LoonyBin: Keeping Language Technologists Sane through Automated Management of Experimental (Hyper)Workflows

Jonathan H. Clark, Alon Lavie

Language Technologies Institute
Carnegie Mellon University
{jhclark,alavie}@cs.cmu.edu

Abstract

Many contemporary language technology systems are characterized by long pipelines of tools with complex dependencies. Too often, these workflows are implemented by ad hoc scripts; or, worse, tools are run manually, making experiments difficult to reproduce. These practices are difficult to maintain in the face of rapidly evolving workflows while they also fail to expose and record important details about intermediate data. Further complicating these systems are hyperparameters, which often cannot be directly optimized by conventional methods, requiring users to determine which combination of values is best via trial and error. We describe LoonyBin, an open-source tool that addresses these issues by providing: 1) a visual interface for the user to create and modify workflows; 2) a well-defined mechanism for tracking metadata and provenance; 3) a script generator that compiles visual workflows into shell scripts; and 4) a new workflow representation we call a HyperWorkflow, which intuitively and succinctly encodes small experimental variations within a larger workflow.

1. Introduction

Empirical research in natural language processing – and compiling resources to this end – have become complex multi-stage processes. Data preparation alone can require tokenization, text normalization, re-encoding, cleaning of noisy data, etc. Experiments on this data often involve training and tuning of multiple models, testing on a held-out test set, and evaluation of the result – all of which are conducted under multiple experimental conditions (i.e. with different corpora and different sets of hyperparameters).

For example, in syntactic statistical machine translation, a typical experiment consists of over 20 tools with a complex network of dependencies spanning multiple machines or even clusters of machines. Parsing and phrase extraction might be run on a large cluster of hundreds of low-memory machines, preprocessing and word alignment might be run on a local server where software installation is easier, while tuning and decoding might be done on a small cluster of large-memory machines.

The management of such workflows presents a real challenge in terms of keeping results organized, analyzing results at every stage, and automating the workflows. Some find this task so frustrating that they forgo automation altogether, both making experiments difficult to reproduce and wasting CPU cycles when tasks finish while the user is asleep or otherwise away from a terminal. Those who automate their tasks often use ad hoc scripts that are brittle to workflow changes, perform sanity checking in an inconsistent way (if at all), and keep log files in disparate formats. Even special-purpose workflow management systems (see Section 5.) are awkward at best for running experiments under multiple conditions.

LoonyBin was designed to handle medium-scale[1] arbitrary scientific workflows keeping the needs of natural language processing in mind. Specifically, it accommodates workflows that:

- span various machines, clusters, and schedulers
- involve many separate tools, which can be invoked by arbitrary UNIX commands
- have components that are run multiple times under multiple conditions
- evolve quickly with tools frequently being added, removed, and swapped

LoonyBin accomplishes this by providing the following advantages over current common practices:

- associating sanity checks and logging directly with tools, separating these from ad hoc wrappers and automation scripts
- maintaining a cleanly organized directory structure for each step and each condition under which a step is run
- providing a resume-on-failure mechanism for every stage in the pipeline
- making it easy for those without a detailed knowledge of each tool's internals to run the system by providing textual descriptions of each parameter, input file, and output file in a graphical workflow designer
- automatically copying required files between machines or clusters via SSH
- compiling workflows into shell scripts, a medium already in widespread use by NLP researchers

2. Workflow Creation

2.1. Workflow Semantics

We now discuss the representation of workflows in LoonyBin. In their most basic form, LoonyBin represents workflows as Directed Acyclic Graphs (DAGs) as shown in Figure 1. In this form, each vertex represents a TOOL, which produces output files given input files and parameters, and

[1] By medium-scale, we mean workflows having hundreds of vertices. Mega-scale workflows have been identified by Deelman as those having hundreds of thousands of vertices.

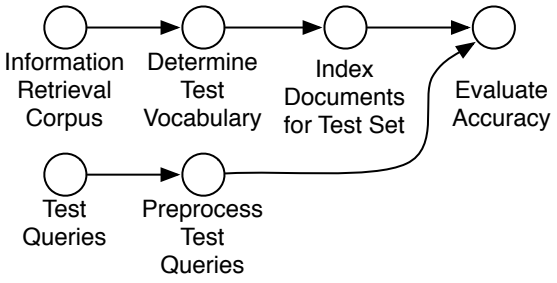

Figure 1: A simple workflow represented as a Directed Acyclic Graph (DAG) as described in Section 2.1.

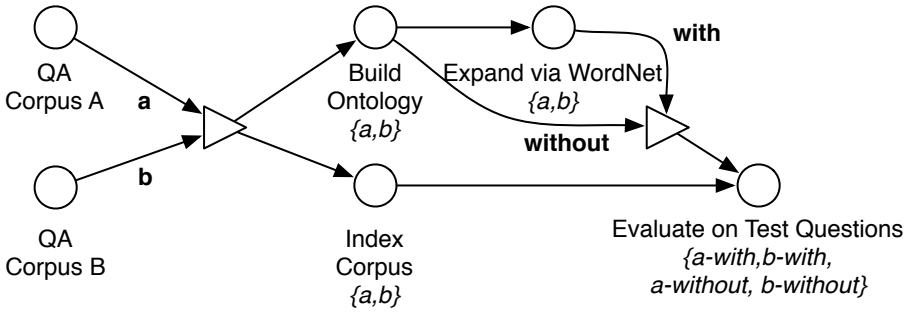

Figure 2: A HyperWorkflow with multiple REALIZATIONS, represented as a Semantic DAG as described in Section 2.2. Hyperedges are visually represented as edges originating from triangular vertices.

directed edges indicate relative temporal ordering of tools and information flow (files or parameters) by mapping the outputs of one tool to the inputs of the next. A TOOL DESCRIPTOR defines the commands necessary to run a tool given inputs, outputs, and parameter. Custom tool descriptors can be implemented via simple user-defined Python scripts that generate shell commands. These tool descriptors contain PRE-ANALYZERS to check the sanity of the inputs and to log information and POST-ANALYZERS to check the sanity of the output files, log information about the outputs, and extract log data from any third-party log file formats.

2.2. HyperWorkflow Semantics

LoonyBin also represents the running of workflows under multiple experimental conditions (i.e. with different input files or parameters). We call this a HYPERWORKFLOW. A HyperWorkflow contains REALIZATION VARIABLES, which introduce variations into a shared workflow. Each realization variable can take on a REALIZATION VALUE, which is a set of files and parameters. For instance the realization variable "language model file and order" could take on the realization value {english.txt, 4}. Finally, a REALIZATION INSTANCE is a regular workflow unpacked from a HyperWorkflow; it is a configuration of a HyperWorkflow such that all realization variables have been assigned a particular realization value. Note that some realization variables might take on a null value if they are irrelevant for a given instance (see Figure 8 in the appendix). HyperWorkflows are useful for performing exploration of hyperparameters, ablation studies, variation of input corpora, and so forth.

To meet this requirement, we use a new data structure based on directed hypergraphs[2]. A hypergraph is a straightforward generalization of a graph in which each endpoint of an edge can connect multiple vertices (Gallo et al., 1993). For HyperWorkflows, we use a HYPERDAG[3], the hypergraph formulation of a DAG, shown in Figure 2. Since this data structure is difficult to draw in a visual interface, we present it to the user by giving shaped vertices special semantics; we call this representation the SEMANTIC DAG format. In LoonyBin, a DIRECTED HYPEREDGE originates as an edge entering a PACKING VERTEX (displayed as a triangle in Figure 2). A packing vertex corresponds exactly to a realization variable. Each hyperedge must have a name and each hyperedge introduces a realization value of the realization variable. A hyperedge has multiple sources when multiple edges with the same name enter the packing vertex, and it has multiple destinations when multiple edges exit the packing vertex. A packing vertex acts like a switch to select one of its realization values. Thus, each unique named edge entering a packing vertex creates a new realization instance in the workflow.

These realization variables are then propagated through the remainder of the workflow. Where multiple realization variables meet, LoonyBin produces the cross product between their realization values. A HyperWorkflow is a packed representation of multiple workflow DAGs, and a

[2]Hypergraphs are already used in NLP both in parsing (Klein and Manning, 2002) and syntactic machine translation (Zollmann and Venugopal, 2006).

[3]In practice, we use a MetaHyperDAG, which generalizes a HyperDAG by allowing meta-edges to take hyperedges as inputs. See the appendix for details.

realization instance is a particular unpacked instance of a workflow. Put another way, a realization instance is an experimental condition (with regard to input files and parameters) under which a workflow is run. For example, in Figure 2 edges *a* and *b* enter a packing vertex and then propagate realizations *a* and *b*. Notice that at the final "Evaluate on Test Questions" vertex, the realizations combine with each other to form a full cross product of experimental conditions.

By representing workflows in this way, we can also exploit the inherent shared substructure in these workflows in a dynamic programming fashion (Huang, 2008). We can both cleanly represent all of the steps required to reproduce the experiments while not rerunning any steps having the same set of experimental conditions.

2.3. Designing

LoonyBin provides a visual editor, which lists all tools (see Figure 4) in browsable tree. Tools can simply be dragged and dropped into the workflow as vertices and edges can be drawn by dragging arrows between these vertices. The only requirement on the design machine is a recent version of Java (Python scripts are executed via Jython). LoonyBin also allows for automatic generation of simple documentation for tools by using documentation strings that are required for every input, output, and parameter as a part of every tool descriptor.

2.4. Deploying

Once a workflow has been designed, LoonyBin can then compile it into an executable shell script. Thus, the only requirement on the machine that executes the workflow is bash. Before any tools are ever executed, the generated script checks that all input files and all directories containing required tools exist. Because LoonyBin handles all filenames other than the initial inputs, this eliminates the common issue of pipelines crashing due to typos in file and directory names. The generated script will log into remote machines, copying files and executing processes as necessary.

LoonyBin also offers the option of executing workflows asynchronously. In this mode, a Java process launches individual bash scripts when dependencies have been satisfied, and a browser-based web portal is provided to monitor workflow progress.

LoonyBin allows each vertex to run on a different machine. A machine can be a target on which to simply run commands, or it can be the head node of a cluster through which one can submit jobs. LoonyBin natively supports schedulers such as Torque, Sun Grid Engine, and Condor. In this respect, LoonyBin can be thought of as a meta-scheduler: it primarily relies on other schedulers to ensure that resources are effectively allocated. LoonyBin also provides integration with the Hadoop Distributed File System.

3. Data, Metadata, and Provenance

While being able to automatically execute and reproduce workflows is good, simply completing the job is not enough. We also want to know how the input files at a given step came to be; this history of the tool's ancestors along with their parameters and inputs is called PROVENANCE – dubbed "the bridge between data and experiments" by Miles (2008). In addition, we may also wish to store METADATA, qualities associated with the data such as how long it took to run a tool and on which machine a tool ran. All the provenance and metadata is stored in plaintext log files in a standard format: tab-delimited key-value pairs and newline-delimited records, making it easy to process these log files using standard command-line tools or scripts. Finally, the log files for all antecedent steps for the same realization are concatenated together at each vertex so that logs for each realization can be processed via a single file.

Since the user might want to run further analysis later, it is important to be able to easily find the data itself. To accommodate this, LoonyBin maintains a highly organized directory structure for each workflow. Under a master directory, LoonyBin creates a directory with the name of each vertex in the HyperWorkflow with child subdirectories for each realization.

4. Real-World Usage

The first real-world tests of LoonyBin has been in the CMU StatXfer machine translation system for GALE Phase 4 (see Figure 3) and the 2010 Workshop on Machine Translation. Aside from the experimental benefits provided by HyperWorkflows, the sanity-checking and logging capabilities turned out to be some of the most useful features of LoonyBin for this task. For instance, the word alignment program GIZA++ is notorious for creating blank output files, crashing, and then returning a successful error code. Our GIZA++ tool descriptor has a post-analyzer that detects failures and logs the Alignment Error Rate. During corpus preprocessing, we log the number of types, tokens, etc. for the entire parallel corpus *after each processing step*. This has proven useful both for understanding how each tool affects the data and for comparing different systems at later dates.

5. Related Work

In this section, we describe various alternative methods of implementing workflows (see Table 1 for a summary). Perhaps the most common tool for implementing workflows are scripts. Perhaps the most common tool for implementing workflows are scripts, which can be hard to maintain as new steps and tools inevitably make their way into a workflow. Others have used build utilities such as GNU Make since this more explicitly models the dependencies between steps. However, as a build tool, it is an awkward fit for experiment management, since it does not have the notion of HyperWorkflows and keeping experimental conditions separate via Make variables can be error-prone.

Workflow Management Systems (WMS) have been a focus of research in the e-Science community for many years, though they have remained largely unnoticed by the NLP community. The DAGMan (DAGMan Team, 2009) project grew out of the Condor scheduler and allows the user to describe dependencies between Condor jobs in the form of a DAG. While certainly a better fit than a build utility, DAGMan does not support handling of metadata or provenance

	Manual Execution	Manual Scripting	Make	DAGMan	Kepler	Pegasus	Dryad	LoonyBin
Supports Hyper-Workflows	Time-consuming	Difficult for large workflows	Not automatic	Not automatic	Not automatic	Not automatic	Not automatic	✓
Reproducible Results	χ	✓	✓	✓	✓	✓	✓	✓
Flexible Pipeline Changes	✓	Usually not	For simple workflows	✓	✓	✓	✓	✓
Resumes on Failure	Manual	Depends on user	With custom configuration	✓	✓	✓	✓	✓
Sanity Checking	Manual examination of data	Usually primitive	Usually primitive	Must be embedded in job	Must be embedded in job	Must be embedded in job	Unknown	Embedded centrally in tools
Records Metadata	χ	Usually not	Usually not	Not automatic	✓	✓	✓	✓
Records Provenance	χ	Usually not	Usually not	Not automatic	✓	✓	Unknown	✓
Progress Monitoring	Console output	Console output	Console output	Condor Logs/ Console	Unknown	In GUI (Experimental)	Yes	Console output and web interface
Multi-Cluster Support	χ	χ	χ	χ	via Condor Flocking	✓	Unknown	✓
Multi-Scheduler Support	χ	χ	χ	χ	via Condor-G	χ	χ	✓
Workflow Definition Syntax	χ	Script language	Proprietary	Proprietary	Visual	Visual / XML	C++	Visual

Table 1: Tools commonly used to implement workflows.

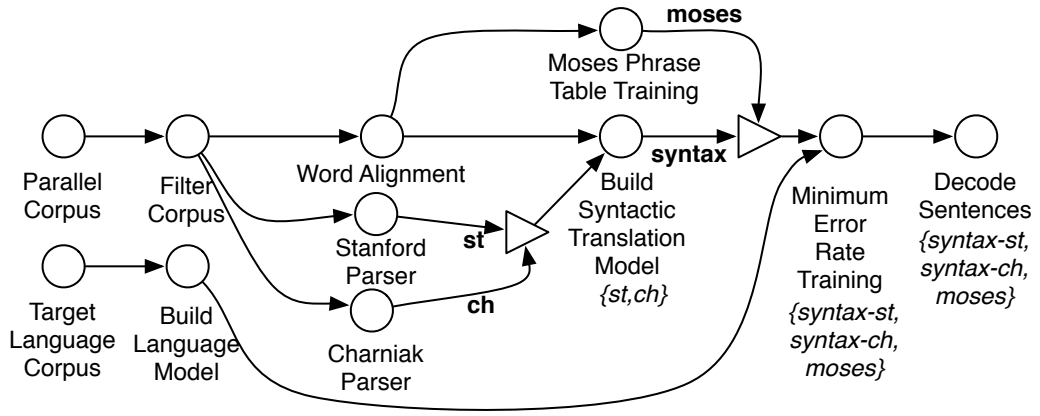

Figure 3: A simplified version of the CMU StatXfer system HyperWorkflow for the GALE Phase 4 Machine Translation Evaluation showing the multiple experiments that were run

on its own. The Pegasus Project (Deelman et al., 2003) aims to solve this by building on top of DAGMan and providing a mechanism for describing workflows in a more abstract way while providing comprehensive support for metadata and provenance management. Similarly, Kepler (Altintas et al., 2004) provides a stand-alone system for designing and running workflows. Dryad (Isard et al., 2007) also allows constructing workflows on Windows HPC clusters using a C++ library. In terms of raw performance, Dryad is a clear winner among all of these systems, including LoonyBin. However, none of these systems allow the user to explicitly encode multiple experiments as Hy-

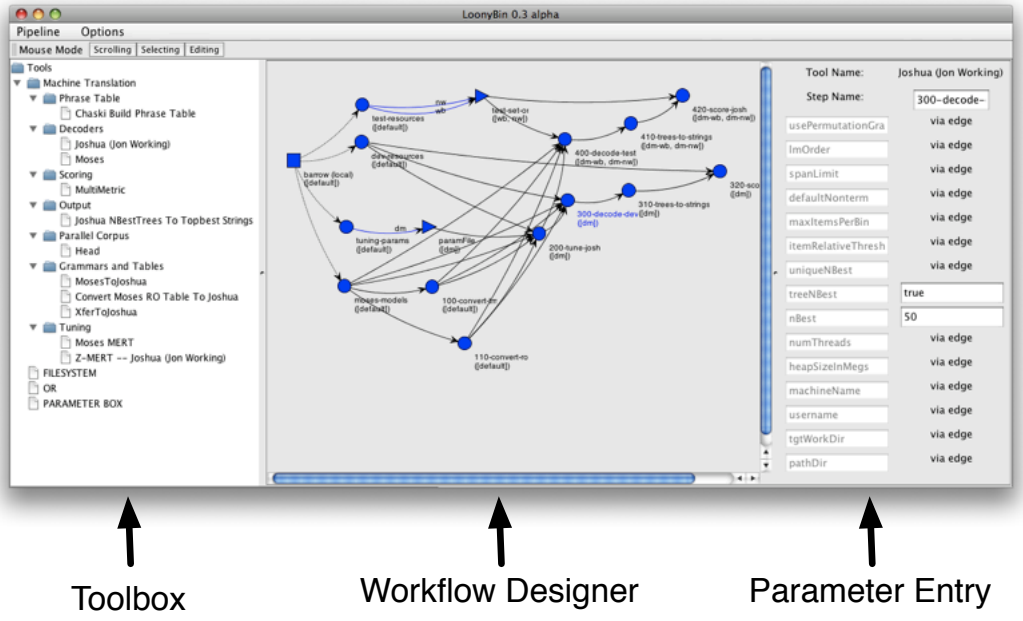

Figure 4: A screenshot of the LoonyBin user interface

perWorkflows without resorting to looping constructs. In contrast, LoonyBin explicitly avoids Turing-completeness to ensure determinism while still allowing the user to easily encode experimental variations.

For a general disussion of terminology used in WMS, we recommend McPhilips (2008). For a more complete discussion of the current state of scientific workflow management systems, we recommend Deelman (2008) or Ludäscher (2009).

6. Future Work

Though LoonyBin attempts to encode reproducible workflows, this is only achievable up to a point. Changes in software versions, environment variables, and hardware resources could cause workflows to fail in new environments. While it is not practical to consider hardware limitations, it is easy to generate workflows such that all environment variables must be captured within the workflow definition rather than externally. However, since this would place a large burden on the user, we have not yet pursued this. On this issue of software versions, we plan to link each tool in a workflow to a particular revision of a source code management system so that the original software versions can be rebuilt if desired. With this addition, we foresee the ability to publish and share LoonyBin workflows online.

Besides HyperWorkflows in which certain portions of a workflow are run multiple times guided by packing vertices, certain portions may also be shared at different points within the same realization. For example, a group of vertices might create a translation model, filter it to a specific set, and then prune it. Currently, one must duplicate these portions of the workflow. Programming languages, in contrast, deal with this cleanly with the notion of functions. Eventually, we would like to integrate this into LoonyBin with the ability to use workflows (but not HyperWorkflows) as vertices.

LoonyBin gives the user the ability to easily execute paths through a HyperWorkflow either exhaustively or by selecting a subset of realizations. Yet exhaustive exploration of these workflow configurations is not ideal with regard to time and resource limitations while the user often uses 1) simple criteria for selecting which path to select next or 2) does not have a good idea of what to select next. This suggests that we could do automatic optimization of hyperparameters. Since a HyperWorkflow already defines a search space over a workflow, we would need only to define an objective function to choose the next realization to explore next.

Another aspect of scientific workflows is analyzing and recording results. While LoonyBin records these in organized log files, we would like to automatically generate charts, graphs, and tables from these. Further, it would be desirable to have a method of archiving experiments from a collection of workflows run over a long period of time within a research group.

Some users are very experienced and comfortable with text editors such as vim and Emacs. Therefore, we also plan to provide a human-editable format for storing workflows so the visual editor can be bypassed.

Currently, LoonyBin generates bash scripts that implement workflows. However, it should also be possible to generate DAGMan files to run on a Condor cluster (although LoonyBin already supports Condor as a scheduler). Another option would be generating Pegasus DAX files to be mapped on to the Grid. Finally, we could consider generating Dryad process wrappers to be mapped on to a Windows HPC cluster by adding a new `WorkflowVisitor` in LoonyBin.

7. Conclusion

We have presented LoonyBin, which we are releasing[4] as an open-source tool for managing workflows in language technology systems. In our own experiences with the StatXfer MT system, we have found the tool to be extremely useful. To encourage adoption, we release the source under the nonrestrictive LGPL license and provide quickstart video screencasts as tutorials. We hope that by providing this tool to the community, experiments will become more reproducible and researchers become more productive.

8. Acknowledgements

Thank you to Kenneth Heafield, Alok Parlikar, and Nathan Schneider for their many insights in writing this paper. Thanks to Michael Denkowski and Greg Hanneman for patiently helping to work the bugs out of early versions of LoonyBin. We are grateful to Ondřej Bojar, Philipp Koehn, and Lane Schwartz for sharing their experiences in writing experiment management systems for their machine translation workflows. Thanks to Yahoo! for providing access to the M45 research cluster, which we used to test the Hadoop/HDFS integration of LoonyBin. Finally, thank you to the anonymous reviewers for their useful comments.

9. Appendix: MetaHyperDAGs

In this section, we sketch the equivalence of MetaHyperDAGs and the semantic DAGs presented in Section 2.2. We also present a complex example that highlights a few important boundary cases and shows the power of this structure. This information is not intended for casual users and will likely be of interest only to users who wish to know the boundaries of LoonyBin's expressive power or those wishing to reimplement the core algorithms.

While semantic DAGs provide a convenient way of incrementally creating hyperedges, they are not so convenient when unpacking HyperWorkflows into their unpacked counterparts. Therefore, it is easier to think about visiting a HyperWorkflow in terms of a MetaHyperDAG in which each hyperedge (or meta-edge) represents an atomic decision. We must generalize a HyperDAGs to a METAHYPERDAG by allowing meta-edges, which may take multiple hyperedges as their input[5], so that a traversal of the HyperWorkflow visits each vertex the minimum number of times. Consider Figure 5. If we were to model the necessarily joint decision of which realization values to select for z as hyperedges alone, we arrive at hyperedges that conflate the decision of choosing the value of realization variables A, B, and C. This would produce two hyperedges, one for *a1-b1-c1* and one for *a1-b1-c2*. Both of these hyperedges would have y as a destination, causing y to get visited twice. Yet y should be visited only once. By introducing the large bold meta-edges in Figure 6, we can atomically choose a combination of realization values for z while not causing spurious ambiguity for y. We insert a meta-edge whenever the parents of a vertex are influenced by differing sets of realization variables. In this way, we can construct an equivalent MetaHyperDAG for a semantic DAG. The remaining figures show the unpacking process, starting at a semantic DAG and ending with separate unpacked DAGs.

[4] LoonyBin is available for download at http://www.cs.cmu.edu/~jhclark/loonybin

[5] Alternatively, we could have inserted no-op vertices at the source of the meta-edges, making the meta-edges normal edges.

10. References

Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. 2004. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management*.

DAGMan Team. 2009. Dagman documentation, http://www.cs.wisc.edu/condor/dagman/.

Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. 2003. Mapping abstract complex workflows onto grid environments. In *Journal of Grid Computing*.

Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. 2008. Workflows and e-science: An overview of workflow system features and capabilities. In *Future Generation Computer Systems*.

Giorgio Gallo, Giustino Longo, Sang Nguyen, and Stefano Pallottino. 1993. Directed hypergraphs and applications. In *Discrete Applied Mathematics*.

Liang Huang. 2008. Advanced dynamic programming in semiring and hypergraph frameworks. In *Coling 2008 Tutorial Notes*.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM.

Dan Klein and Chris Manning. 2002. Parsing and hypergraphs. In Bunt, Carroll, and Satta, editors, *New Developments in Parsing Technology*. Kluwer Academic Publishers.

Bertram Ludäscher, Ilkay Altintas, Shawn Bowers, Julian Cummings, Terence Critchlow, Ewa Deelman, David De Roure, Juliana Freire, Carole Goble, Matthew Jones, Scott Klasky, Timothy McPhillips, Norbert Podhorszki, Claudio Silva, Ian Taylor, and Mladen Vouk. 2009. Scientific process automation and workflow management. In *Scientic Data Management: Challenges, Existing Technology, and Deployment, Computational Science Series*. Chapman & Hall.

Timothy McPhillips, Shawn Bowers, Daniel Zinn, and Bertram Ludaescher. 2008. Scientific workflow design for mere mortals. In *Future Generation Computer Systems*.

Simon Miles, Paul Groth, Ewa Deelman, Karan Vahi, Gaurang Mehta, and Luc Moreau. 2008. Provenance: The bridge between experiments and data. In *Computing in Science and Engineering*.

Andreas Zollmann and Ashish Venugopal. 2006. Syntax augmented machine translation via chart parsing. In *Workshop on Machine Translation (WMT) at the Association for Computational Linguistics (ACL)*.

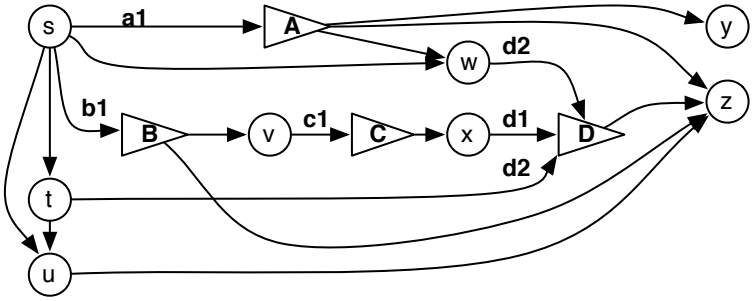

Figure 5: A complex semantic DAG. The packing vertices A and B are for illustrating hyperedges; usually, there is little point in having a packing vertex with a single input edge. There is a 1-1 mapping between a LoonyBin Semantic DAG and a corresponding MetaHyperDAG.

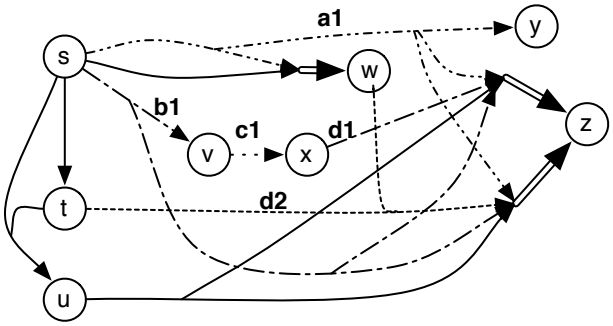

Figure 6: The corresponding MetaHyperDAG. Notice that packing vertices have been transformed into hyperedges and meta-edges. See that the two *d2* edges entering C created a hyperedge with multiple sources while the multiple edges leaving A creates a hyperedge with multiple destinations. When there are multiple packing vertices as the direct parents of a vertex, a cross product is implied as shown at z. Finally, whenever the parents of a vertex are influenced by differing sets of realization variables, we create meta-edges shown in bold as seen at w and z

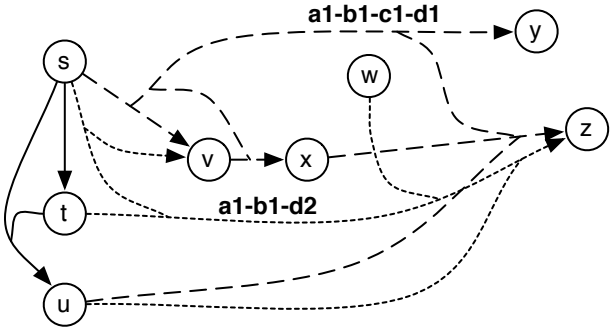

Figure 7: A broken HyperDAG-only conception of the above semantic DAG. Notice that to satisfy the constraint that each decision be atomic, composite hyperedges that jointly decide at least A, B, and C at the same time were introduced. However, this causes v to have an two hyperedges (one for *d1* and one for *d2*), which include realization variables that do not even affect v. This violates the constraint that each vertex be visited a minimal number of times.

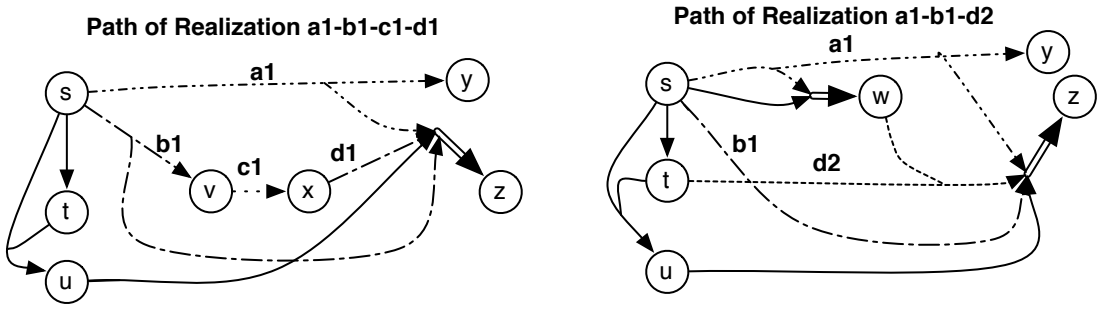

Figure 8: The corresponding paths through this MetaHyperDAG. Note that the realization variable C becomes irrelevant for realization value $d2$ so that we no longer need to pick a value for it when exploring that path. In this form, it is natural to have multiple edges with the same source and destination vertices since they might be linking different files or parameters.

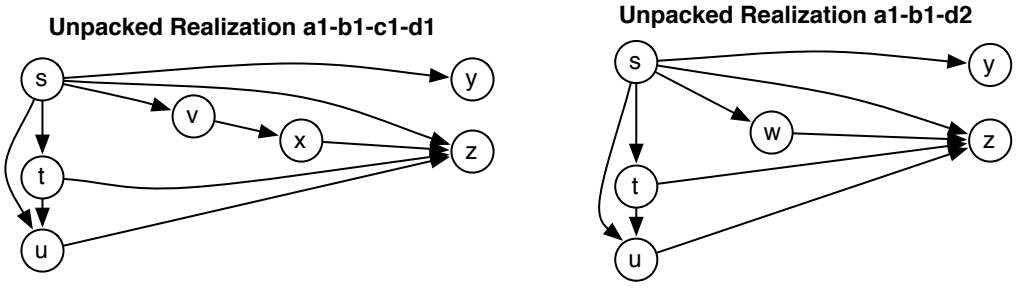

Figure 9: The corresponding unpacked DAGs, containing no hyperedges or meta-edges.