo requires collecting information about the execution of every statement, which can generate huge amounts of fine-grained provenance that poses a big challenge for storage and subsequent querying. Pignotti et al. [12] discuss typical queries over a provenance record in ABM, and find that as the number of simulation runs and agents in the simulation increases, these queries become exponentially complex.

In this paper we propose a dependency provenance model for ABM. The utility of the approach is verified by means of a tool we built for tracing, capturing, storing and exposing the dependency provenance from NetLogo. Dependency provenance itself provides deeper understanding of an ABM simulation through answers to questions such as “*How did the simulation evolve?*”, “*What changes occur after changing a parameter’s value?*”, “*How influential is a parameter?*”, and “*Which parameter is the most influential for a particular type of agent?*”. We propose several use-inspired filters that dramatically reduce the amount of persistent storage, and propose a technique that we call non-preprocessing (NP) provenance slicing that avoids recovering entities and dependencies from the provenance traces that are irrelevant to the query. We demonstrate the utility of the techniques using a classical ABM model for ecological studies called “wolf-sheep-predation” [15].

The remainder of the paper is organized as follows: Section II reviews related work. Section III describes the dependency provenance model in ABM and Section IV discusses the capture of dependency provenance in NetLogo. Section VI introduces the non-preprocessing provenance slicing technique

and Section V introduces some use-inspired filters. The experimental evaluation with NetLogo model “wolf-sheep-predation” is presented in Section VII. Section VIII demonstrates some additional use cases of dependency provenance in analyzing agent-based simulation. Section IX concludes the paper and discusses future work.

## II. RELATED WORK

Bennett [6] illustrates the importance of explicitly considering provenance in agent-based modeling through the development of a spatially explicit agent-based land use simulation framework. While their research is speculative, we implement the automated provenance tracing and capture for NetLogo and demonstrate some example provenance queries that can help understand and analyze the simulation.

Systems that gather fine-grained provenance metadata must process large amounts of information, and there is some existing research on provenance filtering. SPADE [11] is an open source software platform that supports collecting, filtering, storing, and querying provenance metadata. SPADE provides a framework for implementing filters (that can be stacked in arbitrary order). A filter receives a stream of provenance graph vertices and edges, and can rewrite their annotations (in which domain-specific semantics are embedded). However, our research studies on how to filter provenance traces generated by probes in the agent based model, and proposes several use-inspired filters that can keep provenance traces that are relevant to particular provenance queries.

Program slicing is a well-explored technique in software engineering. Intuitively, program slicing attempts to provide a concise explanation of a bug or anomalous program behavior, in the form of a fragment of the program that shows only those parts “relevant” to the bug or anomaly. Cheney [9] argues that there is a compelling analogy between program slicing and data provenance and defines the dependency provenance of an output as the set of all input fields on which it depends (a data slice). However, the data slice alone does not explain why there are dependences, thus we propose the provenance slice as a combination of data slice and its related program fragment (program slice).

Typical precise dynamic slicing needs to build dependence graph from the programs execution trace (preprocessing) before slicing. However, the dependence graph can be extremely large and run out of memory, so is the provenance graph if the query needs to recover all dependencies beforehand. Zhang et al. [17] present the design and evaluation of three precise dynamic slicing algorithms called the full preprocessing (FP), no preprocessing (NP) and limited preprocessing (LP) algorithms. These algorithms differ in the relative timing of constructing the dynamic data dependence graph and its traversal for computing requested dynamic slices. The no preprocessing (NP) algorithm does not perform any preprocessing but rather during slicing it uses demand driven analysis that recovers dynamic dependencies and caches the recovered dependencies for potential future reuse, and we derive the non-preprocessing (NP) provenance slicing technique from this no preprocessing (NP) algorithm.

Pignotti et al. [12] investigate the role of provenance in agent-based simulation and discuss typical queries over a

provenance record. They describe three types of provenance that can be recorded from a simulation: the provenance about model development, the provenance about executing the model and the provenance about simulation. The dependency provenance we that explore is close to the type of provenance about simulation in their classification. However, they capture the provenance of simulations as actions performed by agents in discrete-event modeling, while we trace the dependency provenance by source code instrumentation; they concluded that as the number of simulation runs and agents in the simulation increases, the queries become exponentially complex, while we address this scalability problem by first using the use-inspired filters to drop off irrelevant provenance traces and then applying the non-preprocessing (NP) provenance slicing technique to avoid recovering irrelevant provenance.

## III. DEPENDENCY PROVENANCE IN ABM

Dependency provenance is the information relating each part of the output of a query to a set of parts of the input on which the output depends. Dependency provenance can be used to compute data slices, or summaries of the parts of the input relevant to a given part of the output [10]. We want to define dependency provenance in ABM similar to this, but we also want to compute the program slice (relevant processes) as a complement to the data slice. While the data slice tells what the relevant input data is, the program slice tells how the output data depends on the input data. Specifically, the dependency provenance in ABM that we propose contains the information of:

- All data products and their dependencies;
- Procedures associated with these dependencies.

We choose W3C PROV [5] to record the dependency provenance in ABM (specifically from the NetLogo model), because PROV allows us to express the provenance of agents and the evolution of a variable. Unlike its predecessor OPM, an agent in PROV is also a particular type of entity and the PROV has the concept of versions. The mapping of ABM provenance to PROV is accomplished by first identifying the entities and dependencies in a NetLogo program, then mapping concepts in PROV to them (see Table I).

Note that in our mapping, the agent in ABM is mapped to an agent in PROV, this is because an agent in ABM has attributes and carries out actions that match the definition of agent in PROV; we define the state of a variable as an entity to capture its evolution over time; since the procedure in NetLogo can access global variables and any other agent without explicitly accepting them as input parameters, we need to capture the accurate dependencies at the statement level of the NetLogo program, however, we do not model the execution of a statement as an activity, as it can produce overwhelming amount of activities. Instead, we decide that the procedure level information is a good abstraction to explain why the output data depends on some input data.

## IV. PROVENANCE TRACING

Provenance of a NetLogo model is captured through the process of adding probes to the model source code. These

TABLE I. MAPPINGS FROM PROV ONTOLOGY TO NETLOGO CONCEPTS

Concept in PROV	Concept in NetLogo	Code example
Agent	Agent	breed [wolves wolf] ;; <i>wolf is an agent</i>
Activity	Execution of a procedure	ask sheep [ move ;; <i>an activity</i> ... ]
Entity	State of a global/agent/local variable	set color white ;; <i>the current state of color (an agent variable) is an entity</i>
Relationship: used	1) Procedure reads the current value of a variable 2) Procedure depends on the current value of a variable	1) to reproduce-sheep ... set energy (energy / 2) ;; <i>the activity "reproduce-sheep" used the entity "energy"</i> ... end  2) if grass? [ ... eat-grass ;; <i>the activity "eat-grass" used the entity "grass?"</i> ]
Relationship: wasGeneratedBy	Procedure writes the value of a variable	to reproduce-sheep ... set energy (energy / 2) ;; <i>the entity "energy" was generated by activity "reproduce-sheep"</i> ... end
Relationship: wasDerivedFrom	1) A statement reads var2 before writing var1 2) A statement writing var1 depends on var2	1) set energy random (2 * wolf-gain-from-food) ;; <i>the entity "energy" was derived from entity "wolf-gain-from-food"</i>  2) if grass? [ set energy energy / 1 ;; <i>the entity "energy" was derived from entity "grass?"</i> ]
Relationship: wasRevisionOf	If an variable was derived from itself	to reproduce-sheep ... set energy (energy / 2) ;; <i>the entity "energy" was a revision of itself</i> ... end
Relationship: wasInformedBy	A procedure is invoked inside another procedure	to go ... ask sheep [ move ;; <i>activity "move" was informed by activity "go"</i> ... end
Relationship: wasAssociatedWith	A procedure is invoked by an agent	ask sheep [ move ;; <i>the activity "move" was associated with a sheep agent</i> ]
Relationship: wasAttributedTo	Variable belongs to an agent	sheep-own [energy] ;; <i>the entity "energy" was attributed to a sheep agent</i>
Relationship: alternateOf	A variable is a reference to an agent	let prey one-of sheep-here ;; <i>the entity "prey" is an alternate of a sheep agent (an agent is also an entity in PROV)</i>

probes generate provenance traces. We propose three types of probes for this purpose, probes that log:

- Procedure invocations (Type 1),
- Write and read operations (Type 2), and
- Conditional statements (Type 3).

Type 1 probes generate information used to compute the program slice. Type 2 and Type 3 probes produce provenance about data dependencies that is used to compute the data slice.

The open source tool that we built to implement our abstractions is called PIN (acronym for "Provenance in NetLogo") [2]. It can trace, capture, query and visualize the dependency provenance in NetLogo. It consists of four main

TABLE II. CODE SNIPPET BEFORE AND AFTER INSTRUMENTATION

Code snippet before instrumentation	After instrumentation
<pre>if grass? [   ask patches [     set countdown random     grass-regrowth-time     set pcolor one-of [green   brown]   ] ]</pre>	<pre>if grass? [   provenance:write (word "dependsOn global grass?   "grass? )   ask patches [     provenance:write "recordStart"     provenance:write (word "read global grass-   regrowth-time "grass-regrowth-time )     set countdown random grass-regrowth-time     provenance:write (word "write agent count-   down "countdown )     provenance:write "recordEnd"     provenance:write "recordStart"     set pcolor one-of [green brown]     provenance:write (word "write agent pcolor   "pcolor )     provenance:write "recordEnd"   ]   provenance:write "End dependsOn" ]</pre>

components: a source code analyzer used to automatically add probes to the model's source code, a NetLogo extension for capturing the provenance traces generated from probes, a non-preprocessing (NP) provenance slicing technique for computing provenance slices using provenance traces, and a visualization component for visualizing the provenance slices. Figure 1 shows how the NetLogo provenance tool works. The tool is compatible with NetLogo version 5.0.3.

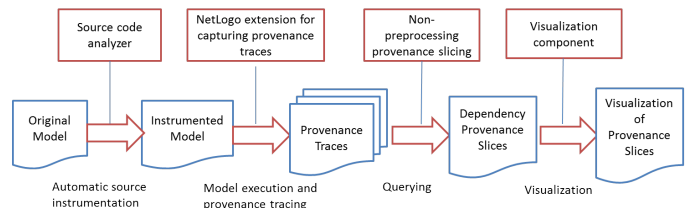


Fig. 1. PIN overview: Red rectangles represent major components, and blue document charts represent input and output of each component

Table II gives a sample instrumented code snippet. Note that the primitive "provenance:write" is implemented in PIN's NetLogo extension to write an input string to the log file and each probe passes a provenance trace as a string into it; for each statement in the original source code, we add different probes for its reading/writing operations on different variables and use two extra probes to enclose all of them.

To understand the scale of provenance traces generated and the overhead introduced by our method, we perform an empirical study using a classic NetLogo model for ecologists called "wolf-sheep-predation". The model, designed by Uri Wilensky [15], explores the stability of predator-prey ecosystems. A system is stable if it trends to maintaining itself over time, despite fluctuations in population sizes. A system is similarly unstable if it trends to extinction of one or more species involved. The "wolf-sheep-predator" model has two variations. In the first variation, wolves and sheep wander the landscape randomly while wolves are busy looking for sheep to prey on. Each model step is an action by a wolf that costs them energy units, and they must eat sheep in order to replenish their energy. When a wolf runs out of energy, it dies. Each of wolf and sheep has a fixed probability of reproducing at each time step. This variation produces interesting population dynamics,

TABLE III. SIZE OF PROVENANCE LOG AFTER CERTAIN NUMBER OF ITERATIONS

	10	50	100	200	300	400
Number of sheep	129	338	170	38	2,413	112,402
Number of wolves	63	82	349	1	1	116
Log size	523KB	3.63MB	12.1MB	20.4MB	35.3MB	773MB
Number of traces	20,793	144,590	475,061	799,677	1,351,128	27,860,316

but is ultimately unstable. The second variation includes grass in addition to wolves and sheep. The behavior of the wolves is identical to the first variation, however this time the sheep must eat grass in order to maintain their energy—when they run out of energy they die. Once grass is eaten it will only regrow after a fixed amount of time. This variation is more complex than the first, but it is generally stable. We choose the first variation to study the scale of provenance under conditions where the model is unstable, and run it up to 400 iterations.

In Table III, we show the size of the provenance log and the agent population after 10, 50, etc. iterations. The size of provenance captured actually depends on both the number of iterations and the size of agent population. The provenance size increases dramatically at 400 iterations, which is because the wolves almost die out and the number of sheep has increased exponentially at 400 iterations. This means that, for simulations that have a large population of agents or need to run a long period of time, the size of provenance traces captured using our method can be overwhelming. To cope with this, we propose some use-inspired filters in Section V and evaluate their performance in Section VII.

To understand the time overhead introduced by provenance tracing and capture, we identify four performance metrics that are measured through timing information gathered during the “Model execution and provenance tracing” step (in Figure 1). These metrics are:

- *Execution time*: model execution;
- *Message passing time*: the probes pass the trace messages to the NetLogo extension by calling the primitive “provenance:write”;
- *Collecting time*: the NetLogo extension collects trace messages and their context information from the model;
- *Writing time*: the NetLogo extension writes traces into a provenance log file.

Execution time is calculated by running the model without instrumentation, and the others are computed indirectly by running simulations with different versions of the NetLogo extension and calculating the average time differences. The model is running in NetLogo 5.0.3 on a single machine with Win8 64bit OS, 8GB memory, and Core i5 2.53GHz dual core CPU. Figure 2 summarizes the results.

From Figure 2 we see (a) that the “message passing time” is small compared to the “execution time” and “collecting time”, which means that we can simply turn off tracing once the capture finishes by not processing the trace messages, and this strategy can dramatically reduce the overhead. We

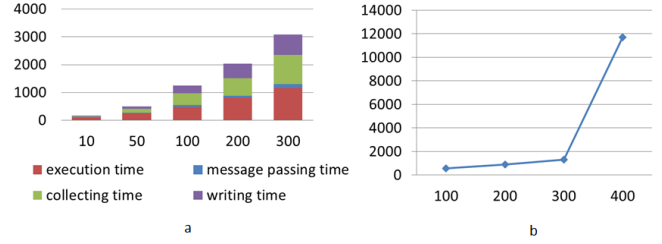


Fig. 2. Provenance tracing and capture: X-axis is number of iterations run; Y-axis has time cost (ms). (a) shows where time costs lie for up to 300 iterations. (b) shows the total time for 100, 200, etc. iterations.

can also use filters that perform aggregation and filtering to reduce the “writing time”, but these filters do not reduce the “collecting time”, and even introduce additional computation time. Figure 2(b) shows that the total time (including the overhead) scales linearly until 300 iterations and then increases dramatically when the population of sheep starts to increase exponentially.

## V. FILTERING ABM PROVENANCE

To reduce the overhead and simplify subsequent querying, we propose filters that apply intelligence to reduce provenance before it is written to log files. Some of the filters abstract statement level traces and others drop unrelated traces and turn off the tracing once the capture finishes. In the remainder of this section, we discuss two types of provenance filters and their example outputs.

### A. Aggregation

Keeping track of fine-grained provenance provides a detailed view of the dependencies among entities in a simulation, but at the expense of additional storage and processing overhead. We can extract the fine-grained provenance from provenance traces and aggregate them into high level provenance records.

For example, one of the interactions among sheep agents and wolf agents in the model “wolf-sheep-predator” is that a wolf agent acquires reference to a sheep agent and then kills it. We can aggregate these two steps into a single activity “kill”, or to a generic relationship “interact with”. We propose an aggregation filter that uses the generic relationship “interact with” to represent all the complex interactions among agents, since identifying less generic activities like “kill” is more complicated and may need the user effort. This aggregation filter is inspired by the interest of domain scientist in studying the interactions among agents in simulation, and we can visualize its output as a social network (see Figure 3 for an example). We implement this filter by buffering all provenance traces within an iteration at runtime and then extracting and writing the abstract provenance to the log file.

### B. Filtering

Inspired by the forward and backward data slices discussed in [5], we propose filtering that computes a forward provenance slice or backward provenance slice for a given variable. Our PIN captures provenance traces of all iterations by default,

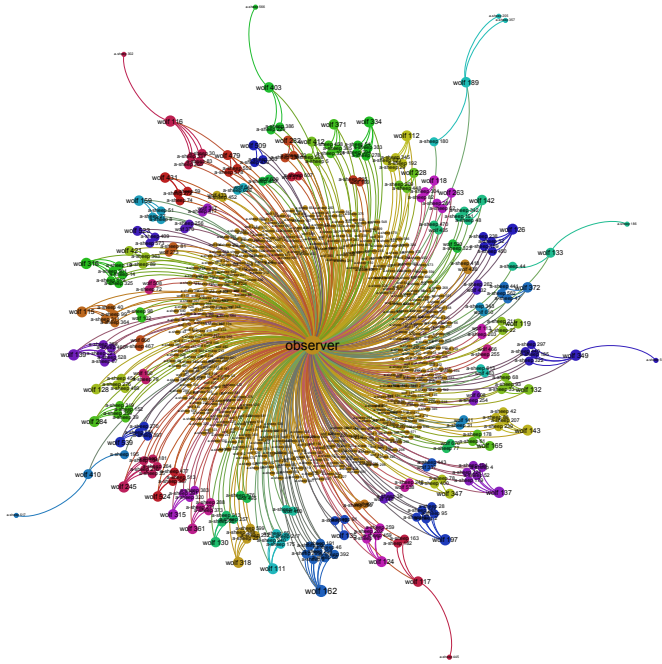


Fig. 3. An example visualization of interactions among agents in “wolf-sheep-predation”. Edges represent interactions among vertices (agents). Edge and vertex are partitioned into strongly (or weakly) connected communities and are dyed accordingly. The biggest community in the center has the observer (a manager agent) and the agents that have no other interactions. Each of the smaller communities far from the center has one wolf agent and its preys

but we developed one filter that collects the provenance traces from only a single iteration. In this section, we introduce and describe the use cases and implementations of the forward, backward and single-iteration filters.

The backward provenance slice includes the processes, input data, intermediate data and agents that are involved in the generation of an output data. This can help user to better understand why a particular result is achieved and can be used for debugging. Figure 4 is the visualization of an example backward provenance from the model “wolf-sheep-predation”. That backward provenance slice explains how the final value of the variable “energy” of agent “wolf 130” is generated: there are three types of processes involved in the generation—“go”, “catch-sheep” and “wolf-reproduce”; the agent variable “energy” is reduced by 1 each time the “observer” agent (global manager) invokes the “go” procedure (which means one iteration starts); the agent variable “energy” increases by the value of “wolf-gain-from-food” after catching sheep agent “sheep 26”; the “energy” is reduced from 46 to 23 after the process “wolf -reproduce” took place.

It is usually difficult to decide the backward provenance slice for a variable before the simulation terminates, this is because the target variable can be directly or indirectly affected by any current data/process in the future. However, by observing the backward provenance we can get from the provenance traces of a finished simulation, we find that it is more like a linear structure rather than an upside down tree—it consists of information about how a given variable evolves as processes running on the same agent to which the variable belongs. So we implement an imprecise filter that drops all

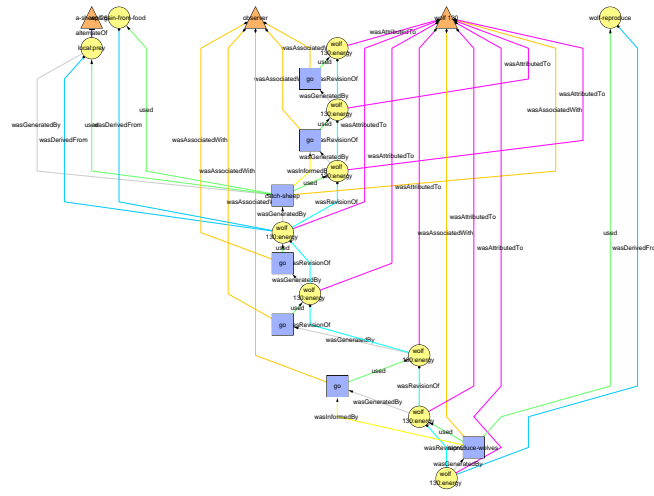


Fig. 4. The backward provenance of data product “energy” of agent “wolf 130”

provenance traces that are not associated with the agent to which the target variable belongs.

The forward provenance slice of one input data tells information about the future data products that are derived from the input data. It can be used to understand the impact of a input parameter, or to control the error propagation if an input data is corrupted. Figure 5 is the visualization of an example forward provenance of a parameter named “wolf-reproduce”. Note that each “was derived from” relationship means that the source entity was used in the generation of the target entity, and we do not consider any other indirect dependency in this way. We implement the forward provenance filter by keeping track of all data products that are derived from the input data or the previously derived data.

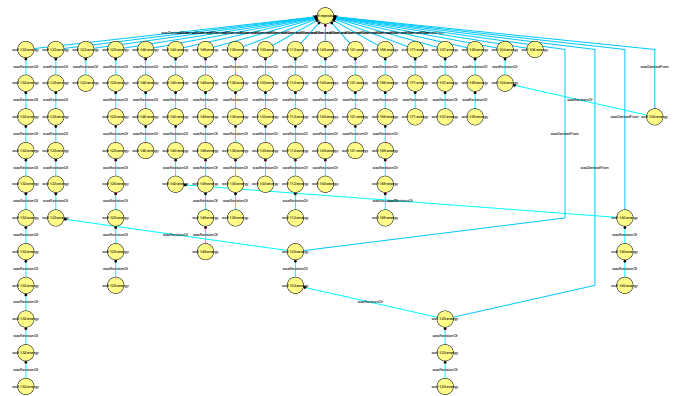


Fig. 5. The forward provenance of global variable “wolf-reproduce”

The last filter we propose keeps the execution traces only for a single iteration, which is inspired by the user’s interest in studying the various agent behaviors and the interactions among them. By recovering and visualizing the provenance from provenance traces of a single iteration, we can discover the different agent behavior patterns. Figure 6 is an example visualization that shows six different agent behaviors within an iteration: 1) the global agent “observer” invokes “display-

labels” on each wolf and sheep agent; 2) the wolf agents invoke the procedure “go”, “move” and “death”; 3) the wolf agents invoke the procedure “wolf-reproduce” in addition to the procedure “go”, “move” and “death” ; 4) the sheep agents invoke the procedure “go”, “move” and “death” ; 5) the sheep agents invoke the procedure “sheep-reproduce” in addition to the procedure “go”, “move” and “death”; 6) the wolf agents catch sheep agents via the procedure “catch-sheep”.



Fig. 6. An example single-iteration provenance

The implementation of the single-iteration provenance filter is simple. We ask the user to start the filtering by entering the name of the entry procedure for the next iteration (like the procedure “go” in the model “wolf-sheep-predator”). After the simulation exits the entry procedure, the filter shuts down the tracing to minimize the overhead.

In sum, we have proposed two types of filters and included visualizations of the provenance computed from the output provenance traces of these filters. We examine their performance in Section VII .

## VI. NON-PREPROCESSING PROVENANCE SLICING

The existing method that captures provenance from program logs needs to recover all the entities and dependencies from the logs before being able to answer queries [3]. However, our empirical study indicates that this can be infeasible for large provenance graph. For example, the log file that has the 20,793 provenance traces from a 10-iteration simulation of the model “wolf-sheep-predation” is only 523KB. However, recovering the full provenance (that has 8,855 nodes and 18,330 edges) from this log file and store it into the Neo4J [1] graph database takes 1,035,845ms (about 17 mins) in a machine with 8GB memory and Core i5 2.53GHz dual core CPU.

However, we can actually avoid recovering and maintaining provenance entities and dependencies that are unnecessary for answering a specific query. To achieve this, we propose a query technique we call non-preprocessing (NP) provenance slicing that is derived from Zhang’s no preprocessing (NP) program slicing algorithm [10]. The non-preprocessing (NP) provenance slicing technique employs demand driven analysis of the provenance traces to recover dependency provenance. When a provenance query begins we traverse the provenance traces forward (or backward) to recover the dynamic dependencies required for the provenance slice computation. For example, if we need the provenance slice for the final value of some variable  $v$  (backward provenance), we traverse the execution traces backward till the last access of the variable was found. If that value of  $v$  depends on another variable  $w$  (see possible dependencies in Table 1), we resume the traversal to also calculate the provenance slice of  $w$ .

In essence this algorithm performs partial preprocessing for extracting entities, activities and dependencies relevant to a provenance query. It is possible that two different querying requests involve common information. In such a situation, the non-preprocessing (NP) provenance slicing algorithm will recover the common information from the execution trace during both provenance slice computations. To avoid this repetitive work we can cache the recovered entities, activities and dependencies. Therefore at any given point in time, all entities, activities and dependencies that have been computed so far can be found in the cache. Similar to Zhang’s definition, we also define two versions of this demand driven algorithm, that is, without caching and with caching, as non-preprocessing without caching (NPwoC) provenance slicing and non-preprocessing with caching (NPwC) provenance slicing. We evaluate the performance of NPwoC provenance slicing in Section VII and find that it can be even faster than traversing the pre-recovered full provenance graph.

## VII. PERFORMANCE EVALUATION

In this section, we first evaluate the proposed filters by comparing the performance of provenance capture with these filters with the performance of a full provenance capture without any filter.

Table IV compares the size of provenance logs generated using different filters with the size of a full provenance log. We can see that all filters can dramatically reduce the storage cost. While the provenance log generated from single-iteration filter remains the same size (since the capture finishes), the logs generated from other provenance filters get larger as simulation goes on: the log size from aggregation filter keeps increasing since there are more and more agents; the log size from backward provenance filter keeps increasing until the target agent dies; the log size from forward provenance filter does not increase much after 200 iterations since we were tracing the usage of the global variable “wolf-reproduce” by wolf agents and there are few wolf agents after 200 iterations.

TABLE IV. EVALUATION OF STORAGE COST

Number of Iterations	10	50	100	200	300	400
full provenance capture	523KB	3.63MB	12.1MB	20.4MB	35.3MB	773MB
aggregation filter	8.22KB	30.9KB	100KB	140KB	230KB	4.53MB
backward provenance filter	5.36KB	18.1KB	18.1KB	18.1KB	18.1KB	18.1KB
forward provenance filter	13.9KB	163KB	925KB	2.08MB	2.10MB	2.75MB
single-iteration filter	46.8KB	45.2KB	45.2KB	45.2KB	45.2KB	45.2KB

Figure 7 shows the average time of provenance capture with different filters and without any filter (the full provenance capture), which proves our result of analysis in Section IV. It shows that while the single-iteration filter can significantly reduce the time overhead, the backward provenance filter doesn’t reduce the time overhead much, and the forward provenance filter and aggregation filter have even higher overhead than the full provenance capture. However, we argue that this can be

optimized by detaching the filtering from the tracing and the model execution using a message bus. In this way, the “writing time” and the additional computation time are replaced with less communication time.

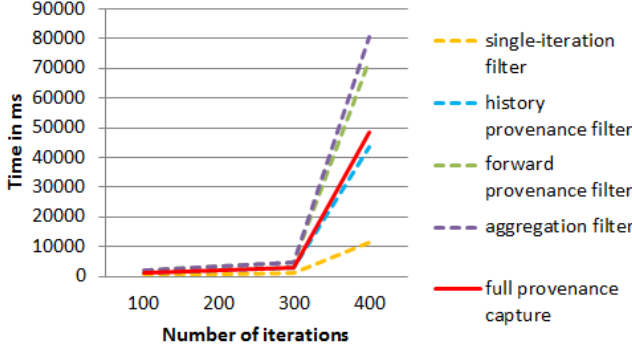


Fig. 7. Evaluation of capture time cost

We also evaluate the performance of proposed non-preprocessing (NP) provenance slicing technique (the NPwoC version) with various query requests (Table V). The non-preprocessing (NP) provenance slicing technique can query either on the full provenance traces or on the filtered provenance traces, and we measure both performance to demonstrate that the filtered provenance traces of reduced size can further improve the querying performance. Finally, we compare the performance of the non-preprocessing (NP) provenance slicing technique with the performance of traversing pre-recovered full provenance in a Neo4j graph database (using Neo4j Java API).

TABLE V. EVALUATION OF QUERY TIME COST

	Same query issued to the full provenance in Neo4j (ms) <sup>a</sup>	NP Provenance Slicing on full provenance traces (ms)	NP Provenance Slicing on filtered provenance traces (ms)
Aggregation filter	6,046	2,565	2,025
Backward provenance filter	3,043	1,964	1,823
Forward provenance filter	3,127	2093	1,785
Single-iteration filter	6,059	3,819	3,750

<sup>a</sup>Note that the construction of the full provenance database costs 1,035,845ms (about 17 min) beforehand

In sum, our proposed filters can dramatically reduce the storage cost, and though some of them can introduce extra time overhead, the single-iteration filter can reduce the time overhead by turning off the provenance tracing at runtime. In addition, we also find that using proposed filters and the non-preprocessing (NP) provenance slicing technique together can significantly improve the performance of subsequent querying.

## VIII. USING PROVENANCE TO ANALYZE SIMULATIONS

In Section V, we illustrated several use-inspired filters. To recap, we have demonstrated that:

- 1) Agent interactions that are abstracted from fine-grained provenance traces can provide an overview

of how agents are interacting and their community structure;

- 2) Backward provenance slice can answer the question of why/how a particular data was achieved and the history behavior of that agent;
- 3) Forward provenance slice shows future data items that depend on given data for controlling error propagation;
- 4) Single-iteration provenance slice helps understand what happened during an iteration and different patterns of agent behaviors.

We continue by demonstrating more advanced uses of dependency provenance. To understand simulation progress, we take snapshots of the agent interactions after certain time intervals and visualize them. In Figure 8, we visualize the agent-interaction network of model “wolf-sheep-predator” at different moments, and it shows that there are more wolf agents catching sheep agents after every 5 iterations.

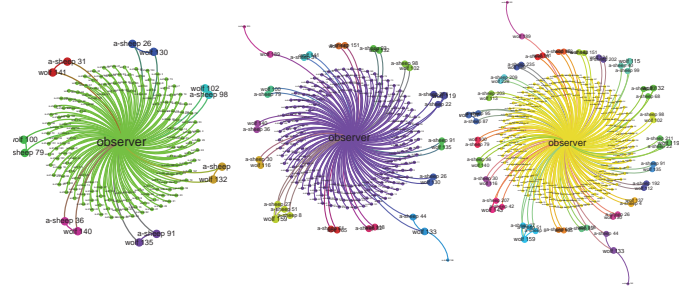


Fig. 8. Evolution of agent-interaction network. Left: after 5 iterations; middle: after 10 iterations; right: after 15 iterations

To study the sensitivity of the model to parameter calibration, agent based modelers often run the simulation with different values of that parameter and compare the outputs—so called parameter sweeps. However, this high level of comparison can only reveal aggregated differences in the final results. Forward provenance is very useful for this purpose since it can tell the exact impact of a given parameter by showing all intermediate data items influenced by that parameter. We can compare the forward provenance slices of that parameter to tell the exact differences caused by the different values. For example, the default value of the parameter “wolf-reproduce” in the model “wolf-sheep-predation” is 5%, which means that each time a wolf has a probability of 5% to reproduce itself. We first run a 10-iteration simulation with the default value of the parameter “wolf-reproduce” and compute its forward provenance slice, and then we change that value to 10% and re-run it and compute the same forward provenance slice. Figure 9 shows the comparison of the two forward provenance slices using a provenance graph matching algorithm [8]. The algorithm matches sub-graphs based on node/edge attributes and graph topology. It assigns one color to matching nodes and a different color to unmatched nodes.

The comparison shows that there are more wolves reproduced after changing “wolf-sheep-predation” from 5% to 10%, and while a few wolves that were reproduced in the former simulation remains the same in the latter simulation, other wolves that were reproduced in the former simulation either act differently or are not reproduced in the latter simulation. In

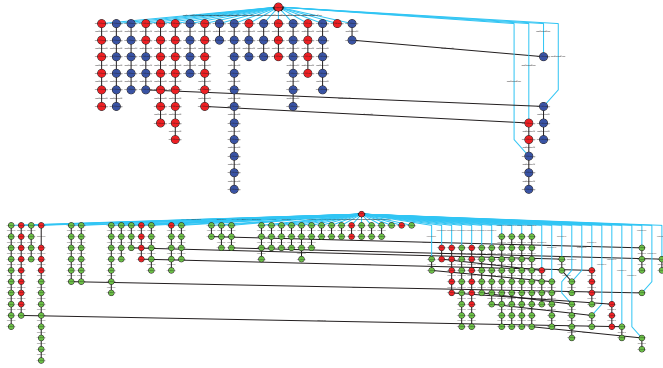


Fig. 9. Comparing forward provenance graph from a 10-iteration-execution with a 5 “wolf-reproduce” value (top) and another 10-iteration-execution with a 10 “wolf-reproduce” value (bottom). The red nodes are matched nodes according to the provenance graph matching algorithm, and others are unmatched nodes

sum, the parameter “wolf-sheep-predation” can influence the number of wolves reproduced and the way they behave.

Besides studying the model’s sensitivity to a parameter, we can also use forward provenance slice to compute the impact of that parameter on the model. Thus if the model has multiple parameters, the researcher can tell how influential each parameter is and which parameter is the most influential.

**Definition** The impact of a parameter  $v$  on an agent group  $S$  is defined as the ratio of the agents in  $S$  that are influenced by that parameter to the entire population in  $S$ :

$$Impact(v) = \frac{Cardinality(D)}{Cardinality(S)}$$

in which  $v$  is a global variable and  $D, S$  are agent groups.  $D = \{a | a \in S \text{ and } a \text{ has at least one data item in the forward provenance slice of } v\}$ , which is the subset of agents in  $S$  that are influenced by  $v$ .

Table VI shows the impact of four different global variables on the wolf agents and the sheep agents in a 10-iteration simulation of the model “wolf-sheep-predation”. The result shows that the global variable “wolf-reproduce” is the most influential for the wolf agents and the “sheep-reproduce” is the most influential for sheep agents. The result also shows that the global variable “sheep-gain-from-food” has no impact on the model, this is because we are running the less stable variation of that model, the one that does not include grass.

TABLE VI. IMPACT OF GLOBAL VARIABLES ON AGENT GROUPS

	Wolf-reproduce	Wolf-gain-from-food	Sheep-reproduce	Sheep-gain-from-food
wolf agents	25%	20.8%	0%*	0%
sheep agents	0%*	0%*	30.8%	0%

\*Recall that forward provenance slice is computed based on direct dependencies

## IX. CONCLUSION AND FUTURE WORK

In this paper, we introduce a methodology for automatic tracing and capture of provenance from agent based models. Since provenance can grow extremely large as the model progresses, we propose filters to reduce it. Inspired by several

typical provenance queries and user interests, we propose aggregation and filtering filters that help understand the evolution of simulation, the model’s sensitivity to a parameter, and the impact of a parameter on the model. We also propose the non-preprocessing (NP) provenance slicing techniques that can directly query over provenance traces to avoid recovering and maintaining provenance entities and dependencies that are irrelevant to the query request. The experimental evaluation shows that the proposed filters and non-preprocessing (NP) provenance slicing technique can work together to dramatically reduce the demands on persistent storage and simplify the subsequent querying.

Provenance tracing using source instrumentation has the limitation that we cannot capture provenance within the system calls or library functions. So one possible future work is to create a patch to the NetLogo platform to trace the provenance in library functions. We illustrated that using filters can add extra computation time to the simulation, which can be resolved in the future by detaching them from the NetLogo extension using a message bus. Besides, the model “wolf-sheep-predation” is a very simple agent based model, so our next step is to apply our tool and the same analysis onto a more complicated model.

## REFERENCES

- [1] (2013) Neo4j. [Online]. Available: <http://www.neo4j.org/>
- [2] (2013) PIN, Provenance in NetLogo. [Online]. Available: <https://sourceforge.net/projects/pin/>
- [3] M. S. Aktas, B. Plale, D. Leake, and N. K. Mukhi, “Unmanaged workflows: Their provenance and use,” in *Data Provenance and Data Management in eScience*. Springer, 2013, pp. 59–81.
- [4] L. An, “Modeling human decisions in coupled human and natural systems: Review of agent-based models,” *Ecological Modelling*, vol. 229, pp. 25–36, 2012.
- [5] K. Belhajjame, R. B’Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gil, P. Groth, G. Klyne, T. Lebo, J. McCusker, S. Miles, J. Myers, S. Sahoo, and C. Tilmes, “Prov-dm: The prov data model,” *W3C Candidate Recommendation 11 December 2012*, 2012. [Online]. Available: <http://www.w3.org/TR/2012/CR-prov-dm-20121211/>
- [6] D. A. Bennett, W. Tang, and S. Wang, “Toward an understanding of provenance in complex land use dynamics,” *Journal of Land Use Science*, vol. 6, no. 2-3, pp. 211–230, 2011.
- [7] P. Buneman, S. Khanna, and T. Wang-Chiew, “Why and where: A characterization of data provenance,” in *Database Theory—ICDT 2001*. Springer, 2001, pp. 316–330.
- [8] P. Chen, B. Plale, Y.-W. Cheah, D. Ghoshal, S. Jensen, and Y. Luo, “Visualization of network data provenance,” in *Workshop on Massive Data Analytics on Scalable Systems (DataMASS), co-located with High Performance Computing Conference, Pune India, Dec 2012*.
- [9] J. Cheney, “Program slicing and data provenance,” *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 22–28, 2007.
- [10] J. Cheney, A. Ahmed, and U. A. Acar, “Provenance as dependency analysis,” in *Database Programming Languages*. Springer, 2007, pp. 138–152.
- [11] A. Gehani and D. Tariq, “Spade: Support for provenance auditing in distributed environments,” in *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 2012, pp. 101–120.
- [12] E. Pignotti, G. Polhill, and P. Edwards, “Using provenance to analyse agent-based simulations,” in *Proceedings of the Joint EDBT/ICDT 2013 Workshops*. ACM, 2013, pp. 319–322.
- [13] Y. L. Simmhan, B. Plale, and D. Gannon, “A survey of data provenance in e-science,” *ACM Sigmod Record*, vol. 34, no. 3, pp. 31–36, 2005.

- [14] S. Wang, A. Padmanabhan, J. D. Myers, W. Tang, and Y. Liu, "Towards provenance-aware geographic information systems," in *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems*. ACM, 2008, p. 70.
- [15] U. Wilensky. (1997) Wolf sheep predation model. [Online]. Available: <http://ccl.northwestern.edu/netlogo/>
- [16] —. (1999) Netlogo. [Online]. Available: <http://ccl.northwestern.edu/netlogo/>
- [17] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 319–329.