Managing and Documenting Legacy Scientific Workflows

Ruben Acuña[1], Jacques Chomilier[2,3] and Zoé Lacroix[1,2*]

[1]Scientific Data Management Laboratory, Arizona State University, Tempe, AZ 85287, USA, http://bioinformatics.engineering.asu.edu/

[2]Institut de Minéralogie, de Physique des Milieux Condensés et de Cosmochimie (IMPMC), Centre National de la Recherche Scientifique (CNRS), Institut de Recherche pour le Développement (IRD), Muséum National d'Histoire Naturelle (MNHN), Université Pierre et Marie Curie, Sorbonne Universités, 4 Place Jussieu, Paris, France

[3]Ressource Parisienne de Bioinformatique Structurale (RPBS), Université Paris Diderot, 35 Rue Hélène Brion, Paris, France

Summary

Scientific legacy workflows are often developed over many years, poorly documented and implemented with scripting languages. In the context of our cross-disciplinary projects we face the problem of maintaining such scientific workflows. This paper presents the Workflow Instrumentation for Structure Extraction (WISE) method used to process several ad-hoc legacy workflows written in Python and automatically produce their workflow structural skeleton. Unlike many existing methods, WISE does not assume input workflows to be preprocessed in a known workflow formalism. It is also able to identify and analyze calls to external tools. We present the method and report its results on several scientific workflows.

1 Introduction

Scientific discovery is supported by countless scientific workflows describing wet or digital data processes. Once only recorded in laboratory notebooks, workflows are too often only kept as the program used to implement them. Yet these workflows overgo multiple revisions and each revision may not only be poorly documented but erase the previous state of the workflow program, breaking continuity with existing data. Datasets collected with these various versions of the workflow are difficult to integrate and maintain. The need for technology to record scientific workflows and support their life cycle from their design, implementation, execution, and reuse remains poorly addressed by current systems. The Workflow Instrumentation for Structure Extraction (WISE) method can process an ad-hoc legacy workflow written in Python and automatically produce its structural skeleton which can be used for semantic mapping and indexing.

*To whom correspondence should be addressed. Email: zoe.lacroix@asu.edu

In the context of our collaboration with the RPBS team hosted at the Université Denis Diderot in Paris, France[1] we have addressed many issues related to resource reuse and workflow transformations. We designed a method to map resources to a domain ontology to support their retrieval through conceptual queries [1, 2] as well in ProtocolDB [3, 4] that supports the documentation of workflows in conceptual terms and the specific resources used to implement each of their steps. However, without the knowledge of the resources invoked in the workflow and the way they are orchestrated, legacy ad-hoc workflows could only be mapped as a whole resource, providing their overall purpose is even documented, limiting the ability to record and support their maintenance. This is the motivation for designing the Workflow Instrumentation for Structure Extraction (WISE) method. Once processed with WISE the workflow can be mapped to a conceptual workflow representation expressed in terms of a domain ontology as in ProtocolDB. This mapping exploits a *semantic map* of bioinformatics resources [5, 2].

The paper is organized as follows. Related work is discussed in Section 2. The method is introduced in Section 3. Section 4 presents the implementation and evaluation. The results are discussed in Section 5. Performance is addressed in Section 6 followed by future work and conclusion in Section 7.

2 Related Work

Nowadays scientists have the ability to design and implement workflows with a Workflow Management System (WFMS) [6, 7, 8], store them [9, 10], deploy them on the grid [11, 12] or the cloud [13] and visualize them in various manners [14, 15, 16]. But workflow authors may instead take a less structured approach with a workflow programming language or a general programming language with a workflow framework [17, 18, 19]. Many scientific workflows are not implemented with a WFMS, or even a workflow framework [20]. Yet current solutions do not address the problem of understanding legacy ad-hoc workflows such as the Structural Prediction for pRotein fOlding UTility System (SPROUTS), one of the workflows developed with our collaborators [21]. First developed as a set of independent scripts, SPROUTS was later revised into a Python workflow and underwent a series of transformations [22].

Previous work on analyzing and understanding dataflow in dynamic languages using compiling-time techniques has encountered issues stemming from dynamic features (e.g., [23, 24]). Despite using formalisms to model dynamics (e.g., [25]), the execution of such programs has so few constraints that their behavior cannot be predicted. Provenance tracking in ad-hoc workflows has had success in applying run-time techniques (e.g., [26]) to record data but does not address workflow understanding.

Provenance can also be leveraged to support data validation, reuse and integration. WFMSs such as Taverna [8] capture data provenance, but do not focus on data reuse. Before, during, and after, run time, Chiron [27] stores provenance information based on the flow of relations between workflow activities. Chiron supports data reuse and monitors the run time state of executions to identify deviations. VisTrails [16] maintains provenance for visualization results by

[1] See http://bioserv.rpbs.univ-paris-diderot.fr/.

storing the pipeline process which created it. During development, data exploration provides core insight. This can be enhanced by providing better tools for iterative development (e.g., parameter sweep); a user may interact with a tree representing different cumulative changes [28]. The Ediflow platform [15] enables the convergence of visual analytics and workflows in creating workflow visualization processes by integration of a persistent Database Management System (DBMS). A second focus is on providing a live interface between a DBMS and a visualization system to enable change propagation [15].

The analysis of traces of workflow execution is also studied. For instance, Bao et al. [29] give a method for differencing executions to understand control flow in provenance. They define the differencing problem as the computing of a list of transformation between two traces which adhere to a workflow specification. Workflows are specified in so-called series-parallel (sp) workflow format, which is comprised of a sp-graph [30], a model for program controls and dataflows, annotated with information about looping and forking. This was implemented in PDiffView [31], a graphical application which imports workflows into a sp-workflow format, generates valid runs, and then shows the operations in differencing them. PDiffView differs from our work as it focuses on comparing structure between executions, not extracting executions or comparing elements within an execution to identify the overall structure.

Provenance tracking has been addressed for Python based systems in several ways. One way to track provenance is by enabling the user to define explicitly the dataflow using a provenance application programming interface (API) [32]. This approach is intrusive because the workflow must be engineered to use the API. IncPy [33] is a non-intrusive and low level approach which replaces the standard Python interpreter with one designed to catch automatically the result of functions as they are called. Starflow [34] provides a data analysis environment at the level of a Python's interactive interpreter. Using a combination of static analysis, dynamic analysis, and user annotations, Starflow builds a dependency graph of functions in terms of the files and folders they use. Based on the dependency graph, Starflow detects changes in functions or input and so determines what functions must be reexecuted. ProvenanceCurious [35] provides a method to extract provenance from a Python program for debugging. From a Python file, ProvenanceCurious constructs a provenance graph from the syntax of the program while interacting with the user to annotate elements with information on file access. The provenance graph, similar to a program dependency graph, is refined using a number of graph rewriting rules to propagate properties between nodes, thus making some redundant and so removable. Once a provenance graph has been extracted, ProvenanceCurious supports analyzing, and querying, the dataflow of that program's execution. noWorkflow [26] provides a non-intrusive and systematic way to collect provenance information in a general Python program. Like Starflow, noWorkflow is file centric, but, unlike Starflow, it is based on programs instead of interactive development. During workflow execution, functions are tracked if they are user created or if they are part of the standard library and involve accessing a file. The result is source code indexed for functions, function parameters, data files; all of which are stored in a local database. noWorkflow then provides analysis functionality with this provenance database: graph analysis, difference analysis, and query analysis. noWorkflow captures version information for external modules as well as environment variables; these help to fully define the execution environment. Although these provenance methods track data in a workflow, and factors leading to their computation, they do not support data in external tools to produce the full dataflow structure, or, reducing the

complexity of the dataflow.

The WISE method presented and discussed in this paper captures the dataflow of an ad-hoc workflow (i.e., a provenance graph) through its *execution* and *abstracts* it with a process for simplifying repetitive regions. An *instrumentation* engine processes the workflow to produce an instrumented workflow, which is then executed on an input. The log produced by the execution then undergoes *trace analysis* to produce a file-based provenance graph. The dataflow of the graph is examined for equivalent regions, which are collected into a singular unit. The workflow is thus characterized in terms of its treatment of the sample input. This representation provides a data-centric view of the ad-hoc workflow's structure, demonstrating its essential behavior, and which may be used for other purposes such as rewriting or transformation.

3 Method

The overall method is summarized in Figure 1. Legacy workflows are first processed automatically with WISE [36]. The tool, user manual and related material, as they become available, are published on the project Web site[2]. The output of WISE is a workflow implementation with an explicit structure that can be uploaded in the ProtocolDB database. Each tool that is used in the workflow implementation is registered in the semantic map where it is mapped to its corresponding concepts. That will allow the mapping of the implementation workflow to a conceptual workflow expressed in the terms of a domain ontology. The conceptual workflow is stored in the design component of the ProtocolDB database. In ProtocolDB[3], a workflow becomes *data* which can be queried and searched.

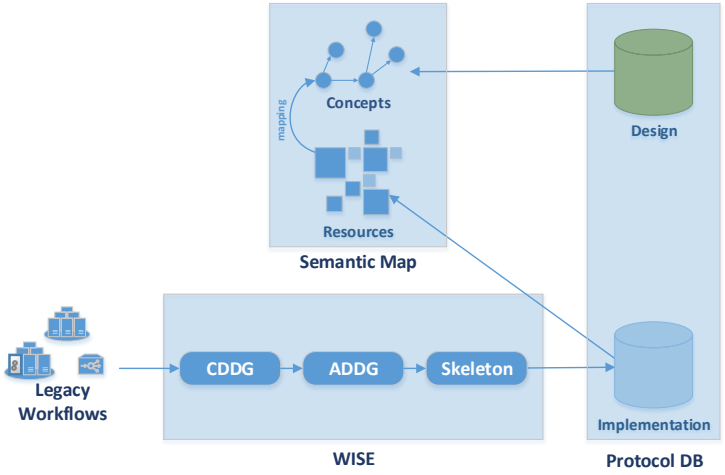


Figure 1: Overall method.

We illustrate the overall WISE method in Figure 2. The first step of the WISE method is to produce an *instrumentation* of the workflow. Workflow instrumentation is the process of adding

[2] http://bioinformatics.engineering.asu.edu/WISE/

[3] At publication of the paper, ProtocolDB is no longer available and is going under massive revision. Future release will be made available at http://bioinformatics.engineering.asu.edu/ProtocolDB/.

elements (i.e., code instructions) to monitor and record behavior during workflow execution. The instrumented workflow is an *equivalent workflow* which produces a log at run time, containing information on its interactions with the file system. The instrumentation is transparent to the execution in the sense that it does not affect any part of the dataflow and only monitors the relevant calls with a wrapper which intercepts functions when they are executed by the workflow's flow control, records their parameters, and returns control to the standard library. Such an instrumentation produces a description of the workflow structure in terms of calls to tools, algorithms and methods, for a given execution. A workflow instrumentation thus provides a layer between the workflow and the language libraries. This first automated step is the only one that depends on the programming language (here Python).

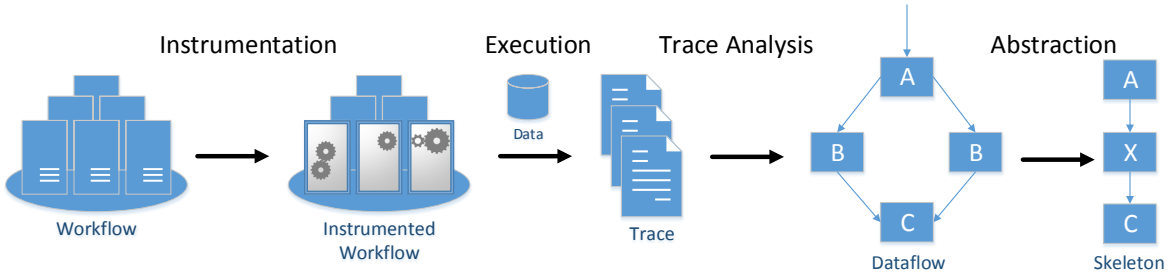


Figure 2: Workflow Instrumentation for Structure Extraction (WISE) method.

The second step is to analyze a *trace* produced by the execution of the instrumented workflow to construct a *dataflow graph* as well as a *library of application resources*. The *nodes* of the dataflow graph represent events recorded in the trace and the *edges* correspond to file dependencies. The library is the list of tools identified in the trace and usage profiles which can be checked for equivalence and define a list of ports. A usage profile also includes information on which program is to be executed and the manner in which its system command may be generated. This additional information is only needed when a tool command is to be actually executed. When events such as a system call, or accessing a file, occur when executing the instrument workflow, they are recorded within the file system state. The execution of the instrument workflow produces an event log which can be analyzed to determine data dependencies. The dataflow graph is created by analyzing file system changes in the context of the commands being executed. This graph represents the file dependencies for events and can provide an initial workflow structure.

Steps one and two were presented in more details in [37]. Tracing by WISE is similar to data provenance methods (e.g., [34, 26]), while dataflow abstraction is similar to methods for reducing workflow complexity (e.g., [38, 39]). The WISE approach differs from the latter method in the sense that it can operate on workflows not limited to those represented with a specific workflow format such as sp-graph models. Moreover, WISE performs without any knowledge of the workflow specification.

The third step is an *equivalence analysis* of the concrete data dependency graph using the resource library (not shown in Figure 2). Recall that an usage profile contains information on the inputs, outputs, and parameters, used by a program with a specific hash. They represent a cer-

tain tool being executed in a specific manner. Invocations using the same profile are guaranteed to be identical. Thus, profiles define a basic level of equivalence. However, profile equivalence is not a necessary condition for two commands to be semantically equivalent. Two commands may be semantically equivalent when viewed at a level of abstraction above execution. Equivalent transformation consists in identifying and merging all similar events into a more abstract data dependency graph. The abstraction process focuses on identifying repetitive regions and replacing them with simpler graphical elements that designate the repetition. Two nodes may be equivalent taken in isolate but not in the context of the entire workflow. That is, the data they produce may be transformed in different ways before the completion of the execution. This second type of equivalence, called flow equivalence, is also addressed by WISE. This process preserves the exact structure of the execution.

The final step consists in simplifying the overall graph to produce a *skeleton* of the abstract data dependency graph. Skeletonization consists of the removal of parallel edges and all graphical elements denoting repetition. The graph is simplified by rewriting all nodes and profiles to take a set of inputs and have exactly one output. Then, all parallel edges are removed. If a node designates a repetition, then it is removed and a dependency added between each of its predecessors and successors. This preserves the connectedness of the graph. The workflow skeleton can then be semantically indexed by identifying the resources recorded by WISE in the library and use their conceptual representation in a *semantic map* where each resource is registered with its input and output as a labeled edge between its complex datatypes all mapped to a domain ontology [5]. The workflow may then be expressed as an *implementation workflow* in the ProtocolDB formalism mapped to a *conceptual workflow* which captures the aim of the workflow in the terms of a domain ontology [4].

4 Implementation and Evaluation

The trace engine and dataflow reconstruction programs were developed and executed on Python 2.7.6 and Xubuntu 14.04. Performance evaluations were performed on a virtual machine with an Intel 2500K (at 4.5Ghz), 4GB RAM, and a 30GB hard drive. All instrumented workflows and trace analyses were run on CPython 2.7.6. Workflow traces were performed in a virtualized filesystem for reproducibility. Workflows were installed using the instructions specified in their respective readme files.

Our application domain is structural biology and, in particular, protein structure. The Ressource Parisienne en Biologie Structurale (RPBS) [40] is developed by six teams: MTi (Molécules Thérapeutiques in silico), LBSR (Laboratoire de Biologie Structurale et Radiobiologie), DSIMB (Dynamique des Systèmes et Interactions des Macromolécules Biologiques), ABI (Atelier de BioInformatique), CNAMSTIC (Chaire de Bioinformatique), and IMPMC (Protein Structure Prediction). RPBS is also an active contributor to the network MobyleNet supporting the design of bioinformatics workflows from 1,183 available bioinformatics services (as of March 2015). While these resources support the publication of new services and the development of new workflows, they do not address the problem of retrofitting and maintaining legacy work-

flows. In this paper we report on the analysis of the SPROUTS workflow published by RPBS[4] which examines the impact of point mutations on protein stability.

In addition to the analysis on an in-house workflow such as SPROUTS, we evaluate our method on a dataset of workflows developed by other groups. We collected legacy scientific workflows on GitHub[5], a free provider of GIT source code hosting for open source projects. GitHub was searched for workflows with the keywords 'python protein workflow' and 'python protein pipeline'. The term 'python' helps to identify repositories with missing or incorrect language tags by matching readme files or documentation. The search is confined to the domain of protein analysis using 'protein'. Some repositories storing workflows are labeled 'pipeline'. The queries returned several hundred results that we filtered as follows. We first selected workflows that were written in Python, and published with a sample input file; removing hybrid workflows (not fully orchestrated in Python). We considered the workflows that presented some complexity with at least four tools. The complexity was determined by reading the instruction file made available with the workflow and tool dependencies at execution when the workflow failed to install or run them. Finally, we limited the selection to one workflow per author (GitHub user) to avoid potentially similar designs. While not representative of all scientific workflows, the selection process was designed to minimize bias in an effort to assemble a (small) random sample. In this paper, we report on the first four workflows meeting our requirements: hybseqpipeline [41], a sequence assembly workflow for Illumina reads, Inmembrane [42] which checks if a bacterial protein codes for a surface-exposed region, miR-PREFeR [43] which predicts plant microRNA from RNA sequences, and pycoevol [44] which analyzes the coevolution of a pair of proteins.

The characteristics of the workflows are the number of lines of code (LOC), the number of lines of comments in the code (C), the number of tools invoked (T), the sample input size (I) expressed as number of concept (e.g., a protein) instances, and the type of information comprised in the workflow description (D). The characteristics of the test workflows are listed in Table 1. In addition to the four workflows retrieved from GitHub, we included Protein Synthesis, a simple workflow designed to apply the protein synthesis process to a number of sequences, that we used to illustrate our method in the user manual, and SPROUTS. Workflows from GitHub, and SPROUTS, revealed more dynamic language features, and data driven configuration, than their functionality would suggest. The former is partially due to the use of Python that unintentionally invokes dynamic language features [22].

Comments covered from 7% to 22% of the Python code. All of the GitHub workflows included a README file which discussed the overall purpose of the workflow and the tools utilized. Only SPROUTS gave a top-level graph of the interactions between tools, although Pycoevol included a conceptual overview. All of test workflows use external tools to carry out their analysis. This is problematic for existing Python provenance methods (e.g., ProvenanceCurious, StarFlow, noWorkflow) because they lack support for tracking external tools. Indeed, existing methods track direct file access and function calls but neglect the dataflow produced by external tools. In contrast, with WISE all direct file access is tracked in addition to external tools. Tool execution is tracked to determine how the tool interacted with the filesystem and how the

[4] http://sprouts.rpbs.univ-paris-diderot.fr
[5] github.com

Table 1: Characteristics of test workflows. LOC is the total of lines of code, C denotes the number of lines of comments. T is the number of tools identified in the code, I the size of input and D the availability of a workflow description.

Workflow	LOC	C	T	I	D
Protein Synthesis	25	2	3	3	N/A
HybSeqPipeline	307	41	4	44	text
Inmembrane	2341	694	4	1702	text
Pycoevol	3648	502	4	1	graph
miR-PREFeR	2966	340	1+toolkit	3	text
SPROUTS	3438	951	8	1	graph

command issued to run the tool was structured. The six workflows invoked a total of 18 external tools, including: BLAST, CAP3, DFIRE, EXONERATE, FoldX, HMMER3 I-Mutant 2, I-Mutant 3, LipoP MAFFT, MEMSAT3, MIR3, MUpro, Samtools, SignalP, TMHMM, Velvet, and ViennaRNA. Although we recorded all tools listed in workflow documentations and comments for validation, WISE does not require prior knowledge of external tools. WISE discovers the tools, stores them as resources, and assigns them nodes in the skeleton of the abstract data dependency graph automatically for workflow indexation and mapping. Note that the number of tools used by miR-PREFeR is not precised because it uses samtools, a collection containing multiple tools.

Comments range from well documented (e.g., a description of every function is given) in Inmembrane and Pycoevol, to descriptive in miR-PREFeR and HybSeqPipeline, extremely sparse in Protein Synthesis, and incomplete in SPROUTS. SPROUTS was developed by several students over eight years. Its comments were written in two languages (French and English[6]) and were not maintained, therefore they often do not describe the workflow code accurately. Like many in-house legacy workflows, SPROUTS and Protein Synthesis were not meant to be publicly released. This explains that their code was poorly commented and documented. When a workflow is aimed at public release from its early development stage such as those retrieved from GitHub, comments and documentation are likely more accurate and informative. When the documentation includes a graphical representation of the workflow, we use the graph provided by the workflow authors to evaluate and compare with the abstract representation produced by WISE.

The input size (see column I in Table 1) is used to predict the level of parallelism of the workflow execution. The execution of a workflow that may run on a single instance with an input sample containing several instances will likely display significant parallelism. Although some workflows demonstrate parallel dataflow, they are implemented in a sequential manner. WISE is expected to capture such hidden parallelism. The parallelism revealed by an abstracted workflow shows how performance can be improved significantly when the input dataset contains several instances.

The input of Protein Synthesis is a set of sequences in FASTA format. The input of HybSeqPipeline is a set of sequences of different proteins in a single FASTA file. The input of

[6]Some kind of 'Frenglish' was also used.

Inmembrane is one or more bacterial genes (and a parameters file which indicates if it is a gram- or gram+ strain). The input of Pycoevol is a pair of proteins. The input of miR-PREFeR is one or more small RNA-Seq data samples of the same species. The input of SPROUTS is a protein. We run the WISE method on the six workflows using their sample data files: a set of genes for Protein Synthesis, a set of 44 sequences for HybSeqPipeline, a set of 1,702 sequences for Inmembrane, a pair of proteins for Pycoevol, a set of three sequences for miR-PREFeR, and a single protein for SPROUTS. We report on the results of the WISE method in Section 5 and its performance in Section 6.

5 Results

The WISE method produced a complete dataflow graph for Protein Synthesis, HybSeqPipeline, Inmembrane, and SPROUTS. A dataflow graph is complete when it provides unambiguous sources for each intermediate dependency, that is, all dataflow for tool execution is known. The dataflow graph for HybSeqPipeline contains 512 nodes with edges to 3,065 files. HybSeqPipeline is a workflow designed to control the execution of tools and the paths to files that they read or write. Therefore it does not modify the contents of files directly. In contrast, WISE produces only 55 nodes with edges to 3,713 files on Inmembrane although its code has 759% more lines than HybSeqPipeline. All nodes correspond to external bioinformatics tools with the exception of a series of internal nodes reading each result produced by a single internal event which produces an output. (See Section 5.1 for detailed results for Inmembrane.) The results for SPROUTS display a similar scale with 60 nodes and 3,719 edges. Several of the nodes are internal commands, they represent sanity checking of data. Some workflow executions contain unused nodes. Workflows may create tool nodes without file dependencies when they are checking whether a tool is installed.

The results of the WISE method are reported in Table 2. Each instrumented workflow is listed with its size in terms of lines of code (ILOC), exclusive of tools, and the number of raw dataflow graph elements (i.e., nodes and edges) produced by WISE. The abstraction retains all of the tool nodes which were discovered by the dataflow construction and which were found by reading the associated readme files.

Table 2: WISE results on test workflows. ILOC is the number of lines of code of the instrumented workflow. DN and DE are the number of nodes and edges in the concrete dataflow dependency graph, respectively. A is the number of nodes obtained in the abstract dataflow graph and skeleton.

Workflow	ILOC	DN	DE	A
Protein Synthesis	33	11	16	4
HybSeqPipeline	319	512	3065	34
Inmembrane	2421	55	3713	18
SPROUTS	3518	60	3719	31
Pycoevol	3696	3112	2092	N/A
miR-PREFeR	2978	1160	1194	N/A

Pycoevol and miR-PREFeR demonstrate the limits of the method: specific libraries instru-

mented and assumptions about file system use. Pycoevol makes use of web services to download files, however, the library used to access services is not yet tracked by our implementation. miR-PREFeR uses the assigned temporary folder of the computer executing it for processing. Because the folder is outside of the workflow, the dataflow within it is not tracked by the implementation. In both case, the recovered dataflow graph includes annotations about missing data required by specific nodes.

For all workflows but Pycoevol and miR-PREFeR, WISE is able to discover all the tools which were given in the workflow's description. WISE also discovers tools (e.g., data preparation scripts), typically custom made for the workflow, which are not in its description (e.g., SPROUTS: 7 graph generators, 1 output formatter, 1 output validator, and 2 data uploaders).

5.1 Inmembrane

Inmembrane determines whether a bacterial protein sequence may include coding for a surface-exposed region. Inmembrane is documented by a readme file which discusses the workflow's purpose and usage, as well as a published research paper describing the workflow's science and software architecture. The input is processed with a suite of tools: HMMER (a.k.a., nmmsearch) uses probabilistic models called profile hidden Markov models (profile HMMs) [45], SignalP uses neural networks trained on separate sets of prokaryotic and eukaryotic sequences and a hidden Markov model algorithm to identify signal peptides and their cleavage sites [46], LipoP predicts lipoproteins out of signal peptides [47], and TMHMM predicts transmembrane helices in proteins [48]. The results from each tool are used to produce a summary spreadsheet and citations list. Each tool is documented with version information and a web link. Neither the readme file, nor the workflow publication, provide a graphical view of the relation between inputs and data. They do not mention conditions for running tools or how tools use the input.

We apply WISE to the Inmembrane workflow with the input file `AE004092.fasta` and the gram+ option (used in the documentation). WISE produces a graph composed of 55 nodes with edges for 3,713 files. All nodes are produced by tools with the exception of a series of internal nodes reading each result produced by a single internal event which produces an output. An additional two nodes represent a pair of internal events to write the summary and citation list.

The concrete and abstract data dependency graphs are respectively displayed in Figures 3 and 4. (These figures are not aimed to be read in the paper but rather to display the overall graphical characteristics of the graph.) Figure 3 displays a graph with two types of nodes: the circles indicate a command (i.e., executing a tool) whereas a box represents an internal reading or writing of a file. At the top and reading from left to right, the first node represents the workflow run, the second the access to the library, the remaining nodes correspond to `which`, a Linux command which locates an executable. (This particular workflow uses `which` to determine whether each of the four tools it uses is installed.) The second layer of the graph displays 17 tool nodes (that act on a file). Note that the 8^{th} node represent a copy operation (it records the input file to include it as part of the output) and is not a tool. The three disconnected nodes displayed on the right side are, again, external command running `which`. The graph overall displayed seems to indicate that the workflow is parallel. This is the characteristics that the abstraction is aiming at capturing to simplify the workflow data dependency graph. Once

the abstraction algorithm is applied, the simplification is dramatic and all similar 11 nodes - similar because instances of a single tool TMHMM - are combined into one as illustrated in Figure 4. The abstraction step makes it easier to see that a single tool is executed many times, each time combining the same input file with a different file from the library. The other tools occur only once and directly take the input and process it. At this point, the graph still retains more information than is strictly necessary to understand the workflow and map it to a domain ontology. The preliminary skeletonization method is shown in Figure 5. The graph simplifies the abstract data dependency graph for easier reading.

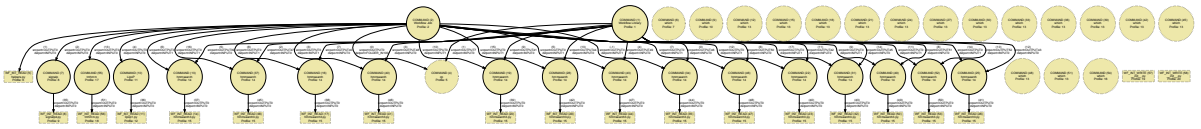


Figure 3: Inmembrane dataflow graph.

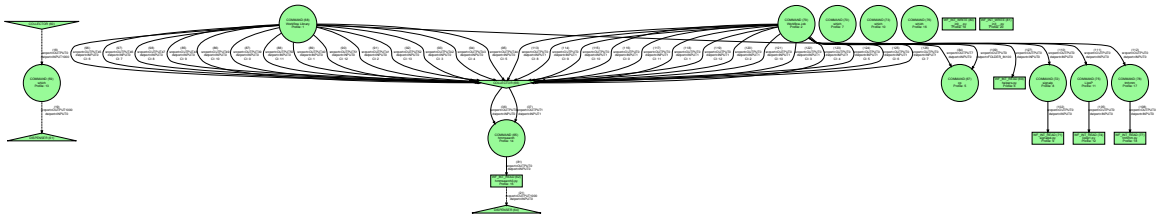


Figure 4: Abstract data dependency graph for Inmembrane.

The skeleton graph indicates the same information as the abstract data dependency graph but is more human readable. If the implicit data dependencies for the creation of the spreadsheet and citation files were added, then edges would occur from nodes 62, 71, 74, and 77, displayed as rectangle nodes labeled by `WF_INT_READ` at the bottom of Figure 5, to both 80 and 81, the two rectangular nodes labeled `WF_INT_WRITE` at the right top of Figure 5. `WF_INT_READ` nodes represent events when the workflow is reading a file whereas nodes `WF_INT_WRITE` denote events when the workflow is saving data in a file. Nodes 62, 71, 74, and 77 are events created when the workflow opened the result produced by each of the four tools. The information contained in these files is then used to produce a summary spreadsheet, saved in event 80, and a finding listing citation information, saved in event in 81.

The final abstraction steps consists of the mapping of all four command nodes labeled by `COMMAND` corresponding to the tools hmmsearch, signalp, LipoP, and tmhmm, respectively. The four tools are on parallel branches of the workflow skeleton and are mapped to the algebraic expression $(hmmsearch \oplus (signalp \oplus (LipoP \oplus tmhmm)))$ using the ProtocolDB formalism and illustrated in Figure 6. In ProtocolDB, workflows are designed top-down. First a a workflow is defined as a workflow task. This task can then be split into two subtasks with either the parallel operator (i.e. $\oplus$) or the sequence operator (i.e. $\otimes$). Here the workflow is first split with the operator $\oplus$ to denote two parallel branches: one for the tool $hmmsearch$ and one for the rest of the workflow. An intermediate mapping of the workflow would be $(hmmsearch \oplus t_1)$, where t_1 denotes an intermediate subtask. The workflow task t_1 can also be split with either $\otimes$, the sequence operator, or $\oplus$, the parallel operator. Here it is split into two parallel branches and

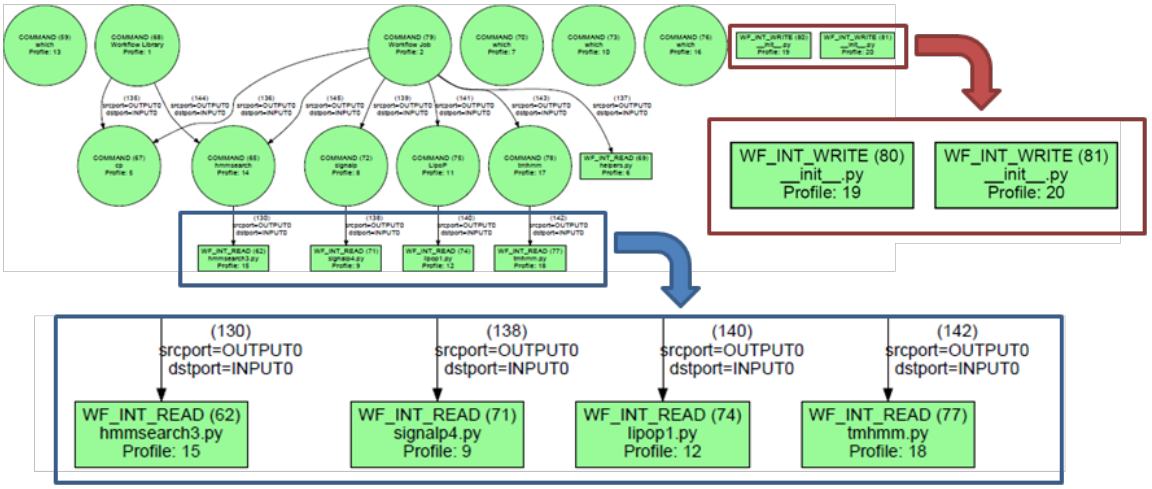


Figure 5: Skeletonized abstract data dependency graph for Inmembrane. The four events labeled by `WF_INT_READ` at the bottom of the figure are magnified in the bottom window whereas the two events labeled by `WF_INT_WRITE` at the top of the graph are shown magnified in the right window.

the workflow can be mapped to the intermediate expression $(hmmsearch \oplus (signalp \oplus t_2))$, and so on until all workflow tasks are bioinformatics tools.

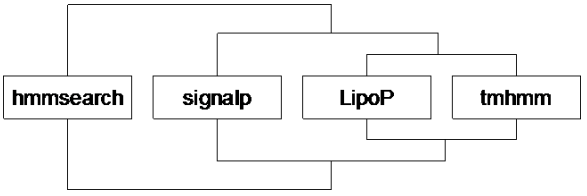


Figure 6: ProtocolDB graph for Inmembrane.

5.2 SPROUTS

SPROUTS is one of the legacy ad-hoc workflows developed with our collaborators. Like many legacy scientific workflows it is partially described in the articles where it is published. SPROUTS performs predictions using a suite of computational tools to examine the impact of point mutations on protein structure. The input to the workflow is either a PDB ID, used to retrieve raw input files, or a raw files themselves. The input is processed with eight different tools: MUPro, DFIRE, I-Mutant (4 versions), FOldX, and MIR. Documentation does not discuss the internal workflow architecture or custom tools. Internally, results from each tool are read to produce both a graph and uploadable SQL file. The SQL file is then uploaded for each tool if the tool generated correct output.

SPROUTS is often illustrated with protein 1ASU (PDB code), however the execution on this input takes several hours. For the purpose of reducing run-time, WISE is run on SPROUTS with the PDB code 1LFC, a peptide (β hairpin) of 25 amino acids. The dataflow graph for this execution, displayed in Figure 7a, seems to indicate that SPROUTS is a parallel workflow with some regions with disconnected nodes. Recall that `WF_INT_READ` nodes represent events

when the workflow is reading a file whereas nodes `WF_INT_WRITE` denote events when the workflow is saving data in a file. In some occurrences, the two events are implicitly related, that is a file is opened, the workflow performs some function and then writes the results in the file. Such a function acts as an *implicit tool*, that is a part the workflow's code which makes up an internal tool. Since the trace depends on the manipulation of files by tools, it does not represent the logic internal to the workflow. Although a workflow may read or write in a file, WISE cannot yet determine the manipulation applied and its dependencies. The regions in a workflow between internal read and write events can be examined to determine if they are independent (e.g., taking only parameters) from the rest of the workflow. The internal events corresponding to such regions can be refactored into a proper tool representation. This can be seen as a problem of program slicing [49] where the portion of a program which some variable depends on must be determined. Here the goal is to determine exactly the part of the workflow orchestrational program which corresponds to an internal tool that manipulates files.

We illustrate the result of processing implicit tools in Figure 7b. Post-processing for implicit tools consists in merging an incoming read event with an outcoming write event on a file. In the dataflow graph, five new implicit tools with both input and output were created by merging the internal commands 10 and 11, 17 and 18/19, 23 and 14, 28 and 30, and 101 and 102. Dependencies were also added between nodes 3 and 7/8/9 to indicate that the job encodes information required to download the initial data files. Had SPROUTS run on local files instead of retrieving a file online with an ID, this step would have been unnecessary. The graph displayed in Figure 7b is shown to illustrate the process. WISE abstraction step does not require such post-processing.

The three nodes labeled `wget` in the middle top of the graph indicate three instances of the tool used to download three separate input files from the ID in the job file. The file resulting from one of these downloads (left most) flows through a two node validation process (152, 153) before reaching MUPro, DFIRE, I-Mutant (4 versions), and FoldX. The file also undergoes one more formatting step (151) before being passed to MIR. The other two files require no validation and are directly used by the tools that need them. One can observe that one tool (MIR) is treated differently than the others, its output is directly read by the workflow (for uploading). The other tools generate output that goes to two other commands, `insert_result.py` and a parser. The data from each parse is then read by a series of internal events, which validate the file and then upload it. One of the tools, DFIRE, failed to run properly during this execution, reason why its output does not reach the command `populateDB.py`.

The WISE method is applied to the dataflow graph displayed in Figure 7a to produce the abstract data dependency graph displayed in Figure 8. The overall logical organization of the workflow is now appearing to the human eye. One can see that the output of each tool is being processed in the same way. The graph includes multiple nodes with the same label, `insert_result.py`. All these nodes refer to the same script, because the parameters given are different in each instance they cannot be combined. The skeleton of the abstract data dependency graph is displayed in Figure 9a. Only the mapping to ProtocolDB formalism (see algebraic expression in Figure 9b) will remove the interactions with the files and database such as nodes labeled with `insert_result.py`. The seven first tools implement the conceptual task *point_mutation_stability_prediction* whereas only MIR implements the conceptual

task *residue_interaction* as shown in Figure 9c. In ProtocolDB, many workflow implementation may be mapped to the same workflow design (i.e., conceptual workflow expressed in the terms of a domain ontology). In the case of SPROUTS all former versions of SPROUTS, and in particular those using older versions of the tools, can be stored in the database and linked to the same expression shown in Figure 9c. ProtocolDB aims at expressing queries such as *retrieve all workflows that perform point_mutation_stability_prediction* which would retrieve all versions of SPROUTS as well as all the stability tools registered in the semantic map (a tool is a workflow composed of a single task). By using navigation in the domain ontology, a conceptual relationship such as *point_mutation_stability_prediction* may be the specialization of a more general conceptual relationship such as *point_mutation_prediction* where other prediction methods of the impact of a mutation on the structure could be mapped. ProtocolDB queries aims at exploiting the structure of the domain ontology captured in the semantic map to answer queries.

6 Performance

Table 3: Performance (in seconds) of the execution of the original workflow (O) vs. its instrumented version (I).

Workflow	O	I
Protein Synthesis	0.071	0.091
HybSeqPipeline	51.204	54.129
Inmembrane	75.578	77.919
SPROUTS	1781.37	1996.371
Pycoevol	906.871	1376.225
miR-PREFeR	25.165	230.332

Table 4: WISE performance (in seconds) recorded for the production of the instrumented workflow (I), its dataflow (D) and its abstraction (A).

Workflow	I	D	A
Protein Synthesis	0.101	0.004	0.549
HybSeqPipeline	0.084	15.870	6.640
Inmembrane	0.064	10.611	2.289
SPROUTS	0.075	15.757	3.743
Pycoevol	0.455	32.400	N/A
miR-PREFeR	0.383	77.015	N/A

The instrumentation of the workflow has an impact on the execution time. The execution times of the original and instrumented versions of the workflows on the inputs described in Table 1 are listed in columns I and D of Table 3 (in seconds). The impact of instrumentation on workflow Protein Synthesis is negligible because there are few tool invocations and they involve small files. For HybSeqPipeline and Inmembrane, the execution of the instrumented workflow is 5.7% and 3.1% more slowely than the initial workflow, respectively. SPROUTS runs 12.1% more slowely. Note that SPROUTS uses an internal timer to determine when it may exit - this obscures the actual runtime. For Pycoevol, the instrumented workflow is 51.8% more slowely

that its non-instrumented version while the instrumented version of miR-PREFeR runs 815.3% more slowely than before instrumentation. The poor performance of instrumented workflows is principally caused by repeated IO access for logs and hashing the filesystem. Workflows that generate many files, or large files, are likely to be the slowest. However, because an instrumented workflow is needed only once to generate logging information, not as a permanent refactoring, these execution times are acceptable.

The only step that is not dependent of the input chosen to produce the dataflow graph is the instrumentation of the workflow. The time required to produce the instrumentation of the workflow is shown in the column I of Table 4. The third column lists the time for the dataflow construction algorithm to execute once the instrumented workflow is executed. The times corresponding to the production of the abstract data dependency graph from the dataflow graph are listed in column A. The abstraction of repetition is quick, indeed the times in column A also include the time required to write DOT files at each stage of processing. For Pycoevol and miR-PREFeR, the dataflow construction is slowed because of missing dataflow information. This causes worse case runtime in some internal procedures. The sum of the times recorded in columns D and A correspond to the total time required to convert the instrumented execution trace on the sample input to an abstract data dependency graph.

7 Conclusions

In this paper we present the WISE method and report on its application on several legacy scientific workflows written in Python. The method is first useful to document legacy workflows and support their semantic indexing with a domain ontology. The WISE method provides a framework to better understand workflows and can be used to identify workflow components that may no longer be used, transform and adapt workflows, compare and store workflows, workflow reasoning [50], optimization [51], reuse [52] and discovery [53].

Future work include addressing current limitations of the method. For example, the dataflow construction step can fail to capture intermingled parallelism (when two tools use the same output folder in parallel) as it was demonstrated by two of the test workflows. Another direction for improvement is to use the trace of several executions of the instrumented workflow on different inputs and merge the dataflow graphs into a common abstract skeleton. Abstract workflow skeleta produced by the WISE method can be mapped to a generic workflow formalism such as ProtocolDB where they can be index and optimized. Such indexing comprises two layers: semantic (in terms of a domain ontology) and tool-based using a semantic map of bioinformatics resources. We expect to automate the indexing of workflows to support their comparison, versioning, retrieval and optimization. The next step will be to translate the optimized workflow obtained by reasoning on the abstract workflow back to a transformation of the concrete datagraph which can be executed. Updates will be posted at http://bioinformatics.engineering.asu.edu/WISE .

Acknowledgements

We thank Rida A Bazzi for contributing to the development of the WISE method and K. Selcuk Candan for his valuable input on workflow analysis. We also thank Matthew G. Johnson for his help with executing HybSeqPipeline. We also acknowledge our RPBS collaborators at the Université Pierre et Marie Curie, Université Denis Diderot, and Institut Pasteur, Paris, France. This research was partially supported by the National Science Foundation[7] (grants IIS 0431174, IIS 0551444, IIS 0612273, IIS 0738906, IIS 0832551, IIS 0944126, and CNS 0849980) and several invitations to Zoé Lacroix as a visiting full professor from the Université Pierre et Marie Curie, Paris, France.

References

[1] P. Tufféry, Z. Lacroix and H. Ménager. Semantic Map of Services for Structural Bioinformatics. In *Proc. 18th IEEE Int. Conf. on Scientific and Statistical database Management*, pages 217–224. 2006.

[2] E. Strauser, M. Naveau, H. Ménager, J. Maupetit, Z. Lacroix and P. Tufféry. Semantic Map for Structural Bioinformatics: enhanced service discovery based on high level concept ontology. In *Resource Discovery*, volume 6799 of *Lecture Notes in Computer Science*, pages 57–70. Springer Berlin Heidelberg, 2012.

[3] M. Kinsy, Z. Lacroix, C. Legendre, P. Wlodarczyk and N. Y. Ayadi. ProtocolDB: Storing Scientific Protocols with a Domain Ontology. In *Web Information Systems Engineering Workshops*, volume 4832 of *Lecture Notes in Computer Science*, pages 17–28. Springer Berlin Heidelberg, 2007.

[4] Z. Lacroix, C. Legendre and S. Mousses. Storing Scientific Workflows in a Database. *Proceedings of the VLDB Endowment*, 2(2):1474–1480, 2009.

[5] Z. Lacroix and M. Aziz. Resource descriptions, ontology, and resource discovery. *Int. J. of Metadata, Semantics and Ontologies*, 5(3):194–207, 2010.

[6] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao and Y. Zhao. Scientific Workflow Management and the KEPLER System. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2005.

[7] C. B. Medeiros, J. Perez-Alcazar, L. Digiampietri, J. G. Z. Pastorello, A. Santanche, R. S. Torres, E. Madeira and E. Bacarin. WOODSS and the Web: annotating and reusing scientific workflows. *ACM SIGMOD Record*, 34(3):18–23, 2005.

[8] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web Server Issue):729–732, 2006.

[7] Any opinion, finding, and conclusion or recommendation expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[9] D. De Roure, S. Bechhofer, C. Goble and D. Newman. Scientific Social Objects: The Social Objects and Multidimensional Network of the myExperiment Website. In *Proc. of the 3rd IEEE Int. Conf. on Social Computing and Privacy, Security, Risk and Trust*, pages 1398–1402. IEEE, 2011.

[10] P. Mates, E. Santos, J. Freire and C. T. Silva. CrowdLabs: Social Analysis and Visualization for the Sciences. In *Proc. of the 23rd Int. Conf. on Scientific and Statistical Database Management*, SSDBM'11, pages 555–564. Springer-Verlag, Berlin, Heidelberg, 2011.

[11] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi and M. Livny. Pegasus: Mapping Scientific Workows onto the Grid. In *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer Berlin Heidelberg, 2004.

[12] I. Taylor, M. Shields, I. Wang and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In I. Taylor, E. Deelman, D. Gannon and M. Shields (editors), *Workflows for e-Science*, pages 320–339. Springer London, 2007.

[13] M. Abouelhoda, S. Issa and M. Ghanem. Tavaxy: Integrating Taverna and Galaxy workflows with cloud computing support. *BMC Bioinformatics*, 13(1):77, 2012.

[14] B. Howe, D. Halperin, F. Ribalet, S. Chitnis and E. Armbrust. Collaborative Science Workflows in SQL. *Computing in Science Engineering*, 15(3):22–31, 2013.

[15] V. Benzaken, J. Fekete, P. Hemery, W. Khemiri and I. Manolescu. EdiFlow: Data-intensive interactive workflows for visual analytics. In *27th IEEE Int. Conf. on Data Engineering*, pages 780–791. 2011.

[16] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva and H. Vo. VisTrails: enabling interactive multiple-view visualizations. In *Proc. of IEEE Visualization Conf.*, pages 135–142. 2005.

[17] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun and M. Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Proc. of the IEEE Congress on Services*, pages 199–206. 2007.

[18] J. Koster and S. Rahmann. Snakemake, a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2510 – 2522, 2012.

[19] R. Filguiera, I. Klampanos, A. Krause, M. David, A. Moreno and M. Atkinson. Dispel4Py: A Python Framework for Data-intensive Scientific Computing. In *Proc. of IEEE Int. Workshop on Data Intensive Scalable Computing Systems*, pages 9–16. 2014.

[20] B. Ludäscher, M. Weske, T. Mcphillips and S. Bowers. Scientific Workflows: Business As Usual? In *Proc. 7th Int. Conf. on Business Process Management*, volume 5701 of *Lecture Notes in Computer Sciences*, pages 31–47. Springer-Verlag, 2009.

[21] M. Lonquety, Z. Lacroix, N. Papandreou and J. Chomilier. SPROUTS: a database for the evaluation of protein stability upon point mutation. *Nucleic Acids Res.*, 37:D374 – D379, 2009.

[22] R. Acuña, Z. Lacroix and J. Chomilier. Refurbishing Legacy Biological Workflows. In *Proc. IEEE 8th World Congress on Services*, pages 41–49. 2012.

[23] M. Dufour. *Shed skin: An optimizing python-to-c++ compiler*. Master's thesis, Delft University of Technology, 2006.

[24] Z. Chen, L. Chen, Y. Zhou, Z. Xu, W. Chu and B. Xu. Dynamic Slicing of Python Programs. In *Proc. 38th IEEE Annual Computer Software and Applications Conf.*, pages 219–228. 2014.

[25] Z. Xu, J. Qian, L. Chen, Z. Chen and B. Xu. Static Slicing for Python First-Class Objects. In *Proc. of the 13rd International Conference on Quality Software*, pages 117–124. 2013.

[26] L. Murta, V. Braganholo, F. Chirigati, D. Koop and J. Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In *5th Int. Provenance and Annotation Workshop*, volume 8628 of *Lecture Notes in Computer Science*, pages 71,83. 2015.

[27] F. Horta, V. Silva, F. Costa, D. de Oliveira, K. Ocaña, E. Ogasawara, J. Dias and M. Mattoso. Provenance Traces from Chiron Parallel Workflow Engine. In *Proc. of the Joint EDBT/ICDT Workshops*, pages 337–338. ACM, 2013.

[28] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva and H. T. Vo. Using provenance to streamline data exploration through visualization. Technical report, SCI Institute, 2006.

[29] Z. Bao, S. Cohen-Boulakia, S. Davidson, A. Eyal and S. Khanna. Differencing Provenance in Scientific Workflows. In *25th IEEE Int. Conf. on Data Engineering*, pages 808–819. 2009.

[30] J. Valdes, R. E. Tarjan and E. L. Lawler. The Recognition of Series Parallel Digraphs. In *Proc. of the 11th Annual ACM Symposium on Theory of Computing*, STOC '79, pages 1–12. ACM, New York, NY, USA, 1979.

[31] Z. Bao, S. C. Boulakia, S. B. Davidson and P. Girard. PDiffView: Viewing the Difference in Provenance of Workflow Results. *PVLDB*, 2(2):1638–1641, 2009.

[32] C. Bochner, R. Gude and A. Schreiber. A Python Library for Provenance Recording and Querying. In *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 229–240. Springer Berlin Heidelberg, 2008.

[33] P. J. Guo and D. Engler. Using automatic persistent memoization to facilitate data analysis scripting. In *Proc. ACM Int. Symp. on Software Testing and Analysis*, pages 287–297. 2011.

[34] E. Angelino, D. Yamins and M. Seltzer. StarFlow: A Script-Centric Data Analysis Environment. In *Provenance and Annotation of Data and Processes*, volume 6378 of *Lecture Notes in Computer Science*, pages 236–250. Springer Berlin Heidelberg, 2010.

[35] M. R. Huq, P. M. G. Apers and A. Wombacher. ProvenanceCurious: A Tool to Infer Data Provenance from Scripts. In *Proc. 16th Int. Conf. on Extending Database Technology*, pages 765–768. ACM, 2013.

[36] R. Acuña. *Understanding Legacy Workflows through Runtime Trace Analysis*. Master's thesis, Arizona State University, 2015.

[37] R. Acuña, Z. Lacroix and R. Bazzi. Instrumentation and Trace Analysis for Ad-hoc Python Workflows in Cloud Environments. In *Proc. IEEE 8th Int. Conf. on Cloud Computing*, pages 114–121. 2015.

[38] P. Sun, Z. Liu, S. Natarajan, S. B. Davidson and Y. Chen. WOLVES: Achieving Correct Provenance Analysis by Detecting and Resolving Unsound Workflow Views. *PVLDB*, 2(2):1614–1617, 2009.

[39] S. Cohen-Boulakia, J. Chen, C. Goble, P. Missier, A. Williams and C. Froidevaux. Distilling Structure in Taverna Scientific Workflows: A refactoring approach. *BMC Bioinformatics*, 15(1):S12, 2014.

[40] C. Alland, F. Moreews, D. Boens et al. RPBS: a web resource for structural bioinformatics. *Nucleic Acids Res.*, 33(Web Server issue):W44W49, 2005.

[41] M. Johnson. HybSeqPipeline: First DOI Release, 2014. URL `http://dx.doi.org/10.5281/zenodo.11977`.

[42] A. Perry and B. Ho. Inmembrane, a bioinformatic workflow for annotation of bacterial cell-surface proteomes. *Source Code for Biology and Medicine*, 8(1):9, 2013.

[43] J. Lei and Y. Sun. miR-PREFeR: an accurate, fast, and easy-to-use plant miRNA prediction tool using small RNA-Seq data. *Bioinformatics*, 30(19):2837–2839, 2014.

[44] F. Madeira and L. Krippahl. Pycoevol - A Python Workflow to Study Protein-protein Coevolution. In *Proc. BIOINFORMATICS*, pages 143–149. SciTePress, 2012.

[45] R. D. Finn, J. Clements and S. R. Eddy. HMMER web server: interactive sequence similarity searching. *Nucleic Acids Research*, 39(suppl 2):W29–W37, 2011.

[46] T. N. N. Petersen, S. Brunak, G. von Heijne and H. Nielsen. SignalP 4.0: discriminating signal peptides from transmembrane regions. *Nature methods*, 8(10):785–786, 2011.

[47] O. Rahman, S. Cummings, D. Harrington and I. C. Sutcliffe. Methods for the bioinformatic identification of bacterial lipoproteins encoded in the genomes of Gram-positive bacteria. *World J. of Microbiol. and Biotech.*, 24(11):2377–2382, 2008.

[48] E. L. L. Sonnhammer, G. v. Heijne and A. Krogh. A Hidden Markov Model for Predicting Transmembrane Helices in Protein Sequences. In *Proc. of the 6th Int. Conf. on Intelligent Systems for Molecular Biology*, pages 175–182. AAAI Press, 1998.

[49] J. Silva. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, 2012.

[50] Z. Lacroix, C. Legendre and S. Tuzmen. Reasoning on Scientific Workflows. In *Proc. of the IEEE World Conference on Services*, pages 306–313. 2009.

[51] Z. Lacroix, M.-E. Vidal and C. Legendre. Customized and Optimized Service Selection with ProtocolDB. In *Proc. of Data Management in Grid and Peer-to-Peer Systems Conference*, volume 5697 of *Lecture Notes in Computer Science*, pages 112–123. 2009.

[52] M. Zohrevandi and R. Bazzi. The Bounded Data Reuse Problem in Scientific Workflows. In *27th IEEE Int. Symp. on Parallel Distributed Processing*, pages 1051–1062. 2013.

[53] J. Starlinger, B. Brancotte, S. Cohen-Boulakia and U. Leser. Similarity Search for Scientific Workflows. *Proc. VLDB Endow.*, 7(12):1143–1154, 2014.

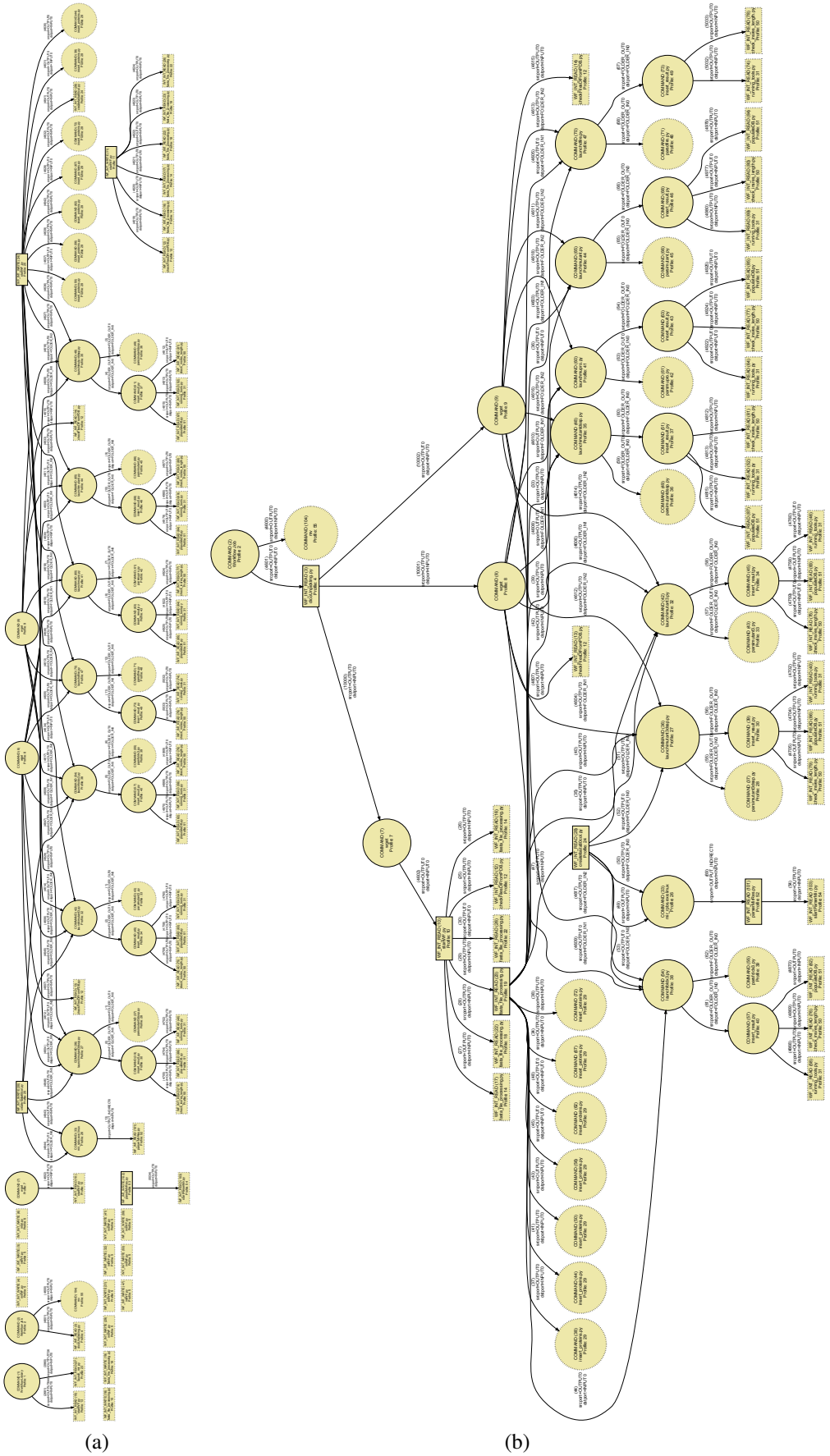


Figure 7: SPROUTS dataflow graph (a) after creating implicit tools (b).

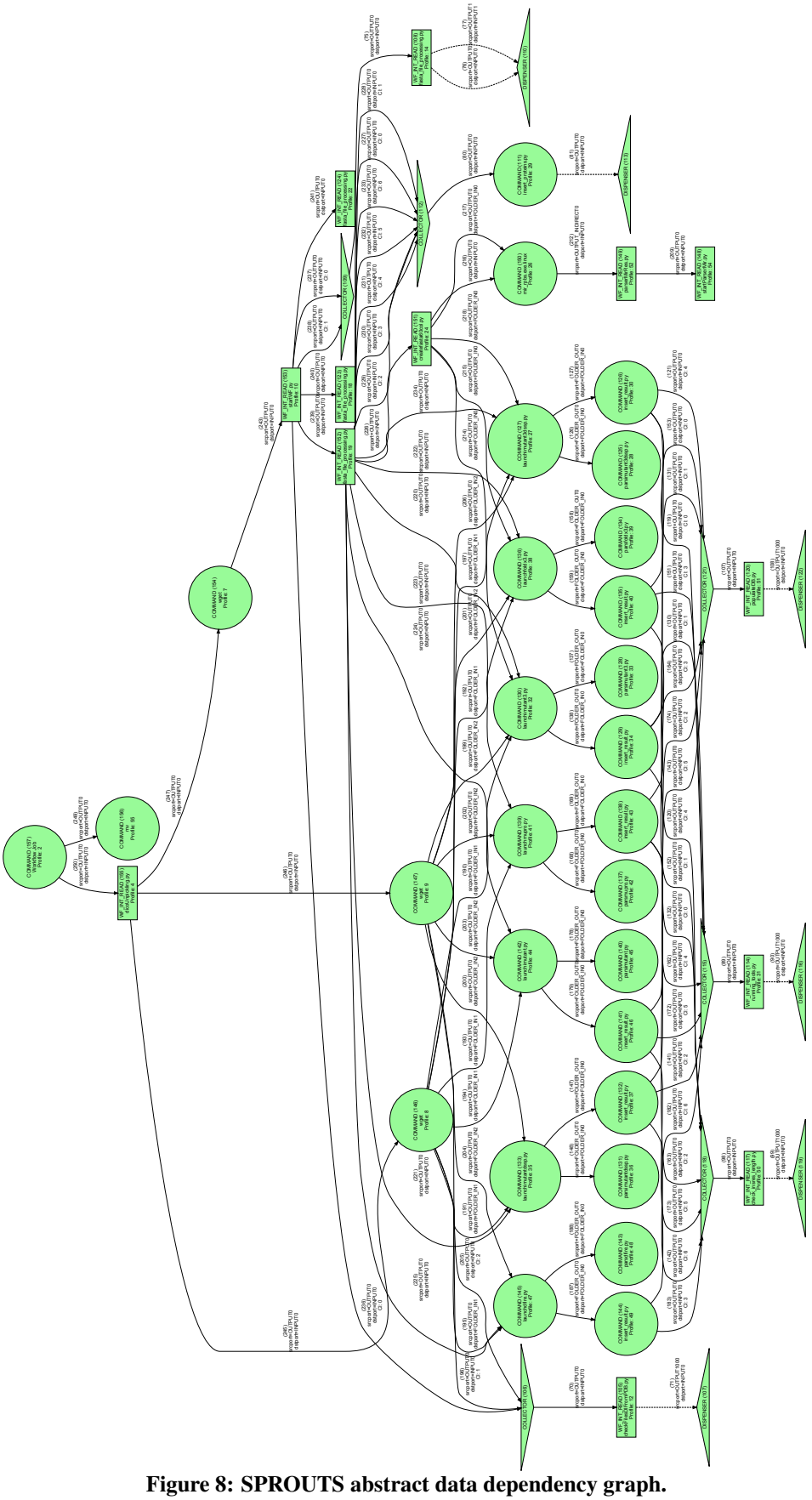
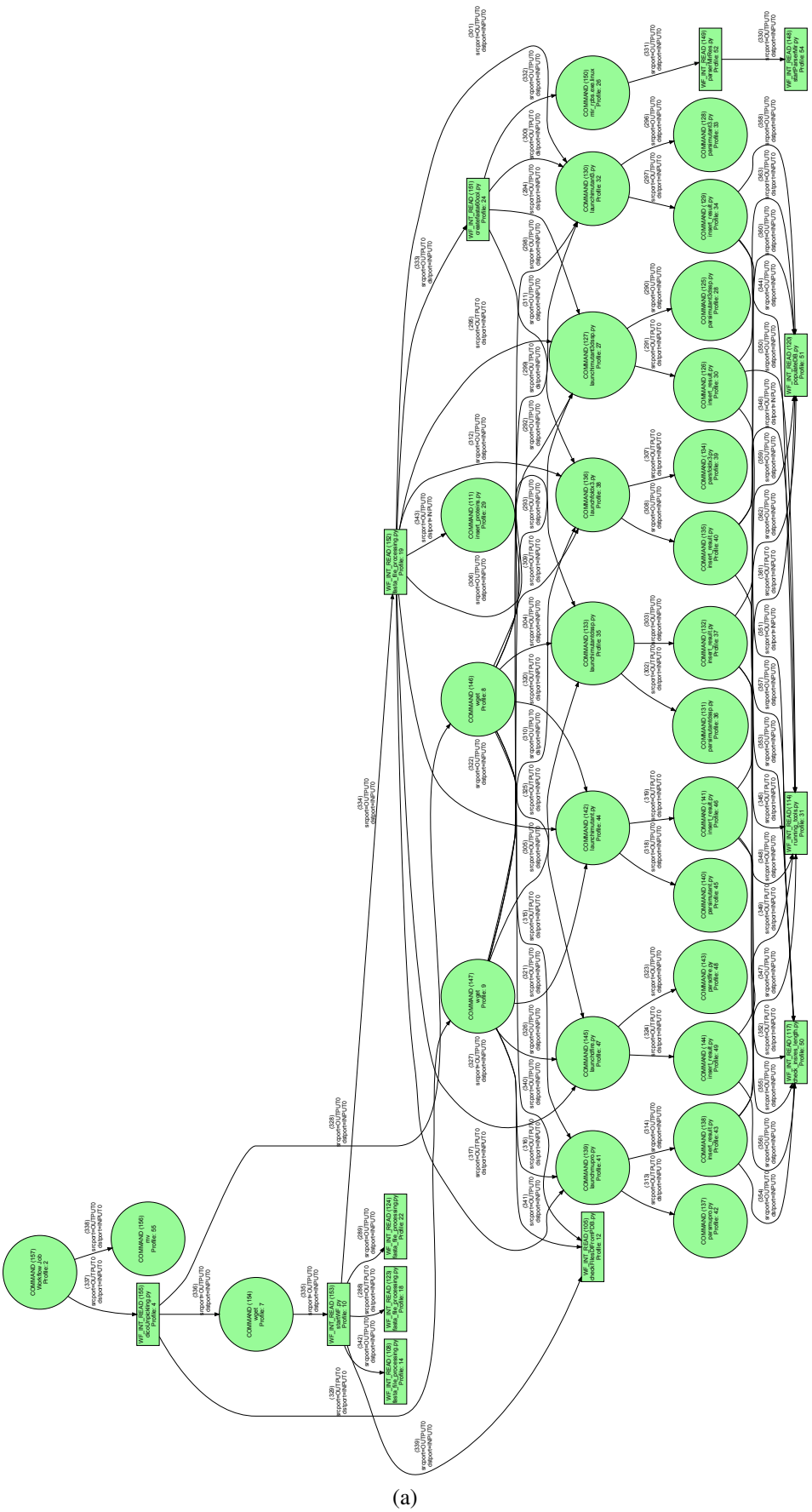


Figure 8: SPROUTS abstract data dependency graph.



(a)

(b) (c)

$$(mupro \oplus (dfire \oplus (imutant \oplus (imutantdssp \oplus (foldx3 \oplus (imutant3dssp \oplus imutant3)))))) \oplus mir$$
$$point\_mutation\_stability\_prediction \oplus residue\_interaction$$

Figure 9: SPROUTS skeleton (a), implementation (b) and conceptual design (c).