Intermediate Notation for Provenance and Workflow Reproducibility

Danius T. Michaelides[1], Richard Parker[2], Chris Charlton[2], William J. Browne[2], and Luc Moreau[1]

[1] Electronics and Computer Science, University of Southampton, UK
{dtm,L.Moreau}@ecs.soton.ac.uk
[2] Graduate School of Education, University of Bristol, UK
{Richard.Parker,C.Charlton,William.Browne}@bristol.ac.uk

Abstract. We present a technique to capture retrospective provenance across a number of tools in a statistical software suite. Our goal is to facilitate portability of processes between the tools to enhance usability and to support reproducibility. We describe an intermediate notation to aid runtime capture of provenance and demonstrate conversion to an executable and editable workflow. The notation is amenable to conversion to PROV via a template expansion mechanism. We discuss the impact on our system of recording this intermediate notation in terms of runtime performance and also the benefits it brings.

1 Introduction

Reproducibility of scientific results is a key challenge to the modern scientist[1]. Systems that have been built to tackle this often focus on recording provenance, especially scientific workflow systems.

The focus of the EBook project[3] is on a suite of tools called StatJR[4] designed to aid in the use and teaching of statistical analysis techniques with a particular emphasis on their use in social science. The tool suite consists of a Web front-end to statistical processes, a command line interface and a dynamic document system, whereby interactive computations can be embedded in a document[2]. They are designed to support the user as their experience and understanding of statistical methods improves by surfacing different levels of detail of the underlying computations. More recently, the suite was enriched by a workflow system that enables the composition of processes from low-level operations to broad methodological steps. The overall aim is to allow users to move seamlessly between the StatJR tools in order to refine the activity they are engaged in, whilst maintaining their context and the choices they have made so far in their interactive computational investigations. Concretely, it is a requirement to be able to capture both the interactive investigations and the batch processing that took place, convert them to editable workflows that may be further refined, before

[3] http://www.bristol.ac.uk/cmm/research/ebooks/
[4] http://www.bristol.ac.uk/cmm/software/statjr/

being packaged as downloadable web-enabled documents, which support full reproducibility of the computations.

Many scientific workflow tools, such as Taverna[3], Kepler[4] and VisTrails[5], are monolithic "integrated development environments". Instead of locking a user into a single tool, we seek to facilitate their mobility between tools, to allow them to use the best tools for the job. This motivates the need for recording provenance in a manner that allows multiple tools to be used. Our approach is more akin to YesWorkflow[6] allowing multiple tools to be used in the scientific processes.

Furthermore, reproducibility is a key direction of development for many workflow tools. For example, ReproZip[7] enables reproducible experiments by monitoring command-line executions and packaging required resources into a single, distributable package. Its integration with VisTrails aids reproducibility by creating a suitable workflow of the original experiment for running within VisTrails. In that context, Moreau[8] sees provenance as a "program", which when interpreted, can reproduce results. Our take on this is the ability to translate the trace of a process execution back into an executable specification that is also editable.

The aim of this paper is to present a PROV-based technique to capture provenance at runtime from multiple tools, and to provide the capability to automatically convert them into reproducible and editable workflows. Specifically, our contributions are fourfold. *i*) INPWR (Intermediate Notation for Provenance and Workflow Reproducibility), an intermediate representation capturing key values logged at runtime, from the various tools in a tool suite; *ii*) A conversion of INPWR to PROV by means of a template expansion mechanism; *iii*) A conversion of INPWR into an editable and executable workflow, which when executed would result in the same provenance; *iv*) A quantitative evaluation demonstrating that the approach is tractable in terms of size of representations and computational costs.

The paper is organised as follows. We discuss some application requirements in Sect. 2 and our computation model in Sect. 3. In Sect. 4 we introduce INPWR and demonstrate how we capture logs in Sect. 5. We demonstrate how the INPWR representation enables the generation of PROV graph data as well as generation of new workflows in Sect. 6. In Sect. 7 we evaluate the costs of our notation and go on to look at related work. Finally, we present our conclusions and further work.

2 Application Requirements

In this section we discuss some of the requirements of application to provide some context and motivation for this work.

Guide a reader through steps of an analysis The system should enable a user to step through a complex analysis. The steps in the analysis could vary in size from broad methodological steps down to low-level operations. They should match the user's cognitive understanding of what is going on.

Adaptable The user should be able to influence the path taken through an analysis. Analyses should support user input, branching and repetition.

Allow the results to be reproduced The system should support the reproducibility of the analysis in light of choices made by the user. Specifically: 1) published material - provide supporting evidence for publication 2) automation - rerun the analysis e.g. to verify results or run with a different dataset 3) logbook - be a record of what actions were performed in the analysis for the user to refer/return to.

Ease of authoring Authoring of an analysis should be available to all types of users. It should be easy to adapt/extend/repurpose an analysis. There should be a tight link between edit and running to facilitate explorative analysis and pedagogy.

From these application requirements, we derive the following technical requirements:

1. Capture information about the steps taken in the analysis into a log in sufficient detail for re-use whilst remaining concise.
2. Be able to transform that log into outputs for different purposes.
3. Transformation of log back into analysis should include user input and unwind branching and repetition. This revised analysis is different to storing the original analysis alongside inputs made as reflects the exact steps taken to complete the analysis.

3 Computational Model

StatJR uses the Blockly visual programming system[9]. Blockly is designed to aid non-programmers in writing short scripts and as a toolkit for building visual programming language, it focusses on extensibility. Blockly opts to take an imperative programming approach in contrast to the dataflow approach of many scientific workflows. Blocks can be statement blocks or expression blocks and they have the notion of containment of other blocks for scoping and traditional flow control. Statement blocks are composed into a sequence of blocks.

In StatJR, we extend the selection of blocks available to include blocks that perform common statistical processes. In addition, as blocks can encapsulate large computations, statement blocks can produce named outputs.

A simple workflow is shown in Fig. 1 which consists of a sequence of two blocks: 1) add a new variable (`normexam2`) to current dataset by squaring another variable (`normexam`) and 2) calculate some summary statistics about the revised dataset. Variables, in this context, are statistical variables (i.e. named columns in a dataset).

We model the computation in our system in terms of tasks with named inputs and outputs. All tasks and values are uniquely identified. Tasks are invoked by a parent task.

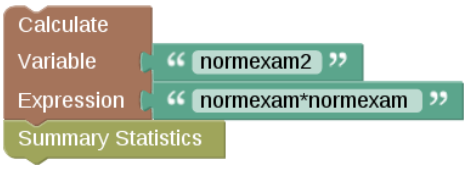

Fig. 1. Simple StatJR Workflow.

4 INPWR Notation

The INPWR notation captures the salient detail of the execution of a task as defined in the previous section. The notation consists of a set of variables and their values as shown in Fig. 2. They fall into two categories; 1) pertaining to

Variable name	Type	Description
block_instance	uri	identifier for this execution
parent	uri	the parent of this execution
starttime	date	when the block started executing
endtime	date	when the block finished executing
block_uri	uri	refers to the block
block_title	string	the name of the block
block_type	uri	the type of the block
consumed	list of URIs	entities consumed
consumed_at	list of dates	when they were consumed
consumed_name	list of strings	their names
produced	list of URIs	entities produced
produced_at	list of dates	when they were produced
produced_name	list of strings	their names
literal	list of URIs	identifiers of literal values
literal_value	list of strings	their value
literal_type	list of strings	their type

Fig. 2. Binding variables for an execution of a block.

information about the task and 2) linking to resources consumed and produced. Tasks are uniquely identified by the instance URI[5], but also have a link to the block in the original workflow document and also its type. The parent task is stored to indicate the task hierarchy. Resources are linked to a task via a named port as with many dataflow workflow systems[10]. Values of literals are also stored in variables. All inputs and outputs (including literals) are given a generated URI with the exception of resources that are supplied with StatJR (such as datasets) which have a static URI. The `consumed`, `consumed_at` and `consumed_name` variables are lists used to store details about the consumption of resources i.e. the nth resource $\texttt{consumed}_n$ was used at $\texttt{consumed\_at}_n$ on the named port $\texttt{consumed\_name}_n$. A similar pattern is used for produced resources and any literals.

[5] We generate UUIDs and use the `urn:uuid:` scheme.

Variable name	Block 1	Block 2	Block 3
block_instance	urn:uuid:1	urn:uuid:2	urn:uuid:8
parent	-	urn:uuid:1	urn:uuid:1
starttime	2016-02-12T15:12:28.543093	2016-02-12T15:12:28.546712	2016-02-12T15:12:28.679199
endtime	2016-02-12T15:12:29.527988	2016-02-12T15:12:28.677037	2016-02-12T15:12:29.527225
block_uri	rqvik2xqakayemazt813	pgno3ns6cur7ej7yxhju	6fdqrmkq5n8fuq57qfti
block_title	Sequence	Calculate	DatasetSummary
block_type	estatwf:Sequence	estatwf:Calculate	estatwf:DatasetSummary
consumed		urn:uuid:3 (1) urn:uuid:4 (2) estat:datasets/tutorial	urn:uuid:5
consumed_at		2016-02-12T15:12:28.546846 2016-02-12T15:12:28.546846 2016-02-12T15:12:28.546846	2016-02-12T15:12:28.679253
consumed_name		expression column dataset	dataset
produced		urn:uuid:5 urn:uuid:6 urn:uuid:7	urn:uuid:9 urn:uuid:10 urn:uuid:11
produced_at		2016-02-12T15:12:28.676943 2016-02-12T15:12:28.676943 2016-02-12T15:12:28.676943	2016-02-12T15:12:29.527171 2016-02-12T15:12:29.527171 2016-02-12T15:12:29.527171
produced_name		inputs output script.py	script.py inputs table
literal		urn:uuid:3 (1) urn:uuid:4 (2)	
literal_value		normexam*normexam (1) normexam2 (2)	
literal_type		xsd:string (1) xsd:string (2)	

Fig. 3. Example INPWR records for 3 blocks.

INPWR notation generated from executing the example workflow Fig. 1 are shown in Fig. 3. For clarity uuid references have been renamed. Note that Block 2 consumes two literal values "`normexam2`" (the name of the new dataset variable) and "`nornexam*normexam`" (an expression which the Calculate block which is passed directly to the underlying code in StatJR). In this case, the URIs in the consumed variable (marked 1 and 2 in circles) refer to values defined in the variables for literals. Both blocks output intermediate resources such as marshalled inputs and the underlying Python code executed by the step in line with our project goals of exposing pedagogical material to the user. A JSON representation of the variables from block 2 can be seen in the appendix.

5 Capture

The StatJR workflow interpreter was augmented to capture execution information in INPWR notation. The key points for capture during interpreting a workflow are at the beginning of a block evaluation, at the point its input arguments have been evaluated, at the point when outputs are generated and finally

at the end of block evaluation. At each capture point, a subset of variables are captured; we call this a *binding fragment*. The binding fragment is appended to a log along with the type of recording: *begin*, *input*, *output* and *end*.

To generate the complete notation log after execution, we iterate over the binding fragment log and use a stack to aid in combining the fragments. A *begin* fragment pushes a new INPWR record onto the stack, filling in appropriate variables including the parent variable (the `block_instance` from the next block on the stack). *Record input/output* appends values to the appropriate variables (i.e. `consumed*` or `produced*`). *End* finalises a binding setting the `endtime` variable, pops the INPWR record from the stack, and commits it to the log.

6 Transformations

In this section, we look at how the INPWR records that we captured in Sect. 5 can be transformed into outputs useful in the StatJR system.

6.1 PROV output via Templates

The PROV-Template system[11] allows the generation of PROV[12] documents by combining a template with a binding. The template is a PROV document which contains variables acting as placeholders for values and a binding document contains values for those variables. A provenance document is created by expanding a template against a binding; templates include special attributes that control this expansion process.

The process log recorded using INPWR is an ordered list of sets of variables, one for each step in the execution. Each set of variables in INPWR are the equivalent of a binding in PROV-Template. Hence, creating a complete PROV graph for an execution of a workflow involves expanding each binding in the INPWR log against a template and merging the resulting documents.

Figures 4 and 5 show an example template in graphical form and PROV-N representation. This PROV document is modelled on the computation model from Sect. 3 and articulates the process hierarchy, entities consumed and produced, and their derivations as well as supplementary information about time and types. In PROV-Template, attributes in the `tmpl` namespace map to specific elements in PROV-DM[12] where the typing does not allow a Qualified Name, for example the attribute `tmpl:startTime` on an Activity corresponds to the activity's start time. Note that in this template, we choose to model literal values as entities and whilst in the template the `var:literal` entity is disconnected, references to it will appear in the `consumed` or `produced` variables and the graph will be connected after expansion. This can be seen in Fig. 6 showing the provenance graph derived from the INPWR log from an execution of the workflow from Sect. 5. In this paper we use a single template for all INPWR log entries, however templates could be applied selectively, perhaps based on the value of a variable (`block_type` for instance).

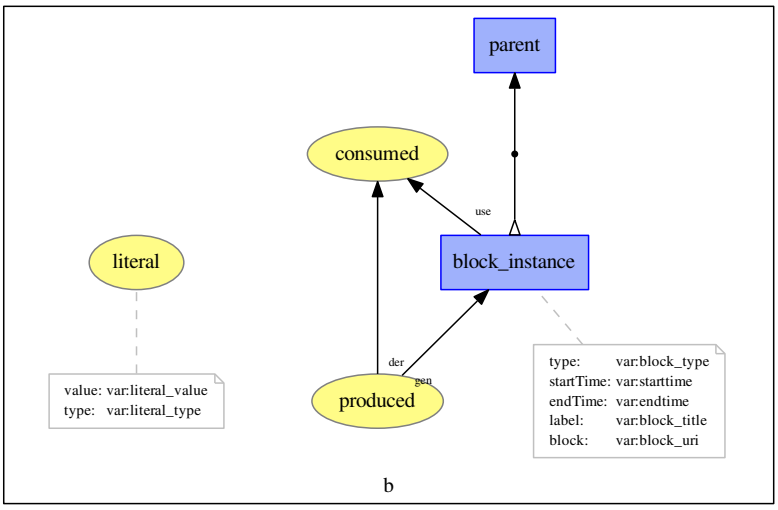

Fig. 4. Graphical representation of a template.

```
document
  prefix tmpl <http://openprovenance.org/tmpl#>
  prefix var <http://openprovenance.org/var#>
  prefix vargen <http://openprovenance.org/vargen#>
  prefix estat <http://purl.org/net/statjr/ns#>
  prefix estatwf <http://purl.org/net/statjr/wf#>

  bundle vargen:b
    activity(var:block_instance, -, -,
             [ tmpl:startTime='var:starttime', tmpl:endTime='var:endtime',
               prov:type='var:block_type', tmpl:label='var:block_title',
               estatwf:block='var:block_uri' ] )
    activity(var:parent,-,-)
    wasStartedBy(var:block_instance, -, var:parent, -, [tmpl:time='var:starttime'])
    entity(var:consumed)
    used(var:block_instance, var:consumed, -,
         [ tmpl:time='var:consumed_at',
           estat:bindingname='var:consumed_name'] )
    entity(var:produced)
    wasGeneratedBy(var:produced, var:block_instance, -,
                   [ tmpl:time='var:produced_at',
                     estat:bindingname='var:produced_name'] )
    entity(var:literal, [estatwf:value='var:literal_value', estatwf:type='var:literal_type'])
    wasDerivedFrom(var:produced, var:consumed, -, -, -)
  endBundle
endDocument
```

Fig. 5. PROV-N representation of the template in Fig. 4.

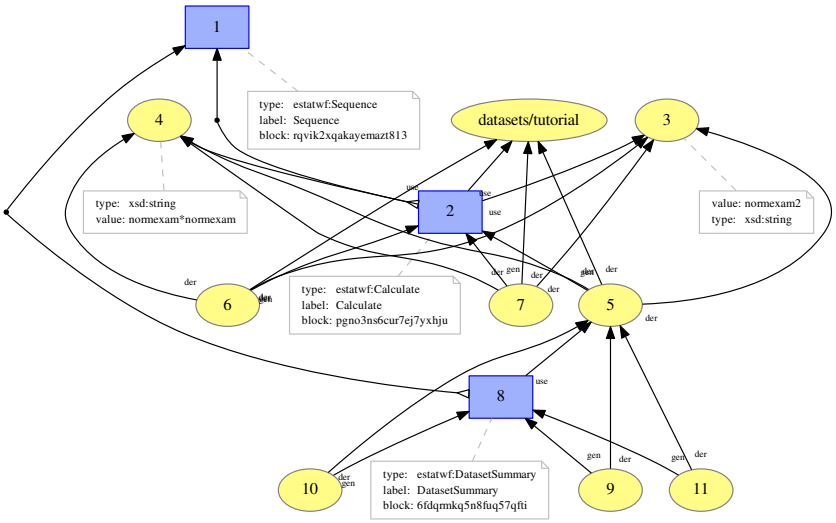

Fig. 6. Provenance graph derived from expansions of template of Fig. 5 and INPWR records of Fig. 3.

6.2 Workflow output

The INPWR log captured is sufficiently descriptive to allow conversion to a workflow. Conversion from INPWR back into the original workflow is not possible (due to conditionals), however a reconstruction of the exact steps taken to generate a given output is beneficial for reproducibility and also for moving activity between tools in our system. In addition, the INPWR notation may not have been generated directly by the workflow system, and may instead come from one of the other StatJR tools.

Blockly uses an XML format to serialise the abstract syntax tree of a Blockly program. Converting an INPWR record to Blockly XML involves creating the appropriate XML node and recursively generating nodes for the consumed resources. A consumed resource is found by searching the outputs of the immediate children of the current block. Special cases occur for sequences, control structures and user input. Reconstruction of sequences is performed by looking for blocks which have the same the parent and then ordering by time. For control structures, special care has to be taken to evaluate any conditions in case they created a side-effect; their results are discarded in the generated workflow. We unwind all loop structures. User input blocks are replaced with literal values that represent the input made.

Figure 7 shows an example workflow (left) and a reconstructed workflow from the INPWR log or a run (right). The workflow (left) consists of a loop asking

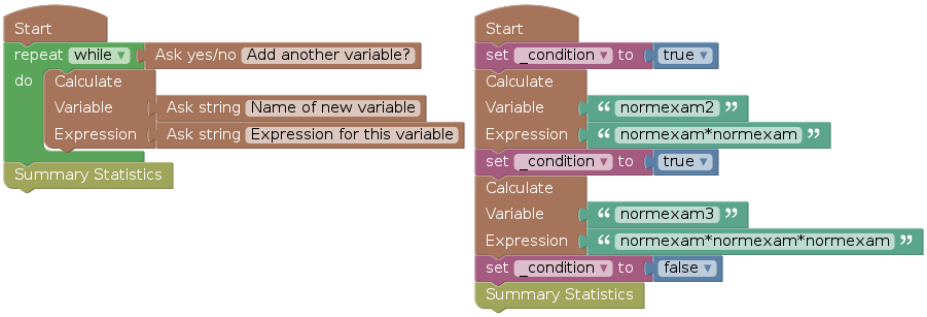

Fig. 7. Workflow (left) enabling the user to iteratively add new variables to a dataset, and resulting workflow reconstruction (right) after two variables were added.

the user whether they want to add a new variable, and a calculate block to construct a new variable based on asking the user the name of the new variable and an expression for it. A variable in this context is a statistical variable i.e. a column in a dataset. This calculate block modifies the currently active dataset by adding the column. This workflow was executed in which the user adds 2 variables (`normexam2` and `normexam3`), answering the "Add another variable?" question `Yes`, `Yes` and `No`. The reconstruction of this run is shown on the right of the figure, with the evaluation. The unwinding of the repeat/while loop can be seen with the repetition of the Calculate block and the condition translated to 3 instances of setting the variable `_condition`. The values of the loop conditional were inputs made by the user, and were translated to boolean literals `true`, `true` and `false`.

7 Evaluation

To establish that the cost of recording an INPWR log is not too onerous we look at memory and CPU usage when running a selection of workflows. Workflows were based on real-world examples and vary in size from small workflows which, for example, perform linear regression with 3 variables (`reg3`) or generate X-Y plots of multiple variables (`plotloop`) to larger workflows which perform more lengthy analysis (`lemma3` and `big`).

Figure 8 shows some memory usage metrics for the workflows. Python sizes are calculated by traversing the Python structures and applying `sys.getsize()`. The Python fragment sizes equate to the runtime memory overhead. Python records are the post-computation in memory cost and PROV-N records is the cost of serialising to disk. On average, we see a per record overhead of approximately 6kB during runtime, 3.4kB post-computation and on disk cost of 1.8kB per record.

	blocks	records	fragments		INPRW records		expansion	variables
name	#	#	#	python	python	prov-n	prov-n	#
reg3	18	40	308	283,714	177,309	81,126	81,439	1,364
lemma3	74	69	441	414,514	267,197	122,939	97,324	2,082
plotloop	25	75	393	371,821	245,102	120,898	74,557	2,004
big	259	235	1,501	1,360,741	859,482	415,609	305,844	7,088
Average per record	–	–	6	5,962	3,395	1,798	1,436	30

Fig. 8. Memory overhead for a number of workflows (Python memory usage and PROV-N in bytes), including per record averages.

	no provenance	provenance	overhead
name	seconds	seconds	%
reg3	1333.16	1334.97	0.14
lemma3	1196.58	1202.45	0.49
plotloop	198.78	200.15	0.86
big	783.61	790.37	0.86

Fig. 9. Runtime overhead of tracking provenance across a selection of workflows - total runtime for 100 runs.

Runtime was measured as the wall-clock time for running each workflow 100 times with and without provenance capture. Figure 9 shows the results and the runtime overhead calculated as percentage.

8 Related work

Many scientific workflow systems capture provenance[13] with a distinction made between prospective and retrospective provenance. VisTrails[5] uses an SQL database for storage of retrospective provenance and an XML serialization for storing prospective provenance. It also records provenance about the evolution of workflows as they are edited. Taverna[3] collects retrospective provenance for use internally with in its workbench and provides export in the form of PROV-O. The noWorkflow system[14] captures provenance information from scripts without the need to instrument them. They use language dependent (primarily Python) methods to capture runtime provenance from of function activations and I/O events. They store their structured data in an SQL database. YesWorkflow[6] provides script authors with an annotation mechanism to describe prospective provenance. Their annotations are placed in language comments to be extracted by YesWorkflow tools; many scientific languages are supported.

Converting workflow traces back into valid workflows was the subject of the third Provenance Challenge[15]. To guarantee losslessness in the conversion from OPM back into a valid Taverna workflows additional annotations are needed[16].

9 Conclusion and Future work

This paper has introduced an intermediate notation for recording provenance for use across multiple tools in StatJR. Our notation is compatible with existing PROV tools allowing easy generation of PROV graphs. We also demonstrated that the notation is expressive enough to support conversion to executable workflow and we have discussed benefits to our system of being able to do so. We measured the overhead of recording our notation at runtime and found that it is tractable.

It is clear that in-memory and on-disk management of the INPWR notation lends itself to optimisation, in particular in the presence of large workflows. Whilst we have made no attempt to optimise storing of INPWR records, one possible avenue is to look at storing the records in SQL databases as many workflow systems do.

Our notation is amenable to other transforms. For example, D-PROV[10] introduces extensions to PROV to express structural features in typical dataflow models. Generating D-PROV from INPWR would simply involve introducing the new relations to the PROV-Template system.

Acknowledgments

This research was supported by the UK's Economic and Social Research Council (grant reference ES/K007246/1).

References

1. Stodden, V., Leisch, F., Peng, R.D.: Implementing Reproducible Research. CRC Press (2014)
2. Yang, H., Michaelides, D.T., Charlton, C., Browne, W.J., Moreau, L.: DEEP: A Provenance-Aware Executable Document System. In: 4th International Provenance and Annotation Workshop, IPAW 2012, Santa Barbara, CA, USA, June 19-21, 2012. Springer (2012) 24–38
3. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., Bhagat, J., Belhajjame, K., Bacall, F., Hardisty, A., Nieva de la Hidalga, A., Balcazar Vargas, M.P., Sufi, S., Goble, C.: The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic Acids Research **41**(W1) (2013) W557–W561
4. 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. Concurr. Comput. : Pract. Exper. **18**(10) (August 2006) 1039–1065
5. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T.: VisTrails: Visualization Meets Data Management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. SIGMOD '06, New York, NY, USA, ACM (2006) 745–747

6. McPhillips, T., Song, T., Kolisnik, T., Aulenbach, S., Belhajjame, K., Bocinsky, R., Cao, Y., Cheney, J., Chirigati, F., Dey, S., Freire, J., Jones, C., Hanken, J., Kintigh, K.W., Kohler, T.A., Koop, D., Macklin, J.A., Missier, P., Schildhauer, M., Schwalm, C., Wei, Y., Bieda, M., Ludäscher, B.: YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts. International Journal of Digital Curation **10**(1) (2015) 298–313
7. Chirigati, F., Shasha, D., Freire, J.: ReproZip: Using Provenance to Support Computational Reproducibility. In: Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance, Berkeley, CA, USENIX (2013)
8. Moreau, L.: Provenance-Based Reproducibility in the Semantic Web. Web Semantics: Science, Services and Agents on the World Wide Web **9**(2) (2011)
9. Fraser, N.: Blockly: A library for building visual editors. https://developers.google.com/blockly/
10. Missier, P., Dey, S., Belhajjame, K., Cuevas-Vicenttin, V., Ludäscher, B.: D-PROV: Extending the PROV Provenance Model with Workflow Structure. In: 5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13), Lombard, IL, USENIX Association (April 2013)
11. Michaelides, D., Huynh, T.D., Moreau, L.: PROV-Template: A Template System for PROV Documents. https://provenance.ecs.soton.ac.uk/prov-template/
12. Moreau, L., Missier, P.: PROV-DM: The PROV data model. World Wide Web Consortium, Recommendation REC-prov-dm-20130430 (April 2013)
13. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for Computational Tasks: A Survey. Computing in Science & Engineering **10**(3) (2008) 11–21
14. Murta, L., Braganholo, V., Chirigati, F., Koop, D., Freire, J.: noWorkflow: Capturing and Analyzing Provenance of Scripts. In: 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Springer (2015) 71–83
15. Simmhan, Y., Groth, P., Moreau, L.: The Third Provenance Challenge on using the Open Provenance Model for interoperability. Future Generation Computer Systems **27**(6) (2011) 737 – 742
16. Missier, P., Goble, C.: Workflows to open provenance graphs, round-trip. Future Generation Computer Systems **27**(6) (2011) 812 – 819

Appendix

This material to be made available online for final version.
JSON representation of a variables from Block 2 in Fig. 3:


```
{
  "context" : {
    "xsd" : "http://www.w3.org/2001/XMLSchema#",
    "estat" : "http://purl.org/net/statjr/ns#",
    "estatwf" : "http://purl.org/net/statjr/wf#",
    "urn_uuid" : "urn:uuid:"
  },
  "var" : {
    "block_instance" : [ { "@id" : "urn_uuid:2" } ],
    "parent" : [ { "@id" : "urn_uuid:1" } ],
    "starttime" : [ {
      "@value" : "2016-02-12T15:12:28.546712",
      "@type" : "xsd:dateTime"
    } ],
    "endtime" : [ {
      "@value" : "2016-02-12T15:12:28.677037",
      "@type" : "xsd:dateTime"
    } ],
    "block_uri" : [ "pgno3ns6cur7ej7yxhju" ],
    "block_title" : [ "Calculate" ],
    "block_type" : [ { "@id" : "estatwf:Calculate" } ],
    "consumed" : [ { "@id" : "urn_uuid:3" },
                   { "@id" : "urn_uuid:4" },
                   { "@id" : "estat:datasets/tutorial" } ],
    "consumed_name" : [ "column", "expression", "dataset" ],
    "consumed_at" : [ {
      "@value" : "2016-02-12T15:12:28.546846", "@type" : "xsd:dateTime" }, {
      "@value" : "2016-02-12T15:12:28.546846",
      "@type" : "xsd:dateTime"
    }, {
      "@value" : "2016-02-12T15:12:28.546846",
      "@type" : "xsd:dateTime"
    } ],
    "produced" : [ { "@id" : "urn_uuid:5" },
                   { "@id" : "urn_uuid:6" },
                   { "@id" : "urn_uuid:7" } ],
    "produced_at" : [ {
      "@value" : "2016-02-12T15:12:28.676943",
      "@type" : "xsd:dateTime"
    }, {
      "@value" : "2016-02-12T15:12:28.676943",
      "@type" : "xsd:dateTime"
    }, {
      "@value" : "2016-02-12T15:12:28.676943",
      "@type" : "xsd:dateTime"
    } ],
    "produced_name" : [ "a", "inputs", "script.py" ],
    "literal" : [ { "@id" : "urn_uuid:3" },
                  { "@id" : "urn_uuid:4" } ],
    "literal_value" : [ "normexam2",
                        "normexam*normexam" ],
    "literal_type" : [ { "@id" : "xsd:string" },
                       { "@id" : "xsd:string" } ]
  },
  "vargen" : { }
}
```

Provenance graph from Figure 6 in PROV-N representation:

```
document
  prefix var <http://openprovenance.org/var#>
  prefix estat <http://purl.org/net/statjr/ns#>
  prefix estatwf <http://purl.org/net/statjr/wf#>
  prefix urn_uuid <urn:uuid:>

  entity(urn_uuid:10)
  entity(urn_uuid:9)
  entity(estat:datasets/tutorial)
  entity(urn_uuid:4,[estatwf:type = 'xsd:string', estatwf:value = "normexam*normexam" %% xsd:string])
  entity(urn_uuid:5)
  entity(urn_uuid:6)
  entity(urn_uuid:7)
  entity(urn_uuid:11)
  entity(urn_uuid:3,[estatwf:value = "normexam2" %% xsd:string, estatwf:type = 'xsd:string'])
  activity(urn_uuid:8,2016-02-12T15:12:28.679Z,2016-02-12T15:12:29.527Z,
           [prov:type = 'estatwf:DatasetSummary', prov:label = "DatasetSummary",
            estatwf:block = "6fdqrmkq5n8fuq57qfti" %% xsd:string])
  activity(urn_uuid:1,2016-02-12T15:12:28.543Z,2016-02-12T15:12:29.527Z,
           [prov:type = 'estatwf:Sequence', prov:label = "Sequence",
            estatwf:block = "rqvik2xqakayemazt813"])
  activity(urn_uuid:2,2016-02-12T15:12:28.546Z,2016-02-12T15:12:28.677Z,
           [prov:type = 'estatwf:Calculate', prov:label = "Calculate",
            estatwf:block = "pgno3ns6cur7ej7yxhju" %% xsd:string])
  used(urn_uuid:2,urn_uuid:3,2016-02-12T15:12:28.546Z,
      [estat:bindingname = "column" %% xsd:string])
  used(urn_uuid:2,urn_uuid:4,2016-02-12T15:12:28.546Z,
      [estat:bindingname = "expression" %% xsd:string])
  used(urn_uuid:2,estat:datasets/tutorial,2016-02-12T15:12:28.546Z,
      [estat:bindingname = "dataset" %% xsd:string])
  used(urn_uuid:8,urn_uuid:5,2016-02-12T15:12:28.679Z,
      [estat:bindingname = "dataset" %% xsd:string])
  wasGeneratedBy(urn_uuid:5,urn_uuid:2,2016-02-12T15:12:28.676Z,
      [estat:bindingname = "a" %% xsd:string])
  wasGeneratedBy(urn_uuid:6,urn_uuid:2,2016-02-12T15:12:28.676Z,
      [estat:bindingname = "inputs" %% xsd:string])
  wasGeneratedBy(urn_uuid:7,urn_uuid:2,2016-02-12T15:12:28.676Z,
      [estat:bindingname = "script.py" %% xsd:string])
  wasGeneratedBy(urn_uuid:9,urn_uuid:8,2016-02-12T15:12:29.527Z,
      [estat:bindingname = "inputs" %% xsd:string])
  wasGeneratedBy(urn_uuid:10,urn_uuid:8,2016-02-12T15:12:29.527Z,
      [estat:bindingname = "script.py" %% xsd:string])
  wasGeneratedBy(urn_uuid:11,urn_uuid:8,2016-02-12T15:12:29.527Z,
      [estat:bindingname = "table" %% xsd:string])
  wasDerivedFrom(urn_uuid:5, urn_uuid:3)
  wasDerivedFrom(urn_uuid:5, urn_uuid:4)
  wasDerivedFrom(urn_uuid:5, estat:datasets/tutorial)
  wasDerivedFrom(urn_uuid:6, urn_uuid:3)
  wasDerivedFrom(urn_uuid:6, urn_uuid:4)
  wasDerivedFrom(urn_uuid:6, estat:datasets/tutorial)
  wasDerivedFrom(urn_uuid:7, urn_uuid:3)
  wasDerivedFrom(urn_uuid:7, urn_uuid:4)
  wasDerivedFrom(urn_uuid:7, estat:datasets/tutorial)
  wasDerivedFrom(urn_uuid:9, urn_uuid:5)
  wasDerivedFrom(urn_uuid:10, urn_uuid:5)
  wasDerivedFrom(urn_uuid:11, urn_uuid:5)
  wasStartedBy(urn_uuid:2,-,urn_uuid:1,2016-02-12T15:12:28.546Z)
  wasStartedBy(urn_uuid:8,-,urn_uuid:1,2016-02-12T15:12:28.679Z)
endDocument
```