HAL
open science

Designing scientific workflows following a structure and provenance-aware strategy

Jiuqiang Chen

To cite this version:

Jiuqiang Chen. Designing scientific workflows following a structure and provenance-aware strategy. Databases [cs.DB]. Université Paris Sud - Paris XI, 2013. English. NNT : . tel-00931122

HAL Id: tel-00931122

https://theses.hal.science/tel-00931122v1

Submitted on 14 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre:

THÈSE

Présentée pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ PARIS-SUD

Spécialité: Informatique

par

Jiuqiang CHEN

Designing scientific workflows following a structure and provenance -aware strategy

pour une soutenance en Octobre 2013

Mme.	Christine Froidevaux	(Directeur de thèse)
Mme.	Sarah Cohen-Boulakia	(Co-encadrante)

Rapporteurs:

Mme.	Therese Libourel
M.	Mohand-Hacid Said

Thèse préparée au
Département de Informatique d'Orsay
Laboratoire de Informatique (UMR 8623), Bât. 650
Université Paris-Sud
91,405 Orsay CEDEX

Abstract

Bioinformatics experiments are usually performed using scientific workflows in which tasks are chained together forming very intricate and nested graph structures. Scientific workflow systems have then been developed to guide users in the design and execution of workflows. An advantage of these systems over traditional approaches is their ability to automatically record the provenance (or lineage) of intermediate and final data products generated during workflow execution. The provenance of a data product contains information about how the product was derived, and it is crucial for enabling scientists to easily understand, reproduce, and verify scientific results. For several reasons, the complexity of workflow and workflow execution structures is increasing over time, which have a clear impact on scientific workflows reuse.

The global aim of this thesis is to enhance workflow reuse by providing strategies to reduce the complexity of workflow structures while preserving provenance. Two strategies are introduced.

First, we propose an approach to rewrite the graph structure of any scientific workflow (classically represented as a directed acyclic graph (DAG)) into a simpler structure, namely, a series-parallel (SP) structure while preserving provenance. SP-graphs are simple and layered, making the main phases of workflow easier to distinguish. Additionally, from a more formal point of view, polynomial-time algorithms for performing complex graph-based operations (e.g., comparing workflows, which is directly related to the problem of subgraph homomorphism) can be designed when workflows have SP-structures while such operations are related to an NP-hard problem for DAG structures without any restriction on their structures. The SPFlow rewriting and provenance-preserving algorithm and its associated tool are thus introduced.

Second, we provide a methodology together with a technique able to reduce the redundancy present in workflows (by removing unnecessary occurrences of tasks). More precisely, we detect "anti-patterns", a term broadly used in program design to indicate the use of idiomatic forms that lead to over-complicated design, and which should therefore be avoided. We thus provide the DistillFlow algorithm able to transform a workflow into a distilled semantically-equivalent workflow, which is free or partly free of anti-patterns and has a more concise and simpler structure.

The two main approaches of this thesis (namely, SPFlow and DistillFlow) are based on a provenance model that we have introduced to represent the provenance structure of the workflow executions. The notion of provenance-equivalence which determines whether two workflows have the same meaning is also at the center of our work. Our solutions have been systematically tested on large collections of real workflows, especially from the Taverna system. Our approaches are available for use at https://www.lri.fr/∼chenj/.

Keywords: scientific workflows, provenance, provenance-equivalence, series-parallel graphs, Taverna, anti-patterns

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Problem Statement . . . 5

1.3 Contributions . . . 6

1.4 Thesis Structure . . . 7

2 Preliminaries **9**

2.1 Basic graph concepts and notations . . . 10

2.2 General workflow model . . . 15

2.3 Series-Parallel graphs . . . 16

2.3.1 Definitions . . . 16

2.3.2 SP reduction . . . 18

2.3.3 Properties of SP graphs related to their recognition . . . 19

2.4 Workflows in Taverna . . . 20

2.5 Summary . . . 22

3 Provenance Model **23**

3.1 Related work . . . 24

3.2 Our Provenance Model . . . 26

3.3 Provenance-equivalence . . . 30

3.4 Conclusion and Discussion . . . 31

3.4.1 Conclusion . . . 31

3.4.2 Towards "problematic" data dependencies in Taverna . . . 33

3.4.3 Possible solutions . . . 36

3.5 Summary . . . 38

4 Rewriting scientific workflows while preserving provenance **39**

4.1 Motivating Scenarios . . . 41

4.1.1 Designing workflows . . . 42

4.1.2 Querying workflows . . . 43

4.1.3 Scheduling workflows . . . 44

4.2 Distance from non-SP to SP graphs 44
4.2.1 Vertex reduction 45
4.2.2 Vertex duplication 47
4.2.3 Complexity measures 49
4.3 Graph rewriting problems (non-SP to SP) 50
4.3.1 Review of existing approaches 50
4.3.2 Compositions of forbidden graphs 53
4.4 SPFlow: a new provenance-equivalent rewriting algorithm for workflows 57
4.4.1 Principle of SPFlow 57
4.4.1.1 Duplicated subgraph 58
4.4.1.2 Reduction sequence 59
4.4.1.3 Reducing redundancy of duplicated vertices 62
4.4.1.4 Algorithm description 63
4.4.2 Example of use of SPFlow 65
4.4.3 Complexity 65
4.4.4 Soundness of vertex duplication algorithm 68
4.5 Experimental study 69
4.5.1 Workflow Structures 69
4.5.2 Evaluating SPFlow 70
4.6 Implementation of the algorithm 71
4.6.1 SPFlow architecture 71
4.6.2 Functionalities of SPFlow 72
4.7 Discussion 75
4.8 Summary 77

5 Distilling Structure in Taverna Scientific Workflows: A refactoring approach 79
5.1 Use cases 81
5.2 Anti-patterns and Transformations 82
5.2.1 Assumptions 83
5.2.2 Transformations 83
5.2.3 Safe Transformations 85
5.3 Refactoring approach 86

5.3.1 Principle of the algorithm 87
5.3.2 Illustration of the algorithm 88
5.4 Experimental Study 91
5.4.1 Anti-patterns in workflow sets 91
5.4.2 Results obtained by DistillFlow 92
5.5 Discussion 94
5.5.1 Simpler structures 94
5.5.2 SP structures 95
5.5.3 Towards other kinds of (anti-)patterns 99
5.5.4 Provenance-equivalence 100
5.6 Related Work 101
5.7 Summary 102

6 Conclusion and Future work 103
6.1 Conclusion 103
6.2 Future Work 105

A DistillFlow: refactoring scientific workflows for better (re)use (Tool) 113
A.1 DistillFlow architecture 113
A.2 Functionalities of DistillFlow 114
A.3 Extensibility of DistillFlow 118

B Why scientific workflows have non-SP structures (Preliminary study) 121
B.1 On the influence of the kind of processors 122
B.2 Trace links and trace nodes 123

Bibliography 125

Chapter 1

Introduction

Contents

1.1 Motivation 1

1.2 Problem Statement 5

1.3 Contributions 6

1.4 Thesis Structure 7

1.1 Motivation

Scientific workflow management systems, (e.g., Taverna [HWS+06], Kepler [LAB+06], Chimera [FVWZ02], Galaxy [GNT+10], Wings [GRD+07]) are increasingly being used by scientists to construct and execute complex scientific analyses. Such analyses are typically data-centric and involve "gluing" together data retrieval, computation, and visualization components into a single executable analysis pipeline [BLNC05]. Such a pipeline is represented by a workflow which is modeled as a graph, where edges denote scheduling dependencies between computation tasks [HFYS04, CBFJ12]. Intuitively, a *workflow specification* is a framework for the execution of workflows, which specifies the set of tasks that are performed and the order to be observed between the different tasks executions. According to the input data given to the workflow specification and assignments of values to the task parameters, different *workflow runs* are obtained. A run is then also represented as a graph where each vertex represents the execution of a task and edges are labeled by the data consumed and produced at each step. In this thesis, following what is in several workflow systems, we will consider that the specifications have a directed cyclic graph (DAG) structure and the runs have the same structures

as their specifications. The main goal of scientific workflows, is to represent in-silico experiments, which entails frequent reuse and repurposing throughout their life-cycle [CM11].

Figure 1.1 provides (a) an example of workflow specification from Taverna [HWS+06], (b) its representation as a graph and (c) an example of run. Faced

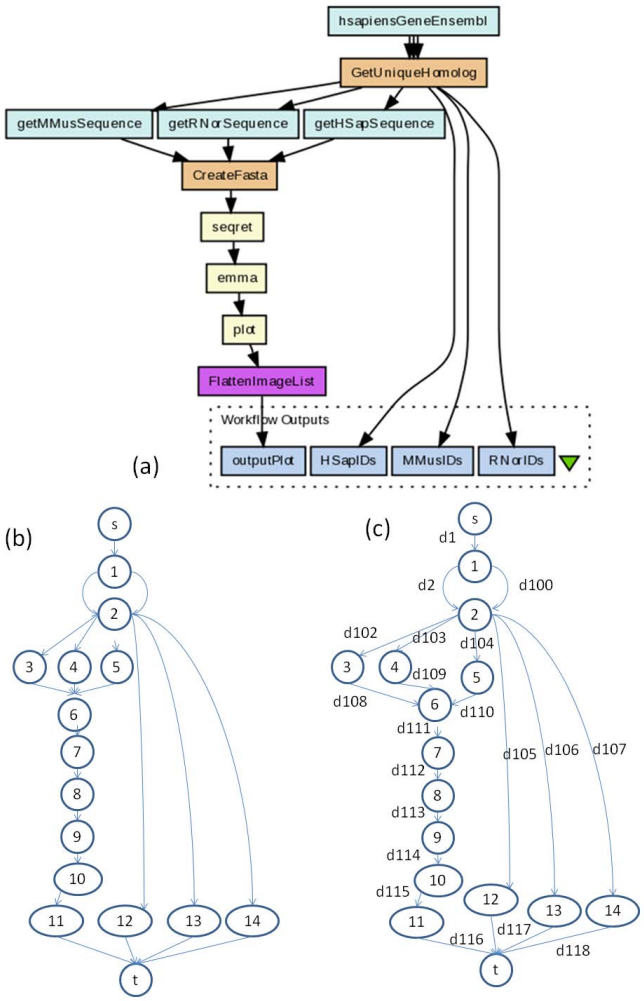


Figure 1.1: (a) Taverna workflow; (b) specification graph; (c) run graph

with the increasing complexity of runs and the need for reproducibility of results, provenance has become an important research topic [CBFJ12]. The provenance (also referred to as *lineage*, and *pedigree*) of a data product contains information about the process and data used to derive the product [DF08]. It is often organized as dependency graphs [MFF+08]. The visualization of such dependency graphs is especially useful for scientific workflow reuse, since the data, processes, and dependencies associated with a workflow run can be clearly seen by work-

flow users. By analysing and creating insightful visualizations of provenance data, scientists can debug their tasks and obtain a better understanding of their results. With the help of provenance, scientists who wish to perform new analyses should be able to find workflow specifications with same or similar meanings of interest to reuse or modify. They can also search for executions associated with a specification to understand the meaning of the workflow, or to correct/debug an erroneous specification. Furthermore, structural provenance queries can help scientists to determine what produced data might have been affected by its input, or to understand how and why the process that led to create a given data has actually failed. Therefore, provenance information is clearly useful for scientific workflows users and systems. However, due to the complexity of workflows, the provenance information which is organized into a graph becomes very large, for which understanding and exploring provenance information becomes a significant challenge for users [DF08, MFF+08]. While most systems record and store data and process dependencies, a few provide easy-to-use and efficient approaches for accessing provenance information [Koo12]. Additionally, some workflow systems take complex data structure (e.g., lists [HWS+06], trees [BML08], $\cdots$) into account, which makes provenance presentation a very challenging point. However, to support better reuse of scientific workflows, provenance should be more exploited to present the meaning of scientific workflows for both workflow systems and users.

In the last decade, considerable effort has been put into the improvement of sharing and reusing scientific workflows. Workflow reuse in e-Science is intrinsically linked to a desire that workflows be shared and reused by the community as (part of) best practice scientific protocols [GSLG05]. It has the potential to [GSLG05, LRL+12]: reduce workflow authoring time (less "re-inventing the wheel"); improve quality through shared workflow development (leveraging the expertise of previous users); and improve experimental provenance at the process level through reuse of established and validated workflows (analogous to using proven algorithms or practices rather than inventing a new which is potentially error-prone). However, as stated by recent studies [SCBL12, CBL11, LRL+12], while the number of available scientific workflows is increasing along with their popularity, workflows are not (re)used and shared as much as they could be. Several years ago, Goderis *et al.* [GSLG05] summarized several bottlenecks of workflow

reuse and repurposing, in which they argue that the main reasons are the restrictions on service availability, lack of a comprehensive discovery model and the complexity of workflows. According to Zhao *et al.* in [ZGPB+12], one of the main impediments to workflow reuse is due to the decayed or reduced ability of the resources required for executing workflow, like services and data, which can be either local and hosted along with the workflow or remote, such as public repositories or web services hosted by third parties. The causes of this impediments include: (1) it is difficult to volatile third-party resources; (2) missing example data (it is not always obvious which data can be used as inputs to the workflow execution, and example inputs are often most helpful); (3) missing execution environment (the execution of a workflow may rely on a particular local execution environment, e.g., a local R server or a specific version of workflow execution software); (4) insufficient descriptions about workflows (sometimes a workflow workbench cannot provide sufficient information about what caused the failure of a workflow run). Several solutions for these causes can be found in [ZGPB+12].

In this thesis, we have focused specifically on the Taverna workflow management system, which for the past ten years, has been popular within the bioinformatics community [HWS+06]. Despite the fact that hundreds of Taverna workflows have been available for years through the myExperiment public workflow repository [RGS09] (http://www.myexperiment.org), their reuse by scientists other than the original author is generally limited [CBL11]. Recently, several studies [SCBL12, CBL11, LRL+12, TZF10] highlight the complexity of workflow structures as one of the main reason of the limited reuse of (Taverna) scientific workflows. The complexity of workflow structure, involves the number of nodes and links but is also related to intricate workflow structure features [CBCG+13]. Again, several factors may explain such a structural complexity including the fact that the bioinformatics process to be implemented is intrinsically complex, or the workflow system may not provide appropriate expressivity, forcing users to design arbitrary complex workflows. Therefore, to obtain a simpler workflow structure for a complex workflow while preserving the meaning (provenance/semantics) becomes especially important.

Motivated by the facts above, rewriting complex scientific workflow structures into simpler ones to make them easier to (re)use thus is the main topic of this

thesis. In the next section, we will state the problems on this topic in details.

1.2 Problem Statement

In this thesis, our aim is to provide strategies to design scientific workflows. The originality of our approach lies in considering two notions, namely, *provenance* and *workflow structures*. Our contributions have been introduced in our published papers [CBFJ12] and [CBCG+13]. In this document, we recall them and provide detailed explanations and discussion on our work.

As provenance provides support in scientific workflow reuse, a significant number of tools for managing the vast amounts of data provenance have been designed to assist the storage of provenance data (e.g., indexing...), query the data (e.g., difference between executions, search for patterns), visualize the workflow provenance or (re)schedule executions... These tools all make intrinsically complex operations on graph structures (search for subgraphs in a graph, comparing graphs, ...), which, if carried out on Directed Acyclic Graphs (DAGs), with no other restriction of structure, may lead to NP-hard problems. Instead, these problems can be solved in polynomial time when specific restrictions are imposed on graphs, such as considering *series-parallel* (SP) structures [BKS92]. Some provenance management approaches from [BBD+09, BBDH08, BCC+05] have therefore chosen to restrict workflow graphs to SP structures. However, in general, workflows obtained using workflow systems are DAGs with any structure. Providing a procedure to rewrite any workflow graph into an SP graph while preserving provenance information would allow to better exploit the provenance management tools and should make scientific workflows easier to (re)use. This is the first goal of this research. The first research question addressed in this thesis is:

(1) How to rewrite any workflow graph into SP graph while preserving provenance?

The second main contribution of our work focuses on scientific workflow structures themselves. We argue that one of the contributing factors for the difficulties in reuse, is the presence of certain design "anti-patterns", a term broadly used in business process modelling and program design, to indicate the use of idiomatic

forms that lead to over-complicated design, and which should therefore be avoided. Our preliminary analysis of the structure of 1,400 scientific workflows collected from myExperiments reveals that, in numerous cases, such a complexity is due mainly to redundancy, which is in turn an indication of over-complicated design, and thus there is a chance for a reduction in complexity which does not alter the workflow semantics. Our main contention in this fact is that such a reduction in complexity can be performed automatically, and that it will be beneficial both in terms of user experience (easier design and maintenance), and in terms of operational efficiency (easier to manage, and sometimes to exploit the latent parallelism amongst the tasks). So, the second research question addressed in this thesis is:

(2) How to rewrite scientific workflows to make them free or partly free of redundancy without altering the workflow semantics?

The solutions for these two research questions are respectively presented in Chapter 4 and Chapter 5. The next section describes in more details the actual contributions.

1.3 Contributions

Our main contributions are summarized as below.

First series of contributions (have been published in the 8th IEEE International Conference on eScience 2012 [CBFJ12] and the 28th Journees de Bases de Donnees Avancees (BDA) 2012 [CBFC12]):

- We propose a model to represent scientific workflows and provenance generated in their execution.
- We give a definition of the notion of *provenance-equivalence* which can be used to identify whether two workflows have the same meaning.
- We review several rewriting strategies for transforming non-SP graphs into SP graphs and prove that they are not provenance-equivalent.
- We design a provenance-equivalent algorithm, named "SPFlow", to translate non-SP workflows into SP workflows.

- We illustrate our algorithm by providing an evaluation of our approach on a thousand of scientific workflows.
- We develop a tool based on SPFlow, which takes in a non-SP Taverna workflow and provide an SP version of the workflow usable in Taverna.

Second series of contributions (have been published in the "BMC Bioinformatics" Journal [CBCG+13] and the 12th International Workshop on Network Tools and Applications in Biology, Nettab 2012 (poster) [CCBF+12]):

- We identify and automatically detect a set of anti-patterns that contribute to the structural workflow complexity.
- We design a series of *refactoring* transformations to replace each anti-pattern by a new semantically-equivalent pattern with less redundancy and simplified structure.
- We introduce a *distilling* algorithm that takes in a workflow and produces a distilled semantically-equivalent workflow.
- We provide an implementation of our refactoring approach that we evaluate on both the public Taverna workflows and on a private collection of workflows from the BioVel project.

1.4 Thesis Structure

This thesis is organized as follows:

- Chapter 1 gives the introduction of this thesis by stating the motivation, research problems and contributions.
- Chapter 2 presents a collection of mathematical notations used throughout the rest of this dissertation (2.1). Based on such notations, the workflow model used in this dissertation is introduced (2.2). We then give an introduction to series-parallel graphs and their properties (2.3). At the end of chapter 2, we provide an introduction to the workflows of Taverna system, which is the system that we have chosen to mainly work on.

- Chapter 3 starts with related work on provenance models (3.1), and then proposes a model to represent the provenance of workflow executions (3.2). Later we give a definition of the notion of provenance-equivalence which can be used to identify whether two workflows have the same meaning (3.3). Finally, a discussion about extending our provenance model to better support lists of data is given (3.4).

- Chapter 4 first gives an in-depth explanation of the motivation of rewriting non-SP workflows into SP workflows (4.1). Then we introduce the concept of measuring the distance from non-SP to SP, which inspires some transformation techniques of rewriting non-SP graphs into SP graphs. (4.2). We then analyze the existing strategies to identify whether they are provenance-preserving and propose a new provenance-equivalent strategy (4.3). We introduce the *SPFlow* algorithm for transforming non-SP graphs into SP graphs and discuss the complexity and soundness of the algorithm in 4.4. We demonstrate the feasibility of our approach on real scientific workflows in 4.5. We finally present a tool with the same name of our algorithm, which takes in a non SP Taverna workflow and provide an SP version of the workflow usable in Taverna (4.6).

- Chapter 5 first gives a deep explanation of the second research problem we have considered by presenting several use cases (5.1). Then we introduce the anti-patterns we have identified and the transformations we propose to do while ensuring that the semantics of the workflow remains unchanged (5.2). We then introduce the DistillFlow refactoring algorithm (5.3). In 5.4, we provide the results obtained by our approach on a large set of real workflows. Finally, we discuss several points related to our approach.

- Chapter 6 gives the conclusions and the future works.

Chapter 2

Preliminaries

Contents

2.1 Basic graph concepts and notations **10**

2.2 General workflow model **15**

2.3 Series-Parallel graphs **16**

2.3.1 Definitions 16

2.3.2 SP reduction 18

2.3.3 Properties of SP graphs related to their recognition 19

2.4 Workflows in Taverna **20**

2.5 Summary **22**

Workflows in general, and scientific workflows in particular, are directed graphs where the nodes represent tasks, and the edges represent the relations between the tasks [TDGS07]. Various operations can be performed on scientific workflows, such as designing workflows, visualizing them, querying repositories of workflows, executing workflows (involving scheduling executions, indexing executions, $\cdots$). Each of the operations is then intrinsically related to complex operations on graph structures: clustering graphs, comparing graphs, leading to the problem of (sub)graph isomorphism. Such operations are then very time-consuming on general graph structures while they can be sloved more easily when a particular structure of graphs is considered. With this respect, a special kind of graphs named "series-parallel" graphs (SP graphs) is a useful class of graphs which are simple and layered, and their edges do not intersect, making the main phases of workflow easier to distinguish. More efficient solutions for workflow operations can thus be carried out when SP structures are considered.

This chapter mainly presents a collection of mathematical notations used throughout the rest of this dissertation (2.1). Based on such notations, the workflow model used in this dissertation is introduced (2.2). We then give an introduction to series-parallel graphs and their properties (2.3). At the end of this chapter, we provide an introduction to the workflows of Taverna system, which is the system that we have chosen to mainly work on.

2.1 Basic graph concepts and notations

We define here the basic concepts related to graphs and introduce the notations used in this dissertation, mainly adapted from [BKS92, Val78, Esc03].

We use upper case alphabetic characters $(A, B, C, \cdots)$ to denote sets, and use lower case $(a, b, c, \cdots)$ to denote the elements of a set.

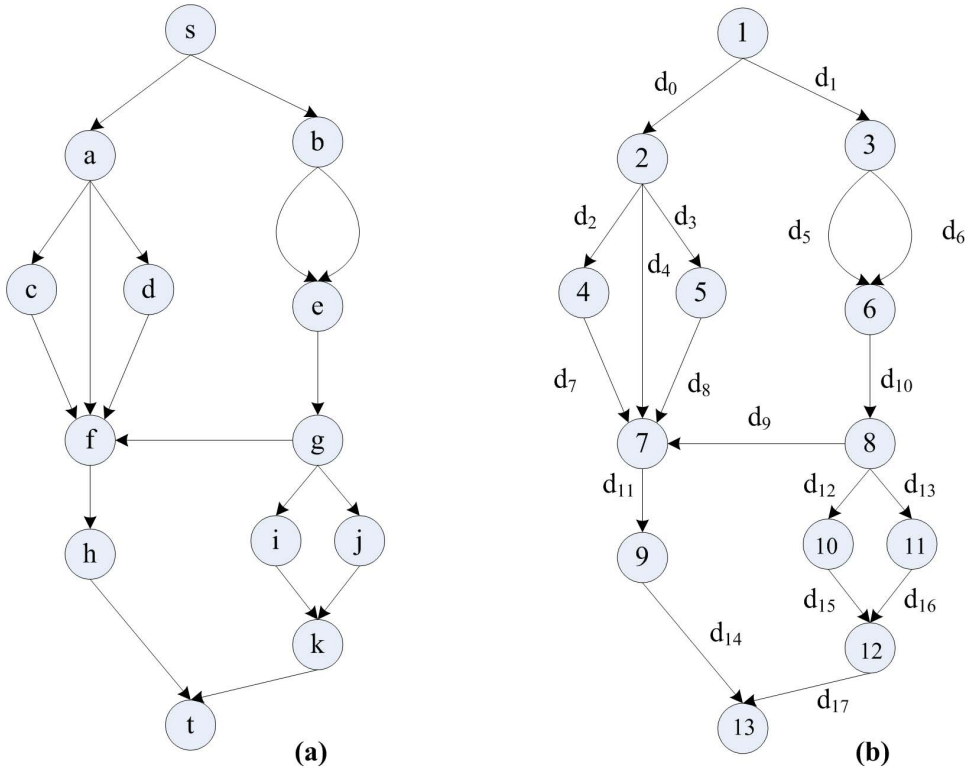


Figure 2.1: Example of dag. (a) a dag, (b) a labeled graph of (a).

Definition 2.1.1 [Val78] A **Directed Graph(digraph)** $G =< V, E >$, consists of a finite set of vertices V and a finite set of edges $E \subseteq V \times V$ which are ordered pairs. The number of vertices is denoted by $n = |V|$, and the number of edges by $m = |E|$.

We allow multiple edges between the same two vertices in the graph. The graphs with multiple directed edges are called **multidigraphs**. Cycles will not be considered in our study. Thus, the graphs used in our study are **Acyclic multidigraphs**, abbreviated as **multidag** (cf. Figure 2.1 (a) with multi edges between vertices b and e).

Definition 2.1.2 Let $G = (V, E)$ be a *multidag*. For each edge $e \in E \subseteq V \times V$, which is denoted by (u, v) or $e(u, v)$, in E, u is the **source** of the edge and v is the **target** of the edge.

Definition 2.1.3 Let $G = (V, E)$ be a *multidag*. For each vertex $v \in V$, the **indegree**, $d^-(v)$, is the number of edges that end by v (means v is the target of the edges) and the **outdegree**, $d^+(v)$ is the number of edges that start from v (means v is the source of the edges). More formally we have:

$$d^-(v) = |\{e(u, v) \in E\}|$$

$$d^+(v) = |\{e(v, u) \in E\}|$$

Example 2.1.1 In Figure 2.1 (a), $d^-(a) = 1$ and $d^+(a) = 2$.

Definition 2.1.4 Let $G = (V, E)$ be a *multidag*. The **successors** set of a vertex $v \in V$ is the set of target vertices of edges outgoing from v, denoted by $Succ(v)$. The **predecessors** set of a vertex v is the set of source vertices of edges for which v is the target, denoted by $Pred(v)$. More formally we have:

$$Succ(v) = \{u : e(v, u) \in E\}$$

$$Pred(v) = \{u : e(u, v) \in E\}$$

Example 2.1.2 In Figure 2.1 (a), $Succ(f) = \{h\}$ and $Pred(f) = \{a, c, d, g\}$.

Definition 2.1.5 Let $G = (V, E)$ be a *multidag*. A **source** of a graph is a vertex v with $d^-(v) = 0$. A **target** of a graph is a vertex v with $d^+(v) = 0$. $S(G)$ is the set of all sources in G, and $T(G)$ is the set of all targets in G.

$$S(G) = \{v \in V : d^-(v) = 0\}$$

$$T(G) = \{v \in V : d^+(v) = 0\}$$

Example 2.1.3 As in Figure 2.1 (a), $S(G) = \{s\}$ and $T(G) = \{t\}$.

Definition 2.1.6 Let $G = (V, E)$ be a *multidag*. A **path** is an ordered sequence of vertices $p(v_1, v_k) = [v_1, v_2, \cdots, v_k]$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$.

To distinguish paths that have the same source and the same target, we use $p(u, v)_x$ to denote the path $p(u, v)$ which contains vertex x. If there is only one single path from u to v or it consists of a single edge, we denote it as $p(u, v)$.

A **full path** is a path p(u,v) where u is a source of G and v a target of G.

Example 2.1.4 In Figure 2.1 (a), $p(a, f)$ is the single path consists of edge $e(a, f)$ and $p(a, f)_c = [a, c, f]$ while $p(a, f)_d = [a, d, f]$, and path $p(s, t)_c = [s, a, c, f, h, t]$ is a full path.

Definition 2.1.7 [Esc03] A vertex v is said to be **reachable** in the multidag G from another vertex v' iff there exists a path $p(v', v)$.

Example 2.1.5 Consider Figure 2.1 (a) again. c is reachable from a, but c is not reachable from b.

Definition 2.1.8 [Val78, Esc03] A multidag $G = (V, E)$ is **transitive** iff if there is a path $p(u, v)$ in G, there also exists an edge $e(u, v)$. The **transitive closure** of $G = (V, E)$ is another graph $G_{tc} = (V_{tc}, E_{tc})$ where $V_{tc} = V$ and E_{tc} is the minimal subset of $V \times V$ that includes E and makes G_{tc} transitive.

Definition 2.1.9 [Val78, Esc03] An edge $e(u, v)$ of a multidag is **redundant** if there is a path $p(u, v)$ not including the edge. The **transitive reduction** of a multidag is the multidag obtained by removing all the redundant edges.

Example 2.1.6 The edge $e(a, f)$ in Figure 2.1 (a) is redundant, because there are paths $p(a, f)_c$ and $p(a, f)_d$ which do not include edge $e(a, f)$.

Definition 2.1.10 [BKS92] A multidag G is an **st-multidag**, also called **two-terminal multidag**, if there exists exactly one source and exactly one target in G.

Example 2.1.7 Figure 2.1 (a) is an st-multidag with source s and target t.

A multidag may have several sources and targets. As we will see in the following, we will consider graphs with one source and one target (as classically when there is a need to compare graph structures). We will thus introduce the notion of generalized st-multidag to define graphs that we "root". In such structures, any vertex will appear in a path from the source to the target.

Definition 2.1.11 The **generalized st-multidag** $G_{st} = (V_{st}, E_{st})$ of a multidag $G = (V, E)$ is a two-terminal multidag, constructed from G, by adding at most two vertices v_s, v_t and $O(n)$ edges as follows:

$$V_{st} = V \cup \{v_s\} \;\; and \;\; E_{st} = E \cup \{e(v_s, v) : v \in S(G)\} \qquad if \;\; |S(G)| > 1$$

$$V_{st} = V \cup \{v_t\} \;\; and \;\; E_{st} = E \cup \{e(v, v_t) : v \in T(G)\} \qquad if \;\; |T(G)| > 1$$

If $|S(G)| = 1$ and $|T(G)| = 1$, then $G_{st} = G$.

Example 2.1.8 Let us consider Figure 2.2 with only solid lines. In (a), $S(G) = \{1, 2\}, T(G) = \{t\}$, because $|S(G)| = 2 > 1$, we should add a single source "s" and edges $e(s, 1), e(s, 2)$ to G_0 (represented with dashed lines). We do the same process for (b),(c) and (d). In (d), nothing is added. It means that G_3 itself has already a single source and a single target, thus G_3 is an st-multidag.

Property 2.1.1 Properties of st-multidags [Val78, Esc03] :

1. Any vertex in an st-multidag is reachable from the source.

2. The target of an st-multidag is reachable from any vertex in the graph.

3. For any vertex $v \in V$ there exists at least one full path that contains v.

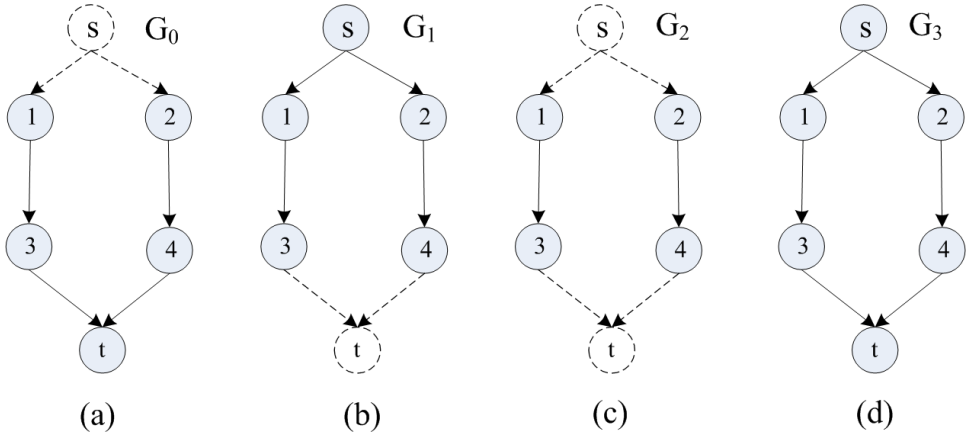


Figure 2.2: Example of generalized st-multidags. For each graph, the vertices and edges drawn in dashed lines are the vertices and edges that should be added to the initial graph which is drawn in solid lines to get an st-multidag. (a) adding a source; (b) adding a target; (c) adding both a source and a target; (d) the graph is an st-multidag.

Definition 2.1.12 [Wik13b] Let $G = (V, E)$. A **labeling** of G is a function $\ell : V \cup E \to L$ for some set L of labels. For every x in the domain of ℓ, the $\ell(x) \in L$ is called the label of x. Three of the most common types of labelings of a graph G are:

1. total labeling: ℓ is a total function (defined on all $V \cup E$),

2. vertex labeling: the domain of ℓ is V, and

3. edge labeling: the domain of ℓ is E.

A **labeled graph** is a pair (G, ℓ) where G is a graph and ℓ is a labeling of G.

Example 2.1.9 Figure 2.1 (b) is a labelled graph for (a) in which $L = L_V \cup L_e$, with $L_v = \{1, 2, 3, \cdots, 13\}$ and $L_e = \{d_1, d_2, \cdots, d_{17}\}$ and $\ell(a) = 2, \ell(b) = 3, \cdots$ and $\ell(e(s, a)) = d_0, \ell(e(s, b)) = d_1, \cdots$.

We now have all the concepts needed to define the workflow model as the basis of our provenance model.

2.2 General workflow model

A workflow model has two components [CCPP99, BBD+09]: a *specification* that serves as a template for executions, and a set of *runs* for the given specification. Informally, a workflow specification consists of a set of different modules and defines the order in which they can be executed. A workflow run is a partial order of steps where each step is an instance of a module defined in the underlying specification, and the partial order conforms to the ordering constraints in the given specification. However, in this thesis we work on a workflow structure together with its provenance information, and we consider the workflow run that has the same graph structure as the workflow specification based on the constraints of the Taverna system detailed in the last section.

Formally, we model a workflow specification as a directed acyclic labeled multigraph whose vertices represent the workflow tasks and edges represent the *data flow* between tasks.

As most scientific workflow systems allow only stateless, functional behavior, and do not allow looping [DF08], we consider st-multidags. Because scientific workflows do not contain a unique source and a unique target, for each specification and its runs, we consider their generalized st-multidags, that is, we add when necessary one single source and one single target and corresponding edges to "root" the workflow .

Definition 2.2.1 [CBFJ12] A **workflow specification** is an st-multidag $G_{spec} = (V_{spec}, E_{spec})$ where vertices are labelled by the function $L_{vs} : V_{spec} \rightarrow L_{VS}$, with L_{VS} a set of labels for vertices, which is related to task names.

Definition 2.2.2 [CBFJ12] A **workflow run** is an st-multidag $G_{run} = (V_{run}, E_{run})$ with labeled vertices and labeled edges, using the functions $L_{vr} : V_{run} \rightarrow L_{VR}$, where L_{VR} is a set of labels of the vertices, which is related to task names and $L_{er} : E_{run} \rightarrow L_{ER}$, where L_{ER} is a set of labels of edges, which is related to the data produced by tasks. We will note $\tilde{\mathbf{x}}$ the label of the vertex x, *i.e.* $L_{vr}(x) = \tilde{\mathrm{x}}$ and d_i the label of the edge e_i, *i.e.* $L_{er}(e_i) = d_i$.

2.3 Series-Parallel graphs

In this subsection we examine series-parallel graphs (SP graphs) which are a common type of graph, and have been introduced by Duffin [Duf64] to model electrical networks. They have a significant use in several applications that make them interesting to examine. As stated in [DB99], using SP graphs we can successfully visualize flow diagrams [Wik13a], dependency charts [RG00], and PERT networks [htt13]. The construction of series-parallel DAGs and their relation with general DAGs are the main focus of this section. We present here formal definitions and properties of this kind of graphs. The following definitions are adapted mainly from [BKS92, Val78].

2.3.1 Definitions

Definition 2.3.1 The class of **series-parallel graphs (SP-graphs)** is recursively defined as follows:

1. **Basic SP graph**: G, the st-multidag that contains two vertices s and t joined by a single edge is an SP-graph (called "**BSP**");

2. **Series Composition**: if G_1 (source s_1 and sink t_1) and G_2 (source s_2 and sink t_2) are two SP-graphs, G obtained by identifying s_2=t_1 is an SP-graph with source s_1 and sink t_2;

3. **Parallel Composition**: if G_1 (source s_1 and sink t_1) and G_2 (source s_2 and sink t_2) are two SP-graphs, G obtained by identifying $s_1 = s_2$, $t_1 = t_2$ is an SP-graph with source s_1 and sink t_1.

The above definition can be understood by inspecting Figure 2.3. In this figure we construct the parallel composition of the basic graphs by joining the sources at the top and sinks at the bottom (see Figure 2.3 (b)). Similarly we construct the series composition by joining the sink with the source of the two basic graphs(see Figure 2.3 (c)). In the same way, we can compose more complex graphs by combining these compositions.

Definition 2.3.2 A st-multidag is **non-SP** iff it is not an SP graph.

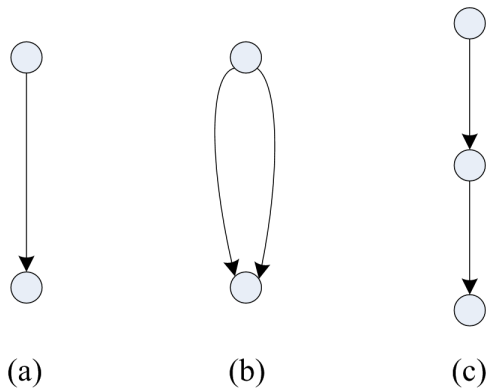


Figure 2.3: Recursive construction of SP graphs: (a) Basic SP graph (BSP), (b) parallel composition, (c) series composition.

Another way to define the class of SP graphs is to state that they do not contain a subgraph homeomorphic (intuitively, similar) to a "forbidden subgraph" shown in Figure 2.4. In other words, such a graph does not contain any series components or parallel components due to an "across edge" inside the subgraph. This forbidden subgraph represents the basic characteristic of non-SP graphs. More formally:

Definition 2.3.3 [Esc03] An **induced subgraph** $G' = (V', E')$ of another graph $G = (V, E)$, is obtained by eliminating some vertices from V and eliminating from E the edges incident to those eliminated vertices, formally:

$$G' \subseteq G \; iff \; V' \subseteq V, E' = \{(u, v) \in E : u, v \in V'\}$$

Definition 2.3.4 [Val78, Esc03] A graph $G = (V, E)$ is **homeomorphic** to another graph G' iff its transitive closure G_{tc} contains G' as an induced subgraph:

$$G \; homeomorphic \; to \; G' \; iff \; G_{tc} \supseteq G'$$

Theorem 2.3.1 [Duf64] An st-multidag is series-parallel iff it does not contain a subgraph homeomorphic to the "forbidden subgraph" of Figure 2.4. (See proof in [Duf64])

Example 2.3.1 The transitive closure of Figure 2.1 (a) contains an induced subgraph $G = (V, E)$ *with* $V = \{s, f, g, t\}$ *and* $E = \{(s, f), (s, g), (g, f), (f, t), (g, t)\}$ which is a forbidden subgraph. According to the definition 2.18, the graph of Figure 2.1 (a) is homeomorphic to the forbidden subgraph of Figure 2.4, as a consequence Figure 2.1 (a) depicts a graph which is a non-SP graph.

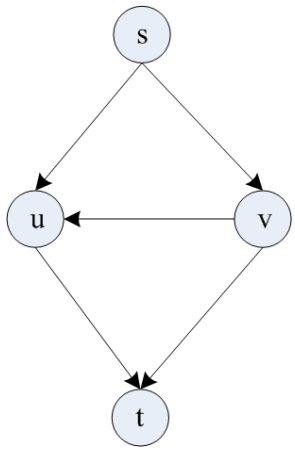


Figure 2.4: The forbidden subgraph for SP-graphs

2.3.2 SP reduction

This subsection introduces techniques able to determine whether a graph is an SP graph. Two operators of reduction have been proposed. The result of using each of these operators in simple graphs is shown in Figure 2.5.

As said in the definition of the workflow model, the labels for vertices are related to the tasks and the labels for edges are related to the data produced by the tasks. Moreover, it is important to save the label information which is related to the provenance trace during each reduction operation. Taking this into account, the reduction operators are defined as follows:

Definition 2.3.5 Let $G_1 = (V_1, E_1)$ be an st-multidag whose vertices and edges are labeled by the functions L_{1vr}: $V_1 \rightarrow L_{VR}$, and L_{1er}: $E_1 \rightarrow L_{ER}$. The **elementary operation** op transforms G_1 into an st-multidag $op(G_1) = G_2 = (V_2, E_2)$, whose vertices and edges are labeled by the functions $L_{2vr} : V_2 \rightarrow L_{VR}$, and $L_{2er} : E_2 \rightarrow (L_{VR} \cup L_{ER}, +, \cdot)$.

1. **Series Reduction.** Let $u, v, w \in V_1$, such that $e = (u, v)$ is the unique incoming edge of v and $f = (v, w)$ is the unique outgoing edge of v. The operation op_{sr} of **Series Reduction** in v replaces e and f by $g = (u, w)$. $G_2 = (V_2, E_2)$ is such that $V_2 = V_1 - \{v\}$, L_{2vr} is the restriction of L_{1vr} on V_2, $L_{2er} = L_{1er}$ on $E_1 \cap E_2$, and $L_{2er}(g) = L_{1er}(f) \cdot L_{1vr}(v) \cdot L_{1er}(e)$.

2. **Parallel Reduction.** Let $v, w \in V_1$ linked by k edges $e_1 \cdots e_k$. The operation op_{pr} of **parallel reduction** in v and w replaces the k edges by

a unique edge $g = (v, w)$ and leaves all the remaining edges unchanged. $G_2 = (V_2, E_2)$ is such that $V_2 = V_1$, $L_{2vr} = L_{1vr}$, L_{2er}=L_{1er} on $E_1 \cap E_2$, and $L_{2er}(g) = (L_{1er}(e_1) + ... + L_{1er}(e_k))$.

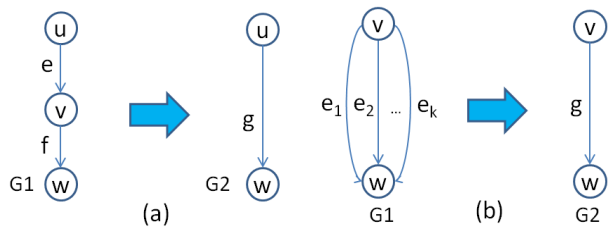


Figure 2.5: (a) Series reduction; (b) Parallel reduction

Definition 2.3.6 Let G be an st-multidag. G is **maximally reduced ("MaxRed")** if and only if no series or parallel reduction can be applied to it.

Definition 2.3.7 A **maximal SP reduction graph** of G, is another graph G_{red} obtained by using all possible series and parallel reduction operations (*i.e.* MaxRed) on G:

$$G_{red} = MaxRed(G)$$

We now introduce a set of properties of SP graphs.

2.3.3 Properties of SP graphs related to their recognition

Determining whether a graph is SP is associated to several properties that we provide below.

Property 2.3.1 [Val78]: Let G be an st-multidag. G is **SP** if and only if there exists a sequence of series and parallel reductions that reduces G to BSP.

Property 2.3.2 Performing series and parallel reduction operations in any order on Graph G to get the BSP will allow to obtain the same resulting graph. (Church-Rosser property [VTL82]).

Figure 2.6 describes reductions performed on the st-multidag of G_0. As only series and parallel reductions are used to reduce graph G_0 to BSP, the initial graph G_0 is thus SP.

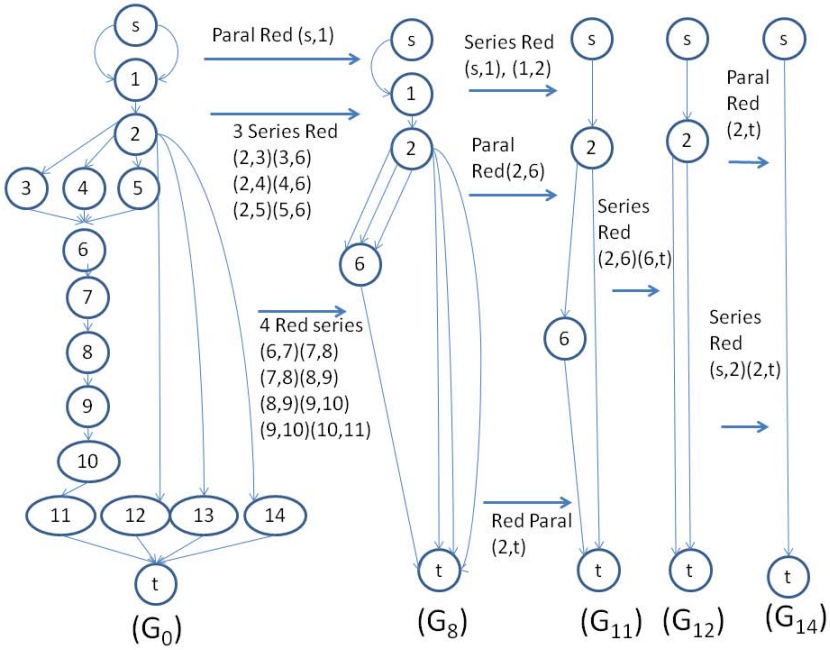


Figure 2.6: Example of reduction operations applied to G_0.

Interestingly, SP graphs are a subclass of planar graphs, and also a subclass of k-terminal graphs (see e.g. [Bod94]). SP graphs are equivalent to partial 2-trees, a subclass of bounded tree-width graphs (see e.g. [Bod94]). Based on the properties of these graph classes, linear time complexity algorithms to recognize SP-graphs are possible.

Property 2.3.3 [Sch95, VTL82]: The recognition of a series-parallel DAG can be done in linear time.

Efficient parallel algorithms for recognizing SP graphs have also been proposed in [BDF96, HHC99, HY87, Epp92].

Now that all the graph-related definitions have been introduced to represent workflows, the next subsection introduces the concrete form of workflows we work on.

2.4 Workflows in Taverna

This subsection gives an introduction to Taverna workflows, because our work currently is mainly based on the Taverna workflow model [HWS+06]. Taverna combines a dataflow model of computation with a functional model that accounts

for list data processing. Examples of Taverna workflows are given throughout this dissertation, and especially in Chapter 5. A workflow consists of a set of processors, which represent software components such as Web Services and may be connected to one another through data dependencies links. This can be viewed as a directed acyclic graph in which the nodes are processors, and the links specify the data flow. Figure 2.7 (a) provides one example of workflow and Figure 2.7 (b) gives the corresponding graph (nodes have been renamed for the shake of clarity). Processors have named input and output ports, and each link connects one output port of a processor to one input port of another processor. A workflow has itself a set of input and output ports, and thus it can be viewed as a processor within another workflow, leading to structural recursion.

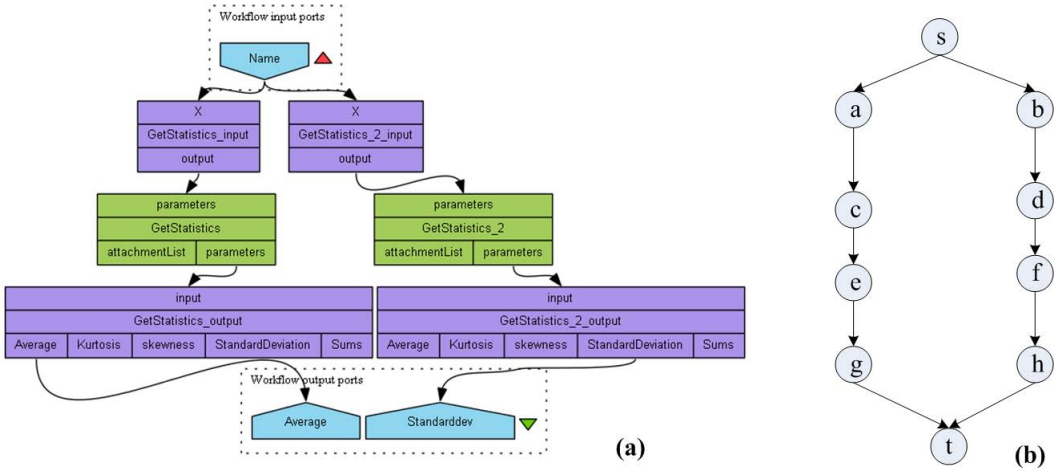


Figure 2.7: An example of Taverna workflow ((b) is the specification graph for (a))

The workflow depicted in Figure 2.7 (a), for instance, has one input called *Name* and two outputs named respectively *Average* and *Standarddev*. In turn, processor *GetStatistics_output* has one input port named *input* and five output ports named *Average*, *Kurtosis*, *Skewness*, *StandardDeviation* and *Sums*. Note that the input and output ports do not appear in the graph representation.

The triple ⟨ <workflow name>, <workflow inputs>, <workflow outputs> ⟩ is called the *signature* of the workflow.

Note that multiple outgoing links from processors or inputs are allowed, as is the case for the workflow input of Figure 2.7 (a) which is used by two processors. Also, not all output ports must be connected to downstream processors (e.g., the

value on output port *attachment_list* in *Get_Statistics* is not sent anywhere), and symmetrically, not all inputs are required to receive an input data (but input ports with no incoming links should have a default value, or else the processor will not be activated).

Input ports are statically typed [MPB10], according to a simple type system that includes just atomic types (strings, numbers, etc.) and lists, possibly recursively nested (*i.e.* the type of a list element may be a list, with the constraint that all sub-lists must have the same depth). The functional aspects of Taverna come into play when one or more list-value inputs are bound to processor's ports which have an atomic type (or, more generally, whose nesting level is less than the nesting level of the input value). In order to reconcile this mismatch in list depth, Taverna automatically applies a higher-order function, the *cross product*, to the inputs. The workflow designer may specify an alternative behavior by using a *dot product* operator instead. This produces a sequence of input tuples, each consisting of values that match the expected type of their input port. The processor is then activated on each tuple in the list. The resulting "implicit iteration" effect can be defined formally in terms of recursive application of the *map* operator [MPB10].

2.5 Summary

This chapter has introduced all the definitions which are at the basis of our work concerning graph structures and has provided a particular focus on Series-Parallel graphs (SP graphs). Such graphs have very interesting features since NP-hard graphs problems posed on general DAGs may be solved by polynomial time algorithms for SP structures. The last section of this chapter gave main terminology of concepts associated to the workflows of the Taverna system, which are the workflows we will mainly study in this thesis.

Chapter 3

Provenance Model

Contents

3.1 Related work **24**

3.2 Our Provenance Model **26**

3.3 Provenance-equivalence **30**

3.4 Conclusion and Discussion **31**

3.4.1 Conclusion 31

3.4.2 Towards "problematic" data dependencies in Taverna . . . 33

3.4.3 Possible solutions 36

3.5 Summary **38**

In this thesis, we are interested in the meaning of the workflow as given by the provenance of its execution outputs. Intuitively, the provenance of a data item is the ordered sequence of tasks performed to produce this data, and input data to each task. Two workflows have the same meaning if, given some input data, they both produce the same intermediate and final data *i.e.* they are *provenance-equivalent*. The aim of this chapter is to introduce a general provenance representation model which is suitable for comparing the structures of the workflow executions which contain provenance information and then give the notion of provenance-equivalence between two workflows.

At the beginning of this chapter, we introduce one simple provenance model [ABML09] that we will call the "basic provenance model" in the following. We then discuss shortcomings of the basic provenance model in accurately capturing data dependencies in several computational scenarios and when complex data is used. We propose a general provenance model that naturally extends the basic provenance model by using regular expressions, to represent scientific workflow

provenance. Based on this general provenance model, the definition of provenance-equivalence is described, which is the basis for evaluating correctness of the two approaches (Chapter 4 and 5) in this dissertation. Finally, a discussion on several "problematic" data dependencies cases is given and possible solutions are drawn.

3.1 Related work

This section aims to show the characteristics of existing provenance models. These characteristics can help us to develop an underlying provenance model for identifying whether two runs are provenance-equivalent. Anand *et al* [ABML09] have compared many workflow systems (e.g., Vistrail [SFC07], Kepler [ABJF06], Taverna [ZGST08] and others [BCC+05, ZWF06, OCE+08]) and provenance approaches(e.g., [BD08, CJR08, HA08, MFF+08, CCBD06]). They found that most workflow systems keep coarse-grained representation of the provenance, and many of them employ a simple provenance model, they called *basic provenance model*. This model can generally be characterized as recording the inputs and outputs for each execution of a processor occurring within a workflow run [Ana10]. Conceptually, the trace of workflow executions in such a model consists of (1) *in-relations* $in(d_x, p)$; (2) *out-relations* $out(p, d_y)$, in which d_x is an input and d_y is an output of the processor p for one execution. These relations describe the observable events that occur during the workflow execution, namely the computation of each instance of each processor. These relations are used to infer data and process dependencies. For example, a run of a simple workflow (see Figure 3.1) is captured by a collection of all observable "in" and "out" events during the execution, says $in(d_x, P_a), out(P_a, d_y), in(d_y, P_b), out(P_b, d_z)$, implying that processor P_a was

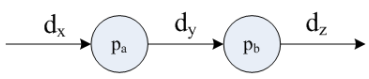


Figure 3.1: A run of a simple workflow

executed directly before processor P_b, and d_y directly depended on d_x while d_z indirectly depended on d_x.

In most systems, each input and output of an execution of a processor can be organized into complex structures: lists in Taverna [HWS+06], trees in Kepler

[BML08], etc. Also, the processors may contain unobservable events, such as filtering input data as a subtask in a processor. But in the basic model described above, we should ignore all the unobservable events and consider processors as black-boxes.

As discussed in [ABML09, CW03, Ana10], this basic provenance model is suitable for representing data and process dependencies of scientific workflows which (1) produce new outputs from their inputs (*i.e.* they do not contain any function that does not change the incoming data); (2) use all inputs to derive their outputs (*i.e.* all outputs of a processor depend fully on all the inputs to the processor). In all of the systems that we have studied, data dependencies are explicitly declared rather than automatically generated from the module functionality specification. These common features are very useful for us to develop a general provenance model to fit most of all these systems. To fit our final goal of rewriting scientific workflow structures which are from most workflow systems, similarly to most systems, we conform our provenance model to the basic model [ABML09]. By nature, the basic model is compatible with the Open Provenance Model (OPM) [OPM] which is a standardized model. The *in-relation* $in(d_x, p)$ and *out-relation* $out(p, d_y)$ simply have different names in OPM: For *in-relation* $in(d_x, p)$, we say in OPM that d_x *was used by* processor p; and for *out-relation* $out(p, d_y)$, we say that d_y *was generated by* processor p.

We aim to capture all the provenance information for a run to identify its equivalent runs, searching for a simpler structure for a scientific workflow. As the *in-relation* corresponds to an input edge and the *out-relation* corresponds to an output edge, the basic model can be adopted for representing the graph structure of a run which contains provenance information. To capture the whole structure of a run, we need a new representation model to organize all the in-relations and out-relations. We thus continue investigating current approaches to find a method that can meet this need.

As in the context of relational databases, provenance representations extend the relational data model with annotations [CTV05], provenance and uncertainty [ABS+06], and semirings of polynomials [GKT07]. In these approaches, we are mainly interested in the concept of provenance semirings proposed by Green *et al.* in [GKT07], in which every tuple of the database is annotated with an element of

a provenance semiring, and annotations are propagated through query evaluation. For example, semiring addition corresponds to alternative derivation of a tuple, thus, the union of two relations corresponds to adding up the annotations of tuples appearing in both relations. Similarly, multiplication corresponds to joint derivation, thus, a tuple appearing in the result of a join will be annotated with the product of the annotations of the two joined tuples. As it is suitable for searching provenance structures and achieving the whole provenance information, which fits our aims, we thus take this concept into account and propose to represent provenance information by regular expressions.

As a result, our approach takes benefits both from the basic model and the concept of using regular expressions to represent provenance. In the next section, we introduce our provenance model, which uses regular expressions to organize all the *in-relations* and *out-relations* for representing provenance trace. The resulting model will be suitable for defining the notion of provenance-equivalence.

3.2 Our Provenance Model

As discussed in the previous section, our aim is to capture the conventional view of scientific workflows, which considers simple task dependency over atomic data and atomic (single invocation) processes. Our new provenance model is naturally compatible with the "basic model" and OPM, and it is based on the graph structures of the scientific workflows.

In the following, we provide definitions of provenance.

Formally, let $G_{run} = (V_{run}, E_{run})$ be a run, with its sets of labels for vertices and edges. We consider regular expressions built on $L_{VR} \cup L_{ER}$, using operations "+" and "·". Both operations are associative, "+" is commutative and "·" is right distributive over "+". Operation "·" allows to track the succession of the tasks, while "+" denotes the alternative data paths reaching a task. Indeed, the "+" operation is commutative because several parallel input data can be considered in any order and the "·" operation is not commutative, because the execution order must be taken into account in our context [CBFJ12].

We distinguish immediate provenance to describe the last step of production and deep provenance to describe the entire sequence of steps that produced the

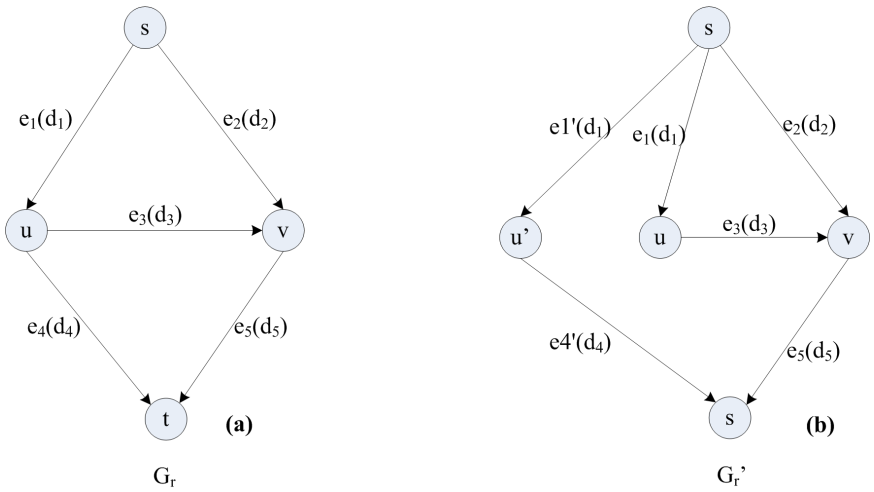


Figure 3.2: Two graph illustrating provenance related notions

data [BBDH08].

Definition 3.1: **Provenance of a data item.**
Let $u \in V_{run}, u \neq s(G_{run})$, with $L_{vr}(u) = \widetilde{u}$; $f \in E_{run}$ one outgoing edge of u with $L_{er}(f) = d$; $e_i \in E_{run}$, $1 \leq$ i $\leq p$ the incoming edges of u, with $L_{er}(e_i) = d_i$.

The **Immediate Provenance** of f in G_{run} is defined by $imProv : E_{run} \rightarrow (L_{VR} \cup L_{ER}, +, \cdot)$, with:

$$imProv(f) = \widetilde{u} \cdot (d_1 + \cdots + d_p)$$

The **Deep Provenance** of f in G_{run} is recursively defined by $DProv : E_{run} \rightarrow (L_{VR} \cup L_{ER}, +, \cdot)$, with:

$$DProv(f) = \widetilde{u} \cdot (d_1 \cdot DProv(e_1) + \cdots + d_p \cdot DProv(e_p))$$

The base case occurs when $u = s(G_{run})$ and f is an outgoing edge of s:

$$DProv(f) = imProv(f) = \widetilde{s}$$

Example 3.2.1 Consider the graph G_r of Figure 3.2 (a). The immediate provenance of data d_5 flowing in edge e_5 is given by task v, directly taking d_2 and d_3 as its inputs. This information can be represented as: $imProv(e_5) = \tilde{v} \cdot (d_2 + d_3)$. We also have:

$$DProv(e_5) = \tilde{v} \cdot (d_2 \cdot DProv(e_2) + d_3 \cdot DProv(e_3)) = \tilde{v} \cdot (d_2 \cdot \tilde{s} + d_3 \cdot [\tilde{u} \cdot d_1 \cdot \tilde{s}])$$

This formula expresses that the data d_5 flowing in edge e_5 has been obtained from task v which took d_2 and d_3 as inputs, in which d_2 is obtained directly from the source s while d_3 requires one more additional recursion step of u which took d_1, obtained by the source s, as input.

It may be interesting to only know the data involved in the production of a given item regardless of the order in which they were consumed or which tasks were executed.

Definition 3.2.1 Given a run $G_{run} = (V_{run}, E_{run})$, let d be the label of an edge f in G_{run}. Let d_i be any edge label appearing in $DProv(f)$, we say that d **depends** on d_i.

It is important to note that we cannot directly use the set of d_i to evaluate the equivalence of two provenances, for it may have different dependencies among the set of d_i, also the number of times of each data used is unknown. However, a workflow run graph $G_{run} = (V_{run}, E_{run})$ also gives rise to a natural view, **a data dependency graph** $G_d = (V_d, E_d)$, in which vertices represent data production and edges represent process dependencies, thus all the dependencies of each data are visualized. Formally, we have:

Definition 3.2.2 A **data dependency graph** $G_d = (V_d, E_d)$ for a run $G_{run} = (V_{run}, E_{run})$ is a labeled multidag with $V_d = \{L_{er}(e) | e \in E_{run}\}$ and $E_d = \{(u, v) | e_1(x, y), e_2(y, z) \in E_{run}\ and\ L_{er}(e_1) = u, L_{er}(e_2) = v\}$, with labelled edges, using the function $L_{ed} : E_d \rightarrow L_{VR}$, where L_{VR} is the set of labels of the vertices of G_{run}. We will note $L_{vr}(y)$ the label of the edge $e = (u, v) \in E_d$ such that $e_1(x, y), e_2(y, z) \in E_{run}\ and\ L_{er}(e_1) = u, L_{er}(e_2) = v$, i.e. $L_{ed}(e(u, v)) = L_{vr}(y)$.

Figure 3.3 shows the data dependency graphs for the runs in Figure 3.2. Dependency graphs are natural views of runs, they have the same data and process dependencies. Of course, all the dependencies of data and processes can be directly obtained from a run itself, by considering in the run all the data items as vertices and all the tasks as edges which link two data items together by taking one data item as input and producing another one as output. Obviously, the two runs have the same structures of data dependency graphs.

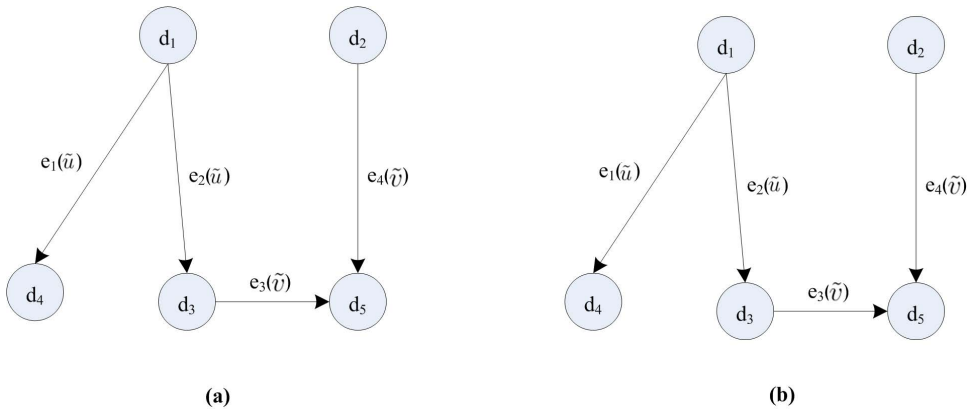


Figure 3.3: Data dependency graphs for runs in Figure 3.2. (a) data dependency graph for G_r; (b) data dependency graph for $G_{r'}$

We now introduce the fundamental concept of *output provenance* of a run, that we define as the **history** of the target of the workflow. The history of a task is closely linked to the provenance of data produced by a task. Formally:

Definition 3.2.3 Let $G_{run} = (V_{run}, E_{run})$ be a run and $DProv$ the function defining the Deep provenance on G_{run}. The **history** of u in G_{run} is given by the function $Hist : V_{run} \rightarrow (L_{VR} \cup L_{ER}, +, \cdot)$:
(i) if $u = s(G_{run})$, $Hist(s) = \varepsilon$ (empty word)
(ii) if $u \neq s(G_{run})$, let $e_i \in E_{run}$ be the incoming edges of u $(1 \leq i \leq p)$, with $L_{er}(e_i) = d_i$: $Hist(u) = d_1 \cdot DProv(e_1) + \ldots + d_p \cdot DProv(e_p)$.

Example 3.2.2 Consider again the graph G_r of Figure 3.2 (a): $Hist(v) = d_2 \cdot \tilde{s}$.

Definition 3.2.4 Output Provenance of a Run. Given a run G_{run}, and its history function $Hist$, its **output provenance** is defined by: $OutProv(G_{run}) = Hist(t(G_{run}))$.

Example 3.2.3 Continuing with Figure 3.2 (a), and using associativity of "$\cdot$" and "$+$" we get $OutProv(G_r) = (d_4 \cdot \tilde{u} \cdot d_1 \cdot \tilde{s}) + (d_5 \cdot \tilde{v} \cdot (d_3 \cdot \tilde{u} \cdot d_1 \cdot \tilde{s} + d_2 \cdot \tilde{s}))$. The output provenance of (G_r) is the sum of the provenances of e_4 and e_5.

Remarks on Provenance expressions. Several equivalent expressions for provenance are possible, due to associativity, commutativity and distributivity

properties. As the deep provenance is recursively defined, the duplicated sub-expressions in the provenance expression cannot be avoided, which will lead to redundancy in the expression. As in Figure 3.2(a), $OutProv(G_r) = (d_4 \cdot \tilde{u} \cdot d_1 \cdot \tilde{s}) + (d_5 \cdot \tilde{v} \cdot (d_3 \cdot \tilde{u} \cdot d_1 \cdot \tilde{s} + d_2 \cdot \tilde{s}))$. We can find that the name of d_1 and the name of u occur twice, like the sub-expression $\tilde{u} \cdot d_1 \cdot \tilde{s}$ and the name of "$\tilde{s}$" occur three times. It means that it is possible to obtain an equivalent expression by following several factoring rules to eliminate some redundancy of duplicated sub-expressions. Right distributive can be used to provide a concise representation of provenance through the following **right factorization rule**:

$$(\alpha_1 \cdot z \cdot \beta + \alpha_2 \cdot z \cdot \beta) \rightarrow (\alpha_1 + \alpha_2) \cdot z \cdot \beta$$

where α_1, α_2 and β are expressions built on vertex and edge labels using "$+$" and "$\cdot$" and z is a vertex label. E.g., we have $OutProv(G_r) = (d_4 \cdot \tilde{u} \cdot d_1 + d_5 \cdot \tilde{v} \cdot (d_3 \cdot \tilde{u} \cdot d_1 + d_2)) \cdot \tilde{s}$, which is more concise than the one above.

Note that given a run G_{run} and a vertex u, all the outgoing edges of u have the same provenance. *I.e.* all the outputs of the same task have the same provenance, as they were (recursively) obtained in the same way.

We now have all the concepts needed to define the provenance-equivalence of two executions that is the subject of the following subsection.

3.3 Provenance-equivalence

In this research, we aim to transform a workflow structure to an SP structure while ensuring that the transformed workflow will work the same as the original workflow. So, how to identify whether two workflow executions have the same provenance structures becomes especially important. We thus introduce here the notion of **provenance-equivalence** of two workflow executions, which is defined as follows.

Definition 3.3.1 Let G_{r1}, G_{r2} be two runs. G_{r1} and G_{r2} are **provenance-equivalent**, noted $G_{r1} \overset{\text{prov}}{\Longleftrightarrow} G_{r2}$, iff $OutProv(G_{r1}) = OutProv(G_{r2})$.

Example 3.3.1 Consider Graphs G_r (a) and G'_r (b) of Figure 3.2. G'_r has been obtained from G_r by duplicating vertex u into vertex u' with the same label. In

the same way, edges e_1 and e'_1 have the same label, together with edges e_4 and e'_4.
$OutProv(G'_r) = [L_{er}(e'_4) \cdot L_{vr}(u') \cdot L_{er}(e'_1) \cdot L_{vr}(s)] + [L_{er}(e_5) \cdot L_{vr}(v) \cdot [L_{er}(e_2) \cdot L_{vr}(s) + L_{er}(e_3) \cdot L_{vr}(u) \cdot Ler(e_1) \cdot L_{vr}(s)]]$.
As $L_{vr}(u') = L_{vr}(u) = \widetilde{u}, L_{er}(e'_4) = L_{er}(e_4) = d_4$ and $L_{er}(e'_1) = L_{er}(e_1) = d_1$, we have:
$OutProv(G'_r) = [d_4 \cdot \widetilde{u} \cdot d_1 \cdot \widetilde{s}] + [d_5 \cdot \widetilde{v} \cdot (d_2 \cdot \widetilde{s} + d_3 \cdot \widetilde{u} \cdot d_1 \cdot \widetilde{s})]$, which is exactly $OutProv(G_r)$.
Thus: $OutProv(G_r) = OutProv(G_{r'})$.
So, we say that G_r and G'_r in Figure 3.2 are provenance-equivalent.

3.4 Conclusion and Discussion

3.4.1 Conclusion

We have introduced a model of provenance which is compatible with the Open Provenance Model (OPM) [OPM]. In the OPM, an atomic data structure d is called an artifact, an invocation of a processor p is called a process, an in-edge e to a processor p with $L_{er}(e) = d$ corresponds to an *used* edge $d \overset{\text{used}}{\dashleftarrow} p$, and an out-edge f from a processor p with $L_{er}(f) = d$ corresponds to a *wasGeneratedBy* edge $p \overset{\text{genBy}}{\dashleftarrow} d$. Similarly, the above expressions built on vertex and edge labels using "+" and "·" has several patterns:

(1) $d \cdot p$ we say in OPM that the artifact d *wasGeneratedBy* the process p.

(2) $p \cdot d$ we say in OPM that the process p *used* the artifact d.

(3) $d_1 \cdot p \cdot d_2$ we say in OPM that the artifact d_1 *was derived from* the artifact d_2.

(4) $p_1 \cdot d \cdot p_2$ we say in OPM that the process p_1 *was triggered by* the process p_2.

Our model is currently useful for representing data and processing dependencies of scientific workflows consisting primarily of black-box transformations, which means that the output of a processor should fully depend on all the inputs. Also note that our model makes use of semirings with several constraints (e.g., "·" is not commutative) for the execution order must be taken into account in our context.

The regular expressions of provenance thus can well capture the whole structure of the executions, which can be used to compare whether two runs are equivalent.

However, not all the scientific workflow systems follow the assumption that considers processors as black-boxes. For example, as discussed by Anand *et al* in [ABML09], many systems (e.g., [MBZL09, MBZ+08, QF07]) and approaches (e.g., [FCB07, KS07, MAA+05, Wal07]) support processors that make only small changes or updates to incoming data, passing on some or all of their input to downstream processors. This means that a processor may take a collection of data values as inputs and produce a new collection as outputs, such as:

$$d_x = [d_1, d_2, \cdots, d_{x0}], d_y = [d_1, d_2, \cdots, d_{y0}]$$

In this case, d_{x0} is changed into d_{y0} and we assume that d_{y0} depends only on d_{x0}. In our model, d_y should fully depend on d_x, which may imply that not only d_{y0} depends on d_{x0}, but also d_{y0} depends on the d_i. As a result, it may lead to a wrong provenance meaning.

Furthermore, various patterns of data dependencies in collection-oriented approaches [CW03,MBK+08,MBZ+08,BML+06,QF07] can arise, so that not all parts of the output depend on all parts of the input. Let us assume that a processor receives input d_x and produces output d_y as follows:

$$d_x = [d_{x1}, d_{x2}, \cdots, d_{xp}], d_y = [d_{y1}, d_{y2}, \cdots, d_{yp}]$$

in which d_{xi} is changed into d_{yi}. Obviously, our model gives coarse-grained provenance information rather than fine-grained provenance information, which means that our model currently cannot achieve data items inside a collection when the collection is considered as a data structure in the scientific workflow systems. Such coarse-grained data dependency can achieve the correct provenance structure for comparing two runs. However, when some special kinds of dependencies are combined with this kind of data dependency, it will lead to a not precise enough meaning of provenance. These special cases also have been mentioned by other works [BML+06, Ana10], which include:

(a) processors having subtasks (such as filtering input data) prior to applying a scientific function, resulting in dependencies where each y_i depends only on some of the x_j;

(b) processors that process each input data in turn (take a collection as input), resulting in dependencies where each y_i depends on a single x_j $(j = i)$;

(c) processors that perform running aggregates over their input, resulting in dependencies where each y_i depends on the set $\{x_1, x_2, \cdots, x_i\}$; and

(d) processors that apply functions over their input using sliding windows of a fixed size w, resulting in dependencies where each y_i depends on the window $\{x_{i-w}, x_2, \cdots, x_i\}$.

Our work currently considers the Taverna system. The next subsection provides details of several special "problematic" data dependencies that may occur in Taverna. Some hints for extending the model are provided in 3.4.3.

3.4.2 Towards "problematic" data dependencies in Taverna

In Taverna, there exist two special processors named **merge** and **split processors**. A merge processor only merges several data items into a collection, while a split processor splits a collection of data items into single data [HWS+06]. These kinds of processors do not perform any change on the input values and forward the input values to their destinations. We argue that there may be a misleading on the understanding of provenance information when these kinds of processors appear in a workflow.

Firstly, we consider Figure 3.4, its data dependency graph is shown in Figure 3.5. It is obvious that the sets of intermediate and final data produced by these two runs are the same, which are $d_1, d_2, \cdots, d_6$. The only difference is that the run G_0 contains a merge processor which merges d_1, d_3 into a collection $d_x = [d_1, d_3]$, then processor v takes this collection as input and produces another collection $d_y = [d_4, d_5]$ as output. But in G_1, processor v separately produces d_4 and d_5 in turn. Intuitively, for any processor, except the merge processor, in the two runs, the immediate data and the final data produced are the same. And a merge processor does not do any change to any data value. As a result, without the merge processor, the two runs have the same meaning, *i.e.* they are provenance-equivalent. However, when querying output provenance on the two runs, we will

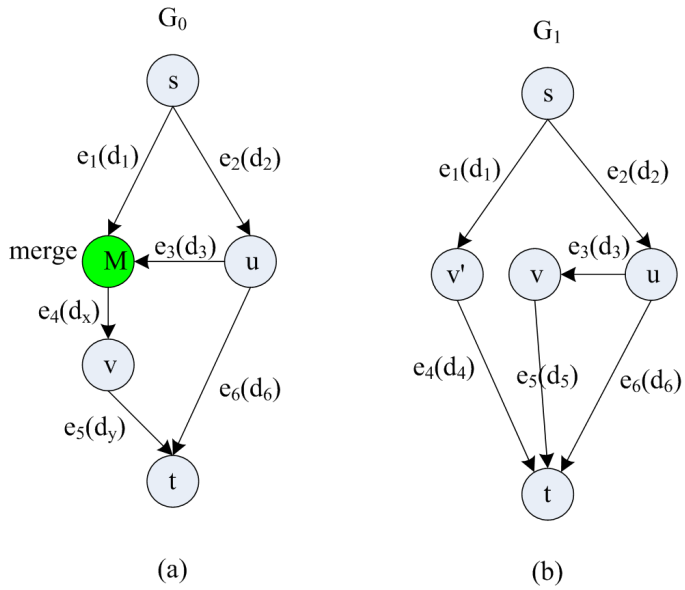


Figure 3.4: Example of a run which contains a merge processor (a) initial run ($d_x = [d_1, d_3], d_y = [d_4, d_5]$); (b) an equivalent run of (a).

obtain:

$$OutProv(G_0) = (d_y \cdot \widetilde{v} \cdot d_x \cdot \widetilde{M} \cdot (d_1 + d_3 \cdot \widetilde{u} \cdot d_2) + d_6 \cdot \widetilde{u} \cdot d_2) \cdot \widetilde{s}$$

$$OutProv(G_1) = (d_4 \cdot \widetilde{v} \cdot d_1 + (d_5 \cdot \widetilde{v} \cdot d_3 + d_6) \cdot \widetilde{u} \cdot d_2) \cdot \widetilde{s}$$

It is obvious that $OutProv(G_1) \neq OutProv(G_0)$.

Figure 3.5 shows the data dependency graphs for the runs in Figure 3.4. In Figure 3.5 (a), we can obtain that $d_y = [d_4, d_5]$ depends on d_1 and d_3. It will raise a risk of misleading a meaning that d_4 depends on d_1 and d_3 or d_5 depends on d_1 and d_3 too, which is not correct since v produced d_4 and d_5 in turn when taking d_1 and d_3 as inputs. However, in Figure 3.5 (b), all the data dependencies are unambiguous.

Indeed, the two runs in Figure 3.4 are equivalent, because they produced the same intermediate and final data. It implies that the output provenance expressions of the two runs should be equivalent. Figure 3.4 also indicates that a run like (a) can be transformed into (b), so that it will have an unambiguous provenance meaning (fine-grained provenance) following the representation of regular expression. How to extend our model to obtain equivalent output provenance expressions from G_0 and G_1 in Figure 3.4 currently remains an open question.

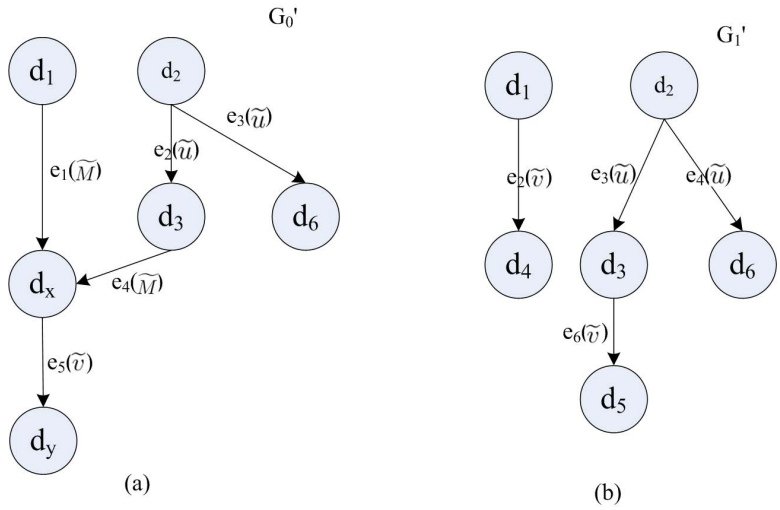


Figure 3.5: Data dependency graphs of runs in Figure 3.4

Another kind of "problematic" task is the split processor which splits a collection into single data items. As shown in Figure 3.6 (a), processor L is a split processor which splits the collection $d_x = [d_2, d_3]$ into single data items d_2 and d_3. The data dependency graphs are shown in Figure 3.7. The same as merge processors, we can obtain:

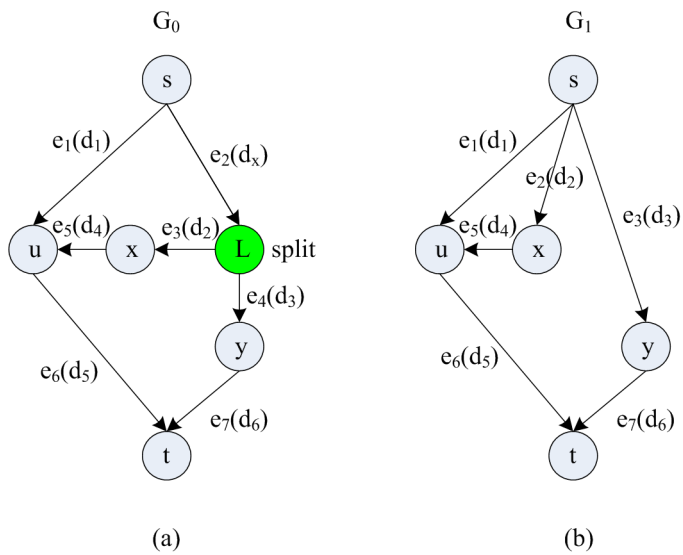


Figure 3.6: Example of a run which contains a split processor (a) initial run ($d_x = [d_2, d_3]$); (b) an equivalent run of (a).

$$OutProv(G_0) = (d_5 \cdot \widetilde{u} \cdot (d_1 + d_4 \cdot \widetilde{x} \cdot d_2 \cdot \widetilde{L} \cdot d_x) + d_6 \cdot \widetilde{y} \cdot d_3 \cdot \widetilde{L} \cdot d_x$$

As shown in the data dependency graph Figure 3.7 (a), d_6 depends on d_3 which

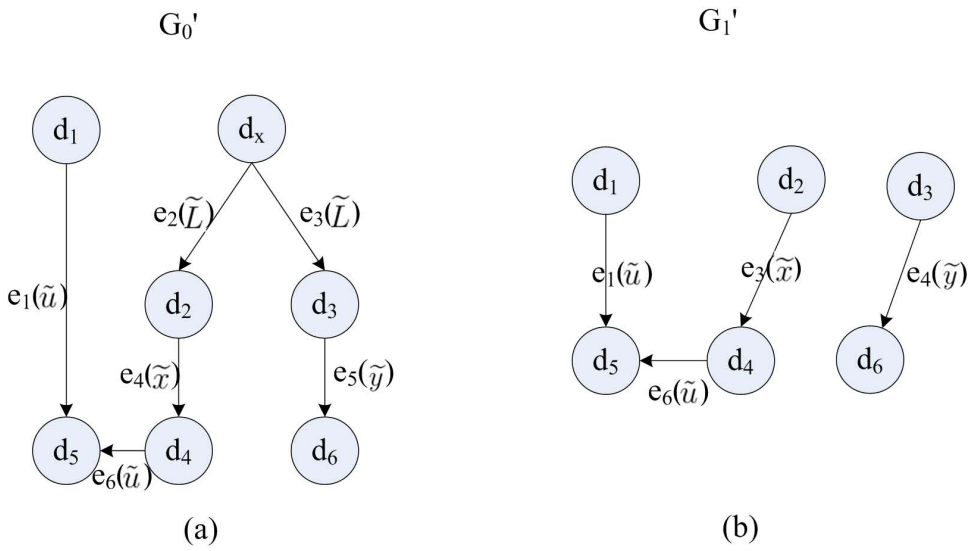


Figure 3.7: Data dependency graphs of graphs in Figure 3.6

depends on $d_x = [d_2, d_3]$, thus d_6 depends on $d_x = [d_2, d_3]$. However, d_6 is derived from d_3. So, a split processor also leads to a misleading of provenance meaning. Figure 3.6(b) is the equivalent run of (a), in which the immediate and final data are the same because a split processor does not do any change to the data.

Furthermore, the combination of merge processors and split processors may occur frequently in Taverna workflows, as shown in figure 3.8. In (a), it is clear that the merge and split processors executed once while processor v executed twice.

The misleading meaning happens because many workflow systems support processors that make small changes or no change to data values but reorganize the data structures (e.g., generate a collection or split a collection). These processors themselves (e.g., a filtering processor, a merge processor, a split processor, etc.) lead to "problematic" data dependencies for most current provenance models, and this situation cannot be avoided in most workflow systems. So, how to address this problem remains an open question.

3.4.3 Possible solutions

Recently, Missier *et al.* [MPB10] have proposed a fine-grained provenance model for Taverna system which considers all the *in-relations* and *out-relations* of each data value inside a collection of data values.

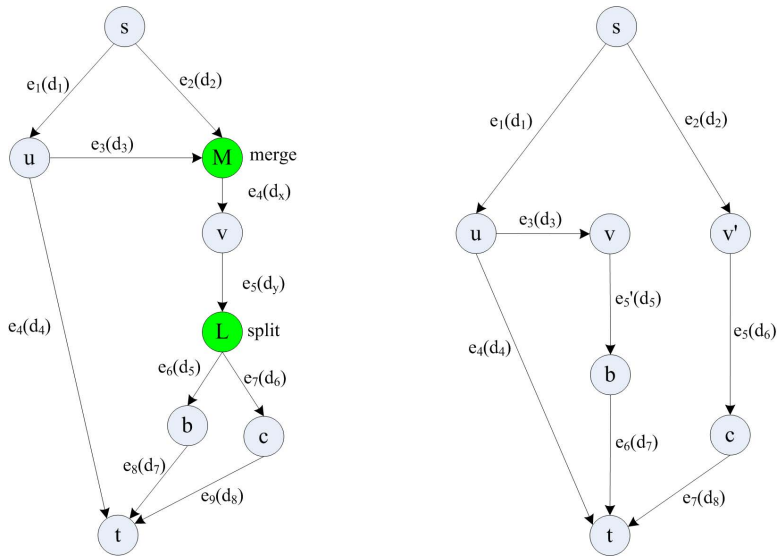


Figure 3.8: Combination of a merge processor and a split processor. (a) initial graph G_r ($d_x = [d_3, d_2], d_y = [d_5, d_6]$); (b) an equivalent graph of G_r

Inspired by Missier's model, we now propose one research hint for allowing our approach to support the "problematic" data dependencies discussed in the previous subsection.

We could extend our provenance model by considering the fine-grained provenance model proposed by Missier *et al.* to directly defined new regular expressions for data values inside a collection.

The key problem of this solution is how to define the regular expression used in our provenance model to support fine-grained provenance. Indeed, each data value in a collection is deterministic according to the processor, so that we can define immediate provenance and deep provenance for each data value. In such a way, it will be possible for our provenance model to support all these "problematic" data dependencies in most workflow systems.

As some workflow systems include loop [BML08] or fork executions, the use of the data dependency graph will also be considered as a possible direction to extend our model to deal with a restricted form of loops in the specifications making runs having fork-loop structures.

3.5 Summary

In this chapter, we discussed the capacity of the "basic provenance model" [ABML09] to accurately capture data dependencies in many computational scenarios and we introduced our provenance model which makes use of regular expressions to organize all the in-relations and out-relations to capture the provenance trace of a run. Our model is suitable for identifying the meaning of a workflow and to compare the provenance structures of two workflows. Based on the underlying model, we gave the notion of *provenance-equivalence*, which is the property used to evaluate whether two workflows will always execute the same way. In the end of this chapter, we discussed several "problematic" data dependencies which are caused by some special processors (such as a merge processor or a split processor). Finally, two research hints for extending our model are provided.

Based on the works introduced in this chapter, two main works will be introduced. Chapter 4 introduces approaches to transform a non-SP workflow structure into an SP workflow structure while preserving provenance. Chapter 5 discusses a new approach of rewriting scientific workflows by removing some anti-patterns without alerting the semantics of the workflows.

Chapter 4

Rewriting scientific workflows while preserving provenance

Contents

4.1 Motivating Scenarios **41**

4.1.1 Designing workflows 42

4.1.2 Querying workflows 43

4.1.3 Scheduling workflows 44

4.2 Distance from non-SP to SP graphs **44**

4.2.1 Vertex reduction 45

4.2.2 Vertex duplication 47

4.2.3 Complexity measures 49

4.3 Graph rewriting problems (non-SP to SP) **50**

4.3.1 Review of existing approaches 50

4.3.2 Compositions of forbidden graphs 53

4.4 SPFlow: a new provenance-equivalent rewriting algorithm for workflows **57**

4.4.1 Principle of SPFlow 57

4.4.1.1 Duplicated subgraph 58

4.4.1.2 Reduction sequence 59

4.4.1.3 Reducing redundancy of duplicated vertices 62

4.4.1.4 Algorithm description 63

4.4.2 Example of use of SPFlow 65

4.4.3 Complexity 65

4.4.4 Soundness of vertex duplication algorithm 68

4.5 Experimental study 69

4.5.1 Workflow Structures 69

4.5.2 Evaluating SPFlow 70

4.6 Implementation of the algorithm 71

4.6.1 SPFlow architecture 71

4.6.2 Functionalities of SPFlow 72

4.7 Discussion 75

4.8 Summary 77

Chapter 2 has introduced the main definitions related to structures of scientific workflows. In particular, we have introduced the notion of SP-graphs which structure is well-known to have good properties (complex graph operations become less complex when SP-graphs are considered). Chapter 3 has introduced a provenance model for scientific workflows and have proposed the definition of provenance-equivalent executions. The aim of this present chapter is to provide an approach for transforming any DAG workflow to an SP-structured workflow while ensuring that the transformation is provenance-equivalent.

Although strategies for rewriting non-SP graphs into SP graphs have been studied in literature, two important questions arise:

1. Do they preserve provenance?

2. Is it possible to design automatic transformation techniques to rewrite non-SP structures to SP structures while preserving provenance?

This chapter is organized as follows. We first introduce several scenarios to give an in-depth explanation of our motivation for this work in section 4.1. After that, section 4.2 gives the concept of measuring the distance from non-SP to SP, which inspires some transformation techniques of rewriting non-SP graphs into SP graphs. Then, in section 4.3, we analyze the graph rewriting approaches of the literature by identifying whether they are provenance-preserving. In section 4.4, a detailed description of our full algorithm is carrying out, together with the

discussion of complexity and soundness of the algorithm. Then we demonstrate the feasibility of our approach on real scientific workflows in section 4.5. We present a tool which takes in a non-SP Taverna workflow and provides an SP workflow in section 4.6. Finally, we summarize our work and a discussion of ongoing work is given.

4.1 Motivating Scenarios

Interestingly, most of the business workflow structures are captured by SP structures [MGLRtH11]. Several approaches [BBDH08, GEvGCP09] have shown that using SP workflows allows to design more user-friendly workflows and provide more efficient execution settings. Others [BBD+09, CFS+06], in particular in the domain of provenance information management, have even chosen to restrict workflow graphs to SP structures. And in the domain of workflow scheduling, many approaches [ZCHW11, MKK+05, BRGRM11] restrict workflow graphs to SP structures to solve some scheduling or mapping problems which can not work on DAGs in polynomial time, such as mapping workflows onto chip multiprocessors [BRGRM11]. Furthermore, in [QF07], Qin and Fahringer discussed several scientific grid workflow applications, which are all structured as SP graphs: the WIEN2k workflow performs electronic structure calculations of solids using density functional theory [Bla], and the MeteoAG workflow is a meteorology simulation application [SQN+06], and the GRASIL workflow calculates the spectral energy distribution of galaxies [SGB+01]; this latter application has actually a fork-join graph. A last example is the fMRI workflow [ZWF+04], which is a cognitive neuroscience application.

Motivated by the facts above, we would like to provide workflows with series parallel structures for achieving more efficient solutions for workflow operations on graph structure (e.g., search for (sub)graphs, comparing graphs). This subsection provides scenarios to illustrate in more details the benefits of considering SP structures for scientific workflows.

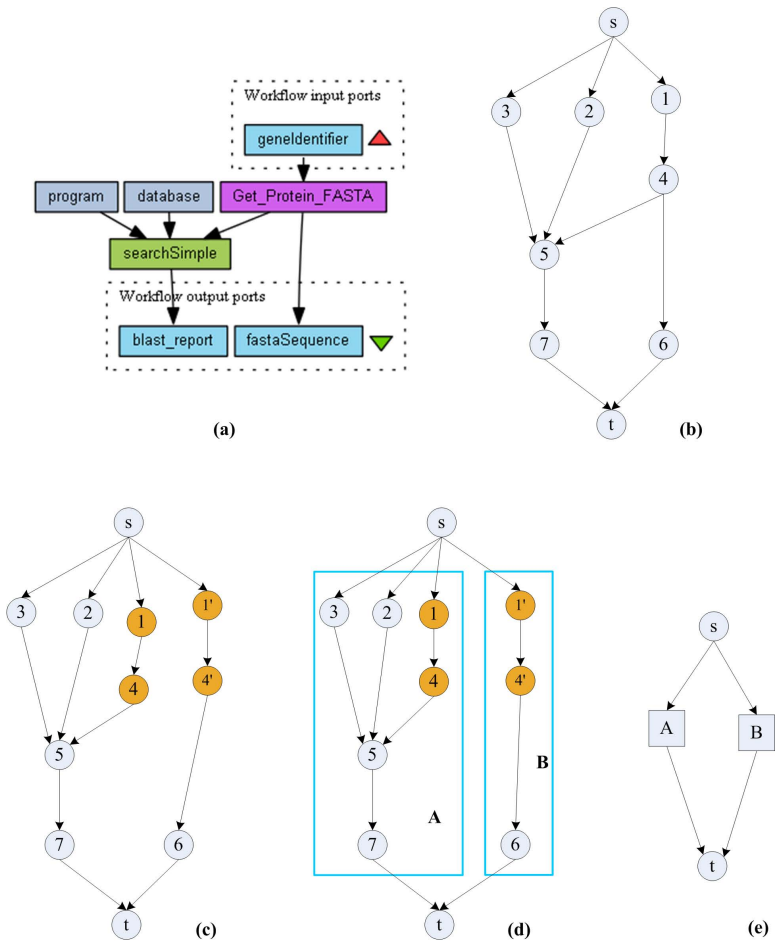


Figure 4.1: (a) Example of simple workflow from Taverna, (b) graph structure of the workflow (non SP); (c) possible SP graph structure; (d) proposal of composite vertices; (e) high-level graph obtained

4.1.1 Designing workflows

As already motivated in the introduction of this manuscript, although scientific workflows have been introduced to help sharing and reusing in-silico experiments, a recent study [SCBL12] showed that authors easily reuse their own workflows but use more rarely workflows of a third party. One explanation is that the graph structure of a scientific workflow can be particularly complex, making the main steps of the analysis difficult to capture. Guiding developers to build workflows that are simple to understand is fundamentally important to improve workflows sharing and reuse. We believe that SP structures should be of great help in this context. Intuitively, and from a purely visual standpoint, SP structures are simple; SP graphs are layered, their edges do not intersect, making the main

phases of the workflow easier to distinguish. Consider the workflow in Figure 4.1 (a) whose structure shown in Figure 4.1 (b) is not SP. It is relatively complex to visually distinguish the main stages of the workflow and build independent subworkflows. Figure 4.1 (d) shows the same workflow in which task 1 has been duplicated into 1 and $1'$ and task 4 has been duplicated into 4 and $4'$ (the data catalogue is thus queried twice) making it being SP. First, restructured this way, the workflow is easier to understand. In particular, designing sub-workflows (A,B) can be performed more naturally. Second, the high level view of the workflow (where subworkflows are black boxes as in Figure 4.1 (e)) is simple and modular while the same type of construction on the original (non SP) workflow would be more complex due to the cross edge $e(4, 5)$ which will lead to an edge imposed between the sub-workflows, making the sharing and reuse of the sub-workflows less easy. Note that the original structure of the workflow was not far from an SP structure. The benefits of exploiting the SP structures increases with the complexity of the workflow structures.

There are approaches dedicated to the design of sub-workflows within workflows. This is the case of ZOOM [BBDH08] that takes in information about the tasks of interest to the user and builds automatically a user view providing a concise representation of the workflow with sub-workflows focused on the tasks of interest. The benefit of considering SP structures has been shown in this context too: [BDKR09] proved that computing the smallest user view (*i.e.* minimum number of composite tasks) cannot be systematically reached for arbitrary DAGs whereas it is the case when SP structures are considered.

4.1.2 Querying workflows

Another way to design workflows is to build on existing workflows. The user can query a workflow warehouse to find workflows having a particular structure or containing a given pattern. The need for the user to be able to do this type of research in warehouses has been expressed for several years [GFG+09, CBL11, GGB11] but is still not considered in the workflows warehouses today, as this type of research is directly associated with problems known to be NP-hard (subgraph isomorphism) on conventional DAGs. SP structures have again a clear advantage: finding a subgraph isomorphic to a given graph can be treated in polynomial

time on such structures [LS88]. Another example of the type of queries involving operations on graphs is the search for differences between workflow structures [CFS+06, BBD+09]. Again the problem of calculating the difference of two subgraphs of the same graph is NP-hard in the general case and polynomial for SP-graphs [BBD+09]. The operation of querying structures or comparing structures in scientific workflows may can benefit from SP structures.

4.1.3 Scheduling workflows

Orthogonally, SP structures can also be particularly interesting in the context of scheduling runs. In the broader field of scheduling tasks in programs, SP structures have been exploited for decades [GEvGCP09], particularly because they have demonstrated their benefits for program analysis [LW98], cost estimation [vG97], and effectiveness of planning [FLMB96]. Many current approaches [ZCHW11, MKK+05, BRGRM11] also show that more efficient solutions can be carried out if scientific workflows have SP structures. With the development of grid and cloud computing, running workflows on multiple, distributed resources is of growing importance. As a combination of series and parallel components, SP-workflows fit particularly well with MapReduce environments.

4.2 Distance from non-SP to SP graphs

Recall that our aim in this chapter is to propose approaches able to rewrite any non-SP graph into an SP graph. Knowing the distance from non-SP to SP graphs gives a better understanding of the basic concept of designing automatic transformation techniques. In this section, we present formal methods, adapted from [BKS92], to define and measure the distance from a non-SP graph to an SP graph. Such definitions are at the basis of the transformation techniques detailed in the next section.

The distance from non-SP to an SP graph can be measured by the number of induced forbidden subgraphs that the non-SP graph has. This distance has shown to be a very important parameter of a graph. Many graph analysis problems are NP-hard, and hence there is probably no polynomial-time algorithm for any of them. But it has been shown that there exist feasible solutions for many of them if

the graph is restricted to an SP graph [Nau95,BKS92]. Nevertheless, it is possible to derive algorithms that are exponential in the distance from the graph to an SP form, rather than in its size (the number of nodes) [BKS92]. Two complexity measure methods have already been proposed to measure the number of forbidden subgraphs in a graph, which are *reductions* and *path expressions* [Nau95, BKS92]. Because the path expression complexity and reduction complexity are related, we introduce in this subsection the operations related to the reductions.

4.2.1 Vertex reduction

Any st-multidag G can be reduced to one single edge by means of three kinds of reductions, the series reduction, parallel reduction, and the vertex reduction [BKS92]. It has already been introduced in Chapter 2 that series and parallel reductions can be used to eliminate all the SP components (series components and parallel components) of the graph. After applying such transformations, only vertices and edges associated with forbidden subgraphs remain. Then one can use the operator of vertex reduction to eliminate the vertices which induced the forbidden subgraphs. The vertex reductions can be divided into two classes, ***out-vertex reduction*** and ***in-vertex reduction***. In the first situation, the vertex has only one input link and a collection of output links. In the second situation, it substitutes a vertex with only one output link and a collection of input links. The effect of vertex reduction in both cases, is shown in Figure 4.2.

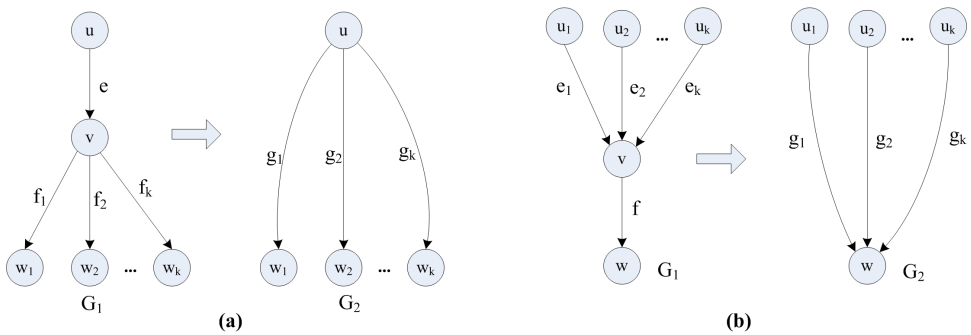


Figure 4.2: (a) Out-vertex reduction; (b) In-vertex reduction

Definition 4.2.1 Let $G_1 = (V_1, E_1)$ be an st-multidag whose vertices and edges are labelled, by the functions $L_{1vr} : V_1 \rightarrow L_{VR}$, and $L_{1er} : E_1 \rightarrow L_{ER}$. The **vertex reduction** operators are defined as follow [CBFJ12]:

(1) Let $v \in V_1$ having a unique incoming edge $e = (u, v)$ and k outgoing edges $f_1 = (v, w_1), \cdots, f_k = (v, w_k)$. The operation of **out-vertex reduction** in v replaces v and its edges $\{e, f_1, ..., f_k\}$ by k new edges $\{g_1, ..., g_k\}$ where $g_i = (u, w_i)$, for $i \in [1, k]$. $G_2 = (V_2, E_2)$ is such that $V_2 \subset V_1, L_{2vr}$ is the restriction of L_{1vr} on $V_2, L_{2er} = L_{1er}$ on $E_1 \cap E_2$, and $L_{2er}(g_i) = L_{1er}(f_i) \cdot L_{1vr}(v) \cdot L_{1er}(e)$. (cf. Figure 4.2 (a)).

(2) The operation op of **in-vertex reduction** can be defined analogously, considering node v with $d^{+1}(v) = 1$ and $d^{-1}(v) > 1$ (cf. Figure 4.2 (b)).

After applying all possible series-parallel reductions on all the vertices, any vertex can be chosen to be reduced under the vertex reduction operation except the source and the target of the graph. In the minimal forbidden subgraph, at least one child (in-degree is one) of the source can be reduced by out-vertex reduction and one child (out-degree is one) of the source can be reduced by in-vertex reduction. Thus, there are always vertices that can be reduced by vertex reduction. After applying a vertex reduction, it is possible again to apply new series-parallel reductions which should always be applied before any new vertex reduction.

The edges in an execution G_r indicate the data dependencies and the labels of an edge represent data production. Recall that our aim is to transform graphs while preserving provenance information. During each transformation we need to keep track of the vertices and edges removed. Following in the definition of out-vertex reduction, the label of a new edge is replaced by the data flow information consists of the eliminated vertex and its edges. In that way, after the graph being reduced to one single edge, the label of the edge remained saves all the data flow information of G_r which related to the expression of the *output provenance* of the execution graph.

Example 4.2.1 Consider Figure 4.3. Initially, each edge has a single label, and after applying a vertex reduction operation on u, edges e_1, e_4 are replaced by g_2 and e_1, e_3 are replaced by g_1. The label of g_1 is replaced by the data flow information which consists of the labels of e_3, e_1 and u, which is $d_3 \cdot u \cdot d_1$. The label of g_2 is replaced by the labels of e_4, e_1 and u, which is $d_4 \cdot u \cdot d_1$. The same way for series and parallel reductions according to their definitions. Finally, as in G_4, the label

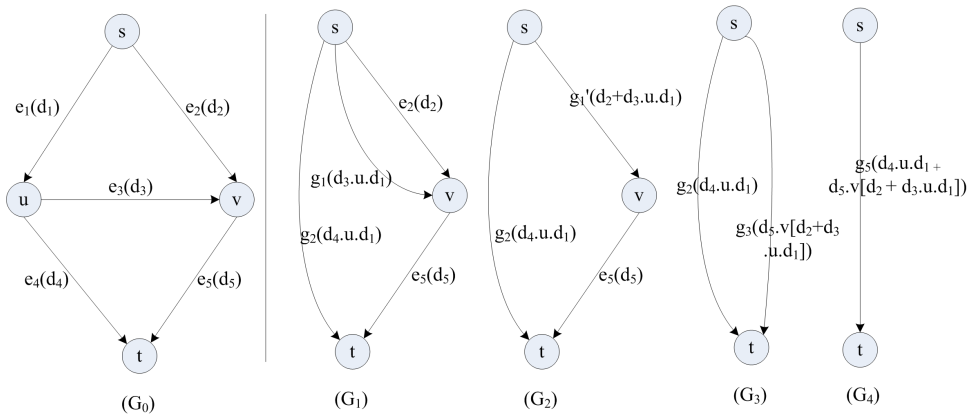


Figure 4.3: Example of reduction operations applied to G_0.

of g_5 is $d_4 \cdot u \cdot d_1 + d_5 \cdot v \cdot (d_2 + d_3 \cdot u \cdot d_1)$. This label is an expression which is equal to $OutProv(G_0)\backslash s$.

Property 4.2.1 The following reduction operations are provenance-preserving. More precisely, considering again definition 2.21 and definition 4.1:

(1) Series reduction: $Hist(w)_{G_1} = Hist(w)_{G_2}$;

(2) Parallel reduction: $Hist(w)_{G_1} = Hist(w)_{G_2}$;

(3) Out-Vertex reduction: $Hist(w_i)_{G_1} = Hist(w_i)_{G_2}$ for all $i \in [1, k]$.

This property comes from the fact that we store in the label of the remaining edges the data flow (in reverse order) the labels of the edges and vertices that have been reduced.

Property 4.2.2 In-vertex reduction is not provenance-preserving. A concrete counter-example will be provided in 4.3.1.

In the following, we will thus only consider out-vertex reduction.

4.2.2 Vertex duplication

The vertex reduction introduced above is directly related to an non-SP to SP transformation, which will be detailed discussed in next section. We propose to introduce new operations on vertices, namely *vertex duplication*. As shown in Figure 4.4, the vertex duplication creates multiple occurrences of the vertex.

If series reductions are applied to the occurrences, the result is thus equal to the graph obtained by vertex reduction. As the duplication only copies the vertices and edges, no additional data dependence will be added to the graph, and once a vertex duplication operation has been applied to a reduction vertex, the "problematic" subgraph disappears.

Formally, the definition of vertex duplications are given as follow [CBFJ12]:

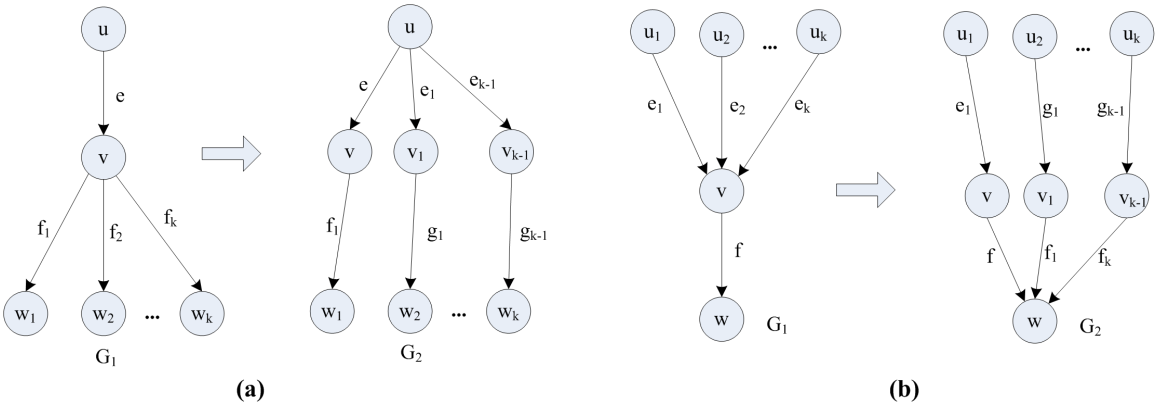


Figure 4.4: (a) Out-vertex duplication; (b) In-vertex duplication

Definition 4.2.2 Let $G_1 = (V_1, E_1)$ be an st-multidag with label functions L_{1vr} and L_{1er}. Let $v \in V_1$ having a single incoming edge $e = (u, v)$ and k outgoing edges $f_1 = (v, w_1), \cdots, f_k = (v, w_k)$.

(1) The **out-vertex duplication** of v transforms G_1 into $G_2 = (V_2, E_2)$, whose vertices and edges are labeled by $L_{2vr} : V_2 \to L_{VR}$ and $L_{2er} : E_2 \to (L_{VR} \cup L_{ER}, +, \cdot)$, such that V_2 is the union of V_1 and the set of new vertices $v_1, ..., v_{k-1}$, which are copies of vertex v. L_{2vr} is an extension of L_{1vr} on V_2, which matches with L_{1vr} on $V_1 \cap V_2$, $L_{2vr}(v_i) = L_{1vr}(v)$ for all $i \in [1, k-1]$. $E_2 = E_1 \cup \{e_1, ..., e_{k-1}\}$, with $e_i = (u, v_i)$ for all $i \in [1, k-1]$, and replacing edges $\{f_2, ..., f_k\}$ by new edges $\{g_2, ..., g_k\}$ with $g_i = (v_{i-1}, w_i)$ for $i \in [2, k]$. $L_{2er} = L_{1er}$ on $E_1 \cap E_2$, $L_{2er}(e_{i-1}) = L_{1er}(e)$, and $L_{2er}(g_i) = L_{1er}(f_i)$ for $i \in [2, k]$. (cf. Figure 4.4 (a))

(2) The operation of **in-vertex duplication** can be defined analogously, considering node v with $d^{+1}(v) = 1$ and $d^{-1}(v) > 1$. So that, V_2 is the union of V_1 and the set of new vertices $v_1, ..., v_{k-1}$, which are copies of vertex v. L_{2vr} is an extension of L_{1vr} on V_2, which matches with L_{1vr} on $V_1 \cap V_2$,

$L_{2vr}(v_i) = L_{1vr}(v)$ for all $i \in [1, k-1]$. $E_2 = E_1 \cup \{f_1, ..., f_{k-1}\}$, with $f_i = (v_i, w)$ for all $i \in [1, k-1]$, and replacing edges $\{e_2, ..., e_k\}$ by new edges $\{g_2, ..., g_k\}$ with $g_i = (u_{i-1}, v_i)$ for $i \in [2, k]$. $L_{2er} = L_{1er}$ on $E_1 \cap E_2$, $L_{2er}(f_{i-1}) = L_{1er}(f)$, and $L_{2er}(g_i) = L_{1er}(e_{i-1})$ for $i \in [2, k]$.(cf. Figure 4.4 (b)).

Property 4.2.3 The operation of **out-vertex duplication preserves provenance**.
More precisely, with the notations above, we have:
$Hist_{G_1}(w_i) = Hist_{G_2}(w_i)$ for all $i \in [1, k]$ (cf. Figure 4.4 (a)).

Property 4.2.4 The operation of **in-vertex duplication does not preserve provenance**.
A concrete counter-example will be provided in 4.3.1.

4.2.3 Complexity measures

As said in section 4.2.1, the number of times the vertex reduction operations are used can give an idea of the distance from a non-SP to an SP structure. Intuitively, the highest number of times vertex reductions are used the farthest from an SP structure it is.

Definition 4.2.3 **The reduction complexity** of a graph G, denoted by $\mu(G)$, is the minimal number of vertex reductions sufficient(along with series and parallel reductions) to reduce G to a BSP graph. (This definition comes from [BKS92])

Definition 4.2.4 The sequence of $\mu(G)$ vertices $(v_1, v_2, \cdots, v_c)$ that reduce the graph G to a BSP graph is called **reduction sequence**.

As was shown by Bein, Kamburowsky and Stallman in [BKS92], it is possible to compute $\mu(G)$ and reduction sequence in polynomial time complexity. As a result, the maximum distance of a graph to an SP form is limited by the number of vertices:

$$\mu(G) \leq n - 3$$

4.3 Graph rewriting problems (non-SP to SP)

In this section, we investigate the basis of full transformation methods to rewrite an non-SP graph into an SP graph.

4.3.1 Review of existing approaches

We are interested in methods of transforming non-SP graphs to SP forms that keep the provenance information of the original graph. Additionally, we will have a special interest in reducing the number of duplicated vertices. Several transformation techniques have been found in literature, which are all based on *adding dependencies*, while another strategy based on *vertex duplication* is possible too. These two different approaches are detailed as follow.

1. Adding dependencies

The first set of strategies found in the literature to rewrite non-SP graphs into SP graphs are based on the concept of adding dependencies. The approach of Escribano [Esc03] is part of such approaches and based on the notion of (re)synchronization. Informally, the idea is "to layer" the graph by adding artificial vertices (and edges), which act as synchronization tasks. Three main synchronization strategies (see Figure 4.5) are possible. From the forbidden graph in (a), the operations of up-, down- and across-synchronization provide respectively graphs (b), (c) and (d). In (b), the edges $e(u, v)$ and $e(v, t)$ are added to forward data value d_4. In (c), edges $e(s, u)$ and $e(u, v)$ are added. And in (d), one zero loaded vertex w is added, which forwards data values d_1, d_2, d_4 to the right destination.

Does it provide an SP graph? Yes. Consider the up-synchronization of Figure 4.5(b). Two parallel reductions remove the double edges between u and v and between v and y. Then one series reduction removing vertex u, followed by one parallel reduction between x and v, and a series reduction on v finally provide the BSP graph. The same for (c) and (d), when applying series and parallel reductions on these graphs, finally, we will obviously obtain two BSP. As a result, (b),(c) and (d) all can provide an SP graph.

Does it preserve provenance? No. If the provenance is preserved, then no data dependency should be added or lost. However, in case (b), d_5 depends on d_4

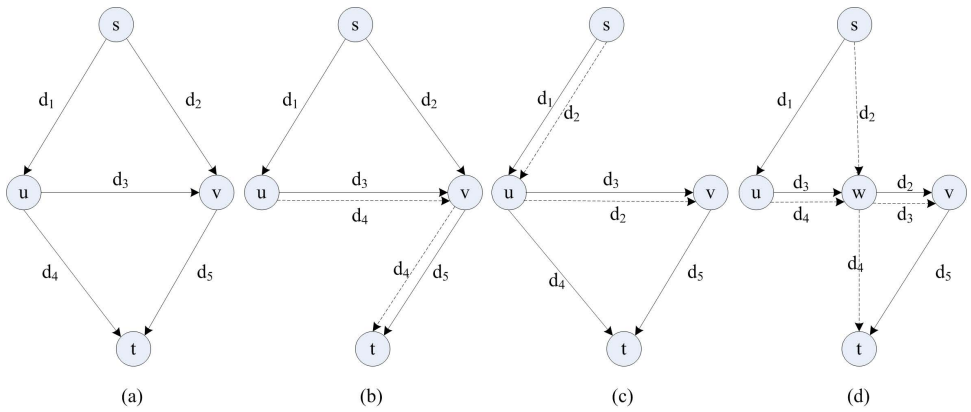


Figure 4.5: Resynchronization. (a) forbidden subgraph; (b) up-synchronization; (c) down-synchronization; (d) across-synchronization.

which is never the case in (a) (d_5 depends on d_2 and d_3, not d_4); In (c), d_3 and d_4 all depend on d_2 which is never the case in (a). In case (d), the outputs of the added vertex w fully depend on all the incoming data, so that d_5 depends on all the incoming data of w, which is not the case in (a).

Note that in the strategy of across-synchronization, we can add a new data forward vertex which only forward data values to the right processors. As discussed in chapter 3, this kind of processor will lead to an unclear data dependency, which will lead to a different provenance meaning. And this pattern is currently unsupported by our provenance model. So we don't take this approach into account.

2. Duplication of vertex

Another family of approaches to transform non-SP to SP graphs is based on vertex duplication. The main interest of this kind of transformation is that it does not add any additional dependency to any task of the original graph, because it only copies the task and its original dependency. However, our aim of preserving provenance raises the following questions which is related to transforming an non-SP graph into an SP graph.

1) Do duplication operations provide an SP graph?

Yes. In Figure 4.6 (b), two series reductions remove the vertices v and v', then one parallel removes the double edges between u and t. Thus, one parallel can be applied to the edges between s and t, followed by two series reductions on u and v. Finally, it is a BSP. The same for (c), when applying series and parallel on

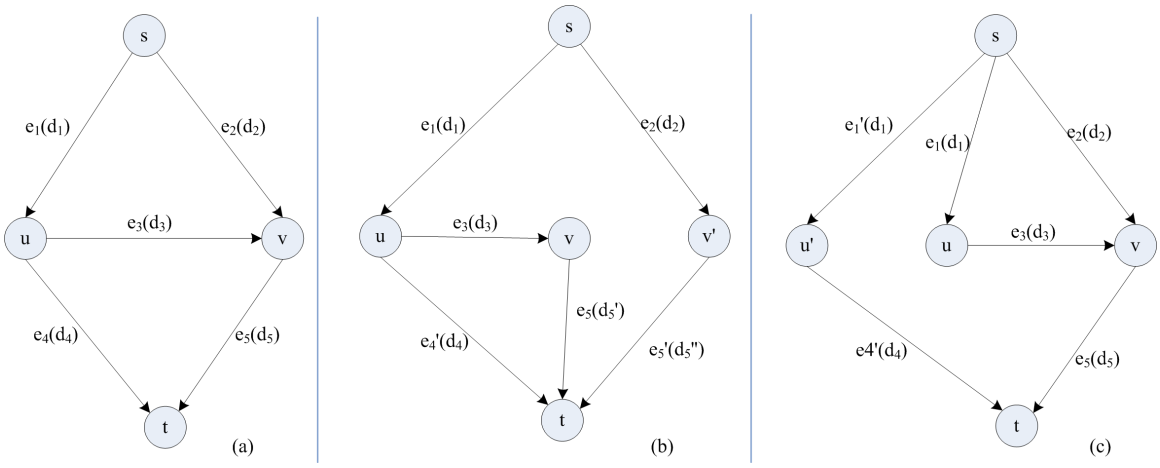


Figure 4.6: From the (a) Forbidden graph, use of (b) in-Vertex Duplication and (c) out-Vertex Duplication.

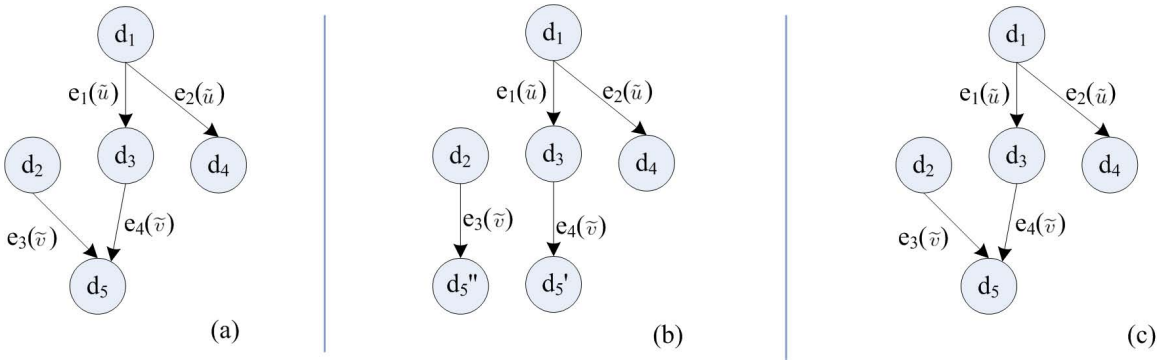


Figure 4.7: Data dependency graphs for runs in Figure 4.6.

the graph, finally we obtain a BSP. As a result, we obtain two BSP. So, the two rewritten subgraphs are SP.

2) Does in-vertex duplication preserve provenance?

No (c.f. property 4.4). Consider Figure 4.6 (b). The deep provenance of e_5 has changed in (b): it does not involve data d_2 any more. The major problem is that one input of task v have been removed so that the task cannot deliver results. Its data dependency graph in Figure 4.7 (b) shows that the data value of d_5 disappeared, but two new data d_5' and d_5'' are produced. It is obviously graph (a) and (b) in Figure 4.6 are not provenance-equivalent.

3) Does out-vertex duplication preserve provenance?

Yes (c.f. property 4.3). As shown in Figure 4.6 (c), the task u is duplicated into u' and each copy receives the same input (u is not modified). As the tasks

are deterministic, they thus provide the same output. Also, it is clear in Figure 4.7 (a) and (c) have the same structure (same vertices and same edges). So, they are provenance-equivalent.

We have now provenance-preserving operations of reductions able to locally provide SP structures. The rest of this subsection aims at providing a general provenance-preserving approach to transform a non-SP to SP structure while minimizing the number of duplicated vertices. The next subsection will thus introduce notions useful to choose the order of reduction operations to perform.

4.3.2 Compositions of forbidden graphs

Transforming a non-SP graph G to an SP graph requires eliminating all the forbidden subgraphs in graph G. When a graph contains several forbidden subgraphs, these subgraphs may be composed. In [BKS92] and [Esc03], three composed forbidden subgraphs are studied to decide which reduction vertices must be chosen to get a shorter reduction sequence. The reduction sequences of these compositions are based on both in-vertex and out-vertex reductions.

However, in the present work, we only consider out-vertex duplication which is provenance-preserving (contrary to in-vertex duplication which is not). We thus summarized three compositions of forbidden subgraphs, which are different from the compositions introduced in [BKS92] and [Esc03], according to the reduction vertices to which we can apply out-vertex reductions. The impact of the order of reduction operations chosen to perform for each composition is then discussed.

Definition 4.3.1 If a forbidden subgraph G contains a reduction vertex v, we say that G is **induced** by v.

Several forbidden subgraphs can be induced by one reduction vertex. (cf. Figure 4.8)

There are two situations where a graph contains two forbidden subgraphs G_1 and G_2 : (1) $G_1 \cap G_2 = \emptyset$; (2) $G_1 \cap G_2 \neq \emptyset$. It is obvious that if $G_1 \cap G_2 = \emptyset$, G_1 will never affect G_2, and any order of reduction operations will give the same result. So, we introduce here the second situation by identifying three compositions.

Single non-SP composition: There exist several similar forbidden subgraphs, which are induced by one single reduction vertex.

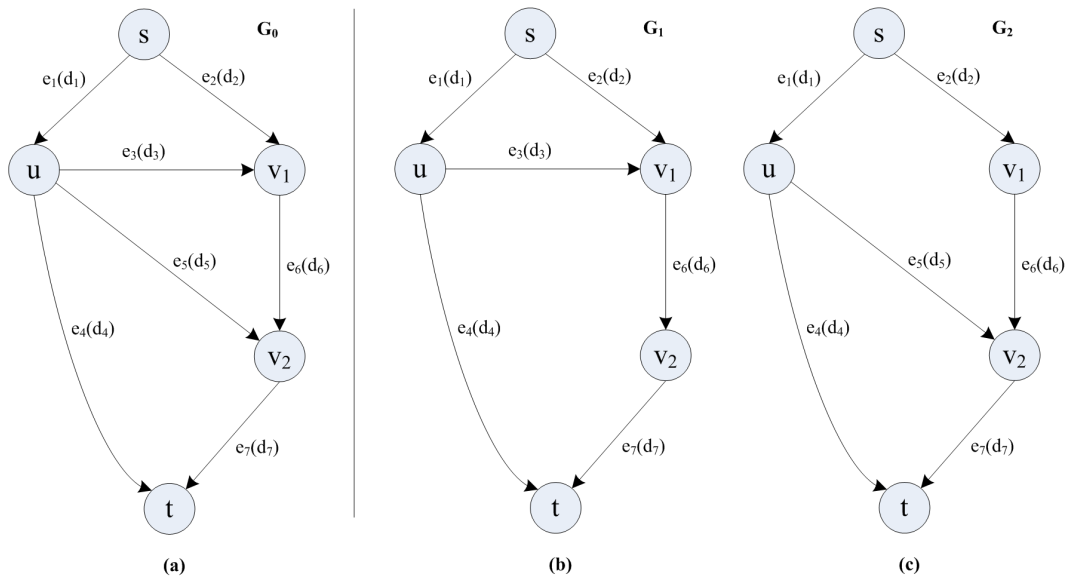


Figure 4.8: Two forbidden subgraphs induced by one reduction vertex. (a) graph with one reduction vertex u; (b) one forbidden subgraph induced by u; (c) another forbidden subgraph induced by u.

In the single composition (cf. Figure 4.8), there is only one reduction vertex which is related to several forbidden subgraphs. The solution for eliminating these kinds of compositions is to duplicate the reduction vertex following one reduction operation. So, the forbidden subgraphs will never affect each other.

Series non-SP composition: There exist several similar forbidden subgraphs, in which the reduction vertices (Out-vertex reduction) form a series composition.

Three kinds of series non-SP compositions are possible.

Let v_1, v_2 be two reduction vertices of graph G, G_1 be a forbidden subgraph induced by v_1 and G_2 be a subgraph induced by v_2. For the sake of readability, labels are omitted.

(1)path $p(v_1, v_2) \subset (G_1 \cap G_2)$ (c.f. Figure 4.9 (a)).

Example of series non-SP composition (1) is shown in Figure 4.9 (a). In this composition, G_1 and G_2 are induced by v_1 and v_2. Although v_1 appears in the path $p(s, v_2)$, the elimination of the two forbidden subgraphs do not affect each other. It means that the duplication operations following any reduction sequence will give the same result.

(2)$G_1 \subset G_2$ (c.f. Figure 4.10 (a)).

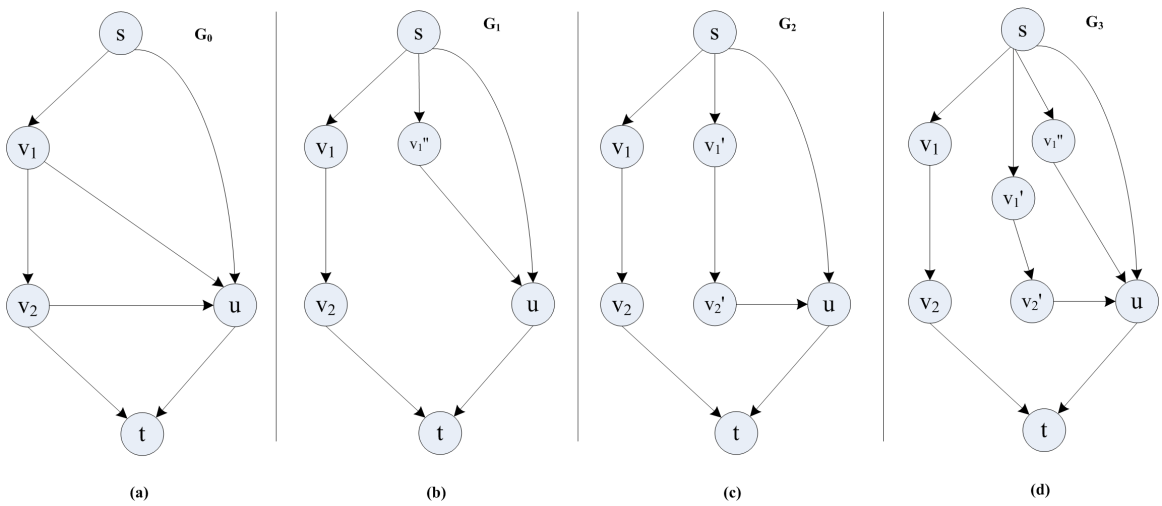


Figure 4.9: Solutions for graphs homeomorphic to Series-non-SP composition (1). (a) a non-SP graph with reduction vertices v_1 and v_2; (b) SP transformation of the forbidden subgraph induced by v_1 in (a); (c) SP transformation of the forbidden subgraph induced by v_2; (d) SP solution for (a).

In series composition (2), as shown in Figure 4.10 (a), G_1 appears in the subgraph which forms all the paths from s to v_2. In such a case, if we duplicate v_2 first as in Figure 4.10 (c), and then duplicate v_1, following the reduction sequence v_2, v_1, the whole forbidden subgraph G_1 may be duplicated, which will make the result unreliable, because it copied a non-SP problem. As a result, in this kind of compositions, the reduction sequence may affect the result, which should be carefully considered.

(3)$v_1 = s(G_2)$ (c.f. Figure 4.11 (a)).

In series composition (3), only one forbidden subgraph can be found. There are two solutions for this kind of compositions, as shown in Figure 4.11 (b) and (c). In (b), vertices v_1 and v_2 both are duplicated following the reduction sequence v_1, v_2. But in (c), only v_2 is duplicated and finally the graph becomes an SP graph. It implies that the reduction sequence also may affect the redundancy of duplicated vertices in the rewritten graphs. It is obvious that (c) has less vertices than (a).

Parallel non-SP composition: There exist several similar forbidden subgraphs, in which the reduction vertices(out-vertex reduction) form a parallel composition.

As shown in Figure 4.12 (a), u_1 and u_2 form a parallel composition and the

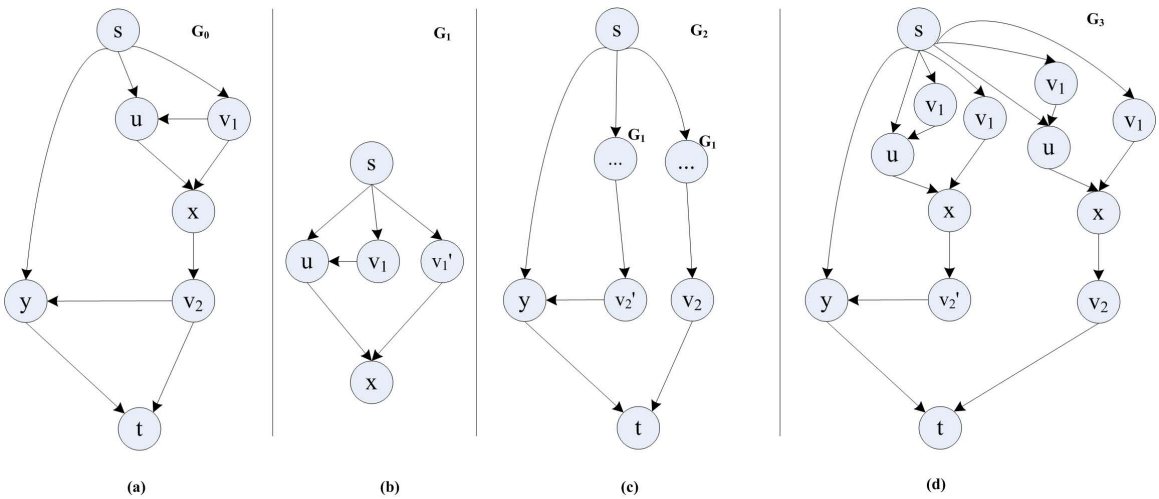


Figure 4.10: Solutions for graphs homeomorphic to Series-non-SP composition (2). (a) a non-SP graph with reduction vertices v_1 and v_2; (b) SP transformation of the forbidden subgraph induced by v_1 in (a); (c) SP transformation of the forbidden subgraph induced by v_2; (d) SP solution for (a). For the sake of readability, we give the same name for the duplicated vertices in (d).

forbidden subgraphs induced by them will never affect each other. Figure 4.12 (b) is the SP graph obtained by duplicating vertex u_1 in the forbidden subgraph induced by u_1 and (c) is the SP graph obtained from the forbidden subgraph induced by u_2. The forbidden subgraphs can be eliminated by duplicating path $p(s, u_1)$ and $p(s, u_2)$. As $p(s, u_1) \cap p(s, u_2) = \emptyset$, so which vertices are duplicated first is not important.

For simple combinations of forbidden subgraphs (single composition or parallel composition), any reduction sequence may be appropriate. But for some series compositions, the reduction sequence may affect the number of duplicated vertices and even create new reduction vertices. Next section will give an in-depth description of our full algorithm, together with the discussion of choosing a shorter reduction sequence, according to which the transformation will duplicate less vertices.

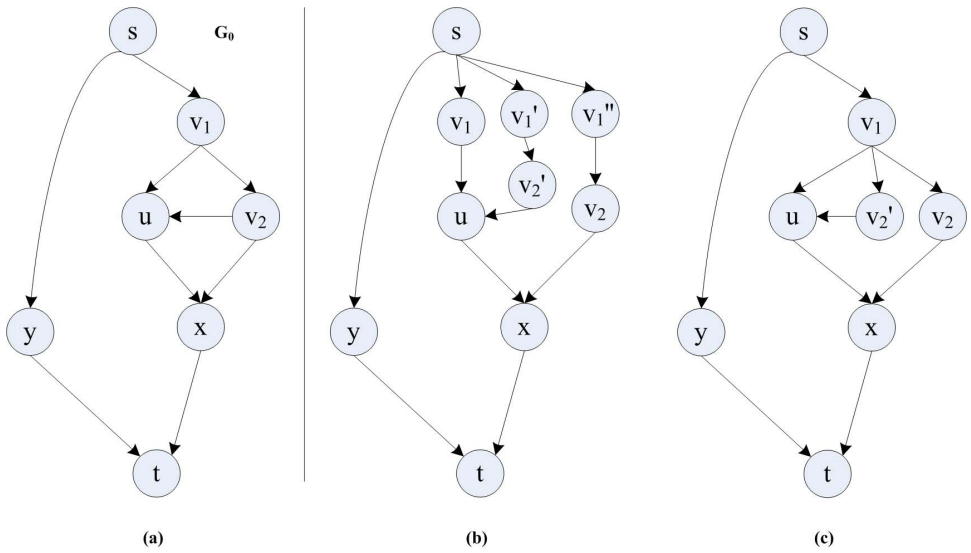


Figure 4.11: Solutions for graphs homeomorphic to Series-non-SP composition (3). (a) a non-SP graph with reduction vertices v_1 and v_2; (b) one SP solution for (a); (d) another SP solution for (a).

4.4 SPFlow: a new provenance-equivalent rewriting algorithm for workflows

This section gives the description of the SPFlow algorithm, a full algorithm based on out-vertex duplication discussed in section 4.3.1, which can be used to rewrite any non-SP workflow to an SP structure while preserving provenance. First, we introduce SPFlow. Then, we illustrate the way SPFlow works with an example. Finally, the complexity and the soundness of SPFlow are discussed.

4.4.1 Principle of SPFlow

We present here our full "SP-ization" algorithm based on vertex duplication, which rewrites a non-SP graph G into a new SP graph, called SPG obtained from G by duplicating vertices of G, while ensuring that G and SPG are provenance-equivalent. As discussed in section 4.2, vertex duplication depends on vertex reduction. Two graphs will be used, one is for vertex reduction which is called G_{red} and the other (SPG) is for vertex duplication, to eliminate the forbidden subgraphs. In our approach, we are interesting in getting a reduction sequence, so that each reduction operation never creates new forbidden subgraphs. For this

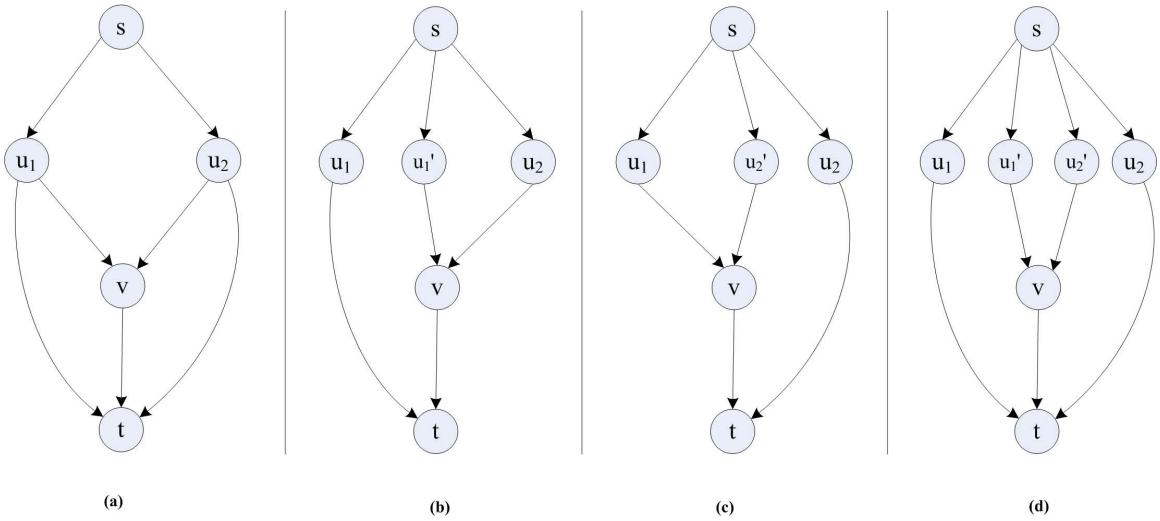


Figure 4.12: Solutions for graphs homeomorphic to Parallel-non-SP composition.

aim, we use a top-down method to eliminate forbidden subgraphs. When all the forbidden subgraphs are eliminated, finally, G_{red} will be BSP while the SPG will be an SP-graph. SPG is the SP graph rewritten from the transformation of the original graph.

This subsection first introduces the notion of duplicated subgraph which is related to a vertex duplication operation. Then the reduction sequence which affects the redundancy of duplicated vertices will be studied. One approach based on the factorization rule discussed in Chapter 2 is then proposed to reduce the redundancy during each parallel reduction. Finally, the algorithm description is given.

4.4.1.1 Duplicated subgraph

The duplication step on SPG is based on the vertex reduction on G_{red}. Each step of vertex reduction operation will trigger a vertex duplication operation. Each edge in G_{red} is related to a subgraph in SPG. The subgraph in SPG induced by the implicit duplicated edge during the vertex reduction is called **duplicated subgraph**. Example of duplicated subgraph is shown in Figure 4.13, in which (a) is a non-SP graph with many series and parallel components, (b) is the maximal reduced graph of (a), and (c) is the duplicated subgraph from (a) which corresponds to the edge $e(s, v)$ in (b).

Property 4.4.1 A duplicated subgraph is an SP graph.

Proof: A duplicated subgraph corresponds to an edge in G_{red}, *i.e.* the duplicated subgraph can be reduced to an edge by applying a maximal reduction operation to it. According to definition 2.2, the duplicated subgraph is obviously an SP graph.

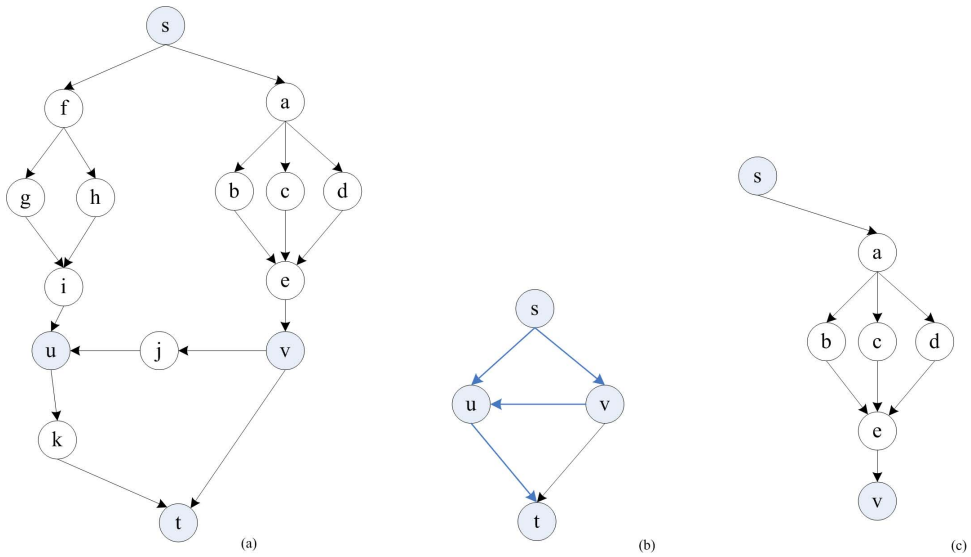


Figure 4.13: Example of duplicated subgraphs. (a)SPG; (b)G_{red} of SPG; (c)duplicated subgraph from SPG induced by edge $e(s, v)$ in G_{red}.

A duplicated subgraph corresponds to one duplication operation. As shown in Figure 4.13, the duplicated subgraph (c.f. (c)) can be obtained by retrieving all the paths in SPG', which is a copy of SPG with all the edges reversed, from the reduction vertex to the source of the edge in G_{red} which ends with the reduction vertex.

4.4.1.2 Reduction sequence

As discussed in section 4.3.2, the reduction sequence will affect the result of the non-SP to SP transformation. This subsection gives a discussion on how to choose a reduction sequence that will lead to a duplicated graph with less redundancy of duplicated vertices.

In Figure 4.14, graph G_0 is a graph in which no series or parallel reduction can be applied. Vertices a,u,c,x are reduction vertices in G_0. Different SP graphs obtained by vertex duplication following different reduction sequences are shown in Figure 4.14 (G_1, G_2). The reduction sequence can be a,u,c,x or u,x etc. G_1 obviously has more duplicated vertices than G_2. When comparing G_1 to G_2, it

is not difficult to find that vertices a and c do not need to be duplicated. To distinguish the class of vertices which have the same situation as vertices a and c in G_0, our algorithm makes use of the notion of **autonomous subgraph** [BKS92].

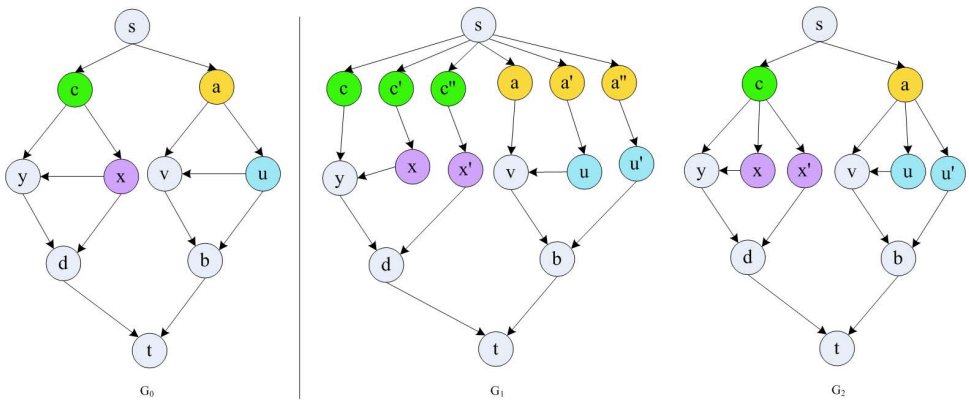


Figure 4.14: Graphs obtained by vertex duplication following different reduction sequences (reduction sequence on G_1: a, u, c, x; reduction sequence on G_2: u, x).

Intuitively, the autonomous subgraphs allow to restrict the initial graph to smaller components of it in order to make duplications of vertices, without interaction with the rest of the graph, since no edge comes in or goes out from the autonomous subgraph.

Definition 4.4.1 [BKS92] Let G be an st-multidag. We note $G[v, w]$ a subdag of G with source v and sink w. $G(v, w)$ is an **autonomous subgraph** of G if it satisfies the following property: For any path P from s to t in G, the set of edges in $P \cap G(v, w)$ is empty or forms a path from v to w. Note that v can be s, and w can be t or both, but $G(v, w)$ cannot be the unique edge or the whole graph G. If $G(v, w)$ is an autonomous subgraph of G, we call (v,w) a **separation pair** of G.

A decomposition of G into autonomous subgraphs can be obtained in linear time as proposed by Bein *et al.* in [BKS92]. In the following, we note $G[v, w]$ the subgraph of G which contains all the vertices and edges of all the paths from v to w in G.

Definition 4.4.2 An autonomous subgraph G_{au} is **minimal**, iff $d^+(s(G_{au})) > 1$ and $d^-(t(G_{au})) > 1$.

Example 4.4.1 In Figure 4.14, G_0 has several autonomous subgraphs such as $G[a,b], G[c,d]$ and $G_[s,b]$ whose separation pairs are $(a,b),(c,d)$ and (s,b). As $G[s,b]$ and $G[a,b]$ have the same forbidden subgraphs, we only consider the minimal autonomous subgraphs for our approach. In this case, $G[a,b]$ and $G[c,d]$ are the minimal autonomous subgraphs, and $G[s,b], G[a,t], G[s,d], G[c,t]$ are not the minimal ones.

An autonomous subgraph $G[v,w]$, which is a non-SP graph, can be reduced into a single edge $e(v,w)$, following vertex reduction operations. The reduction vertices v and w may disappear after all the other reduction vertices have been reduced. If v and w are reduction vertices, they may no longer be reduction vertices when all the reduction vertices within the autonomous subgraph are reduced. This implies that the reduction vertices included into an autonomous subgraph should be eliminated first and then the autonomous subgraph can become SP and be represented as one single edge in G_{red}. Following this process, we can obtain one shorter reduction sequence which will lead to a transformation with less duplicated vertices.

To assure that reduction operations never create any new forbidden subgraphs, we constrain the vertex reduction operation to only start from the source of the maximal reduced graph G_{red}. In other words, we always start from the source to choose a successor v of $s(G_{red})$ which is a reduction vertex in order to eliminate forbidden subgraphs. Because the duplicated subgraph induced by edge $e(s(G_{red}),v)$ is an SP graph(property 4.2), we can ensure that no reduction vertex remains in the duplicated subgraph and no non-SP subgraph will be copied.

Property 4.4.2 Let G be an non-SP graph, and let us apply maximal series-parallel reductions on G until no series and parallel reduction can be applied. **There exists at least one successor v of s(G) which is a reduction vertex with $d^-(v) = 1$ and $d^+(v) > 1$.**

Proof: Let $s = s(G), Succ(s) = \{v_1, v_2, \cdots, v_k\}$ be the set of successors of $s, v_i \neq t(G)$. Assume that all the successors of s have an in-degree greater than one, with $d^-(v_i) = n > 1, v_i \in Succ(s)$. If s and its out going edges are removed, new in-degree of v_i is $d^-(v_i) = n - 1 \geq 1$, which means that there does not exist any vertex with in-degree equals to zero. There does not exist any topological order

for graph G, which means there exist circles in G. However, G is an st-multidag. This contradiction shows that there must exist at least one successor v of s with $d^-(v) = 1$. For v, we have $d^+(v) > 1$, or it should be reduced by series reduction. So, we can conclude that there exists at least one successor of s with in-degree one and out-degree greater than one.

Any autonomous subgraph should be considered as a new st-multidag and should be reduced first by reduction operations, and then go back to G_{red} to find another reduction vertex.

4.4.1.3 Reducing redundancy of duplicated vertices

Although we can achieve a reduction sequence which can lead to a transformation with less duplicated vertices by introducing the notion of autonomous subgraph, the risk of redundancy still exists. As shown in Figure 4.15, G_0 is a graph without any autonomous subgraph, but can be transformed into two different graphs by vertex duplication following different reduction sequences. As discussed in section 4.3, vertex duplication approach is provenance-preserving. So two runs which have the same graph structure as G_1 and G_2 are provenance-equivalent. But G_2 obviously has less vertices than G_1, because G_2 follows a minimal reduction sequence and finally vertex a no longer be a reduction vertex. As well-studied, no technique for providing such a minimal reduction sequence has been proposed, and it is difficult to automatically obtain such a sequence. To solve this problem, we remind here the factorization rule which we have proposed in chapter 2. The idea is to eliminate the unnecessary vertices following the factorization rule. For example, in G_1, the copies of vertex a highlighted in yellow can be merged and finally we can obtain another graph G_1' equal to G_2.

As discussed in 3.2, only right distributivity can be used to provide a concise representation of provenance for the execution order is important in the workflow. Corresponding to the factorization rule for the provenance expression, the factorization rule for graphs is shown in Figure 4.16.

To ensure that the factorization never creates new non-SP subgraphs, our algorithm only performs the factorization operation during each parallel reduction. Once a parallel reduction is applied to G_{red}, the algorithm will check redundancy of vertices in the duplicated subgraphs induced by the edges which are reduced

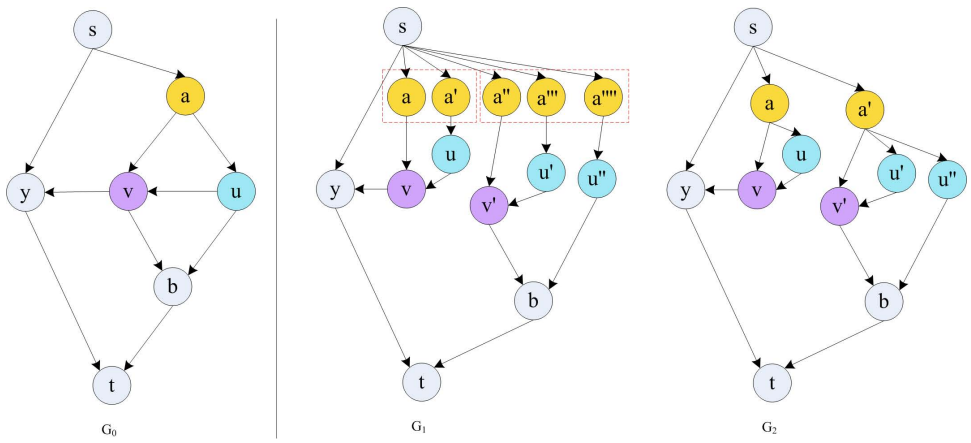


Figure 4.15: G_1 and G_2 are two solutions of rewritting G_0, where G_0 does not contain any autonomous subgraph. (reduction sequence of G_1: a, u, v; reduction sequence of G_2: v, u).

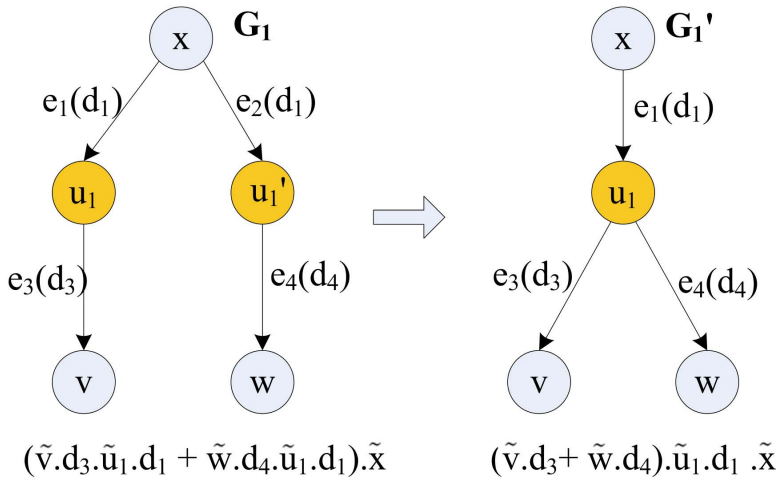


Figure 4.16: Factorization rule for graphs.

in G_{red}. Then, the same vertices will be merged, following the factorization rule shown in Figure 4.16.

We now have presented all the concepts used in the algorithm that we describe in the next subsection.

4.4.1.4 Algorithm description

Principle of the algorithm: Out-vertex duplication algorithm SPFlow takes in the graph G and outputs the two graphs G_{red} and SPG. G_{red} is obtained by successive reductions of G (including out-vertex reductions) until it is the basic

graph BSP. In SPG some vertices of G are duplicated, and these duplications are determined from the out-vertex reductions made in G_{red}. The algorithm will use the procedure $MaxRed$ which takes in and out a graph G and applies iteratively on G series and parallel reductions until such reductions cannot be applied anymore. Vertex duplication algorithm will also use (step 2.ii.3) the procedure *Simpl* that takes in and out SPG and merges some of its subgraphs using the *factorization rule*(cf. Figure 4.16) discussed above each time a parallel reduction performed on G_{red}.

Initialization

(i) $SPG \leftarrow G$; $s \leftarrow s(G)$; $t \leftarrow t(G)$

(ii) $G_{red} \leftarrow MaxRed(G)$

(iii) Split G_{red} into autonomous subgraphs.

(iv) Call $SPFlow(G, G_{red}, SPG, s, t)$.

Procedure SPFlow (IN: G; IN/OUT: G_{red}; IN/OUT: SPG; IN: u, p)

While $G_{red} \neq BSP$ ***do***

Step 1 Choose in G_{red} a vertex v successor of u in G_{red} to which an out-vertex reduction can be applied (cf. Figure 4.2 (b)).

Step 2

(i) v is the source of an autonomous subgraph of sink $w \in G_{red}$. Call SPFlow (G, G_{red}, SPG, v,w), meaning that we consider $G_{red}[v, w]$ instead of G_{red} (v is considered here as the new source of the reduced graph).

(ii) v is not the source of one autonomous subgraph.

1) Duplicate in SPG vertex v $k-1$ times, if v has k successors in G_{red}. In this duplication, instead of duplicating edge e of G_{red}, duplicate the subgraph $SPG[u, v]$ into $SPG[u, v_1], ..., SPG[u, v_{k-1}]$. Similarly, instead of considering edges f_i of G_{red}, consider the subgraphs $SPG[v, w_i]$, which become $SPG[v_1, w_2], ..., SPG[v_{k-1}, w_k]$.

2) Apply out-vertex reduction to v in G_{red}.

3) $G_{red} \leftarrow MaxRed(G_{red})$; $SPG \leftarrow Simpl(SPG)$

End While

End SPFlow

4.4.2 Example of use of SPFlow

We demonstrate the way our algorithm works with an example graph shown in Figure 4.17(a). Its maximal reduced graph G_{red} is shown in Figure 4.17(b) and (c) shown the autonomous subgraphs of (b). Our algorithm will first take the autonomous subgraphs into account and execute procedure SPFlow on each autonomous subgraph in turn. When all the autonomous subgraphs are reduced into a single edge in G_{red}, procedure SPFlow then executes on G_{red}. Finally, G_{red} becomes BSP while SPG be SP.

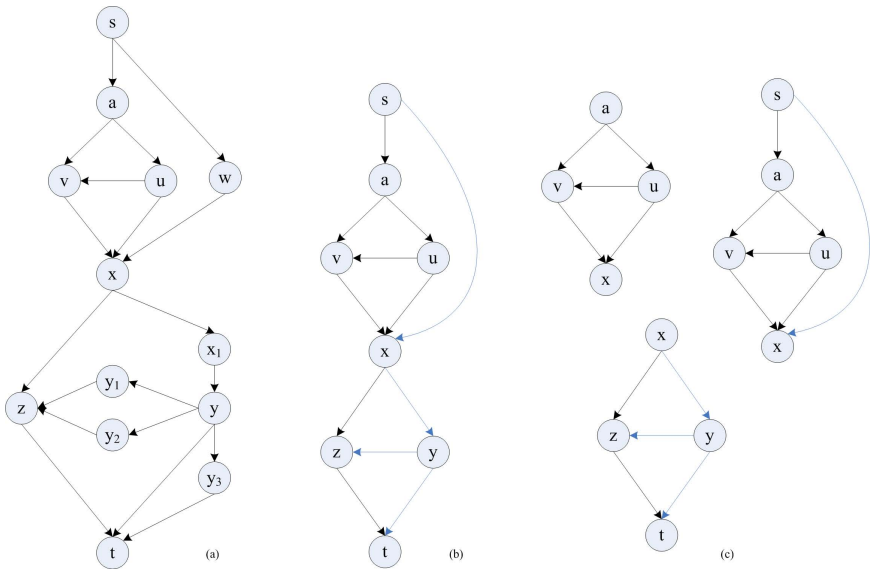


Figure 4.17: Example of non-SP graph (a) and its maximal reduced graph (b) which is split into three autonomous subgraphs (c).

Figure 4.18 illustrates one step of the algorithm for eliminating the forbidden subgraph in autonomous subgraph $G[a, x]$.

4.4.3 Complexity

As expected, SPFlow has an exponential complexity in the worst case. The worst case occurs in the iterated forbidden graph IFG [BKS92] that has $2n + 2$ vertices (see Figure 4.19). We cannot get BSP with less than $2n - 1$ out-vertex reductions from IFG. This number is the maximal number vertex reductions established by [BKS92] for any graph of $2n + 2$ vertices, also called the factoring complexity of the graph. The SPFlow algorithm will build the new SP graph SPG

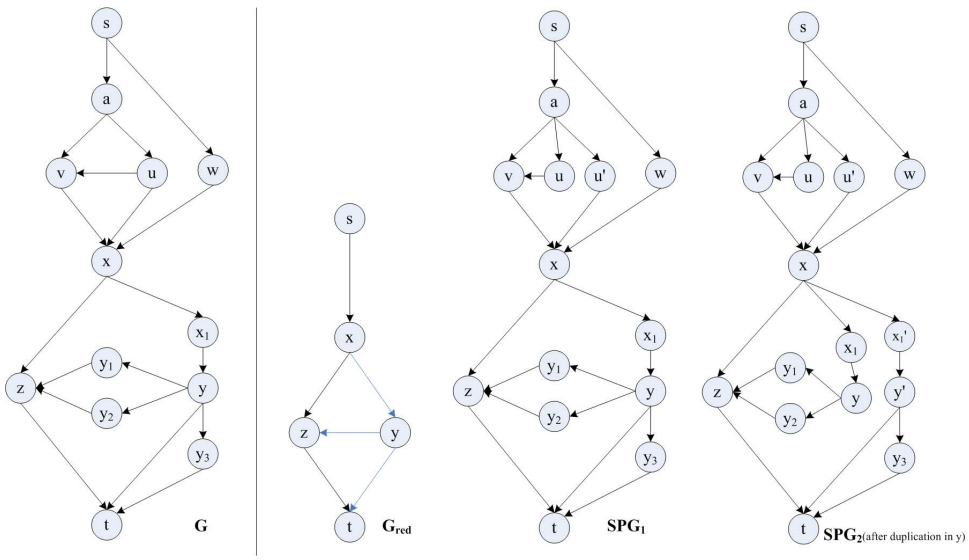


Figure 4.18: Example of one execution step of SPFlow where $G[s,x]$ has already been transformed, giving the edge (s,x) in G_{red} after vertex reduction on u and providing $SPG_1[s,x]$ within SPG_1, after vertex duplication of u. The algorithm then considers x as a successor of s in G_{red}. As (x,t) is a separation pair, it calls again $SPFlow$ considering x as source. Vertex y is the successor of x in G_{red} to which a vertex reduction is applied (in G_{red}). Duplication of y in SPG_1 then leads to SPG_2. For the sake of readability, labels are omitted.

so that the edge outgoing from s(IFG) with in-degree of 1 (edge from s to y_1 will be duplicated an exponential number of times).

Proof:

G is irreducible. There is a single reduction vertex, that is y_1. We perform out-vertex reduction on y_1. Then we can perform out-vertex reduction on the single possible reduction vertex x_1. We iterate the process. Assume that we have reduced $y_1, x_1, y_2, x_2, ..., y_{i-1}, x_{i-1}$. Let us call α_i the new label of the edge from s to x_i and β_i the label of the new edge from s to y_i (cf. Figure 4.19 (b)).

Base case: $\alpha_1 = a_1$ and $\beta_1 = b_1$

Induction case: for $0 < i < n-1$, after reduction of vertices $y_1, x_1, \cdots, y_i$ (cf. Figure 4.19 (c)) and then reduction of vertex x_i (cf Figure 4.19 (d)), we get :

$\alpha_{i+1} = (\alpha_i + \beta_i \cdot c_i) \cdot a_{i_1}$ (label of the new edge from s to x_{i+1})

$\beta_{i+1} = ((\alpha_i + \beta_i \cdot c_i) \cdot d_{i+1}) + \beta_i \cdot b_{i+1}$ (label of the new edge from s to y_{i+1}.)

In the following, we focus on the duplication of the edge from s to y_1 in the

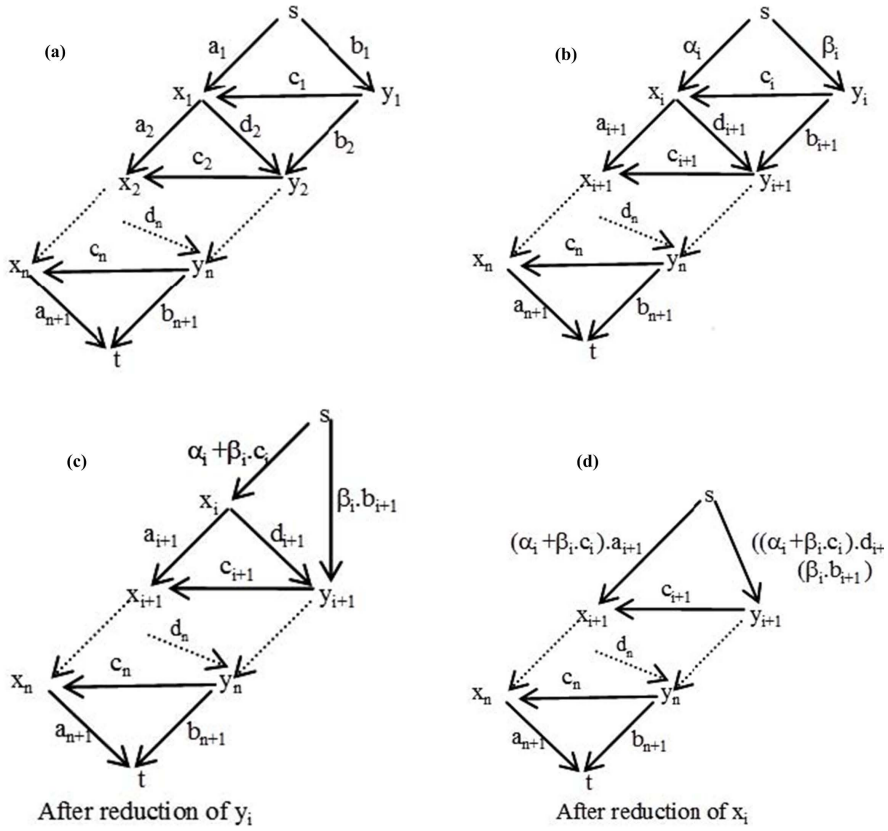


Figure 4.19: Example of iterated forbidden graph (IFG).

initial graph G. For it we count the number of occurrences of its label b_1 in the final label of the edge from s to y_n. Let us call u_i the number of occurrences of b_1 in α_i, and v_i the number of occurrences of b_1 in β_i. We get two recurrence relations:

(1) $u_{i+1} = u_i + v_i$, for $2 \leq i \leq n-1$

(2) $v_{i+1} = u_i + 2v_i$, for $1 \leq i \leq n-1$,

with $v_1 = 1$ and $u_1 = 0, u_2 = 1$. Solving these equations, we get:

$v_n = (\gamma_1 * (\frac{(3+\sqrt{5})}{2})^n + \gamma_2 * (\frac{(3-\sqrt{5})}{2})^n)$ with $\gamma_1 = 0.28$ and $\gamma_2 = 0.72$.

For example, if $n = 20$ (resp. $n = 50$), this edge will be duplicated more than 107 (resp. 1,020) times. The next section of experimental study will show that on real workflows, the complexity is reasonable.

4.4.4 Soundness of vertex duplication algorithm

As a central result we get that the main output of SPFlow is SP and is provenance-equivalent to the non-SP graph taken as input. Additionally, the output provenance of the initial graph is directly obtained from the final label of the unique edge of G_{red}.

More precisely, we establish the following properties by induction on the number of reduction steps of G_{red} in the algorithm.

Property 4.4.3 (i) At any step of the algorithm: $MaxRed(G_{red}) = MaxRed(SPG)$.
(ii) For each vertex w of G_{red} in SPG and G: $Hist_{SPG}(w) = Hist_G(w)$=$Hist_{G_{red}}(w)$.

Indeed, to each edge (u, v) of G_{red} correspond a subgraph $SPG[u, v]$ which is SP and to the vertex reduction in G_{red} corresponds a duplication in SPG

Property 4.4.4 For all vertex w of G_{red} in SPG and in G we have: $Hist(w)_{SPG}$ $= Hist(w)_G$=$Hist(w)_{G_{red}}$.

This property is a consequence of the properties of reduction operations in section 2.3 and 4.2.

Theorem 4.4.1 At the end of the SPFlow algorithm:
(1) SPG is an SP graph ;
(2) $OutProv(G) = OutProv(SPG)$;
(3) let f be the unique edge from s to t in G_{red}, then $OutProv(G)$= $L_{2er}(f) \cdot \tilde{s}$.

Sketch of proof:

1. At the end of the algorithm, $G_{red} = BSP$ and $G_{red} = MaxRed(G_{red})$. Besides, $MaxRed(G_{red}) = MaxRed(SPG)$ (property 4.4). Thus $MaxRed(SPG) = BSP$.

2. $OutProv(G) = Hist(t)_G$; besides $Hist(t)_G = Hist(t)_{G_{red}}$=$Hist(t)_{G_{SPG}}$ (property 4.5). Then $Hist(t)_G = Hist(t)_{SPG} = OutProv(SPG)$.

Table 4.1: Evolution of SP structures in workflows of myExperiment

Date	Number of workflows	SP graphs (proportion)	non-SP graphs (proportion)
2010	681	429 (63%)	252 (37%)
2011	879	554 (63%)	325 (37%)
2012	1014	624 (61,5%)	390 (38,5%)
2013	1454	833 (57,3%)	621 (42,7%)

Table 4.2: non-SP vs SP structures in families

Family (#vertices)	#workflows	% of SP structures
Simple (1-10)	848	82.2 %
Complex (11-20)	282	44 %
Very complex ($\succ$ 20)	324	6.2 %

4.5 Experimental study

Our experiments run on a subset of 1,454 workflows extracted from the Taverna workflows available in myExperiment [GFG+09] in July 2013 (removing duplicates and considering only well-formed workflows). In this section, we study their structures and evaluate SPFlow.

4.5.1 Workflow Structures

We have represented workflows by st-multidags (adding a source and a sink) and have implemented a basic SP structure detection algorithm.

Proportion of SP and non-SP workflows

Our first result (Table 4.1) shows that there is a majority of SP structures and the proportion of SP-graphs is stable over time. Table 4.2 provides the distribution of SP vs non-SP workflows, considering three families of workflows. The figures obtained are particularly clear: while intermediate workflows are almost all SP, the proportion of SP structures in workflows falls over 10 vertices with only 6.2% of SP workflows in very complex workflows.

Features of non-SP workflows

In this second experiment, we evaluate the *distance* between non-SP and SP structures, given by the number of vertex reductions to be applied to get the basic graph BSP [BKS92]. Figure 4.20 shows that among non-SP workflows, 27% of them have one reduction vertex, 62% have only 1 to 3 reduction vertices. They are thus not very far from SP structures.

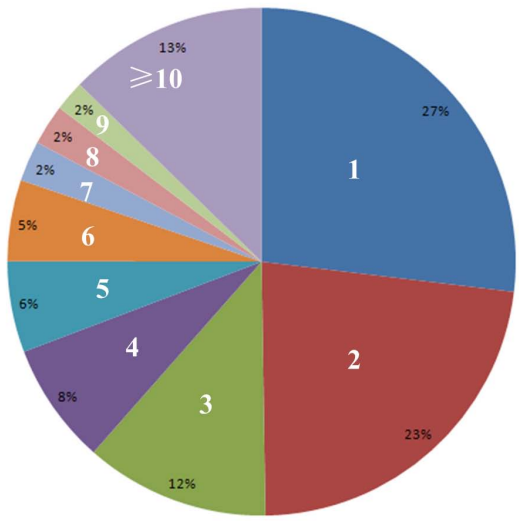


Figure 4.20: Percentage of workflows with a given number of reduction vertices in non-SP structures.

4.5.2 Evaluating SPFlow

We now evaluate the behavior of SPFlow on real data, considering the set of 621 non-SP Taverna workflows available in myExperiment (whose size ranges from 4 to 333 vertices). Figure 4.21 gives the relationship between the size (number of vertices) of the initial graph and the rewritten graph. Although 10 graphs have an important number of duplicated vertices, half of the very large majority of graphs, including huge workflows (having more than 100 vertices), have a small ratio, lower than 5. Additionally, the time to rewrite each workflow is negligible for the current structures of workflows: on a dual core@2.2GHz and 2GB of RAM desktop, the maximum time is 434 ms.

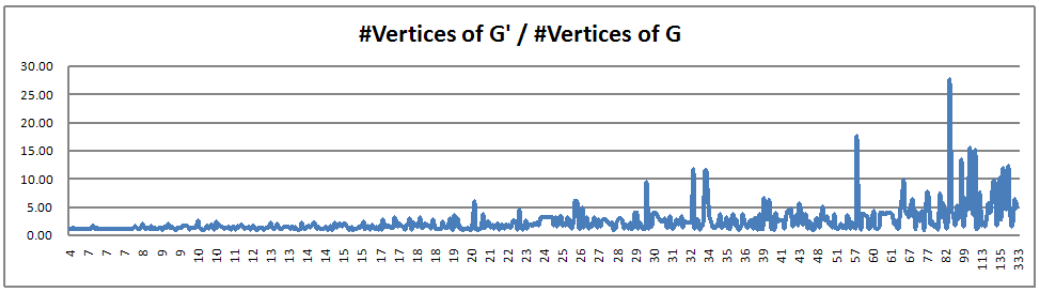


Figure 4.21: Ratio between the number of vertices in the rewritten graph (G') and the initial graph (G) in function of the size of G.

4.6 Implementation of the algorithm

In this section, we introduce a java application tool based on the algorithm described in section 4.4 and in [CCBF13], named **SPFlow**, which aims at rewriting a non SP workflow into an SP workflow while preserving provenance. Current version of SPFlow supports Taverna 2 and ZoomUserView input worflows.

4.6.1 SPFlow architecture

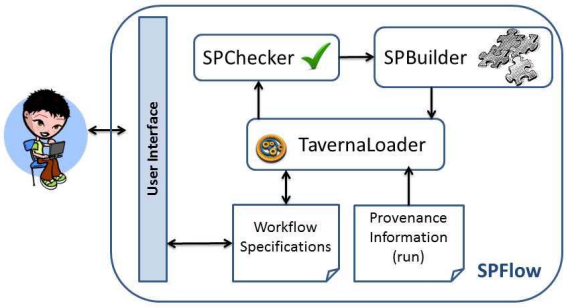


Figure 4.22: Architecture of SPFlow

SPFlow transforms any workflow having a non-SP structure into a provenance-equivalent SP structured workflow. The architecture of SPFlow is provided on Figure 4.22 and described here after. SPFlow makes use of Workflow Specifications and Provenance Information provided by users or workflow systems. The current version of SPFlow is able to rewrite real workflows from the Taverna system (other systems are under consideration). The *TavernaLoader* module is thus responsible for loading the workflow into the SPFlow internal graph structure. *SPChecker* then determines whether or not the workflow taken in has an SP structure and pro-

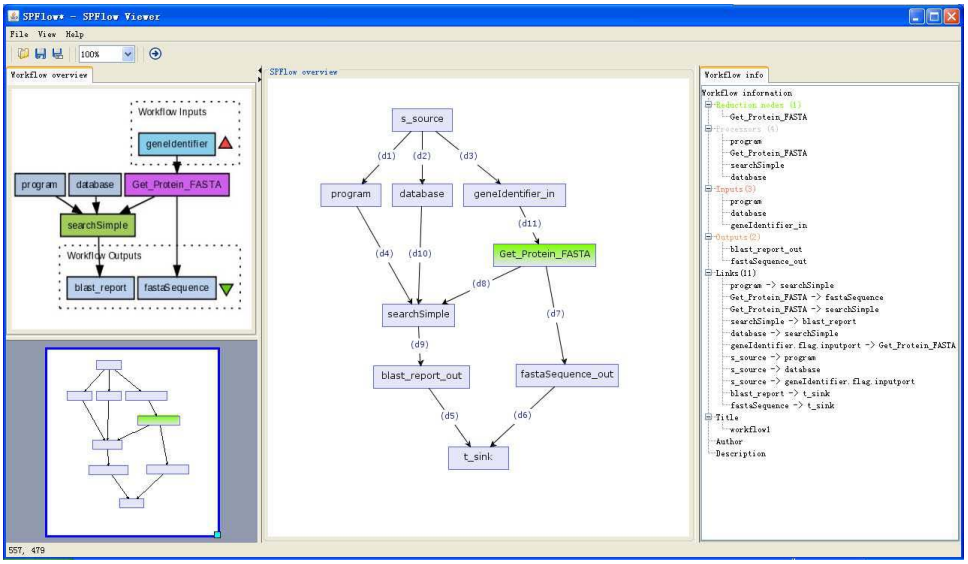


Figure 4.23: Loading a workflow in SPFlow

vides a report with graph features, including the identification of reduction nodes (if any). If the workflow is not SP, it is sent to *SPBuilder* which then creates a new provenance-equivalent workflow graph having some duplicated vertices compared to the original workflow, following the process described in [CBFJ12]. Finally, the TavernaLoader module produces the rewritten workflow into the Taverna XML format and makes it available to the user.

Users communicate with the system by loading and interacting with original and rewritten workflows.

4.6.2 Functionalities of SPFlow

Our implementation of SPFlow is able to provide the following features.

Loading Data: Users may load a workflow specification into the system (see Figure 4.23). SPFlow will display the original picture of the workflow from myExperiment [RGS09] if available (left panel), determine (using SPChecker) the reduction nodes (if any) and highlight them (central panel). A report on graph features is produced (metadata on the workflow, right panel).

Rewriting of the workflow: SPFlow (using SPBuilder) transforms any non-SP workflow into an SP workflow (see Figure 4.24). Both workflows will be displayed and duplicated vertices highlighted.

Provenance information: By clicking on an edge between two tasks, the user

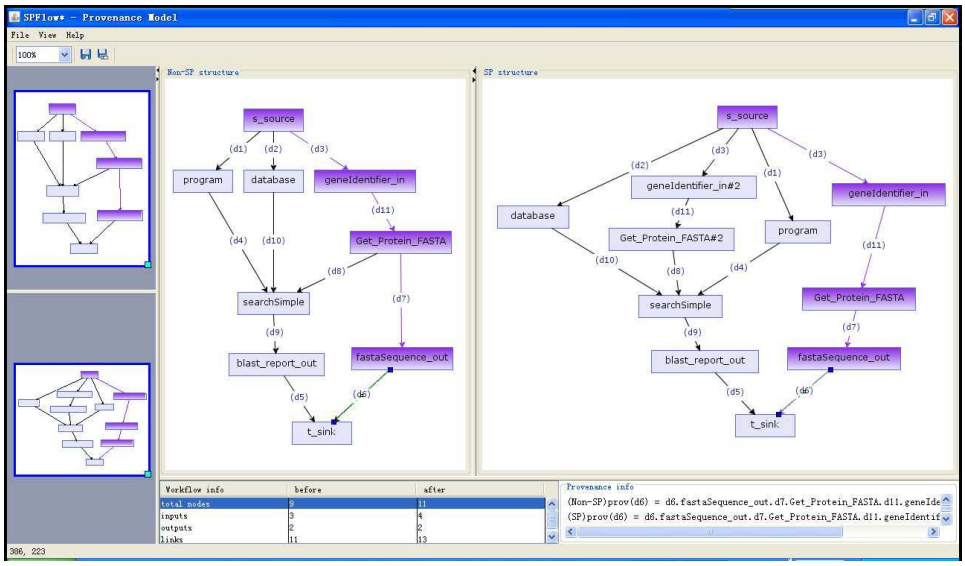


Figure 4.24: Provenance information in SPFlow

can visualize the provenance information (see Figure 4.24) of the data flowing on that edge not only on the initial workflow but also on the rewritten workflow (showing that both workflows are provenance-equivalent). The formal expression associated to provenance information is also displayed (bottom panel).

Running rewritten workflows: Any workflow rewritten by SPFlow can be opened in Taverna. We will show how it can be run and we will demonstrate that both workflow versions (non-SP and SP) provide the same results for the same input (equivalence property).

On the benefit of using SP-workflows: We will take the example of the Zoom*userview system (ZOOM for short) [BBDH08] that takes in a workflow and a set of tasks of interest for the user (other tasks are usually formatting tasks) and provides a user view, that is, a view of the workflow composed of a set of composite tasks. Each composite task contains at most one significant task and takes its meaning. The difficulty for ZOOM lies in ensuring that no data dependencies between significant composite tasks is introduced or lost by the grouping process (*i.e.* consider two relevant tasks t_1 and t_2: t_1 consumes the data produced by t_2 if and only if the composite task containing t_1 consumes the data produced by the composite task containing t_2).

In Figure 4.25, the user has specified two tasks of interest to him (namely, blast-report and Fasta-sequence). Based on the original workflow (figure 4.25 (A)),

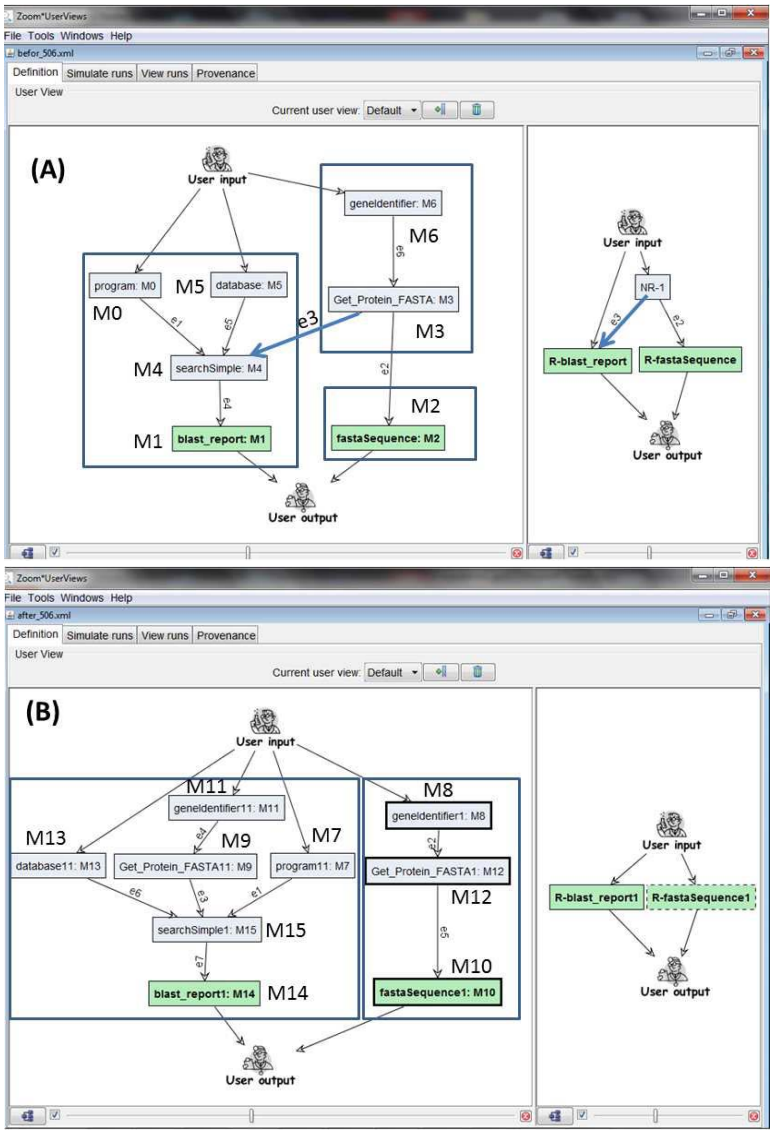


Figure 4.25: Provenance information in (A) non-SP and (B) SP version of the workflow in ZOOM. User views are displayed on the right while full workflows are on the left.

ZOOM designs the user view on the right, which is composed of three composite tasks, one focused on blast-report (R-blast report which contains M_0, M_5, M_4 and M_1), another on fastaSequence (R-fastaSequence which contains only M_2) and unfortunately one task with no significance for the user (NR-1 which contains M_6 and M_3). Note that introducing the tasks of NR-1 into one of the two significant composite tasks would have introduced misleading data dependencies: e.g., if M_3 and M_6 were put into R-fastaSequence then from the user view perspective the edge e_3 would have been displayed from R-fastaSequence to R-blast report, giving the feeling to the user that data provided by R-fastaSequence is used by R-blast report while it was not the case in the original workflow. It has been proved in [BDKR09] that such a situation (having to introduce a composite task without any significance for the user to preserve provenance) can be avoided when SP structures are used while it is not possible for general DAGs.

In Figure 4.25 (B), the rewriting process of SPFlow has duplicated M_6 and M_3 from workflow (A) into M_{11}, M_8 and M_9, M_{12} in workflow (B). As a consequence, the user view designed by ZOOM is only based on significant composite tasks (R-blast report which contains $M_{11}, M_{13}, M_9, M_7, M_{15}$ and M_{14}, and R-fastaSequence which contains M_8, M_{12} and M_{10}). Such a workflow is then more user-friendly. In particular, each of the two composite tasks takes in now only user input and is then clearly easier to share and (re)use in another context.

4.7 Discussion

Scientific workflows are complex graphs that need to be designed, visualized, queried, run, or scheduled. These actions are inherently complex and lead to NP-hard problems when conducted on DAGs like are usual scientific workflows. Instead, these problems can be solved in polynomial time when the structure is series-parallel (SP). Rewriting a non-SP to SP workflow is particularly useful especially if the provenance is preserved. The major contribution of this work is the introduction of an original algorithm for rewriting workflows preserving provenance. More particularly, we: (1) reviewed existing approaches and discussed whether they are provenance-preserving, (2) designed the provenance-equivalent SPFlow algorithm, (3) demonstrated the feasibility of our approach on real sci-

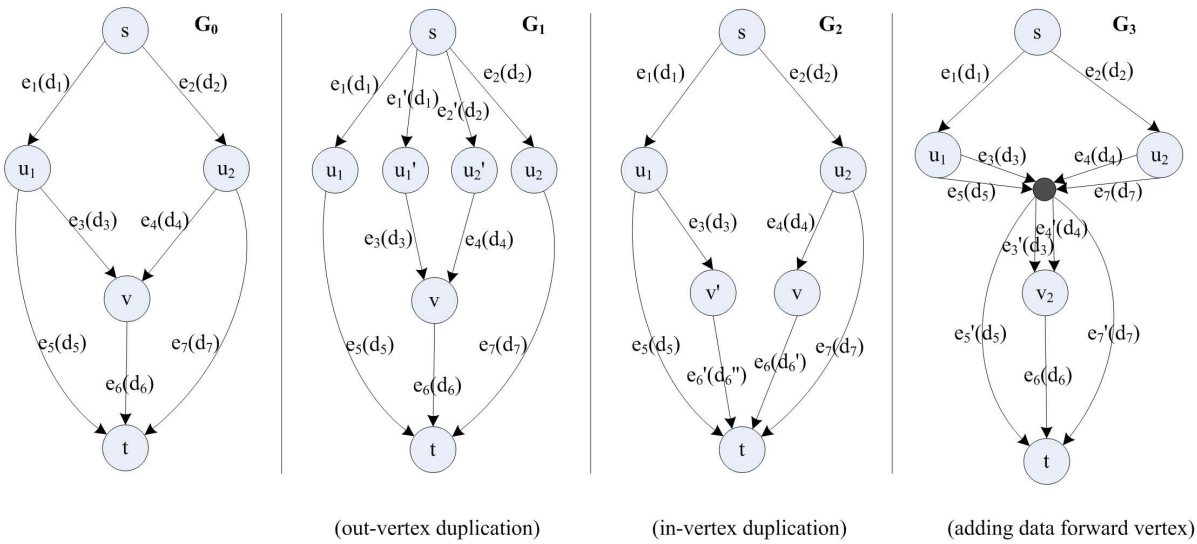


Figure 4.26: Example of different solutions for unsupported patterns.

entific workflows, (4) developed a tool taking in a non-SP Taverna workflow and providing an SP version of the workflow usable in Taverna.

We now provide one direction of extension for our work.

Extending SPFlow to deal with split and merge processors. As discussed in Chapter 3, there are some special cases which are currently not supported by SPFlow, such as when a **merge processor** or a **split processor** appears in a workflow (they are currently considered as any other processor). As the two processors are closely related, we only discuss the case concerning the merge processor. Let us consider Figure 4.26, G_0 is a non-SP graph with parallel composition and let us consider that v is a merge processor and $d_6 = [d'_6, d''_6]$. G_1, G_2, G_3 are different SP solutions for G_0. G_1 is obtained from G_0 based on out-vertex reduction. G_2 is obtained from G_0 based on in-vertex reduction and G_3 is based on the strategy of adding dependencies. Let us observe these SP graphs and the original graph, we could find that they all produce the same intermediate and final data. So, they should be provenance-equivalent. It is obvious that G_2 and G_3 have less vertices duplicated than G_1. This implies that it is possible to reduce the number of duplicated vertices when considering some special processors such as a merge processor. More importantly, the merge processor, contrary to any other processor does not need to get all its inputs to provide one output. As a consequence, solutions based on in-vertex duplications (as in G_2) may be more appropriate than solutions based

on out-vertex duplications (as in G_1) when lists of data are considered. So, finding a strategy to deal with these specific processors can help us to obtain new SP workflows with less duplicated vertices and unambiguous provenance meaning (cf. G_0). How to extend our provenance model and SPFlow to support lists of data and consider these special processors is one direction of our ongoing work.

4.8 Summary

In this chapter, we have presented SPFlow a provenance-based strategy for rewriting any non-SP graph into SP graph. After having studied several current approaches, we have identified that all of them are not provenance-preserving. So that they cannot be directly used to rewrite workflows into equivalent ones. Our approach based on out-vertex duplication which is provenance-preserving was then proposed. We also demonstrated the feasibility of our approach on real scientific workflows. Finally, we gave an introduction of the tool we developed, which takes in a non-SP Taverna workflow and provides an SP version of the workflow useable in Taverna.

As studied in this chapter, we are able to rewrite any scientific workflow into SP structure. A new question is whether it is possible to rewrite a scientific workflow into a new one which is free or partly free of vertices redundancy and without alerting its meaning. In the next chapter, we will inspect the features of Taverna workflows themselves and then provide a refactoring approach to relax redundancy of vertices and make workflows close to SP structures.

Chapter 5

Distilling Structure in Taverna Scientific Workflows: A refactoring approach

Contents

5.1 Use cases **81**

5.2 Anti-patterns and Transformations **82**

5.2.1 Assumptions 83

5.2.2 Transformations 83

5.2.3 Safe Transformations 85

5.3 Refactoring approach **86**

5.3.1 Principle of the algorithm 87

5.3.2 Illustration of the algorithm 88

5.4 Experimental Study **91**

5.4.1 Anti-patterns in workflow sets 91

5.4.2 Results obtained by DistillFlow 92

5.5 Discussion **94**

5.5.1 Simpler structures 94

5.5.2 SP structures 95

5.5.3 Towards other kinds of (anti-)patterns 99

5.5.4 Provenance-equivalence 100

5.6 Related Work **101**

5.7 Summary **102**

Chapter 4 has introduced a provenance-based technique for rewriting any non-SP workflow into an SP workflow. Still in the aim of transforming workflow structures to make them easier to reuse, the present chapter introduces techniques for reducing redundancies in the structure of scientific workflows. Our approach provides workflows which are free or partly free of redundant vertices without alerting their original meaning. Interestingly, we will see that our approach tends to make non-SP workfows closer to SP structures.

More precisely, our approach aims at automatically detecting parts of the workflow structure which can be simplified by removing explicit redundancy and proposing a possible workflow rewriting. As mentioned earlier, our preliminary analysis of the structure of 1,400 scientific workflows of Taverna collected from myExperiments reveals that, in numerous cases, such a complexity is due mainly to redundancy, which is in turn an indication of over-complicated design, and thus there is a chance for a reduction in complexity which does not alter the workflow semantics. Our main contention in this work is that such a reduction in complexity can be performed automatically, and that it will be beneficial both in terms of user experience (easier design and maintenance), and in terms of operational efficiency (easier to manage, and sometimes to exploit the latent parallelism amongst the tasks).

The specific contribution of this chapter is a method for the automated detection and correction of certain Taverna workflow structures which can benefit from refactoring. We call these idiomatic structures "anti-patterns", that is, patterns that should be avoided. Our approach involves the detection of several anti-patterns and the rewriting of the offending graph fragment using a new pattern that exhibits less redundancy and simpler structure while preserving the semantics of the original workflow. We have then designed the *DistillFlow* algorithm and evaluated its effectiveness both on a public collection of Taverna workflows and on a private collection of workflows from the BioVel project.

As the Taverna workflow system features have already summarized in chapter 2, the present chapter begins by illustrating the two main types of anti-patterns found by our workflow study, by means of two use cases (5.1). The formalization

of the anti-patterns and the transformations we propose to do while ensuring that the semantics of the workflow remains unchanged will be then introduced (5.2). After giving a presentation to the anti-patterns, we will introduce the DistillFlow refactoring algorithm (5.3). In the experimental study Section (5.4), we provide the results obtained by our approach on a large set of real workflows. Finally a discussion of this work will be carried out, together with the conclusion.

5.1 Use cases

The first use case (Figure 5.1 (i)) involves the duplication of a linear chain of connected processors *GetStatistics_input*, *GetStatistics* and *GetStatistics_output*. The last processor in the chain reveals the rationale for this design, namely to use *one output port from each copy of the processor*. Clearly, this is unnecessary, and the version in Figure 1 (ii) achieves the same effect much more economically, by drawing both output values from the same copy of the processor.

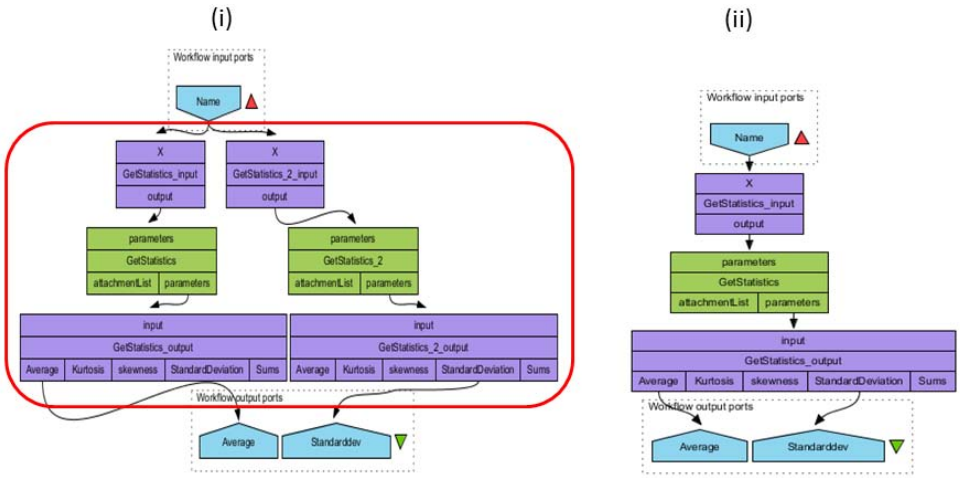


Figure 5.1: Example of workflow (myExperiment 2383)

In the second use case (Figure 5.2(i)), the workflow begins with three distinct processing steps on the same input sequence. We observe that the three steps that follow those are really all copies of a master *Get_image_From_URL* task. This suggests that their three inputs can be collected into a list, and the three occurrences can be factored into a single occurrence which consumes the list. By virtue of the Taverna list processing feature described earlier, the single occurrence

will be activated three times, one for each element in the input list. Also, the outputs of the repeated calls of *Get_image_From_URL* will be in the same order as items in the list. Therefore this new pattern achieves the same result as the original workflow. Note that collecting the three outputs into a list requires a new built-in *merge* node (the circle icon in Figure 5.2(ii)). Similarly, a *Split* processor has been introduced to decompose the outputs (list of values) into three single outputs.

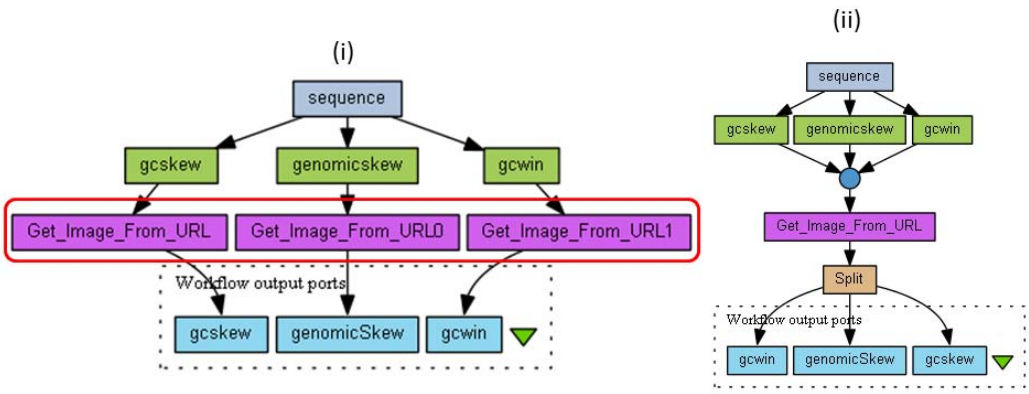


Figure 5.2: Example of workflow (myExperiment 804)

These two examples are instances of the general patterns depicted in Figures 5.3 and 5.4 (left hand side). These are the *anti-patterns* we alluded to earlier, and our goal is to rewrite them into the new structures shown in the right hand side of the figures. In the rest of this chapter we describe this rewriting process in detail.

5.2 Anti-patterns and Transformations

The transformations aim at reducing the complexity of the workflow by replacing several occurrences of the same processor with one single occurrence whenever possible. Although new processors are sometimes introduced in the process (*i.e.* merge and split operators), on balance we expect a cleaner design, better use of the functional features of Taverna (automated list processing) and lower redundancy, and thus fewer maintenance problems.

5.2.1 Assumptions

The following four assumptions must hold for processor instances to be candidates for the transformations described below.

1. A processor must be **deterministic**: it should always produce the same output given the same input.

2. Only processors implemented using the **exact same code** can be merged. Determining that two processors are equivalent is an open problem (see e.g. [SCBL12] for a discussion on that point) since it is directly associated to determining the equivalence of programs. In our setting, two processors are equivalent if they represent identical web service calls, or they contain the same script, or they are bound to the same executable Java program. In practice, this condition is often realized, because processors are duplicated during workflow design by means of a graphical "copy and paste" operation.

3. Only copies of processors that **do not depend on each other** can be merged, that is, if $P^{(1)}$ and $P^{(2)}$ are two occurrences of the same processor P, then there should not be any directed path between $P^{(1)}$ and $P^{(2)}$, for $P^{(1)}$ and $P^{(2)}$ to be merged.

4. We will consider only two cases where we can be sure that the **same input value** L_i can be bound to the input port a_i of r copies of P: (a) the input port a_i is bound to a constant value which is identical across executions (that is, among different copies) of P, or (b) L_i has been produced by the output port of some processor Q_i and has been distributed to the r copies of P.

5.2.2 Transformations

The two proposed transformations are shown in Figures 5.3 and 5.4, where each $P^{(l)}$ $(1 \leq l \leq r)$ denotes an occurrence (*i.e.* a copy) of processor P, with input and output ports $a_1, ..., a_k$ and $b_1, ..., b_q$, respectively.

Anti-pattern A: In the first anti-pattern (Figure 5.3), the input ports a_i of each processor occurrence $P^{(l)}$ are all bound to the same value L_i, for $1 \leq i \leq k$, $1 \leq l \leq r$. It follows from our assumption of determinism that the output ports

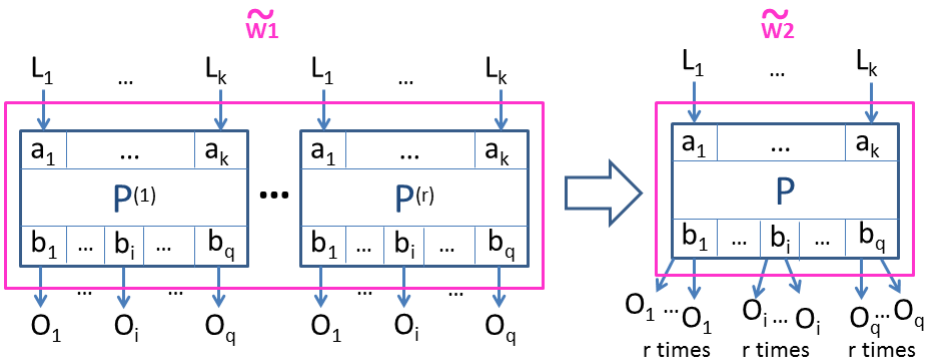


Figure 5.3: Transformation for anti-pattern (A)

b_j all present the same output value O_j across all $P^{(l)}$, for $1 \leq j \leq q$.
The rewriting replaces all $P^{(l)}$ with a single occurrence, P.

Treatment of the outputs: Outgoing links are then added to ports b_j as needed.

Treatment of the inputs: For each input port a_i of P, the unique input value L_i bound to a_i is now either the constant value as previously in the (original) anti-pattern (cf. assumption 4.(a)), or it is one of the distributed values bound to some output port of some processor Q_i (assumption 4.(b)) and in this last case processor Q_i does not need to distribute this output value more than once anymore.

Illustration: One example of anti-pattern A is depicted on Figure 5.1(i) where the same workflow input is sent to two exact copies of the processor *GetStatistics_input*. The workflow input plays the role of processor Q. *GetStatistics_input* and *GetStatistics_2_input* are thus merged and the workflow input (*Name*) is sent only once to the downstream of the workflow, that is, to the (now) single *GetStatistics_input* processor. Outputs are linked to the rest of the workflow and transformations must be applied as many times as necessary. In this example, three successive transformations are applied thus giving the workflow of Figure 5.1(ii).

Anti-pattern B: In the second pattern (Figure 5.4), the input ports a_i of each processor occurrence $P^{(l)}$ are bound to the same value L_i, for $1 \leq i \leq t$ while the input ports a_{t+1} to a_k of each processor occurrence $P^{(l)}$ are bound to different inputs L^l_{t+1} to L^l_k among occurrences, $1 \leq l \leq r$. As for output values, let $O^l_i = P^{(l)}|_{bi}(L_1, ..., L_t, L^l_{t+1}, ..., L^l_k)$ denotes the output value produced by output port b_i of the l-th occurrence of P. For the sake of generality, we consider here

that processor P applies cross product to values on ports a_1 to a_t and dot product to values on ports a_{t+1} through a_k.
The rewriting replaces all $P^{(l)}$ with a single occurrence, P.
Input data that differ from one occurrence to another (L^l_{t+1} to L^l_k) have been merged using the merge processors provided by Taverna (the circle icon in Figure 5.4) to construct lists of data from the original data items to exploit the implicit iterative process of Taverna. As a consequence, the outputs of P are lists of data instead of single values in the original pattern. Since P follows a dot strategy on ports a_{t+1}... a_k, O'_i is the list $O'_i = [P|_{bi}(L_1, ..., L_t, L^1_{t+1}, ..., L^1_k), ..., P|_{bi}(L_1, ..., L_t, L^l_{t+1}, ..., L^l_k)$,...,$P|_{bi}(L_1, ..., L_t, L^r_{t+1}, ..., L^r_k)]$, for output port b_i, $1 \leq i \leq q$.
Treatment of the outputs: For each output port b_i of P, the rewritten pattern contains a *list split* processor called $SPLIT_r$ to decompose the list obtained into r pieces so that the downstream fragment of the workflow remains unchanged. We get: $O'^l_i = P|_{bi}(L_1, ..., L_t, L^l_{t+1}, ..., L^l_k)$ $(1 \leq l \leq r)$.
Treatment of the inputs: Note that for each input port a_{t+1},...,a_k, input values L^l_i are used in the same way both before and after the transformation ($1 \leq l \leq r$, $t + 1 \leq i \leq k$). As for input ports a_1 to a_t, instead of having r occurrences, each L_i has now one single occurrence, $1 \leq i \leq t$ (similarly to anti-pattern A).
Illustration: One example of anti-pattern B is depicted on Figure 5.2(i) where there are three copies of processor $Get_image_From_URL$, each copy receiving input data from distinct processors. The three copies are then merged into one single copy.

The next section will provide more details on how the transformations are extended to the entire workflow.

5.2.3 Safe Transformations

In this subsection, we introduce the notion of safe transformation. Intuitively, a transformation is safe if the semantics of the workflow is preserved (the outputs produced remain the same).

More formally, let W_1 be a fragment of a workflow W consisting of r occurrences $P^{(1)}...P^{(r)}$ of a processor P such that there is no directed path between $P^{(i)}$ and $P^{(j)}$ $(1 \leq i \neq j \leq r)$. Let W_2 be a fragment of the workflow W consisting in one

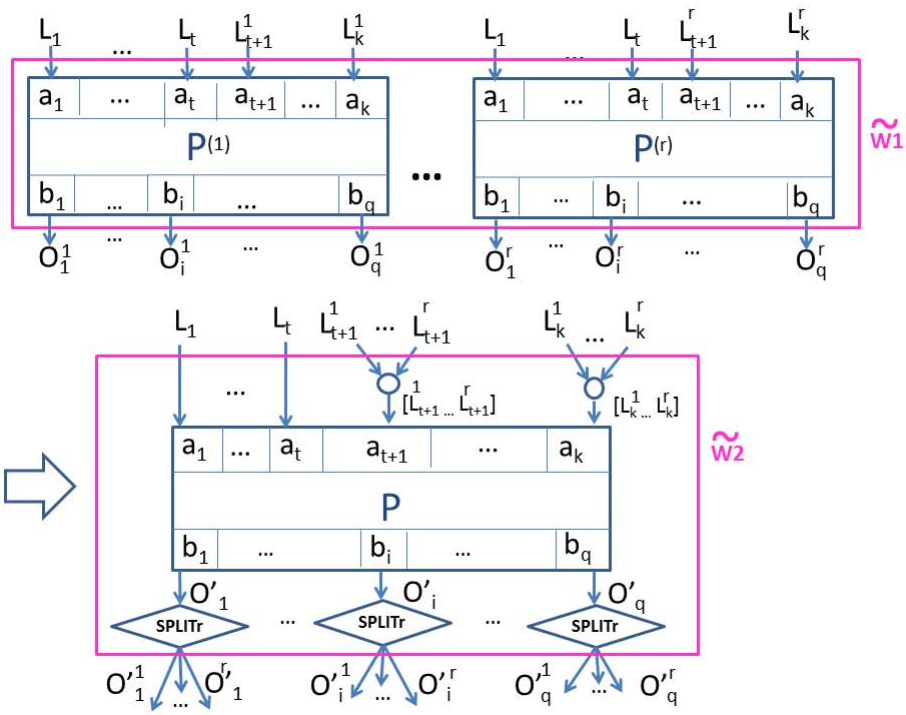


Figure 5.4: Transformation for anti-pattern (B)

occurrence of P and possibly merge and split processors. A transformation that replaces W_1 by W_2 in the workflow W resulting in W' is **safe** if and only if: given the same workflow input values In, for any execution of W using In, named $\tilde{W}$, and any execution of W' using In, named $\tilde{W}'$, the workflow output values Out obtained by $\tilde{W}$ and $\tilde{W}'$ are the same.

It is straightforward to prove that the two transformations we propose to perform are safe.

5.3 Refactoring approach

The previous section has introduced transformations able to locally remove anti-patterns. In this section, we will present the complete refactoring procedure we propose to follow. In particular, we have chosen not to remove all possible anti-patterns when such rewriting operations can make the transformed structures becoming more intricate than the original structures. Example of "simple" structures are *series-parallel* (SP) graphs as introduced in the previous chapters. The challenge of our refactoring approach then lies in minimizing the presence of

anti-patterns while ensuring that the number of structures which are not SP will not increase. Note that it may be the case that our procedure transforms some non-SP structures into SP structures.

As said in the previous chapter, non-SP structures have some specific nodes called **reduction nodes** which cause the structure to be non-SP. Reduction nodes are typically involved in structures illustrated in the subgraph of Figure 5.15 (iii) where u is one reduction node. We will see how we apply our transformations to such nodes and we go back to this point in the Discussion section.

Additionally, in the following, we will also make use of the notion of **autonomous subgraph** introduced in the context of SP structures in Chapter 2. In the same spirit as in the SPFlow approach, the autonomous subgraphs allow to restrict the initial graph to smaller components such that no edge comes in or goes out of the autonomous subgraph (except edges coming in the source of the autonomous subgraph or going out of its target). Recall that several autonomous subgraphs can be nested. Consider the graph G in Figure 5.8(b), examples of autonomous subgraphs are $G[7, 24]$, $G[8, 25]$ and $G[3, 24]$, where $G[7, 24]$ is nested in $G[3, 24]$. We will use this notion in order to apply transformations locally, without interaction with the rest of the graph.

5.3.1 Principle of the algorithm

The Refactoring algorithm takes in an st-DAG G and produces an st-DAG DSG from G by transforming the anti-patterns that can be removed from G while preserving its SP property. For it, the algorithm starts by identifying the set $SetAU$ of autonomous subgraphs, and distills each of them, starting with the minimal ones, in a recursive way. Once each autonomous subgraph has been distilled, the whole graph G must be distilled in turn. Calls of the procedure *Distill* are done from a starting node x that can be either the source of an autonomous subgraph or a reduction node, or the source of G. We consider all the successors p of x, and search among all the other successors (and then descendants of x) whether there is a processor q that would be a copy of p. If it the case, we merge p and q according to the transformation for anti-patterns (A) and (B). Every time a transformation is performed, merging copies of a processor may give rise to new autonomous subgraphs, that lead to new distillations in turn. This last job is done

by the procedure Down-Distillation.

Figure 5.5 presents the main DistillFlow algorithm while the two procedures it uses, namely *DownDistillation* and *Distill*, for transforming workflows are available in Figures 5.6 and 5.7. One major and additional function used by the procedure is introduced here after: $OKTransformation(p, q, GG)$ which specifies the conditions for nodes p and q to be merged. It is true iff the following conditions are satisfied: (i) p and q are copies of each other; (ii) p and q are involved in some anti-pattern (A) or (B) in GG; (iii) for any autonomous subgraph G' of GG, every time p appears in G', q appears in G' too. This last condition ensures that we do not remove an anti-pattern by a transformation that would make an SP-graph becoming non-SP.

```
1 START DistillFlow
2 DSG ← G; s ← Source(G);
3 AU ← set of autonomous subgraphs of G ordered by inclusion;
4 foreach subgraph G[u,v] of AU, starting with minimal subgraphs do
5  |  Distill(G[u,v], DSG, u)
6 end
7 Distill(G, DSG, s);
8 END DistillFlow
```

Figure 5.5: Pseudo-code of the DistillFlow algorithm for removing anti-patterns in workflows

```
1 DownDistillation(IN GG[q,v], IN/OUT DSGG: graphs, IN q: node,
2                  IN/OUT SetAU: set of graphs, IN/OUT ListRed: set of nodes)
3 Distill(GG[q,v], DSGG, q);
4 ListRed ← ListRed ∪ {new reduction nodes of GG[q,v]};
5 SetAU ← SetAU ∪ {new autonomous subgraphs of GG[q,v]};
6 foreach autonomous subgraph GG[a,b] in SetAU do
7  |  Distill(GG[a,b], DSGG, a)
8 end
9 End DownDistillation
```

Figure 5.6: Pseudo-code of the DownDistillation procedure

The function $SameOrientedPath(p, q, GG)$ is true iff there is at least a directed path dp in GG such that p and q belong to dp.
$Visited$ is a function allowing to mark nodes as visited or unvisited.

5.3.2 Illustration of the algorithm

We propose to illustrate the execution of the **DistillFlow algorithm** on the workflow depicted in Figure 5.8(a). We can see that it potentially contains several anti-patterns. Indeed, it duplicates processors many times: #3, #4, #9, #10,

```
1  Distill(IN GG: graph; IN/OUT DSGG: graph; IN x: node)
2  v ← sink(GG);
3  ListRed ← set of reduction nodes of GG;
4  SetAU ← set of autonomous subgraphs of GG;
5  Visited(GG) ← false /* set all the nodes of GG unvisited */
6  foreach successor p of x in GG do
       /* search for copies of p */
7      if Visited(p) ← false then
8          p1 ← p; flagp ← true;
9          while flagp do
               /* flagp allows to consider all the unvisited descendant of p1 if necessary */
10             Distilled ← false /* Distilled says if some transformation on p1 has been done */
11             foreach successor q of x in GG, such that q ≠ p1 do
                   /* successors of x different from p1 are potentially copies of p1 */
12                 q1 ← q; flagq ← true;
13                 while flagq do
                       /* flagq allows to consider all the unvisited descendant of q1 if
                          necessary */
14                     if Visited(q1)=false and SameOrientedPath(p1,q1, GG)=false then
15                         if OKTransformation(p1,q1,GG)=true then
                               /* q1 is a copy of p1 in some anti-pattern and transformation
                                  can be performed */
16                             transformation on DSGG, replacing q1 by mergeq;
17                             flagq ← false; distilled ← true; /* loop on q is stopped */
18                         else
                               /* no transformation has been done on p1 and q1 */
19                             if outDegree(q1) ≠ 1 then
20                                 if there exists a single autonomous subgraph GG[q1, y] in SetAU
                                   then
21                                     q1 ← y; /* the loop on q1 is continued with the sink
                                          of the unique autonomous subgraph */
22                                 else
                                       /* there is no autonomous subgraph GG[q1, y] in
                                          SetAU or more than one */
23                                     if q1 is a reduction node in Listred then
                                           /* search for anti-patterns from reduction node
                                              q1 */
24                                         DownDistillation(GG[q1, v], DSGG, q1, SetAU,
                                           ListRed);
25                                         Visited(GG[q1, v]) ← false;
26                                         if outdegree(q1) > 1 then flagq ← false;;
27                                     else
28                                         flagq ← false;
                                           /* q1 is not a reduction node or there is no
                                              autonomous subgraph GG[q1, y] in SetAU the
                                              loop on q is stopped */
29                                     end
30                                 end
31                                 q1 ← the successor of q1 /* outDegree(q1) = 1 */
32                             end
33                         end
34                         flagq ← false;
35                     end
36                 end
37             end
38             /* while loop on flagp continues*/
39             if distilled then
                   /* if p1 has been merged with some other node then search for
                      anti-patterns from p1 */
40                 DownDistillation(GG[p1, v], DSGG, p1, SetAU, ListRed);
41                 Visited(x, mergeq) ← true; /* set all the nodes on all paths from x to
                      mergeq as visited */
42             else
                   /* p1 has not been merged */
43                 if outDegree(p1) ≠ 1 then
44                     if there exists a single autonomous GG[p1, y] in SetAU then
45                         p1 ← y;
46                     else
47                         flagp ← false;
48                     end
49                 else
50                     let p1 ← the successor of p1 /* outDegree(p1) = 1 */
51                 end
52             end
53         end
54     end
55 end
56 EndDistill
```

Figure 5.7: Pseudo-code of the Distill procedure

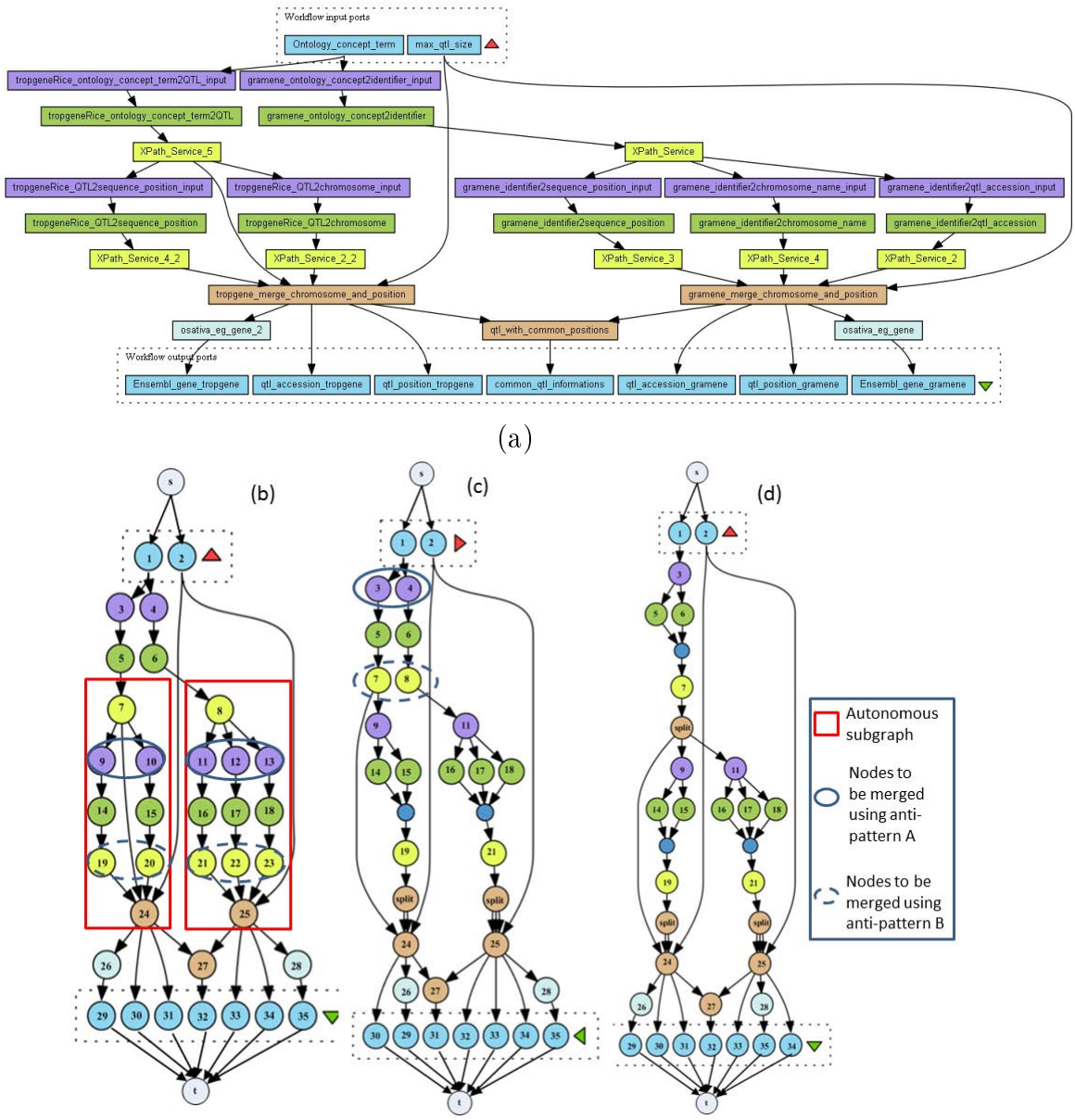


Figure 5.8: Example of transformation of one workflow from myExperiment. (a) Initial Taverna workflow with information on processors; (b) Graph G representing the workflow; (c) Graph DSG obtained after distilling the two autonomous subgraphs; (d) Final distilled workflow obtained by Refactoring.

#11, #12, #13 all perform the same operation, and so do #7, #8, #19, #20, #21, #22, #23. The graph G representing the Taverna workflow is shown in Figure 5.8 (b).

At line 3 of the algorithm, autonomous subgraphs $G[7, 24]$ and $G[8, 25]$ are identified in G. At the first iteration of line 5, the procedure *Distill* is called with $G[7, 24]$ and node #7. During this recursive call, first nodes #9 and #10 are merged according to the transformation of anti-pattern (A), and then nodes #19 and #20, according to transformation of anti-pattern (B). At the second iteration of line 5, *Distill* is called with $G[8, 25]$ and node #8. During this recursive call, nodes #11, #12 and #13 are first merged (anti-pattern (A)), and then nodes #21, #22 and #23 (anti-pattern (B)). At line 7, *Distill* is called with $G[s, t]$ and s. A first recursive call with $G[2, t]$ and node #2 (successor of s that is a reduction node) does not change anything. Recursive calls starting with $G[1, t]$ and node #1 (successor of s that is a reduction node) successively merge nodes #3 and #4 (anti-pattern (A)), and then nodes #7 and #8 (anti-pattern (B), Figure 5.8 (c)). Subsequent calls of *Distill* with $G[24, t]$ and node #24, or with $G[25, t]$ and node #25 do not imply any transformation. Note that nodes #9 and #11 are not merged since $OKTransformation(9, 11, GG)$ is false (such a merge would have introduced a new reduction node, this point is discussed in the next section). Figure 5.8 (d) shows the final workflow where almost all the anti-patterns have been removed.

5.4 Experimental Study

We have implemented DistillFlow into a tool that is presented in more detail in Appendix A.

5.4.1 Anti-patterns in workflow sets

In our study, we have applied the refactoring approach on two workflow sets: the public workflows from myExperiments and the private workflows of the BioVel project (`www.biovel.eu`). BioVel is a consortium of fifteen partners from nine countries which aims at developing a virtual e-laboratory to facilitate research on biodiversity. BioVel promotes workflow sharing and aims at providing a library of workflows in the domain of biodiversity data analysis. Access to the repository

to contributors, however, is restricted and controlled. Because of the restricted access and the focus on a specific domain of these workflows, they are broadly expected to be curated and thus of higher quality than the general myExperiment population.

For each workflow set, the total number of workflows, the number of workflows having at least one anti-pattern (of kind (A) or (B)) are provided in Table 5.1. Note that it is possible that the same workflow contains the two kinds of anti-pattern.

Table 5.1: Initial number of anti-patterns in workflow sets

wf set	# wf	# wf $\geq$ 1 anti-pattern	# wf $\geq$ 1 anti-pattern (A)	# wf $\geq$ 1 anti-pattern (B)
myExperiment	1,454	374 (25.7 %)	80 (5.5 %)	359 (94.5%)
BioVel	71	29 (40.8 %)	0	29 (100%)

Interestingly, 25.7% of the workflows of the myExperiment set contains at least one anti-pattern. Although anti-pattern A appears in only 5.5% of the total, it is particularly costly because it involves multiple executions of the same processor with the exact same input, therefore being able to remove it would be particularly beneficial. The prevalence of pattern B suggests that workflow designers may not know the list processing properties of Taverna (or functional languages).

As for the BioVel private workflows, 40.8% include at least one anti-pattern, all of kind B and thus none contains any kind A. Additionally, other experiments allowed us to observe that a workflow from BioVel contains, on average, fewer anti-patterns than, on average, a workflow from myExperiment.

5.4.2 Results obtained by DistillFlow

Table 5.2 provides the results obtained by DistillFlow in the two workflow sets: the number of workflows in which there is no remaining anti-patterns after applying the DistillFlow procedure, the number of workflows in which at least one anti-pattern has been removed.

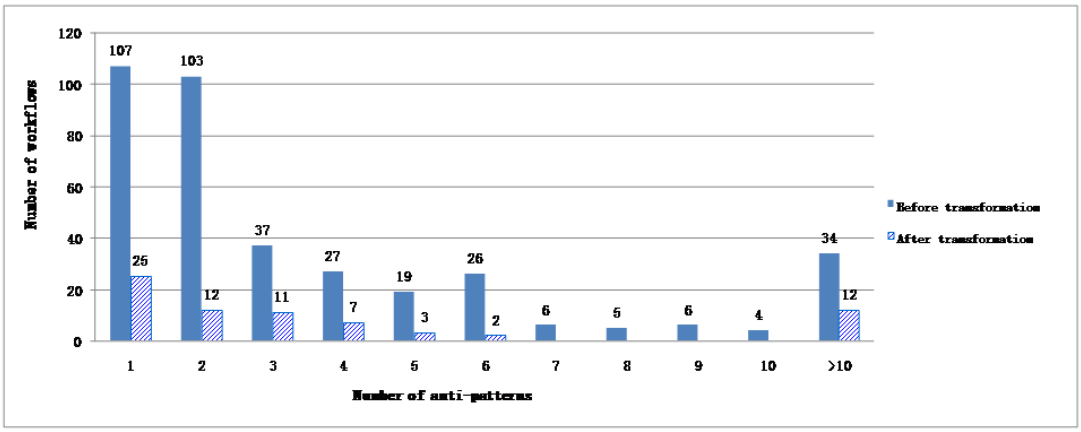


Figure 5.9: Distribution of number of anti-patterns among workflows in myExperiment, before and after applying DistillFlow.

Table 5.2: Results obtained by DistillFlow in the two workflow sets

wf set	# wf without any anti-pattern	# wf with at least one anti-pattern removed
myExperiment	302 (80.7%)	367 (98.1 %)
BioVel	24 (82.7%)	29 (100%)

myExperiment data set. In the set from myExperiment, DistillFlow is able to remove all the anti-patterns in 80.7% of the cases and at least one anti-pattern in 98% of the cases. 72 workflows are not completely free of anti-patterns after the DistillFlow process. However, the majority of these workflows has only one or two remaining patterns as indicated in Figure 5.9. More generally, Figure 5.9 shows that the number of remaining anti-patterns is low compared to the number of anti-patterns in original versions of workflows. Interestingly, additional experiments showed that on average three copies of processors are removed per workflow and this number is even particularly high for some workflows (up to 31).

Biovel data set. In the BioVel data set, DistillFlow is able to remove all the anti-patterns in 82.7% of the cases and at least one anti-pattern in all the workflows (100 %). Only five (particularly big) workflows have remaining anti-patterns. All of them have actually one remaining anti-pattern, as indicated in Figure 5.10. Additional experiments allowed us to state that on this corpus, DistillFlow removes one node per workflow on average, compared to three in myExperiment.

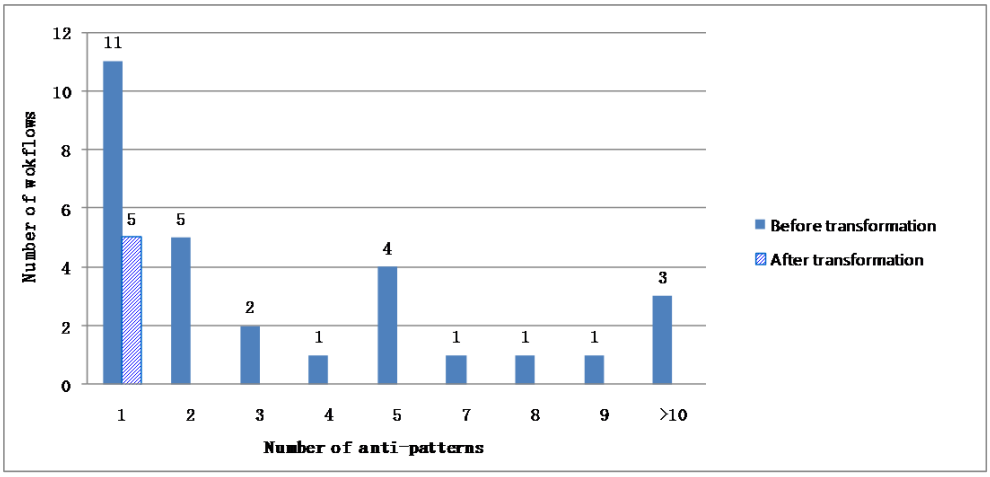


Figure 5.10: Distribution of number of anti-patterns among workflows in BioVel, before and after applying DistillFlow (NB: no workflow of this set has 6 anti-patterns).

In very large workflows of BioVel (these are as large as the largest workflows in myExperiment), up to 15 nodes are removed, compared to 31 in myExperiment. In conclusion, the additional curation steps that occur in the BioVel community clearly make the produced workflows being of better quality; however some of these workflows could still benefit from our distilling approach.

5.5 Discussion

In this section, we discuss several points related to our approach: we provide additional examples to underline the fact that the distilled structures are less intricate (5.5.1); we discuss the impact of our refactoring approach on the SP feature of the workflow structures (5.5.2); we then propose several other kinds of (anti-)patterns which may be directly the cause of non-SP structure (5.5.3); we finally discuss the place of the refactoring approach in the context of provenance-equivalent transformations.

5.5.1 Simpler structures

When all the anti-patterns can be removed by DistillFlow, the resulting workflow structures are particularly simpler, as illustrated in examples provided all along the paper, including the two use cases (Figures 5.1, 5.2). Figures 5.11 and 5.12 provide two additional examples. In Figure 5.11, we have highlighted the

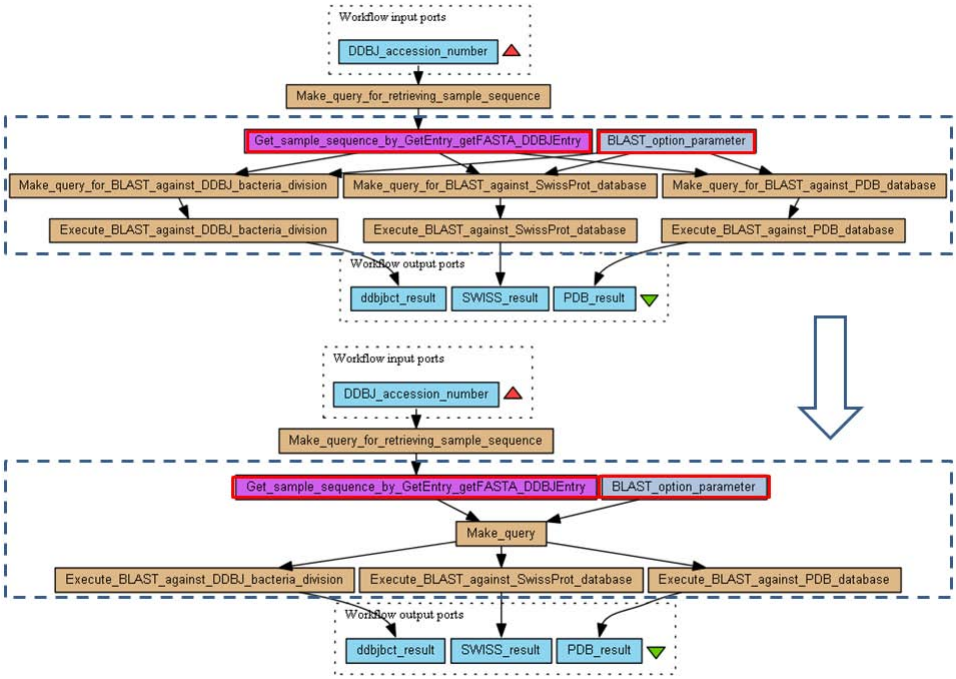


Figure 5.11: Example where the rewritten workflow becomes SP (original workflow on the top and rewritten workflow on the bottom).

rewritten subgraph that is particularly simpler compared to the same fragment of the workflow in the original setting. In Figure 5.12, the global structure is also simpler. Processors have been numbered so that the relationship between the two workflows (before and after the refactoring process) can be seen: in the original workflow p_i denotes the ith occurrence of processor p and in the rewritten workflow, $p_i - ... - pj$ denotes the node resulting of the merging of occurrences $p_i - ... - pj$. For example, f_1, f_2, f_3, f_4, f_5, f_6 are all occurrences of the same processor which are replaced by one occurrence in the rewritten workflow (noted $f_1 - f_2 - f_3 - f_4 - f_5 - f_6$ in the rewritten workflow). As a result of the refactoring process on the workflow of Figure 5.12, three SPLIT processors have been introduced while 18 unnecessary duplications of processors have been removed.

5.5.2 SP structures

As explained in the previous sections, DistillFlow acts carefully on the workflow structures, by removing anti-patterns (A) and (B) while never introducing new intricate structures as non-SP structures may be. We will discuss now two situations. In the first one, we describe situations in which the refactoring al-

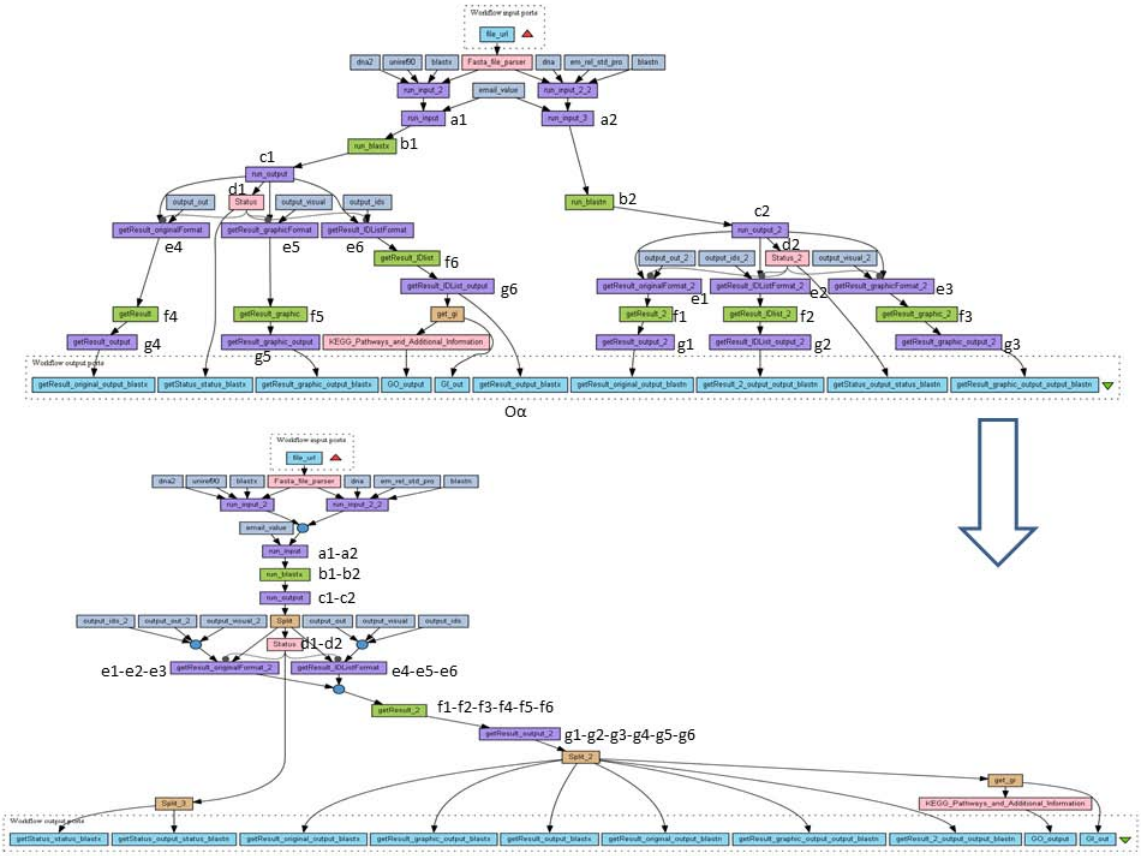


Figure 5.12: Example of transformation obtained using DistillFlow (original workflow on the top and rewritten workflow on the bottom).

gorithm "naturally" removed reduction vertices. In the second case, we propose explanations on the cases where the refactoring cannot be done since it would add reduction vertices.

When refactoring removes reduction vertices. Removing anti-patterns may actually automatically transform a non-SP structure into an SP structure as illustrated in Figure 5.11 in which the original workflow has two reduction nodes underlined in the figure (namely, $Get_sample_sequence_by_GetEntry_getFASTA_DDBJEntry$ and $BLAST_option_parameter$). While these nodes have several input/output links in the original setting they have (at most) one input link and one output link in the transformed version and they are not reduction nodes anymore.

More generally, in the myExperiment corpus, a total of 15 workflows had a non-SP structure before applying the refactoring algorithm and have an SP structure after.

Let us now try to provide an intuition on some situations where refactoring naturally removes reduction vertices. Let us consider another example of non-SP workflow in which two forbidden subgraphs are induced by one reduction vertex, which has been discussed in section 4.3. We claim that merging the successors of a reduction vertex may naturally remove this reduction vertex. Figure 5.13 (i) is an example of such a situation. In the example, we know that processors "$XPath_From_Text0$" and "$XPath_From_Text$" have the exact same code (so that they can be merged). Figure 5.14 shows the specification graphs of workflows in Figure 5.13. The two forbidden subgraphs induced by node #4 in Figure 5.14 (i).(a) are shown respectively in Figure 5.13 (i).(b) and Figure 5.13 (i).(c). After merging nodes #6 and #7 (which are successors of #4), we obtain a new graph shown in Figure 5.14 (ii) (the corresponding original workflow is shown in Figure 5.13 (ii)). Note that the two forbidden subgraphs are no longer present. More precisely, let us notice that in the induced forbidden subgraph of Figure 5.14 (i).(b), when the nodes #6 and #7 are merged, then the node #4 has only one input and one output (one series reduction operation can thus be applied). At the same time, the forbidden subgraph in Figure 5.14 (i).(c) is naturally eliminated.

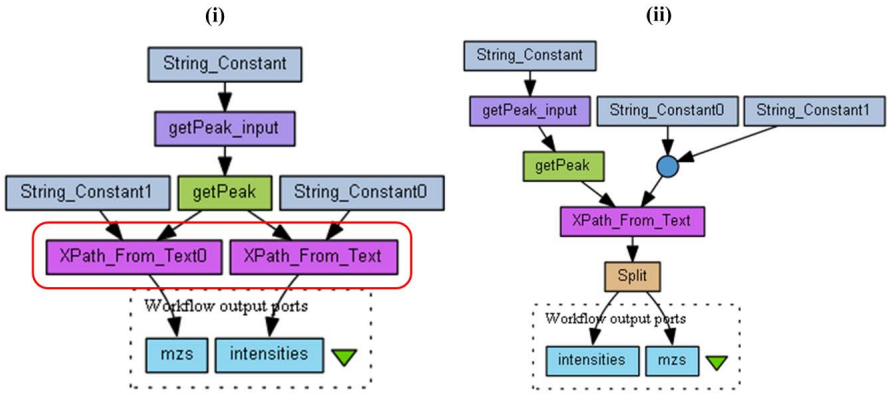


Figure 5.13: Example of workflow (myExperiment 941)

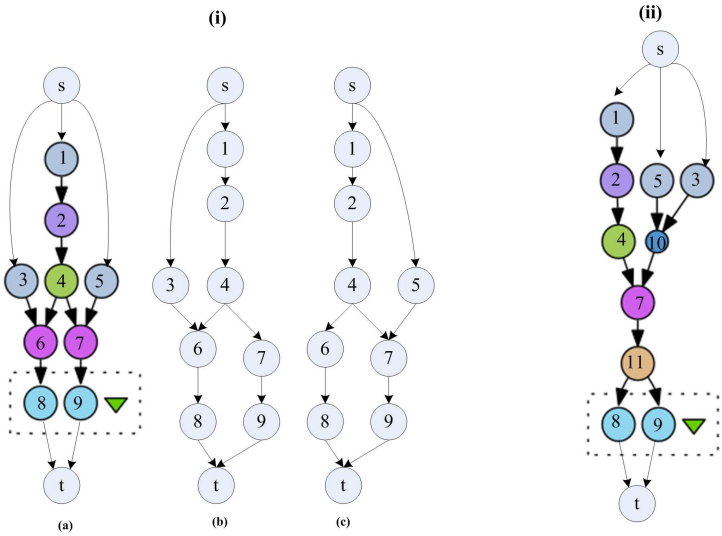


Figure 5.14: Specification graphs of workflows in Figure 5.13 ((i).(b) and (i).(c) are the two forbidden subgraphs induced by node #4 in (i).(a), (ii) is the SP graph obtained when nodes #6 and #7 are merged into one node, called #7)

When removing anti-pattern is not possible. It may also be the case that anti-patterns cannot be removed because removing them would imply merging nodes which would create a new reduction node, making the structure of the transformed workflows more intricate. The number of reduction nodes is actually a commonly used metric to measure how far from an SP structure a structure may be [BKS92]. In that sense, merging such nodes would make the rewritten workflow being further from an SP structure compared to the original workflow structure.

65 workflows from the myExperiment corpus and five from the BioVel data set are involved in such a situation. The illustrative example for DistillFlow of

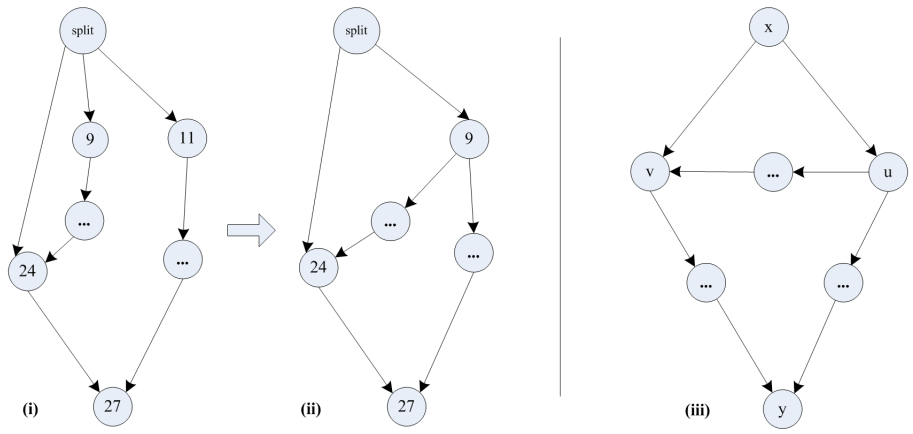


Figure 5.15: (i) Schematic view of a fragment of the workflow of Figure 5.8; (ii) Schematic view of the same fragment but nodes #9 and #11 are now merged; (iii) generic subgraph that is the cause of non-SP structure, u is one reduction node.

Figure 5.8 is one such example: merging nodes #9 and #11 would introduce a new reduction node. In the original graph, node #9 appears in an autonomous subgraph while node #11 does not belong to this autonomous subgraph. If these two nodes were merged, the subgraph formed by all the paths from the SPLIT node to the node # 27 would have the structure of the subgraph responsible for non-SP structures (Figure 5.15 (iii)), and the merged node #9-11 would be the new reduction node. Figure 5.15 (i) shows a schematic view of a fragment of the original graph of Figure 5.8 while Figure 5.15 (ii) shows the structure obtained if nodes #9 and #11 were merged. The graph of Figure 5.15 (ii) is homeomorphic to the generic subgraph represented in Figure 5.15 (iii) which is the cause of non-SP structures (cf. Chapter 2).

A similar situation occurs in the workflow of Figure 5.12 in which nodes #e1-e2-e3 and #e4-e5-e6 cannot be merged by DistillFlow in order to avoid introducing one additional reduction node.

5.5.3 Towards other kinds of (anti-)patterns

Another kind of situation that may occur is when the SP feature is not correlated at all with anti-patterns: the transformed workflows are free of anti-pattern but they still have non-SP structures.

A deep inspection of such workflows reveals that other kinds of patterns may be directly the cause of non-SP structures [CCBF+12]. These patterns have a different nature from the anti-patterns considered so far in this chapter in the sense

that they cannot be removed while keeping the same workflow semantics. One of the most interesting pattern is probably the presence of intermediate processors which are directly linked to the workflow outputs. This situation occurs merely when users want to keep track of intermediate results and "forward" such results to the workflow outputs. We call such intermediate processors *trace nodes* and their outgoing edges linked to the workflow outputs are called *trace links*.

On the total number of workflows in myExperiment, we found 2464 reduction vertices including 853 trace nodes: 34.6% of the reduction vertices have trace links. In the Biovel data set, we found 334 reduction vertices including 60 trace nodes, meaning that 18% reduction vertices have trace links. Trace links are thus important to be considered.

More precisely, several workflows depicted in this chapter have *trace links*. For example, in Figure 5.12 on the top, the link that goes from the processor g_6 directly to the workflow output O_α is a trace link: when the workflow will be executed, the same data (produced by g_6) will be sent both directly to the workflow output O_α and to the downstream part of the workflow. By doing this, the workflow designer may want to keep track of the data produced by g_6. However, as the processor get_gi will consume O_α to produce to its turn some data, these produced data will have O_α in their provenance information. O_α will thus be automatically tracked by the provenance module of Taverna. The trace link from g_6 to O_α is then useless and could be removed. This removal should actually be done very carefully since removing trace links implies removing part of the workflow outputs. As a consequence, the signature of the workflow (the number of outputs) is changed which may have several consequences if the transformed workflow is used as a subworkflow within another bigger workflow that expects the subworkflow to provide given outputs. This kind of transformation should then be done in collaboration with the user so that s/he can estimate the impact of the changes.

5.5.4 Provenance-equivalence

In Chapter 2, we have presented a provenance model and defined the notion of provenance-equivalent runs. As discussed in 3.4 our current provenance model is coarse grain and in particular it does not consider the specificity of Merge and

Split tasks which deal with lists of data items. As a consequence, the refactoring approach cannot be proved to be provenance-preserving in the sense of the definition given in section 3.3 (which does not consider merge and split tasks). However, we have proved in section 5.2 that DistillFlow transforms any workflow into a semantically-equivalent workflow. The extension of our provenance model drawn in section 3.4, together with an extention of the notion of provenance-equivalence in such a new context, should make it possible to state that DistillFlow is provenance-equivalent.

5.6 Related Work

To the best of our knowledge, this is the first attempt at introducing a refactoring approach aiming at reducing workflow redundancy in the scientific workflows setting based on the study of workflow structure.

More research is available from the business workflows community, where several analysis techniques have proposed to discover control-flow errors in workflow designs (see [vdAvHtH+11] for references). More recent work in this community has even focused on data-flow verification [TVdAS09]. However, this work is aimed primarily at detecting access concurrency problems in workflows using temporal logics, making both aims and approach different from ours. Also, it would be hard to transfer those results to the realm of scientific workflows, which are missing the complex control constructs of business workflows, and instead follow a dataflow model (a recent study [MGLRtH11] has shown that scientific workflows involve dataflow patterns that cannot be met in business workflows).

With the increase in popularity of workflow-based science, and bioinformatics in particular, the study of scientific workflow structures is becoming a timely research topic. Classification models have been developed to detect additional patterns in structure, usage and data [RP10]. More high-level patterns, associated to specific cases of use (data curation, analysis) have been identified in Taverna and Wings workflows [GAB+12]. Complementary to this work, graph-based approaches have been considered for automatically combining several analysis steps to help the workflow design process [RMMTS12] while workflow summarization strategies have been developed to tackle workflow complexity [PAK13, BBDH08].

5.7 Summary

In this chapter, we have introduced a new strategy for reducing redundancies in the structure of scientific workflows and have presented an algorithm, DistillFlow, which refactors Taverna workflows in a way that removes explicit redundancy making them possibly easier to use and share. Currently, DistillFlow is able to detect two kinds of *anti-patterns*, and rewrites them as new patterns which better exhibit desirable properties such as maintenance, reuse, and possibly efficiency of resource usage. Then we applied DistillFlow to two workflow collections, the one consisting of myExperiment public workflows, the other including private workflows from the BioVel project. Finally, we have discussed several points related to our approach, in which, additional examples are provided.

Chapter 6

Conclusion and Future work

Contents

6.1 Conclusion 103
6.2 Future Work 105

This chapter concludes our work presented in this dissertation, and finally our future directions are discussed.

6.1 Conclusion

This work proposes two strategies, respectively based on provenance and workflow structure, for rewriting scientific workflows into simpler structures, in order to make scientific workflow easier to (re)use. This conclusion presents a summary of completed contributions.

Note that the first strategy related to rewriting non-SP scientific workflows into SP workflows is introduced in chapter 4 and has been published in eScience 2012 [CBFJ12] and BDA 2012 [CBFC12]; and the second strategy related to rewriting scientific workflows by removing some anti-patterns to reduce redundancy of them is introduced in chapter 5 and has been published in the "BMC Bioinformatics" Journal [CBCG+13] and a poster at NETTAB 2012 [CCBF+12].

Here, we recall the contributions that have been introduced in chapter 1. Broadly stating, the main contributions of this work include the design of (1) a model to present scientific workflows and provenance; (2) SPFlow algorithm; (3) the implementation of the SPFlow system which takes in non-SP Taverna workflows and produces provenance-equivalent SP workflows; (4) the identification and automatic detection of a set of anti-patterns that contribute to the structural workflow complexity; (5) a series of refactoring transformations to replace each anti-pattern by a new semantically-equivalent pattern with less redundancy and simplified structure; (6) a distilling algorithm named DistillFlow that takes in a

workflow and produces a distilled semantically-equivalent workflow; (7) a series of experiments to illustrate our approaches.

In particular, the contributions of this work are as follows:

(1) **Workflow model and provenance model**. Our provenance model is naturally compatible with OPM and uses regular expressions to represent the graph structure of provenance information. This provenance model is currently useful for representing coarse-grained provenance structures. Based on this provenance model, we gave a definition of the notion of provenance-equivalence which can be used to identify whether two workflows have the same meaning.

(2) **Provenance-equivalent SPFlow algorithm**. We reviewed several rewriting strategies for transforming non-SP graphs into SP graphs and proved that they were not provenance-equivalent. Then, we designed a new algorithm, SPFlow, which is a provenance-equivalent approach. It enables us to obtain new provenance-equivalent SP workflows from non-SP workflows.

(3) **Implementation of the SPFlow system**. We have implemented the SPFlow algorithm and developed a tool for transforming any non-SP Taverna workflow into an SP workflow. Current version of SPFlow takes in non-SP Taverna workflow and provides a new SP workflow which can be executed by Taverna workflow system. The tool is currently available from "https://www.lri.fr/ chenj/SPFlow/".

(4) **A set of anti-patterns that contribute to the structural workflow complexity**. We identified and automatically detected a set of anti-patterns by carrying out a series of experiments. Currently, two anti-patterns (cf. Figure 5.3 and Figure 5.4) have been identified.

(5) **A series of refactoring transformations for anti-patterns**. We proposed a series of refactoring transformations to replace each anti-pattern by a new semantically-equivalent pattern with less redundancy and simplified structure.

(6) **A DistillFlow algorithm**. DistillFlow takes in a workflow and produces a distilled semantically-equivalent workflow. The resulting workflow is free or

partly free of anti-patterns and have a more concise and simpler structure, which is closer to SP structure.

(7) **A series of experiments have been provided to illustrate our approaches**. We have illustrated SPFlow by providing an evaluation of our approach on a thousand of the public Taverna workflows. We also have provided an implementation of DistillFlow that we evaluated on both the public Taverna workflows and on a private collection of workflows from BioVel project.

With all these contributions, we are currently able to obtain (1) SP structures for scientific workflows, on which complex workflow operations can easier perform; (2) distilled structures for scientific workflows which are free or partly free of redundancy.

6.2 Future Work

We intend to continue this work in several directions. These directions have already proposed in [CBFJ12] and [CBCG+13]. Here, we recall them and give a discussion on our ongoing work.

The first direction of research focuses on extending our provenance model to support fine-grained provenance, in order to deal with "problematic" dependency discussed in chapter 2. We also intend to extend our provenance model to introduce a restricted form of loops in the specifications making runs having SPFL structures (for Series-Parallel-Fork-Loop) which are structures sharing advantages of SP structures for some operations on graphs [BBD+09].

Based on the extended provenance model, another direction of research deals with generalizing SPFlow to other workflow systems.

The following directions are mainly related to DistillFlow.

The third direction of research deals with generalizing DistillFlow to other workflow systems. In particular, in systems able to exploit multi-core infrastructures or run on Grids or Clouds environments [JCD+13], our distilling approach could be highly beneficial since it pushes the management of multiple activations to system runtime, which can more efficiently parallelize their execution when deployed on a parallel architecture.

The fourth direction includes enriching the distilling approach with new patterns (such as *trace links*) and making it possible to choose whether or not such patterns should be transformed, in an interactive process. In such a framework, users might even have the choice to remove some anti-patterns even if the resulting workflow is non-SP, thus relaxing the SP-constraint. One of the challenges of such an approach will be to provide users with means to estimate the impact of their choices on the workflow structure and its future use.

Instead of considering an automatic procedure, the distilling procedure would be used during the design phase in a semi-automatic way. The refactoring approach would thus be built into the scientific workflow system design environment. It may then be complementary to approaches like [WOvdV09] which help users find and connect tasks following an on-the-fly approach during the design phase or [GGW+09] which supports workflow design by offering an intuitive environment able to convert the users' interactions with data and Web Services into a more conventional workflow specification.

We are also seeking to better understand the reasons why some workflows are not SP. Appendix B provides a preliminary study on the kind of processors which may be more inclined to the reduction nodes and thus to make the workflow structure being not SP.

The longer term goal would then be to propose guidelines for workflow authors to more directly design *distilled* workflows. This work will be achieved in close collaboration with workflow authors and will involve conducting a complete user study to collect their feedback on the distilling approach and possibly resulting in finding again new anti-patterns.

List of Figures

1.1 (a) Taverna workflow; (b) specification graph; (c) run graph 2

2.1 Example of dag. (a) a dag, (b) a labeled graph of (a). 10

2.2 Example of generalized st-multidags. For each graph, the vertices and edges drawn in dashed lines are the vertices and edges that should be added to the initial graph which is drawn in solid lines to get an st-multidag. (a) adding a source; (b) adding a target; (c) adding both a source and a target; (d) the graph is an st-multidag. 14

2.3 Recursive construction of SP graphs: (a) Basic SP graph (BSP), (b) parallel composition, (c) series composition. 17

2.4 The forbidden subgraph for SP-graphs 18

2.5 (a) Series reduction; (b) Parallel reduction 19

2.6 Example of reduction operations applied to G_0. 20

2.7 An example of Taverna workflow ((b) is the specification graph for (a)) 21

3.1 A run of a simple workflow 24

3.2 Two graph illustrating provenance related notions 27

3.3 Data dependency graphs for runs in Figure 3.2. (a) data dependency graph for G_r; (b) data dependency graph for $G_{r'}$ 29

3.4 Example of a run which contains a merge processor (a) initial run ($d_x = [d_1, d_3], d_y = [d_4, d_5]$); (b) an equivalent run of (a). 34

3.5 Data dependency graphs of runs in Figure 3.4 35

3.6 Example of a run which contains a split processor (a) initial run ($d_x = [d_2, d_3]$); (b) an equivalent run of (a). 35

3.7 Data dependency graphs of graphs in Figure 3.6 36

3.8 Combination of a merge processor and a split processor. (a) initial graph G_r ($d_x = [d_3, d_2], d_y = [d_5, d_6]$); (b) an equivalent graph of G_r 37

4.1 (a) Example of simple workflow from Taverna, (b) graph structure of the workflow (non SP); (c) possible SP graph structure; (d) proposal of composite vertices; (e) high-level graph obtained 42

4.2 (a) Out-vertex reduction; (b) In-vertex reduction 45

4.3 Example of reduction operations applied to G_0. 47

4.4 (a) Out-vertex duplication; (b) In-vertex duplication 48

4.5 Resynchronization. (a) forbidden subgraph; (b) up-synchronization; (c) down-synchronization; (d) across-synchronization. 51

4.6 From the (a) Forbidden graph, use of (b) in-Vertex Duplication and (c) out-Vertex Duplication. 52

4.7 Data dependency graphs for runs in Figure 4.6. 52

4.8 Two forbidden subgraphs induced by one reduction vertex. (a) graph with one reduction vertex u; (b) one forbidden subgraph induced by u; (c) another forbidden subgraph induced by u. 54

4.9 Solutions for graphs homeomorphic to Series-non-SP composition (1). (a) a non-SP graph with reduction vertices v_1 and v_2; (b) SP transformation of the forbidden subgraph induced by v_1 in (a); (c) SP transformation of the forbidden subgraph induced by v_2; (d) SP solution for (a). 55

4.10 Solutions for graphs homeomorphic to Series-non-SP composition (2). (a) a non-SP graph with reduction vertices v_1 and v_2; (b) SP transformation of the forbidden subgraph induced by v_1 in (a); (c) SP transformation of the forbidden subgraph induced by v_2; (d) SP solution for (a). For the sake of readability, we give the same name for the duplicated vertices in (d). 56

4.11 Solutions for graphs homeomorphic to Series-non-SP composition (3). (a) a non-SP graph with reduction vertices v_1 and v_2; (b) one SP solution for (a); (d) another SP solution for (a). 57

4.12 Solutions for graphs homeomorphic to Parallel-non-SP composition. 58

4.13 Example of duplicated subgraphs. (a)SPG; (b)G_{red} of SPG; (c)duplicated subgraph from SPG induced by edge $e(s, v)$ in G_{red}. 59

4.14 Graphs obtained by vertex duplication following different reduction sequences (reduction sequence on G_1: a, u, c, x; reduction sequence on G_2: u, x). 60

4.15 G_1 and G_2 are two solutions of rewritting G_0, where G_0 does not contain any autonomous subgraph. (reduction sequence of G_1: a, u, v; reduction sequence of G_2: v, u). 63
4.16 Factorization rule for graphs. 63
4.17 Example of non-SP graph (a) and its maximal reduced graph (b) which is split into three autonomous subgraphs (c). 65
4.18 Example of one execution step of SPFlow where $G[s, x]$ has already been transformed, giving the edge (s, x) in G_{red} after vertex reduction on u and providing $SPG_1[s, x]$ within SPG_1, after vertex duplication of u. The algorithm then considers x as a successor of s in G_{red}. As (x, t) is a separation pair, it calls again $SPFlow$ considering x as source. Vertex y is the successor of x in G_{red} to which a vertex reduction is applied (in G_{red}). Duplication of y in SPG_1 then leads to SPG_2. For the sake of readability, labels are omitted. 66
4.19 Example of iterated forbidden graph (IFG). 67
4.20 Percentage of workflows with a given number of reduction vertices in non-SP structures. 70
4.21 Ratio between the number of vertices in the rewritten graph (G') and the initial graph (G) in function of the size of G. 71
4.22 Architecture of SPFlow 71
4.23 Loading a workflow in SPFlow 72
4.24 Provenance information in SPFlow 73
4.25 Provenance information in (A) non-SP and (B) SP version of the workflow in ZOOM. User views are displayed on the right while full workflows are on the left. 74
4.26 Example of different solutions for unsupported patterns. 76

5.1 Example of workflow (myExperiment 2383) 81
5.2 Example of workflow (myExperiment 804) 82
5.3 Transformation for anti-pattern (A) 84
5.4 Transformation for anti-pattern (B) 86
5.5 Pseudo-code of the DistillFlow algorithm for removing anti-patterns in workflows 88

5.6 Pseudo-code of the DownDistillation procedure 88

5.7 Pseudo-code of the Distill procedure 89

5.8 Example of transformation of one workflow from myExperiment. (a) Initial Taverna workflow with information on processors; (b) Graph G representing the workflow; (c) Graph DSG obtained after distilling the two autonomous subgraphs; (d) Final distilled workflow obtained by Refactoring. 90

5.9 Distribution of number of anti-patterns among workflows in myExperiment, before and after applying DistillFlow. 93

5.10 Distribution of number of anti-patterns among workflows in BioVel, before and after applying DistillFlow (NB: no workflow of this set has 6 anti-patterns). 94

5.11 Example where the rewritten workflow becomes SP (original workflow on the top and rewritten workflow on the bottom). 95

5.12 Example of transformation obtained using DistillFlow (original workflow on the top and rewritten workflow on the bottom). 96

5.13 Example of workflow (myExperiment 941) 98

5.14 Specification graphs of workflows in Figure 5.13 ((i).(b) and (i).(c) are the two forbidden subgraphs induced by node #4 in (i).(a), (ii) is the SP graph obtained when nodes #6 and #7 are merged into one node, called #7) 98

5.15 (i) Schematic view of a fragment of the workflow of Figure 5.8; (ii) Schematic view of the same fragment but nodes #9 and #11 are now merged; (iii) generic subgraph that is the cause of non-SP structure, u is one reduction node. 99

A.1 Architecture of DistillFlow (arrows mean dependencies between the modules) 113

A.2 Loading a workflow in DistillFlow and visualizing the set of anti-patterns detected. The workflow loaded is represented in panel (1) (an outline view is provided by panel (5)), the panel (2) displays metadata of the workflow (number of processors, links, authorship information etc.) while the panel (3) provides the set of anti-patterns detected by DistillFlow. Here, the user has clicked on anti-pattern "13-11-12" in the anti-pattern information panel (3) which has automatically highlighted the corresponding anti-pattern in the workflow (panel (1)). The user has then right-clicked on this pattern on (3), as a consequence DistillFlow proposes to the user to remove it. 115

A.3 Visualizing both initial workflow and distilled workflow. The initial workflow and distilled workflow are respectively represented in panels (a1) and (b1). The panels (a2) and (b2) respectively display metadata of the two versions of workflows (number of processors, links, authorship information etc.) while panel (c) provides the metadata of the two graphs displayed in (a1) and (b1) (number of total nodes, links etc.). The panel (d) is particularly important and provides a table of all the anti-patterns detected in the original workflow. Here, the user has clicked on anti-pattern "13-11-12" in the anti-pattern information panel (d) which has automatically highlighted the corresponding anti-pattern both in the initial workflow (panel (a1)) in which three vertices are involved and in the distilled workflow (panel (b1)) in which the anti-pattern has been removed by the system, merging vertices "13", "11" and "12", resulting in only one vertex, numbered 13. 117

B.1 Distribution of reduction nodes with different processor types . . . 121

B.2 (a) Example of non-SP workflow from myExperiment ("$blastsimplifier$" (p_1) and "$blast_ddbj$" (p_2) are trace nodes and links "$blastsimplifier(p_1) \rightarrow simplified_report(p_3)$" and "$blast_ddbj(p_2) \rightarrow blast_report(p_4)$" are trace links); (b) the specification of (a). If the two trace links in (a) are removed, (b) will not be non-SP graph anymore and the workflow will be an SP workflow (the SP version of the specification is shown in (c)). Recall that this removal should actually be done very carefully since removing trace links implies removing part of the workflow outputs (see section 5.5.3). 124

Appendix A

DistillFlow: refactoring scientific workflows for better (re)use (Tool)

In this appendix, we introduce the java application tool based on the algorithm described in section 5.3 and in [CBCG+13], named DistillFlow, which aims at rewriting complex scientific workflows into new workflows with simpler structure by removing as many anti-patterns as possible. The current version of DistillFlow supports Taverna 2 input workflows.

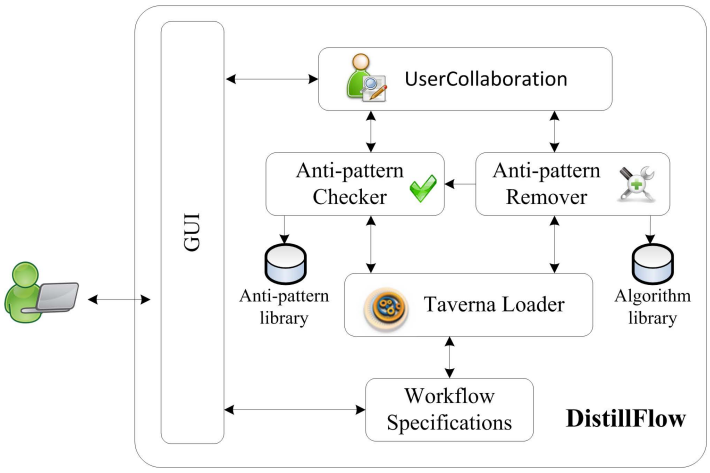


Figure A.1: Architecture of DistillFlow (arrows mean dependencies between the modules)

A.1 DistillFlow architecture

DistillFlow transforms any workflow having "processor redundancy" (where the number of occurrences of a given processor can be reduced without altering the workflow semantics) into a simpler workflow which is free or partly free of anti-patterns. We have implemented the prototype system DistillFlow in Java, whose architecture is shown in Figure A.1.

The process of transforming a workflow is described as follows. The user provides to DistillFlow the specification of the workflow to be considered (such a

specification may have been directly designed by the user or uploaded from myExperiment). The current version of DistillFlow is able to rewrite workflows from Taverna system (other systems are under consideration). The *TavernaLoader* module is thus responsible for loading the workflow into the DistillFlow internal graph structure. Then, the *Anti-pattern Checker* module determines whether or not the workflow taken in contains anti-patterns and provides a report with graph features, including the identification of anti-patterns (if any) and a list of anti-patterns that the system recommends to remove. The user can interact (using the *UserCollaboration* module) with each item of such a list to visualize the corresponding information on the specification graph. If the workflow contains anti-patterns, then the list of anti-patterns selected by the user to be removed (possibly all of them) is sent to the *Anti-pattern Remover* module which removes them. The user can visualize the workflow obtained and decide to stop considering additional anti-patterns. When the user is fine with the workflow obtained (either all possible anti-patterns have been removed or s/he has chosen not to consider some of them), the *TavernaLoader* module produces the rewritten workflow into the Taverna XML format and makes it available for the user.

Users communicate with the system by loading and interacting with original and rewritten workflows. The functionalities of the system are described in more detail below.

A.2 Functionalities of DistillFlow

Our implementation of DistillFlow is able to provide the following features.

Loading Data: Users start using DistillFlow by loading a workflow specification into the system (see Figure A.2). The original picture of the workflow from myExperiment will be displayed by DistillFlow if available (panel (4) in Figure A.2), together with a report on graph features (metadata on the workflow, panel (2) in Figure A.2). The anti-patterns will be determined by Anti-pattern Checker and a list of anti-patterns will be displayed in panel (3) (Figure A.2) of DistillFlow. This list is divided into three groups, including anti-patterns A, anti-patterns B and the anti-patterns which should not be removed (the remove operation of this kind of anti-patterns would create new reduction nodes that we want to avoid as

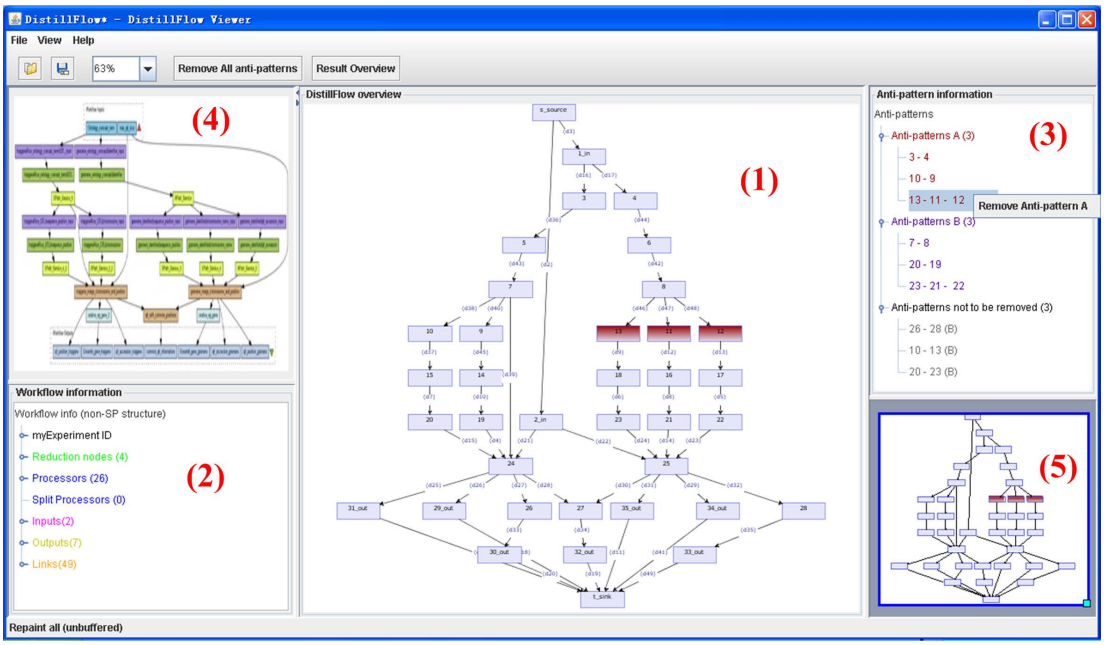


Figure A.2: Loading a workflow in DistillFlow and visualizing the set of anti-patterns detected. The workflow loaded is represented in panel (1) (an outline view is provided by panel (5)), the panel (2) displays metadata of the workflow (number of processors, links, authorship information etc.) while the panel (3) provides the set of anti-patterns detected by DistillFlow. Here, the user has clicked on anti-pattern "13-11-12" in the anti-pattern information panel (3) which has automatically highlighted the corresponding anti-pattern in the workflow (panel (1)). The user has then right-clicked on this pattern on (3), as a consequence DistillFlow proposes to the user to remove it.

explained in Chapter 5).

DistillFlow provides colors to help users easily distinguish different anti-patterns on the workflow graph: "brown" means anti-pattern A, "purple" means anti-pattern B, and "grey" means anti-pattern not to be removed. The same color is systematically used on the workflow (to display the nodes involved in an anti-pattern) and on the text listing the anti-patterns (panel (3) in Figure A.2).

Refactoring the workflow:

a. *Selecting anti-patterns in collaboration with the user:* DistillFlow allows the user to determine which anti-patterns to be removed. By clicking on the anti-patterns information (panel (3) in Figure A.2), DistillFlow will highlight the corresponding processors on the graph (panel (1) in Figure A.2). Then an operation menu (panel (3) in Figure A.2) will be provided by User-Collaboration module to the user to perform the remove operations on the

anti-patterns using Anti-pattern Remover.

Note that this collaboration feature will provide the possibility of extending DistillFlow to support other anti-patterns which can be removed in collaboration with workflow users.

b. *Refactoring once-for-all:* DistillFlow allows to automatically remove all the anti-patterns suggested to be removed by the system. To do so, the user has just to click on the "Remove All anti-patterns" main button (top of Figure A.2). Again using Anti-pattern Remover, DistillFlow transforms any complex workflow with anti-patterns into a simpler workflow which is free or partly free of anti-patterns (see Figure A.3 panel (b1)). Both workflows will be displayed.

   Once the set of anti-patterns to be removed has been selected the user clicks on the "Result overview" button (top of Figure A.2) which automatically opens a new window entitled "Result Overview" and displays the original (Figure A.3 panel (a1)) and distilled workflows (Figure A.3 panel (b1)).

***Visualizing the changes and interacting with the workflows*:** To understand which changes have been done between the initial and distilled workflows, the user can interact with the two workflows. By clicking on a vertex on one graph (initial or transformed graph), DistillFlow will highlight the "corresponding" processors in the other graph (it shows the correspondence between a set of occurrences of a given processor p in the original graph and a set of occurrences of the processor p in the (possibly partly) distilled workflow). With this functionality, the user can make a comparison between the two graphs to see the difference between the initial workflow and the rewritten workflow. A detailed report of all the anti-patterns is also displayed (panel (d) in Figure A.3). By clicking on the items on the anti-pattern information panel, the corresponding processors will be highlighted not only on the initial workflow but also on the rewritten workflow. As an example in Figure A.3, the user has clicked on anti-pattern "11-12-13" which has been removed in the distilled workflow and can be displayed in both workflows (more information is provided on the caption of Figure A.3).

By using such a functionality the user may choose to run again DistillFlow by considering a new list of anti-patterns which may provide a workflow which seems

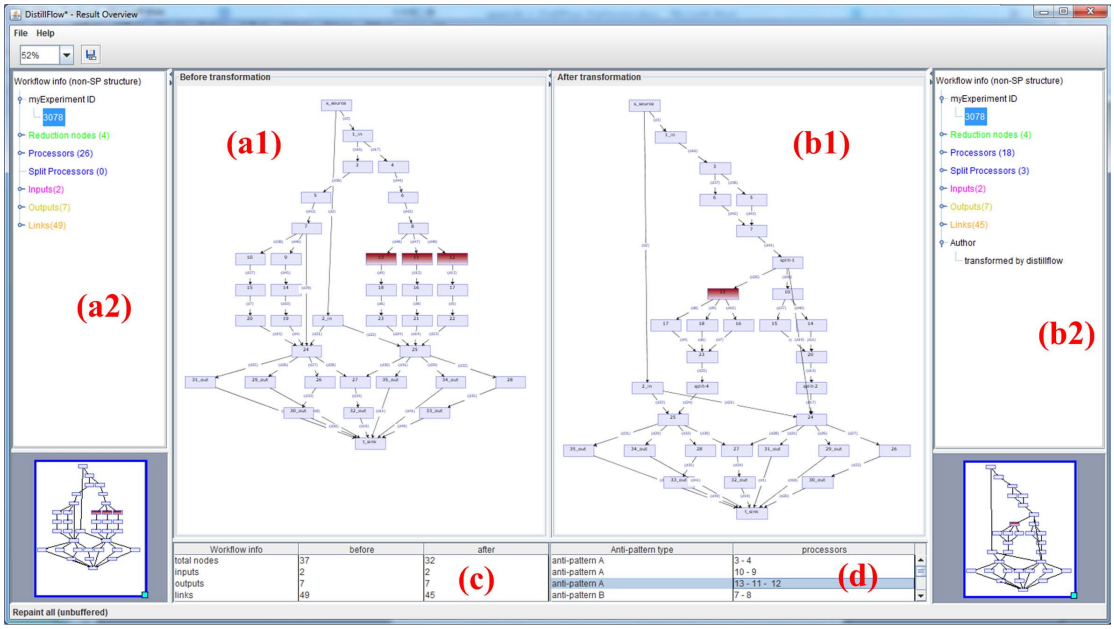


Figure A.3: Visualizing both initial workflow and distilled workflow. The initial workflow and distilled workflow are respectively represented in panels (a1) and (b1). The panels (a2) and (b2) respectively display metadata of the two versions of workflows (number of processors, links, authorship information etc.) while panel (c) provides the metadata of the two graphs displayed in (a1) and (b1) (number of total nodes, links etc.). The panel (d) is particularly important and provides a table of all the anti-patterns detected in the original workflow. Here, the user has clicked on anti-pattern "13-11-12" in the anti-pattern information panel (d) which has automatically highlighted the corresponding anti-pattern both in the initial workflow (panel (a1)) in which three vertices are involved and in the distilled workflow (panel (b1)) in which the anti-pattern has been removed by the system, merging vertices "13", "11" and "12", resulting in only one vertex, numbered 13.

to be more suitable to him.

***Running the distilled workflow*:** Any workflow distilled by DistillFlow can be opened and run by Taverna. We can see that the distilled workflow and the original workflow provide the same output, if given the same input.

A.3 Extensibility of DistillFlow

This section introduces the extensibility of DistillFlow. Some points for extending DistillFlow have been considered in the architecture of DistillFlow.

We consider here three points:

(1) ***Extensible libraries:*** DistillFlow is equipped with two libraries: a library of anti-patterns and a library of distilling algorithms. Any anti-pattern should be registered into the anti-pattern library with several features including the workflow systems they can be applied to and the distilling algorithm which can be chosen to deal with them. For example, anti-pattern A is useful for any workflow system, while anti-pattern B is only useful for workflow systems which support list processing.
Furthermore, for different anti-patterns, the distilling algorithms for detecting or removing all the anti-patterns may also be different. So, DistillFlow currently considers several features of each distilling algorithm, such as for which anti-patterns or workflow system it is suitable. The algorithms should be registered into the algorithm library.
The Anti-pattern Checker and Anti-pattern Remover modules communicate with the two libraries to choose the appropriate anti-patterns to detect and those to distill according to the input workflows. With the help of these libraries, DistillFlow can thus be extended to support other anti-patterns.
For example, trace links can be registered as a new kind of anti-pattern compatible with any workflow system and any distilling algorithm.

(2) ***TavernaLoader:*** This model currently supports Taverna input files, and can be replaced by Galaxy Loader, Kepler Loader, etc. Anti-patterns and distilling algorithms are always in relation with this module. Once a workflow is loaded into DistillFlow, Anti-pattern Checker will first search for

all the anti-patterns related to this workflow in the anti-pattern library and search for the appropriate distilling algorithm in the algorithm library. Then Anti-pattern Checker will identify all the anti-patterns in the workflow according to the anti-patterns found in the library by using the related algorithm.

(3) ***UserCollaboration:*** This module allows users to take part in the distilling process. Because different anti-patterns may require different user operations, DistillFlow provides an interface for extending user operations for different anti-patterns. For example, when removing a trace link, the user may want to replace it by a collaboration link (as it is not a real data link in the workflow, it is not considered in the specification), which is used for querying intermediate data from the internal provenance database of the workflow system.

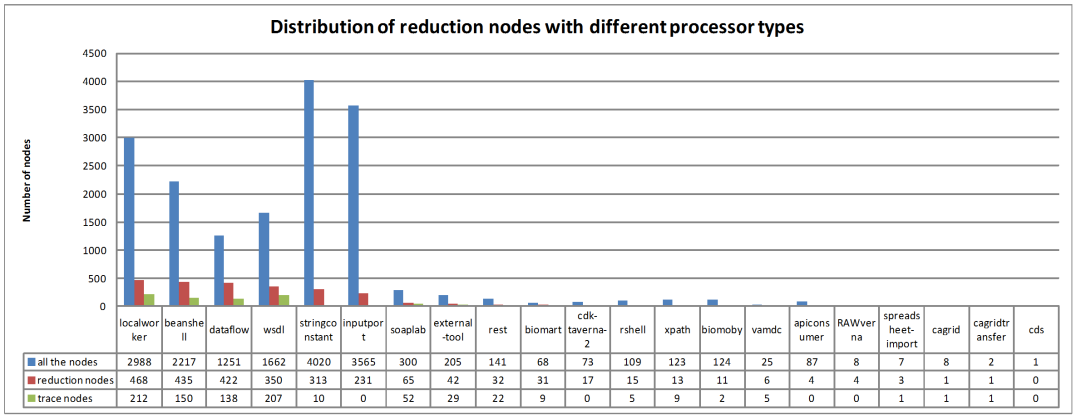
All these features have been considered when DistillFlow has been designed and implemented, which makes DistillFlow able to be extended in our future work.

Appendix B

Why scientific workflows have non-SP structures (Preliminary study)

Determining the reasons why some workflows have non-SP structures may help users to directly design workflows having a structure closer to SP structures. The SPFlow system presented in Chapter 4 may then also be used on less complex, distilled, workflows. The aim of this appendix is to present the results obtained on the study that we have conducted on the set of Taverna workflows available on myExperiment to analyze the reasons why workflows have non-SP structures. A preliminary version of this study has been published in [CCBF+12].

Our study has been conducted on a set of 1,454 distinct workflows extracted from the Taverna workflows available in myExperiment in July 2013. We used SPChecker which was a module included in SPFlow to detect whether workflow graphs were SP. Among the 621 workflows with non-SP structures (42,7%), we have focused on identifying reduction nodes and analyzed the forbidden patterns in which they were involved.



	localworker	beanshell	dataflow	wsdl	stringconstant	inputport	soaplab	external-tool	rest	biomart	cdk-taverna-2	rshell	xpath	biomoby	vamdc	apiconsumer	RAWVerna	spreadsheet-import	cagrid	cagridtransfer	cds
all the nodes	2988	2217	1251	1662	4020	3565	300	205	141	68	73	109	123	124	25	87	8	7	8	2	1
reduction nodes	468	435	422	350	313	231	65	42	32	31	17	15	13	11	6	4	4	3	1	1	0
trace nodes	212	150	138	207	10	0	52	29	22	9	0	5	9	2	5	0	0	1	1	1	0

Figure B.1: Distribution of reduction nodes with different processor types

B.1 On the influence of the kind of processors

a. The first set of experiments aimed at identifying the kinds of processors which may be more likely reduction nodes than others. Figure B.1 provides such results by considering the 21 types of processors in Taverna ("local worker" to "cds"). Six types of processors (namely, "localworker", "beanshell", "dataflow" (subworkflow), "wsdl", "stringconstant", and "inputport") are the types of processors mainly used in Taverna, and represent 92.5% of all the nodes. Such kinds of processors represent 90.1% of the total number of reduction nodes. The distribution of nodes based on the type of processor is thus almost the same when considering reduction nodes only or any node.

   For the first 4 types, to our knowledge, SPFlow is currently the unique solution. For the last two types, namely "inputport" and "stringconstant" (representing 22% of the reduction nodes) we may provide a simpler solution. Indeed, very interestingly, these two types are used as workflow input in Taverna. As an input value can be sent to different processors, some of them usually have more than one output link. As introduced in Chapter 2, we add a single source (an additional node) to make the workflow specification be an st-dag. So, the nodes of input values are thus possible to occur as reduction nodes.

   The non-SP problem caused by such a kind of nodes (workflow inputs) can be solved in a simple way, namely, by duplicating input values to make sure that each input value is sent to one processor. In such a way, the nodes of input values will never be reduction nodes. Furthermore, the nodes of input values are the children of the source, which makes them easier to be detected.

b. ***Non-SP-only processors:*** Our second series of experiments consisted in searching processors that only appear as reduction nodes (such a processor appears in a workflow only as a reduction node). Here again, we studied the types of these processors.

   As a result, most reduction nodes correspond to local processors (processors provided by Taverna to workflow designers) and web services processors. In

particular, among a set of 98 web services, 42 services only appear in non-SP workflows and occur at least once as reduction nodes. More interestingly, nine services appear only as reduction nodes in non-SP workflows. We call them Non-SP-only processors. As for local services, we found one Non-SP-only local processor.

For this point, we need to investigate ways to modify the use of Non-SP-only processors (e.g., changing the processors ports, grouping several consecutive calls of the same processor, designing SP patterns of joint use) so that they are not anymore systematically associated to (and possibly responsible for) non-SP structures.

B.2 Trace links and trace nodes

The third series of experiments has focused on the notion of trace nodes and trace links as introduced in section 5.5.3. Figure B.2 (a) provides an example of non-SP workflow involving two trace links and two corresponding trace nodes. Recall that intuitively a trace link is a link from the output of a processor (vertex) to the final outputs of the workflow that the user may use to keep the trace of some intermediate results produced (although the provenance module is already able to do it). A trace node is the vertex that has a trace link going out of it. According to our definition (definition 2.3.1 in Chapter 2), trace links actually make the workflow be non-SP (if one trace link is removed then the vertex it goes out of may be not a reduction node anymore (cf. Figure B.2)).

Trace nodes are thus very interesting kinds of reduction nodes to look at.

Among a total of 16,984 nodes in the set of non-SP workflows, we found 2,464 reduction nodes including 853 "trace nodes" (representing 34.6% of the reduction nodes) and involved in 423 workflows (representing 68.1% of non-SP workflows). The distribution of trace nodes in different kinds of processors is also shown in Figure B.1. Again, their distribution according to the kind of processor is similar to other kind of node.

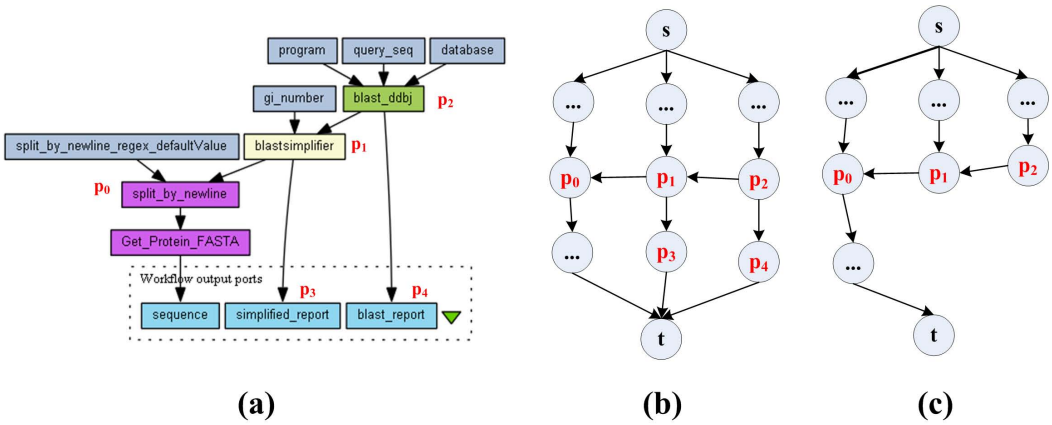


Figure B.2: (a) Example of non-SP workflow from myExperiment ("$blastsimplifier$" (p_1) and "$blast_ddbj$" (p_2) are trace nodes and links "$blastsimplifier(p_1) \rightarrow simplified_report(p_3)$" and "$blast_ddbj(p_2) \rightarrow blast_report(p_4)$" are trace links); (b) the specification of (a). If the two trace links in (a) are removed, (b) will not be non-SP graph anymore and the workflow will be an SP workflow (the SP version of the specification is shown in (c)). Recall that this removal should actually be done very carefully since removing trace links implies removing part of the workflow outputs (see section 5.5.3).

In conclusion, we have identified several reasons why workflows may not have an SP structure. The notion of trace node seems to be promising and from the type of processors point of view, we will study the behavior of some web services further. Following the solutions underlined, we will get distilled workflows in which the number of reduction nodes should importantly be reduced and we hope that a large part of workflows may become SP. In our approach, users do not have to consider structural constraints when they design workflows; our aim is instead to provide them with designing guidelines ensuring that distilled workflows are naturally produced.

Bibliography

[ABJF06] Ilkay ALTINTAS, Oscar BARNEY et Efrat JAEGER-FRANK : Provenance collection support in the kepler scientific workflow system. *In International Provenance and Annotation Workshop (IPAW)*, pages 118–132, 2006. (Cited on page 24.)

[ABML09] Manish Kumar ANAND, Shawn BOWERS, Timothy M. MCPHILLIPS et Bertram LUDÄSCHER : Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. *In 21th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 237–254, 2009. (Cited on pages 23, 24, 25, 32 and 38.)

[ABS+06] Parag AGRAWAL, Omar BENJELLOUN, Anish Das SARMA, Chris HAYWORTH, Shubha U. NABAR, Tomoe SUGIHARA et Jennifer WIDOM : Trio: A system for data, uncertainty, and lineage. *In 29th International Conference on Very Large Data Bases (VLDB)*, pages 1151–1154, 2006. (Cited on page 25.)

[Ana10] Manish Kumar ANAND : *Managing Scientific Workflow Provenance*. Thèse de doctorat, UNIVERSITY OF CALIFORNIA, 2010. (Cited on pages 24, 25 and 32.)

[BBD+09] Zhuowei BAO, Sarah Cohen BOULAKIA, Susan B. DAVIDSON, Anat EYAL et Sanjeev KHANNA : Differencing provenance in scientific workflows. *In 25th International Conference on Data Engineering (ICDE)*, pages 808–819, 2009. (Cited on pages 5, 15, 41, 44 and 105.)

[BBDH08] Olivier BITON, Sarah Cohen BOULAKIA, Susan B. DAVIDSON et Carmem S. HARA : Querying and managing provenance through user views in scientific workflows. *In 24th International Conference on Data Engineering (ICDE)*, pages 1072–1081, 2008. (Cited on pages 5, 27, 41, 43, 73 and 101.)

[BCC+05] Louis Bavoil, Steven P Callahan, Patricia J Crossno, Juliana Freire, Carlos E Scheidegger, Cláudio T Silva et Huy T Vo : Vistrails: Enabling interactive multiple-view visualizations. *In IEEE Visualization 2005 (VIS 05)*, pages 135–142. IEEE, 2005. (Cited on pages 5 and 24.)

[BD08] Roger S. Barga et Luciano A. Digiampietri : Automatic capture and efficient storage of e-science experiment provenance. *Concurrency and Computation: Practice and Experience*, 20(5):419–429, 2008. (Cited on page 24.)

[BDF96] Hans L Bodlaender et Babette De Fluiter : *Parallel algorithms for series parallel graphs.* Springer Berlin Heidelberg, 1996. (Cited on page 20.)

[BDKR09] Olivier Biton, Susan B Davidson, Sanjeev Khanna et Sudeepa Roy : Optimizing user views for workflows. *In Proceedings of the 12th International Conference on Database Theory*, pages 310–323. ACM, 2009. (Cited on pages 43 and 75.)

[BKS92] Wolfgang W. Bein, Jerzy Kamburowski et Matthias F. M. Stallmann : Optimal reductions of two-terminal directed acyclic graphs. *SIAM J. Comput.*, 21(6):1112–1129, 1992. (Cited on pages 5, 10, 13, 16, 44, 45, 49, 53, 60, 65, 70 and 98.)

[Bla] P Blaha : Wien2k, an augmented plane wave plus local orbitals program for calculating crystal properties (karlheinz schwarz, techn. universität wien, austria, 2001). Rapport technique, ISBN 3-9501031-1-2. (Cited on page 41.)

[BLNC05] S Bowers, B Ludaescher, A Ngu et T Critchlow : Structured composition of dataflow and control-flow for reusable and robust scientific workflows. Rapport technique, Lawrence Livermore National Laboratory (LLNL), 2005. (Cited on page 1.)

[BML+06] Shawn Bowers, Timothy M. McPhillips, Bertram Ludäscher, Shirley Cohen et Susan B. Davidson : A

model for user-oriented data provenance in pipelined scientific workflows. *In International Provenance and Annotation Workshop (IPAW)*, pages 133–147, 2006. (Cited on page 32.)

[BML08] Shawn Bowers, Timothy M. McPhillips et Bertram Ludäscher : Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008. (Cited on pages 3, 25 and 37.)

[Bod94] Hans L Bodlaender : Dynamic algorithms for graphs with treewidth 2. *In Graph-Theoretic Concepts in Computer Science*, pages 112–124. Springer, 1994. (Cited on page 20.)

[BRGRM11] Anne Benoit, Paul Renaud-Goud, Yves Robert et Rami G. Melhem : Energy-aware mappings of series-parallel workflows onto chip multiprocessors. *In International Conference on Parallel Processing (ICPP)*, pages 472–481, 2011. (Cited on pages 41 and 44.)

[CBCG+13] Sarah Cohen-Boulakia, Jiuqiang Chen, Carole Goble, Paolo Missier, Alan Williams et Christine Froidevaux : Distilling structure in taverna scientific workflows: A refactoring approach. *BMC Bioinformatics*, 2013. (Cited on pages 4, 5, 7, 103, 105 and 113.)

[CBFC12] Sarah Cohen-Boulakia, Christine Froidevaux et Jiuqiang Chen : Reecriture de workflows scientifiques et provenance. *In Proc. of the 28th Journees de Bases de Donnees Avancees*, 2012. (Cited on pages 6 and 103.)

[CBFJ12] Sarah Cohen-Boulakia, Christine Froidevaux et Chen Jiuqiang : Scientific workflow rewriting while preserving provenance. *In E-Science (e-Science), 2012 IEEE 8th International Conference*, pages 1–9. IEEE, 2012. (Cited on pages 1, 2, 5, 6, 15, 26, 45, 48, 72, 103 and 105.)

[CBL11] Sarah Cohen-Boulakia et Ulf Leser : Search, adapt, and reuse: the future of scientific workflows. *ACM SIGMOD Record*, 40(2):6–16, 2011. (Cited on pages 3, 4 and 43.)

[CCBD06] Shirley Cohen, Sarah Cohen-Boulakia et Susan B. Davidson : Towards a model of provenance and user views in scientific workflows. *In 3rd International Workshop on Data Integration in the Life Sciences (DILS)*, pages 264–279, 2006. (Cited on page 24.)

[CCBF+12] Jiuqiang Chen, Sarah Cohen-Boulakia, Christine Froidevaux, Carole Goble et Alan Williams : Distilling scientific workflow structure. *EMBnet Journal, Proc. of the 12th International Workshop on Network Tools and Applications in Biology, Nettab 2012 (poster)*, 18(B), 2012. (Cited on pages 7, 99, 103 and 121.)

[CCBF13] Jiuqiang Chen, Sarah Cohen-Boulakia et Christine Froidevaux : Spflow: Make your scientific workflows easier to use. *in: Internal report, LRI, University Paris Sud*, (#1564), 2013. (Cited on page 71.)

[CCPP99] Fabio Casati, Stefano Ceri, Stefano Paraboschi et Giuseppe Pozzi : Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999. (Cited on page 15.)

[CFS+06] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Cláudio T. Silva et Huy T. Vo : Vistrails: visualization meets data management. *In ACM SIGMOD Conference*, pages 745–747, 2006. (Cited on pages 41 and 44.)

[CJR08] Adriane Chapman, H. V. Jagadish et Prakash Ramanan : Efficient provenance storage. *In SIGMOD Conference*, pages 993–1006, 2008. (Cited on page 24.)

[CM11] Nadia Cerezo et Johan Montagnat : Scientific workflow reuse through conceptual workflows on the virtual imaging platform. *In*

Proceedings of the 6th workshop on Workflows in support of large-scale science, pages 1–10. ACM, 2011. (Cited on page 2.)

[CTV05] Laura CHITICARIU, Wang Chiew TAN et Gaurav VIJAYVARGIYA : Dbnotes: a post-it system for relational databases based on provenance. *In SIGMOD Conference*, pages 942–944, 2005. (Cited on page 25.)

[CW03] Yingwei CUI et Jennifer WIDOM : Lineage tracing for general data warehouse transformations. *29th International Conference on Very Large Data Bases (VLDB)*, 12(1):41–58, 2003. (Cited on pages 25 and 32.)

[DB99] G. DI BATTISTA : *Graph Drawing: Algorithms for the Visualization of Graphs.* An Alan R. Apt Book. Prentice Hall, 1999. (Cited on page 16.)

[DF08] Susan B DAVIDSON et Juliana FREIRE : Provenance and scientific workflows: challenges and opportunities. *In Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1345–1350. ACM, 2008. (Cited on pages 2, 3 and 15.)

[Duf64] R.J. DUFFIN : *Topology of series-parallel networks.* Numéro 10. J. Math. Anal. Appl., 1964. (Cited on pages 16 and 17.)

[Epp92] David EPPSTEIN : Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1):41–55, 1992. (Cited on page 20.)

[Esc03] Arturo González ESCRIBANO : *Synchronization Architecture in Parallel Programming Models.* Thèse de doctorat, Dept. Informática, University of Valladolid, 2003. (Cited on pages 10, 12, 13, 17, 50 and 53.)

[FCB07] Wenfei FAN, Gao CONG et Philip BOHANNON : Querying xml with update syntax. *In SIGMOD Conference*, pages 293–304, 2007. (Cited on page 32.)

[FLMB96] Lucian Finta, Zhen Liu, Ioannis Milis et Evripidis Bampis : Scheduling uet-uct series-parallel graphs on two processors. *Theor. Comput. Sci.*, 162(2):323–340, 1996. (Cited on page 44.)

[FVWZ02] Ian Foster, Jens Vockler, Michael Wilde et Yong Zhao : Chimera: A virtual data system for representing, querying, and automating data derivation. *In Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 37–46. IEEE, 2002. (Cited on page 1.)

[GAB+12] Daniel Garijo, Pinar Alper, Khalid Belhajjame, Óscar Corcho, Yolanda Gil et Carole A. Goble : Common motifs in scientific workflows: An empirical analysis. *In E-Science (e-Science), 2012 IEEE 8th International Conference*, pages 1–8, 2012. (Cited on page 101.)

[GEvGCP09] Arturo González-Escribano, Arjan J. C. van Gemund et Valentín Cardeñoso-Payo : Performance implications of synchronization structure in parallel programming. *Parallel Computing*, 35(8-9):455–474, 2009. (Cited on pages 41 and 44.)

[GFG+09] Antoon Goderis, Paul Fisher, Andrew Gibson, Franck Tanoh, Katy Wolstencroft, David De Roure et Carole A. Goble : Benchmarking workflow discovery: a case study from bioinformatics. *Concurrency and Computation: Practice and Experience*, 21(16):2052–2069, 2009. (Cited on pages 43 and 69.)

[GGB11] Ahmed Gater, Daniela Grigori et Mokrane Bouzeghoub : A graph-based approach for semantic process model discovery. *In Graph Data Management*, pages 438–462. 2011. (Cited on page 43.)

[GGW+09] Andrew Gibson, Matthew Gamble, Katy Wolstencroft, Tom Oinn, Carole A. Goble, Khalid Belhajjame et Paolo Missier : The data playground: An intuitive workflow specification environment. *Future Generation Comp. Syst.*, 25(4):453–459, 2009. (Cited on page 106.)

[GKT07] Todd J. Green, Gregory Karvounarakis et Val Tannen : Provenance semirings. *In 26th ACM SIGMOD-SIGACT-SIGART Symposium on principles of database system (PODS)*, pages 31–40, 2007. (Cited on page 25.)

[GNT+10] Jeremy Goecks, Anton Nekrutenko, James Taylor, T Galaxy Team *et al.* : Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010. (Cited on page 1.)

[GRD+07] Yolanda Gil, Varun Ratnakar, Ewa Deelman, Gaurang Mehta et Jihie Kim : Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows. *In Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1767. MIT Press, 2007. (Cited on page 1.)

[GSLG05] Antoon Goderis, Ulrike Sattler, Phillip W. Lord et Carole A. Goble : Seven bottlenecks to workflow reuse and repurposing. *In International Semantic Web Conference*, pages 323–337, 2005. (Cited on page 3.)

[HA08] Thomas Heinis et Gustavo Alonso : Efficient lineage tracking for scientific workflows. *In SIGMOD Conference*, pages 1007–1018, 2008. (Cited on page 24.)

[HFYS04] Li Hui-Fang et Fan Yu-Shun : Workflow model analysis based on time constraint petri nets. 15(1):17–26, 2004. (Cited on page 1.)

[HHC99] Chin-Wen Ho, Sun-Yuan Hsieh et Gen-Huey Chen : Parallel decomposition of generalized series-parallel graphs. *J. Inform. Sci. Engineer*, 15(3):407–417, 1999. (Cited on page 20.)

[htt13] http://www.netmba.com/operations/project/pert/ : Pert, 2013. [Online]. (Cited on page 16.)

[HWS+06] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole A. Goble, Matthew R. Pocock, Peter Li et Tom Oinn : Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web-Server-Issue):729–732, 2006. (Cited on pages 1, 2, 3, 4, 20, 24 and 33.)

[HY87] Xin He et Yaacov Yesha : Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation*, 75(1):15–38, 1987. (Cited on page 20.)

[JCD+13] Gideon Juve, Ann L. Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta et Karan Vahi : Characterizing and profiling scientific workflows. *Future Generation Comp. Syst.*, 29(3):682–692, 2013. (Cited on page 105.)

[Koo12] David Allen Koop : *Managing Provenance for Knowledge Discovery and Reuse*. Thèse de doctorat, The University of Utah, 2012. (Cited on page 3.)

[KS07] Christoph Koch et Stefanie Scherzinger : Attribute grammars for scalable query processing on xml streams. *29th International Conference on Very Large Data Bases (VLDB)*, 16(3):317–342, 2007. (Cited on page 32.)

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao et Yang Zhao : Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. (Cited on page 1.)

[LRL+12] Richard Littauer, Karthik Ram, Bertram Ludäscher, William Michener et Rebecca Koskela : Trends in use of scientific workflows: Insights from a public repository and recommendations for best practice. *International Journal of Digital Curation*, 7(2): 92–100, 2012. (Cited on pages 3 and 4.)

[LS88] Andrzej Lingas et Maciej M. Syslo : A polynomial-time algorithm for subgraph isomorphism of two-connected series-parallel graphs. *In International Colloquium on Automata, Languages and Programming (ICALP)*, pages 394–409, 1988. (Cited on page 44.)

[LW98] Kamal Lodaya et Pascal Weil : Series-parallel posets: algebra, automata and languages. *In Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 555–565. Springer, 1998. (Cited on page 44.)

[MAA+05] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun et Frederic Dang Ngoc : Exchanging intensional xml data. *ACM Trans. Database Syst.*, 30(1):1–40, 2005. (Cited on page 32.)

[MBK+08] Archan Misra, Marion Blount, Anastasios Kementsietsidis, Daby M. Sow et Min Wang : Advances and challenges for scalable provenance in stream processing systems. *In International Provenance and Annotation Workshop (IPAW)*, pages 253–265, 2008. (Cited on page 32.)

[MBZ+08] Paolo Missier, Khalid Belhajjame, Jun Zhao, Marco Roos et Carole A. Goble : Data lineage model for taverna workflows with lightweight annotation requirements. *In International Provenance and Annotation Workshop (IPAW)*, pages 17–30, 2008. (Cited on page 32.)

[MBZL09] Timothy M. McPhillips, Shawn Bowers, Daniel Zinn et Bertram Ludäscher : Scientific workflow design for mere mortals. *Future Generation Comp. Syst.*, 25(5):541–551, 2009. (Cited on page 32.)

[MFF+08] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E McGrath, Jim Myers et Patrick Paulson : The open provenance model: An overview. *In Provenance and Annotation of Data and Processes*, pages 323–326. Springer, 2008. (Cited on pages 2, 3 and 24.)

[MGLRtH11] Sara Migliorini, Mauro Gambini, Marcello La Rosa et Arthur HM ter Hofstede : Pattern-based evaluation of scientific workflow management systems. 2011. (Cited on pages 41 and 101.)

[MKK+05] Anirban Mandal, Ken Kennedy, Charles Koelbel, Gabriel Marin, John M. Mellor-Crummey, Bo Liu et S. Lennart Johnsson : Scheduling strategies for mapping application workflows onto the grid. *In The 14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, pages 125–134, 2005. (Cited on pages 41 and 44.)

[MPB10] Paolo Missier, Norman W. Paton et Khalid Belhajjame : Fine-grained and efficient lineage querying of collection-based workflow provenance. *In 13th International Conference on Extending Database Technology (EDBT)*, pages 299–310, 2010. (Cited on pages 22 and 36.)

[Nau95] Valeska Naumann : Measuring the distance to series-parallelity by path expressions. *In Graph-Theoretic Concepts in Computer Science*, pages 269–281. Springer, 1995. (Cited on page 45.)

[OCE+08] Leon J. Osterweil, Lori A. Clarke, Aaron M. Ellison, Rodion M. Podorozhny, Alexander E. Wise, Emery R. Boose et Julian L. Hadley : Experience in using a process language to define scientific workflow and generate dataset provenance. *In 16th ACM sigsoft International Symposium on the Foundations of Software Engineering (SIGSOFT FSE)*, pages 319–329, 2008. (Cited on page 24.)

[OPM] OPM : Open provenance model. http://openprovenance.org/. (Cited on pages 25 and 31.)

[PAK13] Carole Goble Pinar Alper, Khalid Belhajjame et Pinar Karagoz : Small is beautiful: Summarizing scientific workflows using semantic annotations. *In IEEE 2nd International Congress on Big Data*. IEEE, 2013. (Cited on page 101.)

[QF07] Jun Qin et Thomas Fahringer : Advanced data flow support for scientific grid workflow applications. *In Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 42. ACM, 2007. (Cited on pages 32 and 41.)

[RG00] Johannes Ryser et Martin Glinz : Using dependency charts to improve scenario-based testing. *In Proceedings of the 17th international conference on testing computer software (TCS2000)*, 2000. (Cited on page 16.)

[RGS09] David De Roure, Carole A. Goble et Robert Stevens : The design and realisation of the myExperiment virtual research environment for social sharing of workflows. *Future Generation Comp. Syst.*, 25(5):561–567, 2009. (Cited on pages 4 and 72.)

[RMMTS12] Maíra R. Rodrigues, Wagner C. S. Magalhães, Moara Machado et Eduardo Tarazona-Santos : A graph-based approach for designing extensible pipelines. *BMC Bioinformatics*, 13:163, 2012. (Cited on page 101.)

[RP10] Lavanya Ramakrishnan et Beth Plale : A multi-dimensional classification model for scientific workflow characteristics. *In Proceedings of the 1st International Workshop on Workflow Approaches to New Data-centric Science*, page 4. ACM, 2010. (Cited on page 101.)

[SCBL12] Johannes Starlinger, Sarah Cohen-Boulakia et Ulf Leser : (re) use in public scientific workflow repositories. *In Scientific and Statistical Database Management*, pages 361–378. Springer, 2012. (Cited on pages 3, 4, 42 and 83.)

[Sch95] L.A.M Schoenmakers : *A new algorithm for the recognition of series parallel graphs.* Department of Algorithmics and Architecture, 1995. (Cited on page 20.)

[SFC07] Claudio T Silva, Juliana Freire et Steven P Callahan : Provenance for visualizations: Reproducibility and beyond. *Computing in Science and Engineering*, 9(5):82–89, 2007. (Cited on page 24.)

[SGB+01] Laura Silva, Gian Luigi Granato, Alessandro Bressan, Cedric Lacey, Carlton M Baugh, Shaun Cole et Carlos S Frenk : Modelling dust in galactic seds: Application to semi-analytical galaxy formation models. *Astrophysics and Space Science*, 276(2-4):1073–1078, 2001. (Cited on page 41.)

[SQN+06] Felix Schüller, Jun Qin, Farrukh Nadeem, Radu Prodan, Thomas Fahringer et Georg Mayr : Performance, scalability and quality of the meteorological grid workflow meteoag. *In Proceedings of 2nd Austrian Grid Symposium*, pages 21–23, 2006. (Cited on page 41.)

[TDGS07] Ian J Taylor, Ewa Deelman, Dennis B Gannon et Matthew Shields : Workflows for e-science: scientific workflows for grids. *Springer, New York, Secaucus, NJ, USA*, 2007. (Cited on page 9.)

[TVdAS09] Nikola Trčka, Wil MP Van der Aalst et Natalia Sidorova : Data-flow anti-patterns: Discovering data-flow errors in workflows. *In Advanced Information Systems Engineering*, pages 425–439. Springer, 2009. (Cited on page 101.)

[TZF10] Wei Tan, Jia Zhang et Ian T. Foster : Network analysis of scientific workflows: A gateway to reuse. *IEEE Computer*, 43(9): 54–61, 2010. (Cited on page 4.)

[Val78] Jacobo Valdes : Parsing flowcharts and series-parallel graphs. Rapport technique STAN-CS-78-682, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1978. (Cited on pages 10, 11, 12, 13, 16, 17 and 19.)

[vdAvHtH+11] Wil MP van der Aalst, Kees M van Hee, Arthur HM ter Hofstede, Natalia Sidorova, HMW Verbeek, Marc Voorhoeve

et Moe Thandar WYNN : Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011. (Cited on page 101.)

[vG97] Arjan J. C. van GEMUND : The importance of synchronization structure in parallel program optimization. *In International Conference on Supercomputing*, pages 164–171, 1997. (Cited on page 44.)

[VTL82] Jacobo VALDES, Robert Endre TARJAN et Eugene L. LAWLER : The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, 1982. (Cited on pages 19 and 20.)

[Wal07] Norman WALSH : Xproc: An xml pipeline language. *In Technical report, W3c.*, page 13, 2007. (Cited on page 32.)

[Wik13a] WIKIPEDIA : flow, 2013. [http://en.wikipedia.org/wiki/Data_flow_diagram (Online)]. (Cited on page 16.)

[Wik13b] WIKIPEDIA : graph labeling, 2013. [http://en.wikipedia.org/wiki/Graph_labeling (Online)]. (Cited on page 14.)

[WOvdV09] Ingo H. C. WASSINK, Matthijs OOMS et Paul E. van der VET : Designing workflows on the fly using e-bioflow. *In The 7th International Joint Conference on Service Oriented Computing (ICSOC)*, pages 470–484, 2009. (Cited on page 106.)

[ZCHW11] Fan ZHANG, Junwei CAO, Kai HWANG et Cheng WU : Ordinal optimized scheduling of scientific workflows in elastic compute clouds. *In Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 9–17. IEEE, 2011. (Cited on pages 41 and 44.)

[ZGPB+12] Jun ZHAO, José Manuél GÓMEZ-PÉREZ, Khalid BELHAJJAME, Graham KLYNE, Esteban GARCÍA-CUESTA, Aleix GARRIDO, Kristina M. HETTNE, Marco ROOS, David De ROURE et Carole A.

GOBLE : Why workflows break - understanding and combating decay in taverna workflows. *In E-Science (e-Science), 2012 IEEE 8th International Conference*, pages 1–9, 2012. (Cited on page 4.)

[ZGST08] Jun ZHAO, Carole GOBLE, Robert STEVENS et Daniele TURI : Mining taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience*, 20(5):463–472, 2008. (Cited on page 24.)

[ZWF+04] Yong ZHAO, Michael WILDE, Ian FOSTER, Jens VOECKLER, Thomas JORDAN, Elizabeth QUIGG et James DOBSON : Grid middleware services for virtual data discovery, composition, and integration. *In Proceedings of the 2nd workshop on Middleware for grid computing*, pages 57–62. ACM, 2004. (Cited on page 41.)

[ZWF06] Yong ZHAO, Michael WILDE et Ian T. FOSTER : Applying the virtual data provenance model. *In International Provenance and Annotation Workshop (IPAW)*, pages 148–161, 2006. (Cited on page 24.)