Understanding Legacy Workflows through Runtime Trace Analysis

by

Ruben Acuña

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2015 by the
Graduate Supervisory Committee:

Rida Bazzi, Chair
Zoé Lacroix, Co-chair
Kasim Candan

ARIZONA STATE UNIVERSITY

May 2015

ABSTRACT

When scientific software is written to specify processes, it takes the form of a workflow, and is often written in an ad-hoc manner in a dynamic programming language. There is a proliferation of legacy workflows implemented by non-expert programmers due to the accessibility of dynamic languages. Unfortunately, ad-hoc workflows lack a structured description as provided by specialized management systems, making ad-hoc workflow maintenance and reuse difficult, and motivating the need for analysis methods. The analysis of ad-hoc workflows using compiler techniques does not address dynamic languages - a program has so few constrains that its behavior cannot be predicted. In contrast, workflow provenance tracking has had success using run-time techniques to record data. The aim of this work is to develop a new analysis method for extracting workflow structure at run-time, thus avoiding issues with dynamics.

The method captures the dataflow of an ad-hoc workflow through its execution and abstracts it with a process for simplifying repetition. An instrumentation system first processes the workflow to produce an instrumented version, capable of logging events, which is then executed on an input to produce a trace. The trace undergoes dataflow construction to produce a provenance graph. The dataflow is examined for equivalent regions, which are collected into a single unit. The workflow is thus characterized in terms of its treatment of an input. Unlike other methods, a run-time approach characterizes the workflow's actual behavior; including elements which static analysis cannot predict (for example, code dynamically evaluated based on input parameters). This also enables the characterization of dataflow through external tools.

The contributions of this work are: a run-time method for recording a provenance graph from an ad-hoc Python workflow, and a method to analyze the structure of a workflow from provenance. Methods are implemented in Python and are demonstrated on real world Python workflows. These contributions enable users to derive graph structure from workflows. Empowered by a graphical view, users can better understand a legacy workflow. This makes the wealth of legacy ad-hoc workflows accessible, enabling workflow reuse instead of investing time and resources into creating a workflow.

ACKNOWLEDGEMENTS

This thesis would not be possible without the efforts of my committee, and I am exceedingly grateful to them. I thank Dr. Bazzi for keeping me on track with my degree, many technical discussions, and his patience when I was preoccupied with non-thesis work and teaching. I thank Dr. Zoé Lacroix for her aid in improving my manuscript, many non-technical discussions, and opportunities to extend my knowledge in bioinformatics. I thank Dr. K. Selçuk Candan for his valuable input on related work and workflow analysis.

I must also thank my parents, for supporting me both financially and morally in my endeavors. I am beholden to Dr. Yinong Chen and Calvin Cheng for giving me the experience of working as a teaching assistant. In addition, Calvin's many suggestions on places to eat, and his generosity in providing sustenance for many starving undergraduate and graduate students, is much appreciated.

Finally, I would like to thank the National Science Foundation for funding my work under grants CSR-DMSS 0849980, IIS 0431174, and IIS 0612273, as well as Arizona State University for supporting me as a teaching assistant.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

When scientific software is written to specify processes, it takes the form of a workflow. A *workflow* is a graph that describes the execution of tasks to achieve a goal. Many workflows are written in an *ad-hoc* manner in a dynamic scripting language. The accessibility of many dynamic languages has led to a large body of legacy workflows written by non-expert programmers and which cannot be reused because of the difficulty in their understanding and discovery. Previous work on analyzing and understanding dataflow in dynamic languages using compile-time techniques has encountered issues stemming from the difficulty of capturing the semantics of dynamic code (e.g., [36, 26]). The execution of such programs have so few constraints that their behavior cannot be predicted and the use of specialized formalisms to deal with the situation has had little success (e.g., [105]). While using compile-time techniques has had little success, the use of run-time techniques has fared better, albeit not for general workflow analysis. Run-time techniques have been used successfully for provenance tracking (e.g., [81]) but existing results do not apply to the more general problem of workflow understanding. The goal of this work is to develop a new analysis method for extracting workflow structure at run-time, thus circumventing the limitations of compile-time approaches.

The method given here captures the dataflow of an ad-hoc workflow through its *execution* and *abstracts* it with a process for simplifying repetitive regions. To use the method, both an ad-hoc workflow and an input is required. Although an execution of the workflow provides a semi-structured view of the workflow, it is not a full generalization of the workflow process. Functionality in the workflow which is not used in processing the job is unseen, as well as the logic that produced the execution. However, a trace always provides a valid *view* of a workflow in terms of its tools. In fact, such a view of a workflow can reduce the complexity of its structure by containing only relevant elements.

There are four steps to the method in this work. An *instrumentation* engine processes the workflow to produce an instrumented workflow capable of logging events. The instrumented workflow is then executed on a sample input. The execution log undergoes *trace analysis* to produce a dataflow graph. The dataflow graph is analyzed and simplified to produce a graph that captures the essential structure of the workflow. This process is illustrated in Figure 1.1.

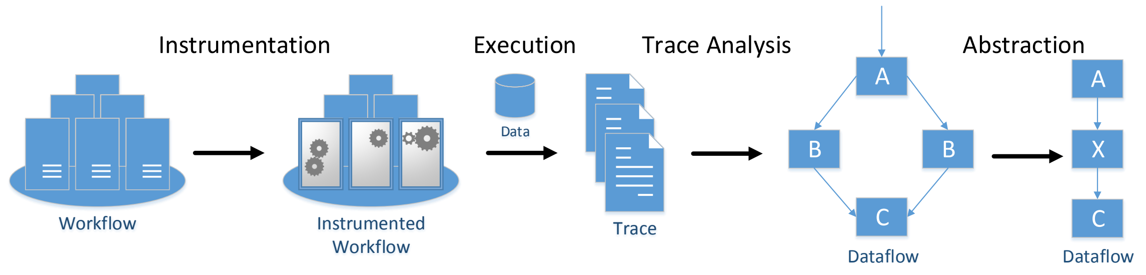

Figure 1.1: Overview of analysis process.

The contributions of this work are the following:

- a trace-based method for recording a provenance graph from an ad-hoc Python workflow,
- a method to analyze the structure of a workflow from provenance.

Methods are implemented in Python and demonstrated on several real world workflows. These contributions enable users to derive graph structure from ad-hoc Python workflows. Unlike other methods, a dynamic approach characterizes workflow's runtime behavior; including elements which static analysis cannot predict. For example, the inmembrane workflow uses a job parameter to dynamically load Python code into the runtime - the static code base simply does not encode the information which describes the workflow. Runtime behavior, whether influenced by configuration files, or programming language features, can indicate critical dataflow in ad-hoc workflows that is otherwise unknown. Runtime analysis also enables characterization of dataflow through external programs by recording an example of their impact on the filesystem. Empowered by a graphical view, users can better understand a legacy workflow. This makes the wealth of legacy ad-hoc workflows more accessible, potentially facilitating the reuse of existing workflows instead of investing time and resources into creating a workflow which may already exist.

The rest of this thesis is organized as follows. Chapter 2 discusses related work in scientific workflows and code analysis in Python. In Chapter 3, three main approaches to extracting the structure of a workflow are discussed. Chapter 4 details the instrumentation portion of the method and how a trace of the workflow is used to produce a dataflow graph. A method for finding and removing repetitive structures is then discussed in Chapter 5. Chapter 6 illustrates the application of the method(s) to several real world workflows and the discusses the performance impact. Chapter 7 concludes with a discussion on future extensions.

Chapter 2

RELATED WORK

A *workflow* is a graph that describes the execution of tasks to achieve a goal. A workflow is represented as a program composed of one or more tasks that are related by flow of data or control. A *task* is an atomic representation of a way to transform data, which is associated with some input and output. A task receives an input produced by other tasks in the workflow and produces an output to be consumed by other tasks in the workflow. A workflow may be executed on any input with valid format and systemically provides output for each. The description of tools used and how they are linked are a workflow's *specification*. The concept of workflows has seen use in both science and business environments. In science, a *scientific workflow* arises from the need of a scientist answering a specific question - e.g., a scientific protocol. Workflows have found application in business environments, where *business processes* must be regulated to ensure quality. Some workflows are created by using a graphical interface to compose various elements. Others are written in a high-level programming language such as Python.

Python is a dynamic programming language that was designed to let users develop systems quickly by providing a feature rich environment that is easy to learn. Python has become pervasive in scientific workflow development because of its accessibility, particularly in life sciences. This has lead to a wealth of existing ad-hoc workflows in Python. The prevalence of Python is shown in Figure 2.1, which shows the languages used in bioinfomatics workflows on GitHub [1] in April 2015, and similar numbers

[1]https://github.com/search?q=bioinformatics

from the Bioinformatics Career Survey [2] in 2012. Note that the survey results are significantly older than the GitHub results but include language use in commercial settings. An *ad-hoc workflow* takes the form of a collection of related scripts with some master script which orchestrates their execution. That is, the workflow is defined in an unorganized way - each piece of code underlying its function exists in some source file but without relation to the whole.

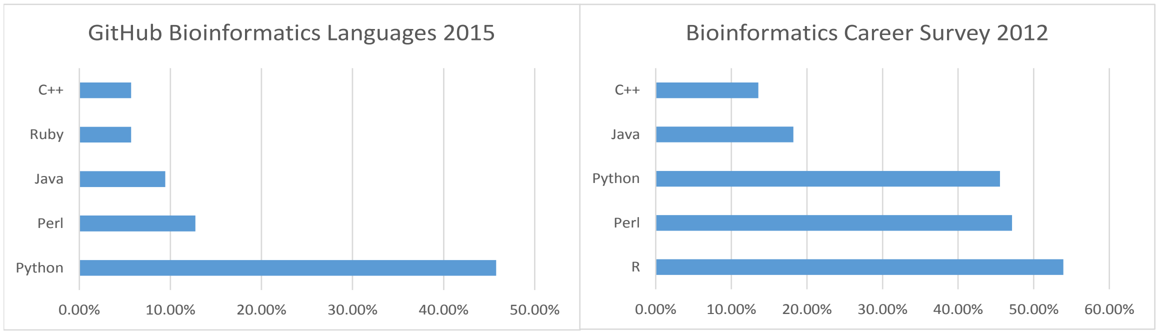

Figure 2.1: Left: Top five languages on GitHub. Right: Top five languages from Bioinformatics Career Survey.

This chapter starts in Section 2.1 with a discussion on systems for managing and executing workflows. The state of the art for recording provenance in ad-hoc workflows is given in Section 2.2. In Section 2.3, methods for finding a workflow specification to address a need are discussed, followed by Section 2.4 which discussing rewriting workflows. Section 2.5 gives general methods for simplifying a graph or tree structure. Lastly, Section 2.6 introduces static and dynamic program analysis in general Python programs.

2.1 Workflow Management

A common motif, lending itself to visual development, is the presentation of a workflow as a graph. This shields the user from the underlying complexity of resources and execution. Thus, workflows are developed with the aid of Workflow Management

[2]https://github.com/michaelbarton/bioinformatics-career-survey

Systems (WFMS), which typically provide a coordinated development environment, specification language, and execution engine. WFMSs require the explicit knowledge of the workflow's structure because they define a high level representation - they express processes as a sequence of related tasks. WFMSs are successful among the scientific community as they provide scientists with the ability to conceptualize scientific protocols as a sequence of related steps. This has lead to a large variety of WFMSs that are available to users.

Kepler [70], a WFMS based on the Ptolemy II system [76, 77], supports modular workflow design (with IDE) and high level task scheduling (using a director/actor system). WOODSS [78] emphasizes the abstraction levels of workflow design and facilitates workflow composition and reuse. Taverna [78] is a workflow system targeting bioinformatics and web service integration. WFMSs that target Grid computing environments, such as Pegasus [35], have been around for some years. Or more recently, Triana [98] which provides a middleware based environment to construct Grid enabled workflows while allowing integration with web services. Some WFMS utilize a web platform to enable collaboration and ease of use. Systems such as the cloud driven Galaxy [46] WFMS, can leverage Taverna workflows by using Tavaxy [3]. SQLShare [54] is a web based platform for doing scientific workflow like data analysis. SQLShare focuses on providing an accessible SQL query view of workflows. Related to workflows are data visualization pipelines. The Ediflow platform [14] enables the convergence of visual analytics and workflows in creating visualization processes by integration of persistent DBMS. VisTrails [12] is a data visualization platform which makes a clear distinction between process and instance results.

Some workflow authors take a less structured approach of using a workflow programming language or a general programming language with a workflow framework. Swift [108] provides a scripting language for describing processes made of loosely coupled and data-centric elements, together with an execution engine for distributed environments. snakemake [62] provides a DSL implemented in Python which gives a makefile-like structure for describing scientific workflows. Dispel4Py [37] is a Python library which provides methods to compose data based workflows and execute them in various environments. However, many scientific workflows are not implemented with a WFMS, or even a workflow framework [71]. So-called *ad-hoc workflows* are implemented directly in a language such as Python. Scripts are developed either as a pure orchestrational program, designed to execute applications and connect their outputs appropriately, or may also contain specific algorithms which are used to guide the process. Using a general programming languages leaves the workflow designer at a disadvantage. Such languages lack support for provenance and repeatability, while promoting unstructured use of tools [71]. These issues are especially problematic when re-targeting a workflow for a new platform [71].

Despite the variety of ways to use workflows, each with corresponding representations and execution engines, there two general types of workflows: Control-driven and Data-driven [91]. In a control flow based workflow, a dependency between tasks A and B indicates that B can start only after A has completed, i.e., it defines task order [71]. In a data flow based workflow, a similar dependency would indicate that A produces data that B consumes, i.e, it defines a dataflow [71]. After preprocessing steps, scientific workflows are typically data-driven and so best represented by a dataflow [71].

2.1.1 Semantics

While WFMSs represent a workflow as a graph to enable its execution, they do not always give useful semantics. In the ProtocolDB [60, 65] WFMS, workflows are expressed in terms of a domain ontology, where each task expresses a specific scientific aim. However, this requires that tasks are available with semantic information. The Structural Bioinformatics Semantic Map (SBMap) [95] is a dual level ontology for storing scientific concepts and resources. SBMap was conceptualized as a method for discovering resources (services) based on ontological concepts instead of textual searches. Other existing systems such as BioMoby, [31] allow mainly textual searches for services or service formats, which are not able to return semantically relevant options. In addition, general resource repositories may be augmented with semantics. In [64], Lacroix and Aziz survey the state of the BioMoby [104] web services registry for bioinformatics. They propose a method for extracting semantics from BioMoby and give concrete results. Semantic models can also be used to represent workflows during their life cycle. One way to capture a workflow (and its life cycle) is by using the the idea of Research Objects (RO) [34]. A RO is the encapsulation of various dimensions (e.g., reusable, repurposable, repeatable, etc.) of some scientific problem as a social object to enable interaction between researchers and existing work. ROs can used together with existing ontologies to capture information about a workflows basic specification, executions, and various annotations [13].

2.1.2 Workflow Similarity

An advantage of using a WFMS is that the structure of a workflow is explicitly defined. The structure of a workflow provides a means to enable comparison via similarity. Comparison is a fundamental operation for workflow discovery - it enables queries. Similarity is a topic already considered for general programs with source code. Liu et al. [67] looked at using a Program Dependence Graph to discover similarity between programs as a way to detect plagiarism. Their approach involves solving a restricted version of the graph isomorphism problem based on the constraints that are mandated by using a Program Dependence Graph on real world programs. When workflows are structured, semantics can aid comparison. For example, in ProtocolDB [65] equivalence of workflows and tasks is realized by mapping elements to an ontology and checking for identity or subtyping of concepts. Another approach is to define a set of rules for assessing the similarity of a workflow and analyzing its graph like representation of nodes and edges in terms of their semantics [15]. Specifically, [15] use this assessment for the retrieval of workflows from a repository. One limitation of [15] is that the authors only consider the semantics of the workflow. Workflow executions provide another way to characterize workflows but are not yet in use [94]. At least in the domain of of scientific workflows, the community lacks repositories to store provenance [94]. Another concern is the quality of service offered by multiple workflows. Ma et al. [72] give a distance based measure for selecting similar workflows based on an execution time constraints.

2.2 Workflow Provenance

A workflow naturally describes the process to create an end product, however the process is generic and may be applied to many inputs, with each input evoking a slightly different process. The record of how a specific product was created is its *provenance*. A *trace* is the provenance of a workflow execution. Provenance is useful in science for tracking analysis progress and enabling reproducibility. Provenance can also be leveraged to support reuse of data for new workflow executions.

WFMSs like Taverna capture data provenance, but do not focus on data reuse. Before, during, and after, run time, Chiron [53] stores provenance information based on the flow of relations between workflow activities. Chiron supports reuse of data as well monitoring the run time state of executions to identify deviations. In some domains, interacting with provenance during a workflow's development is important. VisTrails [12] maintains provenance for visualization results by 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 [25]. The Ediflow platform [14] enables the convergence of visual analytics and workflows in creating visualization processes by integration of persistent DBMS. A second focus is on providing a live interface between a DBMS and a visualization system to enable change propagation [14].

Analysis of traces is also valuable, for instance, Bao et al. [10] give a method for differencing executions to understand control flow in provenance. Bao et al. defines the differencing problem as the computing a list of transformation between two traces which adhere to a workflow specification. Workflows are specified in so-called sp-workflow format, which is comprised of a sp-graph [101] annotated with information about looping and forking. This was implemented in PDiffView [9], an graphical application which imports workflows into a sp-workflow format, generates valid runs, and then shows the operations in differencing them. This differs from workflow structural extraction as it focuses on comparing the structure between executions, not extracting executions or comparing elements within an execution to deduce overall structure.

2.2.1 Python Workflows

Provenance tracking has been addressed for Python based systems in several ways. One way to track provenance is by enabling the user to explicitly define the dataflow using a provenance API (e.g., [18]). However, this is intrusive since the workflow must be engineered to use the API. IncPy [50] is a non-intrusive and low level approach to the issue of data reuse; it involves the replacement of the standard Python interpreter with one which automatically catches the result of functions as they are called.

Starflow [8] addresses the issue of tracking provenance and data reuse in workflows authored in Python. Starflow provides a data analysis environment at the level of Python's interactive interpreter. Starflow takes a function view of programs - functions are versioned and their execution parameters recorded. 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 thus determines what functions must be reexecuted.

ProvenanceCurious [55] provides a method to extract provenance from a Python program for debugging. Using an input 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 then 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 [81] addresses the issue of providing a non-intrusive and systematic way to collect provenance information in a general Python program. Like Starflow, noWorkflow is file centric, but unlike Starflow is based on static 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 for functions, function parameters, data files; all of which are stored in a local database. noWorkflow then provides analysis functionality with this database: graph analysis, difference analysis, and query analysis. noWorkflow also 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 external tools, or, reducing the complexity of the dataflow.

2.3 Workflow Reuse

Since scientific workflows are data-centric, their reuse is attractive as users can provide their own input data and yet leverage an existing system. Users may also want to modify the actual workflow structure. Although WFMSs such as Taverna provide mechanisms like composition to build new workflows from old ones, they lack capabilities for reusing ad-hoc workflows. During the 2006 Challenges of Scientific Workflows workshop [44], many issues were identified including the discovery (for reuse), creation, merging (for reuse), and execution of workflows. Workflow reuse may take the form of a user retrieving an existing workflow from a repository (see Subsection 2.3.2), provided their search yields a satisfactory result.

Unfortunately, repositories may have a low population of workflows. Cohen-Boulakia and Leser [30] indicated that scientific workflow management systems themselves have not yet reached widespread acceptance. Present solutions fail to provide functionality required by users: Reuse, Search and Compare, Adaptation, Assembly, and Run Analysis [30]. As noted in [44], workflow reuse continues to be a significant issue. A major issue is that workflow users are already comfortable with existing ad-hoc methods and do not find the learning curve for WFMSs to be worthwhile [30] even given the inherit advantages. Another ongoing issue in the reuse of workflows is the difficulty of understanding of existing workflows [30, 41]. Issues of maintainability and extendability can restrict users from reusing existing legacy ad-hoc workflows [4]. In fact, Garijo et al. [41] examined updates made to workflows on MyExperiment [45] and found that over half of them were a result of either general maintenance (e.g.

fixing an broken external tool) or fixing bugs for continued use.

2.3.1 Workflow Discovery

Workflow repositories are places where workflows may uploaded, searched, and retrieved. Most workflow repositories use keywords to locate workflows based on title or description, or with additional refinement based on tagging (i.e. to indicate WFMS). However, as repositories grow, better mechanisms for finding a workflow or determining its similarity to another workflow [94] (e.g. a query), are necessary. There are two main ways to compare workflows: annotation or structure [94]. Both methods assume a workflow is available in a structured form. Annotation methods allow comparisons (e.g., label edit distance) across different WFMS (or execution engines) [32]. Structural methods rely on information fundamentally embedded in a workflow (e.g., graph edit distance) but suffer from the NP-completeness of graph isomorphism. Many methods for comparing modules reply on labels or types [94]. A comparison of techniques for determining workflow similarity on a standardized corpus shows that annotations provide the best to way measure module similarity, provided that the annotations are well chosen [94]. Problematically, for legacy workflows [40] found that discovering annotations from textual descriptions with the aid of ontological information was difficult due to heterogeneity of workflows and lack of metadata. Structural approaches can outperform perform annotation based on configuration, especially for poorly annotated workflows [94].

2.3.2 Scientific Workflow Sources

There are variety of repositories a user may use to discover workflows. MyExperiment is a social platform for storing workflows and enabling collaboration [34]. When developing PDiffView, Bao et al. [10] used six workflows from MyExperiment.

However, as of 3/4/2015, none of the six workflows could be retrieved, suggesting that workflow storage may be fleetingly. Although most of the workflows found on MyExperiment (2033 of 2686 on 3/4/2015) are for Taverna, other WFMS are supported (e.g., Kepler, Galaxy, Vistrails). However, of the 2686 entries only 12 are tagged with the Python keyword. All of these entries are in fact workflow elements which execute a python script to complete a task (e.g. find an average and standard deviation). Tavaxy [3] provides a repository containing versions of workflows, originally authored in Taverna or Galaxy, and which have been imported into the Tavaxy format. There is slightly more diversity in the SHIWA workflow repository [3] , which contains workflows using 11 execution engines (including non-WFMS such as BASH, Python, and BinaryExecutable). However, the SHIWA workflow repository contains less than 200 workflow implementations and the SHIWA project [4] itself has ended. Beyond MyExperiment, scientific workflow repositories tend to focus on a particular WFMS (e.g., CrowdLabs [75] for VisTrails) or workflow engine (e.g., Snakemake Workflow Repository [5]). At present, the scientific community lacks targeted repositories for ad-hoc workflows.

An alternative source of ad-hoc workflows is code hosting websites. GitHub [6] is a free provider of GIT source code hosting for open source projects. The following is an example of how workflows were obtained from GitHub to study ad-hoc workflow structure. GitHub was searched for workflows with the keywords 'python protein workflow' and 'python protein pipeline'. The word 'python' helps to identify repositories with missing or incorrect language tags by matching readme or documentation. The search is confined to the domain of protein analysis using 'protein' - it was found

[3]https://shiwa-repo.cpc.wmin.ac.uk/shiwa-repo/

[4]http://www.shiwa-workflow.eu/

[5]https://bitbucket.org/johanneskoester/snakemake-workflows/src

[6]github.com

that the general term 'bioinformatics' gave fewer results. Other keyword choices like 'sequence' resulted in many small programs created for demonstration or practice. Some repositories, despite storing workflows, are labeled 'pipeline' due to their restricted structure. Since they entries are also of interest, a second query was made. These queries gave several hundred results, however, some are not workflows (i.e., made of tools) or are not written in Python. For any author (registered GitHub user), only one workflow was considered. This reduces sample bias from multiple workflows with similar designs. Based on file extensions, it is immediately clear to a reader if a program is written in Python. To determine workflow nature, each result's readme or main source file was reviewed to determine its purpose. Most entries stated constituent tools as part of their readme file, thus identifying themselves as workflows. Others showed tool use from the manipulate of paths and executable names in their source code. Some workflows only revealed certain tools when it failed to install or run due to missing binaries. Of the remaining workflows, only those with at least four tools were selected, this ensured there was sufficient dataflow to give interesting results. Finally, each result was checked for executability: contained input files, and used tools which were available for the Linux environment. Note that due to the large breath and heterogeneity of ad-hoc workflow, it is not strictly possible to construct a representative ad-hoc workflow corpus. Rather, an random section tries to capture diversity in implementation which should include common mechanisms.

From the GitHub results satisfying the selection criteria, the first six workflows (as ordered by search) were selected. Although more workflows may be found on GitHub, these workflows contain a suite of nearly 30 tools which appear to illustrate most common ways to read and write data (see Subsection 3.2.1).

- asr-pipeline [51]: enables creation of phylogenetic trees.

- bacana [83]: predicts and annotates genes in bacterial genomes.
- hybseqpipeline [57]: a sequence assembly workflow for Illumina reads.
- inmembrane [84]: checks if a bacterial protein codes for a surface-exposed region.
- miR-PREFeR [66]: predicts plant microRNA from RNA sequences.
- pycoevol [73]: analyzes the coevolution of proteins.

2.3.3 Business Process Mining

In the business domain, there is interest in creating (or, mining) process models from event logs. The term process is sometimes used interchangeably with workflow in business-related literature. There are three main motivations for process mining [1] : discovering a process, analyzing process performance, and comparing the actual process with its definition. A log is the trace of a workflow's execution as a series of events. Constructing process models was first presented, for logs produced by IBM Flowmark, in [6]. In the business realm, workflows are typically based on Petri nets [71]. The alpha algorithm by Aalst [102] provides algorithmic foundations for mining a process by determining which events succeed others. The alpha algorithm takes a set of event logs, each consisting of an unordered list of tasks which occurred, and constructs a workflow net (a class of Petri nets). Construction occurs by creating a graph with nodes for all events, adding nodes between followers in the traces, and removing patterns of excess edges. Dataflow is implicit in the order of the events implied by the model capable of generating the input logs. Other extensions of this work include letting users find a model by allowing visualization with dynamic parameters [49], adding heuristics to deal with noisy logs [103], and using a genetic algorithm [33] that creates a random model and repeatedly mutates it to produce a model which can produce the log. Instead of event logs, the process mining algorithms may also consider data provenance. Zeng et al. [107] propose using the provenance recorded by WFMSs to create scientific workflow models, which could be compared with the workflow expressed in the WFMS. Zeng et al. tried four process mining algorithms to construct control flow from provenance information (collected using Taverna), but all failed to exactly reconstructed the original scientific workflow. Although the results may be improved by exploiting the additional data dependencies contained in provenance, existing process mining tools do not use data dependency information [107]. A separate issue is logs produced by independent but cooperating processes, which need to be merged before a top-level process can be mined. Clases and Poels [27] addressed this issue with a method that uses the attributes of two logs. A user indicates which attributes of a pair of logs correspond correspond to the same job, and then the method merges attributes to uniform values. Alternatively, the issue of concurrent workflow logs, by using temporal dependencies and refinement is addressed in [69]. Although many methods mine a process, Abdelkafi et al. [1] argue that a workflow is more than its process. Specifiically, these methods do not provide insight into the organization that implements the workflow or the structure of the data it operates over.

Analysis may also occur on a log of user actions. For traces produced by an expert using services in a medical domain, [106] presented a model merging technique to learn repetition and branching. This was explored again in the POIROT project [24] which combines trace analysis and learning methods to deduce procedural models. An ontology query method to infer regions of missing dataflow was used for traces produced by expert users which include unobservable choices or actions in [43]. Outside of digital workflows, Bouarfa and Dankelman [20] propose mining on logs of activities during surgeries. They use a sequence alignment method to deduce a consensus log which could be compared with an ongoing surgery to detect discrepancies.

Models produced by process mining can also aid quality of service. Kraiss and Weikum [61] address the issue of prefetching data in a three tiered data system being queried by clients. A Markov chain model is created using a record of requests and models the behavior of the attached clients to predict their requests. LogO [90] applies this idea in the domain of distributed multimedia, where an automaton is learned from the trace of data requested by a number of clients. The automaton differs from Markov chains in [61] by considering dependencies on when an event occurs, and allowing multiple resources to be requested at once.

Process mining closely relates to portion of this thesis on instrumentation and dataflow construction. The majority of process mining occurs on traces produced by management systems, and does not address ad-hoc workflows. Although instrumentation methods are given in literature, they are mainly for workflows being managed by a person. Hence, this work provides an instrumentation mechanism for ad-hoc workflows. Existing process mining methods depend on an ordered list of events; they do not use the input or output of events. Given many inputs, such methods determine common event orders, and use them to define dependencies. By using multiple inputs, process mining methods are able to characterize a more flexible and complete workflow. In contrast, the work presented here records the input and output of events in a single trace as a semi-structured view of a workflow. Rather than rely on event order, this permits events to be connected based on what produced the data it needs - this complements process mining which does not analyze data flow. This method can provide certitude that discovered dependencies are correct, unlike process mining where dependencies are approximations based on the number traces analyzed. However, using a single trace limits the portion of the workflow that can be constructed.

2.4 Workflow Rewriting

The high complexity of some workflows can prohibit a user's understanding or slow other analysis methods. Garijo et al. [41] suggest that one way to handle the complexity of existing workflows is by providing a more abstract view of the components or sub-workflows within a system. An idea for higher abstraction in workflow representations is *views*, which are composite tasks in a workflow. Views may be computed from workflows but existing tools are not guaranteed to produce views that are sound (preserve data flow) [11, 17, 96]. Directly rewriting a workflow can also address complexity. For example, use of Taverna enables application of DistillFlow [29], which provides methods to rewrite workflows automatically by eliminating known anti-patterns.

Reducing the complexity of mined processes is also studied for business workflows. Kudo et al. [63] introduced the notion of "pseudo-hubs" - elements in a mined process model which are produced by auxiliary events (e.g. task completion, warning message, opening a document for a task, etc.) and which are not needed in a process's representation. Futhermore, a Process Skeletonization method enables the simplification of such elements from a process by searching a process for "pseudo-hubs", ranking the results, and then presenting them to a user for possible removal [63].

2.5 Graph Summarization

Summarizing a graph has become an important topic due to the prevalence of big data that is naturally modeled as a graph, e.g., gene networks, web graphs, and social networks. Handling these graphs can cause several problems: the graph may be too large to store in memory, graph algorithms may become slow, and the volume of information may prevent understanding [82]. This can be addressed with graph summarization. Recently, Pienta et al. [86] performed an extensive survey of methods for making sense of graphs via algorithms, visualization, and interactivity. Cluster approaches, where groups of nodes are selected by some property, involve creating super-nodes and/or super-edges representing a more complex subgraph. One approach is to cluster nodes based on attributes or common relationships [99]. Entire subgraphs may also be used. In [22], the edges making up complete bipartite subgraphs are replaced by a node with a single edge to each subgraph node. Khan et al. [58] combine the idea of clustering on dense subgraphs with an information theoretic approach. The idea of a Minimum Description Length (MDL), which states that the best representation for data is the one with the most compression, is used to select subgraphs to remove. A more general approach is searching for frequent subgraphs. Unique subgraphs can be discovered by enumerating possible subgraphs and comparing their statistical prevalence with a randomized graph [79]. However, graph enumeration is slow and the problem is similar to graph isomorphism, thus subgraphs are limited to around 15 nodes [48]. The method in [48], uses a query rather than enumeration approach and avoiding computation for subgraph symmetries. Instead of subgraphs, Navlakha et al. [82] focus on node pairs and MDL optimization. Given a graph, pairs of nodes are nodes are examined and merged into super nodes when the resultant graph and corrections has a smaller description than the original graph.

These topics relate to the abstraction and skeletonization portion of this thesis. In this thesis, the goal is to find repetitive regions and collapse them. Since graph approaches intend to provide generic simplification, they depend on topology instead of semantics (e.g., name of a tool). Typically, a graph approach identifies a dense subgraph and replaces it by a simpler subgraph. However, dense regions (e.g., a clique) may not occur in workflow dataflow, which often resembles a DAG. Another issue is that the selection is somewhat arbitrary interesting features (e.g., a specific node) may vanish to be replaced by a super-node without explicit semantics. A closer problem is frequent subgraph mining, which would be able to identify repetition. However, frequent subgraph mining has mostly been examined in general graphs, where identification is slow and imposes an upper bound on subgraph size. In contrast, focusing on the workflow domain refines this problem to DAGs with annotations, enabling a greedy, rather than combinatoric approach.

A related problem occurs in semi-structured data; data which contains structure but which is irregular or incomplete with respect to a global view. Extracting the true structure can enable data validation, user understanding, and provide an index to speed up queries [47, 16]. One of the first methods to address this for semi-structured data in databases was DataGuides [47], an automaton approach to creating and maintaining a summary of a graph-based database (Lore). A more recent, and specific, focus is extracting structure (i.e., a schema) from XML data. XML is a hierarchical text format for storing data; the data may be constrained by a schema which specifies the structure and content of its elements. XML often stores data in a semi-structured manner, i.e., lacking the corresponding schema, particularly in the web. Schema extraction may occur on small sets of XML files (where the schema must be generalized), or large sets (where the schema must be kept concise) [16]. Techniques often focus on inferring a regular expression (or similar) from the semi-structured XML. iDTD [16] extracts a DTD (a legacy form of schema) using subclasses of regular expressions that can be determined with only positive examples. XStruct [52], an extension of XTRACT [42, 42], a MDL technique, uses multiple XML files and deduces unambiguous regular expressions for the children of each XML element. Beyond regular expressions, Janga [56] proposed using a context free grammar to model XML, removing dependance on structure format and allowing the extraction of schema, DTD, or other structural representations. Schemas or data may be directly simplified. In [74], schemas are are summarized by ranking each element using the PageRank algorithm and eliminating those with low scores. The summarized schema is then used to filtered XML data, which is then aggregated based on labels, to produce summarized XML data. Szlávik et al. [97] determine which XML elements are important using a probabilistic method that uses eight different features covering element topology, content, and order.

Although the trace view of a workflow provides a semi-structured view like XML files, the solutions for schema extraction do not directly apply. The key difference is XML possessing a hierarchical structure instead of the DAG structure of a workflow. This is used in algorithms (e.g., DataGuides, IDTD, XTRACT) for schema extraction where a regular expression (or similar) is constructed each type of element, thus giving a tree structure. The solutions rely on data encoded in the names of tags and attributes, to determine the top-down similarity between branches. Much of this information is absent in workflows, where only the name of nodes is known. These schema extraction techniques also focus on cases where many XML files are known. Although this permits extracting more generic structures than the trace solution in this work, it can produce a very specific schema when a single XML file is available. Since this work does not use multiple traces, its generalization relies on identifying data collections - which can be discovered from a single trace. Although XML summarization simplifies a graph, its summary is designed as a sample of meaningful XML data, instead of removing only redundant information as in this work. In contrast, this work aims to preserve all elements by eliminating only repetition.

2.6 Program Analysis

Program analysis in Python is typically motivated by optimization. Due to Python's poor performance as an interpreted language, several projects (e.g., [36, 19, 89]) offer the ability to transform the source code of a Python program into a C/C++ program that may be run on a different platform. StarKiller [89] is designed to generate equivalent C++ programs from Python source by a specialized compiler with a type inference mechanism based on the Cartesian Product Algorithm [5]. Shed Skin [36] provides similar functionality [5] but with an additional focus on optimizing memory allocation in the generated result. PyPy [19] is alternative implementation of the Python interpreter based on JIT compiler techniques; part of this project is the RPython (Restricted Python) tool chain which allows analysis of RPython code and generation of code targeting C (POSIX), CLI (.NET), or Java (JVM).

Several techniques analysis have been designed to help understanding a Python program. pycallgraph [7] generates a call graph for the execution of a program. Snakefood [8] recursively parses source code files to determine which other files they depend on. Program slicing [92] is the general problem of determining which part(s) of a program effects the value of a variable at a specific place. The problem of program slicing in Python was first studied by Xu et al. [105]. Xu et al. observed that the dynamics of Python's first-class objects required special attention and so proposed, but did not implement, a new dependance model with additional support for tracking dependencies between variable definition and usage. Chen et al. [26] argued that static analysis was insufficient to determine all dependencies in Python programs. Instead, Chen et al. give a hybrid technique for Python, involving static analysis for control-flow analysis and data dynamic (bytecode level) tracing of memory access.

[7]http://pycallgraph.slowchop.com/

[8]http://furius.ca/snakefood/

Chapter 3

ANALYSIS APPROACHES

There are several ways to capture the dataflow of an ad-hoc workflow. This chapter discusses three main approaches to the analysis of ad-hoc workflows written in dynamic languages, and the associated disadvantages and advantages. **(1) Code review** - the programmer centric method of determining a program's function by manual inspection. **(2) Static analysis** - the automatic analysis of a program to determine information about its behavior without its execution. **(3) Dynamic analysis** - the automatic analysis of a program during its execution to determine information about its behavior.

Dynamic analysis is the approach taken later in this work. In practice, code review becomes unwieldy in legacy scientific workflows. The origin of such workflows means that software engineering practices may not have been applied during development, leading to a code base which is effectively obfuscated. While static analysis has well developed techniques for dataflow analysis, they fail to support behavior in real world workflows. For example, when a configuration file must be loaded to determine how data should be processed, or when dynamic code evaluation is used. In general, static analysis methods focus on compilable/typeable languages and have assumptions which conflict with the Python programming model. A dynamic analysis approach avoids these issues by capturing a workflows behavior at the level of execution events. Thus, the method given in Chapter 4 for workflow structural discovery is a dynamic analysis technique.

3.1 Code Review

A natural approach to discovering the structure of an ad-hoc workflow is to review its implementation. This method was used by [4] who discussed the various impacts of workflow transformation and illustrated them with a case study on the Structural Prediction for pRotein fOlding UTility System (SPROUTS) [68]. SPROUTS is a bioinformatics workflow, implemented in Python. It performs predictions using a suite of six computational tools to examine the impact of point mutations on protein stability. The SPROUTS workflow (WF1) is a set of scripts which are manually executed in a specific order to produce uploadable SQL files. A developer began to develop an automated workflow (WF1.5) but not all needed features were added. WF1.5 was later finalized into a workflow (WF2) which automatically fetches and uploads data. Developing WF2 was accomplished by an author with programming and domain knowledge. No external tools for undeterstanding workflows were used. Today SPROUTS is available online [1] and has users in 22 countries. Although completing WF2 required much effort, in the case of a well structured and documented workflow, code review can be almost trivial. Code review provides a generic approach that can, potentially, deal with unexpected or novel workflow structures. It is language-agnostic and does not require specific tools - or training users on such tools - all that is needed is a source code editor for the language of the workflow. It does not rely on any particular abstractions, or patterns, which the workflow may not align with.

In [4], three ways a workflow's implementation may become difficult to understand are discussed: Problems of Iterative Design, Community-Based Practices, and General Workflow Issues. Two of these are now discussed. Many issues in com-

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

pleting WF2 came from its iterative development from multiple authors. **Iterative design**: Although documentation for a system may be available (e.g., literature), it can subsequently go out of date in favor of maintaining the concrete system. New systems often reuse parts of old systems that are not directly applicable, this results in the formation of 'wrappers' that obscure interactions between tasks. A system may end up fragmented on different execution platforms so that its true workflow representation is undermined. A workflow may accumulate redundant and obsolete documentation and source code, obscuring the actual workflow. Rather than being caused by staggered development, some issues in SPROUTS come systemically from scientific workflows. **Community Based Practices**: Within a community, there may not be strict standards for information exchange, leading to incompatibility between tools or interfaces. Python's unstructured nature can lead authors to apply (intentionally or otherwise) easy to use but hard to understand code constructs. As Chen et al. [26] note, Python workflows may contain "unlimited errors". The implementation of a system may not be consistent because often a developer is learning the language at the same time, over time new techniques are introduced. A reviewer might need strong familiarity with a domain to understand variable names. Variables may have naming which does not following standard conventions, thus misleading the reviewer.

The issue with code review is time and effort - as the complexity of a workflow increases and/or its structure decreases, review code simply becomes unmanageable. In the case of workflow discovery for reuse, the time needed to understand a workflow may exceed that which would take to develop it. However, complexity in a workflow often stems from repetition. The task of tool discovery in a workflow, for example, must be repeated many times for each tool in the workflow. Repetition enables automation. Automation is also a process which can ignore many of the mechanisms which serve to cloud an experts appraisal (e.g., bad variable names).

3.2 Static Analysis

As discussed, code reviewers may have to contend with various issues in an ad-hoc workflow. Automation can address some of these issues. There are two types of automation: *static* which considers a program's without executing it (i.e., via source code) and *dynamic* which considers a program during execution. Automation requires a stricter idea of workflow structure: a program which orchestrates the flow of data *files* between executable *tools*. Many static analysis techniques for dataflow in programs have been developed (see [59] for a discussion). As an automated method, static analysis reduces the time and effort that a user would need to understand a workflow. A general approach would inspect a workflow's source code to track dependencies among code reading and writing files (e.g., ProvenanceCurious [55], Starflow [8], noWorkflow [81]). Since static analysis is typically performed on source code, such a method has access to all control flow. This enables coverage of decisions made by the system. This is the approach of ProvenanceCurious, which records provenance in ad-hoc Python workflows. StarFlow [8] also uses static analysis (with run-time analysis) but acknowledges that a static dependency graph forms a superset of all possible control flow graphs instead of a provenance graph itself [7].

However, existing dataflow techniques are more suitable for programming languages which are compiled and/or statically typed. In contrast, Python is a dynamic language and supports source code introspection. This includes features such as being able to execute (i.e., `eval`) a string as Python code. Introspection allows one to inspect and modify the contents of the runtime environment programmatically. These dynamic features of Python make static analysis challenging. Work such as Shed Skin [36] has attempted to provide type inference functionality, and in the process has demonstrated the issues with static analysis in Python. One approach, seen in the use of RPython by the PyPy project [19], is to define a specific subset of Python that permits static inference of the types. However, this requires that a program be designed to use that restricted language. Workflows from GitHub revealed that workflows make use of more dynamic language features, and data driven configuration, than their functionality would suggest. This is partially due to the use of Python that unintentionally invokes dynamic language features [4]. The SPROUTS [68] workflow loads a file at runtime which contains information on what tools are available and when they may be run. The workflow inmembrane [84] loads a configuration file based on the job parameters it receives and then loads a source file to dynamically `eval` its contents. Tools like ProvenanceCurious which are purely static and have no support for `eval`, are simply unable to to track provenance in these types of workflows. In contrast, Starflow and noWorkflow, which also use dynamic analysis, would intercept file activity even if dependencies were unknown. The miRPREFeR [66] workflow relies on code which is lexically correct but contextual invalid. This causes the interpreter to crash on a line it cannot execute and trigger a runtime exception, thus forcing the workflow to following an unintentional code path. Since ProvenanceCurious relies on a program dependency graph to model provenance upon, it would be unable to detect this behavior which emerges at runtime.

3.2.1 Analyzing Tools

One limitation common to Python provenance methods (e.g., ProvenanceCurious, StarFlow, noWorkflow) is a lack of support for tracking external tools. Although static analysis gives full access to a workflow's implementation, it provides only minimal information about the structure of the tools it invokes. Such a method would need to determine the dataflow for a tool from a (likely symbolic) command string.

For example, if something was prefixed by `-input` or `-i` it would be considered to be an input file. As mentioned previously, it can be imprecise to use such names because of inconsistencies in their meaning, provided the names exist at all. Due to these concerns, the tools in six workflows were reviewed to determine how they interacted with the filesystem. The workflows were asr-pipeline [51], bacana [83], hybseqpipeline [57], inmembrane [84], pycoevol [73], and SPROUTS [68]. These workflows contained a total of 28 command line tools. These were: Alien Hunter, BLAST, CAP3, DFIRE, EXONERATE, FoldX, Glimmer, HMMER3, I-Mutant 2, I-Mutant 3, Lazarus, LipoP, MAFFT, MEMSAT3, MIR3, MUpro, MSAProbs, MUSCLE, PhyML, PRANK, RAxML, Prodigal, RNAmmer, SignalP, TMHMM, tRNAscan, and Velvet. Tools were surveyed to determine how parameters values were passed to them at the command prompt. This was performed by manually analyzing how the workflow invoked them and checking their respective user manuals. The results of the survey are shown in Table 3.1.

Table 3.1 is separated into three parts. The first lists the method a tool uses to label option parameters. The second (rsp. third) is how the tool determines what file or folder to use as input (rsp. output). For input and output, the pattern column shows what a command should look like. `[keyword]` designates a keyword which annotates a parameter. `[delimiter]` designates a character used to show a boundary in a parameter. `[filename]` designates a file name. Note that some tools use a filename to load multiple files (i.e., a common substring). `[exe]` designates the name of the tool. Note that while some workflows use keywords like `input` and `output`, it is unlikely that the text can be used to determine the type of IO.

Reviewing this table, some parameter patterns become apparent. When passing options to a tool, most use dash. However, some instead use the order that parameters are given. About half of the tools take only the raw input filename with no annotation

Table 3.1: Parameter passing patterns for tools.

Parameter	**Pattern**	**Freq.**
Options	`--`	2
	`-`	17
	`-` or `--`	2
	by order	3
	n/a	4
Input File	`[exe] [filename]`	15
	`[exe] [prefix][keyword][delimiter][filename]`	7
	`[exe] [prefix][keyword]=[filename]`	1
	`[exe] [prefix][keyword]([delimiter][filename])`^{2}	2
	`[exe] <[filename] (via STDIN redirect)`	3
Output File	`[exe] [filename]`	5
	`[exe] [prefix][keyword][delimiter][filename]`	7
	`[exe] [prefix][keyword]=[filename]`	1
	`[exe] >[filename]` (via STDOUT redirect)	10
	internal default	1
	determined by input (e.g., substring)	4

to indicate its purpose. The rest of the inputs do use a keyword but require various syntax. For outputs, many tools use the STDOUT stream (which may also be used for logs, not only output). The others use various mechanism similar to the inputs. From these results, the heterogeneity of data passing mechanisms can be seen. In several cases, it is simply impossible to determine dataflow even with manual review. Thus, it seems that additional information about the invocation of a tool is required to understand its relevant dataflow. In order to understand a tool's dataflow, it is useful to observe it's action in the context (i.e., execution) of a workflow.

3.3 Dynamic Analysis

In the previous section, dynamic analysis was briefly introduced. There are two branches of dynamic analysis: analyzing the workflow at *run-time* or recording the workflow's execution (a *trace*) for later analysis. Dynamic analysis is an automated technique which sidesteps the dynamics of Python in favor of viewing exactly the dataflow which takes place - it focuses on 'what' is produced rather than the 'how' it is done. IncPy [50] provides automatic memoization (and potentially provenance tracking) at run-time and takes the approach of creating an instrumented interpreter. StarFlow uses a run-time approach to validate file access against what was discovered during script and annotation analysis, but does not use it to generate dependencies. StarFlow works by injecting modules into the interactive interpreter to create an interactive data analysis environment. In contrast, noWorkflow operates on a whole program. noWorkflows executes a workflow in debug mode and attaches listeners.

A trace of an executing Python program may be obtained in several ways. Existing tools such as *strace* on Linux can be used to log all interactions between a process and the OS. Systems such as Provenance-Aware Storage Systems (PASS) [80] implement their own mechanisms for intercepting system calls to record provenance information. Unfortunately, the file access overhead of the default Python interpreter makes analyzing such a trace difficult [7]. For a general programming language, a trace can instead be performed at the level of abstraction that the workflow designer considers: libraries. This can take the form of a thin layer of code between a workflow and the libraries, thus logging exactly the events the workflow designer has explicitly created. Then, minimal filtering of events is needed and only one version of the interpreter is needed. Another advantage of tracing libraries is that a thin layer is more amenable to changes in the interpreter. Only the portions of the trace method which interface with changes in the interpreter must be updated.

Unfortunately, dynamic analysis suffers several innate limitations. **Correct workflows** - Viewing behavior requires executing a workflow, so a prerequisite is a functional workflow. From a workflow reuse perspective, this is a significant drawback as a workflow must be maintained until it is examined. One option is to make dynamic analysis part of an archival process to be completed after a workflow as served its primary purpose. **Semi-structured workflows** - Behavior can be analyzed as a semi-structured views of workflows. While a single execution gives some insight into the overall orchestration of tasks in a workflow, an execution is not a generalization of how a workflow processes every job. The internal logic which occurred to generate the specific interdependencies exists at a lower level of abstraction. However, an execution always provides a valid view of a workflow. Given multiple executions of a workflow, it is possible to learn a more generalized workflow structure based on the similarities between traces. This is similar to systems trying to determine how data are produced on the Web, or business process mining (see Subsection 2.3.3). Although fully structured by the authority that designed the resource, they appear to the other end *semi-structured* as their structure may have desiccated over time [2]. The secondary issue of determining what latent decisions were made or not made in a workflow's execution, dependent on input, is analogous to that of determining the unobservable choices made by a human operator when implementing a procedural workflow, a problem examined in [43].

3.4 Thesis Approach

Based on these approaches, a dynamic trace approach is the most appropriate method for extracting structure from an ad-hoc workflow as it can capture a wide range of workflow behavior. The first step of the method proposed is to produce an *instrumentation* of the workflow. Workflow instrumentation is the process of adding elements (i.e., code instructions) to monitor and record behavior during workflow execution. Such an instrumentation produces a description of the workflow structure in terms of calls to tools, algorithms and methods, for a given execution. When events such as a system call, or accessing a file, occur in the instrumented workflow, they are recorded. The execution of the instrumented workflow produces an event log which can be analyzed to determine data dependencies. The second step is to analyze a *trace* produced by the execution of the instrumented workflow to construct a *dataflow graph* (i.e., a data dependency graph). The dataflow graph is created by analyzing file system changes in the context of the commands being executed. The result is a provenance graph but also provides an initial workflow structure.

The method in this thesis has several limitations in scope. **Python Workflows** - Workflows are assumed to be written in Python. According to statistics from GitHub, Python and Perl are used in the majority of hosted bioinformatics workflows or pipelines. However, other programming language (e.g., Perl) may benefit from dataflow construction. **Tool based** - The dataflow in a job is analyzed in terms of tool interaction with the filesystem. There are scientific (e.g., reproducibility) as well as practical (e.g., efficiency) reasons for such workflows to be preferred. However, workflows which do rely on external tools such as web services cannot be completely characterized. **Explicit tools** - Since trace depends on the manipulation of files by tools, logic internal which forms an implicit tool is not tracked. This can be seen as a problem of program slicing [92], where the goal is to determine exactly the part of the workflow program which corresponds to an internal tool.

In the next chapter which develops the method, the following terminology is used. A *workflow* is a program (i.e., Python script) which orchestrates a set of external

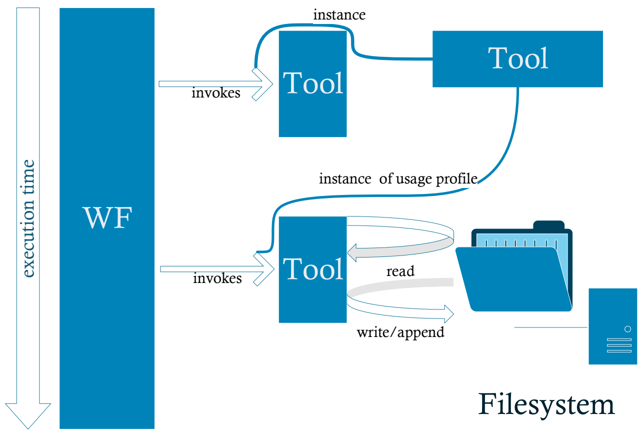

Figure 3.1: Overview of workflow execution process.

tools, managing their interdependencies and file dataflow, to produce an output with respect to some input. A workflow may be executed on any number of input files and systemically provides a set of output files for each. A *job* is the execution of a workflow on a given input. A *tool* is an executable program which takes input files and produces output files. A workflow executes tools when processing a job. Figure 3.1 illustrates a workflow interacting with a tool. Each time a workflow run executes a tool, it is an *invocation* of the tool. A tool may be invoked in various ways, based on a *usage profile* (i.e., the parameters that it is given). The tools and associated usage profiles are stored in a *resource library.* All *invocations* constitute an instance of a *usage profile* with respect to some specified input and output data. A *trace* is a representation of the execution of a workflow on a specific job, typically an event log. As a trace may be used to form a graph of file dependencies, we also use the term *concrete data dependency graph* (CDDG) for a dataflow graph constructed from a trace. A complementary term is *abstract data dependency graph* (ADDG) which, instead of designating concrete data flow, expresses abstractly how tools are linked. A job relies on information gathered from two sources: a workflow's *library* and the job's (specific) *input.* A *library* is a collection of information that is built into the workflow itself as a local resource. Rather than being specific to a job, libraries are part of workflows as standardized inputs. A job's *input* is the collection of files specific to its execution.

Chapter 4

DATAFLOW ANALYSIS

As discussed, ad-hoc workflows demonstrate a heterogeneity of workflow implementations, and runtime dynamics, whose capture must be performed in a generalized way to obtain real world applicability. This is addressed with a trace based method which captures a workflow through it's execution, including dataflow that emerges only at run-time, and while avoiding implementation intricacies.

In this chapter, a method is given to instrument an ad-hoc workflow, and analyze its log to determine tool dataflow. The mechanism to instrument a workflow is to provide a layer between a workflow and the language's libraries. When events such as a system call, or accessing a file, occur, they are recorded with the file system state. To execute an instrumented workflow, a valid input to the original workflow is required. A provenance graph representing event data dependencies is thus created by analyzing file system changes in the context of the commands being executed, and serves as an initial workflow structure.

4.1 Instrumentation

While the focus of this section is Python, this thesis provides a general mechanism for understanding workflows, The instrumentation captures events common to programming (e.g., system calls, file access), not events specific to Python.

At the first stage of the method, an instrumented workflow is constructed. The instrumented workflow is an *equivalent workflow* which produces a log, containing information on its interactions with the file system. This is accomplished by instrumenting the relevant calls. The instrumentation layer is transparent to the execution since it does not affect the dataflow and only monitors the relevant calls with a wrapper which intercepts functions when they are executed by the workflow's control flow. An overview of this is shown in Figure 4.1. The wrappers record function parameters, before returning control to the existing library.

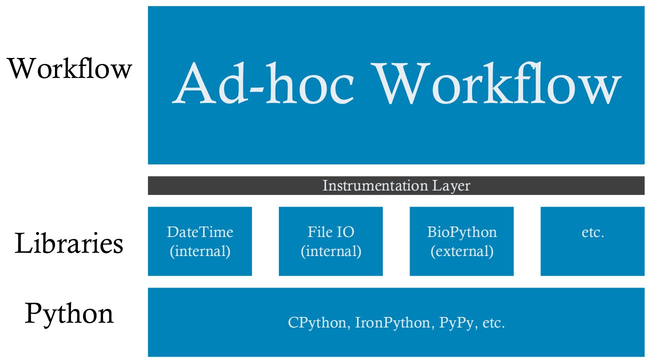

Figure 4.1: Overview of instrumentation layer between workflow and Python.

The instrumentation aims at recording *internal* and *external* events. The former represents the workflow accessing a file, while the latter represents a program invocation. Internal events are characterized by the workflow's use of file IO. External events are system calls, typically invoking a command, which embody a task (e.g., running a tool) or data operation (e.g., copying a file). For each event type, the log records information about the workflow and the filesystem. The filesystem is recorded as a snapshot of MD5 [88] hashes for each file in the workflow's folder. Workflows are identified by a path, which represents a folder containing the workflow's source code as well as its data. A specific path denotes the extent of the workflow and so limits the filesystem that must be analyzed. The representation of an entire region of the filesystem is required because an event's interaction with the filesystem cannot be determined solely from its parameters. The comparison of before and after snapshots of a filesystem (e.g., Figure 4.2) for each event captures the behavior of the system. For example: if a file exists prior to an event and is not changed, it is possible, but not certain, that a tool may have read it. If a file changes, then data were written in and the file was possibly read. If a file exists only in the after snapshot, and barring parallelism, it is likely an output of the invocation.

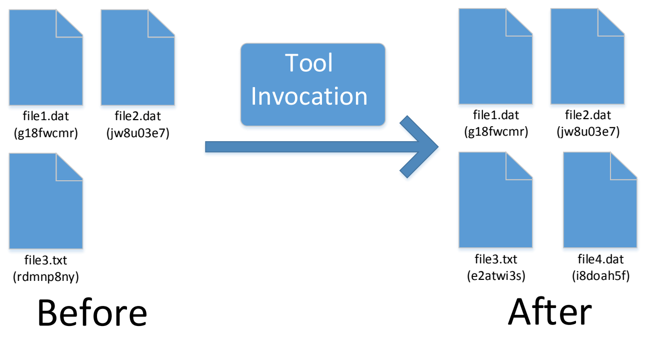

Figure 4.2: Example disk interaction from tool invocation.

Algorithm 1 gives a top level view of the instrumentation and trace process. Initially, users provide a name for the workflow, a path to a clean install of a workflow, a command to execute the workflow, and a list of input files. If the workflow has not been run before, then a backup of its install will be made, otherwise the backup is restored so instrumentation and execution is performed on a clean install. Next, the path is analyzed to find the scripts it involves, and each is checked for uses (imports) of libraries with relevant functions (see Subsection 4.1.1). For each use of a library, the appropriate wrapper (Subsection 4.1.2) is inserted into the workflow script. Once instrumentation has completed, the command provided by the user is executed to run the workflow and produce a trace. The trace is finally annotated with the input information, and is ready for dataflow construction (Section 4.2).

Algorithm 1 Instrument and trace a workflow.

```
1: procedure TRACE($name, path, cmd, inputs$)
2:     if not backed up then                                   ▷ prepare workflow folder
3:         make_backup($path$)
4:     else
5:         restore_back($path$)
6:     $scripts$ ←find_workflow_scripts($path$)                ▷ find scripts
7:     for $s \in scripts$ do
8:         $import\_lines$ = find_imports($s$)                 ▷ identify libraries
9:         for $line \in import\_lines$ do
10:            insert_hook($s, line$)                          ▷ insert library instrumentation
11:    run($path + cmd, name + ".log"$)                        ▷ execute instrumented workflow
12:    annotate_inputs($name + ".log", inputs$)                ▷ annotate trace
```

4.1.1 Finding Event Sources

A workflow may contain multiple files that need to be instrumented. There are two approaches to determining these files. The first is to identify all Python files in the path by selecting the appropriate extension. However, if the path contains source code files which are not a part of the workflow, e.g., tools, then the log will include events inside the tool(s) as well. Alternatively, scripts may be identified with *snakefood*, which creates a module dependency graph for a set of files. This provides a minimal set of files which typically form the workflow. Both approaches determine a list of files whose instrumentation produces valid logs but at different levels of abstraction. snakefood is the default approach. Each of the scripts identified is scanned to determine which libraries are being used. Python's built-in functionality is used to generate a program's abstract syntax trees (AST) for this task. The AST

Table 4.1: Instrumented standard library components.

Module	Type	Name	Event Type
builtin	function	open	internal
codecs	function	open	internal
os	function	system	external
shutil	function	mv	external
shutil	function	cp	external
subprocess	function	call	external
subprocess	function	check_call	external
subprocess	class	Popen	external
urllib	function	urlretrieve	external

is explored to find where a module (Python library) from Table 4.1 is loaded. Each of place is rewritten with instrumentation set immediately after the original module loading code. The `builtin` library is always loaded, and so is instrumented at the beginning of each file. The instrumentation module works by preserving access to the latest loaded function(s) and inserting wrapper function(s) in the runtime. Loading a module brings function names into the current scope, so each time it happens, the trace engine module which contains the instrumented functions must be loaded.

4.1.2 Wrapper Mechanisms

All external events record the filesystem's state for dataflow construction. When *os.system* or *subprocess.Popen* are called, the specific command issued to the system is also recorded. Unlike *os.system*, Popen includes functionality for streams and returns an object representing the ongoing execution of the command. The subprocess module also includes *call* and *check_call* which are degenerate cases of Popen.

In normal operation, Popen is designed to be non-blocking. To properly capture the filesystem after the execution of the tool, the instrumented Popen waits for the command's completion. Because it is possible that a Popen invocation completes at any time, this causes no side effects in the workflow. In addition to file system access, tools executed by Popen may interact with the standard streams: STDIN, STDOUT, and STDERR. It is a somewhat common pattern that scientific tools use these streams as their default means to input or output a single file. Prior to executing a command, STDIN is checked for presence of a file like object and its hash is recorded. Additionally, STDOUT or STDERR streams are checked for the presence of a file like object. If either is a file, it is flushed, hashed, and then replaced with an similar file object. This captures the file exactly as it was written by the command.

In contrast to external events, *_BUILTIN_.open* and *codecs.open*, act differently. Rather than executing a command, they produce a *file* object which is later manipulated by the workflow. When such an object is created, Python immediately loads the file. The instrumentation delays this operation and loads the file first for hashing. The instrumentation code also returns a modified *file* object (constructed with inheritance) to enable logging execution events. In addition to recording opening a file, the instrumentation also records when the file is closed. The process is analogous except that the hash is recorded after the library mechanism has closed it. Note that in some workflows, authors inadvertently leave out a final call to close. For these cases, the trace data will be saved when the deconstructor for the object is called.

Calls to certain functions in the *shutil* and *urllib* libraries are also wrapped. In *shutil* , this is *shutil.mv* and *shutil.cp*. This module provides shell-like functionality for manipulating functions (i.e., moving, copying). These operations are used by workflows to position files in directories prior to executing tools. The operations are captured as if they were *os.command* calls to the Linux command line tools of the same name. The same idea is applied to *urllib* where *urllib.urlretrieve* is reduced to *wget*. Although the action of these commands can be recorded, even without this information their dataflow will be discovered during analysis.

4.2 Dataflow Construction

The remainder of this thesis is independent of Python. Supposing an appropriate trace engine can be constructed, traces provided by another language (e.g., Perl) may be analyzed with the techniques described in the rest of this thesis.

The method for constructing a dataflow graph is now described. From a trace, a library of application resources and a data flow graph will be constructed. The resource library contains a list of tools that were invoked in the execution, usage profiles, and their expected interaction with the file system. In the graph, each node represents an event (likely an command) and each in-edge is a file dependency and each out-edge is a file produced by that event.

The graph contains two types of nodes: *external* and *internal*. Each node contains an `IORecord` produced from analyzing the associated event, and a reference to the application library. An `IORecord` is the encapsulation of how a particular event interacted with the file system. Each `IORecord` is generated by the application of a usage pattern to an invocation. Each usage profile has a number of *ports*, which represent specific files or folders that an invocation relies upon. For invocations, ports become bound to values (file or folder names). An `IORecord` tracks the values of the input and output ports and the files (via path and hash) which those values relate. In the case of files used by an invocation but not related to a specific port, a general pool of file access is also maintained. An `IORecord` optionally contains zero or one input and output streams.

As a prerequisite to constructing the dataflow graph, the initial input files must be known. A dataflow graph is initiated with three special nodes, which are taken to be external events. The graph starts with a node called `Source` which acts a tool with no inputs which produces the initial input files. The second node is the `Library` node which acts as a tool which produces any file existing prior to the workflow's execution which is not an input. Lastly, a node called `Sink` acts as a tool whose input dependency is every file which exists at the end of a workflow's execution. This ensures that the graph details any output file the workflow generates. Although the user may only be interested in some files, this subset may not be clearly delineated. Hence, all files available at the end of execution should be considered.

4.2.1 Event Refactoring

The goal of this step is to transform external events so each corresponds to the execution of exactly one program. A system command typically involves the execution of a program but may display additional behavior from shell syntax. These events are thus restructured into a more simple form that accounts for the action of these features. The trace is analyzed for three cases: 1) Auxiliary commands; 2) Commands communicating by pipe operator; and 3) GNU `parallel`.

Some commands can be wrapped within an application in such way that the wrapper does not effect the execution of the command or workflow. For example, programs like `time` display the run time of a command. The pipe operator is used to pass streams between programs. Many scientific tools utilize stream data therefore the pipe operator provides a simple mechanism to compose programs by passing streams from one to another. When a command contains a pipe command, the command is the split into two commands. Some workflows authors utilize GNU `parallel` to enable a degree of parallelism. These commands are expanded into multiple concrete commands that can be executed in parallel.

4.2.2 Command Analysis

Initially, each event is analyzed to identify how it interacts with the file system. An *internal* event denotes the direct interaction of a workflow with the filesystem. An internal event can be either be the opening of a file, indicating the possible dependency of workflow execution on a specific file, or the closing of a file, indicating the workflow making a file available for tools to depend upon. When a file is read (resp. written), it is captured as a command with the file being opened (resp. closed) as its sole dependency. *External* events denote when the workflow is making a system call. These are typically the invocations of scientific or data preparation tools. While any execution of an application is an invocation, different invocations of an application may involve different parameters. This is encapsulated by the idea of usage profiles.

Applications are uniquely defined by their path in the file system. For each external event, we check if the application being executed has already been seen. Existing usage profiles are then used to analyze the state of the workflow. `IORecord`s for events are constructed using the usage profiles as follows. The local file system is examined to determine the files relevant to the event which satisfy the ports and patterns found in the usage profile. A usage profile defines not only an abstract system command used to execute a tool, but expected input and output dependencies for the tool. As a first step, the method computes the added, removed, and changed files for the snapshots around an event. These are the basis to determine if a file is an input or an output. There are three types of dependencies to capture between files. A *direct* file dependency occurs when an invocation names exactly the file or folder being used.

A file dependency is *indirect* when an invocation names a substring of a file or folder being used. The third category includes the cases when a file or folder changes between invocation filesystem snapshots and some general data usage pattern is able to capture it. The latter are called *implicit* dependencies. These three types also form a hierarchy of heuristics use. A particular indirect heuristic is only applied when its use would cause no conflict (overlap) with the initial direct ports discovered or another indirect heuristic being applied. A particular implicit heuristic is only applied when it would cause no conflict with the direct or indirect ports.

For each dependency type, there are several IO identification heuristics. The first step consists of the identification of the portions of the active command which may correspond to a port; these are *direct* dependencies. This is done by tokenizing the command on spaces and then trying to mount each token as either a file or folder in the filesystem known from the snapshot. Each matching token that matches is greedily assumed to be a port. Each of token which matches in the filesystem are then classified. If a file exists before a tool is executed, and it's state does not change after the invocation, then it is an input. If a file exists only after the invocation it is a output. In cases where additional information is known, for instance the >> operator in Unix, a file may be labeled as being appended. Folders are analyzed in a similar way. Any file within an input folder may be an input, so each is internally marked as a dependency. There are two types of output folders: pure, no files exist within it prior to invocation, or impure, some files exist within it prior to invocation but are not changed. A command *pattern* is then created by replacing each port substring with a symbolic port name and number. The files and folders found in this step make up the direct dependencies. Next, the method checks for indirect access with two heuristics. The first indirect heuristic detects grouped files. This pattern is typically seen in programs which utilize a named database shared between a collection of files.

For each of the file ports, the local file system is examined for files which contain its concrete name as a prefix. Matching files are determined to be indirectly used by the invocation, as named following the port. This has the effect of a expanding a single file port into a set of files. This is separate from a folder dependency since the exact name of each file can be determined from the name bound to the port. The second indirect heuristic rule enables the detection of collections of folders. This is seen in tools which perform some division operation on a singular input. This rule is only active when more than one folder containing files has been created. For each folder, we assume each may be a element of a set of folders. Each folder is assumed to have some pattern. A pattern is guessed from a sample folder, where local names (e.g., folder or file names) are abstracted away. The potential pattern is then checked against the other folders. Whichever sample folder provides the best coverage of folders (i.e., captures the entire folder collection) while not overlapping with other heuristics is selected as the indirect folder collection rule. Last, the method checks for *implicit* outputs. This heuristic assumes that only the active event is being executed by the workflow. When GNU `parallel` is being used, this heuristic must be disallowed. For any file which appears in the working directory of the active program, it is assumed that each file is an output. Such a file is assumed to have a static name which will be the same for any invocation.

After analyzing each event, the method has produced a set of `IORecord`s to construct the dataflow graph as explained in Section 4.2.3. Because the process for generating a `IORecord` is greedy, its correctness is partially checked by tracking a virtual representation of the filesystem. Any file being read must be available in the virtual filesystem. Provided this is the case, files being written are then updated in the virtual filesystem and are checked against the log. In this way, the cumulative changes to file system are simulated.

4.2.3 Graph Construction

Once `IORecord`s have been constructed, the dataflow graph may be constructed. A node is created for each event in addition to three special nodes (i.e., `Library`, `Source`, and `Sink`). Nodes reference their usage profile, input and output ports, as well as possible streams (i.e., STDIN, STDOUT). When tools use files provided by another tool, edges will connect the nodes, and be annotated with the file's name, hash, and IO port names. For a technical description of the format, see Section 5.1.

Using a nested loop, the inputs of each event's `IORecord` are compared to the output of the other `IORecord`s to match files read with those written. Due to parallelism, the algorithm permits any tool in the sequence to produce output for any other tool. This is validated by ensuring that use of a file is not ambiguous (i.e., only one file matches). There are two criteria for matching a file: hash and path. When a file is used by an invocation both must match, whereas only the hash must match for a stream. For both types of inputs (file and stream), a list of candidate source nodes is gathered. A node is a candidate if it provides the data's hash as either a file or an output stream. For streams, which lack the filename criteria, the method defines a priority order for choosing a candidate node. In general external events are preferred over internal events and file sources are preferred over stream sources.

The graph is stored in GraphML [21], and annotated with display information to enable user understanding. For visualization, Graphviz [39] is used in hierarchical layout mode. Nodes which are *external events* are shown as circles while nodes which are *internal events* are shown as rectangles, and each is labeled with respect to the program names and an unique event ID. Since the three special nodes are essentially tools, they are shown as circles with special labels. There are two types of *internal events*: reads and writes. Respectively, they are labeled `WF_INT_READ` and

`WF_INT_WRITE`. Directed edges go from data producer to data consumer. Each edge is labeled with the corresponding filename and represents a file dependency.

Every node in a CDDG can be understood with the same meaning: a command which had access to the filesystem. However, each node may be produced by mechanisms with slightly different meanings. If the Source has no outputs, then none of the input files provided to the trace engine were used during its execution. Likely, the program analyzed is not a file-based workflow or the input files were incorrectly identified by the user. *External events* may have input dependencies and/or output dependencies. When a tool has no input dependencies, the tool is decoupled from the workflow's input and likely an independent data source (e.g. downloads a file). If the output of a tool is not used by a later command, it is likely a final product of the workflow, and the graph will omit an edge. When a tool has no output dependencies (not simply omitted), the tool does not create data. This can occur when a tool fails or has an unseen action (e.g., uploads a file). A `WF_INT_READ` node represents the workflow reading a file for internal purposes, while `WF_INT_WRITE` nodes represent writing. Neither includes opposite dependencies since workflow state is not tracked.

Since control flow is not tracked, the context of files accessed by *internal events* is unknown. The use of the data read is unknown, and the source of the data written is unknown. However, internal events can be implicitly related. That is, a file is opened, the workflow performs some operation, f, and then writes the file back. This produces a `WF_INT_READ` and a `WF_INT_WRITE` node which together form an *implicit tool* (performing f) in the workflow that could not be captured. See Figure 4.3 for a comparison. *Implicit tools* are discussed further in Chapter 6, with SPROUTS as context. Note that when a workflow has many internal events, it is not trivially clear which nodes form an implicit tool.

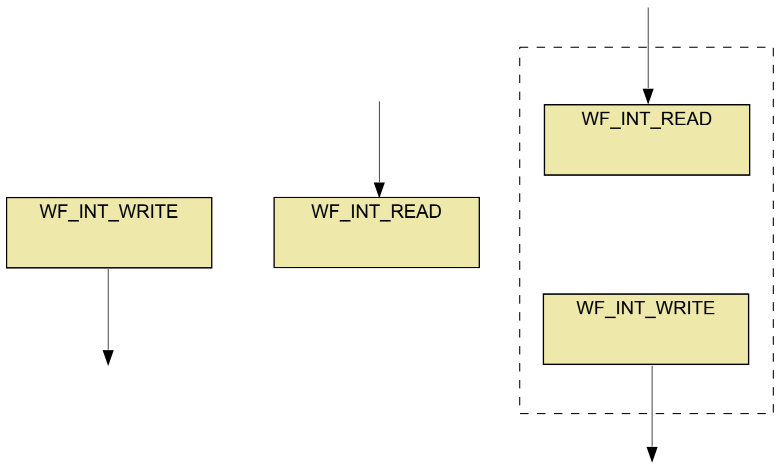

Figure 4.3: Left: standalone `WF_INT_WRITE` node. Middle: standalone `WF_INT_READ` node. Right: example of implicit tool formed by two internal events.

4.3 Protein Synthesis Example

The trace method is now illustrated with a bioinformatics workflow which simulates *Protein Synthesis*. The input is a file of one or more DNA sequences stored in FASTA format. The output is a folder called `aa` which contains separate FASTA files for each sequence that was found in the input. Each input sequence undergoes the protein synthesis process of transcription followed by translation. The full source code for this workflow is shown in Figure 4.4. This workflow relies on three external tools (`split_multifasta.py`, `dna2rna.py`, and `rna2aa.py`) to split a FASTA file, perform transcription, and perform translation, respectively.

The instrumentation method was applied to the source code shown in Figure 4.4. Since the main source code file does not `import` any modules, no other files will be instrumented. The instrumented workflow can be seen in Figure 4.5, where two places have been instrumented (blue text). The first instrumentation is performed at the top to monitor the built-in `open` function. The second is performed after the `os` module is imported, to monitor `os.command`. Since these are the only module imports, they are the only places instrumented. This workflow was run on an input FASTA

Figure 4.4: Source code for *Protein Synthesis* workflow.

```
import sys, os

input_name = sys.argv[1]

cmd = "split_multifasta.py␣-input␣" + input_name + "␣-outfolder␣dna"
os.system(cmd)

files = [f for f in os.listdir("./dna")]
for fn in files:
  cmd = "dna2rna.py␣-inputfile␣dna/" + fn +"␣-outputfile␣rna/" + fn
  os.system(cmd)

files = [f for f in os.listdir("./rna")]
for fn in files:
  cmd = "rna2aa.py␣-inputfile␣rna/" + fn +"␣-outputfile␣aa/" + fn
  os.system(cmd)
```

(`seq_col.fa`) containing three DNA sequences: 3OE0, 1ASU, and 1BNI. Table 4.2 shows the external events recorded at execution time. All events are external. The File Changes column gives a summary of how the file system changed during the execution of each command. A plus symbol indicates an added file.

The log indicates that the execution of the workflow triggered seven external events (see Table 4.3). The first was executing `split_multifasta.py` on the input file. During this command, three new files were created in the `dna` subfolder. The tool `dna2rna.py` was then executed three times. Each time it read one of the files in the `dna` subfolder and produced a corresponding file in the `rna` subfolder. Last, the tool `rna2aa.py` tool was executed three times. Each time it read one of the files in the `rna` subfolder and produced a corresponding file in the `aa` subfolder.

Figure 4.5: Instrumented source code for *Protein Synthesis* workflow.

```
import sys
sys.path.append("/home/ruben/Desktop/wf-trace/")
import trace_engine
open = trace_engine.hook_open

import sys, os
import sys
sys.path.append("/home/ruben/Desktop/wf-trace/")
import trace_engine
os.system = trace_engine.hook_system

input_name = sys.argv[1]

cmd = "split_multifasta.py␣-input␣" + input_name + "␣-outfolder␣dna"
os.system(cmd)

files = [f for f in os.listdir("./dna")]
for fn in files:
  cmd = "dna2rna.py␣-inputfile␣dna/" + fn +"␣-outputfile␣rna/" + fn
  os.system(cmd)

files = [f for f in os.listdir("./rna")]
for fn in files:
  cmd = "rna2aa.py␣-inputfile␣rna/" + fn +"␣-outputfile␣aa/" + fn
  os.system(cmd)
```

The log was analyzed to produce the dataflow graph displayed in Figure 4.6. Three applications were discovered with one usage profile, each with a number of instances. Each usage profile corresponds to one of the system calls, while the instances cor-

Table 4.2: Summarized event log.

Source	ID	Command	File Changes
main.py	1	split_multifasta.py -input seq.fa -outfolder dna	+ dna/1BNI.fa
			+ dna/1ASU.fa
			+ dna/3OE0.fa
main.py	2	dna2rna.py -in dna/1BNI.fa -out rna/1BNI.fa	+ rna/1BNI.fa
main.py	3	dna2rna.py -in dna/1ASU.fa -out rna/1ASU.fa	+ rna/1ASU.fa
main.py	4	dna2rna.py -in dna/3OE0.fa -out rna/3OE0.fa	+ rna/3OE0.fa
main.py	5	rna2aa.py -in rna/1BNI.fa -out aa/1BNI.fa	+ aa/1BNI.fa
main.py	6	rna2aa.py -in rna/1ASU.fa -out aa/1ASU.fa	+ aa/1ASU.fa
main.py	7	rna2aa.py -in rna/3OE0.fa -out aa/3OE0.fa	+ aa/3OE0.fa

respond to each time a particular call was executed. The graph contains one node for each of the external events that were logged as well as three special nodes. The `Source` produces the initial input, `seq_col.fa`, while the `Library` node is unused. The dataflow graph represents the *structure* of the workflow, as a dataflow view of the workflow. The user can observe that a repetitive process is being applied to a elements created by some process. This is not immediately clear from the serial implementation, and implies the workflow could be parallelized.

Table 4.3: Usage profiles and invocations discovered.

Application	Profile	Command
split_multifasta.py	1	-input `INPUT0` -outfolder `FOLDER_OUT0`
	(instance)	-input seq_col.fa -outfolder dna
dna2rna.py	2	-inputfile `INPUT0` -outputfile `OUTPUT0`
	(instance)	-inputfile dna/TEST1.fa -outputfile rna/TEST1.fa
	(instance)	-inputfile dna/TEST2.fa -outputfile rna/TEST2.fa
	(instance)	-inputfile dna/TEST3.fa -outputfile rna/TEST3.fa
rna2aa.py	3	-inputfile `INPUT0` -outputfile `OUTPUT0`
	(instance)	-inputfile rna/TEST1.fa -outputfile aa/TEST1.fa
	(instance)	-inputfile rna/TEST2.fa -outputfile aa/TEST1.fa
	(instance)	-inputfile rna/TEST3.fa -outputfile aa/TEST1.fa

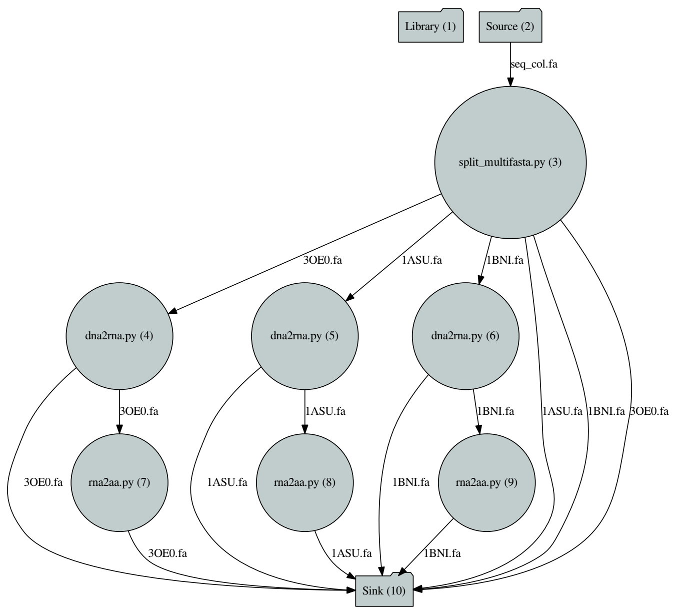

Figure 4.6: Visualization of the dataflow graph.

Chapter 5

DATAFLOW ABSTRACTION

In Chapter 4, the extraction of data dependencies from a trace was presented. This can be represented as a graph with edges for data dependencies and nodes for commands. A *concrete data dependency* is the use of a specific file by a specific command in a trace. When every edge in a graph is a concrete data dependency, it is called a *concrete graph*. A concrete graph does not represent a generalized form of the workflow - it captures provenance. The next task is to identify nodes with the same character (i.e. invocation and dataflow) and combine them. As scientific workflow tend not use looping structures, this is typically seen as parallel execution structures. For example, see the concrete graph given on the left in Figure 5.1. This figure shows the same two node linear process being applied to each output of a predecessor node. Thus, it includes three repetitive regions. Given multiple repetitive regions, they may be systemically combined into a single process (right of figure) with input-gathering connections to regions which provide instances of input.

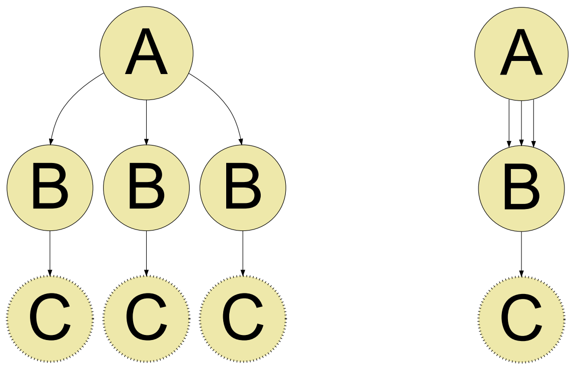

Figure 5.1: Left: sample CDDG with repetition. Right: repetition reduced to region.

Continuing the aim of extracting scientifically relevant workflow structure, this chapter discusses building an Abstract Data Dependency Graph (ADDG) where equivalent nodes have been combined into *collection regions*. A *collection region* indicates that a set of inputs has the same process applied on a partition of its input. In the previous figure, the bottom six nodes which would be replaced by two nodes linear that process each output from the initial node. The method given performs the iterative merging of equivalent nodes as a concrete graph is explored. At present, equivalence focuses on the usage profiles found during dataflow construction. However, this is not a requirement. Setting multiple commands to the same profile may make them equivalent to the algorithm and thus mergeable. See Subsection 7.1.3 for an example.

Section 5.1 describes the format of the graph for concrete dataflow. The algorithm for discovering repetition of parallel regions is given in Section 5.3. Section 5.4 gives an example of the algorithm applied to the protein synthesis workflow. A method for further simplifying a ADDG is given in Section 5.5

5.1 Concrete Data Dependency Graph

A *Concrete Data Dependency Graph* (CDDG) is a digraph that represents dependencies between commands during a job. Nodes are called *concrete commands* and represent a trace event. During dataflow construction, each event is analyzed to determine how it refers to files or folders being read or written. Each of these dependencies is called a port. Usage profiles thus define a list of input and output ports. Edges are called *concrete edges* and represent a dependency between ports on a pair of concrete commands. Whenever a concrete edge connects to a concrete command, the port that the edge uses must belong to the usage profile of the concrete command. When multiple edges exist with the same port name, it implies that the port accepts a collection (e.g., a folder, or an implicit grouping). By its construction, every CDDG has at least one node with in-degree zero (the source). For CDDGs, the definition is given is not rigorous. Since CDDGs are mostly the product of heuristics, enforcing rigorous formation reduces real-world applicability.

A CDDG contains one type of node: *concrete commands*. Each node, n, contains one attribute: $n.profile$, an identifier for the usage profile used to create that node. For a node n, we use the notation inedges(n) to represent the set of edges incoming to n and outedges(n) to represent the set of edges out going from n.

Definition 1 (cddg nodes) *Let n be a CDDG node if*

- $n.profile \in Z$.

A CDDG contains one type of edge: a file. For an edge e, the notation *e.src* represents node it came from, and *e.dst*, the node it enters. On either end, a file may be bound to a file port, or, may be the element of a folder port. Each edge, e, contains four attributes: *e.srcport*, the port which produced the file, and *e.dstport*, a port which reads the file, and two for referencing the file (*e.hash* and $e.filepath$).

Definition 2 (cddg edges) *Let e be a CDDG edge if*

- *e.srcport and e.dstport are port names.*
- *e.src and e.src are CDDG or ADDG nodes.*
- $n.filepath$ *is a string*
- *n.hash is a string*

Each port name is composed of a port type from Table 5.1, or 5.2, composed with an ID number. For example, a tool which takes two input files and produces

Table 5.1: Types of input ports.

Class	Class	Description
INPUT	file	A file input dependency.
APPEND	file	An existing file which may be read and written.
FOLDER_IN	folder	A folder input dependency.
FOLDER_IMPURE	folder	A folder dependency which may be read or written.
STDIN	stream	The standard input stream.

Table 5.2: Types of output ports.

Class	Class	Description
OUTPUT	file	A file output dependency.
APPEND	file	An existing file which may be read and written.
FOLDER_OUT	folder	A folder output dependency.
FOLDER_IMPURE	folder	A folder dependency which may be read or written.
FOLDER_OUT_SCATTERN	special	A pattern of files which is repeated ('scattered') across a number of output folders.
OUTPUT_INDIRECT	file	A indirectly named file output dependency.
STDOUT	stream	The standard output stream.

one output folder would have input ports $INPUT0, INPUT1$ and output ports $FOLDER\_OUT0$.

When a port type is listed with class *file*, each name and port ID combination is bound to exactly one file. When a port type is *folder*, a name and port ID combination is bound to some number of files, together representing a folder. During abstraction, the dependency becomes the folder rather than its contents.

There are two exceptions. First, $FOLDER\_OUT\_SCATTERN$ does not follow the pattern for output folders, it creates a number of folders which are not known statically. In a CDDG, each scattered folder will be port labeled from $SCATTER1$ to *SCATTERnth* while the tool itself will have only a single $SCATTERN$ to represent the scatter operation. The second exception is that a file input port may have multiple files bound to it, representing a file matching pattern (e.g., a star).

5.1.1 Command Library and Equivalence

As introduced in Chapter 4, when a trace is analyzed, a resource library of all executable tools, and how they were executed, is created. The library is used to define concrete command equivalence. Recall that an usage profile contains information on the inputs, outputs, and parameters, used by a program. They represent a tool being executed in a specific manner - the same profile implies identical process. Thus, profiles are a basic level of *process equivalence*. However, profile equivalence is not necessary for commands to be semantically equivalent; different tools may perform the same semantic task.

For the purposes of Section 5.3, the resource library is a list of usage profiles such that: 1) profiles can be identified, and 2) define a list of ports. A usage profile also includes a program's location and parameters but this information is only needed if a tool is to be executed.

Definition 3 (process equivalent) *Let n_1, n_2 be CDDG or ADDG nodes. n_1 and n_2 are process equivalent iff $n_1.profile$=$n_2.profile$.*

This notion of equivalence lies in transformation. Even if two nodes are equivalent in isolation, the data they produce may be transformed in different ways. This second type of equivalence, *flow equivalence*, is further discussed in Section 5.3. This is similar to how Starlinger et al. [94] identified two aspects of workflow similarity: single modules (i.e., tools), and whole workflows.

5.2 Abstract Data Dependency Graph

A CDDG refined to contain repetition in specific regions is called a *Abstract Data Dependency Graph* (ADDG). The nodes of an ADDG are called *abstract command* nodes. All the structure characteristics of a CDDG are used by ADDGs, i.e., nodes and edges, but ADDGs introduce additional elements (collection operators). Each ADDG corresponds to a CDDG, and is a subgraph (excluding operators) of it.

There are two types of nodes in the graph: abstract commands (rectangles) and collection operators (see below). There are two subtypes of collection operators: *collectors* (inverted triangles) and *dispensers* (triangles). Operators are special nodes, which indicate dataflow over a collection (or repetition), and whose bounds denote a collection region. A collection contains some number of elements which are unordered, have an identical representation, and serve as inputs to a collection region. An ADDG contains two types of edge: files and folder. These naturally correspond with the concepts of a file, and a set of files, which exist in CDDGs. Edges may be connected between ports of the same class, or, represent the construction or destruction of a set of files when the class changes. A port may reference either a file or a folder.

Nodes in an ADDG have several attributes:

- (abstract command) abstracted: links to the concrete node that the abstract node replaced.

- (collectors) portmap: records the connectivity between collector ports, and ports on nodes inside.
- (dispensers) portmap: analogous to above.

Edges in an ADDG have several attributes:

- (optional) collected_index: When an edge enters a collector, this attribute contains a number (index) which groups a set of edges satisfying the input of the collector.
- (optional) dispensed_index: When an edge exits a dispenser, this attribute contains a number (index) which groups a set of edges satisfying the output of the dispenser.

5.2.1 Collection Operators

Collection regions are defined by operator pairs which indicate collection of inputs, and dispensing of outputs. The subgraph between a collector and dispenser represents a data process applied repetitively on a partition of inputs. The partition on the edges into or out of a collection region are formed as nodes are merged.

A *collector* operator represents the formation of a collection, where each element is a valid input produced from various nodes. Each value of *collected_index* defines edges making up an element, with each element containing inputs for the ports used by the collector. A collector has out-edges to match the abstract command(s) which operates on the elements. A collector node, n, has one attribute $portmap[n]$ which records node connectivity between the operator's ports and the inside nodes.

A *dispenser* operator represents dispensing data contained as an element of a collection. Each value of dispensed_index defines the edges making up an element, with each element containing outputs for the ports provided by the dispenser. A dispenser has in-edges from the abstract command(s) which operate on the elements the dispenser forms. These edges are determined by output ports available on the abstract command, rather than the output edges from the command, which may be omitted if the command produces a final output. A dispenser node, n, has one attribute $portmap[n]$, same as a collector.

5.3 Abstraction Algorithm

This section discusses an algorithm for identifying and combining repetition in a dataflow graph. The algorithm functions by searching for a pair of equivalent nodes, and replacing them with a collection region. Once a collection region exists, the algorithm folds other equivalent nodes into it. As the algorithm advances along the dataflow, collection regions can occur in sequence, and provided their cardinalities are the same, they are merged to represent sequential repetition. Two words are commonly used to describe the actions of the algorithm: *seed* and *solute*. A seed is set of nodes (concrete or abstract) that can be compared to a set of concrete nodes, a solute. If the two sets of nodes are equivalent, then the solute nodes will be merged into a collection region equivalent with the seed. Hence the name given to the algorithm: crystallize.

The pseudocode for the top level mechanism is given in Algorithm 2. The inputs are a CDDG, and its resource library. Note that during abstraction, the algorithm acts in place on the graph, and so the graph may contain both CDDG and ADDG elements. By the termination, all CDDG nodes will be transformed to ADDG elements. Algorithm 2 makes use of six functions:

- **is_collector(n)** takes a node, and returns true if it is a collector.
- **gen_solutes**($G, concretes, seed$) takes a set of concrete nodes, a seed, and re-

turns a list of all solutes that may merge with the seed. To reduce the number of solutes, each is required to contain nodes process equivalent to the seed.

- **try_seed(**$G, P, solutes, seed\_nodes$**)** takes a set of solutes, and a seed. It compares the seed with each solute, if there are matches, then the nodes are combined. See Algorithm 3.
- **create_abstract_command(**n**)** takes a concrete command node, and returns an abstract command node with identical node attributes and a reference to the concrete node.
- **transfer_edges(n1, n2)** takes two nodes, and moves edges from $n1$ to $n2$.
- **collectors_simplify_inputs(G)** takes an ADDG, and returns it with file edges simplified to folder edges. Checks collectors for folder input ports and replaces file edges with a single folder edge.

The algorithm maintains a set of *leading nodes*, comprised of all the concrete nodes that have no concrete successors, as a list of nodes which may be merged. As nodes are merged, this set is refreshed, similar to a topological sort. The *leading nodes* perform a partition between the abstracted nodes, and the remaining concrete nodes which cannot be abstracted. The main loop implements three ways to perform abstractions. 1) If collectors have already been formed, their predecessors hint at repetition. Thus, the collectors predecessors are partitioned on index with the first set used as a seed, and rest as solutes. try_seed is then run on these sets. 2) For each leading node, try_seed is run with each as a seed, thus merging repetition within the leading nodes. 3) If no nodes were abstracted, then a leading node is selected and is transformed into an abstract node. Note that since the CDDG is acyclic, and each loop iteration abstracts at least one node, this algorithm always terminates. After the main loop, the algorithm cleans up the graph by simplifying edges between nodes producing folders and collectors.

Algorithm 2 Main crystallization algorithm.

```
1: procedure CRYSTALLIZE($G, P$)                    ▷ a CDDG and its usage profiles.
2:     for $cn \in G.nodes$ do
3:         $cn.abstracted \leftarrow null$
4:     $leading \leftarrow \{n|n \in G.nodes \wedge |successors(n)| = 0\}$
5:     $abstract \leftarrow \emptyset$
6:     while $leading \neq \emptyset$ do
7:         for $an \in abstract$ do      ▷ try extending collectors with upward elements
8:             if is_collector($an$) then
9:                 $likely\_solutes \leftarrow \{nodes|$predecessors of a collected_index of $an\}$
10:                $likely\_solutes \leftarrow \{s|s \in likely\_solutes \wedge s \subseteq leading\}$
11:                if $|likely\_solutes| > 1$ then
12:                    $collector = try\_seed(G, P, likely\_solutes[1:], likely\_solutes[0])$
13:        for $n \in leading$ do                    ▷ discover repetition in leading nodes
14:            if $abstracted[n] = null$ then
15:                $collector = try\_seed(G, P, gen\_solutes(G, leading, \{n\}), \{n\})$
16:        $abstract \leftarrow abstract \cup$ any new collectors
17:        if did not abstract node then                 ▷ ensure a node is abstracted
18:            $an \leftarrow$create_abstract_command($G, leading.pop()$)
19:            transfer_edges($cn, an$)
20:            $cn.abstracted \leftarrow an$
21:        $concrete \leftarrow \{n|n \in N[G] \wedge \neg abstracted[n] \wedge$ n has no concrete successors $\}$
22:    collectors_simplify_inputs(G)
```

try_seed (Algorithm 3) finds the *solutes* of the equivalence class defined by flow equivalence with *seed* and merges them into a collection region. Equivalence is determined by the existence of an *equivalence map*; a bijective function between two sets of flow equivalent nodes such that each pair is also process equivalent. It uses three functions:

- **seek_flow_eq(**$G, nodes1, nodes2$**)** takes two sets of nodes, and returns an equivalence map if it exists. See Algorithm 4.
- **create_collection_region(**$G, P, nodes$**)** takes a set of nodes, and returns a collector node. The collector is followed by abstract nodes derived from *nodes*, and then a dispenser. The collector has input ports corresponding (but renumbered) to *nodes*, the dispenser has output ports the same as *nodes*, and corresponding internal edges. When internal edges are added, but did not exist in the CDDG, they are called *artificial.* Each collector and dispenser records the connectivity between the nodes it contains and the ports it exhibits as a *portmap*, a list of 3-tuples containing a exposed port name, a node within the collection region, and the name of the port on the node.
- **merge_into_collector(**$G, nodes, collector, eq\_map$**)** takes a set of nodes, a collector, and an equivalence map between nodes to be merged. It moves the in- and out-edges between from *nodes* to the abstract nodes inside the collection region.

The core of this function is a loop checking flow equivalence between *seed* and each *solute*. When they are flow equivalent, they need to be merged. If no merge has taken place, equivalence is compared between the seed and the solute. If the seed was already merged with another solute, then successors of the collector (nodes equivalent to the seed) and the solute are compared. If an equivalence map exists, then a collector will be introduced into G if needed, and the solute will be merged into the collector.

Algorithm 3 Try merging seed with some solute.

```
1: procedure TRY_SEED($G, P, solutes, seed$) ▷ a CDDG, its resource profiles, set of solutes, and a seed.
2:     $collector \leftarrow null$
3:     for $solute \in solutes$ do
4:         $eq\_map \leftarrow null$
5:         if not $collector$ then                                ▷ not collected
6:             $eq\_map \leftarrow$ seek_flow_eq($G, seed, solute$)
7:         else                               ▷ collected, compare with collector
8:             $eq\_map \leftarrow$ seek_flow_eq($G, successors(collector), solute$)
9:         if $eq\_map$ then
10:            if not $collector$ then
11:                $collector, collector\_eq\_map$ = create_collection_region($G, P, seed$)
12:                merge_into_collector($G, seed, collector, collector\_eq\_map$)
13:                $eq\_map$ = composition of $eq\_map$ and $collector\_eq\_map$
14:            merge_into_collector($G, solute, collector, eq\_map$)
```

seek_flow_eq (Algorithm 4) tries to find a equivalence map between *nodes*1 and *nodes*2. It uses two functions:

- **are_commands_process_eq(**n_1, n_2**)** takes two nodes, and returns true if they are process equivalent.

- **are_nodes_flow_eq(**$G, nodes1, nodes2$**)** takes two equivalent concrete nodes, and returns true if they are flow equivalent. See Algorithm 5.

Two sets of nodes are flow equivalent if there exists an exchange of nodes that does not change the set's output. This function uses nested loops to compare each pair of possible matches. Each $n \in nodes1$ will be mapped to exactly one $n \in nodes2$; although the correspondence may not be unique. For each comparison, the set of *friend* nodes is computed; it comprises every node in one of the input sets except the node being computed. This represents the *dataflow context* of the comparison. For nodes to be matched, they must be both process and flow equivalent.

Dataflow context represents the downward dependency of some nodes on a child. Consider Figure 5.2. It is clear that the two bottom nodes, labeled E, are equivalent since they have same process and no later dataflow. The next step in the algorithm would be to examine the newly exposed leading nodes: left C, left D, right C, right D. If these nodes were compared only on their process equivalence and downstream dataflow, then left C and right C would be equivalent, as they have the same process and provide input to E (likewise for the D nodes). However, this is incorrect. The two Cs are not be equivalent because the output of the left CD pair gives output to an E, and the right CD pair gives input to another E. The C and D nodes cannot be mixed - other nodes give them context - so the pairing must be maintained. This is properly represented as the left CD nodes being equivalent to the right CD nodes, i.e., they must be treated as unit.

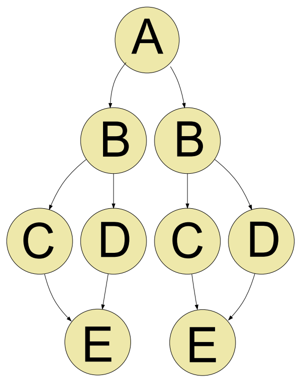

Figure 5.2: Dataflow context example.

Algorithm 4 Checks if two sets of nodes are flow equivalent.

```
1: procedure SEEK_FLOW_EQ($G, nodes1, nodes2$)    ▷ a CDDG, two sets of nodes.
2:     $matches$ = list the size of $nodes1$
3:     for $n1 \in nodes1$ do
4:         for $n2 \in nodes2$ do
5:             $friends1$ = nodes1.remove($n1$)
6:             $friends2$ = nodes2.remove($n2$)
7:             if are_commands_process_eq($G, n1, n2$) then
8:                 if are_nodes_flow_eq($G, n1, n2, friends1, friends2$) then
9:                     if n2 not already matched then
10:                        $matches[n1] = n2$
11:        if not $matches[n1]$ then
12:            return $null$
13:    $eq\_map$ = matches between nodes1 and nodes2
14:    return $eq\_map$
```

are_nodes_flow_eq (Algorithm 3) takes two equivalent concrete command nodes, two sets of friend nodes, and returns true if they are flow equivalent. That is, if their output dependencies could be exchanged without change in the output of successors. Friend nodes make up the data context of node - they are parallel nodes which produce the data within the local collection region. It uses four function:

- **is_abstract(**n**)** takes a node, and returns true if it is an abstract node.
- **get_outedges(**n**)** returns the edges comprising the non-collection data flow out of n. When a node is not adjacent to a dispenser, then its outedges are used. Otherwise, a sample of the dispensed edges is selected, remapped to real port names, and then returned. See Algorithm 5.

- **get_collector_inedges_by_index**($n, index$) takes a collector, and returns all inedges with collected_index = $index$.

are_nodes_flow_eq first retrieves the outedges for each node it is comparing. The edges are checked to determine if their the successor is a collector. If the nodes have friends, they can only be equivalent if they belong to the same index. Recall that each index may contain other nodes processing other data - this gives the nodes context. If this context is different, such nodes cannot be equivalent. Next, all edges must enter a existing collection region, this recursively perseveres any previous repetition. Lastly, all of the outedges between the nodes must be matched such that they come from the same port, and go to the same port. Since previous caller has already verified that $n1$ and $n2$ are process equivalent, and the previous statement ensures they go to the same subgraph, thus, these nodes are flow equivalent.

Algorithm 5 Checks if two concrete command nodes are flow equivalent.

```
1: procedure GET_OUTEDGES($n, dispenser$)
2:     $edges = out\_edges(n)$
3:     $\triangleright$ if n is next to dispenser, use its edges instead
4:     if $\forall e \in edges, e.dst.type =$ DISPENSER then
5:         $dis = n\_out[0].dst$
6:         $dis\_out =$ out_edges($dis$)
7:         if $|dis\_out| > 0$ then
8:             $index =$ a dispensed_index from $dis\_out$
9:             $used\_ports = \{pm.newport | pm \in dis.portmap \land pm.abstract = n\}$
10:            $edges = \{e | e \in edges \land e.dispensed\_index = index\}$
11:            $edges = \{e | e \in n1\_out \land e.srcport \in used\_ports\}$
12:            for $e \in edges$ do
13:                for $averted \in portmap[dis]$ do
14:                    if $averted.abstract = n \land e.srcport = averted.newport$ then
15:                        $e.srcport = averted.realport$
16:        else
17:            $edges = dis\_out$
18:    return $edges$
```

```
19: procedure ARE_NODES_FLOW_EQ($G, n1, n2, n1\_friends = \emptyset, n2\_friends = \emptyset$) ▷ a CDDG, two nodes, two sets of nodes.
20:     $n1\_out \leftarrow$ get_outedges($n1$)
21:     $n2\_out \leftarrow$ get_outedges($n2$)
22:     ▷ If either node has external nodes, they must have same index.
23:     if $n1\_out$ has edge with collected_index then
24:         $n1\_index = n1\_out[0].collected\_index$
25:         $n1\_index\_edges =$ get_collector_inedges_by_index($G, n1\_out.dst, n1\_index$)
26:         $n1\_external = \{e.src | e \in n1_index\_edges \wedge\ e.src \neq n \wedge e.src \notin n\_friends\}$
27:         $n2\_index = n2\_out[0].collected\_index$
28:         $n2\_index\_edges =$ get_collector_inedges_by_index($G, n2\_out.dst, n2\_index$)
29:         $n2\_external = \{e.src | e \in index\_edges \wedge\ e.src \neq n \wedge e.src \notin n\_friends\}$
30:         if $|n1\_external| > 0 \wedge |n2\_external| > 0$ then
31:             if $n2\_index \neq n1\_index$ then
32:                 return $false$
33:     if $\exists e \in (n1\_out \cup n2\_out)$ s.t. $e.dst.type \neq$ COLLECTOR then
34:         return $false$
35:     $matches =$ list the size of $n1\_out$
36:     for $e1 \in n1\_out$ do
37:         for $e2 \in n2\_out$ do
38:             if $e1.srcport = e2.srcport \wedge e2.dstport = e2.dstport$ then
39:                 if e2 not already matched then
40:                     $matches[e1] = e2$
41:         if not $matches[i]$ then
42:             return $False$
43:     return $True$
```

5.4 Protein Synthesis Example

In Chapter 4, a simple workflow for simulating protein synthesis was given. In this section, that example is continued to illustrate abstraction. The CDDG produced by dataflow construction was shown in Figure 5.3. This graph differs from the Chapter 4 as it lacks nodes for the library (unused) and sink (only for sanity checking). The workflow contains four profiles: 2 (source node), 4 (use of `split_fasta.py`), 5(use of `dna2rna.py`), and 6 (use of `rna2aa.py`). The edges of the graph have been labeled with the ports used by the concrete dependencies.

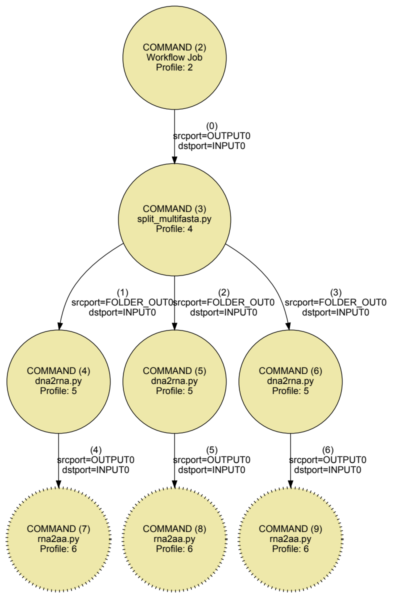

Figure 5.3: Raw CDDG for protein synthesis.

First Iteration: Initially, the algorithm selects the leaf nodes (7, 8, 9) as the leading nodes. These are the nodes which have a dashed outline in the figure. All three node use the same usage profile and will be merged. The first of these nodes (7) is selected as a seed. The node is then compared against the rest of the concrete node set. Initially, this is node 8. Node 7 is compared with node 8, to determine if they are equivalent. Since they have the same profile they are process equivalent.

Since they have no outputs, they are trivially flow equivalent. Thus, they may be combined. From node 7, a collection region is constructed. This is composed of a collector (11), a command (10), and a dispenser (12). Note that the edge between 10 and 12 is dashed. This represents a port used by the command profile but whose output was not used in the CDDG - an artificial edge. The collection region is then merged with the main graph. The edge input to 7 is directed to the new collector and given index 1. The edge input to 8 under goes the same process. The result is shown in Figure 5.4.

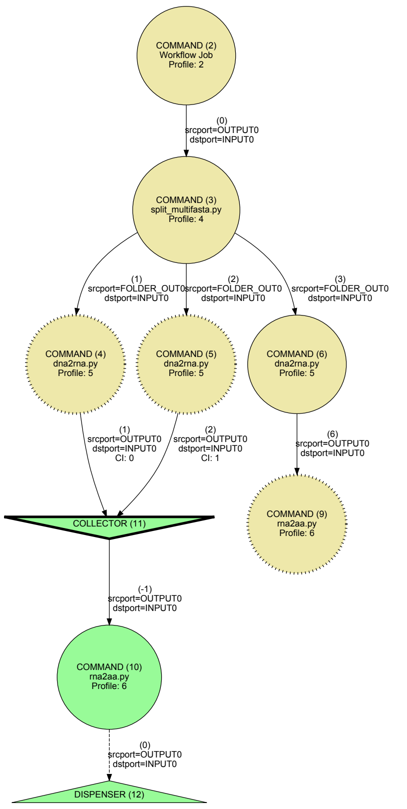

Figure 5.4: Protein synthesis CDDG during second iteration, after merging first two nodes.

Next, node 9 is examined. This node is also equivalent with node 7 in the same way that node 8 was. However, node 7 has already been abstracted. Thus, node 9's input edge is simply merged as another element (index 3) of the collector. This completes the first iteration as there are no more concrete nodes to examine. The result is shown in Figure 5.5.

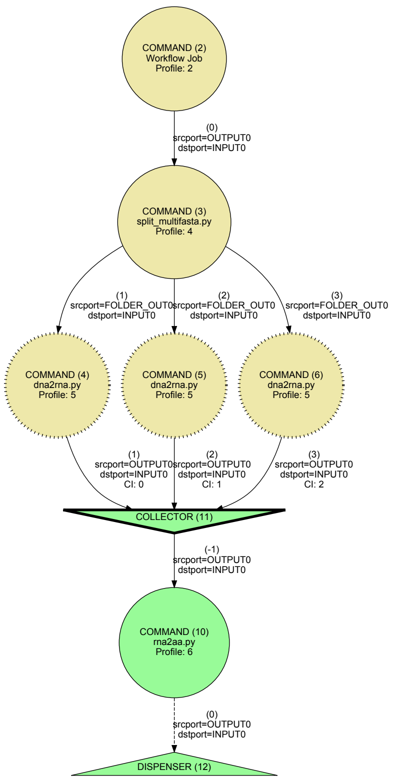

Figure 5.5: Protein synthesis CDDG after one iteration.

Second Iteration: The leading nodes are updated to include all nodes which are concrete and have only abstract successors (4, 5, 6). The first of these nodes (4) is selected as a seed. The node is then compared against the rest of the concrete node set. Initially, this is node 5. The nodes are compared to determine if they are equivalent. Since they have the same profile they are process equivalent. To be flow equivalent nodes must have outputs which are exchanged. Both nodes output into a collector, each as a different input. Since a collector region applies the same process to each input index, they are naturally flow equivalent. Thus, they may be merged. The process of creating a collector region and attaching the input dependencies is now repeated for node 6. The result is shown in Figure 5.6.

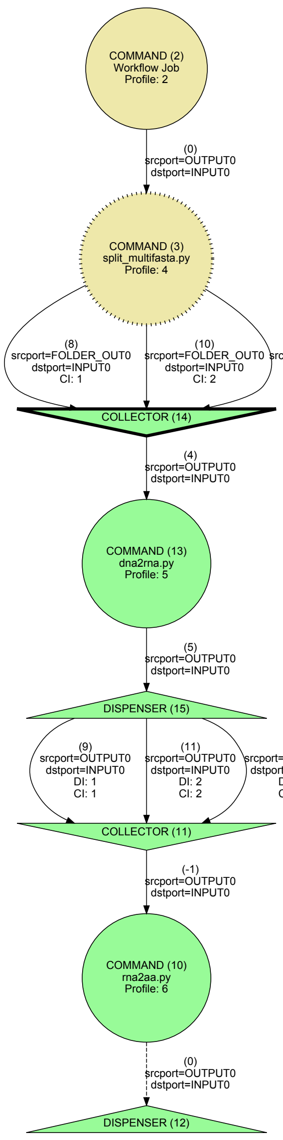

Figure 5.6: Protein synthesis CDDG during second iteration, after second abstraction.

The resulting graph has a collector and dispenser pair with matching cardinality. In this case, the information about the dataflow between collector and dispenser is redundant. Since each index is being processed the same, and the collection regions can be combined. This is directed by selecting the first index and using it to map between the dispenser's predecessor and the collectors' successor. Then the extra collector and dispenser nodes are removed. The complete iteration is shown in Figure 5.7.

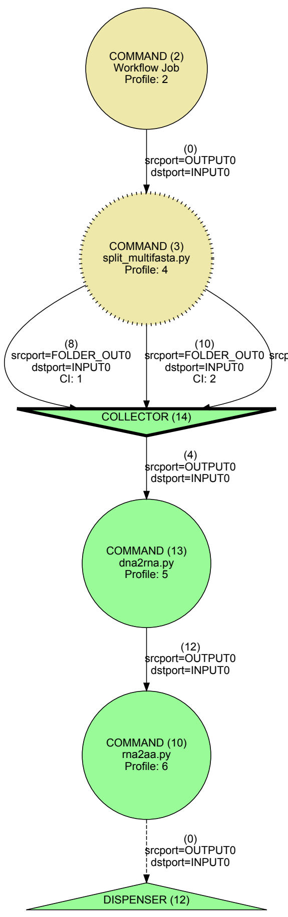

Figure 5.7: Protein synthesis CDDG after two iterations.

Two more iterations of the algorithm are applied at this point but are not shown. Both iterations simply mark the predecessor node of the already simplified region as being abstracted. Since there is no more repetition, no nodes can be combined to form a new collection region.

Final Result: As a post process, the edges between command 15 and collector 14 are removed. This happens because command 15 provides a folder, and each element of the folder makes up an input to the collector. Thus, collector is refactored to take a folder as input with the meaning that it must process each element it contains. The final result is shown in Figure 5.8.

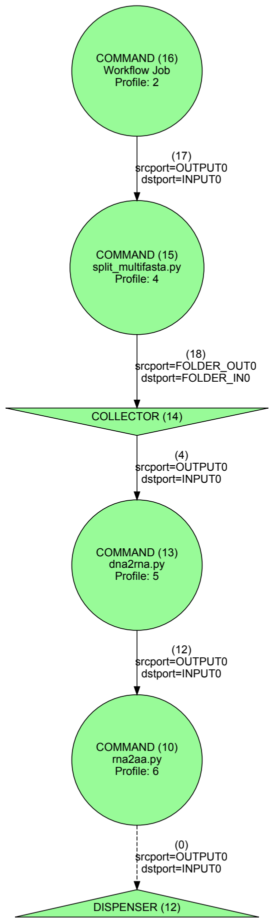

Figure 5.8: ADDG for protein synthesis.

5.5 Skeletonization

Previously, simplification has preserved the exact execution and dataflow structure. Skeletonization removes such details in favor of enabling user understanding.

The abstraction process focuses on identifying repetitive regions and replacing them with simpler graph elements that designate the repetition. Since this process preserves the exact structure of the execution, dataflow is recorded in a higher fidelity than is necessary for understanding its semantics. These additions obfuscate the overall design of the workflow. As a preliminary effort, skeletonizing the ADDG is proposed. Skeletonization consists of two steps: 1) Removing parallel edges. 2)

Removing collection operators. The dataflow in 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. To remove collection operations, each node in the graph is examined. If that node is a collection operator, then it is removed and dependencies added between its predecessors and successors. This preserves the connectedness of the graph. While skeletonization may be performed on either a CDDG or ADDG, a skeletonized graph may not adhere to the format of either.

The clarity that this process brings is demonstrated in Chapter 6 when it is shown with a ADDG representation in several workflows. Skeletonization is discussed again in Section 7.1.2 which discusses which future work based on formalism.

Chapter 6

RESULTS

In addition to *Protein Synthesis*, a simple workflow to apply the protein synthesis process to a number of sequences, the method was applied to a number of workflows on GitHub. Evaluation has focused on four GitHub workflows (see Subsection 2.3.2): hybseqpipeline [57], a sequence assembly workflow for Illumina reads, Inmembrane [84] which checks if a bacterial protein codes for a surface-exposed region, miR-PREFeR [66] which predicts plant microRNA from RNA sequences, and pycoevol [73] which analyzes the coevolution of a pair of proteins. In addition, the method is evaluated on SPROUTS [68], a true legacy workflow not intended for public release, which examines the impact of point mutations on protein stability. As discussed in Subsection 2.3.2, selection of test workflows was retrieved from GitHub. While not representative of all workflows, the selection was meant minimize bias in an effort to assemble a (small) random sample. All of the workflows except SPROUTS included a README file which discussed the purpose of the workflow and the tools utilized. These workflows contain more dynamic language features, and data driven configuration, than their functionality would suggest. In particular, inmembrane and SPROUTS would be difficult to analyze with a static approach. For the first three workflows and SPROUTS, the method produces a complete dataflow graph. The remaining workflows demonstrate some of the limitations discussed in Subsection 3.4. In both workflows, the method identifies that dataflow is incomplete but not the cause.

The instrumentation program, instrumented workflow(s), and dataflow construction algorithm were executed on CPython 2.7.6 and Xubuntu 14.04. Performance evaluations were performed on a virtual machine with an Intel 2500K (at 4.5Ghz), and 4GB RAM. Workflows were installed using the instructions specified in their respective readme files, and were executed in a virtualized filesystem for reproducibility.

6.1 Workflow Implementations

The scale 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 given in the workflow description (D). The characteristics of the test workflows are listed in Table 6.1.

Table 6.1: Characteristics of test workflows.

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+set	3	text
SPROUTS	3438	951	8	1	graph

These workflows use external tools to carry out their analysis. This is problematic for existing provenance methods (e.g., ProvenanceCurious, StarFlow, noWorkflow) because they track direct file access, and function calls, but neglect external tools. In this work, tool execution is tracked to determine how they interact with the filesystem. The six workflows invoked a total of 21 external tools, including: BLAST, CAP3, DFIRE, dna2rna.py. EXONERATE, FoldX, HMMER3 I-Mutant 2, I-Mutant 3, LipoP, MAFFT, MEMSAT3, MIR3, MUpro, rna2aa.py, Samtools, SignalP, split_multifasta.py, TMHMM, Velvet, and ViennaRNA. Although the tools listed in documentation are shown for validation, the method does not require this prior knowledge. The method discovers tools, stores them as resources, and assigns them nodes in the dependency graphs. Note that the number of tools used by miR-PREFeR is not precise because it uses samtools, a collection of tools.

Comments (measued with PyLint [1]) covered from 7% to 22% of the code base(s). While the GitHub workflows included a README file, with Pycoevol also providing a conceptual overview, only SPROUTS gave a top-level graph of the interactions between tools. Comments range from well documented (e.g., descriptions for every function) in Inmembrane and Pycoevol, to descriptive in miR-PREFeR and HybSeqPipeline, extremely sparse in Protein Synthesis, and incomplete in SPROUTS. Like many in house legacy workflows, SPROUTS and Protein Synthesis were not meant to be publicly released, thus explaining their poorly documented code. When a workflow is aimed at public release, such as those retrieved from GitHub, documentation is more likely accurate and informative.

The input size (column I in Table 6.1) 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. The method is expected to capture such hidden parallelism, and show how performance can be improved significantly when the input dataset contains several instances.

6.2 Inputs and Executions

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

[1] www.pylint.org

input for Inmembrane is one or more bacterial genes in FASTA and a parameters file. The input of Pycoevol is a pair of proteins as PDB IDs. The input for miR-PREFeR is one or more small RNA-Seq data samples of the same species as SAM files. The input of SPROUTS is a protein as a PDB ID. The method was run on the workflows using their respective sample data: a set of 3 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.

The 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 for 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, the method produces only 55 nodes with edges for 3,713 files on Inmembrane although it is 759% longer than HybSeqPipeline. The results for SPROUTS display a similar scale with 60 nodes and 3,719 edges. Both inmembrane and SPROUTS contain a majority of external events with a few internal events for data preparation.

The graph results are reported in Table 6.2. 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 found. 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 skeleton.

For all workflows but Pycoevol, the method discovers all the tools given in the workflow's description. It also discovers tools (e.g., data preparation scripts), typically

Table 6.2: Results on test workflows.

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

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). Pycoevol and miR-PREFeR demonstrate the limits of the implementation: specific libraries instrumented and assumptions about file system use. Pycoevol makes use of web services to download data files, instead of local applications. miR-PREFeR uses the assigned temporary folder of the computer executing it for processing. Since 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.

6.2.1 Inmembrane

Inmembrane is a workflow to determine whether a bacterial protein sequence may include coding for a surface-exposed region. Inmembrane is documented by a readme which discusses the workflow's purpose and usage, as well as a publication describing the workflow's science and 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) [38], SignalP uses neural networks trained on separate sets of prokaryotic and eukaryotic sequences and an hidden Markov model algorithm to identify signal peptides and their cleavage sites [85], LipoP predicts lipoproteins out of signal peptides [87], and TMHMM predicts transmembrane helices in proteins [93]. Each tool is documented with version information and a web link. The results from each tool are used to produce a summary spreadsheet and citations list. The readme describes the main input format and the format of the parameters file.

Inmembrane exhibits dynamic run-time behavior in two ways. First, by requiring a parameters file which is stored as a source file that must be `eval`ed to inject parameters into the run-time. Second, after the parameters have been dynamically loaded, one of them is used to select a source folder (representing a scientific protocol) containing workflow scripts which are `eval`ed to enact the workflow in the run-time. If a static approach (e.g., ProvenanceCurious) was applied to inmembrane, static analysis would fail to detect tools, or file access, as they are defined at run-time using a parameter value. A dynamic approach (e.g., StarFlow, noWorkflow) is needed to capture the tools and writing the summary files. None of the existing provenance methods are able to capture this workflow's tool use although StarFlow and noWorkflow would detect the run time file access.

The input to this workflow is a set of bacterial genes and a parameters file which indicates if it is a gram- or gram+ strain. Although the repository contains five sample files, users are left to construct parameter files. Inmembrane was executed on input file `AE004092.fasta`, with the gram+ option, and produced a graph composed of 55 nodes with edges for 3,713 files. The concrete and abstract data dependency graphs are respectively displayed in Figures 6.1 and 6.2. 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. In Figure 6.1, 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 the tools it uses are installed.) The second layer of the graph displays 17 tool nodes (that act on a file). Note that the 8^{th} node represents 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, instances of `which`. The graph overall indicates that the workflow is parallel. Once the abstraction algorithm is applied, the simplification is dramatic and 11 similar nodes - instances of a single tool TMHMM - are combined as illustrated in Figure 6.2. 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. The skeletonized result is shown in Figure 6.3, which simplifies the abstract data dependency graph further.

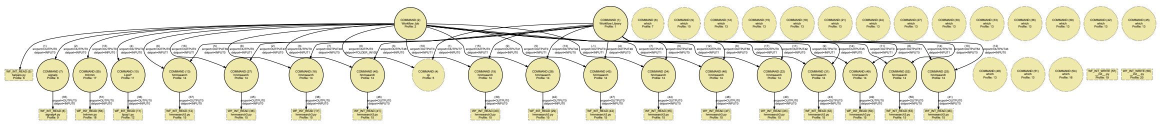
Figure 6.1: Inmembrane dataflow graph.

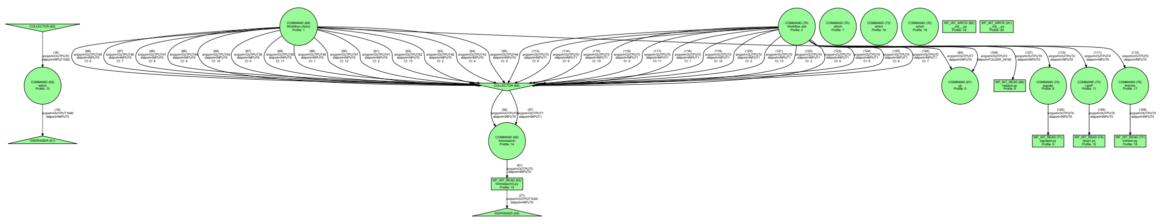
Figure 6.2: Abstract data dependency graph for Inmembrane.

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, to both 80 and 81. 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 citation list, saved in event in 81.

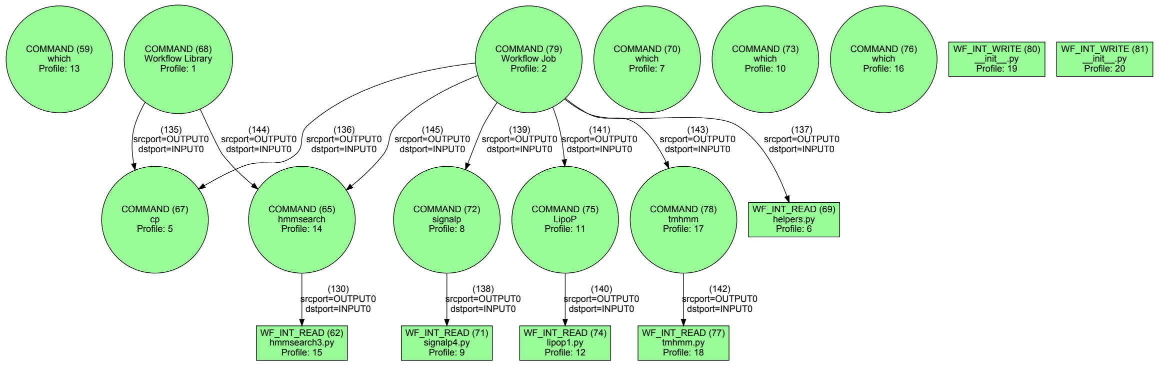

Figure 6.3: Skeletonized abstract data dependency graph for Inmembrane.

6.2.2 SPROUTS

SPROUTS is an ad-hoc workflows developed by Arizona State University with international collaboration. Like many legacy scientific workflows it is partially described in published articles. SPROUTS performs predictions using a suite of eight computational tools (i.e., MUPro, DFIRE, I-Mutant (4 versions), FOldX, and MIR) to examine the impact of point mutations on protein stability. An online database is used to store the data that is generated. The first seven tools produce identical information describing stability but in a variety of formats. The last tool, MIR, provides a linear description of a protein in terms of interaction density. The input to the workflow is either a PDB ID, used to download remote input files, or user provided input files. The results from each tool are parsed by scripts to produce a graph (by `parser_X.py`, where X is a tool name) and an uploadable SQL file. The SQL files are created by a script called `insert_result.py` while the script `insert_protein.py` uploads the protein sequence to the SPROUTS database to register the protein. A final script called `populateDB.py` uploads the result of `insert_result.py` to the SPROUTS database. SPROUTS exhibits dynamic behavior by loading a configuration file to populate a list of tools to execute. If a static approach (e.g., Provenance-Curious) was applied to inmembrane, it would be able to detect the data preparation steps, which are statically defined, as well as MIR, but would find only a tool invocation point without knowledge of the seven tools that will be executed. A dynamic approach (e.g., StarFlow, noWorkflow) is needed to capture the tools. None of the existing provenance methods are able to capture this workflow due to tools but all would detect the file access. SPROUTS was run with the PDB code 1LFC. The dataflow graph for this execution, displayed in Figure 6.4, seems to indicate that SPROUTS is a parallel workflow with some disconnected regions.

Recall that `WF_INT_READ` nodes represent events when the workflow is reading a file whereas `WF_INT_WRITE` nodes denote the workflow saving data in a file. In some occurrences, the two events are implicitly related, thats is a file is opened, the workflow performs some function and then writes the results in the file. This acts as an *implicit tool*: a part of 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, the implementation 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.

The result of processing implicit tools is illustrated in Figure 6.5. Post-processing for implicit tools consists in merging a read event with a write event. 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. Note that the 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 two other 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, with 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 then is 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, thus its output does not reach the command `populateDB.py`.

The abstraction method is applied to Figure 6.4 to produce the abstract data dependency graph displayed in Figure 6.6. 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, but, since the command parameters are different in each instance, they cannot be combined. The skeleton of the abstract data dependency graph is displayed in Figure 6.7.

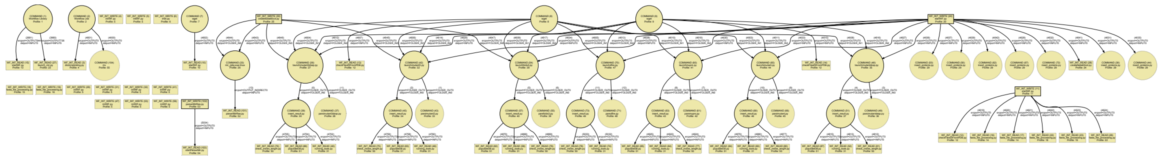
Figure 6.4: SPROUTS dataflow graph.

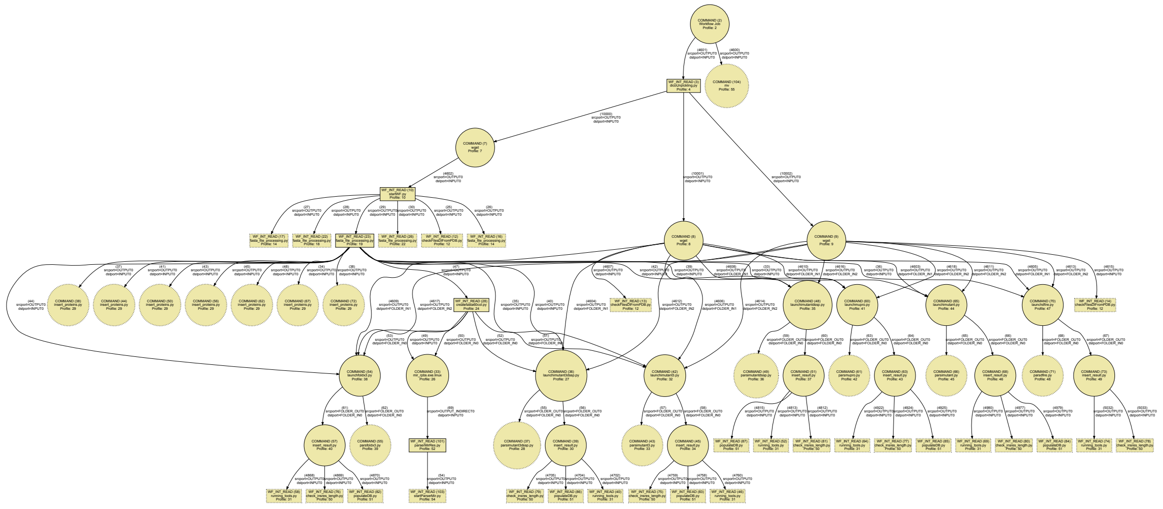
Figure 6.5: SPROUTS dataflow graph after creating implicit tools.

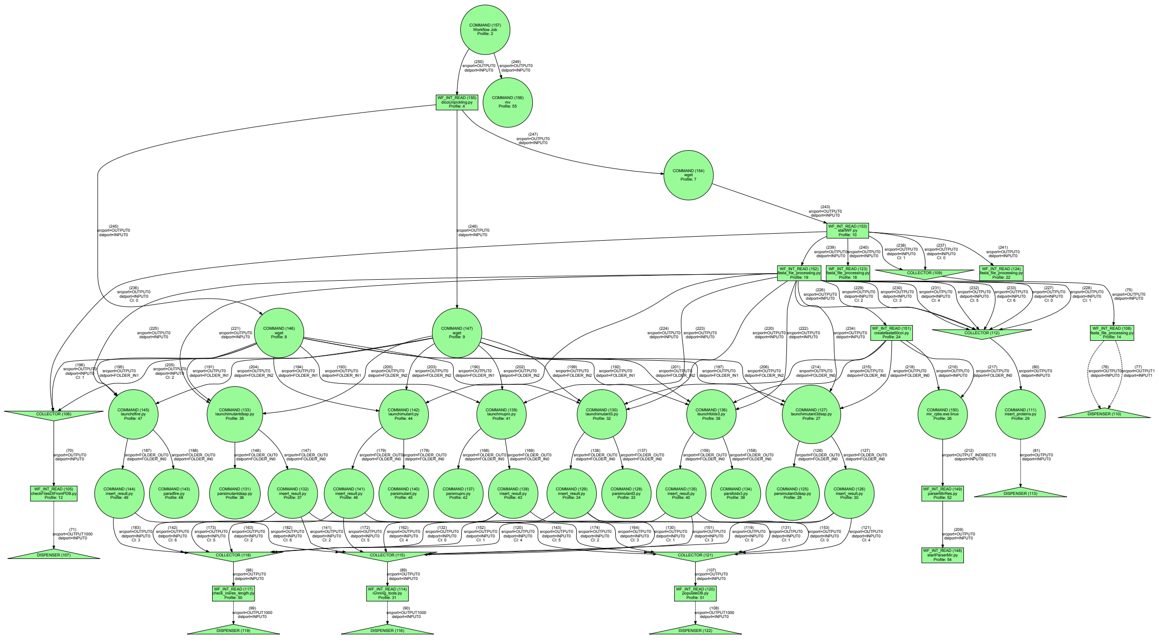
Figure 6.6: SPROUTS abstract data dependency graph.

6.3 Method Performance

As expected, instrumenting a workflow impacts its execution performance. The execution times of the original and instrumented workflows on the inputs described

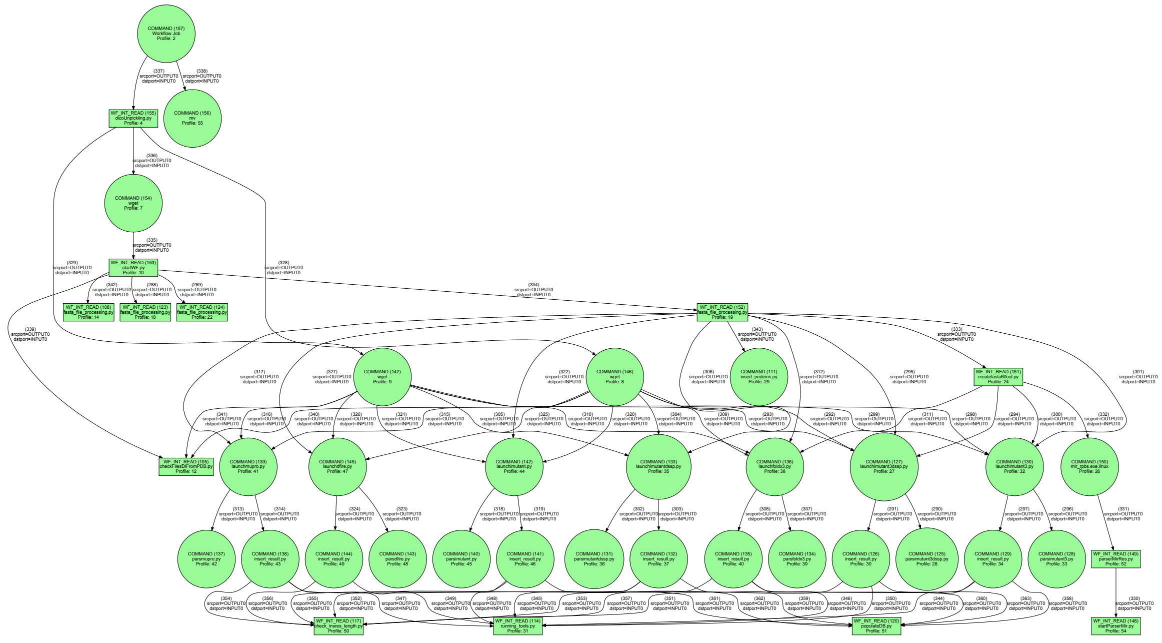

Figure 6.7: Skeletonized abstract dependency graph for SPROUTS.

in Table 6.1 are listed in columns I and D of Table 6.3 (in seconds). The impact of instrumentation in Protein Synthesis is negligible since 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% slower than the initial workflow, respectively. SPROUTS runs 12.1% slower. Note that SPROUTS uses an internal timer to determine when it may exit - this obscures the actual run time. For Pycoevol, the instrumented workflow is 51.8% slower than its non-instrumented version while the instrumented version of miR-PREFeR runs 815.3% slower than before instrumentation. The slower 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, since an instrumented workflow is only needed to generate traces, not as a permanent refactoring, these execution times are acceptable.

In the method, only the time to produce the instrumented workflow is independent

Table 6.3: 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

of an input. These times are shown in the column I of Table 6.4. The dataflow and abstraction times scale based on the size of the input. The third column (D) gives the time to build the dataflow graph from the trace. The time for abstraction algorithm to identify repetition, and simplify it, is given in column A. The abstraction is quick, in fact, the times in column A also include time for writing DOT files at each stage. For Pycoevol and miR-PREFeR, the dataflow construction is slowed since the algorithm constructs placeholder edge(s) and node(s) for missing data, and reports debug information. For these two workflows, the abstraction algorithm was not applied since its result would inherit the missing dataflow. The sum of columns D and A correspond to the total time required to convert the trace to an ADDG.

The quality of skeletonization can be judged by examining its ability to capture tools as nodes while omitting non-tool nodes. The goal is to capture all tools contained by a workflow as nodes of the skeleton. Based on Table 6.1 (column T), and the workflow documentation, each skeleton was checked for expected tools. The first four workflows were confirmed to contain at least one node corresponding to each expected tool. The second two workflows were omitted since they were not skeletonized. The method should also produce a skeleton where are all nodes correspond to tools.

Table 6.4: 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

Comparing Table 6.1 (column T) which lists the number of tools per workflow and Table 6.2 (column A) which lists the number of nodes in the workflow skeletons, it is seen that no skeleton exactly matches the number of tools. However, since graphs contain a Source node to designate input, they contain at least one more node than needed by the tools. Thus, Protein Synthesis is an ideal case, with 3 expected tools and 4 nodes in the skeleton. The other workflows do not give such precise results: HybSeqPipeline contains 750% more nodes, Inmembrane 350% more, and SPROUTS 387.5% more. In contrast, the concrete data dependency graphs include even more nodes: Protein Synthesis with 175% more nodes, HybSeqPipeline with 12700% more, Inmembrame with 1275% more, and SPROUTS with 650% more. Thus, while the current implementation does not identify only scientific tools, it significantly reduces the raw number of nodes.

Chapter 7

CONCLUSION

This thesis presented a method for discovering the tool-based structure of a ad-hoc Python workflow. The method starts with instrumenting an ad-hoc workflow written in Python to produce a log, and then using the log to determine file dependencies to build a dataflow graph by analyzing file system changes. This approach enables the characterization of workflows with behavior that only emerges at run-time, either from implementation specifics or language features, precluding the use of existing static analysis methods. A graph representing the file dependencies for events is created by analyzing file system changes. The data dependency graph generated by this process contains repetition based on the workflow's structure. This repetition is removed by a process in which identical commands are combined. Gradually, regions of dataflow which operate on different sets of inputs are formed. This simplification enables the viewer to better understand the structure of the workflow. The collected repetition provides a view of the workflow which can be used as a base for a WFMS or other formal representation. The method was applied to a example synthetic workflow for illustration purposes and a set of real world workflows to demonstrate relevance. These workflows demonstrated behavior via configuration files (SPROUTS) or language dynamics (inmembrane) which can only be captured at run-time. For three of these workflows, the method fully captures the file data flow. For the other two, some of system's limitations produced incomplete dataflow. Moving forward, the aim is to finish extracting the semantic workflow organization. This will take the form of iterative refinement of a datagraph of workflow into a higher level form where scientific tasks are explicitly defined. The process for abstracting repetition may be leveraged to achieve this.

7.1 Future Work

Future work includes addressing the limitations mentioned in the paper to increase the accuracy and generality of the method. As discussed in Chapter 2, process mining techniques address the issue of eluding a workflow's control flow from multiple executions, and are complementary to the dataflow approach taken in this work. Process mining can extend this work by recovering an overall workflow structure with multiple executions. The dataflow knowledge discovered by this work gives the exact dependencies between workflow elements in that overall structure, thus characterizing both control- and data-flow in a workflow. Although the trace method in Chapter 4 correctly captures the test workflows, it is possible that more exotic tools produce filesystem snapshots which do not capture their action unambiguously. The present technique for determining a program's interaction with the file system is greedy. A backtracking mechanism would enable safer application of more aggressive heuristics for determining file access and is a natural extension. The algorithm could be designed to optimize the rules for a usage profile over the set of invocations while preserving the filesystem consistency checks. For dataflow issues from threaded workflows, one can also consider forcing a workflow to run on single physical processor, to allow to examining the system as if only one tool is running. In addition to improving the dataflow construction scheme, three more areas demand improvement: 1) Detecting implicit tools, 2) Skeletonization, and 3) Extracting Semantics

7.1.1 Implicit Tools

One of the limitations that was illustrated with SPROUTS, is the method's dependance on explicit representation of tools. As mentioned in Section 6.2.2, the dataflow of the SPROUTS workflow is interrupted at points due to workflow functions which open a file, do some computation, and then save the result. Such a function forms an implicit tool - a region of the workflow's code base which makes up a tool. Since the trace depends on the manipulation of files by tools, it does not represent logic internal to workflow. Although a workflow may read or write a file, the method cannot determine the manipulation applied or its dependencies. To rectify this, the regions in a workflow between internal read and write events could be examined to determine if they are independent (e.g., taking only parameters) from the rest of the workflow. The internal events making such regions could be refactored into a proper tool representation. This can be seen as a problem of program slicing [92], 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 program which corresponds to an internal tool that manipulates files.

7.1.2 ProtocolDB Ecosystem Integration

In Section 5.5, a preliminary method to simplify an ADDG was given. It reduced the complexity of the workflow's graph by eliminating redundant edges and unnecessary nodes. The aim was to move closer to the semantics of the workflow. The semantics of a workflow are its key structure. However, the skeletonization process exists in this work as the final end product, requiring users to manually investigate a graph (instead of using a query system), and not providing facilities to execute the workflow. A key extension is to provide a complete system for managing, modifying, and storing workflow skeletons and their associated provenance. Doing so would form the foundations for allowing ad-hoc workflow adaptation, optimization, data provenance, and data integration.

Previously [60, 4], Lacroix et al. proposed ProtocolDB with a two layer approach to design and record workflows. In ProtocolDB, a scientific workflow is composed of a *design protocol* that captures the scientific aim of the workflow expressed in terms of a domain ontology and one or more *implementation protocols* that specify the resources selected to implement each task. Complementary to ProtocolDB, Strauser et al. [95] developed Semantic Map, a dual level ontology for storing scientific concepts and resources. Together, ProtocolDB and Semantic Map provide the infrastructure to manage the workflow and resource knowledge discovered in this thesis. Since ProtocolDB addresses the need of a scientist to structure a high-level protocol, it does not currently address the needs of digital workflow users. The existing layers are

- *Semantic*: captures the design protocol of the workflow. This is viewed as a network of conceptual relationships that describe the workflow's conceptual tasks.
- *Implementation*: a specific network of resources which is used to implement a conceptual task, as needed for its execution - i.e. which concrete tools are used with their input and output requirements.

The implementation layer in ProtocolDB mirrors the result of skeletonization. However, the skeletonization information also corresponds to an ADDG and a CDDG. Thus, two addition layers for these granular views of workflows are necessary.

- *Execution*: specifies a program that executes the workflow in concrete terms. This duty is typically taken by a WFMS, a script, or a similar mechanism. The information implicit in a particular program is key to tracking tool versions and understanding how changes propagate across layers.
- *Dataflow*: the trace of the program flow that is produced by executing the workflow. The dataflow can support data provenance, which may impact the

way the data are analyzed, compared, and integrated with other data sets. Based on its relation to the other layers, the dataflow can be expressed as ontology-driven schema mappings.

An overview of this ecosystem is shown in Figure 7.1. In this figure, a workflow's trace is iteratively reduced in step with the ProtocolDB layers until an *implementation* has been extracted. At the same time, the ADDG relates to an *execution*, and a CDDG to data. An analogue process would support other methods for tracing workflows, thus populating ProtocolDB with workflows which were originally created in many different ways. The workflows managed by ProtocolDB could then be uniformly transformed or deployed.

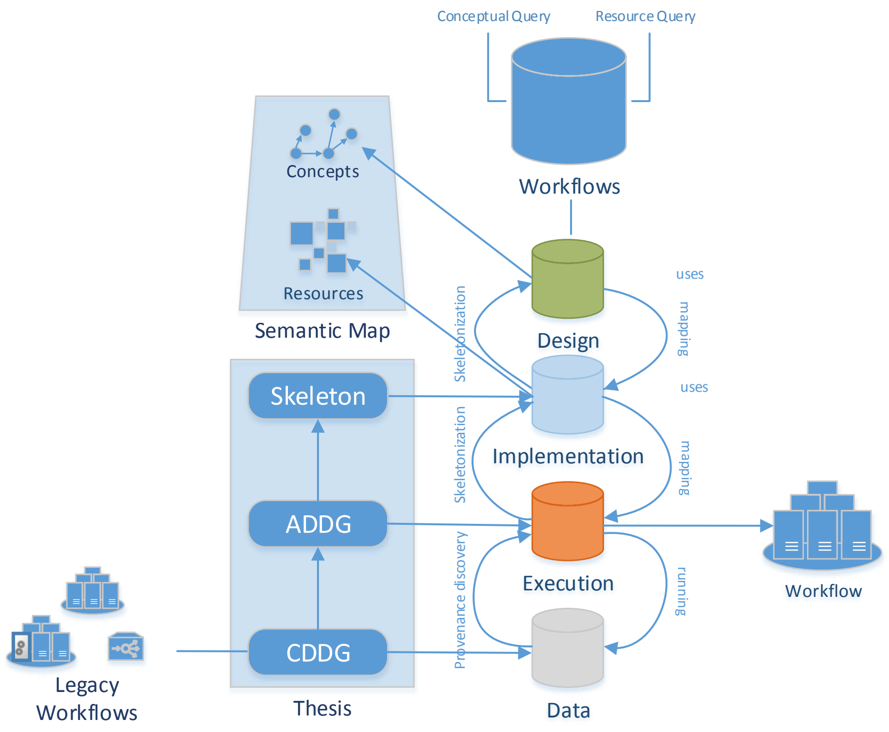

Figure 7.1: Thesis work in the context of ProtocolDB and SemanticMap.

7.1.3 Semantic Extraction

The present work has maintained the goal of constructing a dataflow graph without regards to the *semantics* of the tools used by the workflow. Mapping the dataflow graph to a *semantic map* where all tools are represented as edges in a domain ontology [100] would support the documentation of the workflow in terms of its aim expressed conceptually as proposed in [60] and workflow reuse, optimization, etc. [65]. The present trace mechanism should be extended to extract semantic information at run time and propagate it to the dataflow graph. This can be implemented by connecting tools discovered in a trace to a resource collection. Tools necessarily provide some unique identification (i.e., executable, service URL) for their execution, it is possible to seek its resource counterpart. However, this is difficult in cases where a tool does not exist in a collection, in which an existing resource collection should be extended with any knowledge about that tool that is gained from analysis of its interactions with other, known, tools. It is also possible to extract semantics from a workflow by analysis of the libraries used, provided they have conceptual relation to a scientific domain. In fields like bioinformatics, libraries like BioPython [28] are used to provided standard mechanisms to write and read files - with semantic information like sequence or structure. Since the library has some semantics attached its various components, those semantics can be inferred for the workflow under analysis.

The extracted semantics may be used to further simplify the dataflow graph. At present, the method given in Chapter 5 checks if two elements are equivalent by comparing their usage profile. However, this provides only a low level grouping of elements. In Figure 7.2, the ADDG for SPROUTS is shown again. This workflow contains seven tools which perform the same purpose but cannot be merged since they are different tools. Their equivalence exist at the semantic level which is not currently observed.

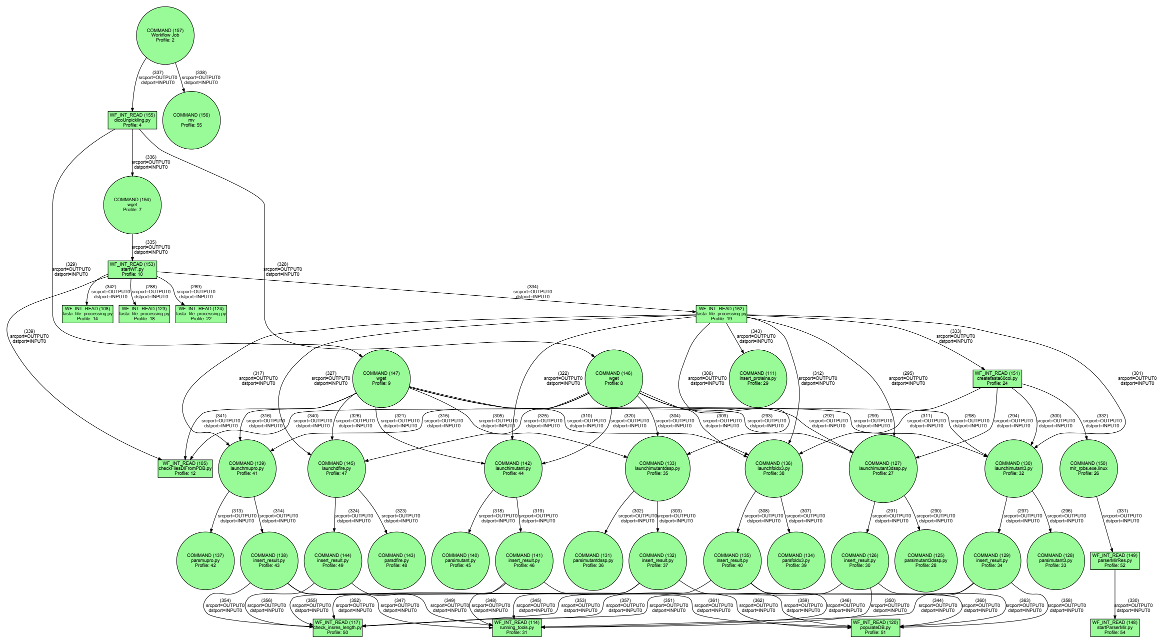

Figure 7.2: Skeletonized ADDG for SPROUTS.

Figure 7.3 shows the result of the method when the different tools are given the same usage profile. This enables the abstraction method to merge them properly. Here, three groups of profiles has been merged: launcher, parser, and insert_result. Each contains seven nodes as implied by the function of the workflow. For this example, the profiles were merged manually. However, by extracting the semantics of the executing workflow, it may be possible to merge these profiles automatically to produce a clearer structure.

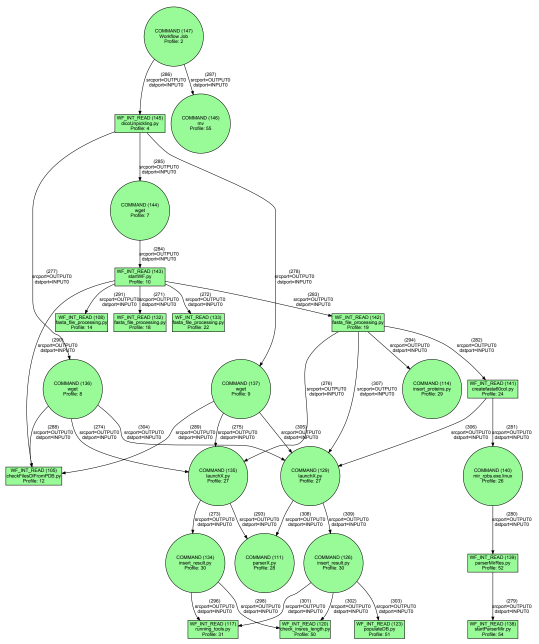
COMMAND (147) Workflow Job Profile: 2
(286) srcport=OUTPUT0 dstport=INPUT0
(287) srcport=OUTPUT0 dstport=INPUT0
WF_INT_READ (145) dicoUnpickling.py Profile: 4
COMMAND (146) mv Profile: 55
(285) srcport=OUTPUT0 dstport=INPUT0
COMMAND (144) wget Profile: 7
(284) srcport=OUTPUT0 dstport=INPUT0
(277) srcport=OUTPUT0 dstport=INPUT0
(278) srcport=OUTPUT0 dstport=INPUT0
WF_INT_READ (143) startWF.py Profile: 10
(291) srcport=OUTPUT0 dstport=INPUT0
(271) srcport=OUTPUT0 dstport=INPUT0
(272) srcport=OUTPUT0 dstport=INPUT0
(283) srcport=OUTPUT0 dstport=INPUT0
WF_INT_READ (108) fasta_file_processing.py Profile: 14
WF_INT_READ (132) fasta_file_processing.py Profile: 18
WF_INT_READ (133) fasta_file_processing.py Profile: 22
WF_INT_READ (142) fasta_file_processing.py Profile: 19
(290) srcport=OUTPUT0 dstport=INPUT0
(294) srcport=OUTPUT0 dstport=INPUT0
(282) srcport=OUTPUT0 dstport=INPUT0
COMMAND (136) wget Profile: 8
COMMAND (137) wget Profile: 9
(276) srcport=OUTPUT0 dstport=INPUT0
(307) srcport=OUTPUT0 dstport=INPUT0
COMMAND (114) insert_proteins.py Profile: 29
WF_INT_READ (141) createfasta60col.py Profile: 24
(288) srcport=OUTPUT0 dstport=INPUT0
(274) srcport=OUTPUT0 dstport=INPUT0
(304) srcport=OUTPUT0 dstport=INPUT0
(289) srcport=OUTPUT0 dstport=INPUT0
(275) srcport=OUTPUT0 dstport=INPUT0
(305) srcport=OUTPUT0 dstport=INPUT0
(306) srcport=OUTPUT0 dstport=INPUT0
(281) srcport=OUTPUT0 dstport=INPUT0
WF_INT_READ (105) checkFilesDlFromPDB.py Profile: 12
COMMAND (135) launchX.py Profile: 27
COMMAND (129) launchX.py Profile: 27
COMMAND (140) mir_rpbs.exe.linux Profile: 26
(273) srcport=OUTPUT0 dstport=INPUT0
(293) srcport=OUTPUT0 dstport=INPUT0
(308) srcport=OUTPUT0 dstport=INPUT0
(309) srcport=OUTPUT0 dstport=INPUT0
(280) srcport=OUTPUT0 dstport=INPUT0
COMMAND (134) insert_result.py Profile: 30
COMMAND (111) parserX.py Profile: 28
COMMAND (126) insert_result.py Profile: 30
WF_INT_READ (139) parserMirRes.py Profile: 52
(296) srcport=OUTPUT0 dstport=INPUT0
(298) srcport=OUTPUT0 dstport=INPUT0
(301) srcport=OUTPUT0 dstport=INPUT0
(302) srcport=OUTPUT0 dstport=INPUT0
(303) srcport=OUTPUT0 dstport=INPUT0
(279) srcport=OUTPUT0 dstport=INPUT0
WF_INT_READ (117) running_tools.py Profile: 31
WF_INT_READ (120) check_insres_length.py Profile: 50
WF_INT_READ (123) populateDB.py Profile: 51
WF_INT_READ (138) startParserMir.py Profile: 54

Figure 7.3: Skeletonized ADDG for SPROUTS with merged profiles.

REFERENCES

[1] Abdelkafi, M., D. Bousabeh, L. Bouzguenda and F. Gargouri, "A comparative study of workflow mining systems", in "Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), 2012 6th International Conference on", pp. 939–945 (2012).

[2] Abiteboul, S., P. Buneman and D. Suciu, *Data on the Web: from relations to semistructured data and XML* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000).

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

[4] Acuña, R., Z. Lacroix and J. Chomilier, "Refurbishing Legacy Biological Workflows", in "Proc. of the IEEE 8th World Congress on Services", pp. 41–49 (2012).

[5] Agesen, O., *Concrete type inference: delivering object-oriented applications*, Ph.D. thesis, Stanford University, Stanford, CA, USA, uMI Order No. GAX96-20452 (1996).

[6] Agrawal, R., D. Gunopulos and F. Leymann, "Mining process models from workflow logs", in "Proc. 6th Intl Conf. Extending Database Technology", vol. 1377 of *Lecture Notes in Computer Science*, pp. 467–483 (Springer Berlin Heidelberg, 1998).

[7] Angelino, E., U. Braun, D. A. Holland and D. W. Margo, "Provenance integration requires reconciliation", in [23].

[8] Angelino, E., D. Yamins and M. Seltzer, "StarFlow: A Script-Centric Data Analysis Environment", in "Provenance and Annotation of Data and Processes", vol. 6378 of *Lecture Notes in Computer Science*, pp. 236–250 (Springer Berlin Heidelberg, 2010).

[9] Bao, Z., S. C. Boulakia, S. B. Davidson and P. Girard, "PDiffView: Viewing the Difference in Provenance of Workflow Results", PVLDB **2**, 2, 1638–1641, URL `http://www.vldb.org/pvldb/2/vldb09-850.pdf` (2009).

[10] Bao, Z., S. Cohen-Boulakia, S. Davidson, A. Eyal and S. Khanna, "Differencing Provenance in Scientific Workflows", in "25th IEEE Int. Conf. on Data Engineering", pp. 808–819 (2009).

[11] Bao, Z., S. B. Davidson and T. Milo, "A fine-grained workflow model with provenance-aware security views", in "Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP)", (2011).

[12] Bavoil, L., 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.", pp. 135–142 (2005).

[13] Belhajjame, K., O. Corcho, D. Garijo, J. Zhao, P. Missier, D. Newman, R. Palma, S. Bechhofer, E. Garca Cuesta, J. M. Gmez-Prez, S. Soiland-Reyes, L. Verdes-Montenegro, D. De Roure and C. Goble, "Workflow-centric research objects: First class citizens in scholarly discourse", in "Proceedings of Workshop on the Semantic Publishing", (2012).

[14] Benzaken, V., J. Fekete, P. Hemery, W. Khemiri and I. Manolescu, "EdiFlow: Data-intensive interactive workflows for visual analytics", in "27^{th} IEEE Int. Conf. on Data Engineering", pp. 780–791 (2011).

[15] Bergmann, R. and Y. Gil, "Similarity assessment and efficient retrieval of semantic workflows", Information Systems **40**, 0, –, URL http://www.sciencedirect.com/science/article/pii/S0306437912001020 (2012).

[16] Bex, G. J., F. Neven, T. Schwentick and K. Tuyls, "Inference of concise dtds from xml data", in "Proceedings of the 32nd International Conference on Very Large Data Bases", VLDB '06, pp. 115–126 (VLDB Endowment, 2006), URL http://dl.acm.org/citation.cfm?id=1182635.1164139.

[17] Biton, O., S. Cohen-Boulakia, S. Davidson and C. Hara, "Querying and managing provenance through user views in scientific workflows", in "Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on", pp. 1072–1081 (2008).

[18] Bochner, C., R. Gude and A. Schreiber, "A Python Library for Provenance Recording and Querying", in "Provenance and Annotation of Data and Processes", edited by J. Freire, D. Koop and L. Moreau, vol. 5272 of *Lecture Notes in Computer Science*, pp. 229–240 (Springer Berlin Heidelberg, 2008).

[19] Bolz, C. F., A. Cuni, M. Fijalkowski and A. Rigo, "Tracing the meta-level: Pypy's tracing jit compiler", in "Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems", ICOOOLPS '09, pp. 18–25 (ACM, New York, NY, USA, 2009), URL http://doi.acm.org/10.1145/1565824.1565827.

[20] Bouarfa, L. and J. Dankelman, "Workflow mining and outlier detection from clinical activity logs", Journal of Biomedical Informatics **45**, 6, 1185 – 1190 (2012).

[21] Brandes, U., M. Eiglsperger, I. Herman, M. Himsolt and M. Marshall, "Graphml progress report structural layer proposal", in "Graph Drawing", vol. 2265, pp. 501–512 (Springer Berlin Heidelberg, 2002).

[22] Buehrer, G. and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities", in "Proceedings of the 2008 International Conference on Web Search and Data Mining", WSDM '08, pp. 95–106 (ACM, New York, NY, USA, 2008), URL http://doi.acm.org/10.1145/1341531.1341547.

[23] Buneman, P. and J. Freire, eds., *3rd Workshop on the Theory and Practice of Provenance, TaPP'11, Heraklion, Crete, Greece, June 20-21, 2011* (USENIX Association, 2011).

[24] Burstein, M. H., F. Yaman, R. M. Laddaga and R. J. Bobrow, "Poirot: Acquiring workflows by combining models learned from interpreted traces", in "Proc. of the 5^{th} ACM Int. Conf. on Knowledge Capture", pp. 129–136 (2009).

[25] Callahan, S. P., J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva and H. T. Vo, "Using provenance to streamline data exploration through visualization", Tech. rep., SCI Institute (2006).

[26] Chen, Z., L. Chen, Y. Zhou, Z. Xu, W. Chu and B. Xu, "Dynamic Slicing of Python Programs", in "Proc. 38^{th} IEEE Annual Computer Software and Applications Conf.", pp. 219–228 (2014).

[27] Claes, J. and G. Poels, "Merging event logs for process mining: A rule based merging method and rule suggestion algorithm", Expert Systems with Applications **41**, 16, 7291 – 7306 (2014).

[28] Cock, P. J., T. Antao, J. T. Chang, B. A. Chapman, C. J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski and M. J. de Hoon, "Biopython: freely available Python tools for computational molecular biology and bioinformatics.", Bioinformatics **25**, 11, 1422–1423 (2009).

[29] Cohen-Boulakia, S., 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).

[30] Cohen-Boulakia, S. and U. Leser, "Search, adapt, and reuse: the future of scientific workflows", SIGMOD Rec. **40**, 2, 6–16, URL `http://doi.acm.org/10.1145/2034863.2034865` (2011).

[31] Consortium, B., "Interoperability with moby 1.0 - it's better than sharing your toothbrush!", Briefings in Bioinformatics **9**, 220–231 (2008).

[32] Costa, F., D. de Oliveira, E. Ogasawara, A. Lima and M. Mattoso, "Athena: Text mining based discovery of scientific workflows in disperse repositories", in "Resource Discovery", edited by Z. Lacroix and M. Vidal, vol. 6799 of *Lecture Notes in Computer Science*, pp. 104–121 (Springer Berlin Heidelberg, 2012).

[33] de Medeiros, A., A. Weijters and W. van der Aalst, "Genetic process mining: an experimental evaluation", Data Mining and Knowledge Discovery **14**, 2, 245–304, URL `http://dx.doi.org/10.1007/s10618-006-0061-7` (2007).

[34] De Roure, D., S. Bechhofer, C. Goble and D. Newman, "Scientific Social Objects: The Social Objects and Multidimensional Network of the myExperiment Website", in "Proc. of the 3^{rd} IEEE Int. Conf. on Social Computing and Privacy, Security, Risk and Trust", pp. 1398–1402 (IEEE, 2011).

[35] Deelman, E., 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", vol. 3165 of *Lecture Notes in Computer Science*, pp. 11–20 (Springer Berlin Heidelberg, 2004).

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

[37] Filguiera, R., I. Klampanos, A. Krause, M. David, A. Moreno and M. Atkinson, "Dispel4Py: A Python Framework for Data-intensive Scientific Computing", in "Proc. of the IEEE Int. Workshop on Data Intensive Scalable Computing Systems", DISCS '14, pp. 9–16 (IEEE Press, Piscataway, NJ, USA, 2014), URL `http://dx.doi.org/10.1109/DISCS.2014.12`.

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

[39] Gansner, E. R. and S. C. North, "An open graph visualization system and its applications to software engineering", SOFTWARE - PRACTICE AND EXPERIENCE **30**, 1203–1233 (2000).

[40] Garcia-Jimenez, B. and M. D. Wilkinson, "Automatic annotation of bioinformatics workflows with biomedical ontologies", in "Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications", edited by T. Margaria and B. Steffen, vol. 8803 of *Lecture Notes in Computer Science*, pp. 464–478 (Springer Berlin Heidelberg, 2014).

[41] Garijo, D., P. Alper, K. Belhajjame, O. Corcho, Y. Gil and C. Goble, "Common motifs in scientific workflows: An empirical analysis", in "E-Science (e-Science), 2012 IEEE 8th International Conference on", pp. 1–8 (2012).

[42] Garofalakis, M., A. Gionis, R. Rastogi, S. Seshadri and K. Shim, "XTRACT: a system for extracting document type descriptors from XML documents", SIGMOD Rec. **29**, 2, 165–176, URL `http://dx.doi.org/10.1145/342009.335409` (2000).

[43] Gervasio, M. T. and J. L. Murdock, "What Were You Thinking?: Filling in Missing Dataflow Through Inference in Learning from Demonstration", in "Proc. of the 14^{th} Int. ACM Conf. on Intelligent User Interfaces", IUI '09, pp. 157–166 (2009).

[44] Gil, Y., E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau and J. Myers, "Examining the Challenges of Scientific Workflows", Computer **40**, 12, 24–32 (2007).

[45] Goble, C. A., J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li and D. D. Roure, "myExperiment: a repository and social network for the sharing of bioinformatics workflows", Nucl. Acids Res. **38**, suppl 2, W677–W682 (2010).

[46] Goecks, J., A. Nekrutenko, J. Taylor and T. G. Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences", Genome Biology **11**, 8, R86 (2010).

[47] Goldman, R. and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases", in "Proceedings of the 23rd International Conference on Very Large Data Bases", VLDB '97, pp. 436–445 (1997), URL `http://dl.acm.org/citation.cfm?id=645923.671008`.

[48] Grochow, J. A. and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking", in "Research in Computational Molecular Biology", edited by T. Speed and H. Huang, vol. 4453 of *Lecture Notes in Computer Science*, pp. 92–106 (Springer Berlin Heidelberg, 2007).

[49] Günther, C. W. and W. M. van der Aalst, "Fuzzy mining: Adaptive process simplification based on multi-perspective metrics", in "Business Process Management", edited by G. Alonso, P. Dadam and M. Rosemann, vol. 4714 of *Lecture Notes in Computer Science*, pp. 328–343 (Springer Berlin Heidelberg, 2007).

[50] Guo, P. J. and D. Engler, "Using automatic persistent memoization to facilitate data analysis scripting", in "Proc. of the ACM International Symposium on Software Testing and Analysis", pp. 287–297 (2011).

[51] Hanson-Smith, V., "asr-pipeline", URL `http://github.com/vhsvhs` (2015).

[52] Hegewald, J., F. Naumann and M. Weis, "Xstruct: Efficient schema extraction from multiple and large xml documents", in "Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on", pp. 81–81 (2006).

[53] Horta, F., 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", EDBT '13, pp. 337–338 (ACM, New York, NY, USA, 2013), URL `http://doi.acm.org/10.1145/2457317.2457379`.

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

[55] Huq, M. R., P. M. G. Apers and A. Wombacher, "ProvenanceCurious: A Tool to Infer Data Provenance from Scripts", in "Proc. of the 16^{th} Int. Conf. on Extending Database Technology", EDBT '13, pp. 765–768 (ACM, New York, NY, USA, 2013), URL `http://doi.acm.org/10.1145/2452376.2452475`.

[56] Janga, P. and K. Davis, "Schema extraction and integration of heterogeneous xml document collections", in "Model and Data Engineering", edited by A. Cuzzocrea and S. Maabout, vol. 8216 of *Lecture Notes in Computer Science*, pp. 176–187 (Springer Berlin Heidelberg, 2013).

[57] Johnson, M., "HybSeqPipeline: First DOI Release", URL http://dx.doi.org/10.5281/zenodo.11977 (2014).

[58] Khan, K. U., W. Nawaz and Y.-K. Lee, "Lossless graph summarization using dense subgraphs discovery", in "Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication", IMCOM '15, pp. 9:1–9:7 (ACM, New York, NY, USA, 2015), URL http://doi.acm.org/10.1145/2701126.2701157.

[59] Khedker, U., A. Sanyal and B. Sathe, *Data Flow Analysis: Theory and Practice* (CRC Press, 2009).

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

[61] Kraiss, A. and G. Weikum, "Integrated document caching and prefetching in storage hierarchies based on markov-chain predictions", The VLDB Journal **7**, 3, 141–162, URL http://dx.doi.org/10.1007/s007780050060 (1998).

[62] Kster, J. and S. Rahmann, "Snakemake a scalable bioinformatics workflow engine", Bioinformatics (2012).

[63] Kudo, M., A. Ishida and N. Sato, "Business process discovery by using process skeletonization", in "Signal-Image Technology Internet-Based Systems (SITIS), 2013 International Conference on", pp. 976–982 (2013).

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

[65] Lacroix, Z., C. Legendre and S. Tuzmen, "Reasoning on Scientific Workflows", in "Proc. of the IEEE World Conference on Services", pp. 306–313 (2009).

[66] Lei, J. 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).

[67] Liu, C., C. Chen, J. Han and P. S. Yu, "Gplag: Detection of software plagiarism by program dependence graph analysis", in "Proc. of the 12th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD06", pp. 872–881 (ACM Press, 2006).

[68] Lonquety, M., 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).

[69] Lou, J.-G., Q. Fu, S. Yang, J. Li and B. Wu, "Mining program workflow from interleaved traces", in "Proc. of the 16th ACM Int. Conf. on Knowledge Discovery and Data Mining", pp. 613–622 (2010).

[70] Ludascher, B., 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).

[71] Ludäscher, B., M. Weske, T. Mcphillips and S. Bowers, "Scientific Workflows: Business As Usual?", in "Proc. of the 7th Int. Conf. on Business Process Management", vol. 5701 of *Lecture Notes in Computer Sciences*, pp. 31–47 (Springer-Verlag, Berlin, Heidelberg, 2009).

[72] Ma, Y., X. Zhang and K. Lu, "A graph distance based metric for data oriented workflow retrieval with variable time constraints", Expert Systems with Applications **41**, 4, Part 1, 1377 – 1388, URL `http://www.sciencedirect.com/science/article/pii/S0957417413006477` (2014).

[73] Madeira, F. and L. Krippahl, "Pycoevol - A Python Workflow to Study Protein-protein Coevolution", in "Proc. BIOINFORMATICS", pp. 143–149 (SciTePress, 2012).

[74] Marciniak, J., "Xml schema and data summarization", in "Artifical Intelligence and Soft Computing", edited by L. Rutkowski, R. Scherer, R. Tadeusiewicz, L. Zadeh and J. Zurada, vol. 6114 of *Lecture Notes in Computer Science*, pp. 556–565 (Springer Berlin Heidelberg, 2010).

[75] Mates, P., 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, pp. 555–564 (Springer-Verlag, Berlin, Heidelberg, 2011), URL `http://dl.acm.org/citation.cfm?id=2032397.2032445`.

[76] McPhillips, T. M. and S. Bowers, "An approach for pipelining nested collections in scientific workflows", ACM SIGMOD Record **34(3)**, 12–17 (2005).

[77] McPhillips, T. M., S. Bowers and B. Ludascher, "Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data", in "Third International Workshop on Data Integration in the Life Sciences (DILS)", vol. 4075 of *Lecture Notes in Computer Sciences*, pp. 248–263 (Springer, 2006).

[78] Medeiros, C. B., 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).

[79] Milo, R., S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii and U. Alon, "Network Motifs: Simple Building Blocks of Complex Networks", Science **298**, 5594, 824–827, URL `http://dx.doi.org/10.1126/science.298.5594.824` (2002).

[80] Muniswamy-Reddy, K.-K., D. A. Holland, U. Braun and M. Seltzer, “Provenance-aware storage systems”, in “Proceedings of the Annual Conference on USENIX ’06 Annual Technical Conference”, ATEC ’06, pp. 4–4 (USENIX Association, Berkeley, CA, USA, 2006), URL `http://dl.acm.org/citation.cfm?id=1267359.1267363`.

[81] Murta, L., V. Braganholo, F. Chirigati, D. Koop and J. Freire, “noWorkflow: Capturing and Analyzing Provenance of Scripts”, in “5th Int. Provenance and Annotation Workshop”, vol. 8628 of *Lecture Notes in Computer Science*, pp. 71,83 (2015).

[82] Navlakha, S., R. Rastogi and N. Shrivastava, “Graph summarization with bounded error”, in “Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data”, SIGMOD ’08, pp. 419–432 (ACM, New York, NY, USA, 2008), URL `http://doi.acm.org/10.1145/1376616.1376661`.

[83] Pajon, A., “Bacterial automatic annotation pipeline”, URL `http://github.com/pajanne/bacana` (2011).

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

[85] Petersen, T. N. N., S. Brunak, G. von Heijne and H. Nielsen, “SignalP 4.0: discriminating signal peptides from transmembrane regions.”, Nature methods **8**, 10, 785–786, URL `http://dx.doi.org/10.1038/nmeth.1701` (2011).

[86] Pienta, R., J. Abello, M. Kahng and D. H. Chau, “Scalable graph exploration and visualization: Sensemaking challenges and opportunities”, in “2015 International Conference on Big Data and Smart Computing, BIGCOMP 2015, Jeju, South Korea, February 9-11, 2015”, pp. 271–278 (2015), URL `http://dx.doi.org/10.1109/35021BIGCOMP.2015.7072812`.

[87] Rahman, O., S. Cummings, D. Harrington and I. 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, URL `http://dx.doi.org/10.1007/s11274-008-9795-2` (2008).

[88] Rivest, R., “The md5 message-digest algorithm”, (1992).

[89] Salib, M., “Faster than c: Static type inference with starkiller”, in “PyCon Proceedings”, vol. 3 (2004).

[90] Sapino, M. L., K. S. Candan and P. Bertolotti, “Log-analysis based characterization of multimedia documents for effective delivery of distributed multimedia presentations”, in “The 12th International Conference on Distributed Multimedia Systems”, (2006).

[91] Shields, M., “Control- versus data-driven workflows”, in “Workflows for e-Science”, edited by I. Taylor, E. Deelman, D. Gannon and M. Shields, pp. 167–173 (Springer London, 2007).

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

[93] Sonnhammer, E. L. L., G. v. Heijne and A. Krogh, “A Hidden Markov Model for Predicting Transmembrane Helices in Protein Sequences”, in “Proc. of the 6^{th} Int. Conf. on Intelligent Systems for Molecular Biology”, pp. 175–182 (AAAI Press, 1998).

[94] Starlinger, J., B. Brancotte, S. Cohen-Boulakia and U. Leser, “Similarity Search for Scientific Workflows”, Proc. VLDB Endow. **7**, 12, 1143–1154, URL `http://dl.acm.org/citation.cfm?id=2732977.2732988` (2014).

[95] Strauser, E., M. Naveau, H. Mnager, J. Maupetit, Z. Lacroix and P. Tuffry, “Semantic map for structural bioinformatics: Enhanced service discovery based on high level concept ontology”, in “Resource Discovery”, (Springer Berlin Heidelberg, 2012).

[96] Sun, P., 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, URL `http://www.vldb.org/pvldb/2/vldb09-66.pdf` (2009).

[97] Szlávik, Z., A. Tombros and M. Lalmas, “Summarisation of the logical structure of xml documents”, Inf. Process. Manage. **48**, 5, 956–968, URL `http://dx.doi.org/10.1016/j.ipm.2011.11.002` (2012).

[98] Taylor, I., M. Shields, I. Wang and A. Harrison, “The Triana Workflow Environment: Architecture and Applications”, in “Workflows for e-Science”, edited by I. Taylor, E. Deelman, D. Gannon and M. Shields, pp. 320–339 (Springer London, 2007).

[99] Tian, Y., R. A. Hankins and J. M. Patel, “Efficient aggregation for graph summarization”, in “Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data”, SIGMOD ’08, pp. 567–580 (ACM, New York, NY, USA, 2008), URL `http://doi.acm.org/10.1145/1376616.1376675`.

[100] Tufféry, P., Z. Lacroix and H. Ménager, “Semantic Map of Services for Structural Bioinformatics”, in “Proc. 18^{th} IEEE Int. Conf. on Scientific and Statistical database Management”, pp. 217–224 (IEEE Computer Society, 2006).

[101] Valdes, J., R. E. Tarjan and E. L. Lawler, “The Recognition of Series Parallel Digraphs”, in “Proc. of the 11^{th} Annual ACM Symposium on Theory of Computing”, STOC ’79, pp. 1–12 (ACM, New York, NY, USA, 1979), URL `http://doi.acm.org/10.1145/800135.804393`.

[102] van der Aalst, W., A. Weijter and L. Maruster, “Workflow mining: Discovering process models from event logs”, IEEE Trans. on Knowledge and Data Engineering **16**, 2004 (2003).

[103] Weijters, A., W. Aalst and A. Medeiros, “Process Mining with the Heuristics Miner-algorithm”, BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven (2006).

[104] Wilkinson, M. and M. Links, “BioMOBY: an open source biological web services proposal”, Briefings in bioinformatics **3**, 4, 331 (2002).

[105] Xu, Z., J. Qian, L. Chen, Z. Chen and B. Xu, “Static Slicing for Python First-Class Objects”, in “Proc. of the 13^{rd} International Conference on Quality Software”, pp. 117–124 (2013).

[106] Yaman, F. and T. Oates, “Workflow Inference: What to do with one example and no semantics”, in “AAAI-07 Workshop on Acquiring Planning Knowledge via Demonstration”, (2007).

[107] Zeng, R., X. He, J. Li, Z. Liu and W. M. P. van der Aalst, “A method to build and analyze scientific workflows from provenance through process mining”, in [23].

[108] Zhao, Y., 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”, pp. 199–206 (2007).