Towards automated provenance collection for experimental runs of agent-based models[⋆]

Doug Salt[1][0000−1111−2222−3333] and Gary Polhill[2,3][1111−2222−3333−4444]

The James Hutton Institute, Craigiebuckler Aberdeen AB15 8QH Scotland
doug.salt@hutton.ac.uk
http://hutton.ac.uk

Abstract. We demonstrate a working framework for the automatic recording of provenance and metadata for primarily agent-based models that could easily be adapted to the other modelling environments. We discuss the need for such a framework, the philosophy behind the design we adopted, the implementation, discuss the results and demonstrate a simple tool for for tracing bad data through a provenance graph.

Keywords: Provenance · metadata · modelling · automation · replication.

1 Introduction

Replication of social simulation results has been highlighted as a significant issue for the ABM for a number of years (e.g. [6]), and in particular the paper that forms the basis of this work [18]. The focus of replication work has previously just been on the model itself, but as was shown in ibid, the analysis of the outputs of the model can potentially be just as complex, and no less difficult to replicate unless adequate records are kept. Indeed there as an additional attempt to reproduce the results and even with the lessons learnt from ibid, it still took months to recreate the final diagrams that were submitted to paper. The TRACE protocol [24, 1] provides some guidance highlighting the need to keep a notebook of the analysis done and a standardised approach to making that notebook. There are also appear to be many scientific workflow tools, examples being, Snakemake [12], NextFlow [5], Kepler [14], Taverna [10] just to name a few. However these are workflow tools, the concentrate on automation and repeatability, and if they do record provenance, then it is an after thought. It appears that these tools are more centred on automation rather than provenance and metadata.

One of the lessons learned from the replication exercise in [18] show that, for the purposes of replicability more detailed guidance on the information that should be recorded, and on tools that could be used to support the process. Ideally the aim should be to completely automate the process, record accurately

⋆ Supported by The Scottish Government

sources of data used, version check applications used essentially providing a complete graph from data to result.

The output analysis replication in this paper concerns earlier work with FEARLUS-SPOMM, which is a coupled agent-based model of agricultural decision-making and species stochastic patch occupancy metacommunity model that has been used to explore incentivisation strategies to improve biodiversity [20, 9]. Belonging to the 'typification' class of social simulations [3]. This work involved the analysis of the outputs of approximately 20,000 runs of the model using a number of techniques aimed at demonstrating non-linearities in the relationship between incentivisation and biodiversity outcome. Recording workflow data on the process used to create analysis can be challenging, and currently there are no codified standards as to how this should be done for agent-based models. For FEARLUS-SPOMM, the methods used drew heavily on statistical techniques available as R packages that are as part of core R functionality. Although R allows transcripts of interactive terminal sessions to be saved, the work involved great deal of exploration of different ways of attempting to visualise and analyse the non-linearities in the model results, not all of which were likely to be reported in the paper. Such logs are therefore not the best way to record the means by which the outputs were analysed, and hence the strategy used was to save each analysis or visualisation in a(n R) script. Since the output from the (Swarm) software that generated the output data being analysed used a mixture of text formats, some Perl scripts were also written to process that output into a CSV file for easy import into R. When the MIRACLE project [17] provided a context in which the replication of that analysis was necessary, an opportunity was created to test the viability of the above strategy.

If we make the assumptions that the input data is accurate and relevant; the model bug-free and rigorously correct then the manifest problem with agent-based modelling is the number of replications required to provide adequate probability that an interesting result is not the null hypothesis, and just down to a matter of chance. Realistically this involves many, many repetitions of the same experiment or small variations on the parameters for a given experiment. Similarly infeasible is the manual running of these many experiments and thus most resort to some kind of automation. We refer to this as the automatic and replicability problem. In addition to this there is the problem of recording which data is being used, which data is being produced, how it was gathered or created and how the resultant data was processed. Given 20,000 experiments the amount of such data is not inconsiderable. Moreover this data needs to be in a form that is analysable and visualisable.

The first of these two problems, the automation and replicability of experiments may be and as a matter of course and routine is already solved by *scripting*. In the context of computing scripting is usually a program or sequence of instructions carried out by another program rather than the processor itself. This means scripting languages may be run by an operating system, for instance, rather than compiled down to machine code running directly on a processor. Scripting languages work in one of two ways. Either a program is is translated to byte code, each code represents an instruction to a *virtual machine*. This may be thought of as a software computer running on the computer, and has the obvious advantage that as long as the byte code does the same thing everywhere, then the code written in this language should be completely agnostic as to the hardware or operating system it is running on. Examples of this type of approach are Python, Java, Julia, Perl and C#. In particular NetLogo uses the Java virtual machine. These languages are very popular, as not only are they portable, they do not require the significant overhead in terms of configuration that truly compiled languages such as C or C++ require. Most of the compilation and linking process, being computationally intensive and historically slow has already taken place in the virtual machine. A modeller can write code once and it should run anywhere capable of running that the virtual machine for which the code was originally written. Because of this these scripting languages tend to be hardware and operating system agnostic (meaning the code does not 'care' where it is run) and more importantly fast to write and share.

More immediately relevant to the problem of automation and replicability, the other form of scripting language is a program consisting of a series of keywords which trigger sets of conveniently grouped, parameterised instructions. A program, known as the *the shell*, in Unix or Linux terminology interprets these keywords and their parameters to the more arcane instruction set required by the underlying processor to perform some computation, such as invoking an executable machine code model. These shells are pre-compiled parsers, so were historically limited in terms of their expressibility and vocabulary size. Hence, they are generally less expressive than the virtual machines discussed above, but are still Turing complete. This means the coding of complex algorithms can be achieved but is awkward and slow because the keywords are not optimised for general purpose computation, but for carrying out operations at level of the operating system. For example, file control, with atomicity at the granularity of a single file can be done very efficiently using these kinds of languages. Because the latter is effectively utilising the operating system of the computer directly, then this grouping of languages is generally used to control an existing computer. That is, they are used as *job control languages* (JCL). Examples of such languages are Bash, Ash, Dash, zsh, csh, Windows Power Shell, MSDOS, etc. Beyond the fact that scripting provides automation for large experiment sets, it also allows a certain degree of replicability, in that you to reproduce the experiment, then all that is required is that the scripts are rerun. However this is repeatability rather than replication. The aim is to just to rerun the experiment, rather than reproduce a given set of results. We believe that anybody who does serious agent-based modelling experimentation should at the very least be scripting most of their experiments. In our experience the preparation and the execution of the experiment is relatively straight-forward to script. Post processing to the final results of data, diagrams, etc., tends to be a little neglected in the hurry to publish results. Scripting is usually (although not exclusively) a text based activity, which means it is amenable to being programmatically produced.

The second problem of recording of data, we decided on the use of a relational database. This was selected on the basis that such technologies have a proven track record of recording and manipulating reasonably large volumes of data. This is an established technology. It is also portable and many proven libraries exist for the manipulation and use of such databases. And it is this context that the discussion of scripting languages and virtual machines becomes relevant. The approach we utilised when manipulating this database was a scripting language that utilised a virtual machine, specifically Python. As mentioned this gives a portability allowing us to develop on laptops and run the resultant code in high-performance computing environments with little, or no reconfiguration.

Thus the solution we selected was to use a proper JCL, in this case Bash to provide the necessary automation and replication models and call the provenance functions, written in a popular scripting language (Python) to update a standard relational database. In this case we selected Sqlite3 for local development and PostgreSQL for more heavy lifting in a distributed environment such as in high-performance infrastructure. Thus we have created a provenance tool which is based purely on scripting tools and a standard kind of database.

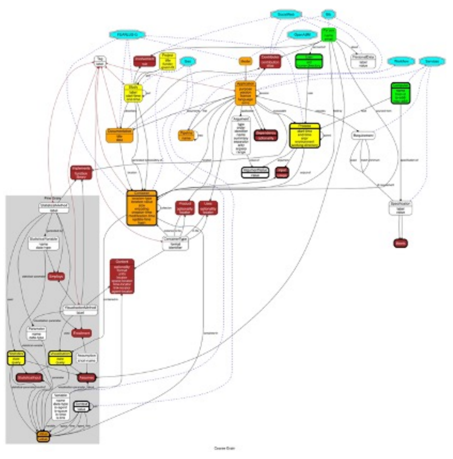

Fig. 1. SSREPI Schema

Since the inception of this project, we have subsequently uncovered a similar project which specifically concentrates on provenance data. This is the FAIR data pipeline [16] in which the provenance is automatically recorded by embedded code in R, C++, Java, Julia and Python. One of the aims was not to modify the original code, i.e. the original experiment as unchanged as possible. That is we wanted to change the framework code rather than the core code. The core code in the example above being the executable, pre-compiled in Objective-C [11]. Additionally the support code, i.e. the code that prepared the data, a Perl [13] script and the code that did the post analysis, a mixture of Perl and R [25]. This meant we developed the database access in Python [22], wrote wrapper scripts in Bash [21]. This differs in approach to [ibid] which is embedded in the code that constitutes the body of the experiment. Moreover our framework generalises to more more source metadata (paper, data, other experiments), and sis written with the requirements of agent-based modelling first and foremost. That is not to say that we do not have similar ambitions to automatically place provenance tracing code automatically embedded in experimental model code.

The rest of this paper, we describe the scripting tool we have developed for automatically recording metadata, which can be incorporated into the analysis replication process, and how this was used to regenerate some of the figures in [20]. In addition we show a simple tool we have already developed to trace data through the provenance graph, and as usual suggest additional work we would like to pursue and other ideas.

2 Method

One of the artefacts of the MIRACLE project [17] was the Social Simulation REplication Interface or SSREPI. This is the schema shown in figure 1. This schema has evolved since it original conception. The newest version of this document may be found here [19]. This is a database schema derived from the Dublin Core [27], the standard XML datatypes [2] and the PROV-O ontologies [15]. The schema is designed with agnosticisim towards the underlying database technology and as such has been implemented both in PostgresSQL [26] and Sqlite3 [4].

There are two important dimensions of distinction to this schema. First is fine- versus coarse- grained metadata. Second is provenance versus workflow. Coarse-grained metadata describes how particular files come (or came) into being, or were (or could be) used to bring other files into being. Fine-grained metadata describes specific values recorded in social simulation outputs. To make the distinction concrete, suppose a simulation produces a CSV file. The data within the CSV file are covered by fine-grained metadata, whilst the fact that the simulation produces the CSV file is coarse-grained. Turning to the other dimension, provenance metadata describes what actually happens (run W of simulation X produced output file Y), whilst workflow metadata describes what could happen (simulation X produces an output file of type Z). The distinctions are summarised in 2 .

Each table in schema shown in fig. 1 was coded as an object type in Python. Each table row was represented as an instance of such an object. This design approach was adopted to enforce a consistent coding methodology across all tables.

This then just left the interface between the two scripting languages. The design decision was taken to do domain specific processing in the Bash scripting language. This was done for due to the repetitive and time consuming nature of system testing: amendments in the job control language being easier to do quickly. This

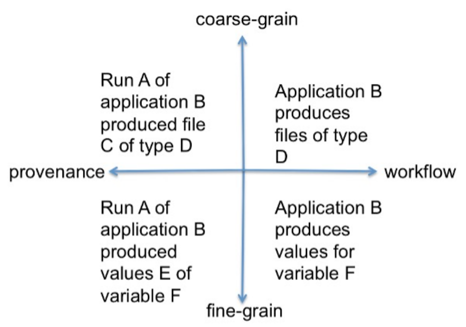
coarse-grain
Run A of application B produced file C of type D
Application B produces files of type D
provenance
workflow
Run A of application B produced values E of variable F
Application B produces values for variable F
fine-grain

Fig. 2. Fine grain vs coarse grain provenance

is a decision that may need revisiting.
The problem being that Bash invocations are computationally and time expensive. Complex processing should be abstracted away from the slower language, in this case job control language to improve speed and remove what might be regarded as inappropriate computation for such an environment. Consequently as the system stands the commands that coordinated between Bash and Python are few and simple, the harder processing having been pushed down into the job control level, i.e. in the Bash scripts.

These coordinating commands are:

- `create_database.py` - creates a database idempotently.
- `exists.py` - checks if a particular row in a table exists.
- `get_value.py` - gets any specified single value from a table given the primary key.
- `get_values.py`
- `next_study.py`
- `update.py` - idempotently updates a particular row in a table.

Note the use of idempotent, indicating that multiple operations on the same entity will leave it unchanged after the initial operation. For example multiplying something by 1 is idempotent. This allows for a less rigorous exception criteria, for instance if a row or entity already exists. This implies that initialisation must be performed with care, as older data will not necessarily be destroyed but overwritten with newer values, but also retaining any values that already exist. Again this design decision was expedient on rapid development and may need revisiting. However the schema is and its interaction is quite complex and this would involve critical path analysis

So each of the Bash commands listed in table **??** composes over the above python commands to populate the SSREPI database in a consistent and logical manner.

For the purposes of this short paper we are concentrating on a demonstration of recording provenance. Indeed we use the example, mentioned in the introduction of [19], and modified the original Bash [21] scripts to include the what we denote as *primitives*. In order to record provenance we coded the following primitives from the SSREPI interface definition [19]. As noted above these are implemented in a mixture of Python and Bash.

Primitive	Type	Purpose
`SSREPI_require\_minimum`	Metadata	Lower bound on software hardware required
`SSREPI_require\_exact`	Metadata	Exact bound on software hardware required
`SSREPI_application`	Provenance & Metadata	specifies some executable
`SSREPI_me`	Provenance & Metadata	Determines executable being run or returns a proper reference to the executable being run.
`SSREPI_run`	Provenance & metadata	Blocking invocation of an executable which will allow the specification of all inputs, outputs and arguments. Creates run-time provenance information as well
`SSREPI_batch`	Provenance & metadata	Non blocking invocation of an executable which will allow the specification of all inputs, outputs and arguments. Creates run-time provenance information as well
`SSREPI_argument`	Provenance	An argument type to an exectuable
`SSREPI_output`	Provenance	An output type from an executable
`SSREPI_input`	Provenance	An input type for an exectuable
`SSREPI_hutton\_person`	Metadata	Uses our institutions databases to populate metadata for a given individual

SSREPI_person	Metdata	Provide metadata for a particular actor within this system
SSREPI_project	Metadata	Specifies a project which contains all studies
SSREPI_study	Metadata	A set of experiments makes up a single study
SSREPI_set	Metdata	Sets the default licence and other metadata
SSREPI_involvement	Metadata	Links personnel to a study
SSREPI_paper	Metadata	A paper associated with this study
SSREPI_make_tag	Metadata	Used for building a folksonomy
SSREPI_tag	Metadata	Used to tag any entity with a folksonomy tag
SSREPI_contributor	Metadata	A person with some kind of relation to an executable or script.
SSREPI_statistical_method	Metadata	A statistical method is an approach to computing some statistics. It may be implemented in or as part of an application. A statistical method generates one or more statistical variables as its results, and may use the results of another statistical method in its computation. For example, computing the standard deviation of some data uses the mean of those data.
SSREPI_visualisation	Metadata	A visualisation is the process of creating an image to depict one or more (typically more than one) values. The results of a visualisation appear in a container.
SSREPI_statistics	Metadata	Statistics are activities that compute and populate the values of statistical rvariables. They operate on raw data that are retrieved from the values using a query. To replicate a set of statistics, the query can be rerun, selecting values that are pointed to by containers entries
SSREPI_visualisation_method	Metadata	This describes methods for generating visualisations, which then may appear in the content of a container produced by a process running an application that implements it.
SSREPI_implements	Metadata	Links a statistical or visualisation method to an application
SSREPI_parameter	Metadata	A Parameter is the name of a parameter taken by a statistical or visualisation method, used to configure the way it behaves.
SSREPI_statistical_variable	Metadata	A name for (one of) the result(s) of a statistical method.
SSREPI_visualisation_variable	Provenance & Metadata	Declares a named variable of interest
SSREPI_variable	Metadata	Names a variable of interest
SSREPI_statistical_variable_value	Provenance & Metdata	Sets an actual value for a named statistical variable
SSREPI_value	Provenance	Sets an actual value. This can be connected to any metadata value such
SSREPI_content	Metadata	Links a kind of output/input/argument to a variable
SSREPI_person_makes_assumption	Metadata	links a person to an assumption

Table 1: Available Bash commands for provenance and metadata gathering

Broadly speaking `SSREPI_application`, `SSREPI_run`, `SSREPI_batch`, `SSREPI_input`, `SSREPI_output` and `SSREPI_argument` are responsible for recording coarse grain provenance. `SSREPI_value`, `SSREPI_visualisation_variable_value`, `SSREPI_statistical_variable_value`, `SSREPI_run` and `SSREPI_batch` record fine-grain provenance. The remaining primitives are largely about recording metadata.

With the tool in the current state this involved laboriously going through the code and inserting the some of the Bash-based commands, shown in table 1. These commands have a fairly complex parameter set as it stands, and no way of checking whether the code worked other than running it, initially on a small subset of the experiment and then scaling up. This was extremely time-consuming.

Elaboration of all this may be found in the man pages and design documents in the public repository at https://github.com:/DougSalt/ABM-metadata.

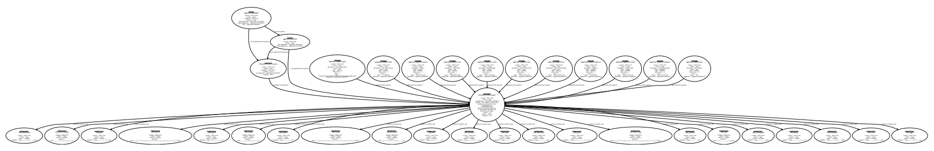

Fig. 3. The workflow sub-graph

We developed a very basic scheduler for use in an high performance computing environment, where there are many resources available both on the local machine. In particular for the situation where they is a single machine that has a large memory and many cores, the non-blocking version of the two execution primitives, specifically `SSREPI_batch` implements a kind of a simple scheduler locally using the sub-processing commands available to Unix-like systems. This was an historical solution to schedule in an high performance computing environment that existed prior to clustered computing situations. Clustering is the utilisation of one or more instances of such powerful computing machinery. Generally schedulers have been developed for such environments, as such SGE [8] and the one we use as default, Slurm [28]. Scheduling is now largely handled outwith the framework, which makes the code less complex.

`SSREPI_run`, the other executing primitive will block until the command, code or model is has invoked has completed. This is necessarily used in single threaded coordination context, such as gathering all outputs necessary before proceeding to the next stage of processing. For instance, initial post processing of an all experiments would represent such a single threaded context.

3 Results

The primary result is that the target results and diagrams from the original paper [18] were able to be recreated. As mentioned the method undertaken in this paper involved inserting the correct Bash-based commands, listed in table 1. These commands have a fairly complex parameter set as it stands, and no way of checking whether the code worked other than running it. This led to one of the immediate problems with this method. The only way to see if the code was working was to run it, a suck it and see methodology. This was extremely time consuming. We took the obvious approach and ran small subsets of the 20,000 runs or so, to see if we could get the framework to work and then scaled it up to the full run, but invariably we ran into new problems the more we increased the scale. It therefore took substantial time to even get this first replication to run. It took several months of coding effort to get this correct. Generally it would alternate between parameter errors for the calls in table 1, and errors in the scripts making calls to record the provenance data to the database. Each iteration would take weeks to run, before failure as we increased the scale.

After a run had completed successfully the database was run through several programs to produce the sub-graphs detailed in fig. 1. These programs are capable of producing the following subgraphs:

– **Analysis** - fine grain provenance pertaining to statistical and visualisation outputs.
– **Finegrain** - a provenance diagram down to the level of variables.
– **Folksonomomy** - a diagram showing annotations against the database, produced and categorised at the discretion of the user doing the annotation
– **Project** - management metadata. Largest granularity of metadata supported

– **Provenance -** provenance diagram at the level of file and parameter
– **Services -** service provided and requirement description
– **Workflow -** the actual workflow

We developed programs that took the relational data in order to produce "dot" files. Such files that can be processed by Graphviz [7], and were used to produce the diagrams in figures 4, 3, 5 and 6 .

To give an idea of the visualisation available we show the workflow graph in fig. 3. This is a section of the workflow and is relative easy to follow. The resultant provenance graph is both massive and massively complex, given there were 20,000 runs (so 20,000 sets of outputs to record), so we only show a *very small* section of the provenance graph in fig. 4.

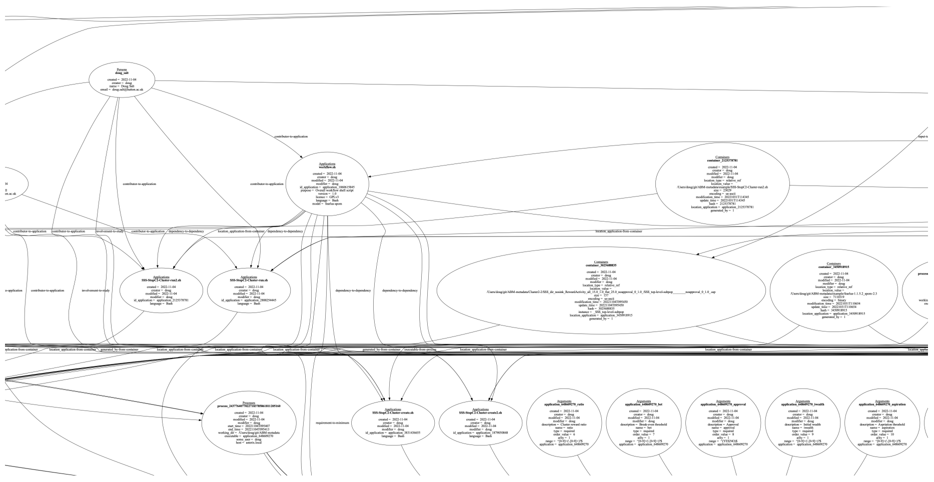

Fig. 4. A small sub-section of the proevenance graph

So what use is this provenance metadata? Beyond replicability? Say for instance we have a bad dataset. The challenge is not only how to visualise the provenance, but to use this data in a useful manner. We propose a simple use here. The "dot" language used by Graphviz constitutes a primitive graph database. Indeed there are programs that can transform "dot" files into TinkerPop GraphSON format [**?**,**?**]. Rather than using the relational database data we used the resultant dot files listed above to trace bad entities. [1]

So for an easy to see example we pick the entity `Applications.application_3831436655`, and see how this affects other entities in the graph. This is `SSS-StopC2-Cluster-create.pl` and we are using the workflow graph to see what might be affected. And this is

[1] These are directed graphs (unproven as to whether they are acyclic - they shouldn't be they are the result of a relational database which was designed to be and *should* be acyclic).

shown in 5. It might be for instance that there is a bug in this program, so we should like to see how the errors might propagate out through the workflow.

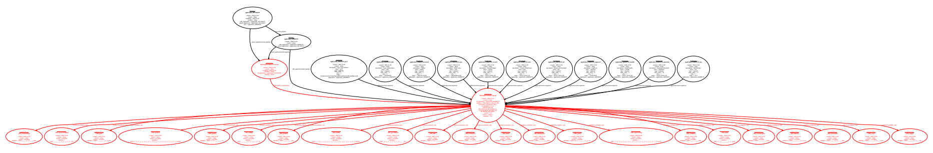

Fig. 5. The workflow sub-graph

That example was somewhat contrived and simplistic and gives a flavour. However, what if we wanted to trace a bad data set? If we ignore for this original example that the datasets for the above example are generated automatically by Perl scripts, and we pick one randomly to illustrate how this approach might work. The dataset in question would be: `Containers.container_42949672955`, a data file that forms a input file to just one experiment. And here we see, again, a *very small* section of of the provenance graph showing what is affected in red. This is shown in fig. 6

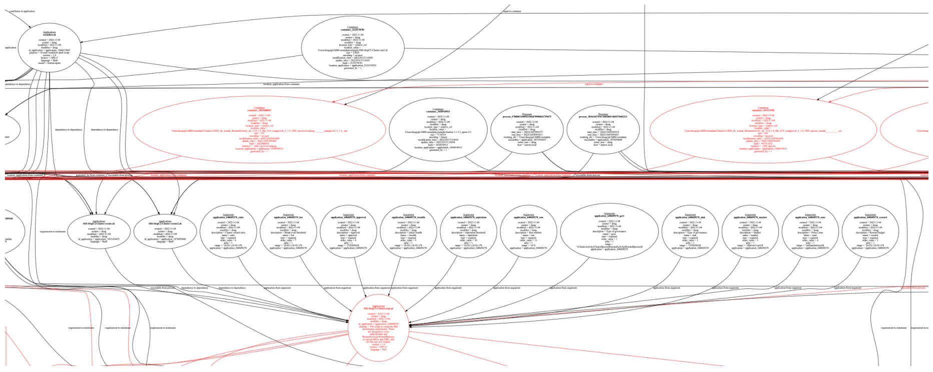

Fig. 6. The workflow sub-graph

Moreover we can list out the affected entities. In this case, those entities would be:

- `Applications.application_3450918915`
- `Applications.application_648609270`
- `Computers.asterix.local`
- `Containers.container_1814970370`
- `Containers.container_1982026419`
- `Containers.container_2050039078`
- `Containers.container_2056384913`
- `Containers.container_2060874102`
- `Containers.container_2387213333`
- `Containers.container_2486610989`
- `Containers.container_2582525701`
- `Containers.container_2759060318`

```
— Containers.container_2865400753
— Containers.container_3025688835
— Containers.container_3307537171
— Containers.container_3470971297
— Containers.container_354343442
— Containers.container_4235735972
— Containers.container_4294967295
— Containers.container_441913555
— Containers.container_505627104
— Containers.container_800277554
— Containers.container_878886043
— Containers.container_900718909
— Persons.doug
— Persons.doug_salt
— Processes.process_23222129888649347509041510897074
— Processes.process_32647549959702293885495865014666
— Specifications.R
— Specifications.bash
— Specifications.cpus
— Specifications.disk_space
— Specifications.memory
— Specifications.os
— Specifications.perl
— Specifications.python
— Studies.1
— Users.doug
```

4 Discussion

What we have learnt from this? As noted the process of running the experiment with the new provenance and metadata commands was time-consuming to the point of almost being unusable. It involved the repeated running of the experiment with a larger and larger array of inputs until we matched the size of the original experiment. This has taken several months of intense work to event get it running. Repeating an experiment in a completely automated fashion turns out to be a really hard task.

However, we have to emphasise that this was the reproduction of the experiment. If this framework were available from the start then most of the work would be done once the experiments had been completed; that is, the work needed to actually repeat the experiment in a consistent and reliable manner. This is the key, and it reduces to the same advice that meta-data advocates everywhere proclaim, which is to incorporate the metadata from the start and not as an after thought.

Provenance should be a directed acyclic graph. There may well be loops in the schema as it stands. Given sufficient time we can certainly normalise the relational database to remove redundancy and the conflict arising from such, and once this complete, we can formally prove the schema to be acyclic. The authors have done their best to normalise it to uncover such cycles, but have yet to formally do so. In a schema this complex they may well still exist. Until the schema is formally analysed, then we can not say with confidence that the schema is actually consistent, in the sense that any recorded provenance is not inherently contradictory. Given that such a schema is likely to be continually revised as the underlying provenance model changes or additional functionality is required, then it would be advisable to automate such proving of consistency (assuming correct normalisation).

In addition to reproducing the experiment automatically, there was some scope in utilising the parallelising infrastructure of the high performance computing environment. One of the nice things was that we were able to parallelise part of the post processing job, and thus speed up the whole experiment as this was proving to be a bit of a bottleneck. However, careful thought needs to be given to this. In the case of agent-based modelling such stochasticism in inherent to the testing methodology, and it would be expected that such changes should have little impact. For other testing frameworks, particular those that do not rely on massive replication, then such changes may well have unintended consequences. Indeed, those with a sharp eye may have noticed the versions of Perl, Python and R we use were not around when the original experiments were run and obviously this has implications. However, the framework does offer the ability to record and enforce metadata about programming versions.

The underlying database is almost certainly unsuitable. We should probably be using a graphing language such as Gremlin [23]. The advantages of using graphing databases over relational databases is that such languages are inherently designed to store, traverse and query the graphs that are the main product of this framework. This makes querying in such languages trivial and fast. In a relational databases the SQL statement to do this are cumbersome, awkward and probably error prone given their size. Moreover on huge datasets they are reportedly slow. Relational databases do have the advantage of being an extremely mature technology and the availability of utilities that implies. As mentioned, the advantage of using the relational database is that structured query language is standard for all such databases, and therefore queries written for any relational database should be the same. A tool we already use, Graphviz already does act like a graphing database language to a certain extent. Indeed we use the Graphviz "dot" files, rather than the relational schema in our simple tools tools to trace bad data through our provenance graphs. However the "dot" language of Graphviz is primarily purposed as a diagramming language and lacks the sophistication, such as a built-in query language of say, Gremlin.

For future we work, we should eventually like to take many provenance databases run some machine learning across them to see if there are any commonalities. We hope to uncover similarities in setup, execution and post-processing that could form core, reusable and proven components for primarily agent-based modelling but also for other modelling frameworks.

The plan is to adapt this provenance framework to other model running language frameworks, in particular R, Python, Julia, Java and thence NetLogo. This would take the approach of the FAIR data pipeline [16], but unlike the example we have presented here such provenance would be embedded in the experiments as a matter of course. We were constrained by the nature of the replication here to use Bash as our means of recording provenance. However this does show that such a provenance framework may be "retrofitted" to existing experiments (although possibly at the cost of the programmer's sanity given the minutiae involved). We have tentative potential adopters using R and Python as their primary modelling language.

References

1. Ayllón, D., Railsback, S.F., Gallagher, C., Augusiak, J., Baveco, H., Berger, U., Charles, S., Martin, R., Focks, A., Galic, N., et al.: Keeping modelling notebooks with trace: Good for you and good for environmental research and management support. Environmental Modelling & Software **136**, 104932 (2021)
2. Biron, P.V., Malhotra, A., Consortium, W.W.W., et al.: Xml schema part 2: Datatypes. https://www.w3.org/TR/xmlschema11-2/ (2004), accessed: 2023-01-16
3. Boero, R., Squazzoni, F.: Does empirical embeddedness matter? methodological issues on agent-based models for analytical social science. Journal of artificial societies and social simulation **8**(4) (2005)
4. Consortium, T.S.: Sqlite syntax. https://www.sqlite.org/lang.html (2023), accessed: 2023-01-16
5. Di Tommaso, P., Chatzou, M., Floden, E.W., Barja, P.P., Palumbo, E., Notredame, C.: Nextflow enables reproducible computational workflows. Nature biotechnology **35**(4), 316–319 (2017)
6. Edmonds, B., Hales, D.: Replication, replication and replication: Some hard lessons from model alignment. Journal of Artificial Societies and Social Simulation **6**(4) (2003)
7. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—open source graph drawing tools. In: Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9. pp. 483–484. Springer (2002)
8. Gentzsch, W.: Sun grid engine: Towards creating a compute power grid. In: Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid. pp. 35–36. IEEE (2001)
9. Gimona, A., Polhill, J.G.: Exploring robustness of biodiversity policy with a coupled metacommunity and agent-based model. Journal of Land Use Science **6**(2-3), 175–193 (2011)
10. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucleic acids research **34**(suppl_2), W729–W732 (2006)
11. Inc, A.: Objective-c programming language. https://developer.apple.com/library/archive/documentation/Cocoa/Co
accessed: 2023-04-11
12. Köster, J., Rahmann, S.: Snakemake—a scalable bioinformatics workflow engine. Bioinformatics **28**(19), 2520–2522 (2012)
13. Language, P.P.: Perl 5.6.3. https://perldoc.perl.org/5.26.3/perldelta, accessed: 2023-04-11, Version = 5.6.3
14. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the kepler system. Concurrency and computation: Practice and experience **18**(10), 1039–1065 (2006)
15. Missier, P., Belhajjame, K., Cheney, J.: The w3c prov family of specifications for modelling provenance metadata. In: Proceedings of the 16th International Conference on Extending Database Technology. pp. 773–776 (2013)
16. Mitchell, S.N., Lahiff, A., Cummings, N., Hollocombe, J., Boskamp, B., Field, R., Reddyhoff, D., Zarebski, K., Wilson, A., Viola, B., et al.: Fair data pipeline: provenance-driven data management for traceable scientific workflows. Philosophical Transactions of the Royal Society A **380**(2233), 20210300 (2022)

17. Parker, D.C., Barton, M.J., Filatova, T., Polhill, J.G., Jin, X., Lee, A., Lee, J.S., (Wright), K.R., , Pritchard, C.: Final white paper: MIRACLE project (MIning Relationships Among variables in large datasets from CompLEx systems). Tech. rep., Arizona State University and University of Twente and University of Waterloo and The James Hutton Institute (01 2019), accessed: 2023-03-11
18. Polhill, G., Milazzo, L., Dawson, T., Gimona, A., Parker, D.: Lessons learned replicating the analysis of outputs from a social simulation of biodiversity incentivisation. In: Advances in Social Simulation 2015, pp. 355–365. Springer (2017)
19. Polhill, G., Milazzo, L., Parker, D., Jin, X., Pritchard, C., Lee, J.S., Salt, D.: Miracle simulation outputs metadata specification version 2.0.0. Tech. rep., The James Hutton Institute (2022), accessed: 2023-04-11
20. Polhill, J.G., Gimona, A., Gotts, N.M.: Nonlinearities in biodiversity incentive schemes: A study using an integrated agent-based and metacommunity model. Environmental Modelling & Software **45**, 74–91 (jul 2013). https://doi.org/10.1016/j.envsoft.2012.11.011, http://linkinghub.elsevier.com/retrieve/pii/S1364815212002824
21. Project, G.: Bash 4.4.20. https://tiswww.case.edu/php/chet/bash/bashtop.html, accessed: 2023-04-11, Version = 4.4.20
22. python.org: Python 3.6.8. https://www.python.org/downloads/release/python-368/, accessed: 2023-04-11, Version = 3.6.8
23. Rodriguez, M.A.: The gremlin graph traversal machine and language (invited talk). In: Proceedings of the 15th Symposium on Database Programming Languages. pp. 1–10 (2015)
24. Schmolke, A., Thorbek, P., DeAngelis, D.L., Grimm, V.: Ecological models supporting environmental decision making: a strategy for the future. Trends in ecology & evolution **25**(8), 479–486 (2010)
25. for Statistical Computing, T.R.P.: R 4.2.2. https://www.r-project.org/, accessed: 2023-04-11, Version = 4.2.2
26. Stonebraker, M., Kemnitz, G.: The postgres next generation database management system. Communications of the ACM **34**(10), 78–92 (1991)
27. Weibel, S.L., Koch, T.: The dublin core metadata initiative. D-lib magazine **6**(12), 1082–9873 (2000)
28. Yoo, A.B., Jette, M.A., Grondona, M.: Slurm: Simple linux utility for resource management. In: Job Scheduling Strategies for Parallel Processing: 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003. Revised Paper 9. pp. 44–60. Springer (2003)