Modeling, Analysis, and Optimization of Data-Driven Scientific Workflows

By

SVEN KÖHLER
Dipl.-Inf. (Ilmenau University of Technology, Germany) 2005
M.S. (University of California, Davis) 2011

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

2014

Abstract

Sven Köhler
March 2014
Computer Science

This dissertation presents improvements to the modeling and efficient execution of scientific workflows. Many scientific workflow systems have been developed to solve a specific problem well, but many fail to address needs of a broader group of scientists. While there may never be a system that can satisfy all needs completely, a better balance between diverging design goals can be found. To this end, this work identifies a number of desiderata that occur in the design of a scientific workflow system and discusses to which degree they are addressed in current scientific workflow systems. A selection of systems is presented in detail and strengths and weaknesses with respect to the desiderata are described. From this discussion, beneficial characteristics, properties and implementation details of scientific workflow systems are derived, yielding a proposal for an improved scientific workflow system. Recently, the declarative database language Datalog gained popularity in research and was used in workflow-oriented projects. Therefore, the use of Datalog as (i) a workflow description language and (ii) as a tool for implementing components is investigated. Different and novel approaches to understand, visualize and profile the evaluation of a Datalog program are developed and demonstrated. Finally, new techniques for capturing and employing data and workflow provenance are developed. For example, provenance information is used to understand and debug database queries and workflow execution traces, or to more efficiently resume workflow execution after parameter changes or even system crashes.

Sven Köhler
March 2014
Computer Science

Abstract Page 2 of 2

Provenance is critical for scientists using workflow systems and is therefore studied extensively. This dissertation presents an overview of current research topics in the field of provenance and some methods used to analyze provenance data using Datalog. When Datalog is used as a workflow description language, provenance of data has to be defined and available. Conversely, research in the field of database systems and Datalog can be extended to scientific workflow systems, for example to capture and analyze provenance. A new game-theoretic notion of provenance is presented that yields a detailed visual description of Why/How provenance for facts but also provide answers to Why-Not questions for missing facts in the result. A novel modification of the provenance game construction is sketched that removes dependencies on the active domain from the provenance explanations. Returning to classical workflow systems, some approaches to model and automate scientific problem solving are studied and discussed. This ultimately leads to the definition of a new scientific workflow system that is based on existing concepts that were identified as beneficial earlier but strives to improve on weaknesses identified in the presented case studies. Finally, a new method to improve fault tolerance of a scientific workflow system, which demonstrates all technologies discussed, is presented. Provenance of the workflow execution is analyzed, for example using Datalog, and used to speed up recovery of the workflow execution after a failure.

Contents

List of Figures

Chapter 1

Introduction

With the availability of new technology and automation, science has become increasingly data-driven. New instruments are able to produce increasingly larger amounts of sensor data. For example, modern telescopes have a much higher resolution and work for longer times so that they produce terabytes of data. The Large Synoptic Survey Telescope (LSST) project [ITA+08] generates and processes 30 terabytes of data every night. In the biological sciences, sequencing machines automatically process larger number of samples with a higher precession, resulting in large collections of data. Such data volumes require special handling and are kept at various locations in various formats. The analysis and the use of this data is extremely computationally intensive. Computations are performed not only on one system but increasingly distributed on clusters and clouds. The runtime of scientific programs can be in the order of hours, days or even weeks.

Scientific workflow systems aim to provide an integrated solution for efficiently executing a pipeline of scientific computations and providing the required data for each step. Workflow systems typically provide a *library* of computational functions readily available for use in models. Workflows are created using a *workflow description language*. Most workflow description languages are based on a visual language representing graphs while a few use a purely textual modeling language. The way a workflow model is executed is determined by the *execution semantics* of this model. The combination of a workflow modeling language with its associated execution semantics forms a *model of computation* (MoC) [Pto14].

Note that "first-class" features of a scientific workflow system, e.g., stream parallelism or structured data models, are determined by the MoC and not by the library. Limitations in the MoC are often compensated by adding additional computational functions to the library that users have to add to workflow models, e.g. string parsers or MapReduce actors in Kepler.

Workflow systems are increasingly adopted in research and science. They have been used for such diverse purposes as the automation of monitoring particle physics simulations as well as archiving and post-processing results in the Center for Plasma Edge Simulation project [PLK07], the characterization of microbial organisms and search for links to diseases [HRM+10], or the data management, simulation and monitoring of reservoir uncertainty analysis for petroleum engineers [CAWK08]. Furthermore, there are numerous commercial systems available, e.g., Pipeline Pilot [Pip13] or InforSense Suite [Inf13].

Scientific workflows allow users to repeatedly perform a set of given task over different sets of input data. Thus, they should be easy to create and modify. However, despite their increasing adoption, solid foundations of scientific workflow systems are still lacking. Scientific workflow systems are promising, but further research is required to improve both, workflow usability and the efficiency of workflow executions. The design of workflows should be as simple as possible, but must also provide enough information to allow an efficient execution.

Preferably, data used and produced by a scientific workflow should be well organized. In order to provide value for science, output data should be reproducible and justifiable. To that end, it should be captured on what inputs a given set of output data depends and through which computational steps it was processed. This history of data is referred to as *provenance*. Scientific workflow systems should support the recording and management of provenance.

Provenance information typically forms a graph describing dependencies of events or data. Scientific workflow systems frequently store provenance information in relational databases or specialized graph databases. Thus, simple and efficient tools to create, manage and analyze provenance data are required. Datalog is a declarative logic programming language well suited for querying graphs in general and provenance specifically [DKBL12]. Datalog has well-studied properties and has enjoyed long-term popularity in research. Besides being used as a query language, efforts are made to use Datalog as a workflow description language itself. Such an approach requires to study the provenance of a Datalog program evaluation itself.

Much research has been done over the past years on how to improve modeling and execution strategies. For example, Frank Kühnlenz addresses usability issues of workflows in his dissertation [Küh13]. First, he describes current problems with adopting workflow systems. Then, he proposes a meta-language to describe workflows that can be compiled to different workflow description languages and thus can be executed on multiple systems. Shahaan Ayyub addresses the difficulty of using distributed computing platforms in his dissertation [Ayy13]. He describes another environment in which workflows should be executed efficiently and the difficulties that arise when using contemporary scientific workflow systems. Finally, he develops a system that uses dynamic process networks to better utilize distributed resources and speed up the execution of a workflow. Furthermore, provenance of scientific workflows is still the focus of many research projects [SPG05, MGS11]. For example, Robert Ikeda studied provenance in data-oriented workflows in his dissertation [Ike12]. First, he provides new definitions for provenance and its properties. Then, he describes new methods to capture provenance in a Map-Reduce based environment and how to use that provenance for selective updates. Finally, he developed a system called Panda (for Provenance And Data) that allows selective recomputing, debugging, and drill-down using logical provenance of outputs in data-oriented workflows. Finally, provenance of database query languages and Datalog especially has been studies extensively [KIT10, KG12, HCDN08]. However, practical systems typically use extensions of Datalog where well established provenance approaches are not applicable anymore, e.g., [GBA10].

This thesis presents further research conducted to improve scientific workflow systems and data provenance. This work can be seen as a continuation of Daniel Zinn's dissertation "Modeling and Optimization of Scientific Workflows" [Zin10] and further extends the presented approaches. In particular, the following contributions are presented:

Desiderata (Chapter 2). The author defines a collection of features that scientific workflow systems are commonly expected to provide. This features serve as guidelines for evaluating workflow systems and as goals for further optimizations.

Scientific workflow systems survey (Chapter 3). A survey created by the author summarizes different state-of-the-art workflow systems and evaluates to which degree they address the expectations presented in Chapter 2. Furthermore, current approaches to improve workflow systems are presented.

Datalog (Chapter 4). The declarative programing language Datalog is introduced as a candidate for a scientific workflow description language as well as a tool to analyze workflows and provenance.

Provenance in scientific workflow systems (Chapter 5). The concept of provenance and commonly used notations are introduced. Furthermore, methods to query, process, and analyze provenance are presented. A concrete method is developed to use the declarative language Datalog to analyze provenance information. This research was conducted by Saumen Day with contributions by the author and was published in [DKBL12].

Datalog Debugging and Profiling (Chapter 6). A framework is presented that was developed by the author to understand, debug and profile the evaluation of Datalog programs. The approach employs various program rewriting techniques to capture the provenance and creates a graphical representation of the Datalog evaluation. When Datalog is used as a workflow description language, this closely corresponds to workflow provenance. In the other use case of Datalog as a tool to analyze provenance, it is important to verify the correctness of the program. This work was published in [KLS12].

First-Order provenance games (Chapter 7). Building upon the previous contribution, the author developed a novel system that views a (non-recursive) Datalog$^\neg$ program evaluation as a game. The developed game construction provides provenance information for data in the output of the program but also for absent data. This work was partially published in [KLZ13].

Towards a new scientific workflow system (Chapter 8). A brief study performed by the author provides a comparison of existing scientific workflow systems and proposes approaches for their improvement. The study provides a list of features that have been proposed to address requirements of scientific workflows. Common attributes and workflow system elements are pointed out. In addition, case studies are presented that show how scientific workflow systems are used in practice and how well they perform. First, a case study performed by the author shows how the Process Networks model of computation in the Kepler scientific workflow system is used for monitoring simulations on a supercomputer and identifying. This study points out design challenges and weaknesses in this model of computations. Another case study by the author shows how the collection oriented COMAD model of computation is used to compute growing degree days on streaming data. This study points out some benefits of a structured data model in workflows and was published in [KGC+12]. In addition, a workflow developed by the author shows how to identify motifs in genome data on distributed resources using Kepler Map-Reduce. The workflow is briefly summarized and an abstract was published in [KSFL12]. A more recent case study demonstrates specimen data curation using the collection oriented COMAD model of computation in the Filtered-Push project [WDK+09, DHL+11, MGH+13]. It identifies shortcomings in scalability imposed by the order of data items in the collection structure. With some contributions be the author, Daniel Zinn devised a method to parallelize the execution of a XML like collection oriented workflow using Map-Reduce. A summary of the publication in [ZBKL10] is presented in this work. Based on the technologies used in workflow and impressions from case studies, this dissertation presents a draft of a new workflow system specification that offers improved workflow execution and the benefits of a structured data model.

Fault tolerance for workflow systems using provenance (Chapter 9). An approach to reduce re-execution time of a workflow after a failure was developed by the author and published in [KRZ+11]. This work defines two *smart resume* strategies to efficiently recover a workflow execution using readily available provenance information.

Finally, Chapter 10 summarizes the contributions of this thesis and proposes directions of future research.

Chapter 2

Requirements and Desiderata of Scientific Workflow Systems

The intend of scientific workflow system development is to provide various features to users and developers. The user's goal is to perform scientific computations with minimal manual labor, less expensive and as fast as possible. Based on this usage profile, certain requirements of scientific workflows exist and should be addressed by all systems. T. McPhillips et al.[MBZL09] described requirements that workflows systems should address. The desiderata described here are inspired by this work, but emphasize different aspects. The requirements should be applicable to a wide range of workflow systems and address usability as well as performance.

2.1 Desiderata

In this section, some basic desired properties are defined. For each property, a concrete example is provided to show how it is addressed or violated in workflow systems. This work will provide more workflow features that address the properties presented here.

Automation. Scientific workflow systems should automate the process of executing different programs or scripts necessary for scientific research [LAB+09]. The order, in which processes must be executed, should be determined by the workflow system. Furthermore, the system should conduct all data transfer necessary for computations transparently for the user. The actual execution of the workflow should not require user intervention. For example, the workflow system should store authentication information for long running jobs instead of prompting the user after an authenticated session expires. Furthermore, the system should handle setup and cleanup of the computing environment.

Simplicity. Users should be able to design a workflow, i.e., create a workflow using existing computational functions and combining them into a pipeline, easily. Workflow components often require access to and processing of complex, nested data structures. This can lead to loss of simplicity [MBZL09, ZBML09a].

Reusability. Workflows and their parts should be reusable to create other workflows containing similar steps. In particular, it should be easy to extract workflow parts, to save them persistently, to share them with collaborators and to integrate them into a different model in a few steps [GDR07].

Generality. It should be possible to model a wide range of different problems in different fields of science within the same scientific workflow system [DGST09]. In particular, the data model as well as the computational functions allowed, should not impose any restrictions on the applicability of a scientific workflow system.

Abstraction. The system should provide a sufficient level of abstraction, so that users do not need a deep understanding of programming languages, the execution semantics of the workflow system and the resources the workflow is run on. A user should not need to study the definition of multiple models of computation to be able to create a workflow that processes a variable amount of data provided in a list.

Understandability. Workflows should be readable and understandable by a different user. For example, after sharing a workflow with collaborators, they should be able to identify the scientific problem solved [MBZL09].

Modifiability. Furthermore, a workflow should be modifiable easily. In particular, local changes should not cascade through larger partitions of a workflow. The system should also provide a convenient way for the user to conduct such modifications. A scientist should be able to replace a computational function by another method without an analysis of hidden dataflows in the file system or re-designing data structures [ZBML09a].

Scalability. When multiple computational resources are available, the workflow system should be able to use them in order to speed up workflow executions [DBG+04, FPD+07]. This should not require complex reconfigurations nor changes in the workflow itself. The execution and the optimization of resource usage should be handled transparently by the workflow system. If a large computing cluster is available, the system should be able to execute an available workflow on multiple computing nodes without changes.

Robustness. A workflow system should be fault-tolerant. It should be able to handle faults in individual computational functions without crashing and loosing successfully computed results. Furthermore, a crash of the whole workflow, e.g., due to hardware failures, power outages or other external events, should not require re-executing the whole workflow from the beginning.

Reproducibility. Finally, the computations done by a workflow system should be reproducible. In order to verify the correctness, scientists should be able to analyze from which input or intermediate data results were derived and which methods were used. If a data item of the workflow output is obviously flawed, the user should be able to identify corrupted input data that lead to the error.

2.2 The Problem: Desiderata vs. Reality

Some requirements defined here are generic and also apply to other systems, e.g., database management systems. However, other requirements, e.g., Automation and Reproducibility, are especially targeting scientific workflows.

Now, the question arises to what degree current workflow systems address those requirements. Developers of different systems frequently have their own perception of requirements and no common foundations have been established. In the next chapter, this work will introduce a selection of scientific workflow system, showing that some requirements have not been addressed by many applications. Only a small number of systems provide good Scalability. Robustness is addressed only partially with various specialized methods.

Chapter 3

State-of-the-art Scientific Workflow Systems

This chapter provides a survey of prominent workflow systems and their associated models. The main characteristics of each approach will be presented in order to evaluate to which degree they address the expectations on workflows presented in the previous Chapter 3.

3.1 DAGMan and Job Dependency Graphs

DAGMan is a job scheduling framework, but can also be viewed as a workflow execution engine. Simple workflows can be expressed as directed acyclic graphs (DAGs) of interdependent jobs. The basic modeling elements are depicted in Figure 3.1.

Workflow Description. Nodes represent a single job (or an actor firing). In the pure DAGMan model, there is no further information available which data this job requires, which data it

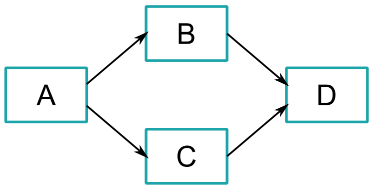


Figure 3.1: A DAGMan "workflow" consisting of four jobs and their dependencies. An edge $X \to Y$ indicates that job Y can only start after job X has finished.

produces or if the job is stateful. Furthermore the system does not know if the same task is invoked again as another job later on, since there is no information about the content of jobs used for scheduling. The workflow designer models edges between nodes in this directed graph based on his understanding of either control flow or data dependencies. The edges themselves do not carry any information on what kind of dependency they represent.

Execution Semantics. During execution, the jobs, whose predecessors all have finished, are started. By default a job's output is available to downstream jobs only after the job completed, i.e., the execution model does not include streaming as a first-class feature[1].

Fault Tolerance. A basic fault tolerance mechanism exists for workflows managed by DAGMan [HC07]. Jobs are scheduled according to a directed graph that represents dependencies between those jobs. DAGMan uses the concept of a *Rescue-DAGs* that contains all job currently running or yet to be executed. If a failure occurs, the workflow execution can be resumed using the Rescue-DAG, only restarting those jobs that were interrupted.

Summary. Because of its nature and tight integration with the job scheduler Condor, DAGMan offers an easy way to run workflows on a distributed system.

DAGMan requires the user to model most of the workflow details. The user has to make sure that data is passed between jobs in the correct way. Data dependencies are hard to understand, since they are not explicitly modeled. This makes it also much harder to modify a workflow.

A major weakness is that DAGMan by itself is completely unaware of data and its shipping. Data dependencies have to be explicitly modeled by the workflow designer and cannot be inferred from a workflow description. Kosar et al. [KL04] presented an approach that should overcome this issue but it still does not provide a fine-grained level of data handling into workflow scheduling.

Finally, the scheduling is based on a whole job as the smallest element. This does not allow any kind of build-in stream or pipeline parallelism. Instance parallelism, where multiple independent jobs can run in parallel depends on the workflow designer's understanding of the exact behavior of jobs.

[1]This does not exclude that a DAGMan provides jobs, that setup streams, but those are then outside the model.

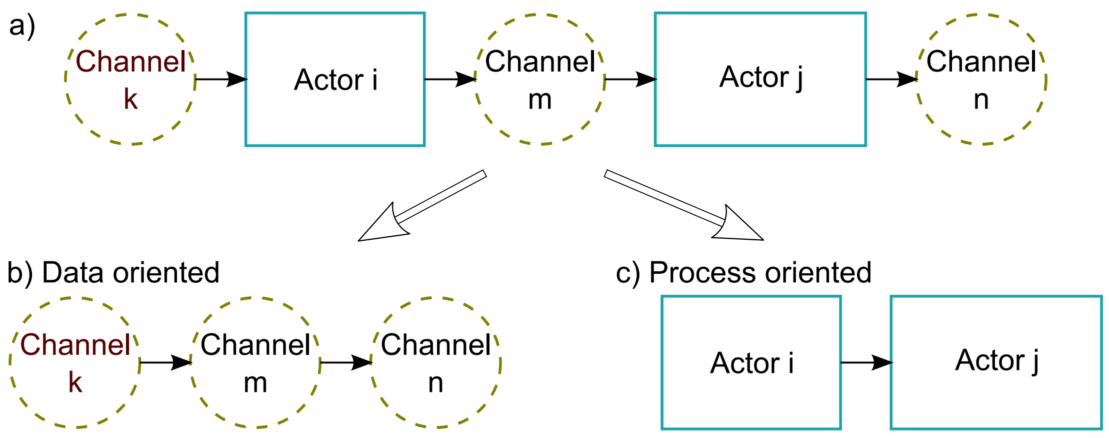


Figure 3.2: Dataflow network with its different representations. a) complete view, b) data-centric view and c) actor-oriented view

3.2 Dataflow Models of Computation in Kepler

Dataflow models represent a family of relatively simple workflow models of computation that are widely used in scientific workflows. The concepts are derived from dataflow programming and hardware design [Den80]. Computational entities (*actors*) perform, e.g., scientific data analysis steps. These actors consume or produce data items (*tokens*) that are sent between actors over uni-directional FIFO queues (*channels*). In general, output tokens are created in response to input tokens. One round of consuming input tokens and producing output tokens is referred to as an actor *invocation*.

Frequently dataflow models are described using a graphical notation of a directed graph as shown in Figure 3.2. A complete view of a dataflow would include the data as input, intermediate results and output as well as the processing entities (actors).

synchronous The following sections describe a selection of models based on dataflow principles in more detail.

3.2.1 Synchronous Dataflow (SDF)

The synchronous dataflow network was proposed by Lee et al. in [LM87a] mainly targeted for hardware-software co-design. A SDF model is a dataflow network that is described graphically. An overview of the model elements is given in Figure 3.3.

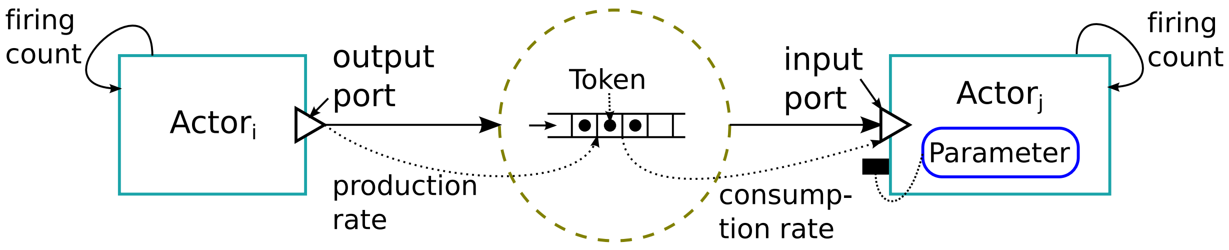


Figure 3.3: SDF model elements overview: Actors can be invoked repeatedly up to the firing count limit. Actors read from and write to channels. Channels behave as FIFO queues. The amount of data tokens produced and consumed by an actor is statically defined. Parameters allow to supply constants to an actor.

Model Description. A SDF model is a directed graph that could have cycles. The nodes of this graph are *actors* that perform computational functions.

Similar to DAGMan, nodes represent processes and data elements are usually not drawn. However, unlike in DAGMan, in dataflow models such as SDF,data tokens are a part of the model and data channels are often drawn together with queues or tokens traveling on the channels. In order to map certain channels to specific arguments of a computational function encapsulated by an actor, the channels are connected to the actor via named *ports*.

Note that these ports are not the data channel's representation. One channel could be fed by one outgoing port from an actor but can connect to many incoming ports. Ports have the semantics of a named FIFO queue that stores incoming tokens.

In addition to the very basic description of actors and their connecting channels, many other properties are defined in the model. Each actor can be annotated with a maximum *firing count* which specified how often this actor can be invoked in total during one workflow execution (run).

A similar firing count is also defined for the whole model and it specifies how often the whole SDF schedule is executed. SDF models are typed, that is, ports only accept tokens with a matching (sub)type.

Finally, the main characteristic of SDF is that the number of tokens produced by an actor through an output port and the number of tokens consumed through an input port are known before the workflow execution. these consumption and production rates are thus port annotations.

Execution Semantics. The execution semantics is based on those properties and will be described in the following: Before the actual execution the model will be verified. This includes mainly checking for incompatible types. The next step is calculating a schedule in which actors will be fired during execution. In order to calculate a schedule, balance equations based on token consumption and token production rates need to be solved. This will make sure that actors are called a different number of times to compensate non matching token rates. The final schedule will contain a number of actor firings that guarantee the consumption of all tokens before a new iteration of the schedule begins. One execution of the schedule is called *round*. This schedule can than be executed repeatedly until either the model firing count or some actors firing counts are reached.

The execution of SDF models heavily depends on the system used. The Kepler scientific workflow engine is an extension of Ptolemy II, which are briefly described here. After the schedule is computed, Kepler iterates through this schedule and executes each actor individually. Independent actors that do not share any connection are executed serially to improve scheduling efficiency.

Summary. The SDF model by itself is simple and is automatically checked for some errors based on typing and token consumption and production rates. However, it requires fixed token consumption and production rates. This restriction does not allow conditional branches or conditional loops. It also can not handle dynamically changing data sizes as appear frequently in scientific workflows unless encapsulated in one data token. Such an encapsulation would also reduce the amount of parallel execution in a possibly parallel implementation of an SDF execution engine. A disadvantage for general application is the serial execution of the schedule. Even if enough tokens are available for multiple invocations, the actor is not invoked concurrently. However, this characteristic is very useful in workflow models to force a fixed execution order, e.g., because of restrictions implied by external programs.

3.2.2 Process networks

Process networks (PN) were proposed by Kahn et al. in [Kah74] as an approach to simplify design of parallel systems. Besides this use case, it also has been used to model scientific workflows in Kepler.

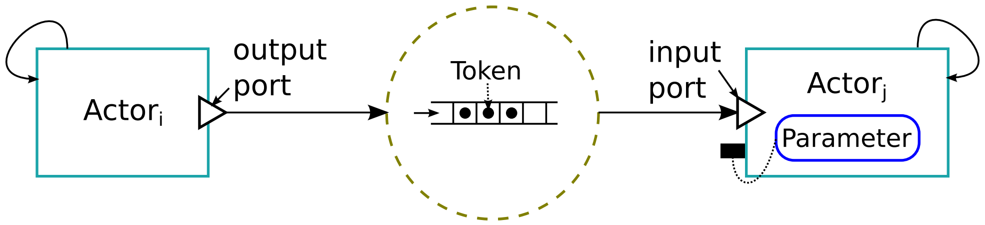


Figure 3.4: PN model elements overview: Actors are invoked multiple times and are only synchronized through channels. Parameters pass constant data to an actor without a port.

Model Description. Process networks are similarly to dataflow networks described as a directed graph. An overview of the model elements is given in Figure 3.4. The nodes of this graph represent processes, which are similar to actors. The directed edges model channels that are used for communication. Processes can consume data from incoming and produce new data on outgoing channels. This Kahn processes [Kah74] are required to be prefix monotonic, i.e., that $F(x') = F(x).\Delta y$ where x is a prefix of x' and where Δx is the remainder. So $y' = F(x).\Delta y = y.\Delta y$, i.e., when more input Δx is revealed, the output y' can only be an extension of $y = F(x)$.

Execution semantics. While the original semantics does not know about firings or iterations practical implementations as the PN director available in Ptolemy II and Kepler use firings to implement the theoretical model. In those PN implementations processes represent actors. An actor will be invoked by the execution engine as long as it does not signal to be done with its task.Each of this invocations is an firing or invocation. During one invocation the actor could read an arbitrary amount of data or possibly no data. In the same invocation the actor also produces and arbitrary amount of data (or no data). To guarantee the monotonicity the actor does not know how much data is available on a channel.

The end of a PN workflow does not need to be signaled directly. It either ends if all actors signaled to be done with all computations or when all actors that did not signal to be done are deadlocked on a read from another actor. This makes the creation of PN very easy and also flexible. However it makes it harder to understand certain characteristics. While an actor could be deadlocked on a read from one channel it is not guaranteed that the data on other channels was read. Therefore the workflows modeled in that way may have data waiting in queues that never get processed, Due to the flexible nature of PN and no further annotations that clarify the

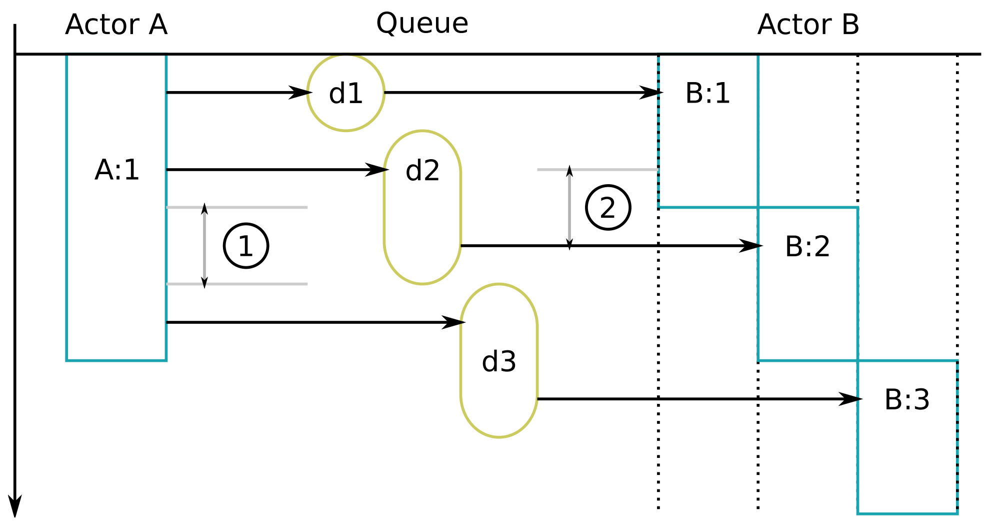


Figure 3.5: Example of suboptimally scheduled PN workflow.

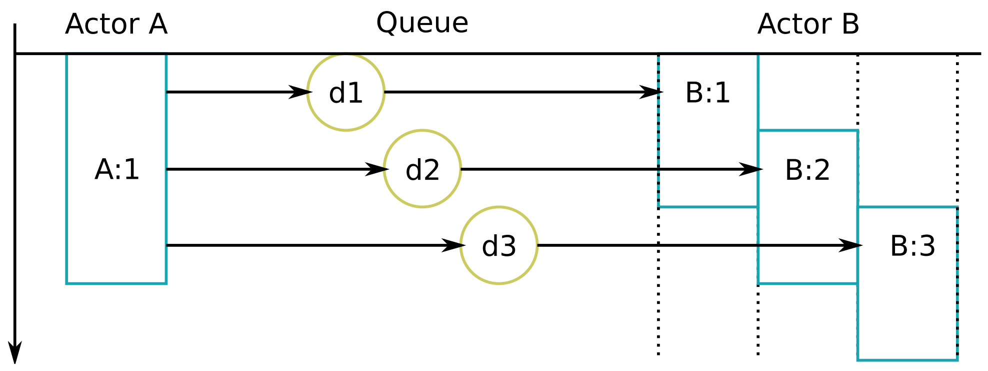


Figure 3.6: Theoretical optimal schedule for executing a stateless actor B that processes only one input per firing.

behavior, the applications for static analysis are very restricted.

Weaknesses. Since PN workflow descriptions do not contain firing rules nor information if an actor is stateful or not, no parallel invocations of one actor is allowed. In the example of Figure 3.5 a stateless actor B could be invoked in parallel for multiple data tokens as shown in Figure 3.6.

Summary. The PN model is very flexible and allows a high level of parallelism. However, the knowledge about actors and their behavior is very limited. This allows no further optimizations of data shipping or instance parallelism. Scientific workflows modeled in PN either require additional actors in order to handle complex data types or complex data has to be encoded into one data token preventing parallelized operations on its components.

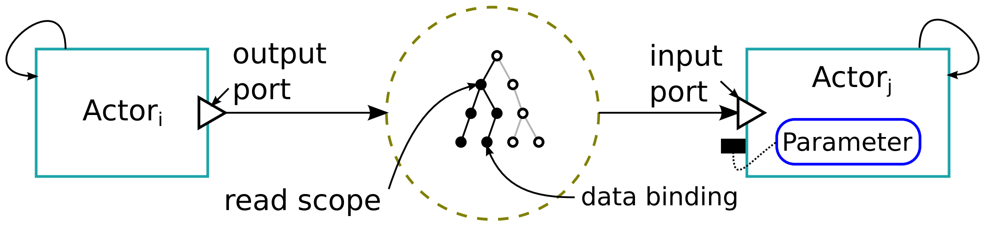


Figure 3.7: COMAD model elements overview: Based on PN, COMAD uses a hierarchical data structure that is streamed over channels. Read scopes mark the sub-tree that is processed by an actor. Data bindings specify which data from the read scope is passed to the actor.

3.2.3 COMAD

Collection Oriented Modeling and Design (COMAD) is a stream-oriented workflow description language in the Kepler scientific workflow system. Its concepts and an application of this workflow language are described by Lei Dou et al. in [DZM+11]. The implementation is based on the PN dataflow network of Ptolemy II but the semantics are modified.

Workflow Description. As the workflow language its implementation is based on, COMAD is a graphical workflow modeling language. The graph structure is usually linear but some extensions allow branching and merging of that graph using special nodes. An overview of the model elements is given in Figure 3.7. Nodes in this graph represent actors and encapsulate the scientific functions. COMAD is designed as a hierarchical modeling language and therefore allows nesting of other COMAD models and even basic data flow network inside an actor. The edges of a workflow graph describe channels on which one data structure is streamed through the actors.

The data transmitted between actors could be viewed as one tree structure, similar to an XML document. One node type in this data tree is a *collection* (ordered list) of other nodes. Another node type is a *data token* that only appears as leaf node. The type of this data token can be one of the complex types available in Kepler but it cannot be a collection or any other type appearing in the data tree. The last tree node type is an *annotation*. Annotations can be associated with collections or data tokens and consist of one label (of type string) and one value of any data type. The data types available in Ptolemy II and Kepler include primitive types such as Boolean, integer and string as well as complex types such as records or arrays. All operations of actors are only allowed to add data to the end of collections. Data removed from the data structure will not be removed entirely but tagged with a special annotation that indicates its deletion.

The workflow language also provides model elements that specify how the data is extracted form and inserted into the stream by actors. Sub-trees of the whole data structure that the actor could operate on are specified by *read scopes* associated with actors. This scope expression is comparable to XPath definitions and may return more than one match. A read scope needs to be absolute, thus starting with a '/'. For the scope expression both child axis '/' and descended axis '//' are allowed.

The read scope only specifies where the actor is allowed to read data from the tree structure. It does not always provide the exact data items used nor does it provide any information where data is written. For this purpose data binding expression are associated with an actor as well. Input data bindings provide a label for the data that will be extracted and the data itself to the actor. Output data bindings define a path within the read scope or a newly created neighbor node in the data tree where data will be stored.

The data binding expressions use the same XPath like language as the read scope but allow more conditions and references to other data binding. The expression language allows specifying constraints on matching nodes of the path just like XPath but with a slightly modified syntax. It is possible to refer to another data binding of the same actor. Output data bindings allow specifying if and when new collections should be created for output values. Both data bindings and read scope also specify the data type that will be read or written by an actor.

In some cases, COMAD allows non linear workflow graphs. A branch in the workflow, i.e., two channels leaving one actor node, results in two copies of the whole data structure that are send over the different branches. All modifications to the data structure on the individual branches are local and do not affect the other branches. Merging of two branches (two incoming channels into one actor) is supported because of restrictions in the model and is handled by a special actor with two input ports. This special actor has annotations that define a dominant and a recessive branch that get merged. Unchanged data will be present in the merged stream once. If differences occur, the dominant's branch changes are merged to the stream first followed by the recessive branches content.

Execution Semantics. The execution of a COMAD model follows the semantics of PN. The data tree structure is split into tokens. Collections are represented by an opening and a closing token. All other elements are represented just by one token. All tokens are streamed through channels. Actors can read tokens from the stream and these elements can be forwarded to the next actor, since changes are made at the end of the collection only. The whole data structure is distributed over the whole workflow and actors could work in parallel on this stream (stream parallelism). During the execution phase each actor keeps track of the stream position it is in and extracts and inserts data according to the data bindings.

Summary. The COMAD workflow language offers some degree of parallel execution of the workflow based on streaming a data structure. Because of the detailed annotations on actors which data is used or produced many optimizations are possible. The data shipping optimization presented in [ZBML09b] could be used to minimize shipping. The expressive scope language allows to arbitrary pick data from a stream and no shims actors are necessary to access nested data. Static analysis allows predicting the resulting data tree and can help to verify correct execution of the workflow.

One major disadvantage is the baroque expression language. It is sometimes not easy to find a good data structure layout to provide an actor with the right combination of data from multiple data bindings. COMAD also does not provide any annotations providing information if an actor keeps an internal state. This makes further optimizations as invoking multiple instances of a stateless actor in parallel impossible.

3.3 Taverna

Taverna [TMG+07] is a widely used scientific workflow system that uses a graphical descriptions of workflows.

Workflow Description. Building on basic dataflow principle, its workflow language supports *processors*, i.e., actors as well as *data links*, i.e., channels, between them. Processors represent computational function either implemented as Java classes or web services. Constant processors just provide basic input values. Data links define the flow of output data of one processor to the input of another processor. In addition, Taverna's workflow language supports *control links* between processors indicating that a processor can start only after another processor has finished its execution. *Ports* connect processors with links and allow multiple input or output values on one processor. Workflows are conceptually similar to processors and also have inputs and outputs. Multiple workflows and processors can be interconnected to form a new workflow.

Data values and ports in Taverna have associated types. The type system supports base types and nested lists of those types. The data itself is XML formatted.

Execution Semantics. Taverna invokes processors in the order defined by dependencies that are implied by data and control link. The workflow system handles the data shipping. If the type of input data does not match the type of a port but differs by one level of hierarchy, Taverna will nest or unnest types automatically. If a port expects a list of type τ and the input is just of type τ then this input will be converted to a single element list $[\tau]$. Unnesting, i.e., transforming a single list of types $[\tau_1, \tau_2]$ into a sequence of multiple values τ_1, τ_2, results in multiple invocations of the processor iterating over all elements of the original list. If types of multiple input ports are unnested, the processor is called with all possible combinations of unnested values in *cross product* style.

Summary. Taverna allows structured data types and it supports implicit nesting and unnesting operations. However, Taverna implements the dot product in a separate actor and not in the workflow description language. The simple nesting and unnesting approach does not support random access to different levels of hierarchical types.

3.4 RestFlow

Restflow [MM10] is a purely textual workflow description language that is partially based on data flow ideas.

Model Description. A workflow is defined using YAML text files, which contain a workflow section. This workflow section contains a list of other entities belonging to this workflow. One entity type is a *node*, which defines a container for a computational function. The actual function

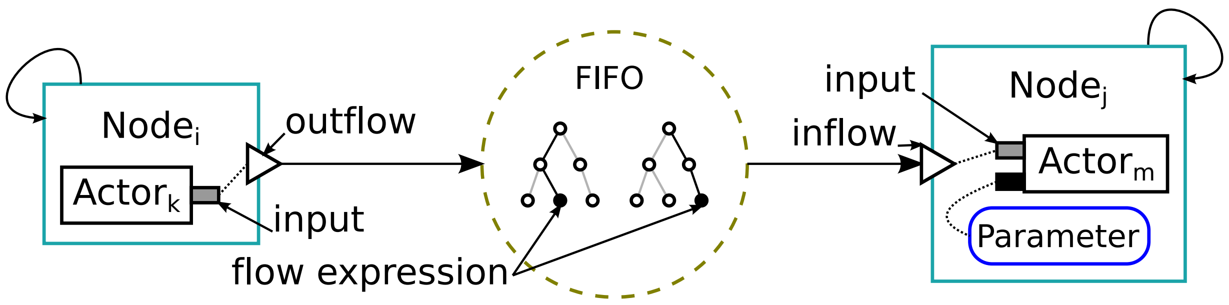


Figure 3.8: Restflow model elements overview: Restflow uses ideas of PN but refines actors. Nodes encapsulate actors and handle data sources. An actor provides the computational function with a fixed signature. Data is published to a folder like structure using flow expressions. Another actor reads this data if its inflow expression, i.e., a subscription URL, matches.

that operates on one set of inputs to produce a set of outputs is described in an *actor*. A node refers to such actors and encapsulates actors written in different languages. The following table summarizes differences of nodes and actors.

Another modeling element is a *director*, which is comparable to the director concept in Ptolemy II. A director defines the execution semantics of this workflow and as Ptolemy II, Restflow provides many different directors to choose from. In contrast to data flow descriptions, this language does not require explicit declarations of channels. Each node defines *inflows*, *outflows* or *parameters*. Inflows describe a location using *flow expressions*, similar to a path, where data could be read and passed on to the actor. Outflows describe where data will be placed. Parameters provide values directly to the actor without a specific location. Data for Parameters can also be specified as lists over which the workflow system will iterate through its execution. The actor itself has *input* names and *output* names declared. The node maps inflows or parameters to those inputs and similarly outputs of an actor to outflows. If one outflow publishes data to the same location where another inflow location reads from (subscribes) then a data channel is implicitly formed. Currently Restflow only supports unique location descriptors.

Restflow also allows nested workflows where another workflow could be enclosed by a node. One workflow could be nested in another workflow multiple times, where each subworkflow is its own instance. For such nested workflows additional node types called Portals are provided to map inflows and outflows. Portals are "one-way gateways" that pass data between workflow and subworkflow. An *InPortal* routes one or more data flows into the subworkflow. Correspondingly,

Node	Actor
Unique A particular node can be employed in only one place in one workflow.	**Reusable** Multiple instances of the same actor can be used by many nodes in the same or different workflows.
Specific Signifies the role played by an actor in a particular workflow.	**General** Is completely unaware of the role played in a particular workflow.
Connected Interacts directly with data flowing through the workflow, exchanging data with upstream and downstream nodes.	**Isolated** Each running instance of an actor exchanges data directly only with the one node that refers to it. It is unaware of other nodes, actors, or the workflow as a whole.
Abstract Independent of how the computational step it represents is performed.	**Concrete** Directly or indirectly defines the implementation of the computational step performed.
Technology-neutral Works independently of the technology used to implement the computation it represents.	**Technology-specific** Depends on specific technologies to implement a computation.
Flow-based Employs data flows to interact with other nodes in the workflow.	**Variable-based** Employs variables to interact with code implementing computation.
Data-source configurable Configurable to accept data via parameters or inflows depending on the role of the node in the workflow.	**Data-source independent** Receives data in a uniform fashion regardless of whether the corresponding node receives data from parameters, inflows, etc.
Pluggable Different actors with the same signature (input and output variable names) can be swapped in easily.	**Node-independent** Can be plugged into any node that provides compatible inputs.
Customizable Parameters and inflows can be used to override all or just a few of the default values provided by an actor.	**Standardizable** Default values for input variables can be used to standardize the use of algorithms or external programs.

Table 3.1: Differences between actors and nodes from [MM10]

an *OutPortal* routes the dataflow from a subworkflow into the surrounding workflow.

Actors could be annotated to be stateful in this workflow language. This means that the output of an actor not only depends on its current inputs but also the previously received input values. A director will fire such actors individually for each data set it receives.

Execution Semantics. Since Resflow provides a number of different directors, the execution semantics can't be described generally. One option provided is the data driven director which to some extend comparable with the data flow approaches described earlier.

During execution Restflow triggers nodes according to the rules provided by the director. Nodes will be triggered if there is sufficient data available on all inflows to fire the actor. Data produced will be published at the specified locations and can in turn trigger new nodes. This means that the director here determines the availability of data directly in contrast to the PN approach which deadlocks on read events. The schedule, in which actors are triggered, has a fixed order - much like SDF. And only one node is triggered at a given time. The director can determine if a workflow has finished executing once it fired all actors and no additional data appeared on any outflow.

If a node has two inflows and there is more than one data set available to fire the actor, data of different inflows will be matched together according to their arrival time (zip or dot product behavior).

This data driven firing allows to implement conditional control flow structures (e.g., if-then-else or while loops) and filtering constructs, which can not be implemented with SDFs static token production and consumption rates. Data sources are a special case, since such nodes do not have any inflows. By default they will step one time for each parameter set assigned to the node. If there are no parameters on a node it will be triggered one time. For stateful data sources the director also has no means to decide when it should not trigger the actor again. In such cases the actor needs to be designed to send a null token explicitly and that actor also has to be annotated with the *endFlowOnNull* property. This annotation is necessary since normal actors could produce null tokens throughout their normal invocations without the intention to end the workflow.

Restflow allows nodes to specify optional data requirements as well as default data. Actors are able to indicate if they require more data on an input during runtime. Another element of the execution semantic is only available within actor code. Through an API an actor could declare to not be ready to receive further data on a given port for one stepping.

Another director available for Restflow workflows is the multithreaded data driven director. It is closely related to the data driven director and shares most of its semantics. The major difference is that actors can be invoked in parallel and will concurrently produce and consume data in an unpredictable order. This allows pipeline parallelism in the workflow where actors work already on new data while nodes subscribed to this actor still work on previously produced output.

Summary. Restflow does not use deadlock detection to end workflows but instead could infer the end of the workflow from the amounts of data produced. This allows distinguishing between real deadlocks and the intended workflow end.

This workflow system makes it easier to integrate external applications such as shell scripts. Reading from external file is also much more integrated since publish and subscribe locations are mapped to file system paths.

Restflow as well as Kepler provide a large selection of different execution semantics and require the user to choose the best fitting one. This requires the user to have a deeper understanding of each execution semantic, which is a limiting factor for user friendliness. There are also some properties of the execution semantics as the blocking of an inflow that can only used from actors source code and are not usable otherwise.

3.5 Vistrails

Vistrails [BCS+05] is a system that is mainly used for visualization but also includes a data flow based workflow system that is used to describe the process of generating visualizations. The dataflow is used to describe the steps how the visualization was created (provenance) and how it could automatically be recreated. It provides a graphical user interface creating and modifying such workflows. The modeling elements are similar to those used in other data flow based languages.

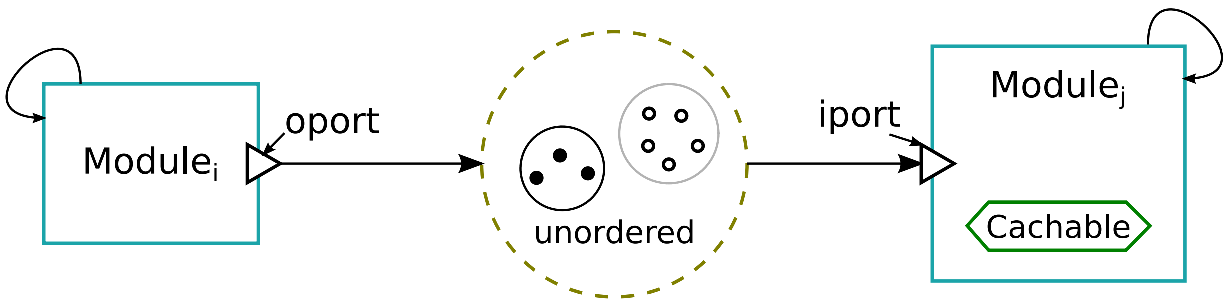


Figure 3.9: Vistrails model elements overview: A *connect element* is used to exchange unordered data tuples between modules. Actors are stateless and their computations can be reused if the *cacheable* annotation is present.

Model Description. The nodes of the data flow graph are formed by Vistrails *modules*. These modules encapsulate one or more functions, each having a set of parameters as attribute-value pairs and a return value. A module can represent different applications ranging from simple scripts or VTK modules to web services. The system also allows the specification of custom modules that have to fit into the data flow model used and they have to provide function for execution.

A module could be annotated with a Boolean flag that provides information of whether this modules computations could be cached to speed up re-execution or not.

The edges in the data flow graph are represented by a *connect element*. These connection elements represent data dependencies between the linked modules. On one side of the connect element a module provides data through an *oport* while the data could be received through an *iport* on the other end of the connection. Those data dependencies are used to define a order over the execution of modules and are also used to cache data produced by one module depending on the its input. The model description elements are summarized in Figure 3.9.

Execution Semantics. Before the execution the Vistrails description is translated to its corresponding data flow graph and classes are created for the modules. During execution Vistrails invokes each module in a *step* starting from the sink of the workflow. The order of invocations is determined by a depth first search through the workflow graph following the connect elements backwards. The execution semantics is derived from a Prolog engine. The Vistrails prototype is based upon the execution engine of XSB Prolog/deductive database system [RSS+97].

To optimize the execution, a signature (e.g., name, parameters and input values) of cacheable modules is stored together with the output of that execution. If during the data flow execution the same set of inputs is received by a module, the result is looked up in the cache in constant time to avoid a redundant and possibly expensive re-execution of the modules function. Subworkflows are handled by the cache manager in a similar way to store as much intermediate results as possible. For more details refer to the description in [BCS+05]. When the workflow itself changes, the data dependencies are used to only re-compute parts of the data flow that was affected by the change. Vistrails shares the cache content between all possible data flow executions in order to optimize the execution of similar or equivalent subworkflows in different Vistrails workflows.

In order to use this type of caching the system restricts modules to be stateless. The computation of modules and subworkflow should only be dependent on the name of the module and its parameter values from outputs of upstream modules. The authors believe that this should be the case for scientific functions because it guarantees the reproducibility of outputs.

Summary. The Prolog based semantics without an order allow an efficient and flexible execution also in a distributed setting. The restriction to stateless modules (actors) may be suitable for the Prolog based execution semantics but is rarely the case in practical use.

3.6 DFL: A dataflow language

Jan Hidders et al. present a data flow oriented workflow modeling language DFL in [HKS+08]. Their approach defines a graphical notation based on Petri nets combined with nested relational calculus. The general control flow is defined by the Petri nets while the computational elements are derived from nested relational calculus. The following paragraphs will give a short introduction to the modeling language.

Model Description. The basic skeleton of this model is formed by common Petri net elements. Places and Transitions form the nodes of a directed graph. Edges connect a place with a transition or a transition with a place, but not two elements of the same type. Places can store a bag of different tokens and they have an associated maximum capacity of tokens that is assumed to be infinite if there is no annotation specifying a number. The distribution of tokens over all places is called a marking or a state of the Petri net. Transitions are computational elements in the

Petri net and an invocation of this computation is called a *firing*. In order for a transition to fire some conditions have to be fulfilled. All places connected with an edge directed to the transition (input places) need to have the number of tokens required by the transition. All places that are connected to the transition with the edge pointing towards the place (output places) need to have enough free space to store the produced tokens.

The dataflow network used in this model adds some restriction to the structure. The network should be an acyclic workflow net which has a designated source and a sink place. A workflow net adds the restriction that the Petri net has to be strongly connected if an edge with a transition would be added between sink and source. Petri nets are strongly connected if there exists a path from every place to every other place, i.e., a place is always reachable from itself.

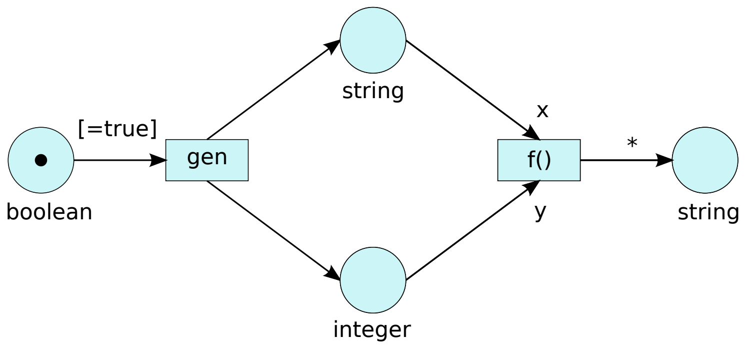


Figure 3.10: A basic DFL graph: First, an input Boolean token is evaluated using an edge annotation. The *gen*() function then creates an integer and a string token. Both tokens are passed to function $f(x, y)$, where edge labels ensure the correct assignment of token values to the corresponding parameters. The output of f is a list of strings that instantly gets unnested to a set of string tokens.

In addition the dataflow network has some extensions. Tokens could have different types: Primitive types such as Boolean, integers and string are supported. Records composed of labels and associated nested types as well as bags of other tokens allow more structured data. Finally this model also allows arbitrary nesting of data for example through sets and records. During execution of the model which is described below the tokens will also carry a history. This extension also implied that there should be types associated with places. These *place types* restrict the types of tokens that could be stored in a place. Also types need to be annotated on edges to and from transitions. Incoming edges to the transition will be labeled with allowed *input types* which define the type of data fed to the transition over this edge. Similarly, *output* *types* assign the types of tokens generated by the transition on an edge.

Transitions are annotated with *transition labels* that specify the computational function this transition performs. A core set of transition labels were borrowed from nested relational calculus and additional labels could be defined by the user. The function of this labeling is to specify the computational function, but it also indirectly specifies the number of incoming transitions, since they are mapped to parameters of the computational function. Also the input and output types are defined by the signature of the function used.

Edge labels are added to disambiguate the association of incoming edges of transition to parameters of the function. The labels carry the name of the parameter to which the edge should map.

The last extension is the *edge annotation* that provides important features: One type of edge annotation could be a comparison to form a conditional branch. There are only four annotations allowed: tokens could be checked for their Boolean values or for empty or non-empty sets. The second type of edge annotations is the '*' for nest/unnest operations. Is an edge leaving a place annotated with a star sets are unnested by passing all contained data items as separate tokens to the transition. If an edge leaving a transition is annotated with a star it will nest tokens again into a complex data structure. The way this nesting is performed depends on the history stored in tokens during a unnest operation.

Each token carries by default an empty history. During an unnest operation, the tokens that were unnested are annotated with a value that allows the nest operation to reassemble the tokens in the same type. This so called *history* is a list of pairs where each pair consists of the whole data token and the item that was just unnested top create the new token. This second part of the history pair is used to capture the position in the original whole token and will not change. During nesting operations the new value of the token will be inserted at the same location in the data structure where the original value was in the whole token according to the history. In order to nest tokens all tokens have to have the same history and all tokens from the original unnest operation have to be present. The transaction only becomes enabled once that condition is fulfilled for one set of tokens. A special case is handling of an empty set for unnesting operations. The place with all unnested tokens will receive no new tokens. By using the construction rules mentioned below, it is guaranteed that there is a second branch in the model that transmits the original set with its unnest history. This branch is created by an edge starting from the same transition as the unnest edge but it is not annotated with '*'. A transition with incoming nest edges can then fire if there is an empty collection token with a history of an empty collection in the place connected by an unannotated edge. The other placed connected with incoming nest edges do not need to have any tokens available to enable the transition. This model allows a high degree of parallelism. Many transitions could be enabled at a given time and could also fire concurrently.

A valid and well defined workflow should be modeled in a certain way. The resulting workflow network should be derivable from a single place. In the following a variety of refinement rules will be applied to replace existing network elements with a slightly larger network. This construction principle guarantees that a nest/unnest branch also has a parallel branch without nest/unnest operation in order to handle empty collections. After the construction is complete the network could be labeled to get the final workflow. A sample DFL graph using various model elements is shown in Figure 3.10.

Workflows constructed using this hierarchical refinement rules described briefly above and in more detail in [HKS+08] are always *semi sound*. That is, if one input token is present in the input place, only the output place will contain the result of the workflow computation after a finite sequence of firing transitions. There will be no data tokens left in other places once there is a data token in the output place. And most importantly there is always a firing sequence that will produce an output token in the output place.

This model shows some similarities with other existing data flow models described above. Transitions of the Petri net could easily be viewed as actors in dataflow network. The places are representing the channels, by making the data that is shipped over channels explicit. In contrast to dataflow models presented above the order of data sent to a channel does not need to equal the order of data read from the channel. This implies that the functions used for transitions need to be stateless in a general use case. The data model is similar to the hierarchical data model used for example in COMAD [DZM+11].

Summary. This modeling language provides a graphical method to describe workflows that includes data and control flow elements. The data could be modeled in different complex data types and allows tree like organization. The underlying principles of Petri nets allow some static analysis, of which the semi soundness property is probably the most important one for workflow design.

The structure of the data is reflected in the workflow, which the authors claim to be desirable. However, in many practical situations data models will change and this will imply a major redesign of the workflow. Also, in order to access data in a complex data structure many subsequent unnest followed by nest operations are required. This introduces many shims in the workflow, that are only responsible for handling data structure design instead of the actually computations that are important to scientists. Finally the execution semantics doesn't introduce any order of transition firing, which requires transitions and therefore also external scientific programs or libraries to be stateless and not interfering with other instances.

3.7 Map-Reduce Online

MapReduce online, an implementation of MapReduce that allows more streaming, was presented in [CCA+10]. A more detailed discussion of MapReduce can be found in Section 8.7. MapReduce can be viewed as a workflow engine where *mappers* and *reducers* are actors that are invoked multiple times. The equivalent to tokens are data *splits* that are fragments of the whole data set. Therefore, this framework can be roughly compared with other streaming workflow execution engines present in this chapter. The standard fault tolerance mechanism in MapReduce re-executes all mappers and reducers that were not committed. It requires checkpointing for the mappers to record which part of a split was already read and has produced output splits for the reducer. This approach is comparable to the standard provenance recording in other scientific workflow systems. However, this fault tolerance solution benefits from the very basic concepts of maps and reduces. All map and reduce operations are stateless, neatly sidestepping the issue of restoring state. Also, the structure of map-reduce programs is highly regular, always consisting of exactly one map step followed by exactly one reduce step. This fault tolerance model is comparable to Rescue DAGs with added checkpointing and speculative execution.

3.8 Others

In addition to the workflow systems presented here, a number of others are used or are in development. Ewa Deelman et al. [DGST09] presented a survey that covers a variety of systems not included here. Triana is an open source workflow system developed at Cardiff University, which is described in more detail in [DGST09]. Askalon [FPD+07] allows an easy creation and optimization of applications running on Grid systems. Pipeline Pilot is prominent, commercial workflow system by SciTegic/Accelrys [Pip13].

Chapter 4

Prelimiaries: Datalog

Scientific workflow systems typically contain a small number of actors that process a larger amount of data. This observation suggests using technologies that store and process data such as database systems or Hadoop that was discussed earlier as workflow systems.

Datalog, a purely declarative programming language, has become more popular in many areas such as database systems, programming languages, and even workflow systems [BP12, AMC+09]. Datalog itself is based on first order logic, programs are guaranteed to terminate and run in PTIME. The syntax of Datalog is similar to that of Prolog, but is evaluated differently. There are various extensions to Datalog allowing negation, adding aggregation functions, allowing creation of new constants, or supporting Skolem functions. All this extensions, modify the expressiveness of the language and the runtime of programs.

A detailed description of Datalog can for example be found in [AHV95]. The remaining chapter provides a brief overview of Datalog and its evaluation. Later chapters will provide more details for the specific aspects of Datalog discussed. In Datalog, data is represented in relation of the form $predicate(attribute_1, \ldots, attribute_n)$, where *predicate* is a string specifying the name of the relation and attributes are constants of some (possibly typed) *domain*.

Datalog programs are collection of *rules* of the form:

$$head(\bar{X}) \leftarrow body_1(\bar{Y_1}), \ldots, body_n(\bar{Y_n})$$

where the *head head* is either empty or a single relation and the *body* of the rule $body_1, \ldots, body_n$ is a (possibly empty) conjunction of relations. $\bar{X}, \bar{Y_1}, \ldots, \bar{Y_n}$ are lists of variables or constants where a single variable or constant can appear in different list. During the evaluation of the program, each variable is bound to constants in a way to make each body term true. If all body terms in the conjunction evaluate to true in a given variable binding, the head is derived to be true with the corresponding variable binding and creating a new IDB fact. When the head is empty the rule represents a integrity constraint and causes an exception or the model being invalid. If the body of the rule is empty and the head only contains constants then the rule represents an EDB fact and can be written as: $head(\bar{X})$.

In the context of scientific workflows, Datalog can serve multiple purposes. First, it can be used as a workflow system itself where the process of processing data is described by declarative rules. The execution of a workflow defined in Datalog can be optimized using the strategies that are successfully employed in database systems. The evaluation (or execution) of a Datalog program can use distributed resources efficiently [ZGL12]. Furthermore, by creating user defined predicates, other computational tasks or even shell scripts can be used in a workflow defined in Datalog.

Furthermore, Datalog can be used to query and analyze workflow descriptions, workflow models of computation, or the execution of workflows. In the following chapters, Datalog is used frequently as a means to describe and analyze workflows and their execution.

Chapter 5

Provenance

The word provenance describes the source or the history of an object. It is used in art or digital libraries, where it refers to the documented history of an art object, or the documentation of processes in a digital object's life cycle [Hig09]. Provenance is also an important concept in the "e-science community" [SPG05] that workflow systems should provide to guarantee Reproducibility of scientific analysis and processes. It describes the dependencies of processes and data items, i.e., the lineage and processing history of data. Scientific workflow systems often record events during their execution to capture such dependencies. Commonly used models for provenance are the Read/Write model [BML+06], and the Open Provenance Model (OPM) [MCF+10]. Events are recorded when actors consume tokens (`read` or `used_by` events) and produce tokens (`write` or `generated_by` events). Therefore, provenance persists all activities necessary to reconstruct the workflow execution and the content of queues.

Provenance in scientific workflow systems is still an active research area. A recent publication [CVDK+12] gave an overview of uses of provenance and open research questions. Datalog can easily be used to describe, query, and analyze provenance information.

Workflow descriptions and provenance are frequently specified as graphs. Sometimes it is desirable to provide a limited but user friendly interface to perform graph queries. Regular path queries have emerged as a tool that can be utilized to query graphs in a simple and intuitive way. In [DCVK+13] it is described how to implement regular path queries in various relational database engines and in Datalog. Datalog performs particular well showing it is well suited even as a back-end for other graph query languages to analyze workflows and provenance.

When using Datalog as a workflow description language but also as a component to analyze provenance in a scientific workflow system, it is important to develop correct code and to understand the behavior of a purely declarative Datalog program. The following chapters present various novel techniques how to visualize and analyze the evaluation of a Datalog program. Later chapters will present novel techniques developed to analyze provenance of workflows using Datalog and to use this information to improve the description and execution of scientific workflows.

The remaining chapter presents the ideas developed by Saumen Dey and the author that appeared in [DKBL12]. Logic rules are proposed as a formal foundation for graph-based, temporal models of provenance, for querying provenance graphs (traces), and for reasoning about traces and their connection to the workflows that generated them. In particular, arguments are provided that Datalog provides a “lingua franca” for provenance.

5.1 Datalog for Provenance Analysis

The Open Provenance Model (OPM) [MCF+11] provides a small, extensible core for representing and exchanging provenance information in a technology-neutral manner. By design, OPM is a least common denominator, leaving aside certain aspects, including how to query provenance information. Similarly, OPM comes with a set of inference rules (e.g., for transitively closing some relations, or for stating temporal constraints that are implied by provenance assertions), but as pointed out by [KMVdB10], the temporal semantics of OPM graphs is only partially defined in [MCF+11], leading to ambiguous or incompletely specified situations.

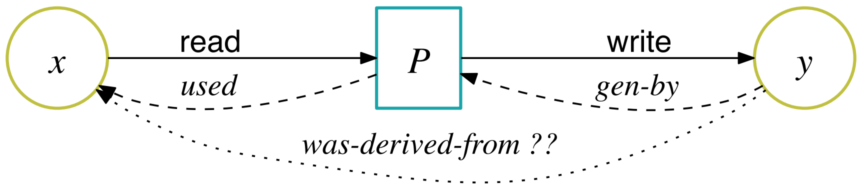


Figure 5.1: Do the observables $x \overset{\text{read}}{\to} P$ and $P \overset{\text{write}}{\to} y$ imply that y *was-derived-from* x? Or that $t_{\text{read}} < t_{\text{write}}$ holds?

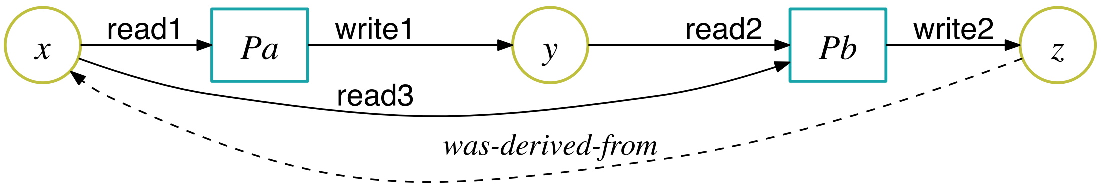


Figure 5.2: z *was-derived-from* x. Does $t_{\mathsf{read}_2} < t_{\mathsf{write}_2}$ follow?

Example 1 Consider the provenance graph in Figure 5.1. In OPM, it shows a (data) *artifact* x that was *used* by *process* P, and another artifact y that *was-generated-by* (short: *gen-by*) P. The rest of this chapter uses the terminology that the graph records a "read" and a "write" observable, denoted $x \overset{\mathsf{read}}{\to} P$ and $P \overset{\mathsf{write}}{\to} y$, respectively. Given this information, it may seem natural to assume that (1) y *was-derived-from* x (the dotted line), and that (2) the read event (t_{read}) occurred *before* the write event (t_{write}), i.e., $t_{\mathsf{read}} < t_{\mathsf{write}}$. However, neither (1) nor (2) are logical consequences of the provenance in Figure 5.1, i.e., it cannot be inferred that y *was-derived-from* x! Indeed, OPM correctly treats *was-derived-from* as a separate observable, e.g., P might have written y first, *then* read x afterwards, and so it cannot be assumed (and thus not inferred) that y *was-derived-from* x.[1]

If, on the other hand, the fact that y *was-derived-from* x has been (independently) asserted, can it be inferred from Figure 5.1 that x was used by P *before* y was generated by P, i.e., that $t_{\mathsf{read}} < t_{\mathsf{write}}$ holds? Again, the somewhat surprising answer is: No! For example, the use (reading) and generation (writing) of x and y by P are not necessarily the only things that happened. In particular, there might be another derivation of y from x (which gave rise to the *was-derived-from* edge in the first place), making $t_{\mathsf{read}} > t_{\mathsf{write}}$ a real possibility. □

Example 2 Consider the provenance graph in Figure 5.2, asserting that z *was-derived-from* x. Does $t_{\mathsf{read}_2} < t_{\mathsf{write}_2}$ or $t_{\mathsf{read}_3} < t_{\mathsf{write}_2}$ follow? As before, if no further information is available, neither proposition can be inferred: it is simply unknown whether the path $x.P_a.y.P_b.z$ or the path $x.P_b.z$ (or yet another one, not included in the figure) are the reason for the *was-derived-from* edge in Figure 5.2. The OPM semantics also handles this case correctly and does *not* imply that $t_{\mathsf{read}_i} < t_{\mathsf{write}_j}$ (for any $i \le j$). On the other hand, OPM also does not provide a means to

[1]If the computation P is a function or service call, then P indeed *first* consumes all inputs, *then* produces all outputs. This assumption is often correct, but a *process* P is not necessarily limited to such strict behavior and may interleave read/write events in many ways [Kah74, LP95].

specify when such inferences would indeed be correct, e.g., in the common case where the result of a write is in fact directly dependent on an earlier read. □

When employing OPM as a model for provenance recorded by a scientific workflow system [DF08], another limitation becomes apparent: OPM only deals with *retroactive provenance* (the usual data lineage captured in a trace graph T), but not with so-called *prospective provenance*[2], i.e., workflow specifications W, which are recipes for future (and past) derivations [LLCF10]. These were out of scope for OPM and, apparently, are also out of scope for current W3C standardization efforts [W3C].

On the other hand, it is easy to see that distinguishing between traces T and workflows W (and then interpreting the former as instances of the latter) can provide valuable information and additional functionality for provenance applications.

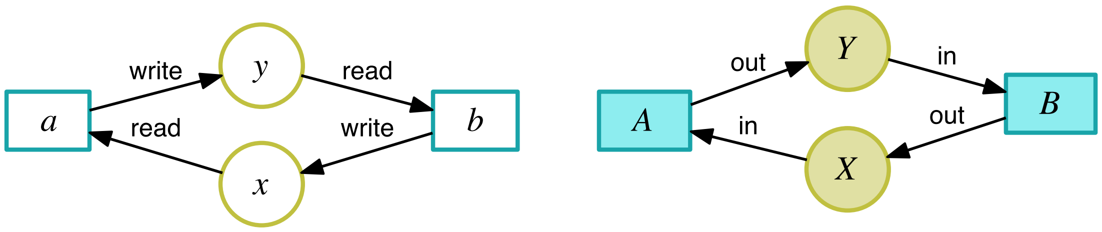


Figure 5.3: Trace T (left) and workflow W (right).

Example 3 Consider the graphs in Figure 3: executing workflow W, might have produced the trace T.

In order to validate T's structure, i.e., to check whether T can be an instance of W, its nodes and edges are linked to W: e.g., edges $x \overset{\mathsf{read}}{\rightarrow} a$ and $a \overset{\mathsf{write}}{\rightarrow} y$ in T (data x was *read* and data y was *written* by process *invocation* a), have corresponding edges $X \overset{\mathsf{in}}{\rightarrow} A$ and $A \overset{\mathsf{out}}{\rightarrow} Y$ in W, linking data *containers* X and Y to the *process* (or: actor) A. Clearly, in order to validate T's structure, a representation of the workflow structure W is needed in the first place.

However, even if T is structurally valid w.r.t. W, other (here: temporal) inconsistencies may arise: The cycle in T indicates an inconsistent trace,[3] but the correctness is uncertain (for similar

[2]This "near-oxymoron" captures a practically useful notion: When a scientist is asked to explain how a certain result was obtained, in the absence of runtime traces, he can point to the script/workflow that was used to generate the data products; so workflows are provenance, too.

[3]If read and write observables are temporally or causually linked, a *strict* partial order is implied and a cycle shouldn't have been observed.

reasons as those in the previous examples). On the other hand, the cycle in W is usually *not* a concern: it simply means that W has a feedback loop, which is a rather common workflow pattern (cf. Section 5.3). $\square$

Motivated by these examples, the next sections present the following contributions: (1) A semi-structured data model is presented, i.e., graphs with labeled edges $x \overset{\ell}{\to} y$, as a uniform representation of all provenance information, i.e., traces (à la OPM) and associated workflows of which they are instances. This allows the use of regular path queries [CDGLV03] as a convenient "macro-language" for concisely expressing generalized reachability queries on traces and workflows. By representing schema-level information (workflows) and instance-level information (traces) together in a single model, structural constraints can be expressed and checked easily using Datalog rules.

(2) The use of integrity constraints is proposed, i.e., rules of the form $\mathsf{false}_{\mathsf{ic}}(\bar{Y}) \leftarrow \mathit{denial}(\bar{X})$[4] as a way to express provenance semantics. Workflow systems differ in their models of computation (MoCs) and thus make different assumptions about how workflow components (a.k.a. *actors*, *processors*, *modules*, etc.) work, i.e., whether, and in which way, they can be stateful; how they consume their inputs, produce their outputs; and so on. As a result, different systems use different models of provenance (MoPs), with different temporal semantics. Thus, instead of "hard-wiring" a fixed temporal semantics to a particular graph-based MoP, logic constraints are used again to obtain a "customizable" temporal semantics.

(3) This concept is illustrated by providing *firing constraints* at the workflow level, which induce *temporal constraints* $\leq_{\mathsf{f}}$ at the level of traces (cf. Figure 5.4). These temporal-constraint generating rules can be chosen to conform to the temporal axioms in [KMVdB10], or to accommodate a different temporal semantics, as implied by the MoC of a specific workflow or workflow system.

The following sections describe only some of the many ways in which logic rules and Datalog can be harnessed for provenance querying and reasoning. An overview of the approach is given and a few illustrative examples are provided.

[4]The rule body $\mathit{denial}(\bar{X})$ specifies the "bad" situations to be avoided; $\bar{Y} \subseteq \bar{X}$ are witnesses of constraint violations.

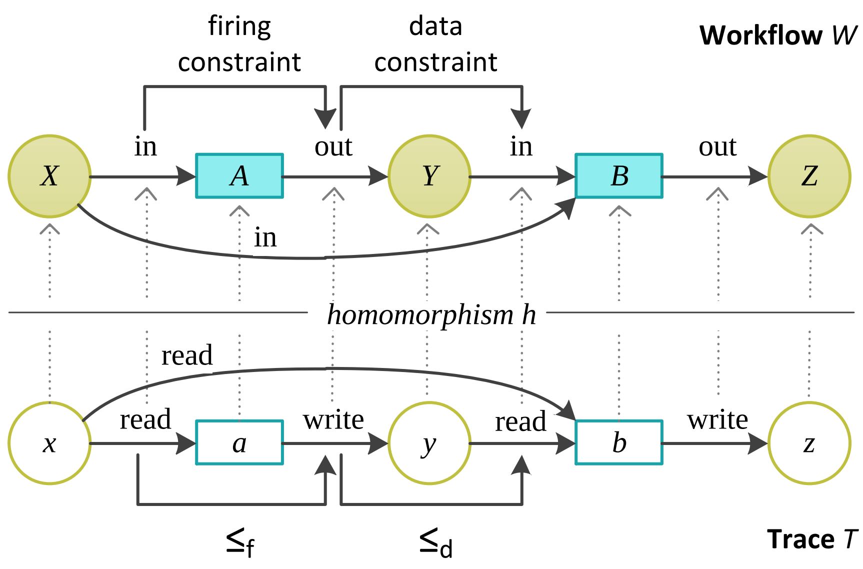


Figure 5.4: Workflow W (top) vs Trace T (bottom): Traces are associated to workflows, guaranteeing structural consistency; workflow-level (firing or data) constraints induce temporal constraints $\leq_{\mathsf{f}}$ and $\leq_{\mathsf{d}}$ on traces.

5.2 A Unified Provenance Model

The approach presented here, aims to integrate trace-level provenance information, schema-level information (workflow specifications), and temporal information in a single, uniform representation. The approach is based on an underlying semistructured data model, which consists of labeled, directed graphs of the form $G = (V, E, L)$, with vertices V, labels L, and labeled edges $E \subseteq V \times L \times V$.

In the following, workflows W and traces T are viewed as subgraphs of G. Similarly, the temporal model consists of labeled edges (modeling one or more "*before*" relations).

Workflows. A workflow $W = (V_W, E_W, L_W)$ is a labeled graph whose nodes $V_W = C \cup P$ are data *containers* C and *processes* P. Processes are computational entities (often consisting of smaller internal steps, a.k.a. *invocations* or *firings*) that can send and receive data. *Containers* represent structures, e.g., FIFO queues, that hold data during the communication between processes. In a workflow W, edges $E_W = E_{\mathsf{in}} \cup E_{\mathsf{out}}$ are either *input* edges $E_{\mathsf{in}} \subseteq C \times \{\mathsf{in}\} \times P$, or *output* edges $E_{\mathsf{out}} \subseteq P \times \{\mathsf{out}\} \times C$, so $L_W = \{\mathsf{in}, \mathsf{out}\}$. We also write $\mathsf{in(C,P)}$ and $\mathsf{out(P,C)}$ to denote edges $\mathsf{C} \overset{\mathsf{in}}{\to} \mathsf{P}$ and $\mathsf{P} \overset{\mathsf{out}}{\to} \mathsf{C}$, respectively. The former means process P can read data artifacts from container C, while the latter means that process P can write data artifacts to container C.

Traces. A trace $T = (V_T, E_T, L_T)$ is a labeled graph whose nodes $V_T = D \cup I$ are *data artifacts* D or *process invocations* I; edges $E_T = E_{\mathsf{read}} \cup E_{\mathsf{write}} \cup E_{\mathsf{df}}$ are *read* edges $E_{\mathsf{read}} = D \times \{\mathsf{read}\} \times I$, *write* edges $E_{\mathsf{write}} = I \times \{\mathsf{write}\} \times D$, or *derived-from* edges $E_{\mathsf{df}} = D \times \{\mathsf{df}\} \times D$, so $L_T = \{\mathsf{read}, \mathsf{write}, \mathsf{df}\}$. The link between traces and workflows is established through homomorphisms:

Definition 1 *Let $G = (V, E, L)$ and $G' = (V', E', L')$ be labeled graphs, and let $h = (h_1, h_2)$ be a pair of mappings $h_1 : V \to V'$ and $h_2 : L \to L'$. Then h is a* homomorphism *from G to G' if*

$$(x \xrightarrow{\ell} y) \in E \ \text{ implies } \ (h_1(x) \xrightarrow{h_2(\ell)} h_1(y)) \in E'.$$

The functions h_1 and h_2 map nodes and labels from G to corresponding ones in G'. The model described here associates read and write edges in T with in and out edges in W, so $h_2 = \{\mathsf{read} \mapsto \mathsf{in}, \mathsf{write} \mapsto \mathsf{out}\}$. The mapping h_1 is usually given as part of a trace, i.e., when testing whether T is valid w.r.t. a workflow W, it is not necessary to *search* for h. Instead, traces needing validation already have corresponding workflow annotations embedded within them: Mappings cont: $D \to C$ and proc: $I \to P$ associate data items and process invocations with data containers and processes, respectively. The following rules derive false iff trace T with workflow mappings cont and proc are *not* a homomorphism:

```
false_H(read(D,I),in(C,P)) :-
    read(D,I), cont(D,C), proc(I,P), !in(C,P).
false_H(write(I,D),out(P,C)) :-
    write(I,D), cont(D,C), proc(I,P), !out(P,C).
```

Thus, if $\mathsf{false}_{\mathsf{Hom}}$ is empty, trace T is *structurally-valid* w.r.t. workflow W. Additional structural constraints for traces can be easily defined in a similar manner:

A *write-conflict* occurs when a data artifact has multiple incoming write edges; a *type-conflict* occurs when edges link nodes of the wrong type (e.g., directly linking invocations, instead of going through data nodes). In the integrity constraint rules, the following auxilliary view for transitive dependencies is used:

```
dep(X,Y) :- read(X,Y).
dep(X,Y) :- write(X,Y).
tcdep(X,Y) :- dep(X,Y).
tcdep(X,Y) :- tcdep(X,Z), tcdep(Z,Y).
cycle(X,Y) :-tcdep(X,Y), tcdep(Y,X), !X=Y.
```

The processes involved in a *write-conflict* is computed using the following rule:

```
false_wc(X,Y) :-
    write(X,D), write(Y,D), !X=Y.
```

Furthermore, a *type-conflict* is computed using the rule:

```
false_tc(X,Y) :-
    dep(X,Y), cont(X,C1), cont(Y,C2).
false_tc(X,Y) :-
    dep(X,Y), proc(X,P1), proc(Y,P2).
```

Temporal Model. A temporal semantics is obtained on top of the graph-based model by using rules to define a "*before*" relation (e.g., $\leq_{\mathsf{f}}$ and $\leq_{\mathsf{d}}$ in Figure 5.4) amongst events. During workflow execution, the following events are observed: When a process is executed an *invocation* event is recorded, when a data artifact is read by an invocation a *read* event is recorded, and when an invocation writes a data artifact a *write* event is recorded. In the temporal model the order of events are constrained: bf(E1,E2) means that E1 happened *before* E2 and bfs(E1,E2) means that E1 happened *before or simultaneously* with E2.

A *data-constraint* between out-edges $\mathsf{P}_A \xrightarrow{\mathsf{out}} \mathsf{C}_Y$ and in-edges $\mathsf{C}_Y \xrightarrow{\mathsf{in}} \mathsf{P}_B$ of a data container C_Y at the workflow-level, implies an obvious temporal constraint at the trace-level: A write (creation) of data y must come before any read of y (also see Figure 5.4):

```
bf(write(I,D), read(D,J)) :- write(I,D), read(D,J).
```

Read, write, and derived-from edges in T capture dataflow. This information gives rise to a temporal order ("*flow-time*") among events. This temporal order is inferred and the corresponding temporal relations bf and bfs are created using the rules described here. Skolem functions are used in rules (safely), e.g., to create unique identifiers, and to reify edges as nodes of a temporal structure: e.g., the (unspecified) execution time of an invocation is represented by a term invoc(I).

Before an invocation reads data, it must have started:

```
bfs(invoc(I), read(D,I)) :- read(D,I).
```

Similarly, a data artifact could not have been written after an invocation has been completed:

```
bfs(write(I,D), invoc(I)) :- write(I,D).
```

When a data artifact is written by a process invocation and read by another invocation, the former must not have started after the latter has completed:

```
bfs(invoc(I), invoc(J)) :- write(I,D), read(D,J).
```

When a data artifact is derived from another data artifact, the latter must have been written before the former:

```
bf(write(J,Y), write(I,X)) :- df(X,Y), write(I,X), write(J,Y).
```

A *firing-constraint* between in-edges $\mathsf{C}_X \overset{\mathsf{in}}{\to} \mathsf{P}_A$ and out-edges $\mathsf{P}_A \overset{\mathsf{out}}{\to} \mathsf{C}_Y$ of a process P_A is defined via a relation $\mathsf{fc}(\mathsf{C}_X,\mathsf{P}_A,\mathsf{C}_Y)$. This constraint ensures that any invocation of process P_A that reads data from container C_X and that writes into container C_Y has an associated temporal constraint: the invocation output depends on the read input.

If a user-defined constraint is provided by fc(X,B,Z) (cf. Figure 5.4), then the bf(read(x,b),write(b,z)) temporal relation at the trace-level can be inferred:

```
bf(read(X,I), write(I,Y)) :-
      fc(C1,P,C2), read(X,I), proc(I,P),
      write(I,Y), cont(X,C1), cont(Y,C2).
```

Wall-Clock Time: In addition to temporal dependencies inferred from dataflow observables ("flow-time"), a provenance recorder may record events with a *wall-clock* timestamp. The observable `time(E,T)` means that event `E` was recorded with wall-clock time `T`. Wall-clock time and flow-time constraints can combined to infer additional temporal information for a trace:

```
bf(E1, E2) :- time(E1, T1), time(E2, T2), T1 < T2.
bfs(E1, E2) :- time(E1, T1), time(E2, T2), T1 <= T2.
```

5.3 Hamming Workflow Variants

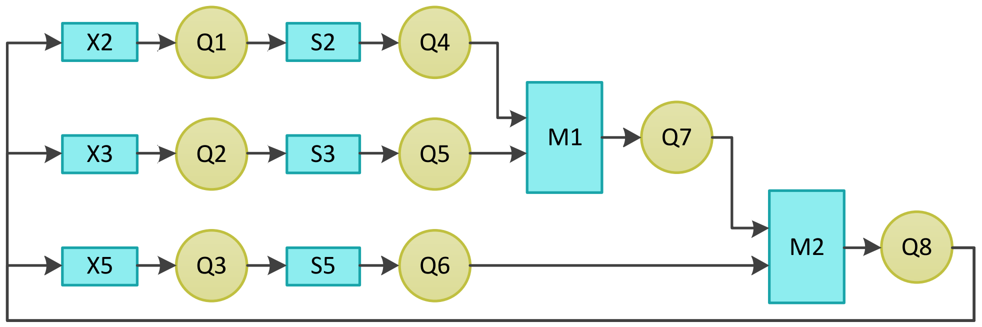
To illustrate the earlier definitions of this chapter, this section uses two variants of a workflow to compute the *Hamming numbers*[5] $H = \{2^i \cdot 3^j \cdot 5^k \mid i, j, k \geq 0\}$ incrementally, i.e., as an ordered sequence $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, \ldots$ Two workflow variants H_1 and H_3 are shown in Figure 5.5. Note that the workflow graphs contain the same nodes (processes and containers), but are wired slightly differently (as it turns out, this makes a big difference). The data containers Q_i are queues (FIFO buffers); Q_8 is the distinguished output, where the Hamming numbers will appear in the correct order. M_1 and M_2 are *merge actors*, i.e., processes which take two ordered input sequences and merge them into an ordered output sequence. If presented with the same item in both streams, the output stream will only contain one copy of the element, so duplicates are removed. The actors X2, X3, and X5 multiply their inputs with 2, 3, and 5, respectively. Last not least, the *sample-delay actors* S2, S3, S5 are used "to prime the pump": initially (i.e., before reading any input), they output the number 1 to get the loop(s) going. Subsequently, they simply output whatever they received as an input. By design, the Hamming workflows H_1 and H_3 define an infinite output stream, i.e., the processes "run forever".

5.3.1 Structural Validity

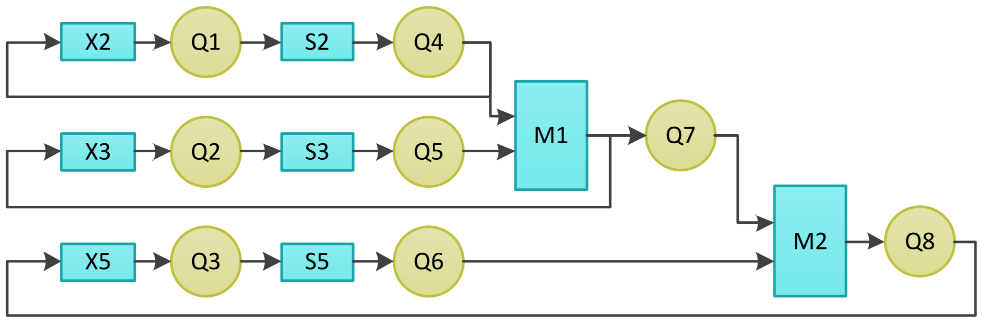
Figure 5.6 shows a (partial) trace T_H, i.e., for computing the Hamming numbers $n \leq 15$.[6] In order to test if a trace T is *structurally-valid* w.r.t. a workflow W, it is checked if there is a homomorphism from T to W. Here, a homomorphism between T_H, in Figure 5.6 and the 1-loop variant H_1 of the Hamming workflow in Figure 5.5a can be established. However,

[5]a.k.a. *regular numbers*; see [Dij81, Hem88] for details

[6]Figure 5.7 shows user-defined trace-views for $n \leq 1000$.



(a) Hamming workflow H_1: "one loop" variant



(b) Hamming workflow H_3: "three loops" variant

Figure 5.5: Workflow variants H_1, H_3; output queue is Q_8.

when attempting to find a homomorphism between the same trace and the 3-loop variant H_3 in Figure 5.5b, corresponding in-edges cannot be found for the dashed (red) read-edges in Figure 5.6. Note that the "bad" (missing) edges can be found automatically with the Datalog (denial) rules for $\mathsf{false}_{\mathsf{Hom}}$ in Section 5.2.

5.3.2 User-Defined Provenance Queries

The unified provenance model can easily be queried further using Datalog. For example, a user might want to know the lineage of a particular Hamming number (i.e., which other Hamming numbers "went into it"), or how many duplicates were derived in their particular workflow variant, etc. The dependencies between data items can be obtained by focusing on the read/write observables of certain processes P:

```
q(D1,P,D2) :– read(D1,I), write(I,D2), proc(I,P), focus(P).
```

Here, focus is a user-defined predicate to limit query answers to processes of interest, e.g., we may focus on X2, X3, and X5. Thus, tuples in the answer relation q can be viewed as edges $d_1 \xrightarrow{p} d_2$, linking data items to each other, with the label p denoting the process (multiplication factor) involved. Figure 5.7 shows the resulting graph structure[7] for a trace containing the computation of Hamming numbers up to 1000. Labels for edges are represented by a different coloring (here: red, blue, and green are used to represent labels "$\times 5$", "$\times 3$", and "$\times 2$", respectively).

One can clearly see on the in-degree of nodes that in Figure 5.7a many Hamming numbers are produced in multiple ways, i.e., the custom-provenance graph is a DAG. In contrast, Hamming numbers in H_3 are produced by one path only *without* unnecessary duplicates as can be seen in Figure 5.7b, i.e., this graph is a tree.

5.4 Summary

The theory and practice of provenance are not always as well aligned as one may wish for.[8] The DBLP (databases & programming languages) community, among others, has been advancing our understanding of fundamental principles of provenance, and notions such as *Why*, *How*, and *Where* provenance [BKT01, CCT09], provenance semirings [GKT07], provenance as proof-trees, dependencies, program slicing [CAA07], relationships to causal reasoning [MGH+10, Che10], etc. are slowly becoming more widely known and understood better. At the same time, the many practical applications of provenance have led practitioners and system developers to move ahead rapidly with "provenance-enabling" their systems, proposing models, languages, and even W3C standards. As a contribution to further grow the connections between theory and practice of provenance [ABC+10], we have proposed to use logic rules, and Datalog in particular for querying provenance, checking structural constraints (e.g., whether a trace is valid, i.e., homomorph to its associated workflow), and specifying temporal constraints.

Datalog seems particularly well-suited as a "lingua franca" and bridge between theory and practice: On the one hand, there is a rich body of research and formal results about the complexity and expressiveness of Datalog and numerous fragments or extensions [DEGV01]. Queries on labeled graphs are well supported, e.g., regular path queries have a direct encoding in Datalog, and theoretical results on query containment and view-based query processing [CDGLV03] can be

[7]Nodes are not intended to be readable, but if desired can be zoomed-into in the PDF version.
[8]In theory, there is no difference between theory and practice; but in practice, there is.

exploited in various ways. Similarly, Datalog variants such as Statelog [LLM98] or Datalog-LITE [GGV02] provide means to naturally express temporal queries over provenance graphs. From a practitioner's point of view, Datalog is also attractive: powerful Datalog engines are available for experimentation with rule sets (e.g., to test and compare different provenance semantics), and can be used to deploy provenance querying and reasoning systems. Datalog prototypes for the approach demonstrated here are under development.

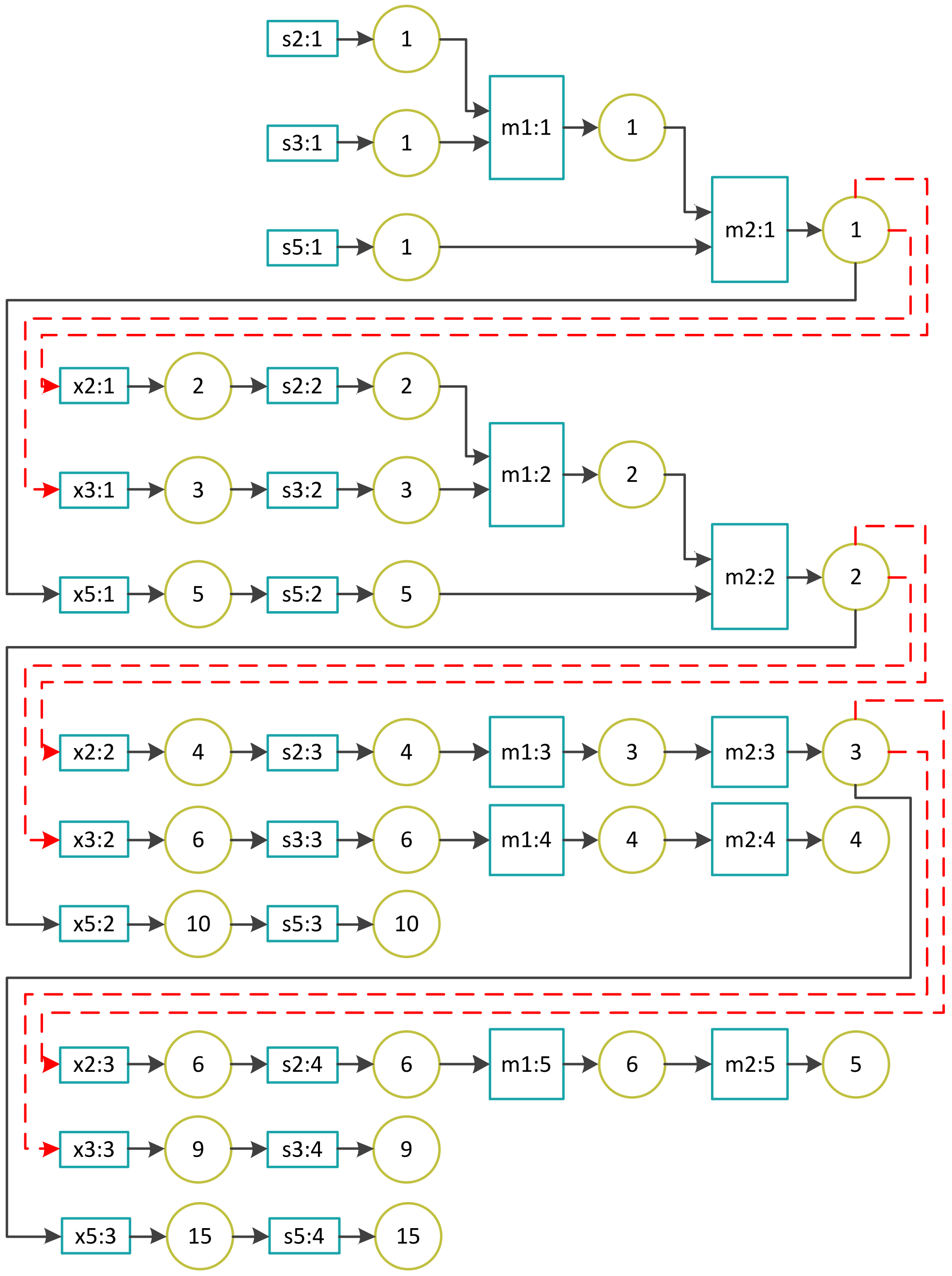


Figure 5.6: Excerpt of a Hamming workflow trace T_H containing solid (black) and dashed (red) edges: This trace is homomorph to H_1 in Figure 5.5a but it is *not* homomorph to H_3 in Figure 5.5b since the dashed (red) edges in T_H cannot be mapped to corresponding edges in H_3.

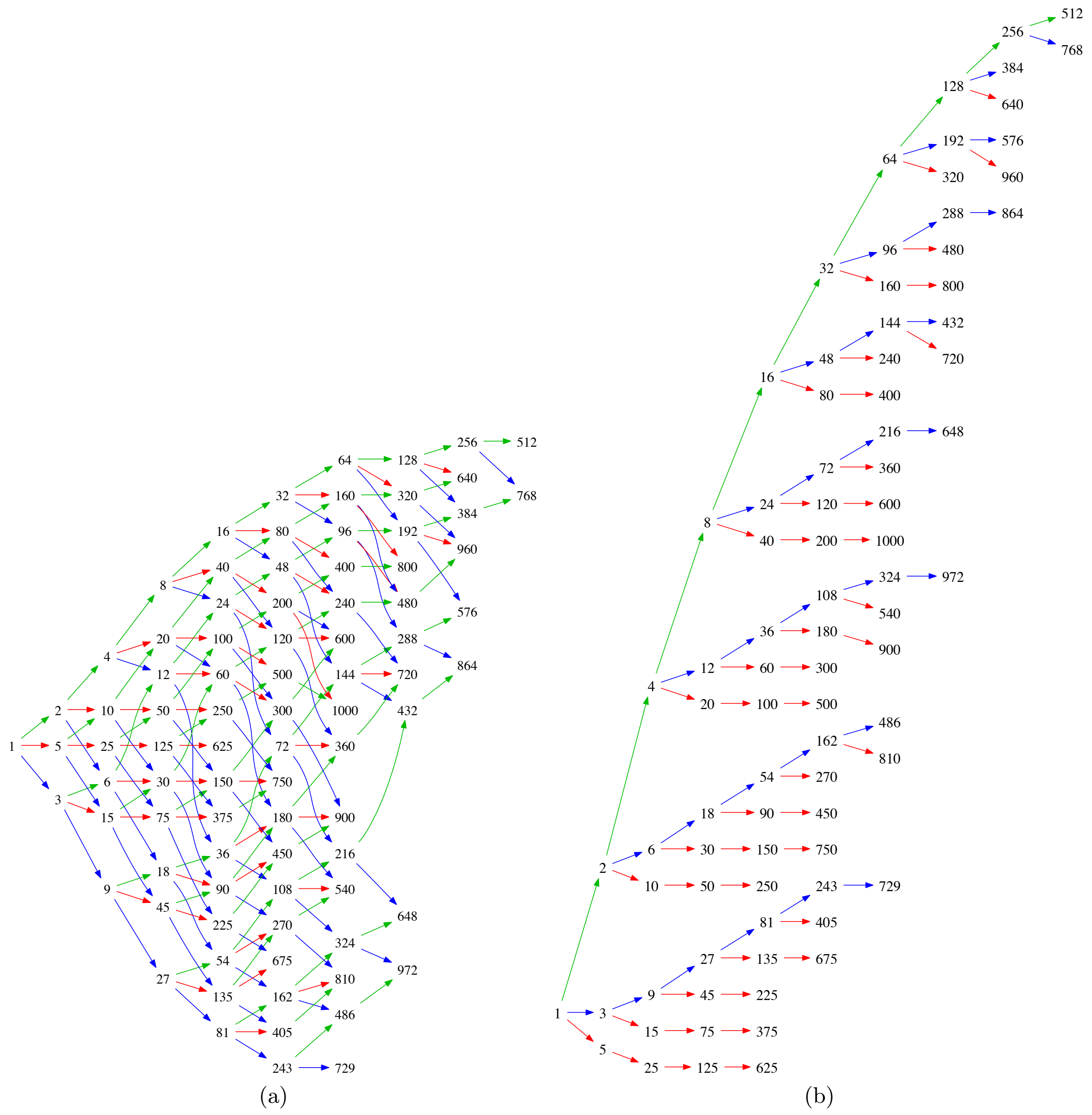


Figure 5.7: User-defined provenance for Hamming numbers up to 1000 (a) for H_1 (“*Fish*”) and (b) for H_3 (“*Sail*”)

Chapter 6

Datalog Profiling and Debugging

A challenge in using Datalog as a workflow description language is the difficulty to express workflow correctly in this declarative language. Addressing these difficulties, this chapter presents a framework for declarative debugging and logic profiling developed by the author and published on in [KLS12]. It also describes a prototypical implementation of this framework called Gpad.

6.1 Introduction

Developing declarative, rule-based programs can be surprisingly difficult in practice, despite (or because of) their declarative semantics. Possible reasons include what Kunen long-ago called *the PhD effect* [Kun91], i.e., that a PhD in logic seems necessary to understand the meaning of certain logic programs (with negation). Similarly, *An Amateur's Introduction to Recursive Query Processing* [BR88] from the early days of deductive databases, rather seems to be for experts only. The following sections describe the difficulties a workflow developer and aspiring Datalog programmer faces who wants to develop complex programs.

The meaning and termination behavior of a *Prolog* program P depends on, among other things, the order of rules in P, the order of subgoals within rules, and even (apparently minor) updates to base facts. Consider, e.g., the program for computing the transitive closure of a directed graph, i.e., $P_{\mathsf{tc}} =$

$$r_1: \quad \mathsf{tc}(X, Y) \leftarrow \mathsf{e}(X, Y).$$

$$r_2: \quad \mathsf{tc}(X, Z) \leftarrow \mathsf{e}(X, Y), \mathsf{tc}(Y, Z).$$

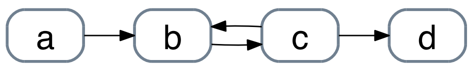
Seasoned logic programmers know that P_{tc} is *not* a correct way to compute the transitive closure in Prolog. [1] Under a more declarative *Datalog* semantics, on the other hand, P_{tc} indeed *is* correct, since the result does *not* depend on rule or subgoal order. The flip side, however, is that effective and practically useful procedural debugging techniques for Prolog, based on the *box model* [TB93], are not available in Datalog. Instead, new debugging techniques are needed that are solely based on the declarative reading of rules.

Let $M{=}P(I)$ be the model of P on input I. Bugs in P (or I) manifest themselves through unexpected answers (ground atoms) $A \in M$, or expected but missing $A \notin M$. The key idea of the developed approach is to rewrite P into a *provenance-enriched* program $\hat{P}$, which at runtime records the derivation history of $M{=}P(I)$ in an extended model $\hat{M}{=}\hat{P}(I)$. A provenance graph G is extracted from $\hat{M}$, which the user can explore further via predefined views and ad-hoc queries.

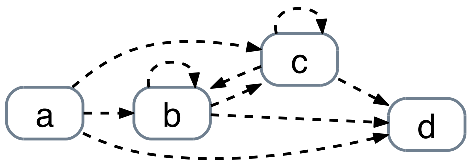
Use Cases Overview. Given an IDB atom A, GPAD allows to answer questions such as the following: What is the *data lineage* of A, i.e., the set of EDB facts that were used in a derivation of A, and what is the *rule lineage*, i.e., the set of rules used to derive A? When chasing a bug or trying to locate a source of inefficiency, a user can explore further details: What is the graph structure G_A of all derivations of A? What is the *length* of A, i.e., of shortest derivations, and what is the *weight*, i.e., number of simple derivations (proof trees) of A?

For another example, assume the user encounters two "suspicious" atoms A and B. It is easy to compute the common lineage $G_{AB} = G_A \cap G_B$ shared by A and B, or the lowest common ancestors of A and B, i.e., the rule firings and ground atoms that occur "closest" to A and B in G_{AB}, thus triangulating possible sources of error, similar to ideas used in delta debugging

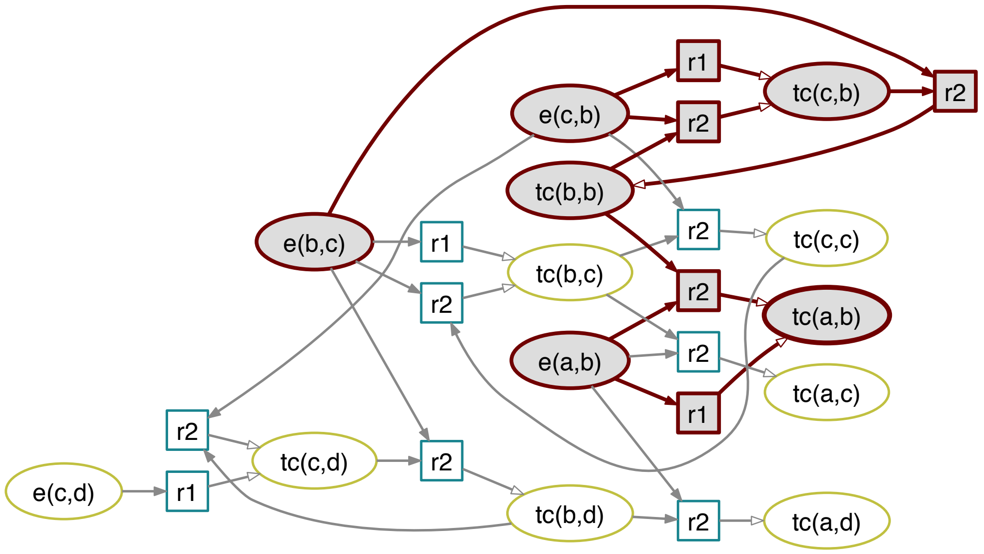
[1] For $I = \{\mathsf{e}(\mathsf{a}, \mathsf{b}), \mathsf{e}(\mathsf{b}, \mathsf{a})\}$ the query ?-tc(c,X) correctly returns "No", while the similar ?-tc(X,c) will not terminate! Prolog's behavior gets worse when swapping rules r_1 and r_2, or when creating left- or doubly-recursive variants P^l_{tc}, P^d_{tc}, respectively.



(a) Input graph: edge relation e



(b) Output: transitive closure $\mathsf{tc} = \mathsf{e}^+$



(c) Provenance graph G; highlighted subgraph $G_{\mathsf{tc(a,b)}}$; firing nodes r (boxes), atom nodes A (ovals); edge types $A \overset{\text{in}}{\to} r$ and $r \overset{\text{out}}{\twoheadrightarrow} A$ (edge labels not shown)

Figure 6.1: P^r_{tc}-provenance graph for input e, with derivations of tc(a,b) highlighted in (c)

[Zel02].

Since nodes in G_A are associated with relation symbols and rules, a user might also want to compute other aggregates, i.e., not only at the level of G_A (ground atoms and firings), but at the level of (non-ground) rules and relation symbols, respectively. Through this schema-level profiling, a user can quickly find the "hot spots" in P, e.g., rules having the most (or least) number of firings.

Running Example. Figure 6.1 gives an overview using a very simple example: (a) depicts an input graph e, while (b) shows its transitive closure $\mathsf{tc} = \mathsf{e}^+$. The structure and number of distinct derivations of tc atoms from base edges in e can be very different, e.g., when comparing the right-recursive P_{tc} ($=P^r_{\mathsf{tc}}$) above, with left-recursive or doubly-recursive variants P^l_{tc} or P^d_{tc}, respectively.

The provenance graph G (or the relevant $G_A \subseteq G$, given a goal A) provides crucial information to answer the above use cases. Figure 6.1c shows the provenance graph for the computation of tc via P^r_{tc} from above. Box nodes represent *rule firings*, i.e., individual applications of the immediate consequences operator T_P, and connect all body atoms to the head atom via a unique firing node. For the goal atom $A = \mathsf{tc}(\mathsf{a}, \mathsf{b})$ the subgraph G_A, capturing all possible derivations of A, is highlighted (through filled nodes and bold edges).

The following sections present the method for debugging and logically profiling a Datalog program P via a provenance-enriched rewriting $\hat{P}$ in more detail. The key idea is to extract from the extended model $\hat{M}$ a provenance graph G which is then queried, analyzed, and visualized by the user. Given a debug goal A, relevant subgraphs G_A can be obtained easily and further analyzed via a library of common debug views and ad-hoc user queries. At the core of the approach are rewritings that (i) capture rule firings, then (ii) reify them, i.e., turn them into nodes in G (via Skolem functions), while (iii) keeping track of derivation lengths using Statelog [LLM98, Lud98], a Datalog variant with states. The simplicity and system-independence is an important benefit of this approach. The approach is rapidly prototyped for rather different Datalog engines, i.e., DLV [LPF+06] and LogicBlox [MHB+10], Furthermore, there is a close relationship of the provenance graphs generated by GPAD with provenance semirings [GKT07]. But the focus of this approach is on presenting a simple, effective method for debugging and profiling declarative rules for "mere mortals".

6.2 Provenance Rewritings for Datalog

This section describes the three Datalog rewritings $P \overset{F}{\leadsto} \cdot \overset{G}{\leadsto} \cdot \overset{S}{\leadsto} \hat{P}$ for capturing rule firings, graph generation, and Statelog evaluation, respectively.

6.2.1 Recording Rule Firings: $P \overset{F}{\leadsto} P^F$

The first rewriting (cf. Green et al. [GKIT07]), records the provenance of rule firings. Let r be a unique identifier of a rule in P. We assume r to be safe, i.e., every variable in r must also occur positively in the body:

$$r : \quad H(\bar{Y}) \leftarrow B_1(\bar{X}_1), \ldots, B_n(\bar{X}_n)$$

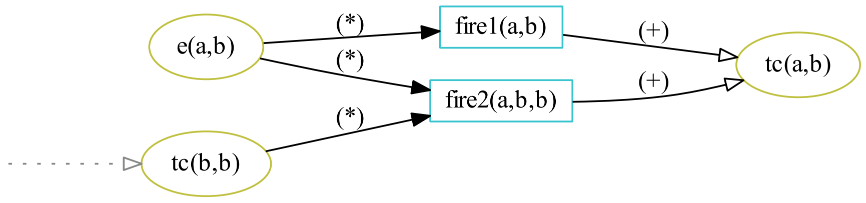


Figure 6.2: Subgraph with two rule firings $\mathsf{fire}_1(\mathsf{a},\mathsf{b})$ and $\mathsf{fire}_2(\mathsf{a},\mathsf{b},\mathsf{b})$, both deriving $\mathsf{tc(a,b)}$

Let $\bar{X} = \bigcup_i \bar{X}_i$ include all variables in r, ordered, e.g., by occurrence in the body. Since r is safe, $\bar{Y} \subseteq \bar{X}$, i.e., the head variables are among the $\bar{X}$. The rule r is now replaced by two new rules in the rewritten program P^F:

$$\begin{aligned} r_{in} : \mathsf{fire}_r(\bar{X}) &\leftarrow B_1(\bar{X}_1), \dots, B_n(\bar{X}_n) \\ r_{out} : \quad H(\bar{Y}) &\leftarrow \mathsf{fire}_r(\bar{X}) \end{aligned}$$

Thus P^F records, for each r-satisfying instance $\bar{x}$ of $\bar{X}$, a unique fact: $\mathsf{fire}_r(\bar{x})$.

Example. The transitive closure program P^r_{tc} from above is rewritten into 4 rules: Rule (1) is converted into two rules $\mathsf{fire}_1(X,Y) \leftarrow \mathsf{e}(X,Y)$ and $\mathsf{tc}(X,Y) \leftarrow \mathsf{fire}_1(X,Y)$. and rule (2) is transformed into $\mathsf{fire}_2(X,Y,Z) \leftarrow \mathsf{e}(X,Y), \mathsf{tc}(Y,Z)$ and $\mathsf{tc}(X,Z) \leftarrow \mathsf{fire}_2(X,\_,Z)$.

6.2.2 Graph Reification of Firings: $P^F \overset{G}{\rightsquigarrow} P^G$

To facilitate querying the results of the previous step, ground atoms and firings are *reified* as nodes in a labeled provenance graph G. For each pair of rules r_{in}, r_{out} above, n rules ($i = 1, \dots, n$) are added to generate the in-labeled edges in G:

$$\mathsf{g}(\ B_i(\bar{X}_i),\ \mathsf{in},\ \mathsf{fire}_r(\bar{X})\ ) \leftarrow \mathsf{fire}_r(\bar{X})$$

and one more rule for generating out-labeled edges in G as well:

$$\mathsf{g}(\ \mathsf{fire}_r(\bar{X}),\ \mathsf{out},\ H(\bar{Y})\ ) \leftarrow \mathsf{fire}_r(\bar{X})$$

Note the safe use of atoms as Skolem terms in the rule heads: for finitely many rule firings in $\mathsf{fire}_r(\bar{X})$, a finite number of in- and out-edges in G is obtained.

Example. After applying both transformations $\cdot \overset{F}{\leadsto} \cdot \overset{G}{\leadsto} \cdot$ to P_{tc} from above, the rewritten program P^G_{tc} can be executed, yielding a labeled graph with edges $\mathsf{g}(V_1, L, V_2)$ in the enriched model $\hat{M}$. Figure 6.2 shows a subgraph with two rule firings, both deriving the atom $\mathsf{tc(a,b)}$. Oval (yellow) nodes represent atoms A and boxed (blue) nodes represent firings F. Arrows with solid heads and label (*) are in-edges, while those with empty heads and label (+), represent out-edges. Note that according to the declarative semantics, in (out) edges, model logical conjunction "$\wedge$" (logical disjunction "$\vee$"), respectively. Thus, w.r.t. their incoming edges, boxed nodes are AND-nodes, while oval nodes are OR-nodes.[2]

6.2.3 Statelog Rewriting: $P^G \overset{S}{\leadsto} P^S$

Statelog [LLM98, Lud98] is a state-oriented Datalog extension for expressing active rules (updates) and declarative rules in a unifying framework. The next rewriting simulates a Statelog derivation in Datalog via a limited (safe) form of "state-generation". The key idea is to keep track of the firing rounds $I_{n+1} := T_P(I_n)$ of the T_P operator ($I_0 := I$ is the input database). This provides a simple yet powerful means to detect tuple re-derivations, to identify unfounded derivations (those whose goal atom A depends on itself in all derivations), etc.

First, replace all rules r_{in}, r_{out} above with their state-oriented counterparts:

$$\begin{aligned} r_{in} &: \mathsf{fire}_r(\mathsf{S1}, \bar{X}) \leftarrow B_1(\mathsf{S}, \bar{X}_1), \ldots, B_n(\mathsf{S}, \bar{X}_n),\ \mathsf{next}(\mathsf{S}, \mathsf{S1}). \\ r_{out} &: \quad H(\mathsf{S}, \bar{Y}) \leftarrow \mathsf{fire}_r(\mathsf{S}, \bar{X}). \end{aligned}$$

The goal $\mathsf{next}(\mathsf{S}, \mathsf{S1})$ is used for the safe generation of new states: The next state $s{+}1$ is generated only if in the previous state s at least one atom A was new:

$$\begin{aligned} \mathsf{next}(0, 1) &\leftarrow \mathsf{true}. \\ \mathsf{next}(\mathsf{S}, \mathsf{S1}) &\leftarrow \mathsf{next}(\_, \mathsf{S}),\ \mathsf{new}(\mathsf{S}, A),\ \mathsf{S1} := \mathsf{S} + 1. \end{aligned}$$

[2]In semiring parlance, they are product "$\otimes$" and sum "$\oplus$" nodes, respectively.

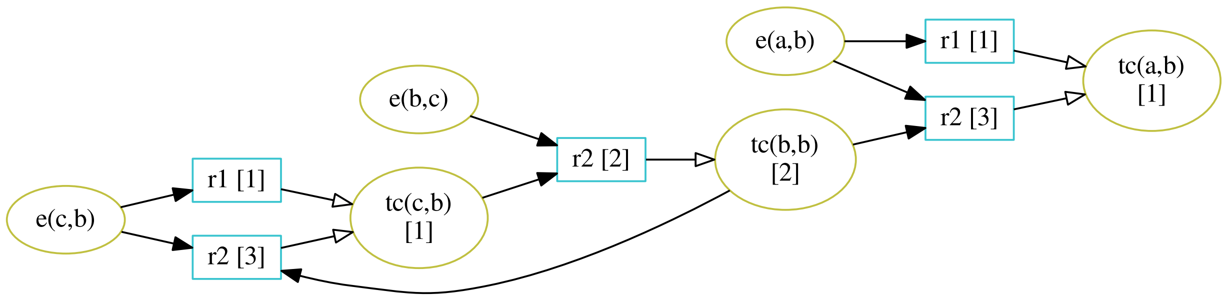


Figure 6.3: State-annotated provenance graph g for the derivation of $\mathsf{tc(a,b)}$. Annotations [in brackets] show the round (state number) in which an atom was first derived. To avoid clutter, firing nodes are often depicted without variable bindings.

An atom A is newly derived if it is true in $\mathsf{S1}$, but not in the previous state S:

$$\mathsf{newAtom}(\mathsf{S1}, A) \leftarrow \mathsf{next}(\mathsf{S}, \mathsf{S1}),\ \mathsf{g}(\mathsf{S1}, \_, \mathsf{out}, A),\ \neg\, \mathsf{g}(\mathsf{S}, \_, \mathsf{out}, A).$$

Similarly, rule firing F is new if it is true now (in $\mathsf{S1}$), but not previously in S:

$$\mathsf{newFiring}(\mathsf{S1}, F) \leftarrow \mathsf{next}(\mathsf{S}, \mathsf{S1}),\ \mathsf{g}(\mathsf{S1}, F, \mathsf{out}, \_),\ \neg\, \mathsf{g}(\mathsf{S}, F, \mathsf{out}, \_).$$

The n rules for generating in-edges are replaced with state-oriented versions:

$$\mathsf{g}(\ \mathsf{S},\ B_i(\bar{X}_i),\ \mathsf{in},\ \mathsf{fire}_r(\bar{X})\ ) \leftarrow \mathsf{fire}_r(\mathsf{S},\ \bar{X})$$

and similarly, for the out-edge generating rules:

$$\mathsf{g}(\ \mathsf{S},\ \mathsf{fire}_r(\bar{X}),\ \mathsf{out},\ H(\bar{Y})\ ) \leftarrow \mathsf{fire}_r(\mathsf{S},\ \bar{X}).$$

It is not difficult to see that the above rules are state-stratified (a form of local stratification) and that the resulting program terminates after polynomially many steps [Lud98]: When no more new atoms (or firings) are derived in a state, then the above rules for next can no longer generate new states, thus in turn preventing rules of type r_{in} from generating new $\mathsf{fire}_r(\mathsf{S1}, \bar{X})$ atoms.

Example. When applying the transformations $\cdot \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \cdot$ to the transitive closure program P^r_{tc}, a 4-ary graph g is created, with the additional T_p-round counter in the first (state) argument position. Figure 6.3 shows the graphical representation of `g` for our running example (observe the cycle in `g`, caused by the cycle in the input `e`).

6.3 Debugging and Profiling using Provenance Graphs

When debugging and profiling Datalog programs typically all program transformations $P \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \hat{P}$ are employed, i.e., the enriched model $\hat{M}$ contains the full provenance graph relation `g` with state annotations.

6.3.1 Debugging Declarative Rules

If a Datalog program does not compute the expected model, it is very helpful to understand how the model was derived, focusing in particular on certain goal atoms during the debugging process. Since relation `g` captures all possible derivations of the given program, various views can be defined on `g` to support debugging. This section presents views for some of the typical questions that arise during a debugging session.

Provenance Graph. The complete description of how a program was evaluated can be derived by just visualizing the whole provenance graph `g`, optionally removing the state argument through projection:

```
ProvGraph(X,L,Y) :- g(_,X,L,Y).
```

Figure 6.1c shows the provenance graph for a transitive closure computation. One can easily see that all transitive edges were derived and, by following the edges backwards, on which EDB facts each edge depends.

Provenance Views. Since provenance graphs are large in practice, it is often desirable to just visualize subgraphs of interest. The following debug view returns all "upstream" edges, i.e., the provenance subgraph relevant for debug atom Q:

```
ProvView(Q,X,out,Q) :- g(_,X,out,Q).
ProvView(Q,X,L,Y) :- ProvView(Q,Y,_,_), g(_,X,L,Y).
```

Figure 6.3 shows the result of this view for the debug goal $Q = \mathsf{tc}(\mathsf{a},\mathsf{b})$; the large (goal-irrelevant) remainder of the graph is excluded. Figure 6.1c, in contrast, shows the same query but now in the context of the whole provenance graph.

Computing the Length of Derivations. A typical question during debugging is when and from which other facts a debug goal was derived. Such temporal questions can be explained using the state rewriting of the program. We annotate atoms and firings with a *length* attribute to record in which round they were first derived. The length is defined as follows:

$$\mathsf{len}(F) = 1 + \max\{\ \mathsf{len}(A) \mid (A \overset{\mathsf{in}}{\to} F) \in \mathsf{g}\ \}\ ; \quad \text{if } F \text{ is a firing node}$$

$$\mathsf{len}(A) = \begin{cases} \min\{\ \mathsf{len}(F) \mid (F \overset{\mathsf{out}}{\to} A) \in \mathsf{g}\ \} & ; \quad \text{if } A \text{ is an IDB atom} \\ 0 & ; \quad \text{if } A \text{ is an EDB atom} \end{cases}$$

A rule firing F can only succeed one round after the *last* body atom (i.e., having maximal length) has been derived. Conversely, the length of an atom A is determined by its *first* derivation (i.e., having minimal length). The Statelog rewriting captures evaluation rounds, so the state associated with a *new* firing determines the length of the firing:

```
len(F,LenF) :− newFiring(S,F), LenF=S.
```

Similarly, the length of an atom is equal to the first round it was derived:

```
len(A,LenA) :− newAtom(S,A), LenA=S.
```

Figure 6.4 shows a provenance graph with such length annotations.

Customized Queries. In addition to the queries presented here, the provenance graph g can be used to answer various user defined queries. For example, one might be interested in common facts and rules in the provenance graph that a set of unexpected facts depends on. This query can be computed as an intersection of multiple provenance queries defined above.

6.3.2 Logic-Based Profiling

There are multiple ways to write a Datalog program that computes a desired query result and the performance of these programs may vary significantly. For example, consider an EDB with a linear graph e having 10 nodes. In addition to the right-recursive program P^r_{tc}, consider the doubly-recursive variant P^d_{tc} with the rules: (1) $\mathsf{tc}(X, Y) \leftarrow \mathsf{e}(X, Y)$ and (2) $\mathsf{tc}(X, Y) \leftarrow \mathsf{tc}(X, Z), \mathsf{tc}(Z, Y)$. When computing tc, the two programs perform differently and the cause should be identified. This section presents queries for profiling measures of Datalog programs that can help to answer such questions.

Counting Facts. When evaluating a Datalog program on an input EDB, a number of IDB atoms are derived. It is assumed here that the resulting model only contains desired facts, i.e., the program was already debugged with the methods described earlier. The number of derived IDB atoms can now be used as a baseline for profiling a program. This number can be computed easily via aggregation:

```
DerivedFact(H) :− g(_,out,H).
DerivedHeadCount(C) :− C = count{ H : DerivedFact(H) }.
```

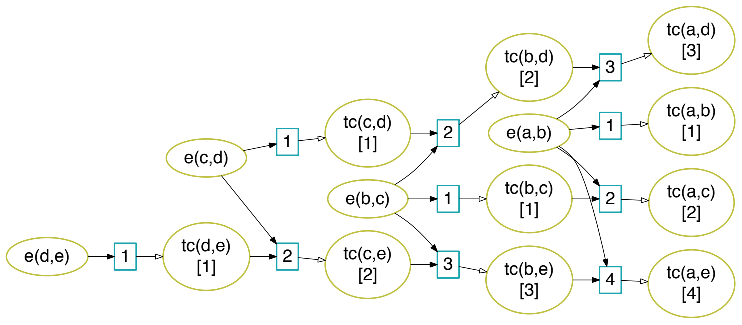
Both P^r_{tc} and P^d_{tc} derive 45 facts for our small graph example, which is exactly the number of transitive edges in the graph.

Counting Firings. An important measure in declarative profiling is the number of rule firings needed to produce the final model. It can be computed from the out-edges and another simple aggregation:

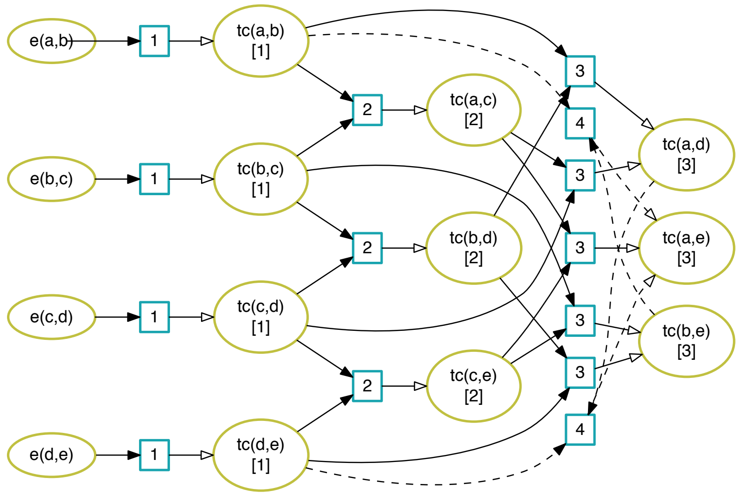
```
Firing(F) :− g(_,F,out,_).
FiringCount(C) :− C = count{ F : Firing(F) }.
```

This measure exposes a clear difference between the two variants of the transitive closure program. While the right-recursive program P^r_{tc} uses 45 rule firings to compute the model, the doubly-recursive variant P^d_{tc} causes 129 rule firings to derive the same 45 transitive edges. The reason is that P^d_{tc} will use all combinations of edges to derive a fact, while P^r_{tc} extends paths only in one direction, one edge at a time.

For better readability, Figure 6.4 shows the provenance graph for a smaller input graph consisting of a 5-node linear chain. Nodes annotated with their length, i.e., earliest possible derivation round. Note, how some atom nodes in the graph for P^d_{tc} in Figure 6.4b have more



(a) P^r_{tc}:right-recursive



(b) P^d_{tc}:doubly-recursive

Figure 6.4: Provenance graphs with annotations for profiling P^r_{tc} and P^d_{tc} on a 5-node linear graph. P^d_{tc} causes more rule firings than P^r_{tc} and also derives facts in multiple ways. Numbers denote $\mathsf{len}(F)$ (in firing nodes) and $\mathsf{len}(A)$ (in atom nodes), respectively.

incoming edges (and derivations), than the corresponding nodes in the P^r_{tc} variant shown in Figure 6.4a.

Computing the Maximum Round. Another measure is the number of states (T_P rounds), needed to derive all conclusions. The state final state created by the rewriting P^S can easily be determined:

MaxRound(MR) **:–** MR = max**{** S **:** g(S,_,_,_) **}.**

This measure shows another clear difference between P^r_{tc} and P^d_{tc}: While P^r_{tc} requires 10 rounds to compute all transitive edges in our sample graph, P^d_{tc} only needs 6 rounds. Generally, the doubly-recursive variant requires significantly fewer rounds, i.e., logarithmic in the size of the longest simple path in the graph versus linear for the right-recursive implementation.

Counting Rederivations. To analyze the number of derivations in more detail, the Statelog rewriting P^S can be used to capture temporal aspects. With each application of the T_P operator, some facts might be rederived. Note that, if a fact is derived via different variable bindings in the body of a rule (or different rules), the re-derivation is captured already in firings. However, re-derivations occurring repeatedly until a fixpoint was reached can only be captured using the Statelog rewriting:

```
ReDerivation(S,F) :− g(S,F,out,A), len(A,LenA), LenA < S.
ReDerivationCount(S,C) :− C = count{ F : ReDerivation(S,F) }.
ReDerivationTotal(T) :− T = sum{ C : ReDerivationCount(S,C) }.
```

When comparing the re-derivation counts, the difference between the P_{tc} variants becomes even clearer. P^r_{tc} rederives facts 285 times until the fixpoint is reached. The double-recursive program P^d_{tc} causes 325 re-derivations.

Schema-Level Profiling. The number of facts per *relation* that are used in each round to derive new facts can be determined easily:

```
FactsInRound(S,R,A) :− g(S,A,in,_), RelationName(A,R).
FactsInRound(S1,R,A) :− g(S,_,out,A), next(S,S1), RelationName(A,R).
NewFacts(S,R,A) :− g(S,_,out,A), !FactsInRound(S,R,A), RelationName(A,R).
NewFactsCount(S,R,C) :− C = count{ A : NewFacts(S,R,A) }.
```

This measure can be used to "plot" the temporal evolution for our result relation $\mathsf{R} = \mathsf{tc}$:

For P^r_{tc} and $\mathsf{S} = 1, \dots, 9$ the counts are $\mathsf{C} = 9, 8, 7, 6, 5, 4, 3, 2, 1$, while for P^r_{tc} and $\mathsf{S} = 1, \dots, 5$ the numbers are $\mathsf{C} = 9, 8, 13, 14, 1$. As expected, P^d_{tc} requires fewer rounds than P^r_{tc}, but the number of new facts increases over time, while for P^r_{tc}, the sequence is decreasing.

Confronting the Real-World. In practical implementations, the doubly-recursive version P^d_{tc} has horrible performance. For a representative, realistic graph[3] with 1710 nodes and 3936 edges,

[3]The specifics are secondary to our argument, but are listed for completeness. The graph is the application-level call-graph (i.e., a graph with the application's methods as nodes and edges indicating whether a method can call another) for the `pmd` program from the DaCapo benchmark suite, as produced by a precise low-level program

the right-recursive P^r_{tc} runs in 2.6 sec, while the doubly-recursive P^d_{tc} takes 15.4 sec. The metrics described above can easily explain the discrepancy. The tc-fact count for both versions is 304,000, but the rule firing count varies widely. Using the program rewriting and the profiling view FiringCount(C), it can be discovered that P^d_{tc} has over 64 million different rule firings, while P^r_{tc} has under 566 thousand. One reason is that the doubly-recursive rule $\mathsf{tc}(X,Y) \leftarrow \mathsf{tc}(X,Z), \mathsf{tc}(Z,Y)$ derives the same $\mathsf{tc}(X,Y)$ fact many times over. This practical example also illustrates the burden of declarative debugging. Adding the profiling calculation to the right-recursive version, P^r_{tc} only grows the running time slightly, to 3 sec. Adding it to the doubly-recursive P^d_{tc} however, makes the running time 51.3 sec. This is due to the cost of storing the over-64-million combinations of variables on the right hand side of a rule firing.

6.4 Gpad Prototype Implementation

By design, the method of provenance-based debugging and profiling only relies on the declarative reading of rules, i.e., is agnostic about implementation details or evaluation techniques specific to the underlying Datalog engine. Indeed, parallel with the development of the method, two incarnations of a ***G****raph-based* ***P****rovenance* ***A****nalyzer and* ***D****ebugger* were developed, i.e., prototypes Gpad/dlv and Gpad/lb, for declarative debugging with the DLV [LPF+06] and LogicBlox [MHB+10] engines, respectively. Both prototypes "wrap" the underlying Datalog engine, and outsource some processing aspects to a host language.

For example, Gpad/dlv uses Swi-Prolog [WSTL10] as a "glue" to automate (1) rule rewritings, (2) invocation of DLV, followed by (3) result post-processing, and (4) result visualization using Graphviz.

6.5 Related Work

Work on declarative debugging, in particular in the form of *algorithmic debugging* goes as far back as the 1980's [Sha82, DNT89]. Algorithmic debugging is an interactive process where

analysis (a 2-object-sensitive with context-sensitive heap points-to analysis). Timings are on a quad-core Xeon E5530 2.4GHz 64-bit machine (only one thread was active at a time) with plentiful RAM (24GB) for the analysis, using LogicBlox Datalog ver. 3.7.10.

the user is asked to differentiate between the actual model of the (presumably buggy) program and the user's intended model. Based on the user's input, the system then tries to locate the faulty rules in an interactive session. The approach presented above differs in a number of aspects. First, algorithmic debugging is usually based on a specific operational semantics, i.e., SLDNF resolution, a top-down, left-to-right strategy with backtracking and negation-as-failure, which differs significantly from the declarative Datalog semantics. Moreover, while algorithmic debugging is applicable, in principle, in an interactive way, this suggests a tighter coupling between the debugger and the underlying rule engine. In contrast, the approach presented above and its GPAD implementations do not require such tight coupling, but instead treat the rule engine as a black box. In this way, debugging becomes a post-mortem analysis of the provenance-enriched model $\hat{M} = \hat{P}(I)$ via simple yet powerful graph queries and aggregations.

Another approach, more closely related to GPAD, is the Datalog debugger [CGRSP08], developed for the DES system. Unlike prior work, and similar to GPAD, they do not view derivations as SLD proof trees, but rather use a *computation graph*, similar to the labeled provenance graph presented here.

GPAD differs in a number of ways, e.g., reification of derivations in a labeled graph allows the use of regular path queries to navigate the provenance graph, locate (least) common ancestors of buggy atoms, etc. Another difference is the use of Statelog for keeping track of derivation rounds, which facilitates profiling of the model computation over time (per firing round, identify the rules fired, the number of (re-)derivations per atom or relation, etc.) Recent related work also includes work on trace visualization for ASP [CLRV09], step-by-step execution of ASP programs [OPT11], and an integrated debugging environment for DLV [PRT+07].

Debugging and Provenance. Chiticariu et al. [CT06] present a tool for debugging database schema mappings. They focus on the computation of derivation routes from source facts to a target. The method includes the computation of minimal routes, similar to shortest derivations in the graphs generated by GPAD. However, their approach seems less conducive to profiling since, e.g., provenance information on firing rounds is not available in their approach.

There is an intriguingly close relationship between *provenance semirings*, i.e., provenance polynomials and formal power series [GKT07], and the labeled provenance graphs G. The semiring provenance of atom A is represented in the structure of G_A. Consider, e.g., Figure 6.2: the in-edges of rule firings correspond to a logical conjunction "$\wedge$", or more abstractly, the product operator "$\otimes$" of the semiring. Similarly, out-edges represent a disjunction "$\vee$", i.e., an abstract sum operator "$\oplus$", mirroring the fact that atoms in general have multiple derivations. It is easy to see that a fact A has an infinite number of derivations (proof trees) iff there is a cycle in G_A: e.g., the derivation of $A = \mathsf{tc}(\mathsf{a}, \mathsf{b})$ in Figures 6.1 and 6.2 involves a cycle through tc(b,b), tc(c,b), via two firings of r_2. This also explains Prolog's non-termination, which "nicely" mirrors the fact that there are infinitely many proof trees. On the other hand, such cycles are not problematic in the original Datalog evaluation of $M = P(I)$ or in the extended provenance model presented here $\hat{M} = \hat{P}(I)$, both of which can be shown to converge in polynomial time.

6.6 Summary

In this chapter, a framework for declarative debugging and profiling of Datalog programs was presented. The key idea is to rewrite a program P into $\hat{P}$, which records the derivation history of $M = P(I)$ in an extended model $\hat{M} = \hat{P}(I)$. $\hat{P}$ is obtained from three simple rewritings for (1) recording rule firings, (2) reifying those into a labeled graph, while (3) keeping track of derivation rounds in the style of Statelog. After the rewritten program is evaluated, the resulting provenance graph can be queried and visualized for debugging and profiling purposes.

The declarative profiling approach was illustrated by analyzing different, logically equivalent versions of the transitive closure program P_{tc}. The measures obtained through logic profiling correlate with runtime measures for a large, real-world example. Two prototypical systems GPAD/DLV and GPAD/LB are implemented, for DLV and LogicBlox, respectively. The presented approach is designed for positive Datalog only. However, it is not difficult to see how it can be extended, e.g., for well-founded Datalog. Indeed, the GPAD prototypes already support the handling of well-founded negation through a simple Statelog encoding [LLM98, Lud98]. The following chapter will present an extension of the GPAD approach that supports non-recursive Datalog$^\neg$ programs and also provides detailed explanations of negated atoms or of missing output tuples.

Chapter 7

First-Order Provenance Games

As argued in Chapter 5, provenance of a workflow execution has many applications. When using Datalog as a workflow description language, provenance of a Datalog program evaluation must be collected and analyzed. To this extend, this chapter presents an approach to record, understand and visualize provenance of a Datalog program using game theory, which was developed by Koehler et al. and appeared in [KLZ13]. This approach is an extension of the GPAD approach presented in the last chapter.

7.1 Introduction

A number of provenance models have been developed in recent years that aim at explaining why and how tuples in a query result $Q(D)$ are related to tuples in the input database D (see [CCT09, KG12] for recent surveys). Motivated by applications in data warehousing, Cui *et al.* [CWW00] defined a notion of data *lineage* to trace backward which tuples in D contributed to the result. Buneman *et al.* [BKT01] refined and formalized new forms of *why*- and *where*-provenance, and introduced a notion of (minimal) witness basis to do so. Later, Green *et al.* [GKT07] proposed a form of *how*-provenance through *provenance semirings* that emerged as an elegant, unifying framework for provenance. For $\mathcal{RA}^+$ (positive relational algebra) queries, provenance semirings form a hierarchy [Gre11], with *provenance polynomials* $\mathbb{N}[X]$ as the most informative semiring at the top (i.e., providing the most detailed account *how* a result was derived), and other semirings with "coarser" provenance information below, e.g., *Boolean provenance polynomials* $\mathbb{B}[X]$ [Gre11], *Trio* provenance [BSHW06], *why*-provenance [BKT01], and *lineage* [CWW00]. The key idea of the unifying framework is to annotate each tuple in the input database D with an element from a semiring K and then propagate K-annotations through query evaluation. Semiring-style provenance support has been added to practical systems, e.g., ORCHESTRA [GKIT07] and LOGICBLOX [HGL11]. However, the semiring approach does not extend easily to negation and other non-monotonic constructs, thus spawning further research [GP10, GIT11, ADT11a, ADT11b].

This chapter presents a fresh look at provenance by employing *games*. Game theory has a long history and many applications, e.g., in logic, computer science, biology, and economics. The first formal theorem in the theory of games was published by Ernst Zermelo exactly 100 years ago [Zer13].[1] In 1928, von Neumann's paper "*Zur Theorie der Gesellschaftsspiele*" [vN28] marked the beginning of game theory as a field. In it he asks (and answers) the question of how a player should move to achieve a good outcome. The approach presented here employs such "good" moves to define a natural notion of provenance for games G, which is called *game provenance* $\Gamma\,(=\Gamma_G)$, and which is thus closely related to *winning strategies*. The crux is that by considering only "good" moves while ignoring "bad" ones, one can get a game-theoretic explanation for why a position is won, lost, or drawn. By viewing query evaluation as a game, game provenance can be applied to obtain an elegant new provenance approach, which is called *provenance games*.

Game Plan. Section 7.2 introduces basic concepts and terminology for games G and shows how to solve them using a form of backward induction. Then the regular structure inherent in solved games G^γ is discussed and is used to define our notion of game provenance Γ. The solved positions imply a labeling of moves as "good" or "bad", which is used to define the game provenance $\Gamma(x)$ of position x as the subgraph of G, reachable from x without "bad" moves. The value of a position is determined by its game provenance, and it captures why and how a position is won, lost, or drawn.

Section 7.3 describes how game provenance is applied to first-order (FO) queries in Datalog$^\neg$ form, by viewing the evaluation of query Q on database D as a game $G_{Q,D}$. By construction, the *provenance games* yield the standard semantics for FO queries. For positive relational queries

[1]Some confusion prevails about Zermelo's theorem, but it is all sorted out in [SW01].

$\mathcal{RA}^+$, game provenance $\Gamma_{Q,D}$ is equivalent to the most general semiring of provenance polynomials $\mathbb{N}[X]$. Variations of the provenance game yield other semirings, e.g., $\mathsf{Trio}(X)$. While provenance games are equivalent to provenance semirings for positive queries, the former also handle negation seamlessly, as complementary claims and negation are inherent in games. Provenance games can thus also answer *why-not* questions easily: The explanation for why x is *not* won is the same as why x is lost (or drawn, for games that are not draw-free). Since provenance games are always draw-free for first-order queries, a simple and elegant provenance model for FO that combines how-provenance and why-not provenance can be obtained. Finally, Section 7.6 concludes this chapter and suggest some future work in this field.

7.2 A Game on Graphs

Here, Games are seen as graphs $G = (V, M)$, where two players move alternately between *positions* V along the edges (*moves*) $M \subseteq V \times V$. It is assumed that G is finite, i.e., $|V| < \infty$,[2] but game graphs can have cycles and thus may result in infinite plays. Each $v_0 \in V$ defines a game $G^{v_0} = (V, M, v_0)$ starting at position v_0.

A *play* π $(= \pi_{v_0})$ of G^{v_0} is a (finite or infinite) sequence of edges from M:

$$v_0 \overset{M}{\to} v_1 \overset{M}{\to} v_2 \overset{M}{\to} \cdots \tag{π}$$

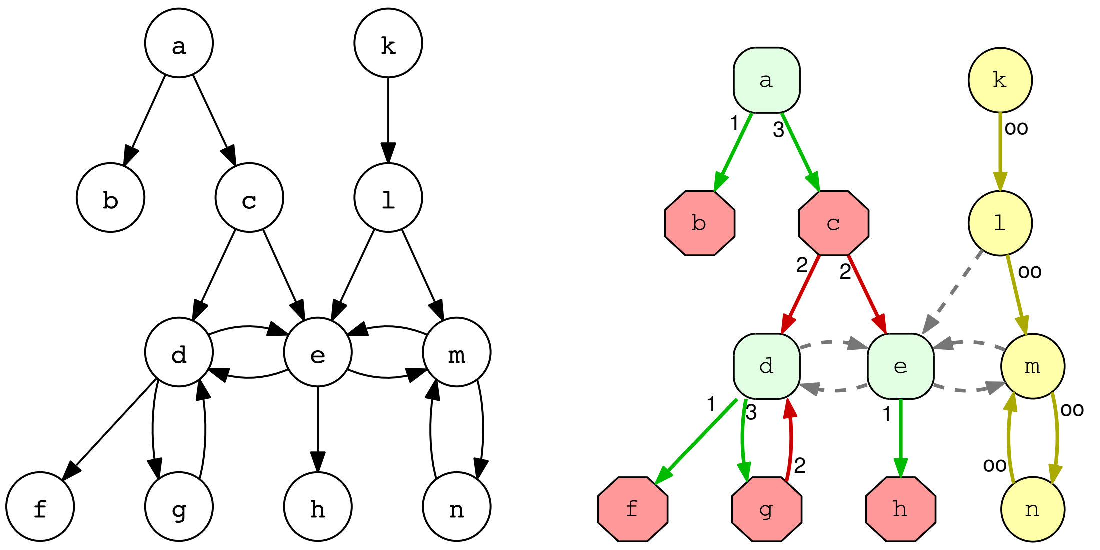
i.e., where for all $i = 0, 1, 2, \ldots$ the edge $v_i \overset{M}{\to} v_{i+1}$ is a move $(v_i, v_{i+1}) \in M$. A play π is *complete*, either if it is infinite, or if it ends after $n = |\pi|$ moves in a sink of the game graph. The player who cannot move loses the play π, while the previous player (who made the last possible move) wins π. Thus, if $|\pi| = 2k + 1$: $\pi =$

$$v_0 \overset{\text{I}}{\to} v_1 \overset{\text{II}}{\to} v_2 \overset{\text{I}}{\to} \cdots \overset{\text{II}}{\to} v_{2k} \overset{\text{I}}{\to} v_{2k+1} \tag{I moves last}$$

and π is *won* for I. Conversely, if II moves last, then $|\pi| = 2k$ for some $\pi =$

$$v_0 \overset{\text{I}}{\to} v_1 \overset{\text{II}}{\to} v_2 \overset{\text{I}}{\to} \cdots \overset{\text{II}}{\to} v_{2k} \tag{II moves last}$$

[2]Many game-theoretic notions and results carry over to the transfinite case; cf. [Flu00].



(a) What are the "good moves", e.g., in position e? Is e won (or lost, or drawn), and if so how?

(b) The solved game reveals the answer: move e→h is winning; the moves e→d and e→m are not.

Figure 7.1: Position values in G (left) are revealed by the solved game $G^\gamma = (V, M, \gamma)$ on the right: positions are *won* (green boxes), *lost* (red octagons), or *drawn* (yellow circles). This separates "good" moves (solid, colored arcs) from "bad" ones (dashed, gray). The length ℓ of a move $x \xrightarrow{\ell} y$ indicates how quickly one can force a win, or how long one can delay a loss, using that move.

so π is *lost* for I, and II wins the play. A play π of infinite length is a *draw* (in *finite* games G, this means that M must have a cycle).

Example. Consider $G = (V, M)$ in Figure 7.1a and a start position for player I, say e. In the play $\pi_1 = \mathtt{e} \xrightarrow{\text{I}} \mathtt{d} \xrightarrow{\text{II}} \mathtt{f}$, I cannot move, so π_1 is lost (for I). However, in $\pi_2 = \mathtt{e} \xrightarrow{\text{I}} \mathtt{h}$, II cannot move, so π_2 is won (for I). So from position e, the best move is e→h; the other moves are "bad": e→d loses (see π_1), while e→m only draws (if II sticks to m→n).

The Value of a Position: Playing Optimally. To determine the true value of $v \in V$, plays with bad moves are not of interest, but instead those plays are considered, where the opponents play optimally, or at least "good enough" so that the best possible outcome is guaranteed. This leads to the question: can I force a win from $v \in V$ (no matter what II does), or can II force I to lose from v? If neither player can force a win, v is a *draw* and both players can avoid losing by forcing an infinite play. This is formalized using strategies.

A (pure) *strategy* is a partial function $S : V \to V$ with $S \subseteq M$. It prescribes which of the available moves a player will choose in a position v.[3] We define v_0 to be *won for player* I in (at most) n moves, if there is a strategy S_{I} for I, such that for all strategies S_{II} of II, there is a number $j = 2k+1 \leq n$ such that $v_j = S_{\mathrm{I}} \circ (S_{\mathrm{II}} \circ S_{\mathrm{I}})^k(v_0)$ is defined, but $S_{\mathrm{II}}(v_j)$ is not: II cannot move. In this case, S_{I} is a *winning strategy* for I at v_0. Conversely, v_0 is *won for player* II in (at most) n moves, if there is a strategy S_{II}, such that for all strategies S_{I}, there is a number $j = 2k \leq n$ such that $v_j = (S_{\mathrm{II}} \circ S_{\mathrm{I}})^k(v_0)$ is defined, but $S_{\mathrm{I}}(v_j)$ is not: I cannot move. With this, the *value* of v_0 is *won* (*lost*) if it is won for player I (player II). If v_0 is neither won nor lost, its value is *drawn*, so neither I nor II can force a win from v_0, but both can avoid losing via an infinite play.

7.2.1 Solving Games: Labeling Nodes (Positions)

Let $G = (V, M)$ be the game in Figure 7.1a. How can G be *solved*, i.e., determined whether the value of $x \in V$ is won, lost, or drawn? The value of x represent using a node labeling $\gamma : V \to \{\mathsf{W}, \mathsf{L}, \mathsf{D}\}$ and $G^\gamma = (V, M, \gamma)$ denotes a solved game.

The following Datalog$^\neg$ query, consisting of a single rule, solves games:

$$\mathtt{win}(X) :- \mathtt{move}(X, Y), \neg\mathtt{win}(Y) \tag{Q_G}$$

Q_G says that position x is won in G if there is a move to position y, where y is not won. For non-stratified Datalog$^\neg$ programs like Q_G (having recursion through negation), the three-valued *well-founded model* $\mathcal{W}$ [VGRS91] provides the desired answer:

Proposition 1 (Q_G Solves Games) *Let $P := (Q_G \cup \mathtt{move})$ be the Datalog$^\neg$ query Q_G plus finitely many "$\mathtt{move}$" facts, representing a game $G = (V, M)$. For all $x \in V$:*

$$\mathcal{W}_P(\,\mathtt{win}(x)\,) = \left\{ \begin{array}{c} \mathsf{true} \\ \mathsf{false} \\ \mathsf{undef} \end{array} \right\} \quad \Leftrightarrow \quad \gamma(x) = \left\{ \begin{array}{c} \mathsf{W} \\ \mathsf{L} \\ \mathsf{D} \end{array} \right\} .$$

[3] In our games, the same positions can be revisited many times. Accordingly, strategies are based on the current position v only and do not take into account how one arrived at v.

When implemented via an alternating fixpoint [VG93], one obtains an increasing sequence of underestimates $U_1 \subseteq U_2 \subseteq \ldots$ converging to the true atoms U^ω from below, and a decreasing sequence of overestimates $O_1 \supseteq O_2 \supseteq \ldots$ converging to O^ω, the union of true or undefined atoms from above. Any remaining atoms in the "gap" have the third truth-value (`undef`). For the game query Q_G above, U^ω contains the won positions V^{W}; the "gap" (if any) $O^\omega \setminus U^\omega$ contains the drawn positions V^{D}; and the atoms in the complement of O^ω (i.e., which are neither true nor undefined) are the lost positions V^{L}.

To solve G directly, consider, e.g., the three moves $\mathtt{e}\to\mathtt{d}$, $\mathtt{e}\to\mathtt{h}$, and $\mathtt{e}\to\mathtt{m}$ in Figure 7.1a. The move $\mathtt{e}\to\mathtt{h}$ is clearly winning, as it forces the opponent into a sink. However, the status of the moves $\mathtt{e}\to\mathtt{d}$ and $\mathtt{e}\to\mathtt{m}$ is unclear unless the game has been solved. Figure 7.1b depicts the solved game G^γ. The set of positions is a disjoint union $V = V^{\mathsf{W}} \mathbin{\dot\cup} V^{\mathsf{L}} \mathbin{\dot\cup} V^{\mathsf{D}}$.

To obtain G^γ, proceed as follows: First, find all *sinks* x, i.e., nodes for which the set of *followers* $\mathsf{F}(x) = \{y \mid (x, y) \in M\}$ is empty. These positions are immediately lost and colored red: $V_0^{\mathsf{L}} = \{x \in V \mid \mathsf{F}(x) = \emptyset\}$. In our example, $V_0^{\mathsf{L}} = \{\mathtt{b}, \mathtt{f}, \mathtt{h}\}$. Then, find all nodes x for which there is *some* y with $(x, y) \in M$ such that $y \in V_0^{\mathsf{L}}$. These positions are won and colored green; here: $V_1^{\mathsf{W}} = \{\mathtt{a}, \mathtt{d}, \mathtt{e}\}$. Then, find the unlabeled nodes x for which *all* followers $y \in \mathsf{F}(x)$ are already won (i.e., colored green). Since the player moving from that position can only move to a position that is won for the opponent, those x are also *lost* and added to V_2^{L}. In the running example $V_2^{\mathsf{L}} = \{\mathtt{c}, \mathtt{g}\}$. Now iterate the above steps until there is no more change. One can show that $V_1^{\mathsf{W}} \subseteq V_3^{\mathsf{W}} \subseteq V_5^{\mathsf{W}} \cdots$ converges to the won positions V^{W}, whereas $V_0^{\mathsf{L}} \subseteq V_2^{\mathsf{L}} \subseteq V_4^{\mathsf{L}} \cdots$ converges to the lost positions V^{L}; the drawn positions are $V^{\mathsf{D}} := V \setminus (V^{\mathsf{W}} \cup V^{\mathsf{L}})$.

Algorithm 1 depicts the details of a simple, round-based approach to solve games. It also computes the *length* of a position, which adds further information to a solved game G^γ, i.e., how quickly one can win (starting from green nodes), or how long one can delay losing (starting from red nodes). In Figure 7.1, the (delay) length of `f` is 0, since `f` is a sink and no move is possible. In contrast, the (win) length of `d` is 1: the next player moving wins by moving to `f`. For `g`, the (delay) length is 2, since the player can move to `d`, but the opponent can then move to `f`. So `g` is lost in 2 moves.

Remark. As described, Algorithm 1 proceeds in *rounds* to determine the value of positions, i.e.,

Algorithm 1: Solve game $G^\gamma = (V, M, \gamma)$

```
$V^{\mathsf{W}} := \emptyset$ ;                                   // Initially no won positions are known
$V^{\mathsf{L}} := \{x \in V \mid \mathsf{F}(x) = \emptyset\}$ ;                    // ... but all sinks are lost ...
$\mathrm{len}(x) := 0$ for all $x \in V^{\mathsf{L}}$ ;                    // ... immediately: their length is 0.
repeat
    for $x \in V \setminus (V^{\mathsf{W}} \cup V^{\mathsf{L}})$ do
        $F^{\mathsf{L}} := \mathsf{F}(x) \cap V^{\mathsf{L}}$;   $F^{\mathsf{W}} := \mathsf{F}(x) \cap V^{\mathsf{W}}$ ;
        if $F^{\mathsf{L}} \neq \emptyset$ then
            $V^{\mathsf{W}} := V^{\mathsf{W}} \cup \{x\}$ ;             // some $y \in \mathsf{F}(x)$ is lost, so $x$ is won
            $\mathrm{len}(x) := 1 + \min\{\mathrm{len}(y) \mid y \in F^{\mathsf{L}}\}$ ;               // shortest win
        if $\mathsf{F}(x) = F^{\mathsf{W}}$ then
            $V^{\mathsf{L}} := V^{\mathsf{L}} \cup \{x\}$ ;             // all $y \in \mathsf{F}(x)$ are won, so $x$ is lost
            $\mathrm{len}(x) := 1 + \max\{\mathrm{len}(y) \mid y \in F^{\mathsf{W}}\}$ ;              // longest delay
until $V^{\mathsf{W}}$ and $V^{\mathsf{L}}$ change no more;
$V^{\mathsf{D}} := V \setminus (V^{\mathsf{W}} \cup V^{\mathsf{L}})$ ;                    // remaining positions are now draws
$\mathrm{len}(x) := \infty$ for all $x \in V^{\mathsf{D}}$ ;                  // ... and can be delayed forever
$\gamma(x) := \mathsf{W}/\mathsf{L}/\mathsf{D}$ for all $x \in V^{\mathsf{W}}/V^{\mathsf{L}}/V^{\mathsf{D}}$, respectively.
```

in each round i, *all* newly won positions, and *all* newly lost positions are determined. This could be used, e.g., to simplify the computation of the length of a position ($\mathrm{len}(x)$ can be derived from the first round in which the value of x becomes known). On the other hand, this is not strictly necessary: one can replace the **for**-loop ranging over all unlabeled nodes by a non-deterministic **pick** of any unlabeled node. As long as nodes are picked in a fair manner, the non-deterministic version will also converge to the correct result, while allowing more flexibility during evaluation [ZGL12].

7.2.2 Game Provenance: Labeling Edges (Moves)

In order to answer the original question: why is $x \in V$ won, lost, or drawn, this section defines a suitable notion of *game provenance* $\Gamma(x)$ that is similar in spirit to the how-provenance devised for positive queries [GKT07], but that works for games and explains the value (won, lost, or drawn) of x. Some desiderata of game provenance are immediate: First, only nodes *reachable* from x can influence the outcome at x, i.e., only nodes and edges in the transitive closure $\mathsf{F}^+(x)$. Thus, one expects $\Gamma(x)$ to depend only on $\mathsf{F}^+(x)$. In addition, one expects the value $\gamma(x)$ of position x to be independent of "bad moves", i.e., which give the opponent a better outcome than necessary. A partial edge-labeling function λ is used to distinguish different types of moves.

Definition 1 (Edge Labels) Let $G^\gamma = (V, M, \gamma)$ be a solved game. The edge-labeling $\lambda : V \times V \to \{\mathsf{g}, \mathsf{r}, \mathsf{y}\}$ defines a color for a subset of edges from M as shown in Figure 7.2. $\square$

As in Figure 7.2, $\gamma(x)$ and $\gamma(y)$, i.e., node labels W, D, and L of moves $(x, y) \in M$ are used to derive an appropriate edge label. The edge labeling allows to distinguish provenance-relevant ("good") moves (*winning*, *drawing*, or *delaying*), from irrelevant (*bad*) moves. The latter are excluded from game provenance:

Definition 2 (Game Provenance) Let $G^\gamma = (V, M, \gamma)$ be a solved game. The *game provenance* $\Gamma(=\Gamma_G)$ is the λ-colored subgraph of G^γ. For $x \in V$, $\Gamma(x)$ is defined as the subgraph of Γ, reachable via λ edges. $\square$

Consider the solved game on the right in Figure 7.1. Since bad (dashed) edges are excluded, the game provenance consists of two disconnected subgraphs: (i) The bipartite "red-green" subgraph, which is draw-free, i.e., every position is either won or lost, and (ii) the "yellow" subgraph, representing the drawn positions.

The figure also reveals that solved games G^γ and thus game provenance Γ have a nice, regular structure. The following is immediate from the underlying game-theoretic semantics of G.

Theorem 1 (Provenance Structure) *Let $G^\gamma = (G, M, \gamma)$ be a solved game, Γ its edge-labeled provenance graph. The game provenance Γ has a regular structure:*

$$\Gamma(x) = \begin{cases} M_{\mathsf{g}.(\mathsf{r}.\mathsf{g})^*}(x) & ;\ \textit{if } x \textit{ is won} \\ M_{(\mathsf{r}.\mathsf{g})^*}(x) & ;\ \textit{if } x \textit{ is lost} \\ M_{\mathsf{y}^+}(x) & ;\ \textit{if } x \textit{ is drawn} \end{cases}$$

Here, for a regular expression R, and a node $x \in V$, the expression $M_R(x)$ denotes a subset of labeled edges of M, i.e., for which there is a path π in Γ whose labels match the expression R. As will be shown below, for positive queries, the bipartite structure of won and lost nodes nicely corresponds to the structure of provenance polynomials [KG12].

	y ***won*** (W)	y ***drawn*** (D)	y ***lost*** (L)
x ***won*** (W)	*bad*	*bad*	g: ***winning***
x ***drawn*** (D)	*bad*	y: ***drawing***	*n/a*
x ***lost*** (L)	r: ***delaying***	*n/a*	*n/a*

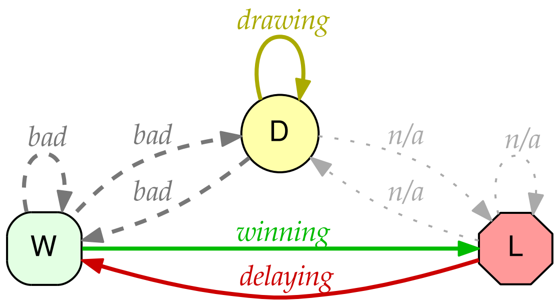


Figure 7.2: Depending on node labels, moves $x \to y$ are either *winning* (or *green*) ($\mathsf{W} \overset{\mathsf{g}}{\rightsquigarrow} \mathsf{L}$), *delaying* (or *red*) ($\mathsf{L} \overset{\mathsf{r}}{\rightsquigarrow} \mathsf{W}$), or *drawing* (or *yellow*) ($\mathsf{D} \overset{\mathsf{y}}{\rightsquigarrow} \mathsf{D}$). All other moves are either *bad* (allowing the opponent to improve the outcome), or non-existent (*n/a*): e.g., if x is lost, then there are only delaying moves (i.e., ending in won positions y for the opponent).

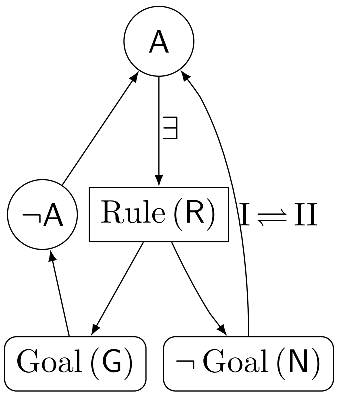
7.3 Provenance Games

The previous section showed that games can be viewed as *arguments* between players I and II. The game semantics (avoiding bad moves) yields a natural model of provenance. Now this notion is applied to queries expressed using non-recursive Datalog$^\neg$ rules. Any first-order query $\varphi(\bar{x})$ on input database D can be expressed as a non-recursive Datalog$^\neg$ program Q_φ with a distinguished relation $\texttt{ans} \in idb(Q_\varphi)$ [4] such that evaluating Q_φ with input D under the stratified semantics[5] agrees with the result of $\varphi(\bar{x})$. In the following,$Q(D)$ denotes the result of evaluating Q on input D.

7.3.1 Query Evaluation Games

Query evaluation of $Q(D)$ can be seen as a game between players I and II who argue whether an atom $A \in Q(D)$. This approach is similar to the ideas of Lorenzen et al. [LL78]. The argumentation structure is stylized in Figure 7.3. There are three classes of positions in the game as shown on the left of Figure 7.3:

[4]The arity of $\texttt{ans}$ matches that of $\varphi(\bar{x})$.
[5]which coincides with the well-founded semantics on non-recursive Datalog$^\neg$



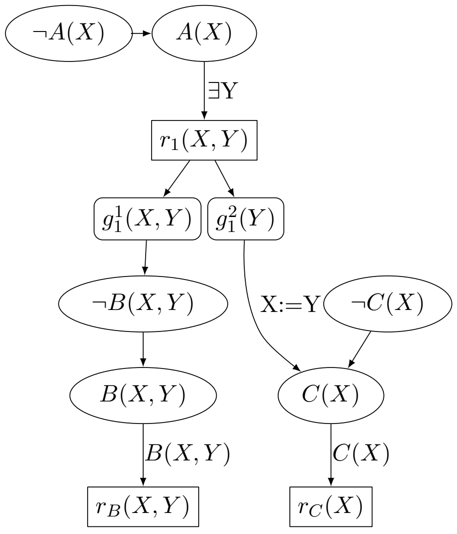
Move	**Claim made by making the move**
$\mathsf{A} \overset{\exists}{\rightsquigarrow} \mathsf{R}$	"A is true: it's the head of this instance of R."
$\mathsf{R} \rightsquigarrow \mathsf{G}$	"Positive goal $g_k(=A')$ in your rule body fails!"
$\mathsf{G} \rightsquigarrow \neg\mathsf{A}$	"No! Its negation $\neg A'$ fails and A' is true."
$\neg\mathsf{A} \rightsquigarrow \mathsf{A}$	"No: atom A' fails!"
$\mathsf{R} \rightsquigarrow \mathsf{N}$	"Negative goal $\neg A'$ in the rule body fails."
$\mathsf{N} \rightsquigarrow \mathsf{A}$	"No: $\neg A'$ succeeds, but A' fails."

Figure 7.3: Move types of the query evaluation game (left) and implicit claims made (right). Moving along an edge, a player aims to verify a claim, thereby refuting the opponent. Initially, player I is a verifier, trying to prove A, while II tries to spoil this attempt and refute it. Roles are swapped (I $\rightleftharpoons$ II) when moving through a negated goal ($\mathsf{R}\rightsquigarrow\mathsf{N}\rightsquigarrow\mathsf{A}$).

- Relation nodes—depicted as circles,
- Rule nodes—depicted as rectangles, and
- Goal nodes—depicted as rectangles with rounded corners.

Both relation nodes and goal nodes can be positive or negative.

Usually, an evaluation game starts with I claiming that a ground atom $A(x)$ is true. That is she starts the game in a relation node for A. To substantiate her claim she moves to a rule that has A as a head atom and specifies constants for the remaining existentially quantified variables in the body of the rule. Now, II tries to reject the validity of the rule by selecting a goal atom (e.g., B) in its body that he thinks is not satisfied (e.g., II moves to the goal node for B). I then moves to a negated relation node for this goal (e.g., a node $\neg B$), claiming the goal is true because its negation is false. From here, II moves to the relation node B, questioning I's claim that B is true. The game then continues in the same way. Note that the graph on the left of in Figure 7.3 is a schema-level description. When one cycle (relation⇝rule⇝goal⇝¬relation⇝relation) is complete, the actual fact that is argued about has changed (e.g., from A to B). If II selects a negated goal (e.g., $\neg C$) in the body of a rule then player I moves directly from the negated goal node to the relation node for C. This essentially switches the roles of I and II since now player II has to argue for a relation node C.



(a) $Q_{\mathtt{neg}}$ as a game diagram

Atoms A,B, and C

$\mathtt{M}(f_{\neg\mathtt{A}}(X), f_{\mathtt{A}}(X)) \quad :\!-\ \mathtt{d}(X).$

$\mathtt{M}(f_{\neg\mathtt{B}}(X,Y), f_{\mathtt{B}}(X,Y)) \quad :\!-\ \mathtt{d}(X), \mathtt{d}(Y).$

$\mathtt{M}(f_{\neg\mathtt{C}}(X), f_{\mathtt{C}}(X)) \quad :\!-\ \mathtt{d}(X).$

IDB A via rule r_1

$\mathtt{M}(f_{\mathtt{A}}(X), f_{\mathtt{r_1}}(X,Y)) \quad :\!-\ \mathtt{d}(X), \mathtt{d}(Y).$

$\mathtt{M}(f_{\mathtt{r_1}}(X,Y), f_{\mathtt{g_1^1}}(X,Y)) \quad :\!-\ \mathtt{d}(X), \mathtt{d}(Y).$

$\mathtt{M}(f_{\mathtt{r_1}}(X,Y), f_{\mathtt{g_1^2}}(Y)) \quad :\!-\ \mathtt{d}(X), \mathtt{d}(Y).$

$\mathtt{M}(f_{\mathtt{g_1^1}}(X,Y), f_{\neg\mathtt{B}}(X,Y)) \quad :\!-\ \mathtt{d}(X), \mathtt{d}(Y).$

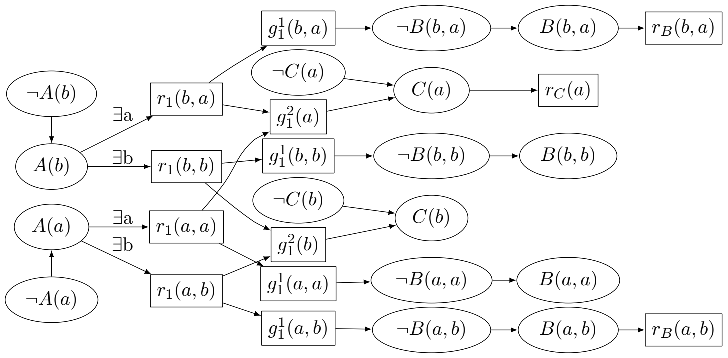
$\mathtt{M}(f_{\mathtt{g_1^2}}(X), f_{\mathtt{C}}(X)) \quad :\!-\ \mathtt{d}(X).$

EDB B and C

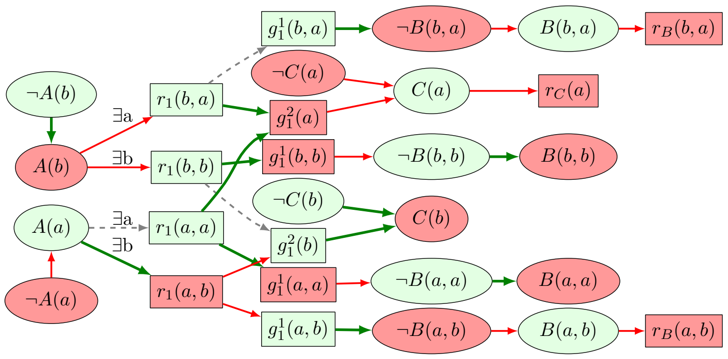
$\mathtt{M}(f_{\mathtt{B}}(X,Y), f_{\mathtt{r_B}}(X,Y)) \quad :\!-\ \mathtt{B}(X,Y).$

$\mathtt{M}(f_{\mathtt{C}}(X), f_{\mathtt{r_C}}(X)) \quad :\!-\ \mathtt{C}(X).$

(b) Rules defining the moves for $Q_{\mathtt{neg}}$



(c) Instantiated game $G_{Q_{\mathtt{neg}},D}$ for $D = \{B(a,b), B(b,a), C(a)\}$



(d) Solved game $G^{\gamma}_{Q_{\mathtt{neg}},D}$. Lost positions are (dark) red; won positions are (light) green. Provenance edges (good moves) are solid; bad moves are dashed. $A(a)$ (resp. $A(b)$) is true (resp. false), indicated by position value W (resp. L). The game provenance $\Gamma(A(a))$ explains why/how $A(a)$ is true; $\Gamma(A(b))$ explains why-not $A(b)$.

Figure 7.4: Provenance game for the FO query $Q_{\mathtt{neg}} := \mathtt{A}(X) \ :\!-\ \mathtt{B}(X,Y), \neg\mathtt{C}(Y)$. The well-founded model of the rule $\mathtt{win}(X) \ :\!-\ \mathtt{M}(X,Y), \neg\mathtt{win}(Y)$, applied to the move graph $\mathtt{M}$, solves the game.

Now, the general argumentation scheme is demonstrated for a concrete Datalog$^\neg$ program $Q_{\texttt{neg}}$. The program $Q_{\texttt{neg}}$ consists of a single rule r_1:

$$r_1: \quad \texttt{A}(X) \text{ :- } \underbrace{\texttt{B}(X,Y)}_{g_1},\ \underbrace{\neg\,\texttt{C}(Y)}_{g_2} \tag{$Q_{\texttt{neg}}$}$$

The game diagram for $Q_{\texttt{neg}}$ is shown in Figure 7.4a. Player I starts in a relation node of type $A(X)$ with a concrete instantiation $X = x$ to prove that $\texttt{A}(x) \in Q(D)$. In her first move, she picks the rule r_1 together with bindings for all existentially quantified variables in r_1, which is just a instantiation y for Y in r_1; essentially picking a ground instance $r_1(x, y)$ such that the variable X is bound to the desired x. She claims the rule body is satisfied. If this is not the case, II can falsify the claim by selecting a *goal* from the body, i.e., either $g_1^1(x, y)$, thus making a counter-claim that $\texttt{B}(x, y)$ is false, or $g_1^2(y)$, claiming instead that $C(y)$ is true. *Positive case,* e.g., II moved to $g_1^1(x, y)$. Player I will move from $g_1^1(x, y)$ to $\neg B(x, y)$, from which II will move to $B(x, y)$. In this node, there is an edge for player I if and only if $B(x, y) \in D$, that is if there is a trivial, bodyless rule $r_B(x, y)$ representing this fact. Thus, I wins the game if $B(x, y) \in D$ and II wins if $B(x, y) \notin D$. *Negative case,* e.g., II just moved to $g_1^2(y)$. Player I moves to the instantiation $\texttt{C}(y)$ of relation node $C(X)$. For this move in the diagram, variables used in the goal node are explicitly renamed to the single variable name used in the corresponding relation node. With this move, II loses and I wins if $C(y) \notin D$; II wins the argument if $C(y) \in D$ by moving to the trivial rule node, forcing I to lose.

Construction of Evaluation Game Graph. A game is constructed in a way that the constants of the program are also encoded within the game positions. Figure 7.4b provides the Datalog rules that define the move relation $\texttt{M}$ of the evaluation game $G_{Q_{\texttt{neg}},D}$ for $Q_{\texttt{neg}}$ with an input database D. Here, $\texttt{d}$ is a relation that contains the active domain of $Q_{\texttt{neg}}$ and D.

A positive and a negative relation node is created for each ground atom. Skolem functions are used to create "node identifiers". For example, the ground atom $\texttt{S}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})$ is represented by its positive relation nodes $f_S(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})$ and its negative relation nodes $f_{\neg S}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})$. The first three rules in Figure 7.4b create an edge from the negative to the positive node.[6]

[6]The use of Skolems is for convenience only. Instead, constants can be used and the arity of relations must be increased accordingly, or constants could be avoided by using [FKL97, FKL00].

Furthermore, a *rule node* is created for each rule r_i in the ground program. The node has a unique identifier $f_{r_i}(X_1, \ldots, X_n)$ that includes the rule number and the assignments of variables found in the rule's body to constants. For simplicity, variables are alphabetically ordered and the constants are provided in this order. There is an edge from the ground head atom to the ground rule node (cf. Figure 7.4b first line of middle block). For example, the Skolem function $f_{r_1}(a, b)$ encodes the whole rule body $r1 : [B(a, b), \neg C(b)]$.

Then, moves are added that start from the rule node and go to the corresponding *goal nodes* g_i^j. Goal nodes are identified by the rule number i they occur in, their positions j within the body, and the bound constants. (cf. lines 2 and 3 of middle block). From positive (negative) goal nodes, moves go to negative (positive) relation nodes keeping the bound constants fixed (cf. lines 4 and 5 of middle block). Finally, for EDB relations, an edge is added starting from the positive relation node $R(\bar{c})$ and going to a rule node $f_{\mathtt{r_R}}(\bar{c})$ iff $R(\bar{c}) \in D$. This ensures that a player reaching the relation node $R(\bar{c})$ wins iff $R(\bar{c}) \in D$. In Figure 7.4c the game graph for $Q_{\mathtt{neg}}$ with input database $D = \{B(a, b), B(b, a), C(a)\}$ is shown. The solved game is shown in Figure 7.4d. Here, I has a winning strategy for e.g., $A(a)$, $B(b, a)$, and $C(a)$.

Acyclicity of FO Games. For FO queries, represented by non-recursive Datalog$^\neg$ programs, no relation node is reachable from itself and the resulting game graph is acyclic.

Theorem 2 (FO Provenance Game) *Consider a first-order query φ in the form of a non-recursive Datalog$^\neg$ program Q_φ with output relation* $\mathtt{ans}$ *and input database facts D. Let $G^\gamma_{Q_\varphi,D} = (V, M, \gamma)$ be the solved game. Then:*

1. *$G^\gamma_{Q_\varphi,D}$ is draw-free.*

2. $$Q_\varphi(\,\mathtt{ans}(\bar{x})\,) = \left\{\begin{matrix}\mathsf{true}\\ \mathsf{false}\end{matrix}\right\} \Leftrightarrow \gamma(\,f_{\mathtt{ans}}(\bar{x})\,) = \left\{\begin{matrix}\mathsf{W}\\ \mathsf{L}\end{matrix}\right\}$$ □

Sketch. It is easy to see that one can associate with every non-recursive Datalog$^\neg$ program Q and input D an evaluation game graph $G_{Q,D}$ together with a solved game $G^\gamma_{Q,D}$. Since the game graph is acyclic, the solved game will not contain any drawn positions. Further, by construction, $G^\gamma_{Q,D}$ models query evaluation of $Q(D)$.

7.3.2 Relationship with Provenance Polynomials – Provenance for $\mathcal{RA}^+$

Game graphs are constructed to preserve provenance information available in program and database. It turns out that for positive Datalog programs Q they generate semiring provenance polynomials as defined in [GKT07, KG12] for atoms $A(\bar{x}) \in Q(D)$.

Semiring Provenance Polynomials. Semiring provenance [GKT07, KG12] attaches provenance information to EDB and IDB facts. The provenance information are elements of a commutative semiring K. A commutative semiring is an algebraic structure with two distinct associative and commutative operations "+" and "$\times$". During query evaluation, result facts are annotated with elements from K that are created by combining the provenance information from input facts. For example, in the join $\mathtt{R}(\mathtt{a},\mathtt{b}) :- \mathtt{S}(\mathtt{a},\mathtt{b}), \mathtt{T}(\mathtt{a})$ with $\mathtt{S}(\mathtt{a},\mathtt{b})$ being annotated with $p_1 \in K$ and $\mathtt{T}(\mathtt{a})$ being annotated with $p_2 \in K$, the result fact $\mathtt{R}(\mathtt{a},\mathtt{b})$ will be annotated with $p_1 \times p_2$. Intuitively, "$\times$" is used to combine provenance information of *joint* use of input facts, whereas "+" is used for alternative use of input facts.

Depending on the semiring used, different (provenance) information is propagated during query evaluation. The *most informative*[7] semiring is the *positive algebra provenance semiring* $\mathcal{P}^{\mathbb{N}[X]}$ [GKT07, KG12] whose elements are polynomials with variables from a set X and coefficients from $\mathbb{N}$. The operators "$\times$" and "+" in $\mathcal{P}^{\mathbb{N}[X]}$ are the usual addition and multiplication of polynomials. Usually, facts from the input database D are annotate by variables from a set X. Formally, $\mathcal{P}^{\mathbb{N}[X]}$ is used as a function that maps a ground atom to its provenance annotation.

Obtaining Semiring Polynomials from Game Provenance. Consider only positive programs, and fix an atom $A(\bar{x})$ with $A(\bar{x}) \in Q(D)$. The provenance graph $\Gamma_{Q,D}(f_{\mathtt{A}}(\bar{x})) = (V, M, \gamma)$ for $A(\bar{x})$ can easily be transformed into an operator tree for a provenance polynomial. The operator tree is represented as a DAG $G^{\Omega}(A(\bar{x}))$ in which common sub-expressions are re-used. $G^{\Omega}(A(\bar{x})) = (V', M', \delta)$ has nodes V', edges M', and node labels δ. For a fixed $A(\bar{x})$, the structures of Γ and G^{Ω} coincide, that is $V = V'$ and $M = M'$. The labeling function δ maps inner nodes to either "+" or "$\times$", denoting n-ary versions of the semiring operators. Leaf nodes in game provenance graphs correspond to atoms over the EDB schema. We here only assign ele-

[7]In the sense that for any other semiring K', there exists a semiring homomorphism $\mathcal{H} : \mathcal{P}^{\mathbb{N}[X]} \to K'$. This has important implications in practice [GKT07, KG12].

ments from K to leaf nodes of the form $f_{\mathtt{r_R}}(\bar{x})$. Formally, the labeling function δ is defined as follows:

$$\delta(v) = \begin{cases} \mathcal{P}^{\mathbb{N}[X]}(A(\bar{x})) & \text{if} \quad \mathsf{F}(v) = \emptyset \quad \text{and} \quad v = f_{\mathtt{r_A}}(\bar{x}) \\ \text{“}\times\text{”} & \text{if} \quad \mathsf{F}(v) \neq \emptyset \quad \text{and} \quad \gamma(v) = \mathsf{L} \\ \text{“}+\text{”} & \text{if} \quad \mathsf{F}(v) \neq \emptyset \quad \text{and} \quad \gamma(v) = \mathsf{W} \end{cases} \tag{7.1}$$

We use Ω to denote the transformation of obtaining $G^{\Omega}(A(\bar{x}))$ from $\Gamma_{Q,D}(f_{\mathtt{A}}(\bar{x}))$. The provenance semiring polynomial of fact $A(\bar{x})$ is now explicit in $G^{\Omega}(A(\bar{x}))$. An inner node "$+$" (or "$\times$") with n children represents an n-ary version of $+$ (or $\times$) from the semiring. Since the semiring operators are associative and commutative, their n-ary versions are well-defined.

Proposition 2 *For positive Q, and $A(\bar{x}) \in Q(D)$, all leaves in $\Gamma_{Q,D}(A(\bar{x}))$ are of type $f_{\mathtt{r_B}}(X, Y)$; thus the labeling described above is complete.*

Sketch. For positive programs, positive relation nodes are reachable from other positive relation nodes over a path of length four as shown on the left side of Figure 7.3. For an atom $A(\bar{x}) \in Q(D)$, all reachable rule nodes are lost and all reachable goal nodes are won. □

The following theorem relates semiring provenance polynomials to the provenance expressions obtained in G^{Ω}:

Theorem 3 *Let $\Gamma_{Q,D}$ be the game provenance of an $\mathcal{RA}^{+}$ query Q (in the form of a positive, non-recursive Datalog program) over database D. Then $\Gamma_{Q,D}$ represents the provenance polynomials $\mathbb{N}[X]$ as follows: for all $A(\bar{x}) \in Q(D)$,*

$$\Omega \circ \Gamma_{Q,D}(\, f_{\mathtt{A}}(\bar{x})\,) \equiv \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(\, A(\bar{x})\,).$$

Sketch. Our game graph construction is an extension of the graph presented in Section 4.2 of [KG12]. Rule nodes correspond to the join nodes presented in [KG12]. Named goal nodes can be seen as labels on the edges between (goal) tuple nodes and join nodes and allow us to identify at which position a tuple was used in the body. □

Proof The evaluation of the transformed game graph $\Omega \circ \Gamma_{Q,D}(\, f_{\mathtt{R}}(\bar{x})\,)$ is structurally equivalent to the evaluation of provenance semiring polynomials of the annotated $Q(D)$:

EDB Facts: Using provenance semirings, a fact $\mathtt{R}(\bar{x})$ has the annotation $\mathcal{P}^{\mathbb{N}[X]}_{Q,D}(\, R(\bar{x})\,)$. The evaluation of provenance polynomials using provenance games starts at the positive relation node $f_{\mathtt{R}}(\bar{x})$. Since $R(\bar{x}) \in Q(D)$ and by definition of the game graph this relation node has one reachable node $\mathsf{F}(f_{\mathtt{R}}(\bar{x})) = \{f_{\mathtt{r_R}}(\bar{x})\}$: $\Omega \circ \Gamma_{Q,D}(f_{\mathtt{R}}(\bar{x})) = \Omega \circ \Gamma_{Q,D}(f_{\mathtt{r_R}}(\bar{x}))$. The node $f_{\mathtt{r_R}}(\bar{x})$ is a leaf node, so the evaluation Ω returns its label $L(f_{\mathtt{r_R}}(\bar{x})) = \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R(\bar{x}))$ and:

$$\Omega \circ \Gamma_{Q,D}(f_{\mathtt{R}}(\bar{x})) = L(f_{\mathtt{r_R}}(\bar{x})) = \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R(\bar{x})).$$

Union: Let $Q(D) := \{r_1 \colon U(\bar{x}) \leftarrow R_1(\bar{x}).\ r_2 \colon U(\bar{x}) \leftarrow R_2(\bar{x}).\}$ When evaluating $Q(D)$, the provenance semiring polynomial for fact $U(\bar{x}) \in Q(D)$ is: $\mathcal{P}^{\mathbb{N}[X]}_{Q,D}(U(\bar{x})) = \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_1(\bar{x})) + \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_2(\bar{x}))$. The evaluation of provenance polynomials for $U(\bar{x}) \in Q(D)$ using provenance games starts at the positive relation node $f_{\mathtt{U}}(\bar{x})$. By definition of the game graph for $Q(D)$, $\mathsf{F}(f_{\mathtt{U}}(\bar{x})) = \{f_{\mathtt{r_1}}(\bar{x}), f_{\mathtt{r_2}}(\bar{x})\}$ and since $\gamma(f_{\mathtt{U}}(\bar{x})) = \mathsf{W}$ both terms are combined with $L(f_{\mathtt{U}}(\bar{x})) =$"+":

$$\Omega \circ \Gamma_{Q,D}(f_{\mathtt{U}}(\bar{x})) \quad = \quad \Omega \circ \Gamma_{Q,D}(f_{\mathtt{r_1}}(\bar{x})) + \Omega \circ \Gamma_{Q,D}(f_{\mathtt{r_2}}(\bar{x}))$$

Each rule node in $\Gamma_{Q,D}$ has exactly one outgoing edge to a goal node. Since the program is positive, each goal node has exactly one following negated relation node. Those negated relation nodes in turn have exactly one corresponding positive relation node. As shown above for EDB facts, for positive programs and a head node $U(\bar{x}) \in Q(D)$, positive relation nodes lead to the corresponding provenance annotations:

$$\begin{aligned}
\Omega \circ \Gamma_{Q,D}(f_{\mathtt{U}}(\bar{x})) &= \Omega \circ \Gamma_{Q,D}(f_{\mathtt{g_1^1}}(\bar{x})) + \Omega \circ \Gamma_{Q,D}(f_{\mathtt{g_2^1}}(\bar{x})) \\
&= \Omega \circ \Gamma_{Q,D}(f_{\neg\mathtt{R_1}}(\bar{x})) + \Omega \circ \Gamma_{Q,D}(f_{\neg\mathtt{R_2}}(\bar{x})) \\
&= \Omega \circ \Gamma_{Q,D}(f_{\mathtt{R_1}}(\bar{x})) + \Omega \circ \Gamma_{Q,D}(f_{\mathtt{R_2}}(\bar{x})) \\
&= \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_1(\bar{x})) + \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_2(\bar{x}))
\end{aligned}$$

Join: Let $Q(D) := \{r_1 \colon J(\bar{x}) \leftarrow R_1(\bar{x}), R_2(\bar{x}).\}$ Evaluating $Q(D)$ for a $J(\bar{x}) \in Q(D)$ using provenance semiring annotations yields: $\mathcal{P}^{\mathbb{N}[X]}_{Q,D}(J(\bar{x})) = \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_1(\bar{x})) \times \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_2(\bar{x}))$. The evaluation of provenance polynomials for $J(\bar{x}) \in Q(D)$ using provenance games starts at the positive relation node $f_{\mathtt{J}}(\bar{x})$. By definition of the game graph for $Q(D)$, $f_{\mathtt{J}}(\bar{x})$ connects to exactly one rule node: $\mathsf{F}(f_{\mathtt{J}}(\bar{x})) = \{f_{\mathtt{r}_1}(\bar{x})\}$. This rule node in turn leads to two goal nodes $\mathsf{F}(f_{\mathtt{r}_1}(\bar{x})) = \{f_{\mathtt{g}_1^1}(\bar{x}), f_{\mathtt{g}_1^2}(\bar{x})\}$, which are combined with $L(f_{\mathtt{r}_1}(\bar{x}))$ =“$\times$”, since $\gamma(f_{\mathtt{r}_1}(\bar{x})) = \mathsf{L}$:

$$\begin{aligned}\Omega \circ \Gamma_{Q,D}(f_{\mathtt{J}}(\bar{x})) &= \Omega \circ \Gamma_{Q,D}(f_{\mathtt{r}_1}(\bar{x}))\\ &= \Omega \circ \Gamma_{Q,D}(f_{\mathtt{g}_1^1}(\bar{x})) \times \Omega \circ \Gamma_{Q,D}(f_{\mathtt{g}_1^2}(\bar{x}))\end{aligned}$$

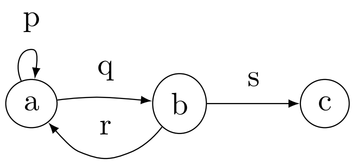
Since the program is positive, each goal node has exactly one following negated relation node. Those negated relation nodes in turn have exactly one corresponding positive relation node. As shown above for EDB facts and for positive programs with a head node $J(\bar{x}) \in Q(D)$, positive relation nodes lead to the corresponding provenance semiring annotations:

$$\begin{aligned}\Omega \circ \Gamma_{Q,D}(f_{\mathtt{J}}(\bar{x})) &= \Omega \circ \Gamma_{Q,D}(f_{\neg\mathtt{R}_1}(\bar{x})) \times \Omega \circ \Gamma_{Q,D}(f_{\neg\mathtt{R}_2}(\bar{x}))\\ &= \Omega \circ \Gamma_{Q,D}(f_{\mathtt{R}_1}(\bar{x})) \times \Omega \circ \Gamma_{Q,D}(f_{\mathtt{R}_2}(\bar{x}))\\ &= \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_1(\bar{x})) \times \mathcal{P}^{\mathbb{N}[X]}_{Q,D}(R_2(\bar{x}))\end{aligned}$$

Example 3hop from [KG12]. Consider the `3Hop` query $Q_{\mathtt{3Hop}}$ used in Figure 7 of [KG12]:

$$r_1: \quad \mathtt{3Hop}(X,Y) :- \mathtt{hop}(X,Z_1), \mathtt{hop}(Z_1,Z_2), \mathtt{hop}(Z_2,Y).$$

The query uses an input database consisting of a single binary EDB relation `hop` representing a directed graph. It asks for pairs of nodes that are reachable via exactly three edges(=*hops*). An input database D and $\mathcal{P}^{\mathbb{N}[X]}_{Q_{\mathtt{3Hop}},D}$ annotations of $Q_{\mathtt{3Hop}}$ are shown in Figure 7.5b. Figure 7.5d shows the game provenance $\Gamma(f_{\mathtt{3Hop}}(a,a))$ of fact $\mathtt{3Hop}(\mathtt{a},\mathtt{a})$. Positive won relation nodes indicate the existence of the corresponding fact in $Q_{3Hop}(D)$. To obtain the provenance polynomial of fact $f_{\mathtt{3Hop}}(a,a)$, Ω is applied to $\Gamma(f_{\mathtt{3Hop}}(a,a))$ as shown in Figure 7.5e: inner won nodes are replaced by “$\times$”, inner lost nodes by “$+$”, and leaf nodes by their respective annotations from K as given



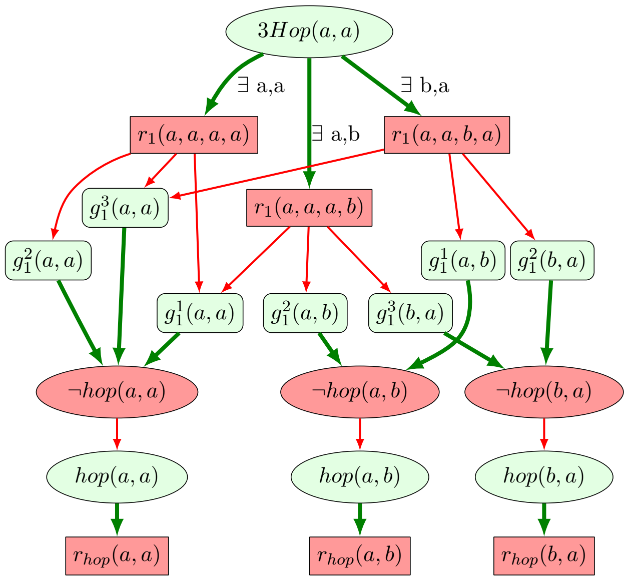
(a) Input database D

hop	$\Gamma_{Q_{\mathtt{3Hop}},D}$	δ
a a	$f_{\mathtt{r_{hop}}}(a,a)$	p
a b	$f_{\mathtt{r_{hop}}}(a,b)$	q
b a	$f_{\mathtt{r_{hop}}}(b,a)$	r
b c	$f_{\mathtt{r_{hop}}}(b,c)$	s

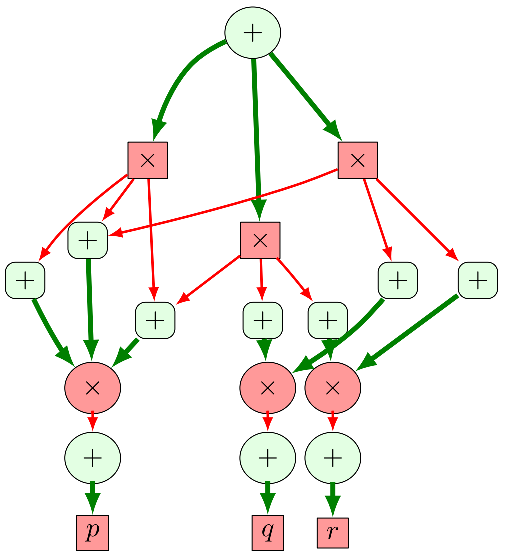
(b) Labels δ for leaf nodes of $\Gamma_{Q_{\mathtt{3Hop}},D}$.

3hop	$\Gamma_{Q_{\mathtt{3Hop}},D}$	$\mathcal{P}^{\mathbb{N}[X]}_{Q_{\mathtt{3Hop}},D}$
a a	$f_{\mathtt{3Hop}}(a,a)$	$p^3 + 2pqr$
a b	$f_{\mathtt{3Hop}}(a,b)$	$p^2q + q^2r$
a c	$f_{\mathtt{3Hop}}(a,c)$	pqs
b a	$f_{\mathtt{3Hop}}(b,a)$	$p^2r + qr^2$
b b	$f_{\mathtt{3Hop}}(b,b)$	pqr
b c	$f_{\mathtt{3Hop}}(b,c)$	qrs

(c) Provenance for inner relation nodes of $\Gamma_{Q_{\mathtt{3Hop}},D}$.



(d) $\Gamma(\,\mathtt{3hop}(a,a)\,)$



(e) DAG $G^{\Omega} = \Omega \circ \Gamma(\mathtt{3hop}(a,a))$. Interpreting G^{Ω} yields $p^3 + 2pqr$.

Figure 7.5: Input graph for program $Q_{\mathtt{3Hop}}$ in (a) using edge labeling according to (b). Game provenance $\Gamma_{Q_{\mathtt{3Hop}},D}$ for the query $\mathtt{3Hop(a,a)}$ on input database of (a) is shown in (d). When labeling leaf nodes according to (b), lost inner nodes by "×", and won inner nodes by "+" then the operator DAG G^{Ω} shown in (e) is created. This DAG represents the semiring-provenance polynomial for the query $\mathtt{3Hop(a,a)}$ shown in (c) and [KG12].

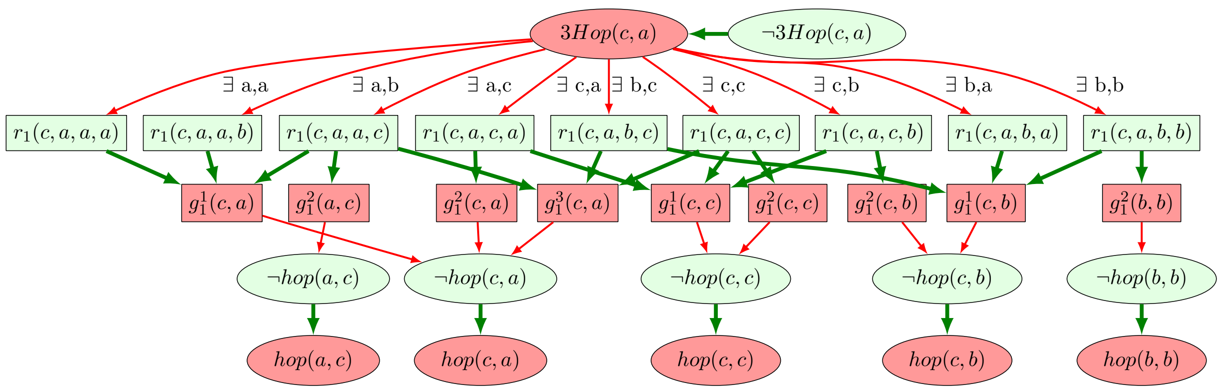


Figure 7.6: Why-not provenance for $\texttt{3Hop}(\texttt{c},\texttt{a})$ using provenance games.

in Figure 7.5b and [KG12]. The so relabeled graph encodes the provenance equation

$$\Omega \circ \Gamma_{Q_{\texttt{3Hop}},D}(f_{\texttt{3Hop}}(a,a)) = (p \times p \times p) + (p \times q \times r) + (p \times q \times r) = p^3 + 2pqr$$

ProgD which is equivalent to the annotation of provenance semiring polynomials as shown in Figure 7.5c and [KG12].

7.3.3 Why-Not Game Provenance for $\mathcal{RA}^+$

Game provenance also yields meaningful explanations for *why-not* questions. the active domain. Consider for example the query $Q_{\texttt{3Hop}}$ and its input database D. The atom $\texttt{3Hop}(c,a)$ is not in $Q_{\texttt{3Hop}}(D)$ and explanation for "why" is desired. Figure 7.6 shows the game provenance $\Gamma_{Q_{\texttt{3Hop}},D}(f_{\neg\texttt{3Hop}}(c,a))$ of the missing fact $\texttt{3Hop}(\texttt{c},\texttt{a})$. The lost relation node $\texttt{3Hop}(c,a)$ indicates that player I will lose the argument that tries to show that $\texttt{3Hop}(c,a) \in Q_{3Hop}(D)$. The game provenance explains why: Any ground instantiation of rule r_1 will be winning node for player II. Consider, e.g., moving to $r_1(c,a,a,a)$ which represents the rule instantiation for $X/c, Y/a, Z_1/a, Z_2/a$. Player II wins the game here by questioning that the first goal $g_1^1(c,a)$ is satisfied. And indeed, player I will move from $g_1^1(c,a)$ to $\neg\texttt{hop}(c,a)$; II to $\texttt{hop}(c,a)$. Now, I loses the game since $\texttt{hop}(c,a) \notin D$ and thus there is no move out of $\texttt{hop}(c,a)$. Another rule instantiation $X/c, Y/a, Z_1/a, Z_2/b$ fails for the same reason: the missing $\texttt{hop}(c,a)$. The instantiation $X/c, Y/a, Z_1/b, Z_2/a$ fails because $\texttt{hop}(c,b)$ is not in the input. Other instantiations, such as $X/c, Y/a, Z_1/c, Z_2/b$, fail because two facts are missing from the input, here $\texttt{hop}(c,b)$ and

$\mathtt{hop}(c, c)$.

It is no coincidence that all leaf nodes represent missing EDB facts for why-not provenance in positive non-recursive Datalog programs:

Proposition 3 *Let Q be a non-recursive Datalog program, D a database, $\Gamma(f_{\mathtt{A}}(\bar{x}))$ the game provenance for facts $A(\bar{x}) \notin Q(D)$. All leaves of $\Gamma(f_{\mathtt{A}}(\bar{x}))$ have type $f_{\mathtt{R}}(\bar{y})$ and represent ground EDB atoms $R(\bar{y})$ that are missing from the input. These nodes are labeled with a unique $\neg k$, where $k \in K$ is a new constant in K that is not used for any other present or missing fact.* □

The above proposition illustrates that for positive queries, the ultimate reason for failure to derive outputs are missing inputs, represented by the leaves in provenance games. Accordingly, the labeling function L of (7.1) supports additional leave nodes of type $f_{\mathtt{R}}(\bar{x})$:

$$L(v) = \begin{cases} \mathcal{P}_{\mathbb{N}[X]}(R(\bar{x})) & \text{if} \quad \mathsf{F}(v) = \emptyset \ \text{ and } \ v = f_{\mathtt{r_R}}(\bar{x}) \\ \neg k_{R(\bar{x})} & \text{if} \quad \mathsf{F}(v) = \emptyset \ \text{ and } \ v = f_{\mathtt{R}}(\bar{x}) \text{ with } k_{R(\bar{x})} \text{ a fresh value in } K \\ \text{“}\times\text{”} & \text{if} \quad \mathsf{F}(v) \neq \emptyset \ \text{ and } \ \gamma(v) = \mathsf{L} \\ \text{“}+\text{”} & \text{if} \quad \mathsf{F}(v) \neq \emptyset \ \text{ and } \ \gamma(v) = \mathsf{W} \end{cases} \tag{7.2}$$

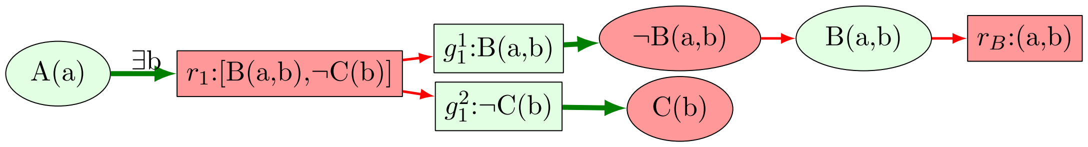
As defined, game provenance is sensitive to the active domain of query and input database, which can lead to interesting effects. Consider the following query variant $Q'_{\mathtt{neg}}$:

$$\begin{aligned} \mathtt{C}(y) \quad &:- \quad \mathtt{E}(y, z). \\ \mathtt{A}(x) \quad &:- \quad \mathtt{B}(x, y), \neg\mathtt{C}(y). \end{aligned}$$

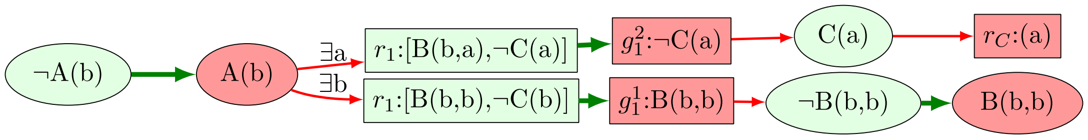
with input $D = \{\mathtt{B}(a, a)\}$. Here, game provenance shows that $\mathtt{A}(a)$ depends on the presence of $\mathtt{B}(a, a)$ as well as on the absence of $\mathtt{E}(a, a)$. The game provenance graph does not mention that the absence, e.g., of $\mathtt{E}(a, b)$ is important as well—simply because b is not in the active domain.

7.3.4 Game Provenance for First-Order Queries

This section presents provenance games in the presence of negation within the query. When constructing game graphs for Datalog$^\neg$ queries with negated goals, graphs are optained in which



(a) Game provenance graph $\Gamma(f_{\mathtt{A}}(a))$ for $A(a) \in Q_{\mathtt{neg}}(D)$.



(b) Game provenance graph $\Gamma(f_{\neg\mathtt{A}}(b))$ for $A(b) \notin Q_{\mathtt{neg}}(D)$

Figure 7.7: Provenance graphs for $Q_{\mathtt{neg}}$ with database $D = \{B(a,b), B(b,a), C(a)\}$. Both *why* and *why-not* graphs might contain leaf nodes representing existent and missing input facts.

there exists a path of length three between positive relation nodes. This switches roles between player I and II. In other words, to explain why a *negated* subgoal is satisfied, an argument like in the why-not case is used. In general, this leads to provenance graphs that contain leaf nodes of both kinds: $f_{\mathtt{C}}(\bar{x})$ representing missing facts $\mathtt{R}(\bar{x}) \notin D$ and $f_{\mathtt{r_R}}(\bar{x})$ representing input facts $\mathtt{R}(\bar{x}) \in D$.

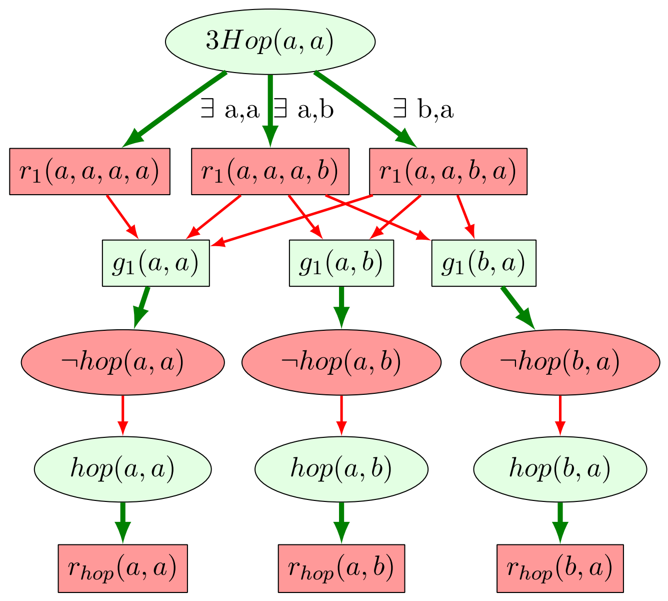
In the following, examples are based on the $Q_{\mathtt{neg}}$ query (cf. Figure 7.4) with input database $D = \{B(a,b), B(b,a), C(a)\}$.

Why Provenance. Figure 7.7a shows the provenance graph for the output fact $A(a)$. One can see that $A(a)$ could be derived via rule r_1 with the bindings $X/a, Y/b$. The positive goal succeeds due to the existence of the EDB fact $B(a,b)$. The negative goal g_1^2 succeeds due to the missing fact $C(b)$ from the input D.

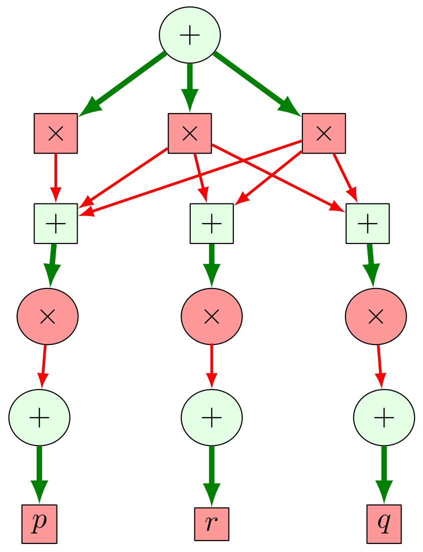
Why-Not Provenance. Figure 7.7b shows the provenance graph for $A(b)$ which is not part of $Q_{\mathtt{neg}}(D)$. It is shown that a player starting in $\neg A(b)$ will win the argument since $A(b)$ cannot be shown. Both attempts to derive $A(b)$ fail. With $X/b, Y/a$ the second goal $\neg C(a)$ is not satisfied since $C(a) \in D$. With $X/b, Y/b$ the first goal $B(b,b)$ fails since $B(b,b) \notin D$.

7.3.5 Variants of the Evaluation Game Graph

In the graph construction for provenance games, the definition of the Skolem functions is critical to capture provenance equivalent to $\mathbb{N}[X]$ provenance polynomials. Recall that the Skolem function for rule node identifiers, e.g., $f_{\mathtt{r_1}}(X,Y)$, depend on the rule (here $\mathtt{r}_1$) as well as the



(a) $\Gamma^{\mathsf{Trio}(X)}_{Q_{\mathtt{3Hop}},D}$ for $\mathtt{3Hop(a,a)}$



(b) $\Omega \circ \Gamma^{\mathsf{Trio}(X)}_{Q_{\mathtt{3Hop}},D}$ for $\mathtt{3Hop(a,a)}$

Figure 7.8: Creating $\mathsf{Trio}(X)$ style provenance game variants for Q_{3Hop} by dropping positional identifiers in the Skolem function for goal nodes. The operator tree on the right reads $p + 2pqr$.

constants assigned to body variables. Skolem functions of goal node identifiers, e.g., $f_{\mathtt{g}_1^2}(X,Y)$, depend on the rule they belong to (here 1), the exact position in the rule body at which that goal occurs (here 2), and values of the bound variables.

By changing the definition of one or more Skolem functions, more compact but also less informative provenance can be encoded. We here only describe a simple variant that will create $\mathsf{Trio}(X)$ [BSHW06] style provenance instead of $\mathbb{N}[X]$ provenance polynomials for $\mathcal{RA}^+$ queries. When changing the Skolem function of goal node identifiers by removing the positional argument for the goal, goals that appear at different positions in the body of a rule collapse into a single node. This construction yields a modified operator graph. In particular, using the same fact multiple times jointly in a rule will be recorded only as a single use—as it is the case in $\mathsf{Trio}(X)$ provenance polynomials.

The game graph $\Gamma^{\mathsf{Trio}(X)}_{Q_{\mathtt{3Hop}},D}\left(f_{\mathtt{3Hop}}(a,a)\right)$ and the corresponding operator graph are shown in Figure 7.8. Reading out the polynomial results in the Trio-provenance-polynomial $p + 2pqr$ for the input fact annotations given in Figure 7.5b.

7.4 Extension to Well-Founded Recursive Datalog$^\neg$

The previous sections described game construction and interpretation for non-recursive Datalog$^\neg$ only. This section describes the case when $Q \in$ Datalog$^\neg$ contains recursion. In this case, the solution of the evaluation game represents the least fixpoint of the Fitting operator [Fit85]. This *Fitting semantics* $\mathcal{F}(D)$ is a 3-valued semantics (with truth values true, false, and undef). It computes the least fixpoint[8] of interpreting the rules under the strong 3-valued Kripke-Kleene logic.

Theorem 4 (Games) *Let $Q \in$ Datalog$^\neg$ with input D[9]. Then the associated game $\Gamma_{Q,D}$ corresponds to the Fitting semantics of $\mathcal{F}(Q, D)$, that is for any ground fact of the Herbrand base $A(\bar{x}) \in \mathcal{B}_Q$:*

1. *$f_{\mathsf{A}}(\bar{x})$ is won in $\Gamma_{Q,D} \Leftrightarrow A(\bar{x})$* is true in $\mathcal{F}(Q, D)$
2. *$f_{\mathsf{A}}(\bar{x})$ is lost in $\Gamma_{Q,D} \Leftrightarrow A(\bar{x})$* is false in $\mathcal{F}(Q, D)$
3. *$f_{\mathsf{A}}(\bar{x})$ is drawn in $\Gamma_{Q,D} \Leftrightarrow A(\bar{x})$* is undef in $\mathcal{F}(Q, D)$ $\square$

Sketch. Proof via an induction that relates the algorithm to determine won and lost positions given in the introduction to the confluent rewrite system $\mapsto_{PNSF}$ from [BDFZ01]; then claim is immediate since $\mapsto_{PNSF}$ corresponds to the Fitting semantics (Theorem 14 in [BDFZ01].

As is the case for $\mathcal{RA}^+$, semiring equations can be derived from the operator tree created from the solved game-graph for IDB facts $A \in Q(D)$. Like in [GKT07], the obtained equations might contain recursive definitions that would require additional properties for the semiring $\mathbb{K}$ to hold in order to allow "solving" for a particular output fact. Here, only the correspondence between semiring provenance and provenance polynomials derived using provenance games is stated briefly. The game provenance is generated only for ground IDB atoms $A(\bar{x}) \in Q(D)$ by $\Gamma_{Q,D}(f_{\mathsf{A}}(\bar{x}))$.

[8]The ordering used is the information ordering where undef $\prec$ true and undef $\prec$ false

[9]We here and in the following identify P with a set of Datalog$^\neg$ rules. The formulation Q with input D means that D is a subset of the Herbrand base over the EDB schema of Q.

Example. Consider the right recursive transitive closure program $Q_{\texttt{TC}}$:

$$\begin{array}{llcl} (r_1) & TC(X,Y) & \leftarrow & E(X,Y). \\ (r_2) & TC(X,Y) & \leftarrow & E(X,Z), TC(Z,Y). \end{array}$$

with an input instance $D = \{E(a,b), E(b,b)\}$ shown in Figure 7.9a. The corresponding provenance graph is shown in Figure 7.9. The following provenance equation can be derived from $\Gamma_{Q_{\texttt{TC}},D}(f_{\texttt{TC}}(b,b))$ for the IDB fact $TC(b,b)$ using the annotations of Figure 7.9b and Figure 7.9c:

$$v \quad = \quad q + (q \times v)$$

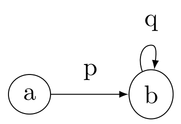
This states that $TC(b,b)$ can be recursively obtained from $E(b,b)$ and $E(b,b)$ together with $TC(b,b)$.

Why-Not Provenance. For recursive Datalog programs, the solved provenance game can have three different solutions for relational nodes, i.e., won, lost, or drawn[10]. Provenance from won and lost-nodes can be derived in the same way as for non-recursive Datalog$^{\neg}$, the *reason* for why a position is drawn is different: Drawn positions represent infinitely failed SLD trees; or *unfounded sets* as first defined in [VGRS91].

The game provenance presented above computes the Fitting semantics. In particular, as seen in the previous example, game solution will yield drawn positions for unfounded sets. This differs from the well-founded semantics which assigns the fact a truth value of false. This discrepancy can now cause problems in stratified and non-stratified Datalog$^{\neg}$ programs: If the drawn fact is used in a negated goal-atom, the goal should be satisfied–yet the provenance game solution will yield another drawn node[11]. The key idea here is to perform explicit *loop-detection* as in [BDFZ01]. This approach will now be demonstrated on an example.

The example program Q_{CP} in Figure 7.10(a) performs Color propagation for a graph with two kinds of edges `E` and `S`. Rule (r_1) states that if Y is colored and there is an Edge to X then X should also be colored. Rule (r_2) states that if node Y is *not* colored and there is a Switch edge

[10]e.g., for P_{tc}under the Fitting semantics
[11]as does the Fitting semantics



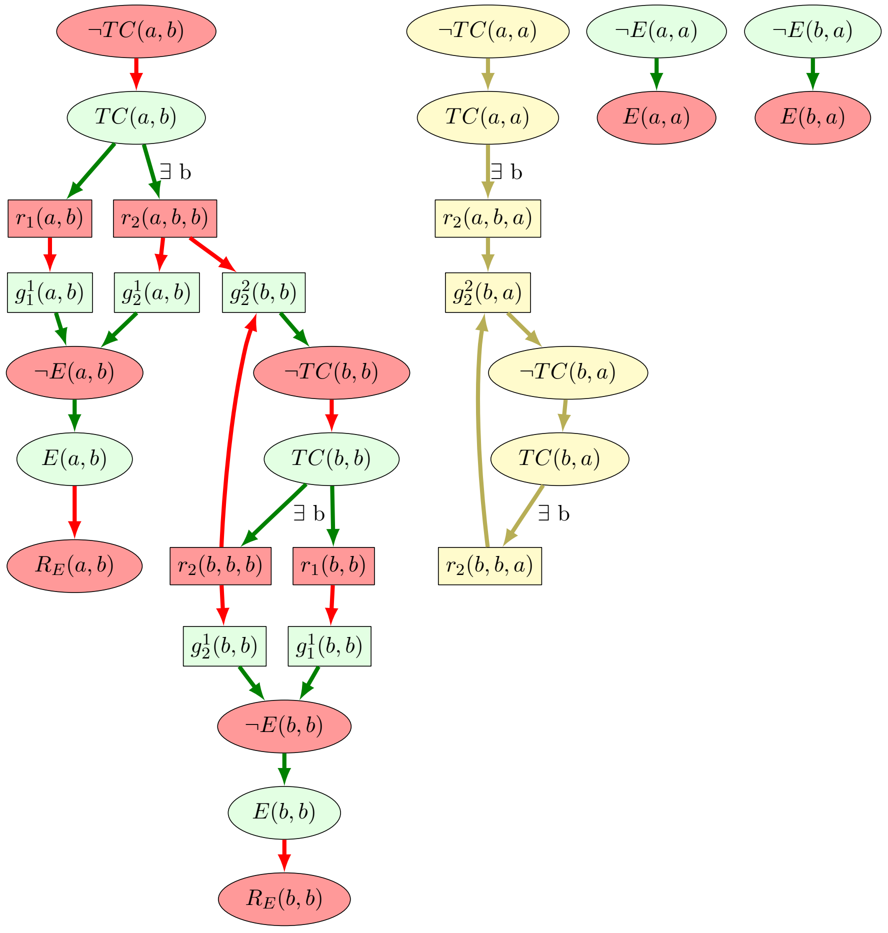
(a) Input database D

E		$\Gamma_{Q_{\texttt{TC}},D}$	δ
a	b	$f_{\texttt{r}_\texttt{E}}(a,b)$	p
b	b	$f_{\texttt{r}_\texttt{E}}(b,b)$	q

(b) Labels δ for leaf nodes of $\Gamma_{Q_{\texttt{TC}},D}$.

E		$\Gamma_{Q_{\texttt{TC}},D}$	δ
a	a	$f_{\texttt{TC}}(a,a)$	s
a	b	$f_{\texttt{TC}}(a,b)$	t
b	a	$f_{\texttt{TC}}(b,a)$	u
b	b	$f_{\texttt{TC}}(b,b)$	v

(c) Labels δ of IDB facts in $\Gamma_{Q_{\texttt{TC}},D}$.



(d) Game provenance graph $\Gamma_{Q_{\texttt{TC}},D}$

Figure 7.9: Game provenance for $Q_{\texttt{TC}}$.

from Y to X then node X should be colored. The game diagram for this program is shown in Figure 7.10(b). Observe how Q_{CP} has a positive cycle as well as a negative cycle in the predicate dependency graph, each translating to a cycle in the game diagram.

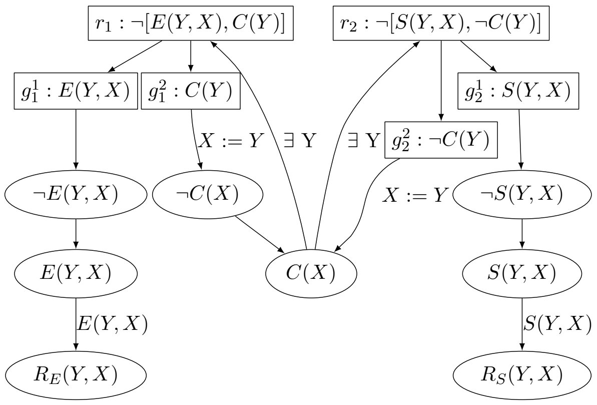
Now consider the input depicted in Figure 7.10(c). The positive cycle $p \leftrightarrows q$ in the ground-atom dependency graph is an unfounded set [VGRS91]. Also in the game graph, the relational node is a drawn position If player I is at position $f_{\mathsf{C}}(p)$ then the following (optimal) moves can be taken to avoid loosing:

$$\begin{tikzcd}
C(p) \arrow[r, squiggly, "\mathrm{I}"] & r_1(q,p) \arrow[r, squiggly, "\mathrm{II}"] & g_1^2(q) \arrow[r, squiggly, "\mathrm{I}"] & \neg C(q) \arrow[d, squiggly, "\mathrm{I}"] \\
\neg C(p) \arrow[u, squiggly, "\mathrm{II}"] & g_1^2(p) \arrow[l, squiggly, "\mathrm{I}"'] & r_1(p,q) \arrow[l, squiggly, "\mathrm{II}"'] & C(q) \arrow[l, squiggly, "\mathrm{II}"']
\end{tikzcd}$$

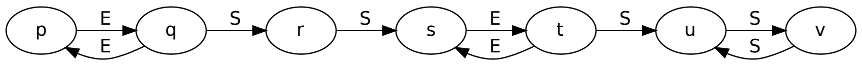
Loop detection. To "patch" the results obtained from the solved provenance game or Fitting semantics, an explicit loop detection step is executed which is similar to loop detection in [BDFZ01]. For this, consider the ground instantiated program P^* and color each atom in rules according to the status in the solved game graph. Then, focus on the instantiated rules with a yellow/draws head atom. In this example, rules with drawn heads are:

$$\begin{array}{rr}
C(p)^{\mathsf{D}}\text{:-}C(q)^{\mathsf{D}}, E(q,p)^{\mathsf{W}}. & C(r)^{\mathsf{D}}\text{:-}\neg C(q)^{\mathsf{D}}, S(q,r)^{\mathsf{W}}. \\
C(q)^{\mathsf{D}}\text{:-}C(p)^{\mathsf{D}}, E(p,q)^{\mathsf{W}}. & C(s)^{\mathsf{D}}\text{:-}\neg C(r)^{\mathsf{D}}, S(r,s)^{\mathsf{W}}. \\
C(s)^{\mathsf{D}}\text{:-}C(t)^{\mathsf{D}}, E(t,s)^{\mathsf{W}}. & C(u)^{\mathsf{D}}\text{:-}\neg C(t)^{\mathsf{D}}, S(t,u)^{\mathsf{W}}. \\
C(t)^{\mathsf{D}}\text{:-}C(s)^{\mathsf{D}}, E(s,t)^{\mathsf{W}}. & C(u)^{\mathsf{D}}\text{:-}\neg C(v)^{\mathsf{D}}, S(v,u)^{\mathsf{W}}. \\
 & C(v)^{\mathsf{D}}\text{:-}\neg C(u)^{\mathsf{D}}, S(u,v)^{\mathsf{W}}.
\end{array}$$

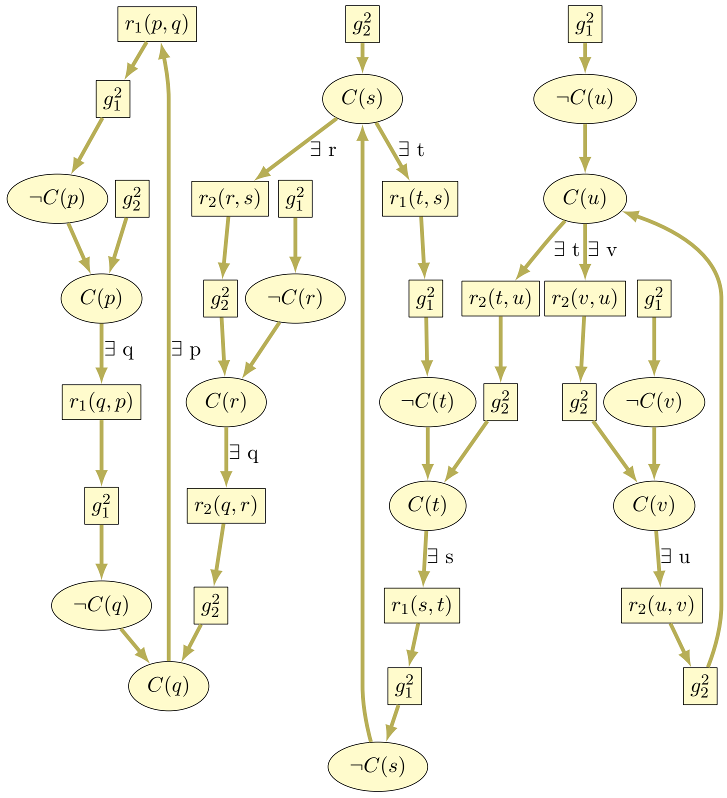
Note that in the body of these rules there can only be won or drawn positions. Then, all sub-goals of the body are removed which correspond to won positionsor are *negated* drawn goals. Thus,a



(a) Game diagram for Q_{CP}



(b) EDB input D for Q_{CP}



(c) Subgraph of drawn positions for $\Gamma_{Q_{\text{CP}},D}$

Figure 7.10: Color-Propagation Example: A positive cycle in the predicate dependency graph can lead to unfounded sets. When computing game provenance, unfounded sets are initially represented as drawn positions yielding the yellow subgraph. By modifying Q_{CP} to Q'_{CP}, unfounded positions are forced to be false. This yields the desired provenance information for the well-founded semantics.

positive program P^+_{D} is obtained:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
C(p)^{\mathsf{D}} & :- & C(q)^{\mathsf{D}}. & C(r)^{\mathsf{D}} & :- & . \\
C(q)^{\mathsf{D}} & :- & C(p)^{\mathsf{D}}. & C(s)^{\mathsf{D}} & :- & . \\
C(s)^{\mathsf{D}} & :- & C(t)^{\mathsf{D}}. & C(u)^{\mathsf{D}} & :- & . \\
C(t)^{\mathsf{D}} & :- & C(s)^{\mathsf{D}}. & C(u)^{\mathsf{D}} & :- & . \\
 & & & C(v)^{\mathsf{D}} & :- & .
\end{array}$$

Since goals were only removed to obtain P^+_{D}, the program computes an *overestimate* of true facts, i.e., the possibly true facts; cf. [BDFZ01]. Any facts that are *not* reproduced by P^+_{D} cannot possibly be true. Thus, yellow facts U that are occurring in heads of P^* but are not part of the model of P^+_{D} are an *unfounded set* according to [VGRS91]. In this example, $U = \{\mathtt{C(p)}, \mathtt{C(q)}\}$.

Accordingly, a new program Q'_{CP} is created from the original program Q_{CP} such that the well-founded semantics of Q_{CP} agrees with the Fitting semantics of Q'_{CP}; cf. Figure 7.10. For this, an additional atom $\neg \mathtt{u}_{\mathtt{C}}(Y)$ is added to r_1. $\mathtt{u}_{\mathtt{C}}$ is initially empty, and unfounded atoms are added as they are discovered. After the first round of loop-detection, $\mathtt{u}_{\mathtt{C}}$ contains $\{(p), (q)\}$ in this example.

Loop-guarded program. More generally, for any Datalog$^\neg$ program Q, the following transformation is used to transform it into the *loop-guarded program* Q':

1. Let $\mathcal{U} \subseteq idb(Q)$ be the IDB predicate symbols of Q that are part of a positive cycle in the predicate dependency graph of Q.

2. $idb(Q') = idb(Q)$. $edb(Q') = \{\mathtt{u}_R \mid R \in \mathcal{U}\}$. The arity of $\mathtt{u}_R$ is the same as the arity of R.

3. The rules in Q' are rules from Q, except that whenever the positive atom $R(\bar{X})$ with $R \in \mathcal{U}$ occurs in a rule body, another atom $\neg \mathtt{u}_R(\bar{X})$ is added.

Games for Datalog$^\neg$. Consider again the example in Figure 7.10. With $\mathtt{u}_{\mathtt{C}} = \{(p), (q)\}$, the game/Fitting semantics determines $\mathtt{C}(p)$ and $\mathtt{C}q$ as lost/false. Consequently, $\mathtt{C}(r)$ will be a won/true position. Now, an additional unfounded cycle is caused by the $\mathtt{E}$-loop $s \leftrightarrows t$. A second

Input: $Q \in \text{Datalog}^\neg$ with input D.
Output: Solved provenance game $\Gamma_{Q,D}$.

```
1. Compute loop-guarded version $Q' = \mathcal{L}(Q)$
2. Let $U = \emptyset$
3. Solve game for $G_{Q,D\cup U}$ of $Q'$ on $D \cup U$.
4. Perform loop detection to obtain the set $UF$ of unfounded ground atoms. Let
       $U := U \cup \{\mathtt{u}_R(\bar{X}) \mid R(\bar{X}) \in UF\}$
5. If $UF \neq \emptyset$ goto 3.
```

Figure 7.11: Solving provenance games for well-founded Datalog$^\neg$.

round of loop detection finally yields

$$\mathtt{u}_{\mathtt{C}} = \{(p), (q), (s), (t)\}$$

In general, given a Datalog$^\neg$ program Q with input D, the procedure from Figure 7.11 is performed to obtain the provenance-enriched game graph $\Gamma_{Q,D}$. It is easy to see that this procedure terminates, and in fact corresponds to a derivation of the well-founded semantics as described in [BDFZ01] or in [HW02]. The following proposition briefly states the correctness of this approach:

Proposition 4 *Let $Q \in$ Datalog$^\neg$ with input D and Q' the corresponding loop-guarded program. Let U be the set computed in Figure 7.11. Then evaluating the game for Q' on input $D \cup U$ yields the same result as evaluating $\mathcal{W}(Q, D)$. That is won/lost/drawn positions in the game correspond to true/false/undefined facts under the well-founded semantics.* □

Sketch. The algorithm mimics a derivation of the well-founded semantics according to [BDFZ01].

Note that it is possible that the procedure terminates with undefined nodes. These nodes are the undefined facts according to the well-founded semantics. For example consider Figure 7.10 again. The third loop $u \leftrightarrows v$ in the EDB is not a positive loop (it is via the $\underline{\text{S}}$witch relation). The positive reduction of the ground program will reproduce $\mathtt{C}(u)$ and $\mathtt{C}(v)$ since the corresponding rules will have an empty body (the negated dependency has been removed). The truth value for $\texttt{C}(u)$ and $\texttt{C}(v)$ is $\texttt{undef}$ under the well-founded semantics.

7.5 Domain Independence

The game construction for evaluating non-recursive Datalog$^\neg$ presented in the last section makes the provenance sensitive to the active domain of query and input database. This is a short-coming which can lead to interesting effects. Consider the following program:

$$\begin{aligned}\texttt{rs}(x) &\ :- \ \texttt{r}(x, y).\\ \texttt{ans}(x) &\ :- \ \texttt{q}(x), \neg\texttt{rs}(x).\end{aligned}$$

with input instance $D = \{\texttt{q}(a)\}$. Here, the game provenance would show that $\texttt{a}ns(a)$ dependents on the presence of $\texttt{q}(a)$ as well as on the absence of $\texttt{r}(a, a)$. The game provenance graph does not mention that the absence of $\texttt{r}(a, b)$ is important as well—simply because b is not in the active domain. To mitigate this effect, this section presents an alternative domain independent construction. Instead of using all instantiated rules with all combinations of domain elements from the herbrand base, this construction uses constraints to capture non-existing tuples in EDB and IDB.

The construction uses the same node and edge types as the evaluation game graphs presented above. Relation nodes in the EDB are constructed as described above but the assignment of constants is done using constrains with equalities (cf. Figure 7.13). For example, the fact $q(a)$ is translated to a fact with variables, named after the relation and the position of the arguments, as well as a constraint: $q(q1) : q1 = a$. All constraints of existing EDB facts are aggregated with a logical *or* operation and negated to form the basic equation for the non-existing EDB facts in conjunctive normal form (CNF). This CNF is then transformed to a disjunctive normal form (DNF). This DNF is processed to generate disjoint conjunctions. This process is similar to algorithms used for constructive negation. For each conjunction in the resulting DNF, a positive and a negative relation node are created and connected. This nodes have no connection to a rule node without a body, thus representing a group of facts that are not in the EDB. Now, following the dependencies in the predicate dependency graph, all rules are processed using the following procedure until no new nodes are created anymore: Given a rule, all possible goal nodes are created by identifying relation nodes with matching predicate name and renaming the variables used in the relation node constraints to the variables used in the specific goal. A new edge connects each goal node with the relation node it was created from. Then, rule nodes are processed by first combining all instantiations of the required goal nodes and creating a big conjunction from all contributing goal node constraints. For each rule node, the conjunction of all goal constraints is checked for unsatisfiability and if there are no contradictions the constraint is simplified. From each simplified constraint, a rule node is created and connected to the contributing goal nodes. Finally, the relation nodes forming the head of the rule are processed in a similar way as EDB relations are created. The constraints of all rule nodes are combined with a logical *or* operation to form a DNF. Then, this DNF is processed to generate disjoint conjunctions. For each resulting conjunction in the processed DNF a positive and a negative relation node is created in the same way as for the standard evaluation game construction. The positive relation node is then connected to all rule nodes for which the constraint of the relation node in conjunction with the rule node constraint is satisfiable. The process of constructing the game graph is illustrated in Figure 7.12. After the game graph is constructed using constraints, the game can be solved and interpreted in the same way as described earlier. However, missing facts have to be linked to the corresponding relation node with matching constraint. The resulting provenance graphs and derived provenance equations provide why and why-not provenance that does not depend on the active domain.

Example. Consider again the query $Q_{\texttt{3Hop}}$ and its input database D discussed earlier in Section 7.3.2. Figure 7.5 presents the original evaluation game, provenance and operator graph for the fact $\texttt{3Hop}(\texttt{a},\texttt{a})$. Now, Figure 7.13 presented the evaluation game using constraints for the same program $Q_{\texttt{3Hop}}$, database D and fact $\texttt{3Hop}(\texttt{a},\texttt{a})$. Note that the graph structures are equivalent and that the constraints correspond exactly to the ground facts.

Similarly, Section 7.3.3 described the standard evaluation game approach for $Q_{\texttt{3Hop}}$ and missing tuples. Figure 7.6 depicts the provenance graph for the missing tuple $\texttt{3Hop}(\texttt{c},\texttt{a})$. Figure 7.14 shows the provenance graph using constraints for the same missing fact. Note that the num-

```
ConstrGame: $Q$ q, $D$ d $\rightarrow$ ($V$, $E$)
 FOREACH p(c1,...,cn) $\in$ d DO
   constr = {(p1 = c1), ..., (pn = cn)}
   inst[p] = inst[p] $\cup$ {constr}
   inst[!p] = inst[!p] $\cup$ {constr}
   inst[R_p] = inst[R_p] $\cup$ {constr}
   E = E $\cup$ {(!p:constr, p:constr)}
   E = E $\cup$ {(p:constr, R_p:constr)}

 FOREACH p $\in$ d:
   FOREACH constr $\in$ inst[p]:
     CNF &= NOT(AND.join(constr))
     dnf = createDNFfrom(CNF)
     ddnf = make_disjoint(dnf)
     FOREACH conj $\in$ ddnf:
       inst[p] = inst[p] $\cup$ {conj}
       inst[!p] = inst[!p] $\cup$ {conj}
       E = E $\cup$ {(!p:conj, p:conj)}

 WHILE new nodes created:
   FOREACH rule $i$ $\in$ q:
     FOREACH goal $j$ $\in$ body(rule):
       goal_constr = inst[predicateOf(goal)]
       rename vars in goal_constr to variables in goal
       inst[g_i^j] = inst[g_i^j] $\cup$ goal_constr
       E = E $\cup$ {(g_i^j:goal_constr, predicateOf(goal):inst[predicateOf(goal)])}

     FOREACH rule_constr $\in$ inst[g_i^0] $\times \cdots \times$ inst[g_i^m]:
       IF rule_constr satisfiable:
         inst[r_i] = inst[r_i] $\cup$ {rule_constr}
         FOREACH g_i^j used in rule_constr:
           E = E $\cup$ (r_i:rule_constr, g_i^j:inst[g_i^j])

     FOREACH rule_constr $\in$ inst[r_i]:
       head_inst[h_i] = head_inst[h_i] $\cup$ {filter_vars(rule_constr, X)}
     inst[h_i] = make_disjoint(head_inst[h_i])
     inst[!h_i] = inst[h_i]
     FOREACH head_constr $\in$ inst[h_i]:
       E = E $\cup$ {(!h_i:head_constr, h_i:head_constr)}
       FOREACH rule_constr $\in$ inst[r_i]:
         IF (rule_const $\wedge$ head_constr) satisfiable:
           E = E $\cup$ (h_i:head_constr, r_i:rule_constr)
```

Figure 7.12: Construction of game graphs using constraints.

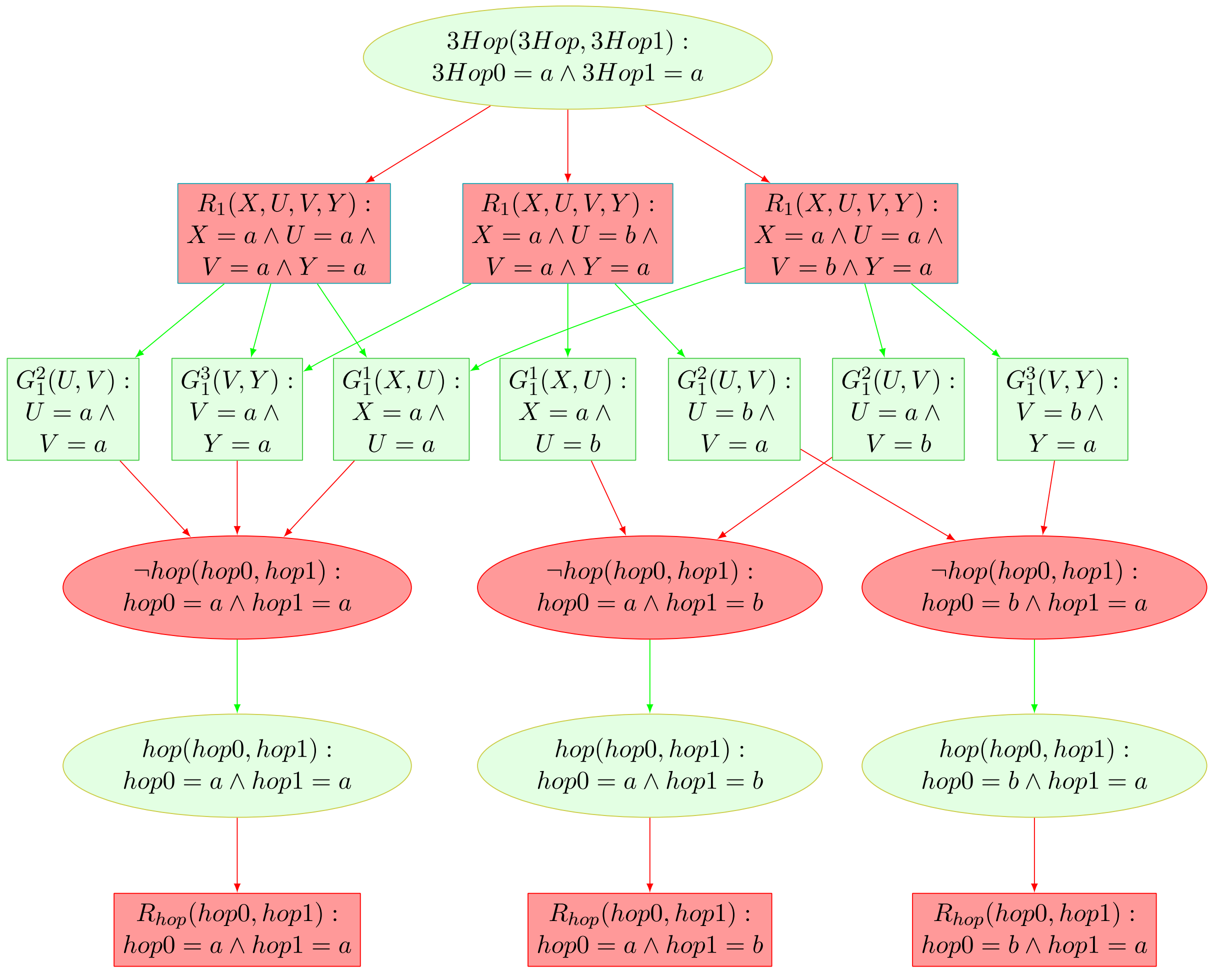

$3Hop(3Hop, 3Hop1)$:
$3Hop0 = a \wedge 3Hop1 = a$
$R_1(X, U, V, Y)$: $X = a \wedge U = a \wedge V = a \wedge Y = a$
$R_1(X, U, V, Y)$: $X = a \wedge U = b \wedge V = a \wedge Y = a$
$R_1(X, U, V, Y)$: $X = a \wedge U = a \wedge V = b \wedge Y = a$
$G_1^2(U, V)$: $U = a \wedge V = a$
$G_1^3(V, Y)$: $V = a \wedge Y = a$
$G_1^1(X, U)$: $X = a \wedge U = a$
$G_1^1(X, U)$: $X = a \wedge U = b$
$G_1^2(U, V)$: $U = b \wedge V = a$
$G_1^2(U, V)$: $U = a \wedge V = b$
$G_1^3(V, Y)$: $V = b \wedge Y = a$
$\neg hop(hop0, hop1)$: $hop0 = a \wedge hop1 = a$
$\neg hop(hop0, hop1)$: $hop0 = a \wedge hop1 = b$
$\neg hop(hop0, hop1)$: $hop0 = b \wedge hop1 = a$
$hop(hop0, hop1)$: $hop0 = a \wedge hop1 = a$
$hop(hop0, hop1)$: $hop0 = a \wedge hop1 = b$
$hop(hop0, hop1)$: $hop0 = b \wedge hop1 = a$
$R_{hop}(hop0, hop1)$: $hop0 = a \wedge hop1 = a$
$R_{hop}(hop0, hop1)$: $hop0 = a \wedge hop1 = b$
$R_{hop}(hop0, hop1)$: $hop0 = b \wedge hop1 = a$

Figure 7.13: Why provenance for $\texttt{3Hop}(\texttt{a}, \texttt{a})$ using constraint provenance games.

ber of explanations has increased. However, the constraint provenance graph provides answers independently of the active domain.

7.6 Conclusions

This chapter, gave an answer to the question: What is the provenance of answers to the game query Q_G? This non-stratified query consists of a single rule:

$$\texttt{win}(\bar{X}) \coloneq \texttt{move}(\bar{X}, \bar{Y}), \neg\texttt{win}(\bar{Y}) \qquad (Q_G)$$

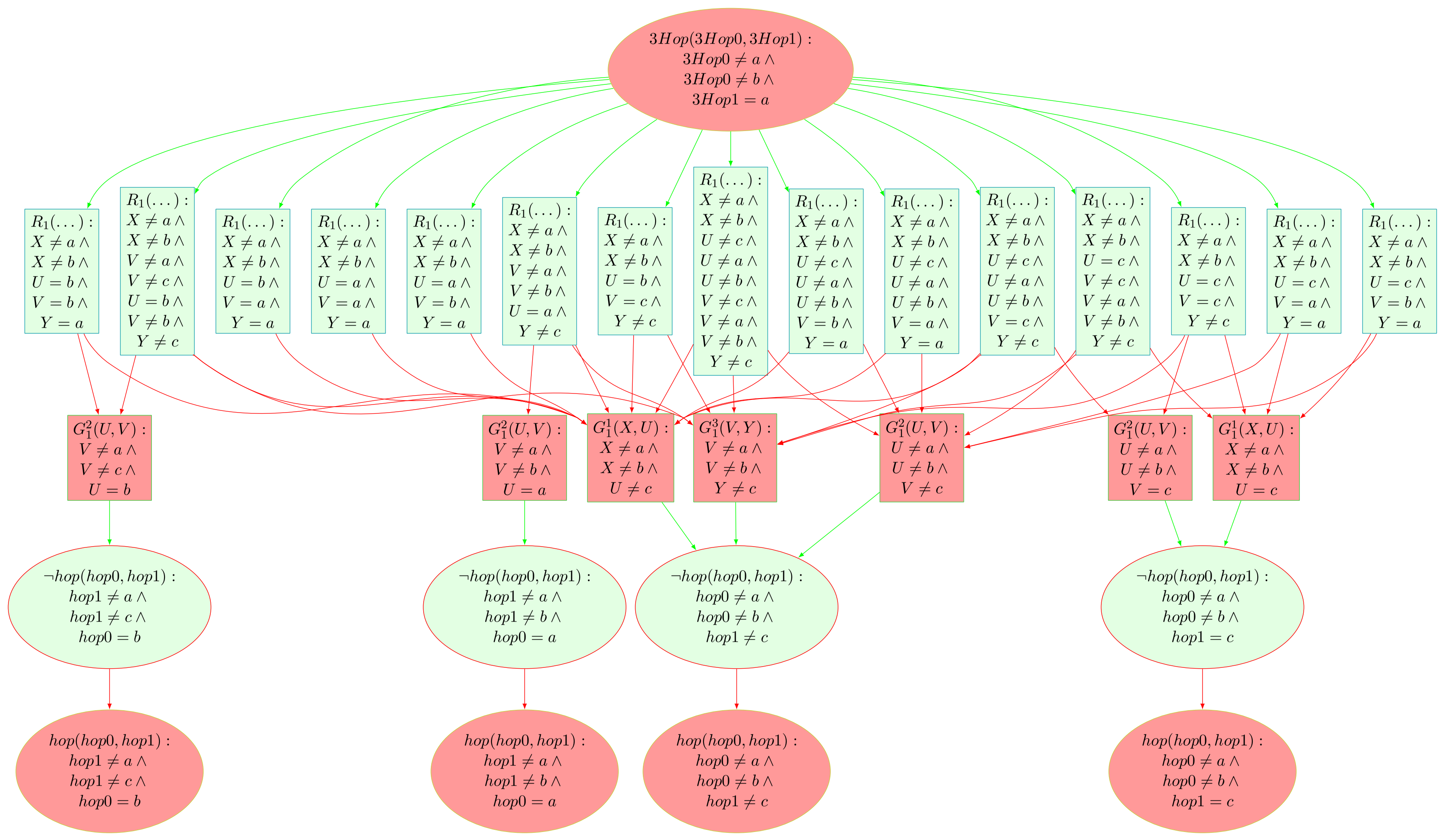

$3Hop(3Hop0, 3Hop1): 3Hop0 \neq a \wedge 3Hop0 \neq b \wedge 3Hop1 = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = b \wedge V = b \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge V \neq a \wedge V \neq c \wedge U = b \wedge V \neq b \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = b \wedge V = a \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = a \wedge V = a \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = a \wedge V = b \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge V \neq a \wedge V \neq b \wedge U = a \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = b \wedge V = c \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U \neq c \wedge U \neq a \wedge U \neq b \wedge V \neq c \wedge V \neq a \wedge V \neq b \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U \neq c \wedge U \neq a \wedge U \neq b \wedge V = b \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U \neq c \wedge U \neq a \wedge U \neq b \wedge V = a \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U \neq c \wedge U \neq a \wedge U \neq b \wedge V = c \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = c \wedge V \neq c \wedge V \neq a \wedge V \neq b \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = c \wedge V = c \wedge Y \neq c$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = c \wedge V = a \wedge Y = a$
$R_1(\ldots): X \neq a \wedge X \neq b \wedge U = c \wedge V = b \wedge Y = a$
$G_1^2(U, V): V \neq a \wedge V \neq c \wedge U = b$
$G_1^2(U, V): V \neq a \wedge V \neq b \wedge U = a$
$G_1^1(X, U): X \neq a \wedge X \neq b \wedge U \neq c$
$G_1^3(V, Y): V \neq a \wedge V \neq b \wedge Y \neq c$
$G_1^2(U, V): U \neq a \wedge U \neq b \wedge V \neq c$
$G_1^2(U, V): U \neq a \wedge U \neq b \wedge V = c$
$G_1^1(X, U): X \neq a \wedge X \neq b \wedge U = c$
$\neg hop(hop0, hop1): hop1 \neq a \wedge hop1 \neq c \wedge hop0 = b$
$\neg hop(hop0, hop1): hop1 \neq a \wedge hop1 \neq b \wedge hop0 = a$
$\neg hop(hop0, hop1): hop0 \neq a \wedge hop0 \neq b \wedge hop1 \neq c$
$\neg hop(hop0, hop1): hop0 \neq a \wedge hop0 \neq b \wedge hop1 = c$
$hop(hop0, hop1): hop1 \neq a \wedge hop1 \neq c \wedge hop0 = b$
$hop(hop0, hop1): hop1 \neq a \wedge hop1 \neq b \wedge hop0 = a$
$hop(hop0, hop1): hop0 \neq a \wedge hop0 \neq b \wedge hop1 \neq c$
$hop(hop0, hop1): hop0 \neq a \wedge hop0 \neq b \wedge hop1 = c$

Figure 7.14: Why-not provenance for `3Hop(c, a)` using constraint provenance games.

The answer to the question is given by a natural and intuitive notion of *game provenance*, which is derived from basic game-theoretic properties of solved games: The value and provenance of a position x depends only on a certain subgraph $\Gamma(x)$ of "good" moves, reachable from x, but is independent of "bad" moves. $\Gamma(x)$ has an elegant regular structure, i.e., alternating *winning* and *delaying* moves for positions that are won or lost, and *drawing* moves for positions that are neither.

Since Q_G is a normal form for fixpoint logic [Kub95, FKL97, FKL00], all fixpoint queries (and in particular all first-order queries FO) can be expressed as win-move games. Inspired by the reduction of query evaluation to games in [FKL97], this chapter then sought to answer the question: Can game provenance be used and applied to query evaluation games, thus hopefully obtaining a useful provenance model for FO queries? It turns out, it can: First-order queries, expressed as non-recursive Datalog$^\neg$ programs, can be evaluated using a simple and elegant game that resembles the well-known SLD resolution. For positive queries this chapter has shown that game provenance coincides with semiring provenance. Moreover, game provenance (unlike semiring provenance) naturally extends to full first-order queries with negation. In particular, a simple form of *why-not* provenance results from the use of a game-theoretic semantics for querying.[12] An extension of the interpretation of provenance games to full Datalog$^\neg$ is achieved by amending the procedure to solve games with an explicit loop detection step. By differentiating the drawn positions in the query evaluation game, the solved game coincides with the well-founded model of Datalog$^\neg$ programs and provides *why* and *wht-not* provenance. Since the construction of provenance games still depends on the active domain, constraints are added to generate domain independent provenance.

[12]See [Hin96], [Hod13], and [Grä11] for other uses of game-theory for query evaluation and model checking.

Chapter 8

Towards a Better Workflow Model

Over the past years, many scientific workflow systems were created and nearly all of them introduced one or more new workflow models as shown in earlier chapters. The earlier chapters have introduced traditional workflow systems. Later chapters provided a closer look on Datalog as a workflow system and as a means to analyze workflows and provenance. Though so many systems are available, scientific workflow systems are still not widely adopted. The reason for this slow adoption are manifold. One could cite the "steep learning curve" associated with describing a scientific workflow in many systems. Another might cite the expected low execution performance compared to customized solutions for each workflow or the lag of integration into distributed computation frameworks such as clouds or grids.

This chapter provides a brief comparison of some workflow systems presented in Chapter 3. For that purpose, criteria are defined that can be used to classify workflow systems. By choosing criteria that are intended to fulfill the expectations presented in Chapter 2, these criteria can also be used to evaluate scientific workflows. To better understand the weaknesses of current workflow systems their models of computation need to be formally analyzed.

Then some case studies are presented that show various aspects of scientific workflow modeling and execution. In this context, some concrete methods to improve the design of scientific workflow systems are identified. The goal is to design a better workflow model and a better execution engine for this model while reusing as many existing components as possible. The use of new model elements is targeted at improving the efficiency of the workflow execution while the workflow language is kept simple.

8.1 Desiderata Revisited

Workflow systems frequently use similar concepts but they are named different, which makes a direct comparison difficult. Timothy McPhillips made an effort to create a naming convention for concepts frequently used in the field of scientific workflows.

This work uses some of these concepts but generalizes and applies them to a broader range of scientific workflow systems. Most of our desiderata are directly related to concepts in models of computation or strategies for workflow execution:

Dataflow oriented. An important comparison criterion is if the modeling of computation is control or data flow oriented. This determines how a user has to think about modeling his scientific workflow. In many cases data and the data processing are the main concern of scientists. Therefore, dataflow oriented MoCs improve SIMPLICITY and UNDERSTANDABILITY of scientific workflows.

Ordered data flow. In a dataflow oriented MoC, another major difference is the order in which data is processed. Data can be processed in the order the data was presented to the workflow or created by other processes. Other approaches can process data in an arbitrary order. Missing restrictions on the order of data allows more optimizations of the workflow execution, thus contributing to SCALABILITY and implicitly to ROBUSTNESS.

Complex data structures. The supported data types vary among different workflow systems. Some systems provide just primitive data types. Others add support for structured types, e.g., lists and records. Some systems also allow more complex types, i.e., they support nested lists and tree structures, e.g., XML documents. The more data types are supported the easier it is to model a variety of different problems in this workflow system, thus contributing to GENERALITY and SIMPLICITY. If the workflow engine is aware of complex data structures, it is able to optimize the workflow execution, e.g., by processing elements of an unordered list in parallel. Thus, complex data structures can also increase SCALABILITY.

Data structure independent design. Some workflow systems have a restricted way to handle complex data type. Some systems use special elements of the workflow description language to implement unnesting of one level of a hierarchical data structure, i.e., converting a list of types into a sequence of these types. In such cases the workflow structure resembles the data structure. Some related work [HKS+08] proposed that this similarity should be the goal of scientific workflow systems. However, this introduces many workflow elements that are not relevant for the actual computations, reducing the readability of the workflow. Thus, a workflow design that is independent of the data structure design increases SIMPLICITY, UNDERSTANDABILITY and ABSTRACTION. In addition, a change in the data structure used does not imply a change in the workflow structure, which improves MODIFIABILITY and REUSABILITY.

Arbitrary accessible data. A very similar criterion is the accessibility of data. The previously described nesting and unnesting of complex data structures limited to one level of the hierarchy only reduces the accessibility of data in the workflow dramatically. But also binding expressions as used in COMAD [DZM+11] can restrict the data access if not designed carefully. More accessible data increases the SIMPLICITY of modeling workflows.

Low Connection Complexity. Based on the MoC a workflow requires different amounts of connections between actors. If complex data types are available the connectivity of the workflow graph is lower. Thus a user can easily create and change a workflow with minimal labor. A lower connectivity increases SIMPLICITY and MODIFIABILITY

Stateful actors. Some workflow system do not allow stateful actors by design. Other workflow systems disallow stateful actors to support certain features. In general, a workflow system should support stateless and stateful actors to allow a high level of ABSTRACTION and GENERALITY.

Statefulness declared. If a workflow system allows stateful actors, it should provide a mechanism to identify them. This allows optimized handling of stateless actors. The support of statefulness annotations increases SCALABILITY and indirectly the ROBUSTNESS of workflows.

Data-driven. During a workflow execution, some systems use a fixed order for invoking actors. This allows scheduling the workflow execution efficiently and removes the overhead for synchronization. However, this approach also restricts parallelism. A better distribution of the workload could be achieved by using a dynamic scheduling based on the availability of data.

Such data-driven architectures improve Scalability.

Multiple invocations. Another concept frequently used is to allow multiple invocations of one actor. In DAGMan every execution of a computational function is a separate job. It is simpler, if one actor is equivalent to one computational function that can be invoked multiple times for different data sets. Therefore, the number of actors is reduced and the Simplicity of the workflow increased.

Concurrent. To improve the Scalability of a workflow, actors can be invoked concurrently. For further improvement not only different actors can be executed in parallel but also multiple invocations of the same actor can be started concurrently.

Streaming. Multiple invocations and concurrency lead to another criterion. Data can not only be passed from one actor to a concurrently invoked actor after the computation is finished but also during invocation. Especially in combination with complex data structures this allows a streaming behavior. Data can be processed more efficiently without wait times, thus further increasing Scalability.

Side-effect handling. During invocation, an actor can cause a variety of side-effects, i.e., it can write files or communication over networks. If a workflow system is aware of side effects or can control them, the Reproducibility and the Robustness of the workflow is increased.

Provenance. In order to help scientists to verify their results workflow systems should provide provenance information. Therefore, general information about an actor's invocation as well as dependencies between input data and output data of an actor are recorded. The availability of provenance increases Reproducibility of workflow results. In addition provenance information can be used to increase workflow Robustness, as presented in Chapter 9.

Workflow evolution provenance. The approach of recording provenance can be extended to the modeling process. Some systems provide provenance information that describes the evolution of the workflow. Changes in a workflow are recorded so that, not only the data can be verified but also the workflow used at that time to produce this data. This

8.2 Comparison of Workflow Systems

Using the criteria defined above, scientific workflow systems can be more easily compared. Table 8.1 lists workflow systems together with the features they provide. COMAD, Restflow and Vistrails have a significant amount of features addressing the requirements on scientific workflows. With the exception of Restflow no scientific workflow systems make statefulness explicit in the workflow description language. Side-effects are not handled by all workflow systems. The criteria presented above and implemented by some scientific workflow systems describe features that improve current systems. The following chapters provide further examples and discussions of scientific workflow systems based on concrete use cases. At the end of this chapter, properties and features identified as essential for a good workflow systems are determined and will form a novel workflow system that addresses the requirements on scientific workflows better.

8.3 Case study: Improving a Monitoring Workflow

Scientific workflow systems were already used to model a variety of problems. An example is the *Monitoring Workflow* developed by Podhorszki et al. [PLK07] using the Kepler workflow system. This workflow is designed to monitor the execution of a scientific workflow pipeline in a supercomputer environment. The Monitoring Workflow runs outside the supercomputer environment and instruments applications within through SSH connections.

Here, the Monitoring Workflow is used for a case study to show the difference between a MoC using a flat data model and one using structured date. A detailed analysis of the initial design reveals weaknesses in the workflow design that are mainly implied by the specific model of computation used. To improve the Simplicity and Understandability the same workflow is modeled using the COMAD MoC that better fits the workflow structure. During the remodeling process the workflow structure was extremely simplified but two workflow concepts were found to be hard to express in COMAD as well. To resolve the difficulties in the COMAD modeling process, this section presents some extensions to the COMAD MoC.

The remaining section is organized in the following: Section 8.3.1 presents the model of the Monitoring workflow in more detail and weaknesses in the design described. Section 8.3.2

describes the remodeling of the workflow using COMAD. Limitations of this approach are then addressed with novel extension in Section 8.3.3. Finally, Section 8.3.4 concludes this section with a short summary.

8.3.1 Workflow Analysis

The Monitoring Workflow was originally modeled by Norbert Podhorszki using a PN director on the top-level and embedded SDF as well as the dynamic data flow (DDF) models of computation. The resulting model is shown in Figure 8.1.

The workflow starts with multiple initialization actors that set up variables and directories on remote hosts. After this step, the workflow describes a *polling loop*, checking the running status of the scientific workflow computation on the supercomputer. Within this outer loop several workflow pipelines are implemented. In order to respond to the completion of the scientific computation monitored, one pipeline just monitors the computation and updates a parameter value with this status. All other pipelines evaluate this parameter value at various points and react differently on the end of the external computation. Note that there is no visible connection between the parameter and all actors that set or read its value, thus forming a "hidden" communication link. Such hidden information flows reduce the SIMPLICITY as well as the UNDERSTANDABILITY of a workflow significantly.

All inner pipelines require a periodic directory listing from remote hosts and use the retrieved file names for further analysis. An important detail is that, explicit stop markers are encoded into a file name in order to indicate the end of a file listing, i.e., stop files. This stop files introduce a significant number of model elements that either evaluate them or filter them out for certain computations, thus extremely worsening the SIMPLICITY of the overall workflow.

The actual computations of the Monitoring Workflow in form of image and video file creation as well as data archiving or performed at the end of each pipeline and on the bottom of the hierarchical workflow structure.

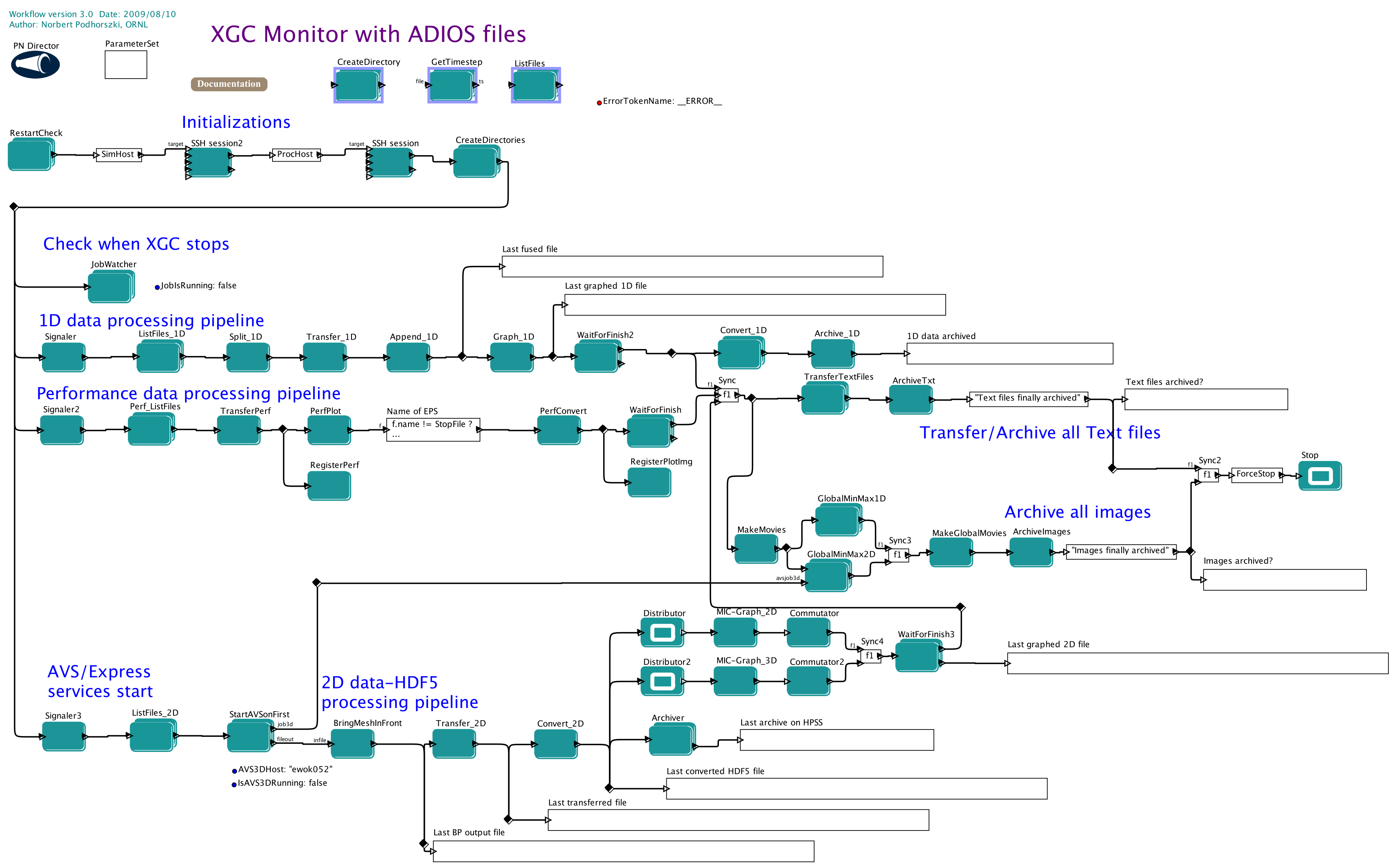


Figure 8.1: Original Monitoring Workflow: The top-level PN workflow shows all steps of the workflow. Additional parameters are contained in the parameter set in the top left corner. In the upper left corner, the workflow starts with initialization actors. Four pipelines that use a polling technique to observe a job execution follow: The first pipeline just contains one actor that observes the job execution and adjusts the flag “JobIsRunning” according to the status of the job. The other three pipelines start with a sampling actor that continuously creates a trigger value until the “JobIsRunning” parameter value changes to false.

8.3.2 Workflow Re-modeling with COMAD

After the analysis of the original Monitoring workflow, the workflow was re-modeled using the COMAD model of computation. COMAD is build upon a PN execution model that determines when actors are invoked. The data model of COMAD uses complex data structures similar to XML documents, i.e., COMAD *collections*, on the workflow level. These collections are split into its elements and streamed through the workflow. Since all data is enclosed in one data stream, conventional COMAD actors require only one input and one output port, thus simplifying the workflow design.

In the redesigned model, the basic workflow structure was conserved, but the individual pipelines inside the polling loops were modeled as linear COMAD workflow graphs. Most original actors that performed actual computations were encapsulated in COMAD sub-workflows, i.e., *composite coactors*. The original actors mainly used parameters as input data, thus hiding the actual dataflow. A re-implementation of those actors was not practical in the given time, but it would have improved the usability greatly. Instead, each composite coactor received the input data through ports from the stream and converted this data using additional modeling elements to parameters.

Directory listings can naturally be represented as COMAD collections of files, making *stop files* unnecessary. Collections are handled internally by the workflow system but can be processed as a stream to allow a efficient execution. Furthermore, COMAD directly supports filtering of collection contents trough *binding expressions*, improving workflow SIMPLICITY. The remodeled COMAD monitoring Workflow is presented in Figure 8.2

Polling processes, i.e., workflows repeatedly reading values to react on a change of the value, are hard to model in COMAD, since the workflow graph is typically linear, thus not allowing any feedback of downstream actors to those upstream. One existing solution is to implement the polling process within one actor where information flow is unrestricted. However, this allows no REUSABILITY of existing actors to facilitate the polling process. Another approach currently supported is a *loop* in the workflow. The loop actors merge two COMAD streams based on their history together. First, a defined part of the data stream is send around the loop until the loop condition is false. Then the data stream processed in the loop is embedded into the

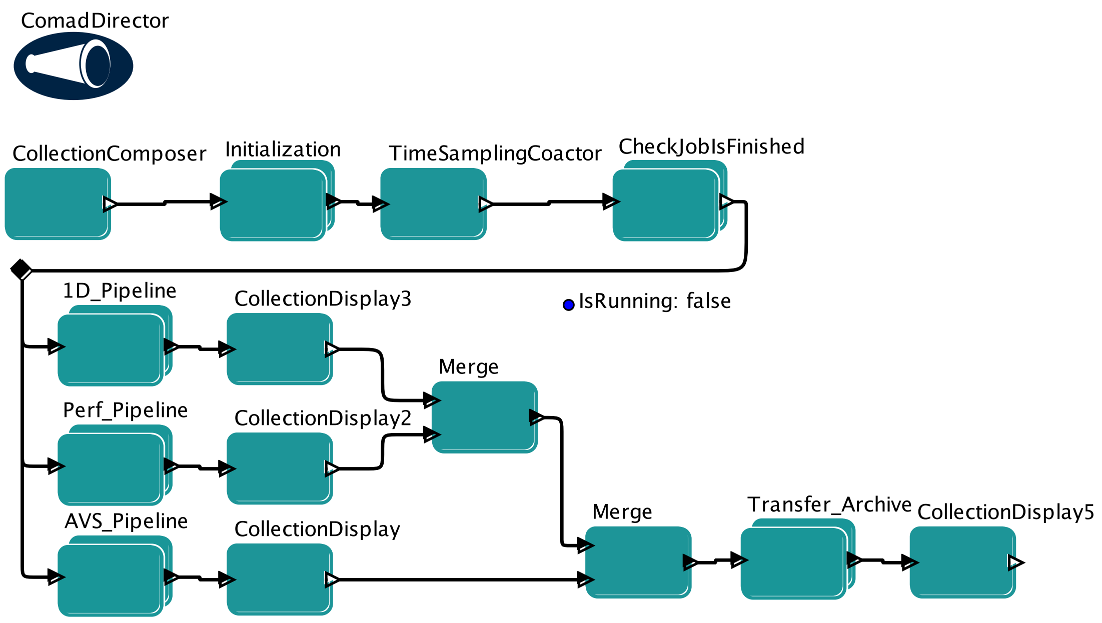


Figure 8.2: Remodeled Monitoring Workflow using COMAD: The top-level model uses the COMAD director. Composites use SDF and DDF directors, but the handling of explicit *stop files* is not required anymore.

complete stream. To support this stream merge operations the streaming in the COMAD model is restricted, which reduces the SCALABILITY.

8.3.3 Extensions of COMAD

During the re-modeling process several shortcomings in the implementation of Kepler and the COMAD MoC were identified. This section describes extensions of the Kepler GUI and a proposed extension of COMAD to further improve workflow SIMPLICITY.

Parameter Data Binding.. To simplify embedding of existing actors into COMAD's composite coactors, the workflow definition language was extended. During the construction of a composite coactor the user has to define if the streamed input data will be bound to an input port within the composite or if an internal parameter will be created and updated with the current value in the data stream. A new flag in the *data bindings* of COMAD now switches between both options.

Crossing Actor.. In order to support a more efficient handling of feedback loops, this work proposes a modified loop construct for COMAD. The *crossing actor* shown in Figure 8.3 supports

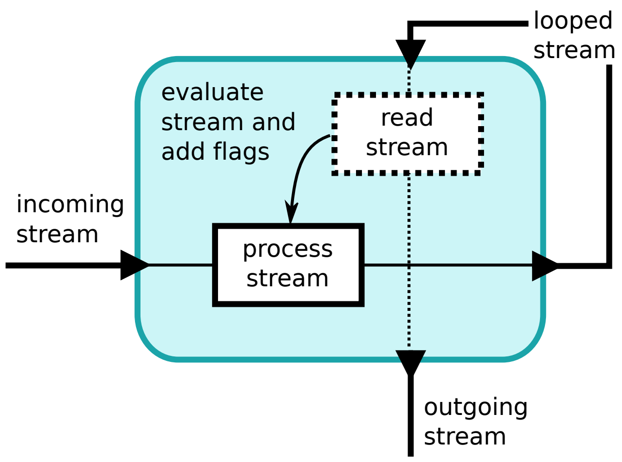


Figure 8.3: Crossing actor: The data stream is passed through into the loop and later read. The stream is not directly modified at this point but additional flags are introduced into the main stream passing through. This allows the construction of a simple loop construct with better streaming support.

polling in streaming workflows. Instead of merging the looped with the main data stream, passes the main data stream on into the loop. Then, it reads the looped stream on another port and passes it on without modifications. If the observed value was changed by an actor in the loop, the *crossing actor* can inject any data into the main stream that will then influence the downstream behavior. This approach avoids blocking the stream and allows modeling the flow of data explicitly.

Drag and Drop.. Finally, the GUI can exploit the simple workflow structure imposed by COMAD MoCs. With the exception of a very few special cases, all actors have two ports. The input port is always labeled “input” and receives the data stream. Analogously, the output port is always labeled “output” and sends out the whole data stream. This design makes it easy to insert actors into a workflow or replace existing actors. To improve workflow SIMPLICITY, the Kepler user interface was extended. Dragging and Dropping of actors on a channel connecting two actors will automatically insert the new actor between these two actors. Input and output ports are automatically reconnected.

A similar extension was created to allow an easy replacement of actors, e.g., to substitute an analysis step in a scientific workflow by an updated version. If an actor is dragged and dropped over another actor, the existing actor is removed from the workflow and the new actor is inserted at the same position. Input and output ports can easily be reconnected based on their names. In addition, parameters that were set on the existing actor are mapped to parameters for the new actor. First, parameters are compared using their names. Then, the data types of those parameters are compared and if they match as well the parameter value is set as a default for the parameter in the new actor as well.

8.3.4 Summary

By using the COMAD MoC and extending it by the features described above, the Monitoring Workflow description was simplified. A comparison using various quantitative measures is shown in Table 8.2. These results show how the choice of a MoC can influence the Simplicity and Understandability of scientific workflows. To be able to make the correct choice of different MoCs, a user needs a deeper understanding of multiple MoC. While this acceptable for simulation and general modeling purposes, this should not be imposed on the user by a scientific workflow system.

8.4 Case study: Growing Degree Day (GDD) Workflow

This sections presents a Kepler workflow published in [KGC+12] performing sliding window calculations on streaming data.

8.4.1 Introduction

In environmental science, data often takes the form of historical or real-time continuous data streams rather than static data sets. Processing and analysis of such data often requires grouping of values within a given time window and subsequently applying aggregation functions. In fact, those operations are a common component in any stream processing.

The REAP project [REA12] developed an example workflow in Kepler [BAJ+10] to integrate sensor networks and to calculate Growing Degree Day (GDD) for a particular sensor. GDD is a measure of heat accumulation used to predict plant maturity. The crop development rate from emergence to maturity reliably depends upon the accrued daily temperatures. GDD is

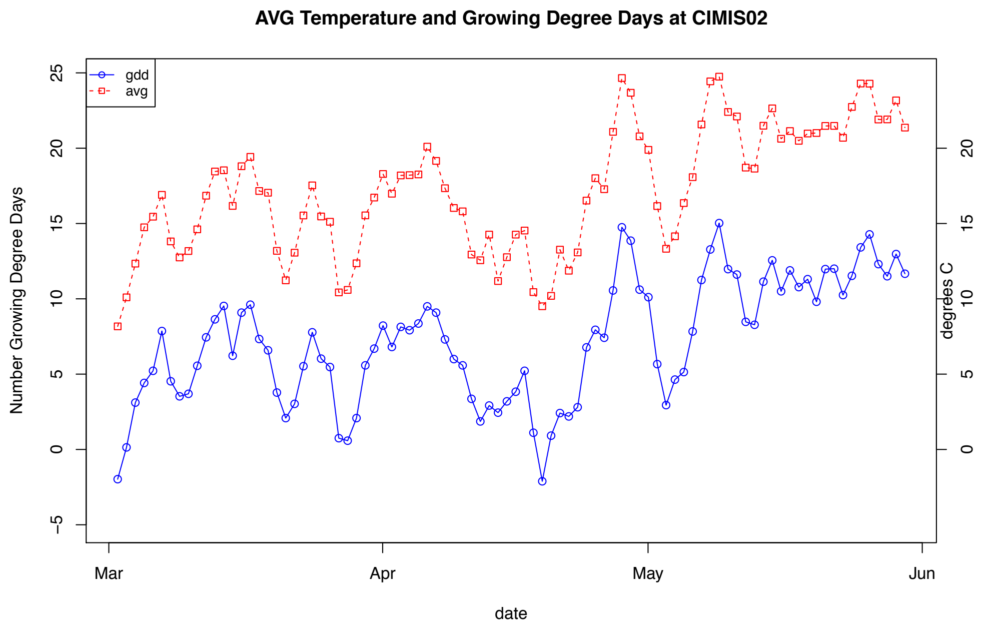


Figure 8.4: Plot of average temperature (upper curve with scale on the right) and GDD (lower curve with scale on the left) for CIMIS02 station.

important in other agricultural studies including pest and weed management, and irrigation scheduling [BE69, ZGW+83]. The GDD measure is calculated as follows:

$$GDD(T_{min}, T_{max}) = \begin{cases} 0 & \text{if } T_{max} < T_{base} \\ \frac{T_{min}+T_{max}}{2} - T_{base} & \text{if } T_{base} \le T_{max} \le T_{top} \\ \frac{T_{min}+T_{top}}{2} - T_{base} & \text{if } T_{top} < T_{max} \end{cases}$$

T_{min} and T_{max} are the day's minimum and maximum temperatures. T_{base}, the base temperature, varies by crop type or application and sets a minimum threshold on the effect of the day's contribution to GDD. Similarly, T_{top} acts as an upper bound on the contribution to GDD for a single day. It is usually capped at 30 °C because most plants do not improve their growth beyond that temperature. Figure 8.4 shows an example plot of the GDD and average temperatures for a time window from the CIMIS02 [Cim12] sensor station.

The conventional Kepler workflow design as developed in the REAP project implements the GDD computation, but also has a number of limitations. For example, grouping of data and therefore aggregation functions can only be applied to time windows of fixed length. Furthermore, overlapping windows and gaps between windows are not realizable. S. Gulati [Gul10] developed the concept of a new actor that supports freely definable time-based sliding window computations and efficient aggregation of streaming data. This case study presents the main concepts of the "`Chunker`" actor that was developed for Kepler. Furthermore, it describes limitations of the conventional design on the GDD workflow example and describes how the design can be improved to fully support streaming.

The rest of this section is organized as follows. Section 8.4.2 describes the conventional design of the GDD workflow in more detail. Section 8.4.3 introduces the new `Chunker` actor. In Section 8.4.4 a new workflow design using the `Chunker` actor is demonstrated.

8.4.2 Conventional GDD Workflow Design

The conventional GDD workflow design implemented in Kepler as shown in Figure 8.5 calculates GDD from temperature measurements and plots it. The sensor data source, here a relational PostgreSQL database, outputs measurementsas tuples (t_i, d_i) where temperatures constitute a data field d_i of the record and the time t_i associated with the temperature reading constitutes a timestamp field for the record. As daily minimum temperatures and daily maximum temperatures are needed to calculate GDD, the incoming data stream is grouped into daily time windows of fixed length using the `Sequence_to_Array` actor where each array contains a single day's temperature measurements. This method can only be applied for regularly spaced inputs where the beginning of a day is known a priori. Kepler inherits aggregation actors from Ptolemy that operate on a single array input at a time. Using such array-based aggregation, the minimum, maximum and average temperatures can be calculated easily: First, the minimum, maximum and average values are computed for each array of temperature readings form a whole day Then, the GDD is computed using an R actor. The R actor in Kepler works on a single input tuple: An array containing all readings is read from an input port. Then, it is converted to an R object, a user-defined script is executed, and the result is emitted on the output port. Finally, the GDD is plotted against timestamps with a plotting actor. This workflow will execute properly if it is guaranteed that the number of temperature readings is constant every day. However, in a realis-

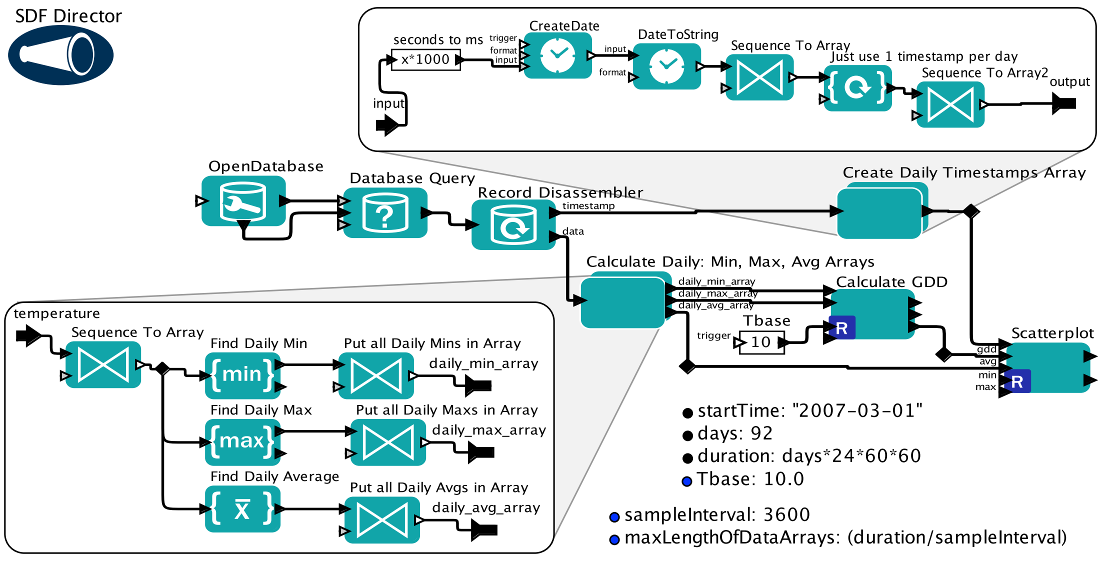


Figure 8.5: Conventional "Token Counting" GDD workflow The data stream is split into non-overlapping windows of fixed size by counting data tokens using `Sequence_to_Array` with a statically defined data token count. Here, the maximum length of data arrays is calculated from constant parameters provided in the lower right of the workflow graph.

tic stream model the rate of measurements may vary, e.g., due to dropped readings or defective sensores. This requires special handling and a strategy to fill "gaps" in the data stream. The `DataTurbine` actor, for example, includes a "fillGaps()" for that purpose.

Aggregation by Token Counting. The `Sequence_to_Array` actor used in the GDD workflow of Figure 8.5 divides the incoming stream into arrays with a given, fixed number of records. The basic principle of this actor is shown in Figure 8.6. One incoming port receives a stream of data $D = d_1, d_2, \ldots$ indexed by timestamps t_i. The actor groups incoming data items into groups of a fixed size specified by parameter s. Grouping of data starts immediately from receiving the first token and thereby implicitly with the timestamp of the first data tokens b. There is no integrated mechanism to introduce gaps between windows while grouping. After receiving the specified number of data tokens, the completed window is output as an array.

Due to the fixed parameters for grouping, windows of different size, as in the case of Window 1 and Window 4 shown in Figure 8.7, are not possible. In the conventional *"Token Counting"* approach, the incoming stream is split into arrays. Thus, it is not possible to insert a data item into multiple time windows preventing overlapping windows as shown for Window 1 and Window 2 in Figure 8.7.

The conventional GDD workflow design also uses the synchronous dataflow director (SDF). This statically scheduling director executes all actors serially according to a precalculated schedule. Thus, it does not allow a variable number of data tokens emitted by an actor on a channel in different scheduling rounds within the same workflow execution. Therefore, the use of the SDF director is another reason preventing the use of variable windows, since the array to sequence actor is required to output an array containing a window in a constant interval after consuming a constant number of tokens from the data stream.

Finally, the R actor used to generate a plot requires all data to be available before the actor can create a graph. Thus, it is not possible to display results in a continuous stream but rather this requires the data stream for GDD measures to terminate.

8.4.3 Chunker Actor: Flexible Window-Based Grouping

In context of scientific workflows, the goal is to develop a single grouping actor that is simple enough to be used easily in a number of scenarios, but at the same time is general enough that it could be reused in a number of different workflows. To that end, the concept of a `Chunker` actor shown in Figure 8.8 was developed for general grouping of streaming data into given time windows. A `Chunker` implementation is available for the Kepler scientific workflow system at [Com12]. The actor can be combined with different computational models including SDF, PN and Comad as shown in [DZM+11].

Window based Approach. Given two input streams, a data stream D and a window stream W, the `Chunker` will group input data for each window. The data stream is a sequence of pairs

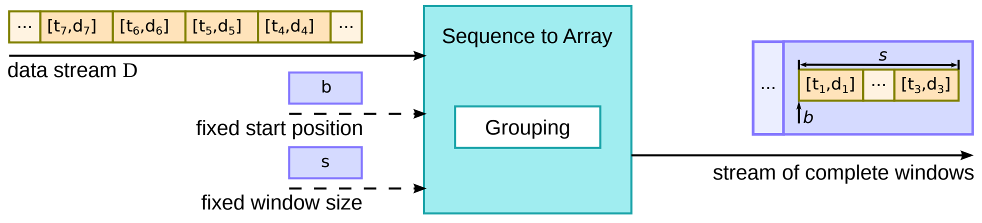


Figure 8.6: Grouping by "Token Counting" with `Sequence_to_Array` actor Incoming data tokens are counted and starting from position b the temperature data stream is split into windows of fixed size s. Thus, grouping of data tokens depends on the correct position in the stream and windows cannot be constructed based on the timestamp t_i associated with data stream.

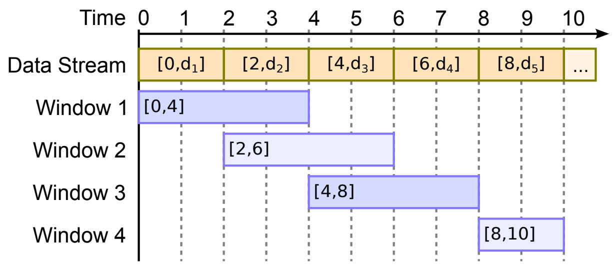


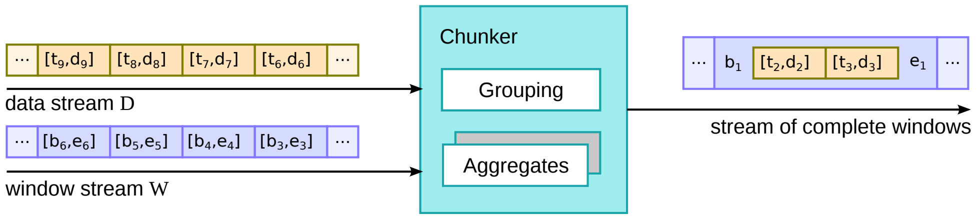
Figure 8.7: Examples for desirable time windows Window 1 and 3 are consecutive, non overlapping and of fixed size. Window 4 has a different window length that cannot be generated using `Sequence_to_Array` because of a fixed parameter for the window size. Window 2 represents an overlapping window, which cannot be realized with `Sequence_to_Array` because the fixed window sizes also determines the start position of the following window.

(t_i, d_i) containing a timestamp t_i and a value d_i. The window stream is another sequence of pairs $w_i = (b_i, e_i)$ containing the beginning b_i and the end e_i timestamp of the window. For each window in W, the `Chunker` will store the window in a list of active windows if the start time b_i is smaller than the current time stamp in the data stream. For each active window $w = (b, e) \in W$ the actor checks if incoming data is contained in that window, i.e., $b \leq t_i \leq e$, and if so the data is stored with this window.

Since both D and W are possibly infinite streams, an order needs to be imposed on the timestamps t_i in D and on the start time of windows b_i in W. Thus, groups can be built and output without storing a possibly infinite amount of data that would otherwise be stored in order to complete windows with start times reaching far back in time. Note that there is no requirement for an order on ending times of windows. In fact, for the `Chunker` actor, windows in W can overlap in an arbitrary manner. Whenever the timestamp t_i of the data stream reaches or exceeds the end time e_i of a stored window, this window can be output and removed from the list of active windows.

Since grouped data is frequently used in aggregation functions and windows can be large, our `Chunker` actor computes aggregates over windows incrementally. The workflow developer can choose if either the complete data for a window, a selection of aggregates, or both should be output. The algorithm supports all standard aggregations, that can be computed using an initialization state, and function that updates the state based on incoming data, and a function that finalizes the aggregation state for output. Currently supported aggregates are: count, sum, average, maximum, minimum, and array. The **array** aggregation collects a stream of input data, into a single array for each window. This is essentially emulating the behavior of the Sequence_to_Array actor described earlier but with support for variable-sized windows.

The Chunker implementation in Kepler requires tokens of type record on input streams. The window stream records need to provide an attribute *window* that holds a tuple containing the beginning and end time of the window. The data stream records need to provide at least two attributes, the *timestamp*, and the value. When Kepler sets up the workflow in preparation for execution, input and output ports are created to handle all streams. In addition, a priority queue is instantiated that is used to maintain the order of windows and as a result, windows are output ordered by window's end timestamp or at their earliest possible computation. An AggregationFactory is provided as a generic interface to any aggregation. New aggregates are derived from the AggregationFactory by calling the *create()* method as shown in Algorithm 8.4.1. Each aggregate has *initialize*, *update* and *finalize* methods used for setup, incremental update and final computation of the aggregate function. The *finalize* method is invoked after all the



Chunker
Grouping
Aggregates
... [t9,d9] [t8,d8] [t7,d7] [t6,d6] ...
data stream D
... [b6,e6] [b5,e5] [b4,e4] [b3,e3] ...
window stream W
... b1 [t2,d2] [t3,d3] e1 ...
stream of complete windows

Figure 8.8: Flexible, window-based Grouping Incoming data tokens from D are matched with all incoming windows W. If the timestamp t_i of a data token is enclosed in window $j : b_j \leq t_i \leq e_j$ then the value of this data token is grouped with this window. If configured by the user, the value is also used to update aggregations stored for the window. Once the end time of window e_j has passed, i.e., incoming data tokens have a greater timestamp, aggregations are finalized and the completed window is output.

required data has been processed and returns the aggregates final value.

Algorithm 8.4.1: FIRE()

```
$cur \leftarrow data.\textsc{getNext}()$
while $windows.\textsc{hasNext}()$ and $cur.\textsc{timestamp}() \geq windows.\textsc{peek}().\textsc{start}()$
  do {
    $win \leftarrow windows.\textsc{getNext}()$
    $newAgg \leftarrow aggFac.\textsc{create}()$
    $newAgg.\textsc{initialize}()$
    $pq.\textsc{insert}(win.\textsc{endTime}(), newAgg)$
  }
while not $pq.\textsc{isEmpty}()$ and $pq.\textsc{minKey}() < cur.\textsc{timestamp}()$
  do {
    $minAgg \leftarrow pq.\textsc{getMin}()$
    $out.\textsc{broadcast}(minAgg.\textsc{finalize}())$
  }
$pq.\textsc{apply}(\textsc{update}(cur.\textsc{value}()))$
```

Algorithm 8.4.1 is implemented in the *fire* method of the `Chunker` to compute groups and sliding window aggregates. First, a data token is accepted from the data port and stored in the variable *cur*. *cur* is checked to see if it is contained in a window enqueued on the channel. The check is performed by comparing the *cur* timestamp with the window's start timestamp and if it is greater, then the window is retrieved from the *windows* port. For each retrieved window a new aggregate is created in *newAgg* and subsequently initialized, e.g., the aggregate sum is initialized with 0. Then the initialized aggregate *newAgg* is stored in the priority queue *pq* sorted by increasing end timestamp of *win*. Note that if *cur* timestamp is greater than *win* end timestamp, then the window *win* is still inserted into *pq*. The algorithm continues to process windows until the *cur* timestamp is less than the next windows start timestamp. After checking for windows, the algorithm processes all windows that are now present in the priority queue and sorted in increasing order of end timestamps. While the heap is not empty and *cur* timestamp is greater than the stop timestamp (*MinKey*) of the entry in front of the priority queue, an aggregate is retrieved from the queue, finalized by calling the corresponding function and output. In the last stage of the algorithm, the *APPLY* method is executed to update the aggregates of every remaining window in the heap with *cur* value. The *postfire* method of the `Chunker` actor returns true so that the fire method will be invoked again to process the next data token.

8.4.4 Growing Degree Days Workflow Using A Chunker

To demonstrate the functionality of our `Chunker` actor, it was used to model the GDD workflow. The workflow shown in Figure 8.9 demonstrates processing of sliding window aggregate queries on stream of hourly temperatures. The data stream is generated by querying a database containing temperature readings together with a timestamp. The `WindowsGenerator` actor creates a stream of windows defined by parameters specifying a start time and a variable length. Multiple `WindowsGenerator` actors can be used to create overlapping windows. For the purpose of calculating and plotting daily GDD and daily temperature averages windows are generated for the start of every day with a window size of one day. To calculate monthly averages overlapping windows with a start time of the first day of a month and a size to cover the whole month are used.

Another limitation of the conventional workflow design is the use of an R actor to calculate and plot the GDD data. The `Scatterplot` R actor of Figure 8.5 has no streaming capabilities, so that all required data has to be available before graphs are drawn. In the new design, a generic `Expression` actor is used, which is able to process streaming data in order to calculate the GDD values. Furthermore, `SequencePlotter` actor is used to display the final graphs. This actor supports streaming data and updates the graph view whenever new data is received.

Finally, the new workflow design uses a PN director that executes every actor of the workflow in a separate thread and therefore in parallel. In contrast to the conventional GDD workflow design, this does not imply a fixed number of tokens flowing over channels since actors work independently and are only synchronized by writing to and reading from FIFO queues representing channels.

The workflow is considerably simpler than the one in Figure 8.5, and aggregations can be

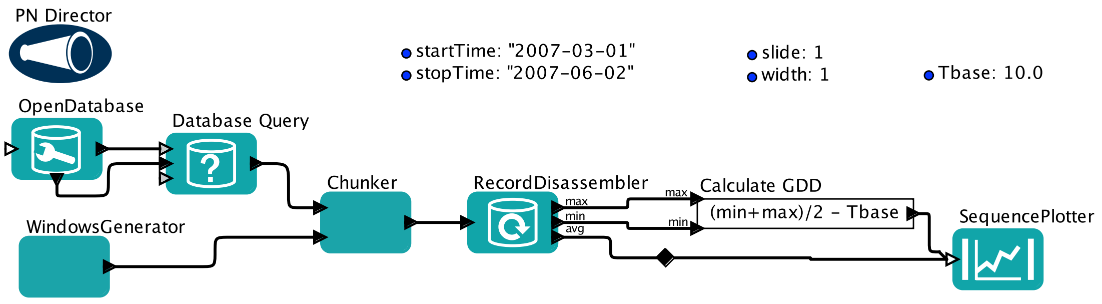


Figure 8.9: New GDD workflow design with `Chunker` actor Through two separate streams, one containing data and the other containing user-defined windows, the `Chunker` actor can group data into arbitrary windows. Here, the actor is configured to output only the running min, max, and average over windows and not the window data itself. This data is subsequently extracted from the output array by a *RecordDisassembler* and plotted in a similar design as before but with streaming to allow continuous monitoring.

calculated independently of the number of measurements present every day, making the workflow more robust. The results of the computation with a time window of one day are equivalent to the one produced by the original workflow.

8.4.5 Summary

Kepler is a scientific workflow management system that exhibits a common problem in supporting the calculation of aggregations in scientific workflows. There are no mechanisms for computing sliding window aggregates based on timestamps apart from the very limited approach of counting tuples in the data stream. A more general `Chunker` actor has been developed that can process aggregations based on timestamps. The `Chunker` actor has shown to be a very useful tool in the world of scientific workflows, and Kepler in particular, for computing sliding window aggregates. It has eliminated the need for `Sequence_to_Array` based constructs employing token counting and instead directly supports extracting data from a stream within arbitrary time windows. Furthermore, this actor directly supports a selection of common aggregation functions in an efficient streaming manner.

As demonstrated for the GDD workflow, the `Chunker` actor provides much more flexibility than the conventional token counting approach in creating workflows for answering sliding window queries. Along with standard aggregates min, max, sum, count and average, an `array` aggregate is provided that collects a stream of input data in a single array for each window, corresponding to those data within that window.

Also, this case study demonstrated that extensions to the library of actors of existing workflow systems are frequently necessary and replace missing features in the scientific workflow system. Furthermore, it showed that the support for streaming is not obvious but critical for completing the task in a real world scenario.

8.5 Case study: MotifCatcher

This section sketches a Kepler workflow for improved motif detection in large sequence sets with random sampling that appeared in [KSFL12]. The discovery of functionally significant short, statistically overrepresented subsequence patterns (motifs) in a set of biological sequences is a challenging and important problem. Oftentimes, not all sequences in the set contain a motif. When using traditional methods these non-motif-containing sequences complicate the algorithmic discovery of motifs. MotifCatcher [Sof12] is a framework developed by Seitzer et al. [SWLF12] that extends the sensitivity of existing motif-finding tools by applying random sampling to input sequences. First, multiple subsets of sequences are randomly constructed. Some of these subsets have a large number of motif-containing sequences. Traditional motif-finding tools are applied to all subsets in parallel, and significant motifs are recovered from appropriate subsets. Finally, this framework returns candidate functionally significant motifs and organizes them into a tree. which allows further analysis. However, the current implementation of MotifCatcher does not scale to large sets of sequences. This is due to the fact that the current implementation is not suited for distributed computing, and the whole sequence set is kept in memory. In order to achieve better scalability, implementation was redesigned by the author using the Kepler scientific workflow system that addresses those shortcomings. Kepler functions as a convenient front end to encode the computation into a Map Reduce pattern [WCA09] to archive high parallelism for computationally intensive steps. Furthermore, Kepler greatly simplifies the substitution of the motif finding algorithm used on each subset. By using the scientific workflow system and a specialized MoC, the maximum size of the sequence sets in which motifs can be discovered and the number of subsets processed in parallel could be increased significantly.

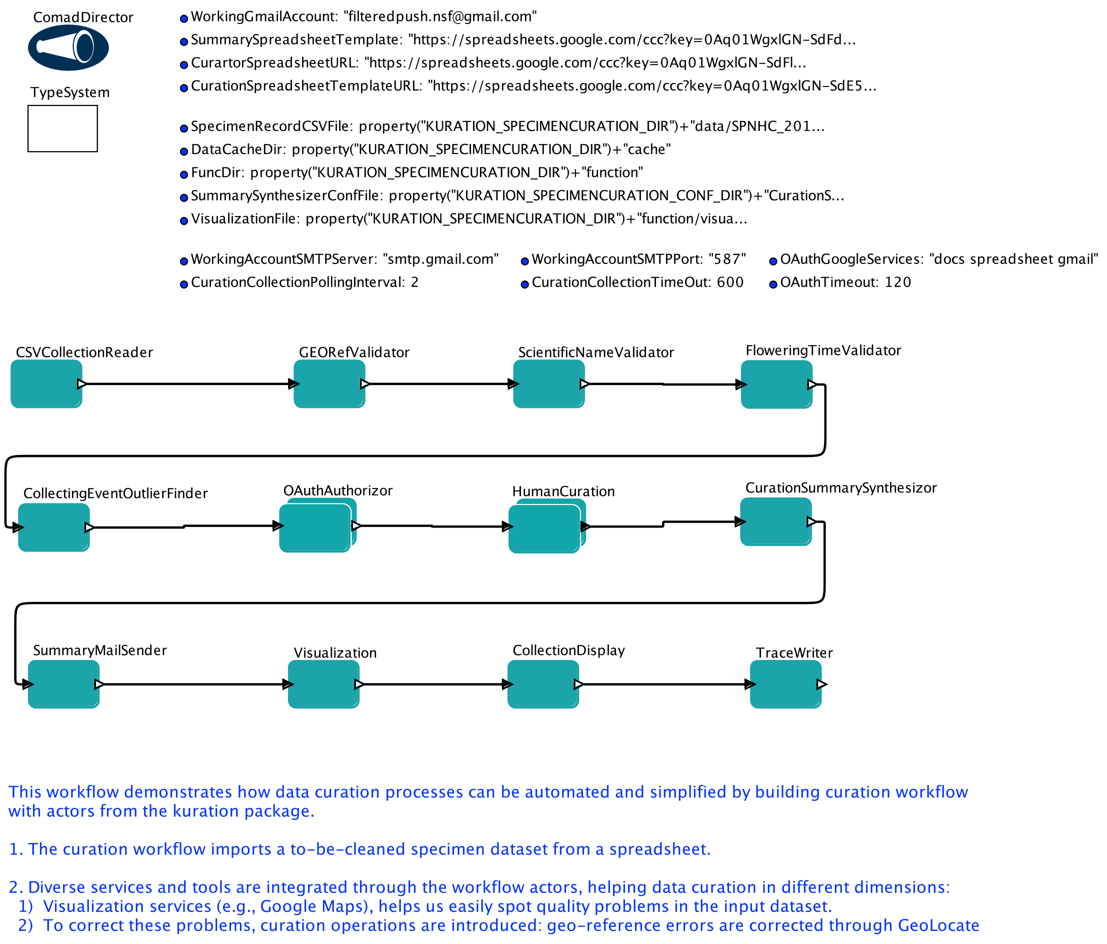
8.6 Case study: Kuration Workflow

This section briefly describes impressions gathered in the process of modeling and optimizing the "Kuration" workflow used in the FilteredPush project. The workflow is used to automate the curation process of specimen collection data of museums. The input to this process are records of specimen collection events in various formats, e.g., Darwin-Core. In some cases, this records have quality problems or simple errors introduced for example by mistakes when entering data into a database. To curate, i.e., detect and resolve, such quality issues a number of checks are performed on single records or groups of records. In some cases, this curation steps cannot be performed automatically and require user feedback. In order to automate the curation process a workflow was modeled in the Kepler scientific workflow system. This Kepler based curation ("Kuration") workflow was initially modeled using COMAD and is shown in Figure 8.10. In addition to the automation, this workflow also provides a platform for easy modification and extension of the workflow as well as provenance.

The workflow reads Darwin-Core records from a CSV file and curates each entry using the following steps: First, geographic coordinates in latitude and longitude a verified or corrected based on country, state, county and location by using a web service. Then, the scientific name is curated using another web service. Then the flowering time is verified. The `CollectionEventOutlierFinder` compares a group of records with similar characteristics to identify outliers. All following steps are used to create a web-accessible spreadsheet that suggests curation steps and allows a user to perform manual corrections. Finally, the resulting corrections are sent out as notifications and the records are visualized and stored in a trace file.

Since the last steps require additional services and manual intervention, a smaller workflow just consisting of the first three curation steps was created. This automated curation workflow is shown in Figure 8.11. In contrast to the original workflow, it reads Darwin-Core records from a database and writes curated records and provenance back to a database.

When executing the automated workflow on a relatively large dataset of approximately 4000 records the workflow execution slows down significantly. Furthermore, the workflow system becomes unstable and frequently crashes after using up all available memory of a system with 4GB of main memory. As a reason for the slow execution and the high memory usage the



● WorkingGmailAccount: "filteredpush.nsf@gmail.com"
● SummarySpreadsheetTemplate: "https://spreadsheets.google.com/ccc?key=0Aq01WgxlGN-SdFd...
● CurartorSpreadsheetURL: "https://spreadsheets.google.com/ccc?key=0Aq01WgxlGN-SdFl...
● CurationSpreadsheetTemplateURL: "https://spreadsheets.google.com/ccc?key=0Aq01WgxlGN-SdE5...

● SpecimenRecordCSVFile: property("KURATION_SPECIMENCURATION_DIR")+"data/SPNHC_201...
● DataCacheDir: property("KURATION_SPECIMENCURATION_DIR")+"cache"
● FuncDir: property("KURATION_SPECIMENCURATION_DIR")+"function"
● SummarySynthesizerConfFile: property("KURATION_SPECIMENCURATION_CONF_DIR")+"CurationS...
● VisualizationFile: property("KURATION_SPECIMENCURATION_DIR")+"function/visua...

● WorkingAccountSMTPServer: "smtp.gmail.com" ● WorkingAccountSMTPPort: "587" ● OAuthGoogleServices: "docs spreadsheet gmail"
● CurationCollectionPollingInterval: 2 ● CurationCollectionTimeOut: 600 ● OAuthTimeout: 120

This workflow demonstrates how data curation processes can be automated and simplified by building curation workflow with actors from the kuration package.

1. The curation workflow imports a to-be-cleaned specimen dataset from a spreadsheet.

2. Diverse services and tools are integrated through the workflow actors, helping data curation in different dimensions:
  1) Visualization services (e.g., Google Maps), helps us easily spot quality problems in the input dataset.
  2) To correct these problems, curation operations are introduced: geo-reference errors are corrected through GeoLocate

Figure 8.10: Kuration workflow modeled in COMAD.

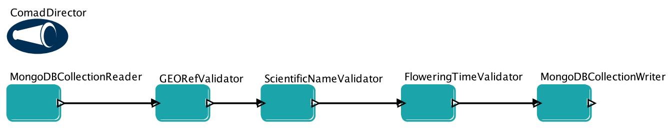


Figure 8.11: Small automated Kuration workflow.

architecture and implementation of COMAD could be identified. Figure 8.12 illustrates the principal of executing a small COMAD workflow with two actors `A` and `B`. COMAD is based upon the the PN model of computation and inherits some of its drawbacks described in Chapter 3. However, the size of queues in COMAD is not limited and grown on demand. This mitigates the performance problem (1) illustrated in Chapter 3. However, COMAD does not provide annotations to identify stateless actors and cannot execute independent invocations concurrently as shown at (2) in Figure 8.12. Furthermore, COMAD imposes a fixed order on elements, requiring annotations on collections to be inserted before the opening tag and forcing additions to collections to happen at the end. Thus, the COMAD system queues up all elements of a newly created collection in order to allow the actor designer to insert annotations at any time as shown at (3) in Figure 8.12. This significantly slows down the workflow execution when a source actor creates a large new collection at the beginning of a workflow execution. Furthermore, this queuing increases the memory consumption significantly.

The required order in the collection structure of COMAD slows down execution. Furthermore, the implementation of annotations in COMAD significantly reduces streaming, increases memory consumption and slows down the workflow execution. In order to mitigate this performance issues, COMAD was modified to allow explicit streaming of data if the user guarantees that no annotations will be added to a collection after forcing the streaming behavior.

8.7 Improved Distributed Execution

Another focus of research on scientific workflow systems is the development of more efficient execution strategies that utilize distributed resources. With some contributions by the author, Daniel Zinn designed a method to parallelize the execution of a workflow definition that forms a pipeline and uses a XML-like structured data model by using Map-Reduce [ZBKL10].

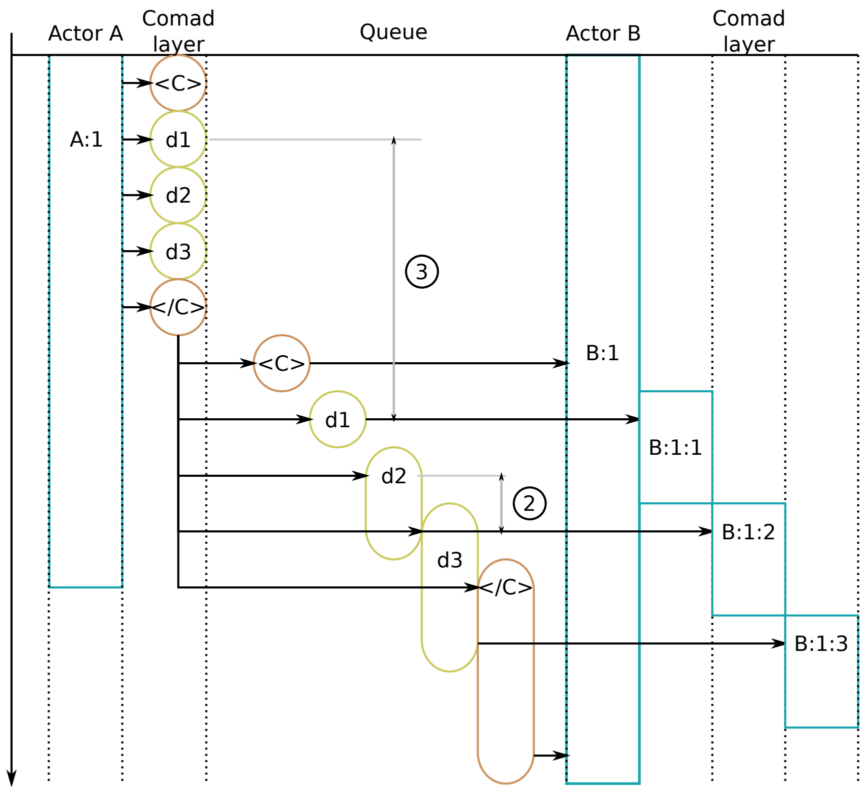


Figure 8.12: Suboptimal COMAD workflow execution. Time is progressing towards the bottom. (2) is the delay caused by serial execution of invocations of stateless actor `B` that can be executed in parallel. (3) is the delay caused by the COMAD implementation to allow adding annotations as long as the collection is open.

Figure 8.13: XML Pipeline with five steps. Each step has a defined *scope* (e.g.,//B, //C) to "work on". Sample input data is shown in the bottom left, data partitioning for Step 1 and 5 is shown in the bottom right of the figure.

8.7.1 Introduction

XML and XML-like tree structures are often used to organize and maintain collections of data, and so it is natural to devise data-driven processing pipelines (e.g., scientific workflows) that work over nested data (XML). A number of such approaches have recently been developed, e.g., within the web [AN05, Fag07, BPM05], scientific workflow [OGA$^+$06, LAB$^+$06, HKS$^+$07], and database communities [TRP$^+$04, WM07, ACGG$^+$02]. These approaches provide mechanisms to create complex pipelines from individual computation steps that access and transform XML data, where each step converts input data into new data products to be consumed by later steps. Additionally, pipelines often include steps that are applied only to portions of their input structures (leaving the remaining portions unchanged), in which case steps can be seen as performing specialized XML update operations [OGA$^+$06, MBZL08, ZBML09b]. The components implementing these computation steps often employ techniques from XML processing (e.g., XPath, XQuery, XSLT), and call built-in functions or external applications to perform "scientifically meaningful" computations (e.g., DNA sequence alignment, image processing, or similarly specialized algorithms).

Many of the above approaches employ *pipeline parallelism* to more efficiently execute pipelines by streaming XML data through components, thus allowing *different* steps within a pipeline to work concurrently over an XML stream. However, these approaches largely ignore *data parallelism*, which can significantly reduce pipeline execution time by allowing the *same* step to be executed in parallel over distinct subcollections of data.

This section presents three strategies that utilize MapReduce [DG08] to facilitate data-parallel computation over XML: *Naive*, *XMLFS*, and *Parallel*. For example, consider the simple XML processing pipeline shown in Figure 8.13. This pipeline consists of five steps, each of which (i) receives XML structures from previous steps, and (ii) works over specific XML fragments (subtrees) within these structures. These fragments are determined through XPath expressions that specify the "*scope*" of a step. Steps are invoked over each scope match (i.e., matching XPath fragment), and steps can perform arbitrary modifications to matched fragments using general XML processing techniques (e.g., XQuery, XSLT). The modifications made by steps often involve calling built-in (scientific) functions whose outputs are added within the matched fragment, or used to replace existing parts of the fragment. The result is a modified (i.e., updated) XML structure that is passed to subsequent steps. As an example, the first step of Figure 8.13 has the scope "//B", allowing it to perform arbitrary changes on "B"-rooted subtrees, i.e., new data items or subtrees can be inserted anywhere below the "B" node. However, for the middle step with scope "//D", changes may only be performed at the leaves of the given structure shown in the bottom-left of Figure 8.13. To exploit data parallelism, scope matches (fragments) are mapped to "work-pieces" that are then processed in parallel by MapReduce. The bottom-right of Figure 8.13 shows how the data is partitioned for the scope "//B" as used in Steps 1 and 5.

This naive strategy, however, can lead to bottlenecks in the splitting and regrouping phase of the parallel MapReduce execution. For example, from Step 1 to 2 the subtrees shown at the bottom right of Figure 8.13 must be partitioned further. Grouping all work-pieces together again to then re-split for the second task is clearly inefficient. Furthermore, from Step 3 to 4, the "D"-rooted trees must be re-grouped to form trees rooted at "C". Performing this grouping in a single global task also adds an unnecessary bottleneck because all required regroupings could be done in parallel for each "C"-rooted subtree.

The XMLFS strategy maps XML data into a distributed file system to eliminate the grouping bottleneck of the Naive strategy. The Parallel strategy further utilizes existing splits to re-split the data in parallel, thereby fully exploiting the grouping and sorting facilities of MapReduce. In general, each of these strategies offers distinct approaches for applying MapReduce to data-parallel processing of XML.

In the following, Section 8.7.2 first describes the MapReduce paradigm and an example that demonstrates the features utilized in the three strategies. Section 8.7.3 describes a framework for XML processing pipelines, introduces important notions for their parallel execution, and gives several pipeline examples. Then Section 8.7.4 presents the three parallelization strategies as well as their advantages and trade-offs in more detail. Section 8.7.5 presents a thorough experimental evaluation of all strategies. The experiments show a twenty-fold speedup (with 30 computing nodes) in comparison to a serial execution, even when the basic Naive strategy is used. It is shown that the Parallel approach significantly outperforms Naive and XMLFS for large data and when the cost for splitting and grouping becomes substantial. The experiments consider a wide range of factors —including the number of mapper tasks, the size of data and the XML nesting structure, and different computational load patterns—and shows how these factors influence overall processing time using the three strategies. Finally, Section 8.7.6 discusses related work and concludes this section.

8.7.2 Preliminaries: MapReduce

MapReduce [DG08] is a software framework for writing parallel programs. Unlike with PVM or MPI, where the programmer is given the choice of how different processes communicate with each other to achieve a common task, MapReduce provides a fixed *programming scheme.* A programmer employing the MapReduce framework implements *map* and *reduce* functions, and the MapReduce library carries out the execution of these functions over corresponding input data. While restricting the freedom of how processes communicate with each other, the MapReduce framework is able to automate many of the details that must be considered when writing parallel programs, e.g., check-pointing, execution monitoring, distributed deployment, and restarting individual tasks. Furthermore, MapReduce implementations usually supply their own distributed file systems that provide a scalable mechanism for storing large amounts of data.

Programming model. Writing an application using MapReduce mainly requires designing a *map* function and a *reduce* function together with the data types they operate on. Map and reduce implement the following signatures

$$\begin{aligned} \text{map} &:: \quad (K_1, V_1) \;\to [(K_2, V_2)] \\ \text{reduce} &:: \quad (K_2, [V_2]) \;\to [(K_3, V_3)] \end{aligned}$$

where all K_i and V_i are user-defined data types. The map function transforms a *key-value pair* (short kv-pair) into a list of kv-pairs (possibly) with different types. The overall input of a MapReduction is a (typically large) list of kv-pairs of type (K_1, V_1). Each of these pairs is supplied as a parameter to a map call. Here, the user-defined map function can generate a (possibly empty) list of new (K_2, V_2) pairs. All (K_2, V_2) pairs output by the mapper will be grouped according to their keys. Then, for each distinct key the user-defined reduce function is called over the values associated to the key. In each invocation of reduce the user can output a (possibly empty) list of kv-pairs of the user-defined type (K_3, V_3).

The MapReduce framework divides the overall input data into kv-pairs, and splits this potentially large list into smaller lists (so-called *input splits*). The details of generating kv-pairs (and input splits) can also be specified by the user via a custom *split* function. After kv-pairs are created and partitioned into input splits, the framework will use one separate map process for each input split. Map processes are typically spawned on different machines to leverage parallel resources. Similarly, multiple reduce processes can be configured to process in parallel different distinct keys output by the map processes.

Example. One task could be to generate a histogram and an inverted index of words for a large number of text files (e.g., the works of Shakespeare), where the inverted index is represented as a table with columns *word*, *count*, and *locations*. For each distinct word in the input data there should be exactly one row in the table containing the word, how often it appears in the data, and a list of locations that specify where the words were found. To solve this problem using MapReduce, type K_1 is designed to contain a filename as well as a line number (to specify a location), and the type V_1 to hold the corresponding line of text of the file. When given a (*location*, *text*) pair, map emits (*word*, *location*) pairs for each word inside the current line of text. The MapReduce framework will then group all output data by words, and call the reduce function over each word and corresponding list of locations to count the number of word occurrences. Reduce then emits the accumulated data (*count*, List of *locations*) for each processed word, i.e., the data structure V_3 will contain the required word count and the list of locations.

The MapReduce framework can additionally sort values prior to passing them to the reduce function. The implementation of secondary sorting depends on the particular MapReduce framework. For example, in Hadoop [Had] it is possible to define custom comparators for keys K_2 to determine the initial grouping as well as the order of values given to reduce calls. In our example above, we could design the key K_2 to not only contain the word but also the location. We would define the "grouping" comparator to only compare the word part of the key, while the "sorting" comparator would ensure that all locations passed will be sorted by filename and line number. The reduce function will then receive all values of type *location* in sorted order, allowing sorted lists of locations to be easily created.

In general, MapReduce provides a robust and scalable framework for executing parallel programs that can be expressed as combinations of map and reduce functions. To use MapReduce for parallel execution of XML processing pipelines, it is necessary to design data structures for keys and values as well as to implement the map and reduce functions. More complex computations can also make use of custom group and sort comparators as well as input splits.

8.7.3 Framework

The general idea behind transforming XML processing pipelines to MapReduce is to use map processes to parallelize the execution of each pipeline task according to the task's scope expression. For each scope match the necessary input data is provided to the map tasks, and after all map calls have executed, the results are further processed to form either the appropriate input structures for the next task in the pipeline or the overall output data. For example, consider again the pipeline from Figure 8.13. The partitioning and re-grouping of XML data throughout the pipeline execution is shown in Figure 8.14: data in the first row is split into pieces such that at most one complete "B" subtree is in every fragment, which is then processed in parallel with the other fragments. Then, further splits occur for scope "C" and "D" respectively. Data is later re-grouped to ensure that all elements corresponding to a scope match are available as a single fragment.

The following sections define the data model and assumptions concerning XML processing pipelines. Furthermore, operations are defined that may be performed on single fragments within map calls (i.e., by pipeline tasks) to guarantee safe parallel execution.

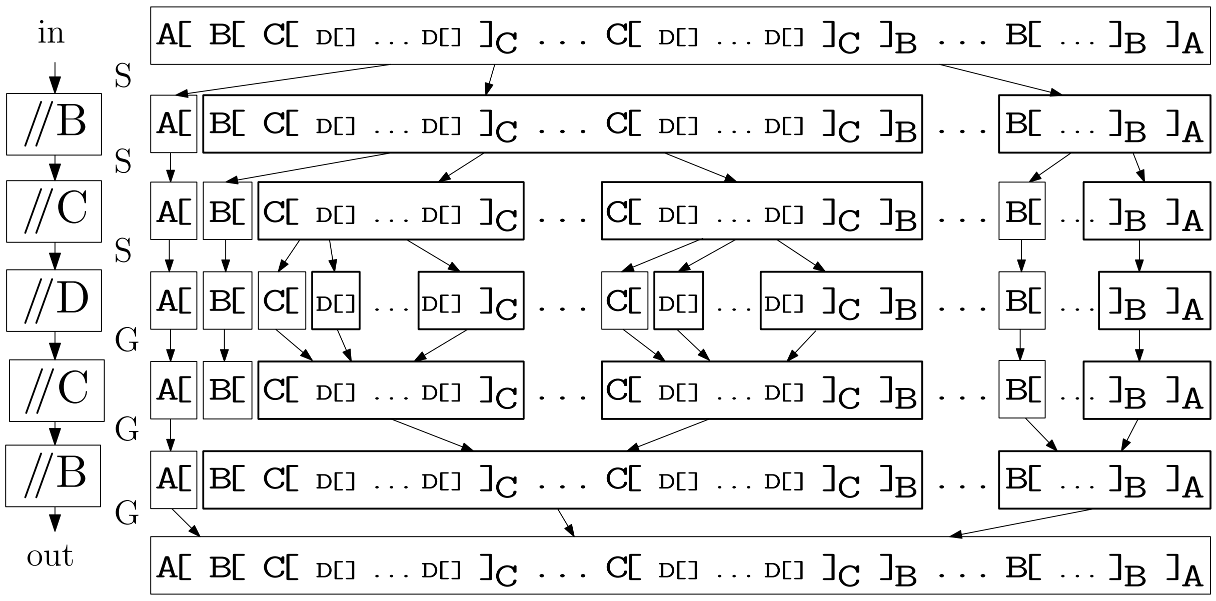

in
//B
//C
//D
//C
//B
out
S
S
S
G
G
G
A[B[C[D[] ... D[]]C ... C[D[] ... D[]]C]B ... B[...]B]A

Figure 8.14: Splits and groups for Parallel execution. For each step in the pipeline the data is partitioned such that all data for one scope match is inside one fragment while each fragment holds at most one match.

XML Processing Pipelines

The XML processing pipelines used here adopt the standard XML data model corresponding to labeled ordered trees represented as sequences of tokens; namely, opening tags "`T[`", closing tags "$]_T$", and data nodes "`#d`". Data nodes typically represent data products whose specific format is understood by the software components implementing pipeline tasks, but not by the XML framework itself, which treats data as opaque `PCData` nodes. For instance, data nodes may represent binary data objects (such as images) or simple text-based data (e.g., DNA sequence alignments).

Pipeline tasks typically call "scientific" functions that receive data nodes as input and produce data nodes as output. In addition, tasks are annotated with scopes that define where in the overall XML structure input data is taken from and output data is placed. Each scope specifies XML fragments within the input structure that represent the context of a task. Pipeline tasks may insert data (including XML tree data) anywhere within their corresponding context fragments or as siblings to the fragments, and remove data or complete subtrees anywhere within their fragments (including removing the fragments themselves). It is often the case that a given XML structure will contain multiple matching fragments for a task. In this case, the task is executed over each such match. We assume tasks do not maintain state between executions, thus allowing them to be safely executed in parallel over a given XML structure via the MapReduce framework.

More formally, a pipeline consists of a list of tasks $\mathcal{T}$ each of which *updates* an XML structure X to form a new structure X'. Further, $\mathcal{T} = (\sigma, \mathcal{A})$ where the scope σ is a (simple) qualifier-free XPath expression consisting of child (/) and descendent-or-self (//) axes, and $\mathcal{A}$ is a function over XML structures.

A subtree s_i in an input XML structure X is a scope match if $\sigma(X)$ selects the root node of s_i. For nested scope matches, only the highest-level match in X is considered—a common restriction (e.g., [BCF03]) for avoiding nested executions. Formally, σ selects n non-overlapping subtrees s_i from X:

$$\sigma(X) = \{s_1, \ldots, s_n\}.$$

Then, the function $\mathcal{A}$ is called on each of these subtrees to produce a new XML fragment, i.e.:

$$\text{for each } s_i : \quad s'_i = \mathcal{A}(s_i).$$

The output document X' is then formed by replacing all s_i subtrees in X by the respective outputs s'_i:

$$X' = X[s_1 \to s'_1, s_2 \to s'_2, \ldots].$$

$\mathcal{A}$ must be a function in the mathematical sense, i.e., a result s'_i only depends on its corresponding input s_i. This implies that s'_i can be computed from s_i independently from data inside other fragments s_j or completely non-matched data in X.[1]

Operations on Token Lists

During pipeline execution, XML data is represented as a sequence (i.e., list) of tokens of the form `T[`, $\texttt{]}_{\texttt{T}}$, and `#d`. By convention, capital letters are used to denote token lists and lowercase letters to denote trees and (ordered) forests. Token lists are partitioned into fragments that are

[1]In essence, a "`map` $\mathcal{A}$" on the list of scope matches is performed with `map` being the standard `map` function of functional programming languages. Thus $\mathcal{A}$ is required to be a function to parallelize $\mathcal{A}$ invocations.

sent to map calls for concurrent processing. Next, the changes the map calls may perform on fragments to avoid jeopardizing overall data integrity are characterized. Note that the proposed rules can be followed locally and thus eliminate the need for more involved locking mechanisms.

Definition (Balanced Token List). Given the following redexes to modify token lists:

$$A \ \texttt{\#d} \ B \ \Rightarrow \ A\,B \qquad A, B \in \text{Token List} \tag{8.1}$$

$$A \ \texttt{X[ ]}_{\texttt{X}} \ B \ \Rightarrow \ A\,B \qquad A, B \in \text{Token List} \tag{8.2}$$

rule (8.1) deletes any data node whereas (8.2) deletes matching Open and Close nodes if they are next to each other within a sequence and have matching labels. As usual, $T \Rightarrow^* T'$ is used if there exists a sequence of token lists T_i such that $T = T_1 \Rightarrow T_2 \Rightarrow \cdots \Rightarrow T_n = T'$. A token list T is balanced if it can be reduced to the empty list, i.e., $T \Rightarrow^* [\,]$.

Note that $\Rightarrow^*$ is normalizing, i.e., if $T \Rightarrow^* [\,]$ and $T \Rightarrow^* T'$ then $T' \Rightarrow^* [\,]$. This means that for a balanced list T, applying deletion rules (8.1) and (8.2) in any order will terminate in the empty list (by induction on list length). Also note that an XML forest naturally corresponds to a balanced token list and vice versa.

As described above, calls to map should compute new forests s_i' from existing trees s_i. In particular, s_i' can be computed by performing tree insertion and tree deletion operations in an arbitrary order on s_i. The following operations on token lists correspond to these allowed operations on trees.

Observation (Safe insertions). Inserting a balanced token list I at any position into a balanced token list T corresponds to inserting the forest i into the forest t (where forests i and t correspond to token lists I and T, respectively). In particular this operation results in a balanced token list T'. This insertion is called a *safe insertion.*

Note that insertions which simply maintain the "balance" of a sequence, but are not safe, can change ancestors of already existing nodes. Consider the case of inserting the unbalanced fragment "$\texttt{]}_{\texttt{A}}$ A[" into the middle of the balanced token list "A[#d #d $\texttt{]}_{\texttt{A}}$". This insertion will result in the balanced list "A[#d $\texttt{]}_{\texttt{A}}$ A[#d $\texttt{]}_{\texttt{A}}$". However, the second #d token has changed parent

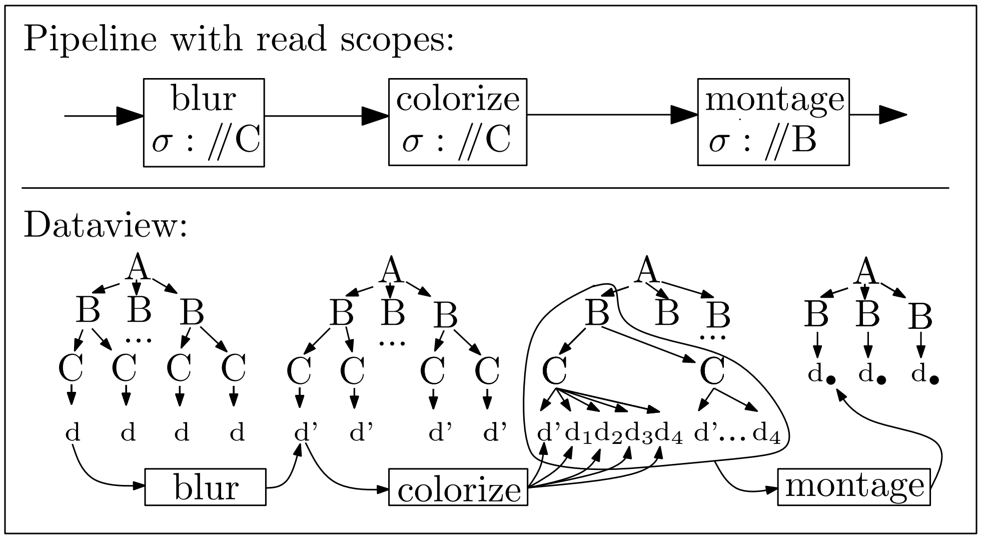


Figure 8.15: Image transformation pipeline: All images are *blurred*; then from each image, four new images are created by *coloring*; and finally a big *montage* is created from all images below each “B”.

nodes without explicitly being touched.

Observation (Safe deletions). Removing a consecutive and balanced token list D from a balanced token list T results in a balanced token list T'. This operation corresponds to deleting the forest d from t. This type of deletion is called a *safe deletion*.

Corollary 1 (Safe operations carry over to fragments of token lists). Viewing token-list fragments as parts of the total (balanced) token list, safe insertions and safe deletions can be performed to achieve the desired operations inside the scope of a pipeline task.

Corollary (1) ensures that map calls can change their fragments by performing safe insertions and deletions without interfering with the data of other map calls. Moreover, since the complete scope is inside the fragment received by the map call, each map call is able to delete its scope match, or to perform any “localized” operations on it using all the data inside its scope.

XML-Pipeline Examples

In addition to the simple pipeline introduced in Figure 8.13, a more complex example of a common image processing pipeline is shown in Figure 8.15. This pipeline is similar to a number of (more complicated) scientific workflows that perform image processing, e.g., in functional Magnetic

Resonance Imaging (fMRI) [ZDF+05] and plasma-fusion data analysis [PLK07]. The pipeline employs the *convert* and *montage* tools from the Image-Magick [Ima] suite to process multiple images organized according to nested XML collections. As shown in Figure 8.15, a top-level "A" collection contains several "B" collections, each of which contains several "C" collections with an image d inside. The first step of the pipeline blurs the images (via the "`convert -blur`" command). Since this operation is performed on each image separately, we define the task's scope σ using the XPath expression //C and its corresponding function $\mathcal{A}$ such that it replaces the image within its scope with a modified image resulting from invoking the blur operation. The next step in the pipeline creates a series of four colorized images from each blurred image d' using the command "`convert -modulate 100,100,`i" with 4 different values for i. The last step of the pipeline combines all images under one "B" collection into a single large image using the montage tool. The scope σ of this last task is //B since all images inside a "B"-labeled tree are input to the montage operation. Here the framework groups previously split fragments to provide the complete "B" subtree to the montage task.

8.7.4 Parallelization Strategies

We consider three strategies, *Naive*, *XMLFS* and *Parallel*, whose main differences are shown in Figure 8.16. These strategies use variations of key-value data structures as well as split, map, and reduce operations, and build upon each other to address various shortcomings that arise in large-scale and compute-intensive processing of nested data.

Naive Strategy

The Naive approach corresponds to a straightforward application of MapReduce over XML data. As shown in Figure 8.16, XML token sequences are cut into pieces for the map calls, the task's operation $\mathcal{A}$ are performed on its scope, and finally the result are merged in the reduce step of the MapReduction to form the final output[2]. The Naive approach uses the following data structures for keys and values.

Key: Integer

[2]This parallelization is a form of a simple scatter and gather pattern.

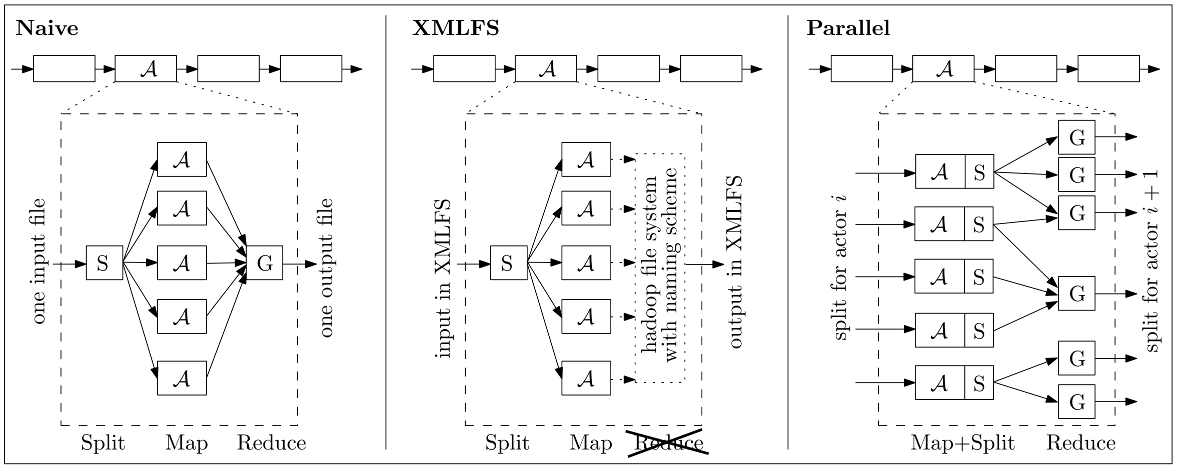


Figure 8.16: Processes and dataflow for the three parallelization strategies.

$$\begin{aligned} \text{Token} &:= \text{XOpen — XClose — Data} \\ \textit{Value}\text{: TList} &:= \text{TokenList} \end{aligned}$$

Given an XML pipeline with multiple tasks, a MapReduction is created with *split*, *map*, and *reduce* as given in Figure 8.17. From an XML structure, *SplitNaive* creates a kv-pair for each match of the task's scope: each pair comprises an *Integer* as key, and a *TList* as value.

To decide if a current token opens a new scope in line 4 of Figure 8.17, a straightforward technique is used to convert the qualifier-free, simple XPath-expression σ into a DFA reading strings of opening tokens. The DFA accepts when the read string conforms to σ. Using a stack of DFA states, we keep track of the current open tags. Here, the current state for an open token is pushed and the DFA is reset to the state popped from the stack when a closing token is read. To prevent nested scope matches, the DFA goes into a non-acceppting state with self-loop after encountering a match. Note that closing the match will "pop" the automaton back to the state before the match. This simple and efficient approach can be used for streaming token lists because of the simplicity of the XPath fragment in which scopes are expressed.[3]

The first pair constructed by *SplitNaive* contains all XML tokens before the first match, and each consecutive pair contains the matching data, possibly followed by non-matching data. Each

[3]In general, this fragment is sufficient for modeling many scientific applications and workflows.

```
SplitNaive: TList input, XPath $\sigma$ $\rightarrow$ [(Integer, TList)]
  int i := 0; TList splitOut := [ ]
  FOREACH token IN input DO
    IF (token opens new scope match with $\sigma$) AND
        (splitOut $\neq$ [ ]) THEN
      EMIT (i, splitOut) // one split for each scope match
      i++; splitOut := [ ]
    splitOut.append(token)
  EMIT (i, splitOut)

MapNaive: Integer s, TList val $\rightarrow$ [(Integer, TList)]
  val' := $\mathcal{A}$(val) // execute pipeline task
  EMIT (s, val')

ReduceNaive: Integer s, [TList] vals $\rightarrow$ [(Integer,TList)]
  TList output := [ ]
  WHILE vals.notEmpty() DO
    output.append(vals.getValue()) // collapse to single value
  EMIT (0, output)
```

Figure 8.17: Split, Map, Reduce for Naive strategy.

pair is then processed in *MapNaive*. Then, *ReduceNaive* merges all data fragments back into the final XML structure. Since the grouping comparator always returns "equal", one reduce task will receive all output from the mappers; also the fragments will be received in document order because the MapReduce framework will sort the values based on the key, which is increasing in document order. The output structure can now be used as input data for another MapReduce that executes the next step in the pipeline.

Shortcomings of the Naive Strategy. The major shortcoming of the Naive approach is that although data is processed in parallel by calls to map, both splitting and grouping token lists is performed by a single task. Split and reduce can thus easily become a bottleneck for the execution of the pipeline.

XMLFS Strategy

The XMLFS strategy removes the bottleneck in the reduce phase of the Naive approach by mapping XML structures to a distributed file system (see Figure 8.16). Many MapReduce implementations, including Hadoop and Google's MapReduce, provide a distributed file system that allows efficient and fault-tolerant storage of data in the usual hierarchical manner of directories and files, and this distributed file system is employed in the XMLFS approach as follows.

Mapping XML structures to a file system. An XML document naturally corresponds to a file-system-based representation by mapping XML nodes to directories and data nodes to files. The ordering of XML data is encoded by pre-pending the XML-labels with *identifiers* (IDs) to form directory and file names. The IDs will also naturally ensure that no two elements in the same directory will have the same name in the file system even though they have the same tag. Note that although XML attributes are not explicitly considered here, they could, e.g., be stored in a file with a designated name inside the directory of the associated XML element.

Using a file system based representation of XML data has many advantages: **(1)** XML structures can be browsed using standard file-system utilities. The Hadoop software package, e.g., provides a web-based file-system browser for its Hadoop file system (HDFS) [Bor07]. **(2)** Large amounts of XML data can easily be stored in a fault-tolerant manner. Both Hadoop-FS and the Google File System provide distributed, fault-tolerant storage. Specifically, they allow users to specify a replication factor to control how many copies of data are maintained. **(3)** The file system implementation provides a natural "access index" to the XML structure: In comparison to a naive token list representation, navigating into a subtree t can be performed using simple directory changes without having to read all tokens corresponding to subtrees before t. **(4)** Applications can access the "distributed" XML representation in parallel, assuming that changes to the tree and data are made at different locations. In particular, pipeline steps can write their output data s'_i in parallel.

XMLFS-Write. XMLFS adapts the Naive approach to remove its bottleneck in the reduce phase. Instead of merging the data into a large XML structure, each task writes its modified data s'_i directly into the distributed file system. Since there is no need to explicitly group token lists together to form bigger fragments, this operation is performed directly in the map calls. This approach removes the overhead of shuffling data between map and reduce calls as well as the overhead of invoking reduce steps. In particular, the XMLFS strategy does not use the grouping and sorting feature of the MapReduce framework since each task is implemented directly within the map function.

In XMLFS, the file system layer performs an *implicit* grouping as opposed to the *explicit* grouping in the Naive reduce function. When map calls write the processed XML token list T to the file system, the current path p from the XML root to the first element in T needs to be available since the data in T will be stored under the path p in the file system. This information is encoded as a *leading path* into the key. IDs for maintaining order among siblings must also be available. Since map calls may not communicate with each other, the decisions about the IDs must be purely based on the received keys and values, and the modifications performed by a task's operation $\mathcal{A}$. Unfortunately, the received token lists are not completely independent: An opening token in one fragment might be closed only in one of the following fragments. Data that is inside such a fragment must be stored under the same directory on the file system by each involved map call independently. It is therefore essential for data integrity that all map calls use the same IDs for encoding the path from the document root to the current XML data. To make these concepts more clear, requirements are now stated for IDs in general, and requirements for ID handling in split and map functions are defined.

Requirements for Token-Identifiers (IDs). The following requirements need to be fulfilled by IDs: *Compact String Representation:* A (relatively small) string representation of the ID is required to be included in the token's filename, since the ID has to be used for storing the XML data in the distributed file system. *Local Order:* IDs can be used to order and disambiguate siblings with possibly equal labels. Note that a total order is not required: IDs only need to be unique and ordered for nodes with the same parent. *Fast comparisons:* Comparing two IDs should be fast. *Stable insertions and deletions:* Updates to XML structures should not effect already existing IDs. In particular, it should be possible to insert arbitrary data between two existing tokens. It should also be possible to delete existing tokens without changing IDs of tokens that have not been deleted.

Choice of IDs. Many different labeling schemes for XML data have been proposed; see [HHMW07] for an overview. Here, any scheme that fulfills the requirements stated above could be used. This includes ORDPATHs described in [OOP$^+$04] or the DeweyID-based labels presented in [HHMW07]. However, note that many proposed ID solutions (including the two schemes just mentioned) provide global IDs, facilitate navigation (e.g., along parent-child axes), and allow testing of certain relationships between nodes (e.g., whether a node is a descendent of another node). All strategies presented here only require local IDs, i.e., IDs that are unique only among siblings, and do not use IDs for navigation or testing, so a simpler (though less powerful) labeling scheme is used. Of course, this IDs could easily be replaced by ORDPATHs, or other approaches, if needed.

Simple decimal IDs. A natural choice for IDs are objects that form a totally ordered and dense space such as the rational numbers. Here, a new number m can found between any two existing numbers a and b, and thus neither a or b need to be changed to insert a new number between them. Using only these numbers that have a finite decimal representation (such as 0.1203 as opposed to 0.3 periodical 3) we would also gain a feasible string representation. However, there is no reason to keep the base 10. Instead max-long is used as a base for the IDs. Concretely, an ID is a list of longs. The order relation is the standard lexicographical order over these lists. In the string representation a "." is added between the single "digits". Since one digit already has a large number of values, long lists can easily be avoided: To achieve short lists, a heuristic similar to the one proposed in [HHMW07] is used. When the initial IDs for a token stream are created, instead of numbering tokens successively, a gap is introduced between numbers (e.g., an increment of 1000). Note that since nodes are only labeled locally, Maxlong/1000[4] sibling nodes are supported with a one-"digit" ID during the initial labeling pass. With a gap of 1000, a large number of new tokens can be inserted into existing token lists before a second "digit" needs to be added.

Splitting input data. Creating key-value pairs for the XMLFS strategy is similar to the Naive strategy with the exception that IDs of XOpen and Data tokens are created and maintained. The XMLFS strategy uses the following data structures for keys and values.

ID	:=	**List of** Long
IDXOpen	:=	**Record{** id: ID, t: XOpen**}**
IDData	:=	**Record{** id: ID, t: Data**}**
IDToken	:=	IDXOpen — IDData — XClose

[4]approximately 9×10^{15} on 32-bit systems

```
SplitXMLFS: TIDList input, XPath $\sigma$ → [(PKey,TIDList)]
  CALL Split(input, $\sigma$, [0], [maxlong], [ ])

MapXMLFS: PKey key, TIDList val → [(PKey, TIDList)]
  IF (key.lp / val[0]) matches scope $\sigma$
    val' := $\mathcal{A}$(val)
  Store val' in distributed file system

// No Reduce necessary, Map stores data
```

Figure 8.18: Split and Map for XMLFS

Key:	XKey	:=	**Record{** start: ID, end: ID, lp: TIDList**}**
Value:	TIDList	:=	**List of** IDToken

In the key, *lp* is used to include the leading path from the XML root node to the first item in the TIDList stored in the value. As explained above, this information allows data to be written back to the file system based solely on the information encoded in a single key-value pair. Finally, the IDs *start* and *end* are added to the key, which denote fragment delimiters that are necessary for independently inserting data at the beginning or end of a fragment by map calls. For example, assume data D should be inserted before the very first token A in a fragment[5] f. For a newly inserted D, an ID that is smaller than the ID of A need to be chosen. However, the ID must be larger than the ID of the last token in the fragment that comes before f. Since the IDs form a dense space, it is not possible to know how close the new ID $D.id$ should be to the already existing ID of A. Instead, the *start* ID in the key is used and it has the property that the last ID in the previous fragment is smaller. Thus, the newly inserted data item can be given an ID that is in the middle of *start* and $A.id$. Similarly, we store a mid-point ID *end* for insertion at the end of a TIDList.[6]

Figure 8.18 and Figure 8.19 gives the algorithm for splitting input data into key-value pairs. A stack *openTags* of currently open collections is maintained to keep track of the IDs in the

[5]The task might want to insert a modified version of its scope *before* the scope.

[6]When using ORDPATH IDs, we could exploit the so-called *careting* to generate an ID *very* close to another one. However, this technique would increase the number of digits for each such insertion, which is generally not desired.

```
1  Split: TIDList input, XPath $\sigma$, ID startID, ID endID, TIDList lp
           $\rightarrow$ [(PKey, TIDList )]
3    TIDList openTags := lp // list of currently open tags
     TIDList oldOpenTags := lp // leading path
5    ID lastEnd := startID // ending ID of last fragment
     ID lastTokenID := startID // ID of last token
7    TIDList splitOut := [ ] // accu for fragment value
     FOREACH token IN input DO
9      IF (openTags / token matches scope $\sigma$) AND
           (splitOut $\neq$ [ ]) THEN
11       ID newend := midPoint(lastTokenID, token.id)
         key := NEW PKey(lastEnd, newend, oldOpenTags)
13       oldOpenTags := openTags
         EMIT key, splitOut // output current fragment
15       lastEnd := newend; splitOut := [ ]
       splitOut.append(token);
17     IF token is IDData THEN
         lastTokenID := token.id
19     IF token is IDXOpen THEN
         openTags.append(token)
21       lastTokenID := [0]
       IF token is Close THEN
23       lastOpenToken := openTags.removeLast()
         lastTokenID := lastOpenToken.id
25   ENDFOR
     key := new PKey(lastEnd, endID, oldOpenTags)
27   EMIT key, splitOut // don't forget the last piece
```

Figure 8.19: Split for XMLFS & Parallel

various levels of an XML structure while iterating over the token list. Whenever the stream is split in fragments (line 11) a mid-point is computed between the previous Token-ID and the current one. The mid-point is then used as an *end* ID for the old fragment, and will later be the *start* ID for the fragment that follows. Note that lastTokenID is reset to "[0]" whenever a new collection is opened since the IDs are only local. Moreover, if a split occurs immediately after a newly opened collection, the mid-point ID would be [500] (the middle of [0] and the first token's ID [1000]). It is thus possible to insert a token both at the beginning of a fragment and at the end of the previous fragment.

Map step for XMLFS. As in the Naive strategy, the map function in the XMLFS approach performs a task's operation $\mathcal{A}$ on its scope matches. Similarly, safe insertions and deletions are required to ensure data integrity in XMLFS. Whenever new data is inserted, a new ID is created that is between the IDs of neighboring sibling tokens. If tokens are inserted as first child into a collection, the assigned ID is between [0] and the ID of the next token. Similarly, if data is inserted as the last child of a node (i.e., the last element of a collection), then the assigned ID is larger than the previous token. Note that when performing safe insertions and deletions only, the opening tokens that are closed in a following fragment cannot be changed. This guarantees that the leading path, which is stored in the key of the next fragment, will still be valid after the updates on the values. Also, XClose tokens that close collections opened in a previous fragment cannot be altered with safe insertions and safe deletions, which ensures that following the leading paths of fragments will maintain their integrity.

After data is processed by a map call, the token list is written to the file system. For this write operation, the *leading path* in the key is used to determine the starting positions for writing tokens. Each data token is written into the current directory using its ID to form the corresponding file name. For each XOpen token, a new directory is created (using the token's ID and label as a directory name) and is set as the current working directory. When an XClose token is encountered, the current directory is changed to the parent directory.

Shortcomings of the XMLFS Strategy. Although the XMLFS approach addresses the bottleneck of having a single reduce step for grouping all data, splitting is still done in a single, serial task, which can become a bottleneck for pipeline execution. Further, even the distributed file system can become a bottleneck when all map calls write their data in parallel. Often only a few (or even only one) master-server administers the distributed file system's directory structure and meta data. As the logical grouping of the XML structure is performed "on the file system", these servers might not be able to keep up with the parallel access. Since both the Google file system and HDFS are optimized for handling a moderate number of large files instead of a large number of (small) files or directories, storing all data between tasks to the file system using the directory-to-XML mapping above can become inefficient for XML structures that have many nodes and small data tokens.

Additionally, after data has been stored in the file system, it will be split again for further parallel processing by the next pipeline task. Thus, the file system representation must be transformed back into TIDLists. This work is unnecessary since the previous task used a TIDList representation, which was already split for parallel processing. For example, consider two consecutive tasks that both have the same scope: Instead of storing the TIDLists back into the file system, the first task's map function could directly pass the data to the map function of the second task. However, once consecutive tasks have different scopes, or substantially modify their data to introduce new scope matches, simply passing data from one task's map function to the next is not sufficient. This problem is addressed in the Parallel strategy defined next.

Parallel Strategy

The main goal of the Parallel Strategy is to perform both splitting and grouping in parallel, providing a fully scalable solution. For this, the existing partitioning of data is reused from one task to the next while still having the data corresponding to one scope inside a key-value pair. Imagine two consecutive tasks A and B. In case both tasks have the same scope, the data can be passed from one mapper to the next if A does not introduce additional scope matches for B, in which case fragments are split further. In case the scope of task B is a refinement of A's scope, i.e., A's σ_1 is a prefix of B's σ_2, A's mapper can split its TIDList further and output multiple key-value-pairs that correspond to B's invocations. However, it is also possible that a following task B has a scope that requires earlier splits to be undone. For example if task A's scope is //A//B whereas task B's scope is only //A, then the fine-grained split data for A needs to be *partially* merged before it is presented to B's mappers. Another example is an unrelated regrouping: here, splitting and grouping are necessary to re-partition the data for the next scope. Even in this situation, the operation in parallel should be performed efficiently. MapReduce's ability of grouping and sorting is used to achieve this goal. In contrast to the Naive Approach, not all data is grouped into one single TIDList. Instead, data is grouped into lists as they are needed by the next task. As shown later, this can be done in parallel. Next, the necessary analysis of the scopes as well as detailed algorithms for splitting, mapping, and reducing are presented.

Regrouping example. Consider an arbitrarily partitioned TIDList. Figure 8.20 shows an example in the second row. Each rectangle corresponds to one key-value pair: The value (a

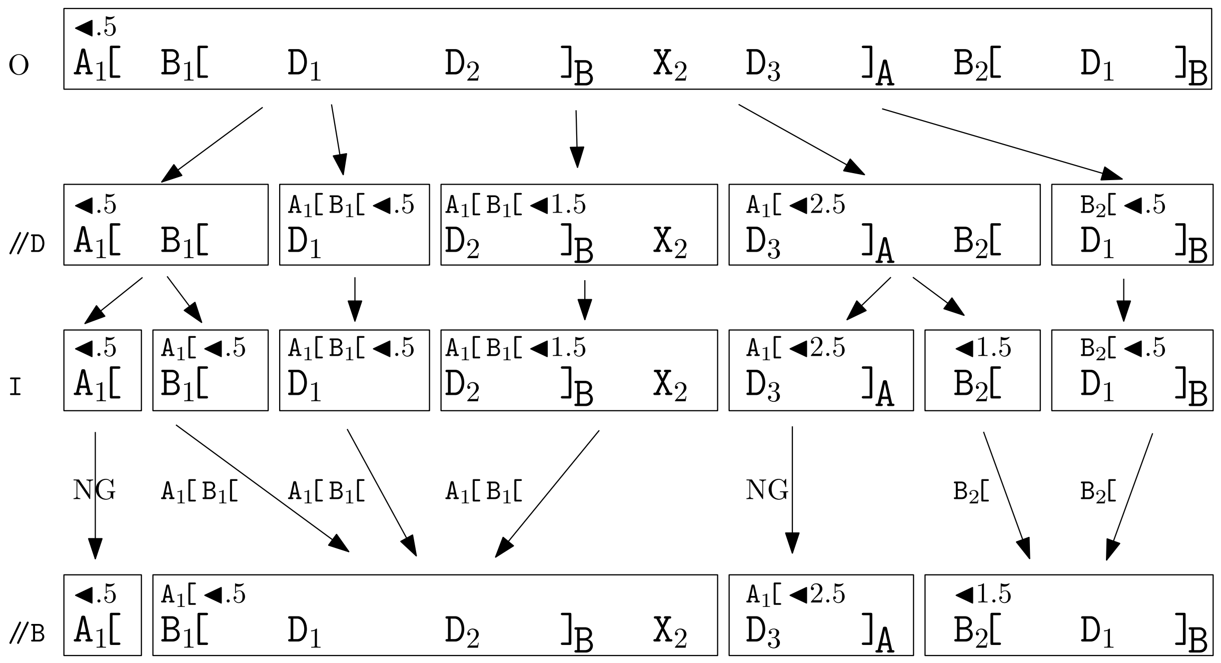


Figure 8.20: Example of how to change fragmentation from //D to //B in parallel. Since splitting from row two to row three is performed independently in each fragment this step can be performed in the Mapper. Grouping from row three to row four is performed in parallel by the shuffling and sorting phase of MapReduce such that the merge can be done in the Reducers, also in parallel.

TIDList) is written at the bottom of the box, whereas the key is symbolized at the top-left of the box. IDXOpen and IDXData tokens are depicted with their corresponding IDs as a subscript; XClose tokens do not have an ID. For ease of presentation decimal numbers are used to represent IDs with the initial tokens having consecutively numbered IDs. The smaller text line in the top of the boxes show the leading path *lp* together with the ID *start*. The key's ID *end* is not shown—it always equals the start-ID of the next fragment, and is a very high number for the last fragment. The first box in the second row, for example, depicts a key-value pair with the value consisting of two XOpen tokens, each of which having the ID of 1. The leading path in the key is empty, and the start-ID of this fragment is 0.5. Similarly, the second box represents a fragment that has as value only a token `D[` with ID 1. Its leading path is $\texttt{A}_1\texttt{[}$ $\texttt{B}_1\texttt{[}$, and the start-ID of this fragment is 0.5.

Now, consider that the split as shown in the second row of Figure 8.20 is the result after the task's action $\mathcal{A}$ is performed in the Mappers. Assume the next task has a scope of //B. In order to re-fragment an arbitrary split into another split, two steps are performed: A split and a merge operation.

Split-Operation. Inside the mapper, each fragment (or key-value pair) is investigated whether additional splittings are necessary to comply with the required final fragmentation. Since each fragment has the leading path, a start and an end-ID encoded in the key. Algorithm *Split* as given in Figure 8.19 is used to further split fragments. In Figure 8.20, for instance, each fragment in the second row is investigated if it needs further splits: The first and the fourth fragment will be split since they each contain a token `B[`. If there were one fragment with many B subtrees, then it would be split in many different key-value pairs, just like in the previous approach. Note that this split operation is performed on each fragment independently from others. Therefore *Split* operations are excecuted in parallel inside the Mappers as shown in the dataflow graph in Figure 8.16 and the pseudo-code for the Mapper task in Figure 8.21, line 6.

Merge-Operation. The fragments that are output by the split-Operation contain enough split-points such that at most one scope match is in each fragment. However, it is possible that the data within one scope is spread over multiple, neighboring fragments. In Figure 8.20, for example, the first `B`-subtree is spread over three fragments (fragment 2, 3, and 4). MapReduce's

```
MapParallel: SKey key, TIDList val → [(SKey, TIDList)]
  IF (key.lp / val[0]) matches scope $\sigma$
    val' := $\mathcal{A}$(val)
  List of (SKey, TIDList) outlist;
  // split according to the scope $\sigma'$ of the following step
  outlist := Split(val', $\sigma'$, key.start, key.end, key.lp)
  FOREACH (key,fragment) $\in$ outlist DO
    EMIT(key, fragment);

ReduceParallel: SKey key, [TIDList] vs → [(SKey, TIDList)]
  TIDList out := [ ]
  WHILE (val := vs.next())
    out.append(val);
    key.end := val.end // set end in key to end of last fragment
  EMIT(key, out)
```

Figure 8.21: Map and Reduce for Parallel

ability to group key-value pairs is used to merge the correct fragments in a Reduce step. For this, additional *GroupBy* information is provided in the key. In particular, the key and value data structures for the parallel Strategy are as follows:

	GroupBy	:=	**Record{** group: Bool, gpath: TIDList **}**
Key:	PKey	:=	**Record of** XKey **and** GroupBy
Value:	TIDList	:=	**List of** IDToken

Fragments, that do not contain tokens that are within scope simply set the *group*-flag to *false* and will thus not be grouped with other fragments by the MapReduce framework. In contrast, fragments that contain relevant matching tokens will have the group flag set. For these, *gpath* stores the path to the node matching the scope. Since there is at most one scope-match within one fragment (ensured by the previous split-operation) there will be exactly one of these paths. In Figure 8.20, this part of the key is depicted in the row between the intermediary fragments `I` and the final fragments split according to //B: The first fragment, not containing any token below the scope //B, is not going to be grouped with any other fragment. The following three fragments all contain $\texttt{A}_1\texttt{[}$ $\texttt{B}_1\texttt{[}$ as *gpath*, and will thus be presented to a single Reducer task, which will in turn assemble the fragments back together (pseudo-code is given in Figure 8.21.

```
GroupCompare: SKey keyA, SKey keyB → { <, =, > }
  IF (keyA.group AND keyB.group) THEN
    // group based on grouping−path
    RETURN LexicCompare(keyA.gpath, keyB.gpath)
  ELSE
    // don't group (returns < or > for two different fragments)
    RETURN SortCompare(keyA, keyB)

SortCompare: SKey keyA, SKey keyB → { <, =, > }
  // always lexicographically compare "leading path ⊕ start"
  RETURN LexicCompare( keyA.lp ⊕ keyA.start,
                       keyB.lp ⊕ keyB.start )
```

Figure 8.22: Group and sort for Parallel strategy

	Naive	**XMLFS**	**Parallel**
Data	XML File	File system representation	Key-value pairs
Split	Centralized	Centralized	Parallel
Group	Centralized by one reducer	Via file system + naming No shuffle, no reduce	Parallel by reducers
Key-Structure	One integer	Leading path with Ids	Leading path with Ids and grouping information
Value-Structure	SAX-elements	SAX-elements with XMLIds	SAX-elements with XMLIds

Figure 8.23: Main differences for compilation strategies

The output will be a single key-value pair containing all tokens below the node B as required.

Order of fragments. The IDs inside the TokenList of the leading path *lp* together with the ID *start* in a fragment's key can be used to order all fragments in document order. Since IDs are unique and increasing within one level of the XML data, the list of IDs on the path leading from the root node to any token in the document forms a global numbering scheme for each token whose lexicographical order corresponds to standard document order. Further, since each fragment contains the leading path to its first token and the ID *start*, a local ID, smaller than the ID of the first token, the leading path's ID-list extended by *start* can be used to globally order the fragments. See, for example Figure 8.20: In the third row (labeled with `I`) the ID lists $0.5 < 1, 0.5 < 1, 1, 0.5 < 1, 2.5 < 1.5 < 2, 0.5$ are ordering the fragments from left to right. This ordering is used for sorting the fragments such that they are presented in the correct order to the reduce functions. Figure 8.22 shows the definitions for the grouping and sorting comparator used in the Parallel strategy. Two keys that both have the *group* flag set, are compared based on the lexicographical order of their *gpath* entries. Keys that have *group* not set are simply compared. This ensures that one of them is strictly before the other that the returned order is consistent. The sorting comparator simply compares the IDs of the leading paths extended by *start* lexicographically.

Summary of Strategies

Figure 8.23 presents the main differences of the presented strategies, Naive, XMLFS, and Parallel. Note, that while Naive has the simplest data structures it splits and groups the data in a centralized manner. XMLFS parallelizes grouping via the file system but still has a centralized split phase. The Parallel strategy is fully parallel for both splitting and grouping at the expense of more complex data structures and multiple reduce tasks.

8.7.5 Experimental Evaluation

This section shows which speedups can be achived over a serial execution. Furthermore, the strategies are evaluated for scalability with an increasing data load. And finally, significant differences between the strategies are presented.

Execution Environment. All experiments were performed on a Linux cluster with 40 3GHz Dual-Core AMD Opteron nodes with 4GB of RAM and connected via a 100MBit/s LAN. Hadoop [Had] were installed on the local disks[7], which also serve as storage space for HDFS. Having approximately 60G of locally free disk storage provides 2.4TB of raw storage inside the Hadoop file system (HDFS). In all experiments, an HDFS-replication factor of 3 is used to tolerate node failures. The cluster runs the ROCKS [roc] software and is managed by SunGrid-Engine (SGE) [Gen01]; a special common SGE parallel environment is used that reserves a fixed number of nodes used as nodes in the Hadoop environment while performing tests. 30 nodes are running as

[7]Running hadoop from a shared NFS-home directory results in extremely large start-up times for Mappers and Reducers.

"slaves", i.e., they run the MapReduce tasks as well as the HDFS name nodes for the Hadoop file system. An additional node, plus a backup-node, are used for running the master processes for HDFS and the MR task-tracker, to which jobs are submitted. Hadoop was used in version 0.18.1 as available on the web-page. Hadoop was configured to launch Mapper and Reducer tasks with 1024MB of heap-space (`-Xmx1024`) and the framework was restricted to use 2 Map and 2 Reduce tasks per slave node. The measurements are done using the UNIX `time` command to measure wall-clock times for the main Java program that submits the job to Hadoop and waits until it is finished. While the experiments were running, no other jobs were submitted to the cluster to prevent any interference with the runtime measurements.

Handling of Data Tokens. A first implementation of the strategies were reading the XML data including the images into the Java JVM. Not surprisingly, the JVM ran out of memory in the split function of the Naive implementation as it tried to hold all data in memory. This happened for as few as `#B` = 50 and `#C` = 10. As each picture was around 2.3MB in size, the raw data alone already exceeds the 1024MB of heap space in the JVM. Although all our algorithms could be implemented in a streaming fashion (required memory is of the order of the depth of the XML tree; output is successively returned as indicated by the `EMIT` keyword), references in form of file-names are placed into the XML data structure, while keeping the large binary data at a common storage location (inside hdfs). Whenever an image reference is placed into the XML data, a free filename is optained from HDFS and the image is stored there. When an image is removed from the XML structure it is also removed from HDFS. The strategy of storing the image data not physically inside the data tokens also has the advantage that only the data that is actually requested by a pipeline step is lazily shipped to it. Another consequence is that the data that is actually shipped from the Mapper to the Reducer tasks is small and thus making even the naive strategy a viable option.

Number of Mappers and Reducers. As described in Section 8.7.2, a *split* method is used to group the input key-valuable pairs into so-called *input splits*. Then, for each input split one Mapper is created, which processes all key-value pairs of this split. Execution times of MapReductions are influenced by the number of Mapper and Reducer tasks. While many Mappers are beneficial to load balancing they certainly increase the overhead of the parallel computation especially if the number of Mappers significantly outnumbers the available slots on the cluster. A good choice is to use one Mapper for each key-value pair if the work per pair is significantly higher than task creation time. In contrast, if the work $\mathcal{A}$ is fast per scope match then the number of slots, or a small multiple of them is a good choice.

All output key-value pairs of the Mapper are distributed to the available Reducers according to a hashing function on the key. Of course, keys that are to be reduced by the same reducer (as in naive) should be mapped to the same hash value. Only the parallel approach has more than one Reducer. Since the work for each group is rather small, 60 Reducers are used in the experiments. The hash-function used is based on the *GroupBy*-part of the *PKey*. In particular for all fragments that have the *group* flag set, a hash value h is computed based on the IDs inside *gpath*: Let l be the flattened list of all the digits (longs) inside the IDs of *gpath*. Divide each element in l by 25 and then interpreted l as a number N to the base 100. While doing so, compute $h = (N \texttt{ mod } 263) \texttt{ mod}$ the number of available reduce tasks. For fragments with the group flag not set, a random number is returned to distribute these fragments uniformly over reducers[8]. The used hash-function resulted in an almost even distribution of all k-v-pairs over the available Reducers.

Comparison with Serial Execution

The experiments were performed on the image transformation pipeline (Figure 8.15), which represents pipelines that perform intensive computations by invoking external applications over `PCData` organized in a hierarchical manner. The number `#C` of "C" collections inside each "B" was varied, i.e., the total number of with "C" labeled collections in a particular input data is `#B`·`#C`. Execution times scaled linear for increasing `#B` (from 1 to 200) for all three strategies. The pipeline was also executed in serial on one host of the cluster. Figure 8.24 shows the execution times for `#B` = 200 and `#C` ranging over 1, 5 and 10. All three strategies significantly outperform the serial execution. With `#C` = 10, the speedup is more than twenty-fold. Thus, although the parallel execution with MapReduce has overhead in storing images in hdfs and copying the data

[8]Hadoop does not support special handling for keys that will not be grouped with any other key. Instead of shuffling the fragment to a random Reducer, the framework could just reduce the pair at the closest Reducer available.

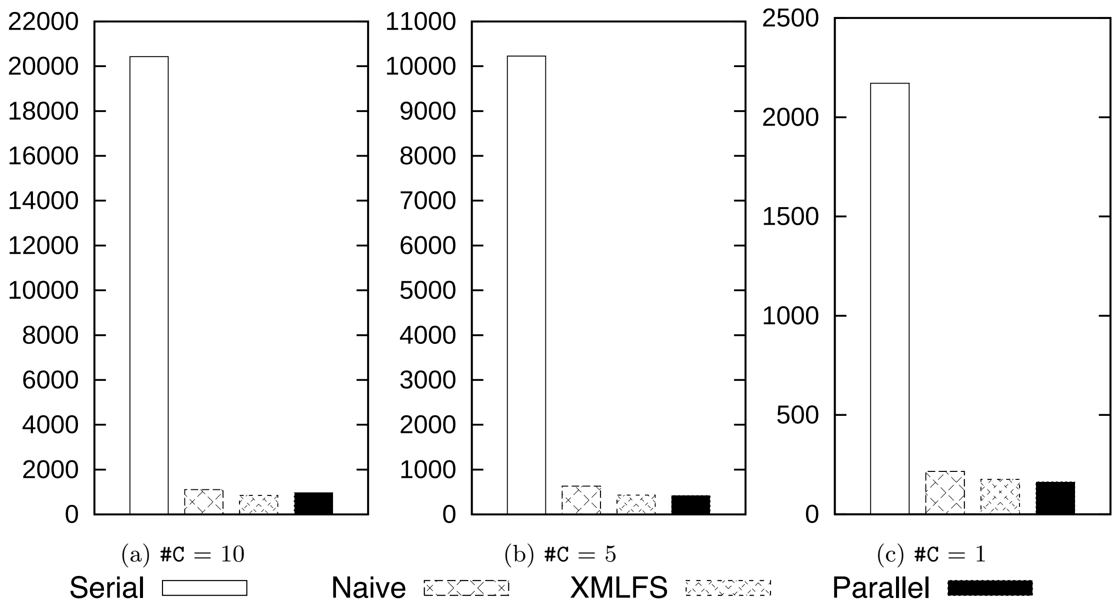


Figure 8.24: Serial versus MapReduce-based execution. Comparing wall-clock runtimes times for image processing pipeline (Figure 8.15) in seconds. All three strategies outperform a serial execution. Relative speedups range from around 20 for `#C` = 10 to above 10 for `#C` = 1. `#B` was set to 200.

from host to host during execution, speedups are substantial if the individual steps are relatively compute intensive in comparison to the data size that is being shipped. In this example, each image is about 2.3MB in size; and blur executed on the input image in around 1.8 seconds, coloring the image once takes around 1 second, the runtime of montage varies from around 1 second for one image to 13 seconds for combining 50 images[9].

In another experiment, the number of Mappers was changed. When creating one Mappers for each fragment, the fastest and most consistent runtimes could be achieved (shown in the graphs). When fixing the number of Mappers to 60, runtimes started to have high fluctuations due to so-called "stragglers", i.e., single Mappers that run slow and cause all other to wait for the stragglers' termination.

For this pipeline, all approaches showed almost the same run-time behavior. The reason is that the XML structure that is used to organize the data is rather small. Therefore, not much overhead is caused by splitting and grouping the XML structure, especially compared to the workload that is performed by each processing step.

[9]There are 5 differently colored images under each "C", with `#C` = 10, thus 50 images have to be "montaged".

Comparison of Strategies

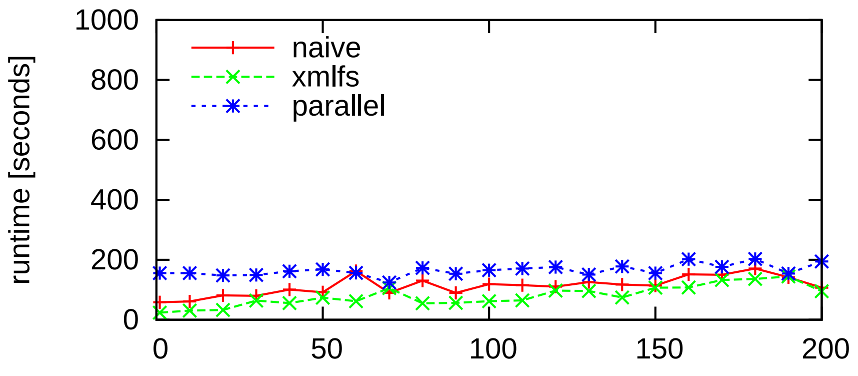
To analyze the overhead introduced by splitting and grouping, the pipeline given in the introduction (Figure 8.13) is used. Since it does not invoke any expensive computations in each step, the run times directly correspond to the overhead introduced by MapReduce in general and the implemented strategies in particular. In the input data, 100 empty "D" collections are used as leaves, then `#B` and `#C` are varied as in the previous example.

The results are shown in Figure 8.25. For small data sizes (`#C` = 1 and small `#B`) Naive and XMLFS are both faster than Parallel, and XMLFS outperforms Naive. This confirms the expectations: Naive uses fewer Reducers than the Parallel approach (1 vs. 60) even though the 60 reducers are executed in Parallel, there is some overhead involved to launch the tasks and wait for their termination. Furthermore, the XMLFS approach has no reducers at all and is thus as Mapper-only pipeline very fast. The pipeline ran with `#C` = 1 until `#B` = 1000 to investigate behavior with more data. From approximately `#B` = 300 to around 700, all three approaches had similar execution times. Starting from `#B` = 800, Naive and XMLFS perform worse than Parallel (380s and 350s versus 230s, respectively).

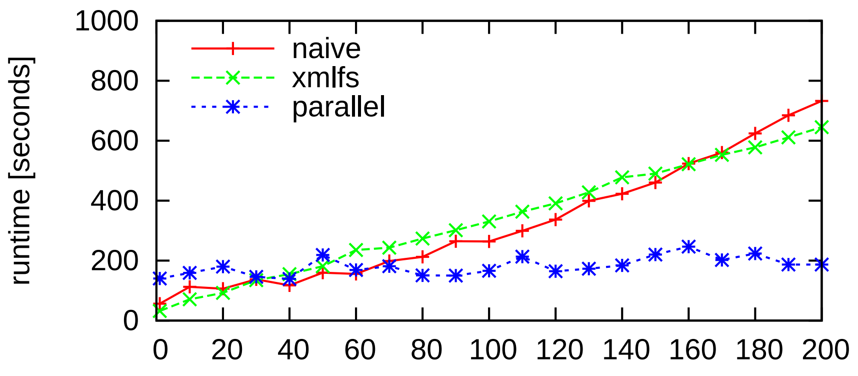
Runtimes for `#C` = 10 are shown in Figure 8.25(b), Here, Parallel outperforms Naive and XMLFS at around `#B` = 60 (with a total number of 60,000 "D" collections). This is very close to the number of 80,000 "D" collections at the "break-even" point for `#C` = 1. In Figure 8.25(c) this trend continues. The fine-grained measurements for `#B` = 1 to 10 show that the "break-even" point is, again, around 70,000 "D" collections.

In this experiment, the number of Mappers was set to 60 for all steps as the work for each fragment is small in comparison to task startup times. As above, 60 Reducers were used for the Parallel strategy.

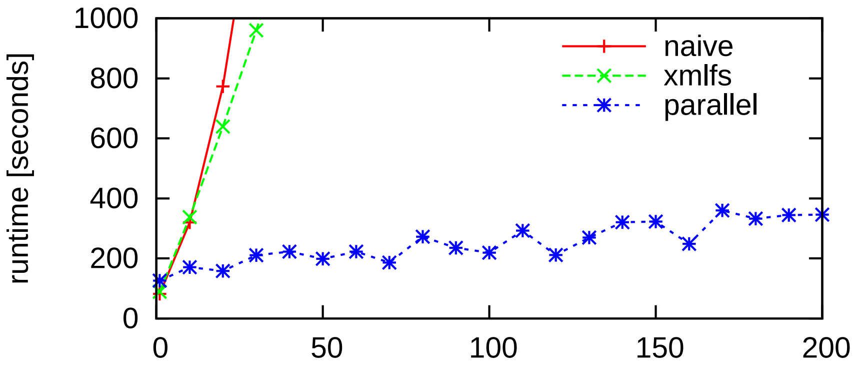
Experimentation result. The experiments confirmed that the three strategies can improve execution time for (relatively) compute-intense pipelines. The image-processing pipeline executed with a speedup of 20x. For XML data that is moderately sized, all three strategies work well, often with XMLFS outperforming the other two. However, if data size increases Parallel clearly outperforms the other two strategies due to its fully parallel split and group.



(a) `#C` = 1



(b) `#C` = 10



(c) `#C` = 100

Figure 8.25: Runtime comparison of the three strategies executing the pipeline given in Figure 8.13. On the X-Axis `#B` is varied, Y-axis shows wall-clock runtime of the pipeline. For small XML structures, Naive and XMLFS outperform Parallel since fewer tasks have to be executed. The larger the data the more superior is Parallel.

Inherited Benefits from using Hadoop

Using the Hadoop MapReduce implementation, not only provides speedups in execution time, but also other features such as: Monitoring of jobs via a web-interface, having failed jobs being re-run automatically by Hadoop, and a fault-tolerant, distributed storage (HDFS) for the data.

8.7.6 Related Work and Conclusion

The main contribution of this section is an approach for exploiting data parallelism in XML-based processing pipelines. In particular, new strategies are created and analyzed which exploit data parallelism in processing pipelines via a compilation to the MapReduce framework [DG08]. This techniques extend existing approaches within the area of scientific workflows and data management (e.g., ETL and XML processing pipelines). For example, the parallel execution and data management capabilities of MapReduce are exploited; and similarly, the workflow model is inspired by existing work in scientific workflow modeling [LAB+06, OGA+06, MB05], dataflow programming [Mor94, LP95], and general XML data processing[BCF03, KSSS04b, ABC+03].

Yang *et al.* present an extension of MapReduce to process relational operations, such as joins, on relational data. The strategies presented here focus on updates to tree-structured data and do not require modifications to the MapReduce framework itself.

Significant work has been done in the area of query processing over XML streams, e.g., see [KSSS04b, CCD+03, CDTW00, BBMS05, KSSS04a, GGM+04, CDZ06] among others. Most of these approaches consider optimizations for specific XML query languages or language fragments, sometimes taking into account additional aspects of streaming data. FluXQuery [KSSS04b] focuses on minimizing the memory consumption of XML stream processors. The approach presented above, however, is focused on optimizing the execution of compute and data intensive "scientific" functions and developing strategies for parallel and distributed execution of corresponding pipelines of such components.

DXQ [FJM+07] is an extension of XQuery to support distributed applications, and similarly, in Distributed XQuery [RBHS04], remote-execution constructs are embedded within standard XQuery expressions. Both approaches are orthogonal to the approach presented in this section in that they focus on expressing the overall workflow in a distributed XQuery variant, whereas the strategies above focus on a dataflow paradigm with actor abstractions, along the lines of Kahn process networks [Kah74]. A different approach is taken in Active XML [ABC+03], where XML documents contain special nodes that represent calls to web services. This approach constitutes a different type of computation model applied more directly to P2P settings, whereas approach presented here is targeted at XML process networks, e.g., applied to the area of scientific applications deployed within in cluster environments.

In [PA06], a number of different parallel execution strategies are described for Grid-based workflows, which includes the "parallel-for" construct implemented by the Askalon Grid workflow system [QF07, FPD+05]. This construct is used explicitly when designing workflows, and implements a variant of the map higher-order function in which the function is called over each element of a given list in parallel. The approach presented here extends this work for XML processing pipelines in a number of ways, e.g., by implicitly applying data parallelism based on task scope expressions at runtime and by supporting nested data collections as opposed to flat lists of values.

8.8 A New Workflow System Prototype

Based on the desiderata and case studies presented earlier, this section will present a prototype of a new scientific workflow model that combines beneficial properties seen mentioned earlier. Due to the lack of time, there exists only a prototypical design of this new workflow model that is designed to improve a modified version of the Kuration workflow presented in Section 8.6.

8.8.1 New Workflow System Overview

The new workflow description language is designed for SIMPLICITY and is based on the PN model of computation. Additional annotations, to the workflow description language, enable performance optimizations that can be applied during workflow execution. Models of computation in Ptolemy use stream parallelism and provide distinct model elements before and after a sub-workflow to allow instance parallelism of the enclosed sub-workflows. However, such explicit approaches increase the complexity of the workflow model. The new model proposed here supports stream parallelism and provides optional annotations on actors for statefulness. Furthermore, provenance of a workflow can be analyzed to identify stateless actors that support instance parallelism and the system can propose to add additional annotations. In MoCs with structured data models like COMAD, parallelism based on streaming is restricted by (1) the definition of data structures, e.g., the order of elements, (2) the placement of annotations, and (3) operations on multiple input data structures, e.g., the cross-product computation performed on lists from multiple data bindings. The new model of computation supports a structured data model similar to Restflow (cf. Section 3.4) but mitigates the performance penalties by dropping the requirement for ordered data structures. Instead a descriptor is added to each data token that contains the ID and location in the data structure. Similar to COMAD and Restflow, data tokens also carry provenance information and the data stream can be enriched with additional provenance tokens, e.g., deleted data tokens, that are not processed by actors but recorded in a provenance trace.

The workflow execution engine is build upon the concurrent and distributed event-driven computing framework Akka [Akk13]. Akka provides a framework for concurrent and distributed actor invocations on distributed resources and has support for fault tolerance mechanisms. The new workflow system supports the parallel execution of a workflow on distributed resources such as multi-core systems, grids, and clouds through Akka. This allows the construction of workflow with similar pattern and features as found in MapReduce. Furthermore, the workflow system is designed to use available annotations to speed up the execution and to reach the maximum degree of parallelism for actor invocations. To that end, the model of computation supports *streaming*, i.e., concurrent actor invocations exchanging data when it is available. Furthermore, the statefulness annotation enables *instance parallelism*, i.e., invoking multiple instances of one actor at the same time. Figure 8.26 illustrated the principal components of the workflow system.

Planned extensions. Building on the dissertation of Daniel Zinn [Zin10] and [ZBML09b], a smart data shipping strategy can be added to the distributed computing framework of Akka which balances data transfer times and actor invocation times. If a data transfer requires a longer time than the following actor invocation given the different machine loads, the actor will be executed locally, i.e., on the same node where the data is stored. However, if some critical resources, such as CPU cycles, memory or IO bandwidth run low, the actor my be executed on

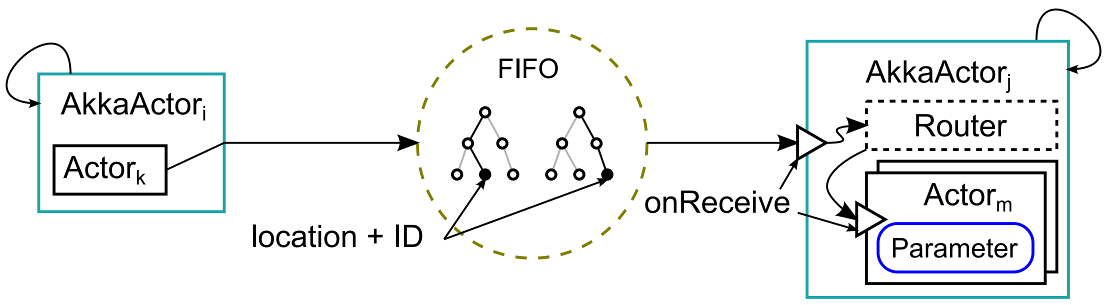


Figure 8.26: Components of the newly proposed workflow system.

a different node, by shipping the input data to this node.

8.8.2 Implementation

This section briefly describes some implementation details of the prototype for the new workflow system. The system is implemented in Java and uses the Java Akka framework. All actors in the new system are encapsulated in a `UntypedActor` Akka actor. To allow parallel actor invocations, stateless actors are encapsulated into a *router* actor that distributes messages according to a user-definable strategy to a given number of actor instances. By default a shortest-message-queue strategy is used. Future extensions can automatically determine the number of router destinations, i.e., routees, that are required. In combination with the statefulness annotation in the workflow description, this implements the Distributor and Commutator pattern used in Ptolemy. Akka actors in the new system have exactly one receiving mailbox, i.e., queue. Whenever a message arrives and no other message is processed the `onReceive()` method of the actor is executed by the Akka framework. Ports can be simulated by messages, i.e., tokens, with different attributes. In the new system, each received data token triggers an invocations and by default each invocation consumes exactly one token but can produce any number of tokens. Due to its similarity with the PN model of computation, simple PN actors with a token consumption rate of 1 can be translated to actors in the new system easily. Other actors require a translation layer that collects all required tokens before invoking the actor's main function. Tokens are immutable messages in Akka that contain the actual payload data and meta-data such as the path in the structured data model, a unique ID, or provenance. If tokens are not modified they can be passed along but in order to modify a token the underlying message has to be copied. In contrast to the PN MoC, the workflow execution shut down is triggered by the source actors and then propagated. Once all actors have terminated themselves, the workflow execution terminates as well.

8.8.3 Kuration Workflow in New System

The performance of the newly proposed workflow system is demonstrated on the small Kuration workflow shown in Figure 8.11. This workflow was modeled using the new workflow systems and the same functions that are used in the COMAD actors are wrapped in the new actors. In both workflow variants, input data is read and output data is written to the same location. Both workflows use the same remote services.

To evaluate which performance improvements can be reached, both workflow variants were executed repeatedly (at least 4 times) for each of the four different datasets. Table 8.3 shows the average runtime and standard deviation of the runtime for all four datasets on COMAD and on the new system. The new workflow system based on Akka is significantly faster in curating all records. For each record, approximately four calls to remote web services are made. Since web service calls might return with large delays, the serial processing of records in COMAD within one actor cant compensate for rare large delays. In the new workflow system based on Akka, multiple instances of one actor process web service requests in parallel and a large delay in one response only slows down the processing of this instance. Another advantages of the new workflow system is the simple and compact design that reduces the memory consumption.

The results of this preliminary test confirm the importance of parallel actor invocations in a workflow execution. Since this instance-parallelism can only be used if the actor is stateless, the annotation if a actor is stateless is critical for good performance as well. The next chapter will present a fault tolerance approach for scientific workflows that demonstrates another use of information about the state of an actor.

Metric	DAG	SDF	PN	COMAD	Restflow	PTN+NRC	Taverna	Vistrails	MR Online
Dataflow oriented	-	+	+	+	+	+	+	+	+
Ordered dataflow	-	+	+	+	+	-	+	-	-
Complex data structures	-	+/-	+/-	+	+	+	+	+/-	-
Data structure independent	-	+/-	+/-	+	+	-	+/-	+	+
Arbitrarily accessible data	-	+/-	+/-	+	+	-	-	+/-	-
Low connection Complexity	-	-	-	+	+	-	-	-	+
Stateful actors	-	+	+	+	+	-	+	-	-
Statefulness declared	-	-	-	-	+	-	-	-	-
Data-driven	-	-	+	+	+	+	+	+	+
Multiple invocations	-	+	+	+	+	+	+	+	+
Concurrent	+	-	+	+	+	+	+	+	+
Streaming	-	-	+	+	+	+	+	+	+/-
Side-effect handling	-	-	-	-	-	-	-	-	-
Provenance	-	+	+	+	+/-	-	+	+	-
Workflow evolution provenance	-	+	+	-	-	-	-	+	-

Table 8.1: Comparison of MoCs The following abbreviations are used: PTN+NRC - Petri nets with nested relational calculus. MR Online - MapReduce Online. A system has a feature +, has it partially +/-, or does not have it -. In case a criteria does not apply due to a missing concept, the cell is marked with a '-'.

Measure	Original Workflow	Comad Workflow
No. of Nodes	273	218
No. of Edge	387	325
No. of used Classes	11	0
Avg. No. of Input Ports	2.1502	2.1376
Avg. No. of Output Ports	1.5311	1.7064
Avg. No. of Ports	3.6813	3.8440
Avg. No. of Parameters	2.5128	3.2541
Parameter Complexity	0.0641	0.1591
Avg. Model Size	7.8000	4.9545
No. of nested Workflow Levels	5	7
No. of Types	10	11

Table 8.2: Comparison of the original workflow model and the COMAD model for the Monitoring Workflow

Dataset	COMAD		new system	
	time [s]	σ [s]	time [s]	σ [s]
1 (535 records)	871.058	473.218	64.240	0.589
2 (915 records)	1134.977	602.052	115.640	11.108
3 (1751 records)	1474.433	472.485	207.210	22.246
4 (4170 records)	4117.219	2243.602	592.427	85.221

Table 8.3: Runtime of the two Kuration workflow variants. Datasets include different records and the runtime is not necessarily proportional to the number of records. The new workflow system based on Akka shows significantly better performance.

Chapter 9

Workflow Fault-Tolerance and Provenance

This chapter presents an approach to increase the fault tolerance of a scientific workflow execution in the Kepler system that was developed by the author and presented in [KRZ+11]. This approach connects the concepts of Datalog, provenance analysis and workflow systems presented earlier.

9.1 Introduction

Besides automating program execution and data movement, scientific workflow systems are developed to provide mechanisms for fault tolerance during workflow execution. There have been approaches that re-execute individual workflow components after a fault [MSRO+10]. However, less research has been done on how to handle failures at the level of the workflow itself, e.g., when a faulty actor or a power failure takes down the workflow engine itself. Circumstances that lead to (involuntary) workflow failures are, for example software errors, power outages or hardware failures—common in large supercomputer environments. Also, a running workflow might be aborted voluntarily so that it can be migrated to another location, e.g., in case of unexpected system maintenance.

Since typical scientific workflows often contain compute and data intensive steps, a simple "restart-from-scratch" strategy to recover a crashed workflow is impractical. In this chapter, two strategies (namely *replay* and *checkpoint*) are developed which allow workflows to be resumed while mostly avoiding redundant re-execution of work performed prior to the fault. The necessary book-keeping information to allow these optimizations is extracted from provenance information that scientific workflow systems often already record for data lineage reasons, allowing this approach to be deployed with minimal additional runtime overhead.

Commonly used models for provenance are the Read/Write model [BML+06], and the Open Provenance Model (OPM) [MCF+10]. In both provenance models, events are recorded when actors consume tokens (`read` or `used_by` events) and produce tokens (`write` or `generated_by` events). Thus, the stored provenance data effectively persists the data tokens that have been flowing across workflow channels. This chapter shows how this data can be used to efficiently recover faulty workflow executions. Here, the main challenges arise due to (1) the existence of stateful multi-invocation actors, i.e., actors that maintain state from one invocation to the next; (2) main-memory actor-actor data transport, i.e. dataflow achieved within the workflow engine and without persisting the data to disk[1]; and (3) non-trivial scheduling algorithms for multiple actor invocations based on data availability.

Example. Consider the small scientific pipeline shown in Figure 9.1, which carries out two tasks automated by the WATERS workflow described in [HRM+10]. As in the full implementation of WATERS, streaming data and stateful multi-invocation actors make an efficient recovery process non-trivial.

The actor `SequenceSource` reads DNA sequences from a text file, emitting one DNA sequence token via the `output` port per invocation. The total number of invocations of `SequenceSource` is determined by the contents of the input file. On the `group_done` port, it outputs a 'true' token when the sequence output is the last of a predefined group of sequences, and 'false' otherwise. `Align` consumes one DNA sequence token per invocation, aligns it to a reference model, and outputs the aligned sequence. The `ChimeraFilter` actor receives the individually aligned sequences from `Align` and the information about grouping from the `SequenceSource`. In contrast to `Align`, `ChimeraFilter` accumulates input sequences, one sequence per invocation, without producing any output tokens until the last sequence of each group arrives. `ChimeraFilter` then

[1]Even if data is persisted to disk due to large data sizes, data handles are usually kept in main memory.

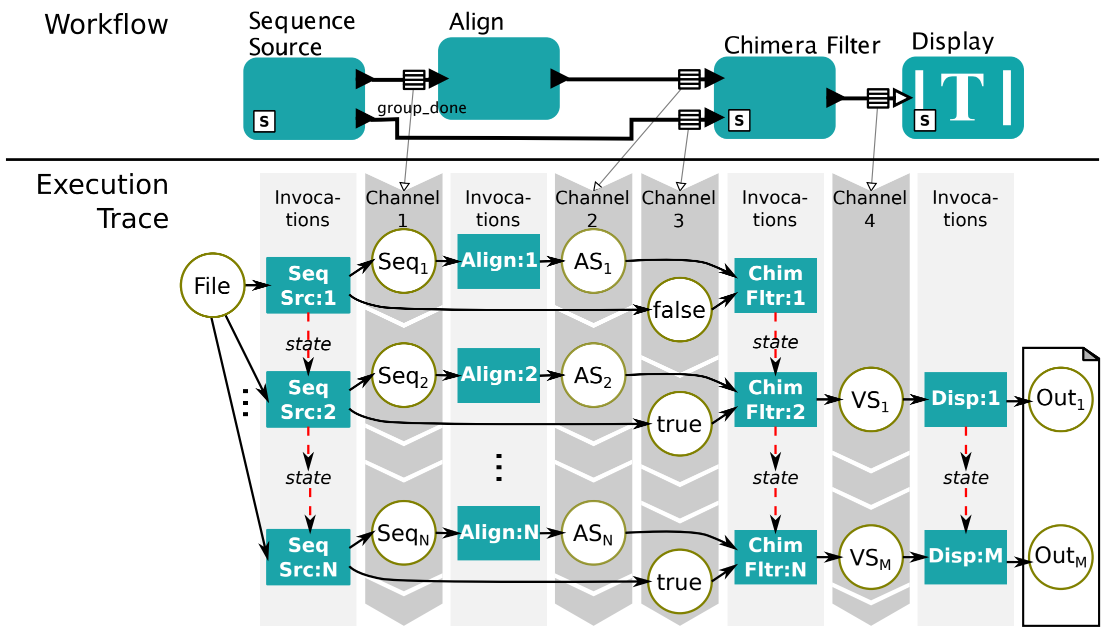


Figure 9.1: Example workflow with stateful actors: To recover the workflow execution after a fault, unconsumed tokens inside workflow channels and internal states of all actors except the stateless `Align` have to be restored.

checks the entire group for chimeras (spurious sequences often introduced during biochemical amplification of DNA), outputs the acceptable sequences en masse, and clears its accumulated list of sequences.

All actors but `Alignment` are stateful across invocations: `SequenceSource` and `Display` maintain as state the position within the input file and the output produced thus far, respectively. `ChimeraFilter`'s state is the list of sequences that it has seen so far in the current group. Now, consider what runtime information would be lost if a fault occurred during the execution of this workflow. Lost information would include (1) the content of the channels between actors, i.e., tokens produced by actors but not yet consumed; (2) the point in the workflow execution schedule as observed by the workflow engine; and (3) the internal states of all actors. Correctly resuming workflow execution requires reconstructing all of this information. This chapter shows how to do so efficiently with low runtime overhead.

9.2 Related Work

A basic fault tolerance mechanism exists for workflows managed by DAGMan [HC07]. Jobs are scheduled according to a directed graph that represents dependencies between those jobs. Initially the rescue-DAG contains the whole DAG but as soon as a job executes successfully, this job is removed from the rescue-DAG. If a failure occurs, the workflow execution can be resumed using the rescue-DAG, only repeating jobs that were interrupted.

Feng et al. [FL08] present a mechanism for fault management within a simulation environment under real time conditions. Starting from a “checkpoint” in the execution of an actor, state changes are recorded incrementally and can then be undone in a “rollback”. This backtracking approach allows to capture the state of an actor through a preprocessing step that adds special handlers for internal state changes wherever a field of an actor is modified. However, this solution can only be used during runtime of the workflow system. It does not provide checkpoints that cover the full state, and, more importantly, no persistent state storage is available for access after a workflow crash.

Dan Crawl et al. [CA08] employed provenance records for fault tolerance. Their Kepler framework allows the user to model the reactions upon invocation failures. The user can either specify a different actor that should be executed or that the same actor should be invoked again using input data stored in provenance records. However, they don’t provide a fast recovery of the whole workflow system. Neither is the approach applicable for stateful actors.

Fault tolerance in scientific workflows has often been addressed using caching strategies. While still requiring a complete restart of the workflow execution, computation results of previous actor invocations are stored and reused. Swift [ZHC$^+$07] extends the rescue-DAG approach by adding such caching. During actor execution, a cache is consulted (indexed by the input data), and if an associated output is found, it will be used, avoiding redundant computation. Swift also employs this strategy for optimizing the re-execution of workflow with partially changed inputs. Podhorszki et al. [PLK07] described a checkpoint feature implemented in the ProcessFileRT actor; this actor uses a cache to avoid redundant computations. A very similar approach was implemented by Hartman et al. [HRM$^+$10]. Both techniques are used to achieve higher efficiency for computation and allow a faster re-execution of the workflow. However, these implementations are highly customized to their respective use cases and integrated in one or several actors rather being a feature of the framework. Also, [PLK07] assumes that only external programs are compute intensive, which is not always the case, as can be seen in [HRM+10], where actors perform compute intensive calculations within the workflow system. Furthermore, caching strategies can only be applied to stateless actors, making this approach very limited. In contrast, the approach presented in this chapter aims to integrate fault tolerance mechanisms into the workflow engine. Stateless actors are not re-executed during a recovery, since input and corresponding outputs are available in provenance, and the actor state does not need to be restored.

Wang et al. [WLF+09] presented a transactional approach for scientific workflows. Here, all effects of arbitrary subworkflows are either completed successfully or in case of any failure undone completely (the dataflow-oriented hierarchical atomicity model described in [WLF+09]). In addition, it provides a dataflow-oriented provenance model for those workflows. The authors assumed that actors are white boxes, where data dependencies between input and output tokens can be observed. They describe a smart re-run approach similar to those presented by Podhorszki et al. and Hartman et al. [HRM+10]. Input data of actors is compared to previous inputs, and if an actor is fired with the same data, the output can easily be restored from provenance information rather than re-executing the actor. This white box approach differs from the black box approach presented here which requires setting the internal state of stateful actors.

9.3 Fault Tolerance Approach

The fault tolerance approach described here generalizes the Rescue DAG method [Fre, DBG+04, HC07], which is used to recover DAGMan workflows after workflow crashes. DAGMan is a single-invocation model of computation, i.e., all actors are invoked only once with a read-input, compute, and write-output behavior. The Rescue DAG is a sub-graph of the workflow DAG containing exactly those actors that have not yet finished executing successfully. After a crash, the rescue DAG is executed by DAGMan, which completes the workflow execution.

To facilitate the execution of workflows on streaming data, several models of computation (e.g., synchronous data flow (SDF) [LM87b], process networks (PN) [LM08], collection oriented modeling and design (COMAD) [DZM+11] and Taverna [TMG+08]) allow actors to have multiple invocations. If the Rescue-DAG approach were applied directly to workflows based on these models of computation, i.e. if all actors that had not completed all of their invocations were restarted, then in many cases a large fraction of the actors in a resumed workflow would be re-executed from the beginning. Instead, the approach presented here aims to resume each actor after its last successful invocation. The difficulties of this approach are the following: (1) The unfolded trace graph (which roughly corresponds to the rescue DAG) is not known a priori but is implicitly determined by the input data. (2) Actors can maintain internal state from invocation to invocation. This state must be restored. (3) The considered models of computation (e.g., SDF, PN, COMAD, Taverna) explicitly model the flow of data across channels, and the corresponding workflow engines perform these data transfers at run time. A successful recovery mechanism in such systems thus needs to re-initialize these internal communication channels to a consistent state. In contrast, data movement in DAGMan workflows is handled by the actors opaquely to the DAGMan scheduler (e.g., via naming conventions) or by a separate system called Stork [KL04]; materializing on disk all data passing between actors simplifies fault-tolerance in these cases.

The following section presents a unifying model of workflow definitions and provenance information collected during workflow. A relational schema facilitates the definition of workflow recovery strategies using logic rules. This approach is described here for the SDF and PN models of computation, but the representation generalizes to other dataflow models such as the one used in RestFlow [MM10], Taverna and Vistrails [BCS$^+$05].

9.3.1 Basic Workflow Model

Scientific workflow systems use different languages to describe workflows and different semantics to execute them. However, since most scientific workflow systems are based on dataflow networks [Kah74, LM08] as discussed earlier, a common core that describes the basic workflow structure can be found in every model of computation (see Figure 9.2).

Core model. Many workflow description languages allow nesting, i.e., embedding a sub-workflow within a workflow. The relation `subworkflow(W,Pa)` supports this nesting in our schema and stores a tuple containing the sub-workflow name `W` and the parent workflow name

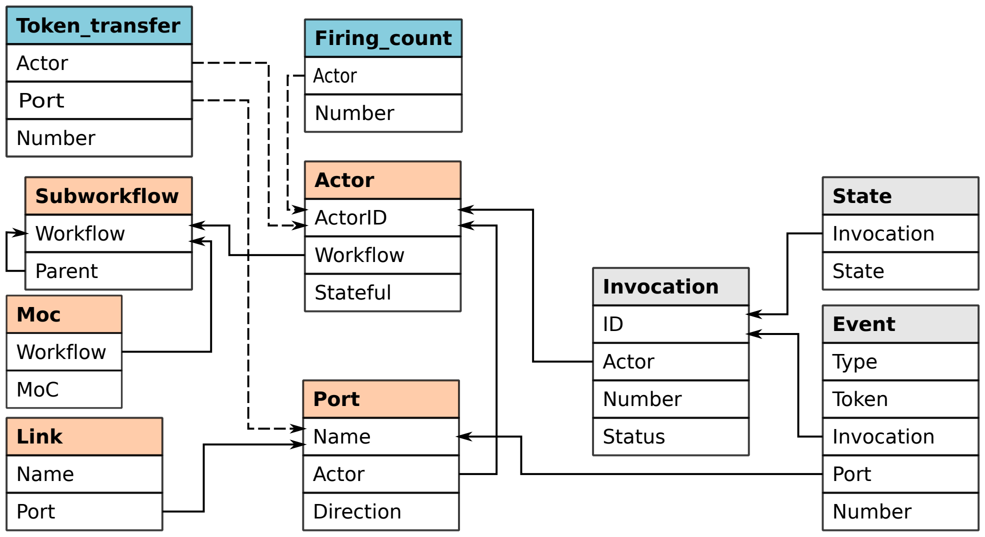


Figure 9.2: Unified model. Relations to describe the workflow are shown on the left. The three relations on the right summarize provenance information. The blue highlighted relations capture SDF specific extensions.

`Pa`. Each workflow in this hierarchy is associated with a model of computation (MoC) using the relation `moc(W,M)` that assigns the MoC `M` to the workflow `W`.

Actors represent computational functions that are either implemented using the language of the workflow system or performed by calling external programs. The predicate `actor(A,W,S)` embeds an actor with unique name `A` into the workflow `W`. The flag `S` specifies whether the actor is stateful or stateless.

Although the data shipping model is implemented differently in various workflow systems, it can be modeled collectively as follows: Each actor has named ports, which send and receive data tokens. One output port can be connected to many input ports. In this situation the token is cloned and sent to all receivers. Connecting multiple output ports to one channel is prohibited due to the otherwise resulting write conflicts. Ports are expressed with the predicate `port(A,P,D)` in our schema. The port with name `P` is attached to actor `A`. `D` specifies the direction in which data is sent, i.e., `in` or `out`. Ports are linked to each other using `link(A,P,L)`. Port `P` of actor `A` is connected to a link with name `L`.

Application to Process Networks with Firing (PN). A Process Network, as defined by [Kah74], is a general model of computation for distributed systems. In Kahn PN, computational processes (actors) are communicating with each other through unidirectional FIFO channels of unlimited size. Workflows of the model *PN with firings* [LM08], a refinement of Kahn PN, can be described with the four core relations `Subworkflow`, `Actor`, `Port`, and `Link`. The PN execution semantics allow a high level of parallelism, i.e., all actors can be invoked at the same time. After an invocation ended, the actor will be invoked again. This procedure stops either when the actor explicitly requests to be stopped or by reaching the end of the workflow execution. A PN workflow ends when all remaining running invocations are deadlocked on reading from an input port.

Application to Synchronous Data Flow (SDF). Besides the data captured by the four core relations, workflow models can provide additional information. As an example, SDF workflow descriptions require annotations on ports. In SDF, output ports are annotated with a fixed *token production rate* and input ports have a fixed *token consumption rate*. Both rates are associated with ports using the predicate `token_transfer(A,P,N)` in our model. During an invocation, each actor `A` is required to consume/produce `N` tokens from the input/output port `P`.

Another extension is the *firing count* of an actor that specifies the maximum number of actor invocations during one workflow execution. The predicate `firing_count(A,N)` provides this number (`N`) for an actor `A`.

Unlike in PN, where the actors synchronize themselves through channels, the execution of SDF is based on a static schedule that is repeatedly executed in *rounds*. The number of firings of each actor per round is determined by solving balance equations based on token production and consumption rates [LM87b].

9.3.2 Review of Provenance Model

Another critical part of the approach is the definition of a unifying provenance model. It defines which observables are recorded during runtime. The open provenance model (OPM) [MFF+08] captures the following basic observables: (1) artifact generation, i.e., token production (2) artifact use, i.e., token consumption (3) control-flow dependencies, i.e., `was triggered by` relation, and (4) data dependencies, i.e., `was derived from` relation. A more complex provenance schema was defined by Crawl et al. in [CA08]. It captures the OPM observables in more detail, e.g., it provides time stamps for the beginning and end of invocations. In addition, it records meta data about the workflow execution as well as the evolution history of a workflow.

The provenance model used here builds up on the basic observables from OPM and adds additional details about events that occurred during an invocation cycle. As soon as an invocation starts, the actor name `A` and its corresponding invocation number `N` are stored in the relation `invocation(I,A,N,Z)` with the status attribute `Z` set to `running`. A unique identifier `I` is assigned to each invocation. Some models of computation allow an actor to indicate that all invocations are completed, for instance if the maximum firing count in SDF is reached. This information is captured in our provenance model as well. When an actor successfully completes an invocation and indicates that it will execute again, the status attribute in the corresponding provenance record should be updated to `iterating`. Otherwise, this attribute status should be set to `done`.

The second observable process in the model used here is the flow of tokens. Many workflow engines treat channels that define the dataflow as first-class citizens of the model. The dependencies between data tokens are of general interest for provenance. They can be inferred from the core workflow model in combination with the token consumption (read) and production (write) events.

The model stores read and write events in the `event(Y,T,I,Po,N)` relation. The first entry `Y` determines the event type, i.e., token production events are indicated by the constant `w` while consumption events are encoded with the value `r`. The data token value `T` is stored directly. The following two attributes specify which actor invocation `I` triggered this event and on which port `Po` it was observed. The last element `N` in the tuple is an integer value that is used to establish an order for events during the same actor invocation on the same port. Establishing an order using timestamps is not practical because of limited resolution and time synchronization issues.

Based on the `event` relation the queues of all channels can be reconstructed for an arbitrary point in time. Figure 9.3 shows a queue at the time of a workflow failure on top. Provenance can be used to restore the whole history of this queue (shown in the middle). Based on this history, one can determine the *rescue sequence* of tokens that are independent of failed invocations, i.e., tokens in state `S2` and `S4`.

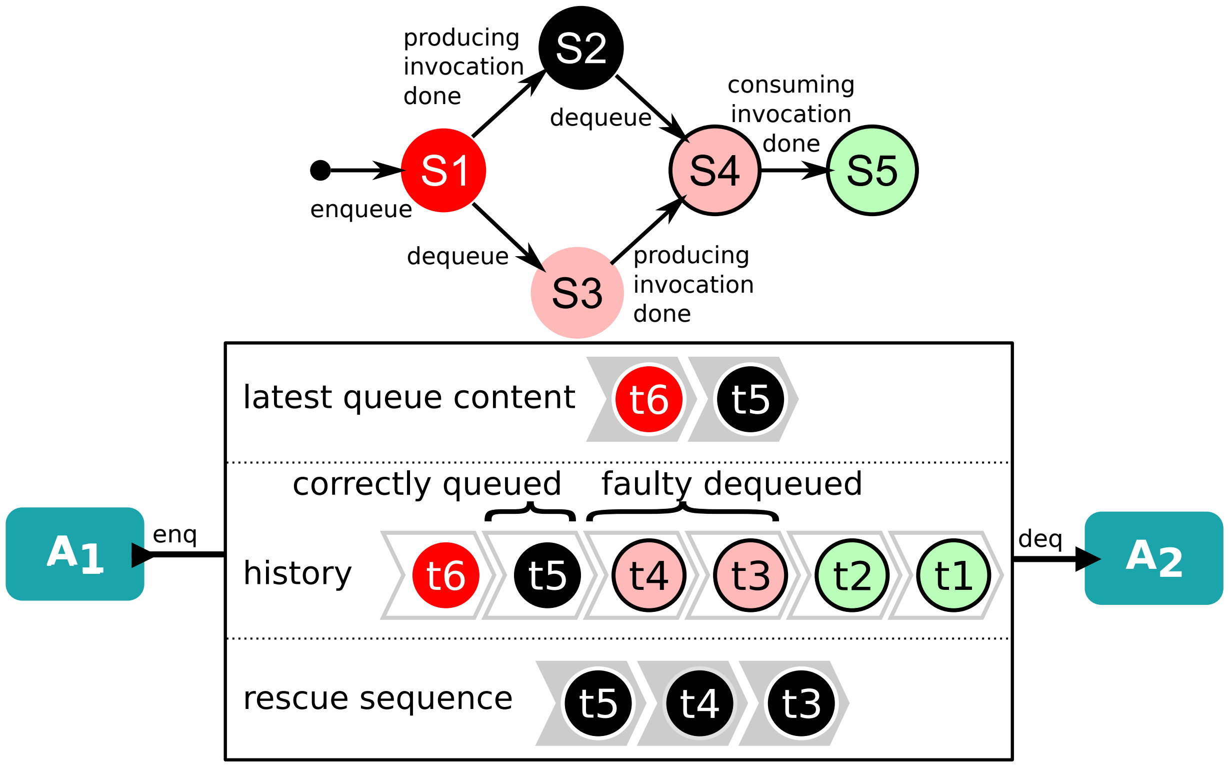


Figure 9.3: Input queues with history and token state: Each token produced during workflow execution can be in one of five states. Events on the producing and consuming actors trigger transitions between token states, shown on the left. The right graph shows three views of a channel: (1) the current content of the queue during an execution in the first row, (2) the history of all tokens passed through this channel associated with their state in the middle row, and (3) the *rescue sequence* of tokens that needs to be restored in the third row.

9.4 Recovery Strategies

Depending on the model of computation and the available provenance data, different recovery approaches can be used. Here, the two strategies *replay* and *checkpoint* are presented.

9.4.1 The Replay Strategy: Fast-Forwarding Actors

Re-running the entire workflow from the beginning is a naive recovery strategy. It is also impractical in many cases, such as when a long-running workflow fails a significant period of time into its execution. The role of provenance in restoring a workflow execution is similar to that of log files used in database recovery.

Stage 1. In the first stage of the replay strategy, the point of a failure is determined using provenance information. Invocations of actors that were running when the fault occurred are considered faulty and their effects have to be undone. Query (1) retrieves the invocation identifiers `I` of faulty invocations.

```
faulty_invoc(I) :- invocation(I,_,_,running).                                    (1)
```

Note that in SDF models, faulty invocations can be determined even without an explicit `running` flag. An actor invocation is faulty if the number of consumed and produced tokens as recorded in the provenance data does not match the expected number declared in the workflow model.

Actors with invocation status `done` are not recovered, since they are not needed for further execution. All other actors `A` are retrieved by query (2) and they need to be recovered.

```
finished_actors(A) :- invocation(_,A,_,done).
restart_actors(A) :- actor(A,_,_), not finished_actors(A).                       (2)
```

Stage 2. If an actor is stateless, it is ready to be resumed without further handling. However, if an actor is stateful, its internal state needs to be restored to its *pre-failure state*, i.e., the state after the last successful invocation. Each actor is executed individually by presenting it with all input data the actor received during successful invocations. This input data is retrieved from the provenance log, where it is readily available, if the input queues are persisted. The `replay(A,I)` query (3) extracts the identifiers of all actor invocations that need to be replayed. The tokens needed for those re-invocations are provided by (4). This query retrieves for each port `P` of actor `A` the tokens `T` that are needed to replay invocation `I`. `N` is the sequence number of token `T` at input port (queue) `P`. The replay does not need to be done in the same order as in the original workflow schedule. All actors can be re-executed in parallel using only the input data recorded as provenance. The actor output could either be discarded or checked against the recorded provenance to verify the workflow execution.

```
replay(A,I) :- actor(A,_,stateful), invocation(I,A,_,_),                (3)
      not faulty_invoc(I).
replay_token(A,P,I,T,N) :- replay(A,I), event(r,T,I,P,N).               (4)
```

In order to correctly recover a workflow execution, the problem of side-effects still needs to be addressed. Either stateful actors should be entirely free of side-effects or side-effects should be idempotent. That is, it must not matter whether the side-effect is performed once or multiple times. Examples of side-effects in scientific workflows include the creation or deletion of files, or sending emails. Deleting a file (without faulting if the file does not exist) is an idempotent operation. Further, creating a file is idempotent if an existing file is overwritten. Sending an email is, strictly speaking, not idempotent, since if done multiple times, multiple emails will be sent.

Stage 3. Once all actors are instantiated and in pre-failure state, the queues have to be initialized with the *restore sequence*, i.e., those valid tokens that were present before the execution failed. Tokens created by faulty invocations of an actor should be removed, and those consumed by such an actor should be restored. This information is available in basic workflow provenance and can be queried using (5). For each port `Po` of an actor `A` the query retrieves tokens `T` with the main order specified by the invocation order `N1`. However, if multiple tokens are produced in one invocation, the token order `N2` is used for further ordering.

The auxiliary view `invoc_read(A,P,T)` contains all actors `A` and the corresponding ports `P` that read token `T`. The view `connect(A1,P1,C,A2,P2)` returns all output ports `P1` of actor `A1` that are connected to actor `A2` over input port `P2` through channel `C`. Using the auxiliary rule (5.1) computes the queue content in state `S2` (see Figure 9.3), i.e., tokens that were written by another actor but not yet read by actor `A2` on port `P2`. The second rule (5.2) adds back the queue content in state `S4`, i.e., tokens that were read by a failed invocation of actor `A2`.

```
current_queue(A2,P2,T,N1,M1) :- queue_s2(A2,P2,T,N1,M1).               (5)
current_queue(A2,P2,T,N1,M1) :- queue_s4(A2,P2,T,N1,M1).
```

```
queue_s2(A2,P2,T,N1,M1) :- connect(A1,P1,C,A2,P2),                    (5.1)
      invocation(I1,A1,N1,_), event(w,T,I1,P1,M1),
      not invoc_read(A2,P2,T), not faulty_invoc(I1).
queue_s4(A2,P2,T,N1,M1) :- connect(A1,P1,C,A2,P2),                    (5.2)
      invocation(I1,A1,N1,_), event(w,T,I1,P1,M1),
      invocation(I2,A2,_,_),event(r,T,I2,P2,_),
      faulty_invoc(I2).
invoc_read(A,P,T) :- invocation(I,A,_,_), event(r,T,I,P,_).
connect(A1,P1,C,A2,P2) :- link(A1,P1,C), link(A2,P2,C),
      port(A1,P1,out), port(A2,P2,in).
```

Stage 4. After restoring actors and recreating the queues, faulty invocations of actors that produced tokens which were in state S3 have to be repeated in a "sandbox". This ensures that tokens in state S3 are not sent to the output port after being produced but are discarded instead. Should these tokens be send, invocations based on them would be duplicated. Rule (6) determines tokens T that were in state S3 and it returns the invocation ID I, the port P this token was sent from, and sequence number in which the token was produced. Query (7) determines which invocations produced tokens in state S3 and therefore have to be repeated in a sandbox environment.

```
queue_s3(I,P,T,N) :- invocation(I,A1,_,_),                            (6)
      faulty_invoc(I), event(w,T,I,P,N),
      connect(A1,P,C,A2,P2), invocation(I2,A2,_,_),
      not faulty_invoc(I2), event(r,T,I2,P2,_).
invoc_sandbox(I) :- faulty_invoc(I), queue_s3(I,_,_,_,_).             (7)
```

Now the workflow is ready to be resumed. The recovery system provides information about where to begin execution (i.e., the actor at which the failure occurred) to the execution engine (e.g., the SDF scheduler) and then the appropriate model of computation controls execution from that point on.

The most expensive operation in this strategy is the re-execution of stateful actors that is required to reset the actor to its pre-failure state. The *checkpoint strategy* presented next provides a solution to avoid this excessive cost.

9.4.2 The Checkpoint Strategy: Using State Information

Many existing workflow systems are shipped with stateful actors or new actors are developed that maintain state. Because actors in scientific workflows usually have complex and long-running computations to perform, the replay strategy can be very time-consuming or even impractical.

Current provenance models, such as the one used in [CA08], either do not include the state of actors or record limited information about state as in [FL08], which is insufficient to restore workflows. The Read-Write-Reset model (as presented in [LPA$^+$08]), e.g., records only state reset events, which specify that an actor is in its initial state again. This can be seen as a special case of checkpointing, where states are only recorded when they are equal to the initial state.

To support a faster recovery, an actor's state should be a distinct recorded entity for provenance. Recording state information not only helps to recover workflows, but also makes provenance traces more meaningful: Instead of linking an output token of a stateful actor to all input tokens, our model links it to the state input and the current input only.

An actor's state can be recorded by the workflow engine at any arbitrary point in time when the actor is not currently invoked. To integrate checkpointing into the order of events, state information is stored immediately after an invocation, using the invocation identifier as a reference for the state. The predicate `state(I,S)` stores the actor's state `S` together with the identifier of the preceding invocation `I` of that actor. The information required to represent an actor state depends on the workflow system implementation.

The procedure for a workflow recovery based on the checkpoint strategy is summarized in the following graph:

Stage 1. Given this additional state information the workflow recovery engine can speed up the recovery process. This checkpoint strategy is based on the replay strategy but extends it with checkpointing.

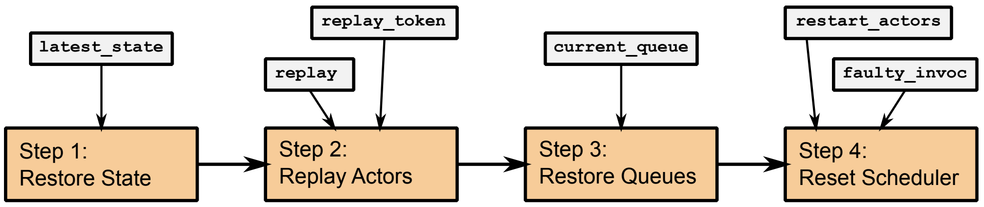


Figure 9.4: Checkpoint strategy. Each recovery step uses provenance views shown on top.

When normally executing the workflow, state is recorded in provenance. In case of a fault, the recovery system first detects the point of failure. Then the provenance is searched for all checkpoints written for stateful actors. Rule (8) retrieves the state `S` of each invocation `I` of a given actor `A`. If no state was recorded then the invocation will not be contained in this relation:

```
restored_state(A,I,N,S) :- actor(A,_,stateful),                                  (8)
      invocation(I,A,N,_),state(I,S), not faulty_invoc(I).
```

If states were stored for an actor, this actor is updated with the latest available state. Rule (9) will determine the latest recoverable invocation `I` and the *restorable pre-failure state* `S` captured after that invocation.

```
restored_stateGTN(A,I,N2) :- restored_state(A,I,N,_), N > N2.
latest_state(A,I,S) :- restored_state(A,I,N,S),                                  (9)
      not restored_stateGTN(A,I,N).
```

Stage 2. Now only those successfully completed invocations that started after the checkpoint have to be replayed. This will use the same methods described above for the replay strategy.

Stage 3. After all actors are set up, the queues have to be filled with the *rescue sequence.*

Stage 4. Now the workflow is ready to be resumed as in the replay strategy. First, faulty invocations are resumed in a sand-box to prevent sending tokens that were already successfully used. Finally, the recovery system provides all necessary information to the scheduler to determine with which actor's invocations to start resuming normal workflow execution.

9.4.3 Recovering SDF and PN Workflows

The SDF example below demonstrates the checkpoint strategy. Later, the differences when dealing with PN models are explained.

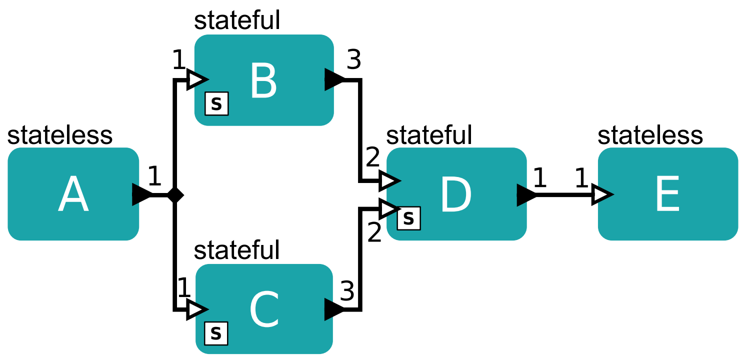


Figure 9.5: SDF workflow example with state annotations.

Synchronous Dataflow (SDF). Figure 9.5 shows a sample SDF workflow with annotated ports. Actors A and E are stateless while all other actors have a state. A is a source and will output one data item (token) each time the actor is invoked. A also has a *firing count* of two, which limits the total number of invocations. Actors B and C consume one token on their input ports and output three tokens. Outputs are in a fixed, but distinguishable, order, so that an execution can fail between the production of two tokens. Actor D will receive two tokens in each invocation from both B and C and will output one new token.

The schedule in Figure 9.6 was computed, as usual, before the workflow execution begins, based on the token production and consumption rates in the model. Actors were then invoked according to the schedule until the point at which the workflow crashed. All invocations up to the second invocation of A (A:2) completed successfully, invocation B:2 was still running and all other invocations were scheduled for the future.

The failed workflow execution, together with checkpointing and data shipping, is summarized in Figure 9.7. For recovery, the workflow description as well as the recorded provenance is used by our checkpoint strategy.

The recovery process is performed in the following stages: Stateless actors are in the correct

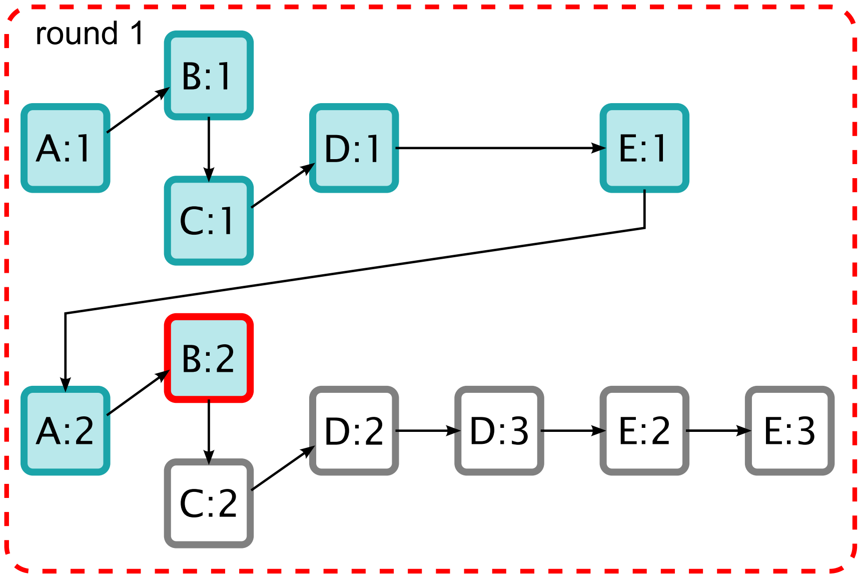


Figure 9.6: Schedule corresponding to Figure 9.5. The schedule describes the execution order. The red circle indicates the failure during the second invocation of B.

state (i.e., pre-failure state) immediately after initialization. That is why actor E is in its proper state after simple initialization and actor A is identified as done and will be skipped. Both actors B and D are stateful and a checkpoint is found in the provenance. Therefore, the recovery system instantiates these actors and restores them to the latest recorded state. The state at this point in the recovery process is shown in Figure 9.8 above the first dotted line.

Second, stateful actors that either have no checkpoints stored (e.g., actor C) or that have successful invocations after the last checkpoint need to be have those invocations replayed. The corresponding input token t1 is retrieved from the provenance store.

In the next stage, all queues are restored. Since the second invocation of actor B failed, the consumed token t9 is restored back to the queue. Additionally, all tokens that were produced but never consumed (e.g., t4 and t7) are restored to their respective queues.

After all queues are rebuilt, the recovery system has to initialize the SDF scheduler so execution begins with the actor that was interrupted by the crash. After setting the next active actor to B in the SDF scheduler, the recovery system can hand over the execution to the workflow execution engine. This final state is shown in Figure 9.8, and when compared to Figure 9.7, the recovery time is visibly shorter when compared to the original running time.

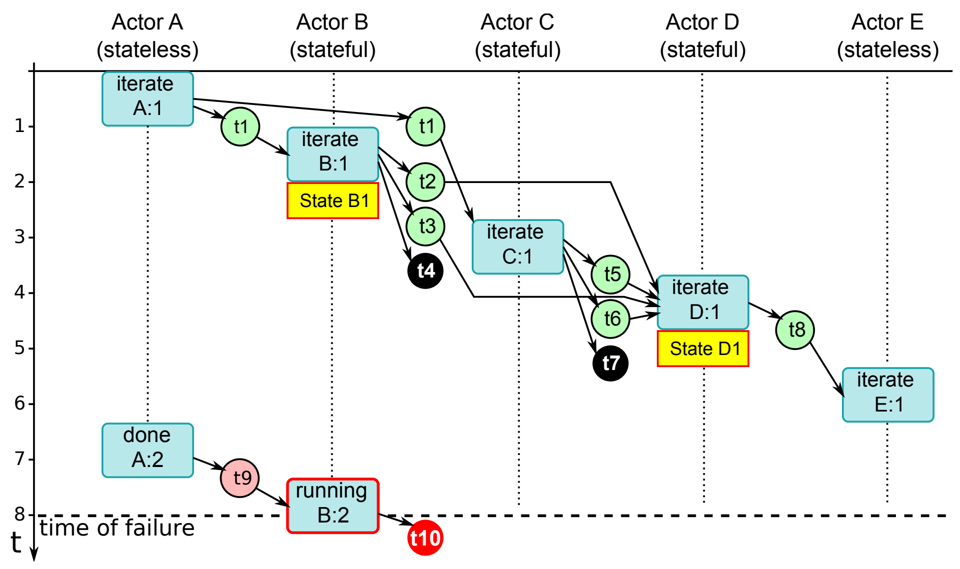


Figure 9.7: Workflow execution up to failure in B:2. The states of actors B and D are stored, but no checkpoint exists for C. Token t1 is only send once, but is duplicated by the link to both actors B and C. Tokens t4 and t7 are never read.

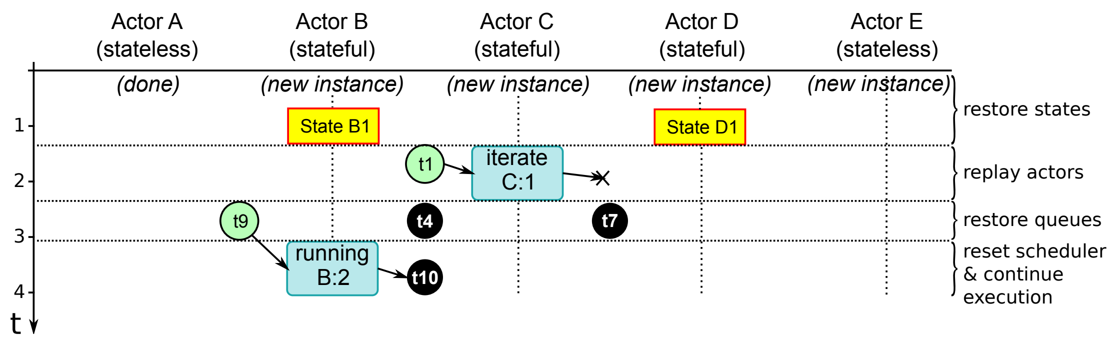


Figure 9.8: Individual stages to recover the sample workflow with checkpoint strategy.

Process Networks (PN). The example SDF workflow shown in Figure 9.5 can also be modeled using PN. Since actors under PN semantics have variable token production and consumption rates, these constraints cannot be leveraged to narrow the definition of a faulty invocation. Additionally, repeated invocations are not required for actors to perform their functions multiple times. For instance, actor D can be invoked only once, while actor B is invoked multiple times. All invocations in PN run concurrently, and tokens on a port have to be consumed after they are produced and only in the order they were produced. Finally, there are no defined firing limits. Many systems allow an actor to explicitly declare when it is done with all computations, which is recorded in provenance. Actors without that information are invoked until all actors in the workflow are waiting to receive data.

These characteristics have some implications on the recovery process. First, since all actors are executed in parallel, a crash could potentially affect all actors in a workflow. Thus, the recovery engine can safely restore actors in parallel. All actors are instantiated simultaneously at the beginning of the workflow run, in contrast to Figure 9.7. Long-running actor invocations reduce the availability of checkpoints and cause longer replay times. Finally, PN uses deadlock detection to define the end of a workflow, which makes it difficult to determine whether a particular actor is actually done (unless it explicitly says so) or just temporary deadlocked. Anything short of all actors being deadlocked by blocking reads (meaning the workflow is done) gives no useful information about which actors will exhibit future activity.

9.5 Evaluation

To evaluate the performance of the different recovery strategies, a prototype of the fault tolerance framework was implemented in Kepler [LAB$^+$06]. The current implementation adds fault tolerance to non-hierarchical SDF workflows. However, most of the existing implementation can easily be reused for other models of computation and can easily be extended to hierarchical models as well.

Implementation. The fault tolerance framework implements all features necessary for the checkpoint recovery strategy. Currently, checkpoints are saved after each execution of the complete schedule. One planned enhancement is to parametrize the time between checkpoint saving as either a number of actor iterations, or in terms of wall clock time. All the recovery logic was implemented in a separate workflow restore class and is instrumented from the director.

During a normal workflow execution, the system collects information about token firings and actor invocations. The provenance system of Daniel Crawl et al. [CA08] was extended to allow the storage of actor states and tokens. After each complete round of SDF invocations the director initiates the checkpointing process. The framework traverses the current level of the model hierarchy and captures the state of each stateful actor. An actor's state is represented by a serialization of selected fields of the Java class that implements the actor. There are two different mechanisms that can be chosen: (1) a blacklist mode that checks fields against a list of certain transient fields that should not be serialized, and (2) a whitelist mode that only saves fields explicitly annotated as state. The serialized state is then stored together with the last invocation id of the actor in the state relation of Kepler's provenance database. The serialization process is based on Java's object serialization and also includes selected fields of super classes.

During a recovery, the latest recorded checkpoint of an actor is restored. All stored actor fields are deserialized and overwrite the actor's fields. This leaves transient member fields intact and ensures that the restored actor is still properly integrated into its parent workflow.

Checkpoints are not necessarily stored after each invocation; but only after the last invocation in a schedule. Successful invocations completed after a checkpoint or where no checkpoint exists are replayed to restore the correct pre-failure state. For the replay, all corresponding serialized tokens are retrieved from the provenance database. Then the input queues of an actor are filled with tokens necessary for one invocation and the actor is fired. Subsequently, input and output queues are cleared again before the next invocation of an actor is replayed. The current implementation replays actors serially.

Next, all the queues are restored. For each actor, all tokens are retrieved that were written to an input port of the actor and not read by the actor itself before the fault. These tokens are then placed in the proper queues, preserving the original order.

Finally, the scheduling needs modifications to start at the proper point. This process is closely integrated with the normal execution behavior of the SDF director. The schedule is traversed in normal order, but all invocations are skipped until the failed invocation is reached. At this stage, the normal SDF execution of the schedule is resumed.

In order to replay tokens to actors and for restoring the queue content, the provenance recorder for Kepler presented in [Kep10] was changed. Instead of storing a string representation of a token, which may be lossy, the whole serialized token is stored in the provenance database. When using the standard token types, this increases the amount of data stored for each token only slightly. Actors can be explicitly marked as stateless using an annotation on the implementing Java class to speed up the recovery process. Stateless actors are not checkpointed, nor are their input tokens replayed.

Experimental Evaluation. The synthetic workflow shown in Figure 9.9 was created to simulate typical actor behavior in scientific workflows. This workflow was run to its completion to measure the running time for a successful execution. Then the execution was interrupted during the third invocation of actor C. After this, the workflow was loaded again and it resumed its execution using the three different strategies: re-execution from the beginning, Replay and Checkpoint.

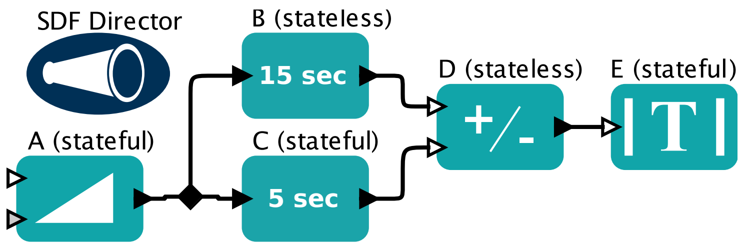


Figure 9.9: Synthetic SDF workflow. Actor A is a stateful actor generating a sequence of increasing numbers starting from 0. B is a stateless actor that has a running time of 15 seconds. C is stateful and needs 5 seconds for each invocation. D is a fast running stateless actor. E is a stateful "Display" actor.

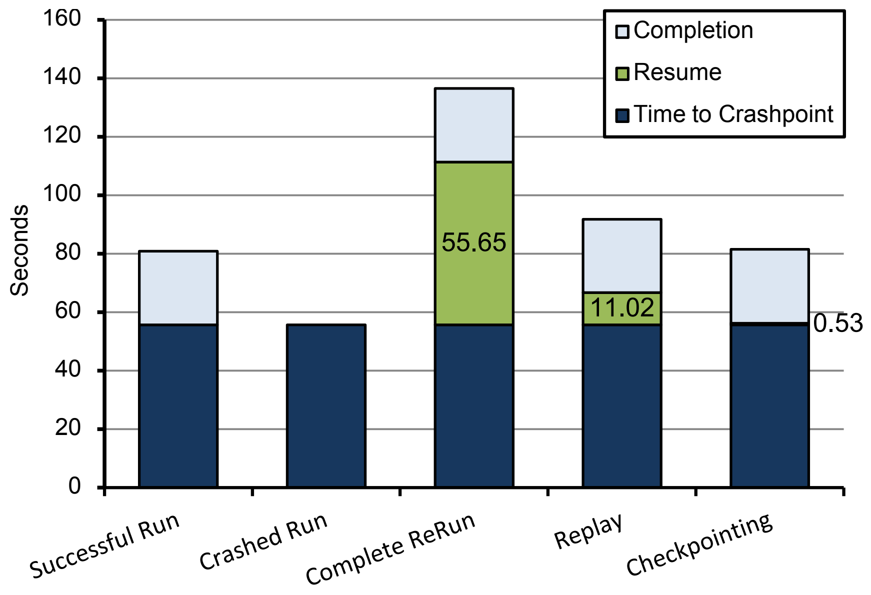


Figure 9.10: Performance evaluation of different recovery strategies.

The experiments were run three times for each strategy and the variation of all results was small compared to the overall running time. The experimental results are shown in Figure 9.10. The naive approach of re-running the whole workflow takes about 80 seconds, repeating about 55 seconds of execution time from before the crash. The replay strategy based on standard provenance already achieves a major improvement. The total time for this recovery strategy of approximately 12 seconds is dominated by replaying stateful actor C (5 seconds execution time) twice. After the recovery, workflow execution finishes in 25 seconds. This strategy reduced the cost for the actual recovery by 80% (55 seconds to 12 seconds). The checkpoint strategy also avoids the expensive replay. Although the deserialization process for the state is complex, the recovery time of this strategy is only about 0.6 seconds. This reduces the restore time by 99% compared to the naive strategy. Checkpointing is so efficient because it does not scale linearly with the number of tokens sent like the naive and replay strategies.

9.6 Summary

This chapter has shown how to use commonly available provenance information to recover a workflow run after the execution system has crashed. It gave examples that show the necessity of capturing the state of actors for faster recovery. This state information should be captured as a separate observation in provenance records. This chapter has also shown queries against the workflow model and a trace of a workflow execution that compute all important information for reinitializing the workflow.

Chapter 10

Conclusions & Outlook

This dissertation presented a current research in overlapping fields of scientific workflow systems, Datalog and provenance. A number of workflow systems were presented and their strengths and weaknesses analyzed. The functionality provided by these systems was compared with requirements from users. That comparison showed potential for simplification of the modeling of workflows as well improving the efficiency of the workflow execution.

The recording and analysis of provenance is important for users of scientific workflow systems. Additionally, this thesis presented approaches where provenance can also be used to improve the repeated execution of scientific workflows by the workflow system.

Datalog has been evaluated as a scientific workflow system in some related projects and is effective in modeling simple workflows [HRM+10] but it is less effective in modeling complex workflows. With the goal to support creating workflows in Datalog, a new approach to visualize, debug and profile Datalog programs was presented. Building upon that approach, an elegant way to see Datalog query evaluation as a game was presented and provides provenance for non-recursive Datalog$^\neg$ programs. In addition to using Datalog as a workflow system, it has been shown to be well suited for analyzing provenance.

The Monitoring workflow, Growing-Degree-Day workflow and Curation workflow indicated that support for a structured data model frequently simplifies design. Provenance traces of the GDD and Curation workflows revealed that the execution of those workflows is frequently not optimal. Motivated by the evaluation of the case studies, a new scientific workflow system was devised but due to the lack of time not fully implemented. A prototype is under development and evaluation in the Filtered-Push project. To improve fault tolerance of workflow systems a new method was developed that demonstrates the use of provenance data by analyzing provenance for a fast recovery of the workflow execution.

This thesis has demonstrated some use cases of provenance in the field of scientific workflow systems but there remain many research opportunities: The provenance of data items can be analyzed while executing workflows in order to mine actor properties such as token production and consumption rates or actual data dependencies. This data could be used to improve the scheduling of the running workflows. The application of provenance games to recursive Datalog$^\neg$ programs currently requires a separate step to determine the solution for drawn positions. An alternative game construction can yield the correct solution directly. Another open problem is how to describe provenance for disjunctive Datalog or answer set programs.

Bibliography

[ABC⁺03] S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *SIGMOD*, pp. 527–538, 2003. 152, 153

[ABC⁺10] U. Acar, P. Buneman, J. Cheney, J. Van den Bussche, N. Kwasnikowska, and S. Vansummeren. A graph model of data and workflow provenance. In *TaPP*, 2010. 44

[ACGG⁺02] I. Avila-Campillo, T. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML toolkit for scalable XML stream processing. *Proceedings of PLANX, October*, 2002. 123

[ADT11a] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pp. 153–164. ACM, 2011. 64

[ADT11b] Y. Amsterdamer, D. Deutch, and V. Tannen. On the Limitations of Provenance for Queries With Difference. In *Workshop on Theory and Practice of Provenance (TaPP)*, Heraklion, Crete, 2011. 64

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 31

[Akk13] Akka Toolkit. Typesafe Inc., 2013. https://http://akka.io/. 154

[AMC⁺09] P. Alvaro, W. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. C. Sears. Dedalus: Datalog in Time and Space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009. 31

[AN05] P. Amnuaykanjanasin and N. Nupairoj. The BPEL orchestrating framework for secured grid services. *Information Technology: Coding and Computing (ITCC)*, 1, 2005. 123

[ASK⁺12] H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors. *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*, volume 9 of *Procedia Computer Science*. Elsevier, 2012. 189, 190

[Ayy13] S. Ayyub. *Dynamic Process Nets*. PhD thesis, Monash University. Faculty of Information Technology. School of Information Technology, 2013. 3

[BAJ+10] D. Barseghian, I. Altintas, M. Jones, D. Crawl, N. Potter, J. Gallagher, P. Cornillon, M. Schildhauer, E. Borer, and E. Seabloom. Workflows and extensions to the Kepler scientific workflow system to support environmental sensor data access and analysis. *Ecological Informatics*, 5(1):42–50, 2010. 108

[BBMS05] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD*, 2005. 152

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Intl. Conf. on Functional Programming (ICFP)*, pp. 51–63, New York, NY, USA, 2003. 129, 152

[BCS+05] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. J. Crossno, C. T. Silva, and J. Freire. VisTrails: Enabling Interactive Multiple-View Visualizations. *Visualization Conference, IEEE*, 0:18, 2005. 23, 25, 164

[BDFZ01] S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 1(5):497–538, 2001. 85, 86, 88, 90, 91

[BE69] G. L. Baskerville and P. Emin. Rapid Estimation of Heat Accumulation from Maximum and Minimum Temperatures. *Ecology*, 50(3):pp. 514–517, 1969. 109

[BKT01] P. Buneman, S. Khanna, and W. C. Tan. Why and Where: A Characterization of Data Provenance. In J. V. den Bussche and V. Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pp. 316–330. Springer, 2001. 44, 63, 64

[BML+06] S. Bowers, T. M. McPhillips, B. Ludäscher, S. Cohen, and S. B. Davidson. A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In L. Moreau and I. T. Foster, editors, *IPAW*, volume 4145 of *Lecture Notes in Computer Science*, pp. 133–147. Springer, 2006. 33, 160

[Bor07] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. Apache Software Foundation, 2007. 135

[BP12] P. Barceló and R. Pichler, editors. *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012. 31, 190

[BPM05] *Intl. Workshop on Web Service Choreography and Orchestration for BPM*, Nancy, France, September 2005. 123

[BR88] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Readings in database systems*, pp. 507–555. Morgan Kaufmann Publishers Inc., 1988. 48

[BSHW06] O. Benjelloun, A. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, pp. 953–964, 2006. 64, 84

[CA08] D. Crawl and I. Altintas. A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows. In J. Freire, D. Koop, and L. Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *LNCS*, pp. 152–159. Springer Berlin / Heidelberg, 2008. 162, 166, 172, 177

[CAA07] J. Cheney, A. Ahmed, and U. Acar. Provenance as Dependency Analysis. In *DBPL*, LNCS 4797, pp. 138–152, Vienna, Austria, 2007. 44

[CAWK08] E. Ceyhan, G. Allen, C. White, and T. Kosar. A grid-enabled workflow system for reservoir uncertainty analysis. In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pp. 45–52. ACM, 2008. 2

[CCA$^+$10] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pp. 21–21, Berkeley, CA, USA, 2010. USENIX Association. 29

[CCD$^+$03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003. 152

[CCT09] J. Cheney, L. Chiticariu, and W. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009. 44, 63

[CDGLV03] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Reasoning on regular path queries. *ACM SIGMOD Record*, 32(4):83–92, 2003. 37, 44

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pp. 379–390, 2000. 152

[CDZ06] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, 2006. 152

[CGRSP08] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. A theoretical framework for the declarative debugging of datalog programs. *Semantics in Data and Knowledge Bases*, pp. 143–159, 2008. 61

[Che10] J. Cheney. Causality and the semantics of provenance. *Arxiv preprint arXiv:1004.3241*, 2010. 44

[Cim12] California Irrigation Management Information System, Office of Water Use Efficiency, California Department of Water Resources. `http://wwwcimis.water.ca.gov/cimis/`, 2012. 109

[CLRV09] F. Calimeri, N. Leone, F. Ricca, and P. Veltri. A visual tracer for DLV. In *Proceedings of the 2nd International Workshop on Software Engineering for Answer Set Programming (SEA 2009)*, pp. 79–93, 2009. 61

[Com12] Comet Kepler Suite. `https://code.kepler-project.org/code/kepler/trunk/modules/comet/`, 2012. 112

[CT06] L. Chiticariu and W. C. Tan. Debugging Schema Mappings with Routes. In U. Dayal, K.-Y. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *VLDB*, pp. 79–90. ACM, 2006. 61

[CVDK$^+$12] V. Cuevas-Vicenttín, S. C. Dey, S. Köhler, S. Riddle, and B. Ludäscher. Scientific Workflows and Provenance: Introduction and Research Opportunities. *Datenbank-Spektrum*, 12(3):193–203, 2012. 33

[CWW00] Y. Cui, J. Widom, and J. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000. 63, 64

[DBG$^+$04] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *Grid Computing*, volume 3165 of *LNCS*, pp. 131–140. Springer Berlin / Heidelberg, 2004. 8, 163

[DCVK$^+$13] S. C. Dey, V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. On implementing provenance-aware regular path queries with relational query engines. In G. Guerrini, editor, *EDBT/ICDT Workshops*, pp. 214–223. ACM, 2013. 33

[DEGV01] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001. 44

[Den80] J. B. Dennis. Data Flow Supercomputers. *Computer*, 13:48–56, November 1980. 11

[DF08] S. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1345–1350. ACM, 2008. 36

[DG08] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 123, 125, 152

[DGST09] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009. 7, 30

[DHL$^+$11] L. Dou, J. Hanken, B. Ludäscher, J. Macklin, T. M. McPhillips, P. Morris, R. Morris, and Z. Wang. Building specimen-data curation pipelines using Kepler workflow technology in a Filtered-Push network. In *26th Annual SPHNC Meeting, San Francisco*, 2011. 5

[Dij81] E. W. Dijkstra. Hamming's exercise in SASL, 1981. EWD-792. 42

[DKBL12] S. Dey, S. Köhler, S. Bowers, and B. Ludäscher. Datalog as a lingua franca for provenance querying and reasoning. In *Proceedings of the 4th USENIX conference on Theory and Practice of Provenance*, TaPP'12, pp. 13–13, Berkeley, CA, USA, 2012. USENIX Association. 2, 4, 34

[DNT89] L. Drabent and S. Nadjm-Tehrani. Algorithmic debugging with assertions. In *Meta-programming in logic programming*. Citeseer, 1989. 60

[DZM$^+$11] L. Dou, D. Zinn, T. M. McPhillips, S. Köhler, S. Riddle, S. Bowers, and B. Ludäscher. Scientific workflow design 2.0: Demonstrating streaming data collections in Kepler. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pp. 1296–1299. IEEE Computer Society, 2011. 16, 28, 100, 112, 163

[Fag07] J. Fagan. Mashing up Multiple Web Feeds Using Yahoo! Pipes. *Computers in Libraries*, 27(10):8, 2007. 123

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. *JLP*, 2(4):295–312, 1985. 85

[FJM$^+$07] M. F. Fernández, T. Jim, K. Morton, N. Onose, and J. Siméon. Highly distributed XQuery with DXQ. In *SIGMOD*, pp. 1159–1161, 2007. 152

[FKL97] J. Flum, M. Kubierschky, and B. Ludäscher. Total and Partial Well-Founded Datalog Coincide. In *ICDT*, pp. 113–124, 1997. 74, 97

[FKL00] J. Flum, M. Kubierschky, and B. Ludäscher. Games and total Datalog¬ queries. *Theoretical Computer Science*, 239(2):257–276, 2000. 74, 97

[FL08] T. Feng and E. Lee. Real-Time Distributed Discrete-Event Execution with Fault Tolerance. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pp. 205 –214, April 2008. 162, 172

[Flu00] J. Flum. Games, Kernels, and Antitone Operations. *Order*, 17(1):61–73, 2000. 65

[FPD$^+$05] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. *International Workshop on Grid Computing*, pp. 122–131, 2005. 153

[FPD$^+$07] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Development and Grid Computing Environment for Scientific Workflows. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science*, pp. 450–471. Springer London, 2007. 8, 30

[Fre] J. Frey. Condor DAGMan: Handling inter-job dependencies. 163

[GBA10] B. Glavic, D. M. BOHLEN, and D. G. ALONSO. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, PhD thesis, University of Zurich, 2010. 3

[GDR07] C. A. Goble and D. C. De Roure. myExperiment: social networking for workflow-using e-scientists. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, WORKS '07, pp. 1–2, New York, NY, USA, 2007. 7

[Gen01] W. Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. In *First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001. Proceedings*, pp. 35–36, 2001. 146

[GGM$^+$04] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *TODS*, 29(4):752–788, 2004. 152

[GGV02] G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: A deductive query language with linear time model checking. *ACM Transactions on Computational Logic (TOCL)*, 3(1):42–79, 2002. 45

[GIT11] T. Green, Z. Ives, and V. Tannen. Reconcilable differences. *Theory of Computing Systems*, 49(2):460–488, 2011. 64

[GKIT07] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, editors, *VLDB*, pp. 675–686. ACM, 2007. 51, 64

[GKT07] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In L. Libkin, editor, *PODS*, pp. 31–40. ACM, 2007. 44, 51, 61, 63, 69, 76, 85

[GP10] F. Geerts and A. Poggi. On database query languages for k-relations. *Journal of Applied Logic*, 8(2):173–185, 2010. 64

[Grä11] E. Grädel. Back and Forth Between Logic and Games. In *Lectures in Game Theory for Computer Scientists*, chapter 4, pp. 99–145. Cambridge University Press, 2011. 97

[Gre11] T. Green. Containment of conjunctive queries on annotated relations. *Theory of Computing Systems*, 49(2):429–459, 2011. 63, 64

[Gul10] S. S. Gulati. Computing Sliding Window Aggregates over Data Streams in a Scientific Workflow System. Master's thesis, Dept. of Computer Science, University of California, Davis, June 2010. 110

[Had] Hadoop. `http://hadoop.apache.org/`. 126, 146

[HC07] I. Hernandez and M. Cole. Reliable DAG scheduling on grids with rewinding and migration. In *Proceedings of the first international conference on Networks for grid applications*, GridNets '07, pp. 3:1–3:8. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007. 10, 162, 163

[HCDN08] J. Huang, T. Chen, A. Doan, and J. Naughton. On the provenance of non-answers to queries over extracted data. In *VLDB*, 2008. 3

[Hem88] D. Hemmendinger. The "Hamming problem" in Prolog. *ACM SIGPLAN Notices*, 23(4):81–86, 1988. 42

[HGL11] S. Huang, T. Green, and B. Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD*, pp. 1213–1216, 2011. 64

[HHMW07] T. Härder, M. Haustein, C. Mathis, and M. Wagner. Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60(1):126–149, 2007. 136, 137

[Hig09] S. Higgins. PREMIS Data Dictionary for Preservation Metadata. *Digital Curation Centre*, March 2009. 33

[Hin96] J. Hintikka. *The Principles of Mathematics Revisited*. Cambridge University Press, 1996. 97

[HKS$^+$07] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. A Formal Model of Dataflow Repositories. In *DILS*, pp. 105–121, 2007. 123

[HKS$^+$08] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. DFL: A dataflow language based on Petri nets and nested relational calculus. *Information Systems*, 33(3):261 – 284, 2008. 25, 28, 100

[Hod13] W. Hodges. Logic and Games. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. `http://plato.stanford.edu/entries/logic-games/`, March 2013. 97

[HRM$^+$10] A. Hartman, S. Riddle, T. McPhillips, B. Ludascher, and J. Eisen. Introducing W.A.T.E.R.S.: a Workflow for the Alignment, Taxonomy, and Ecology of Ribosomal Sequences. *BMC Bioinformatics*, 11(1):317, 2010. 2, 160, 162, 163, 181

[HW02] P. Hitzler and M. Wendt. The well-founded semantics is a stratified Fitting semantics. *KI 2002: Advances in Artificial Intelligence*, pp. 57–59, 2002. 91

[Ike12] R. Ikeda. *Provenance in Data-Oriented Workflows*. PhD thesis, Stanford InfoLab, 2012. 3

[Ima] Image-Magick. http://www.imagemagick.org. 132

[Inf13] InforSense Suite. ID Business Solutions Ltd., 2013. https://www.idbs.com/products-and-services/inforsense-suite/. 2

[ITA$^+$08] Z. Ivezic, J. Tyson, E. Acosta, R. Allsman, S. Anderson, J. Andrew, R. Angel, T. Axelrod, J. Barr, A. Becker, et al. LSST: from science drivers to reference design and anticipated data products. *arXiv preprint arXiv:0805.2366*, 2008. 1

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pp. 471–475. North-Holland, New York, NY, 1974. 13, 14, 35, 153, 164, 165

[Kep10] Getting Started with the Kepler Provenance Module. https://code.kepler-project.org/code/kepler/releases/release-branches/provenance-2.1/docs/provenance.pdf, August 2010. 178

[KG12] G. Karvounarakis and T. J. Green. Semiring-annotated data: queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012. viii, 3, 63, 70, 76, 77, 79, 80, 81

[KGC$^+$12] S. Köhler, S. Gulati, G. Cao, Q. Hart, and B. Ludäscher. Sliding Window Calculations on Streaming Data using the Kepler Scientific Workflow System. In Ali et al. [ASK$^+$12], pp. 1639–1646. 5, 108

[KIT10] G. Karvounarakis, Z. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pp. 951–962, 2010. 3

[KL04] T. Kosar and M. Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *ICDCS*, pp. 342–349. IEEE Computer Society, 2004. 10, 164

[KLS12] S. Köhler, B. Ludäscher, and Y. Smaragdakis. Declarative Datalog Debugging for Mere Mortals. In Barceló and Pichler [BP12], pp. 111–122. 4, 48

[KLZ13] S. Köhler, B. Ludäscher, and D. Zinn. First-Order Provenance Games. In V. Tannen, L. Wong, L. Libkin, W. Fan, W.-C. Tan, and M. Fourman, editors, *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *Lecture Notes in Computer Science*, pp. 382–399. Springer Berlin Heidelberg, 2013. 4, 63

[KMVdB10] N. Kwasnikowska, L. Moreau, and J. Van den Bussche. A formal account of the open provenance model. Technical Report 21819, University of Southampton, December 2010. 34, 37

[KRZ$^+$11] S. Köhler, S. Riddle, D. Zinn, T. M. McPhillips, and B. Ludäscher. Improving Workflow Fault Tolerance through Provenance-Based Recovery. In J. B. Cushing, J. C. French, and S. Bowers, editors, *SSDBM*, volume 6809 of *Lecture Notes in Computer Science*, pp. 207–224. Springer, 2011. 5, 159

[KSFL12] S. Köhler, P. Seitzer, M. T. Facciotti, and B. Ludäscher. Improved Motif Detection in Large Sequence Sets with Random Sampling in a Kepler workflow. In Ali et al. [ASK$^+$12], page 1999. 5, 118

[KSSS04a] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *VLDB*, 2004. 152

[KSSS04b] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. *VLDB*, pp. 1309–1312, 2004. 152

[Kub95] M. Kubierschky. Remisfreie Spiele, Fixpunktlogiken und Normalformen. Master's thesis, Universität Freiburg, Germany, 1995. 97

[Küh13] F. Kühnlenz. *Design und Management von Experimentier-Workflows*. PhD thesis, Humboldt-Universität zu Berlin, 2013. 3

[Kun91] K. Kunen. Declarative Semantics of Logic Programming. *Bulletin of the EATCS*, 44:147–167, 1991. 48

[LAB$^+$06] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, 2006. 123, 152, 177

[LAB$^+$09] B. Ludäscher, I. Altintas, S. Bowers, J. Cummings, T. Critchlow, E. Deelman, D. Roure, J. Freire, C. Goble, M. Jones, et al. Scientific process automation and workflow management. In A. Shoshani and D. Rotem, editors, *Chapter 13 in Scientific Data Management: Challenges, Existing Technology, and Deployment*, pp. 476–508, Boca Raton, FL, 2009. Chapman & Hall/CRC Press. 6

[LL78] P. Lorenzen and K. Lorenz. *Dialogische Logik*. Wissenschaftliche Buchgesellschaft,, Darmstadt, 1978. 71

[LLCF10] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi. Prospective and retrospective provenance collection in scientific workflow environments. In *Services Computing (SCC)*, 2010. 36

[LLM98] G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. *Transactions and Change in Logic Databases*, pp. 69–106, 1998. 45, 51, 53, 62

[LM87a] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, Sept. 1987. 11

[LM87b] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, January 1987. 163, 166

[LM08] E. Lee and E. Matsikoudis. The semantics of dataflow with firing. *From Semantics to Computer Science: Essays in memory of Gilles Kahn. Cambridge University Press, Cambridge*, 2008. 163, 164, 166

[LP95] E. A. Lee and T. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995. 35, 152

[LPA$^+$08] B. Ludäscher, N. Podhorszki, I. Altintas, S. Bowers, and T. McPhillips. From computation models to models of provenance: the RWS approach. *Concurr. Comput. : Pract. Exper.*, 20:507–518, April 2008. 172

[LPF$^+$06] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006. 51, 60

[Lud98] B. Ludäscher. *Integration of active and deductive database rules*. PhD thesis, Albert-Ludwigs Universität, Freiburg, Germany, 1998. 51, 53, 54, 62

[MB05] T. M. McPhillips and S. Bowers. An Approach for Pipelining Nested Collections in Scientific Workflows. *SIGMOD Record*, 34(3):12–17, 2005. 152

[MBZL08] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific Workflow Automation for Mere Mortals. *Future Generation Computer Systems*, 2008. in press. 123

[MBZL09] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541 – 551, 2009. 6, 7

[MCF$^+$10] L. Moreau, B. Clifford, J. Freire, Y. Gil, P. Groth, J. Futrelle, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, Y. Simmhan, E. Stephan, and J. V. den Bussche. The Open Provenance Model - core specification (v1.1). *Future Generation Computer Systems*, 2010. 33, 160

[MCF+11] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, et al. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, 2011. 34

[MFF+08] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson. The Open Provenance Model: An Overview. In *Provenance and Annotation of Data and Processes*, volume 5272 of *LNCS*, pp. 323–326. Springer Berlin / Heidelberg, 2008. 166

[MGH+10] A. Meliou, W. Gatterbauer, J. Halpern, C. Koch, K. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull*, 33(3):59–67, 2010. 44

[MGH+13] P. Morris, E. Gilbert, J. Hanken, M. Kelly, S. Köhler, D. Lowery, B. Ludäscher, J. Macklin, R. Morris, and T. Song. Expanding the scope of collections databases without schema modifications: Using annotations, rules, and semantic web technologies to add a open world layer to natural science collections data. In *Program book and abstracts for the 28th annual meeting of the Society for the Preservation of Natural History Collections (SPNHC)*, page 33, 2013. 5

[MGS11] A. Meliou, W. Gatterbauer, and D. Suciu. Bringing Provenance to its Full Potential using Causal Reasoning. *Theory and Practice of Provenance (TaPP)*, 2011. 3

[MHB+10] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. SecureBlox: customizable secure distributed data processing. In *SIGMOD*, pp. 723–734, 2010. 51, 60

[MM10] T. McPhillips and S. McPhillips. RestFlow System and Tutorial. https://sites.google.com/site/restflowdocs/, September 2010. 19, 21, 164

[Mor94] J. P. Morrison. *Flow-Based Programming – A New Approach to Application Development*. Van Nostrand Reinhold, 1994. 152

[MSRO+10] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In *Scientific and Statistical Database Management*, pp. 471–481. Springer, 2010. 159

[OGA+06] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice & Experience*, 18(10):1067–1100, August 2006. 123, 152

[OOP+04] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 903–908. ACM New York, NY, USA, 2004. 136

[OPT11] J. Oetsch, J. Pührer, and H. Tompits. Stepping through an answer-set program. *Logic Programming and Nonmonotonic Reasoning*, pp. 134–147, 2011. 61

[PA06] C. Pautasso and G. Alonso. Parallel Computing Patterns for Grid Workflows. In *Proceedings of the the Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2006. 153

[Pip13] Pipeline Pilot. Accelrys Inc., 2013. http://accelrys.com/products/pipeline-pilot. 2, 30

[PLK07] N. Podhorszki, B. Ludaescher, and S. A. Klasky. Workflow automation for processing plasma fusion simulation data. In *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, WORKS '07, pp. 35–44, New York, NY, USA, 2007. ACM. 2, 102, 132, 162, 163

[PRT$^+$07] S. Perri, F. Ricca, G. Terracina, D. Cianni, and P. Veltri. An integrated graphic tool for developing and testing DLV programs. In *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, pp. 86–100, 2007. 61

[Pto14] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. 1

[QF07] J. Qin and T. Fahringer. Advanced data flow support for scientific grid workflow applications. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC)*, pp. 1–12. ACM, 2007. 153

[RBHS04] C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. *Workshop on Information Integration on the Web*, pp. 116–121, 2004. 152

[REA12] Real-time Environment For Analytical Processing. `http://reap.ecoinformatics.org/`, 2012. 108

[roc] ROCKS Clusters. `http://www.rocksclusters.org/`. 146

[RSS$^+$97] P. Rao, K. Sagonas, T. Swift, D. Warren, and J. Freire. XSB: A system for efficiently computing well-founded semantics. In J. Dix, U. Furbach, and A. Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pp. 430–440. Springer Berlin / Heidelberg, 1997. 24

[Sha82] E. Shapiro. Algorithmic program debugging. *Dissertation Abstracts International Part B: Science and Engineering,*, 43(5), 1982. 60

[Sof12] `http://www.bme.ucdavis.edu/facciotti/resources_data/software/`, 2012. 118

[SPG05] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34:31–36, September 2005. 3, 33

[SW01] U. Schwalbe and P. Walker. Zermelo and the early history of game theory. *Games and Economic Behavior*, 34(1):123–137, 2001. 64

[SWLF12] P. Seitzer, E. Wilbanks, D. Larsen, and M. Facciotti. A Monte Carlo-based framework enhances the discovery and interpretation of regulatory sequence motifs. *BMC Bioinformatics*, 13(1):1–16, 2012. 118

[TB93] G. Tobermann and C. Beckstein. What's in a Trace: The Box Model Revisited. In P. Fritszon, editor, *AADEBUG*, volume 749 of *Lecture Notes in Computer Science*, pp. 171–187. Springer, 1993. 49

[TMG$^+$07] D. Turi, P. Missier, C. Goble, D. D. Roure, and T. Oinn. Taverna Workflows: Syntax and Semantics. In *Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pp. 441–448, Washington, DC, USA, 2007. IEEE Computer Society. 18

[TMG$^+$08] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *e-Science and Grid Computing, IEEE International Conference on*, pp. 441–448. IEEE, 2008. 163

[TRP$^+$04] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. *SIGMOD*, pp. 479–490, 2004. 123

[VG93] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993. 68

[VGRS91] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3):619–649, 1991. 67, 86, 88, 90

[vN28] J. v. Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928. 64

[W3C] W3C. Provenance Working Group. `http://www.w3.org/2011/prov/`. accessed 4/9/2012. 36

[WCA09] J. Wang, D. Crawl, and I. Altintas. Kepler + Hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pp. 12:1–12:8, New York, NY, USA, 2009. ACM. 118

[WDK$^+$09] Z. Wang, H. Dong, M. Kelly, J. A. Macklin, P. J. Morris, and R. A. Morris. Filtered-Push: a Map-Reduce platform for collaborative taxonomic data management. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 3, pp. 731–735. IEEE, 2009. 5

[WLF$^+$09] L. Wang, S. Lu, X. Fei, A. Chebotko, H. V. Bryant, and J. L. Ram. Atomicity and provenance support for pipelined scientific workflows. *Future Generation Computer Systems*, 25(5):568 – 576, 2009. 163

[WM07] N. Walsh and A. Milowski. XProc: An XML Pipeline Language. *W3C Working Draft, April*, 2007. 123

[WSTL10] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *CoRR*, abs/1011.5332, 2010. 60

[ZBKL10] D. Zinn, S. Bowers, S. Köhler, and B. Ludäscher. Parallelizing XML data-streaming workflows via MapReduce. *J. Comput. Syst. Sci.*, 76(6):447–463, 2010. 5, 121

[ZBML09a] D. Zinn, S. Bowers, T. McPhillips, and B. Ludäscher. Scientific workflow design with data assembly lines. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, WORKS '09, pp. 14:1–14:10, New York, NY, USA, 2009. ACM. 7, 8

[ZBML09b] D. Zinn, S. Bowers, T. McPhillips, and B. Ludäscher. X-CSR: Dataflow Optimization for Distributed XML Process Pipelines. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pp. 577–580, Washington, DC, USA, 2009. IEEE Computer Society. 18, 123, 154

[ZDF+05] Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *SIGMOD Rec.*, 34(3):37–43, 2005. 132

[Zel02] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10. ACM, 2002. 50

[Zer13] E. Zermelo. Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Fifth Intl. Congress of Mathematicians*, volume 2, pp. 501–504. Cambridge University Press, 1913. 64

[ZGL12] D. Zinn, T. J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In A. Deutsch, editor, *ICDT*, pp. 99–113. ACM, 2012. 32, 69

[ZGW+83] F. G. Zalom, P. B. Goodell, L. T. Wilson, W. W. Barnett, and W. J. Bentley. Degree-Days: The Calculation and Use of Heat Units in Pest Management. *University of California Division of Agriculture and Natural Resources Leaflet*, 21373, 1983. 109

[ZHC+07] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *2007 IEEE Congress on Services*, pp. 199–206. IEEE, 2007. 162

[Zin10] D. Zinn. *Modeling and Optimization of Scientific Workflows.* PhD thesis, UC Davis, Davis, California, 2010. 3, 154