AGH University of Science and Technology
in Kraków

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Institute of Computer Science

Mikołaj Baranowski

Optimization of application execution in virtual laboratory

Thesis

Major: Computer Science
Specialization: Distributed Systems and Computer Networks

Supervisor:
PhD Marian Bubak

Consultancy:
PhD Maciej Malawski

Album id: 127088

Kraków 2011

Oświadczenie autora

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Mikołaj Baranowski

Abstract

The Virolab project provides an environment to develop and execute grid applications. Applications are written in Ruby programming language and grid infrastructure is accessible by grid object instances. Existed optimization model does not realize optimization based on application structure and in detail, dependencies between grid object methods.

This thesis discusses opportunities of optimization based on workflow scheduling, goes through the process of building workflow for Virolab applications and gives a view for scheduling techniques.

In order to build a workflow scheduling system, one should encounte issues of analyzing Ruby sources, resolving variables and methods dependencies, building workflow representation and providing workflow scheduling algorithms which can deal with proposed representation.

There were developed solutions to these problems and they were proved by implementing complex grid applications as CyberShake, Epigenomics and Montage. Evaluation is enriched by representing workflow control flow patterns.

This thesis is organized as follows:

Chapter 1 gives an introduction to the problem, describes existing Virolab environment and defines the goals for the thesis. Chapter 2 describes how the workflow scheduling problem in grid applications is handled in other works, what issues can be encountered and how to design an application to enable cooperation with existing tools. In chapter 3 the whole process of GridSpace application source analysis is presented with explanation how particular issues were solved. Chapter 4 focuses on the technical aspects of the developed application, introduces its architecture, describes usage of external tools and answers the question how to invoke desired actions. Chapter 5 is an attempt to prove the concept by creating workflows for non-trivial Ruby scripts, using typical workflow constructs and by recreating existing well-known workflow application as hypothetical GridSpace applications. Chapter 6 points connections between developed solutions and existing workflow scheduling systems. The last chapter 7 summarizes the work by answering questions about which aspects of the problem gives expected results, which brings problems, which trends promise good outcomes and how to modify starting assumptions to reach better results.

Keywords: Ruby, Grid Computing, Optimization, ViroLab, Workflow scheduling, Analysis of script applications

Acknowledgements

I wish to express appreciation to my supervisor - Marian Bubak for priceless comments and motivation and to my irreplaceable adviser - Maciej Malawski for his suggestions and support.

This work is related with the Mapper project which receives funding from the EC's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° RI-261507.

Contents

Acronyms

AGWL Abstract Grid Workflow Language. 30, 31, 36

CPN Coloured Petri-Net. 27–30

DAG Directed Acyclic Graph. 18, 33–35, 94, 95

DS Dominant Sequence. 34, 97

DSC Dominant Sequence Clustering. 34, 97

GO Grid Object. 9–11, 13, 15

GS Grid Scheduler. 32

GSEngine Grid Space Engine. 9–11, 100, 103

HEFT Heterogoneous Earliest-Finish-Time. 33, 34, 95, 96

PSA Peak Spectral Acceleration. 24, 25

SGT Strain Green Tensor. 24, 25

YAWL Yet Another Workflow Language. 31, 36

List of Figures

List of Tables

CHAPTER 1

Motivation and Objectives

This chapter describes application environment in which optimization should be performed - The Virolab Virtual Laboratory. It also introduces existing optimization solution in terms of its limitations and also proposes an approach which exceed these limitations.

1.1 Virolab environment

The target environment - ViroLab Virtual Laboratory[2, 3] runtime (also called Grid Space Engine (GSEngine)[4]) is a part of ViroLab project. Official site of ViroLab project [5] describes virtual laboratory as a "set of integrated components that, used together, form a distributed and collaborative space for science. Multiple, geographically-dispersed laboratories and institutes use the virtual laboratory to plan, and perform experiments as well as share their results." Term experiment, used in this context, means a process of combining data and computations in order to obtain new knowledge.

The main goal of the ViroLab is to provide a virtual labolatory for infactous diseases but technical solutions and concepts are universal enough to cover many domains of science.

1.1.1 Grid environment abstraction

To provide grid environment capabilities and to build interfaces between different technologies, in Virolab, there is a three level Grid Object (GO) abstraction[6]. The top level includes GO classes, these are abstract entities which define operations. One GO class

may have many implementations which are build on various technologies and run on different environments but by dint of GO class, their operations are consistent. The third level of abstraction is GO instance which is in the same relation with GO implementation as GO implementation is with GO class - one GO implementation can have many GO many GO instances, running on different resources or levels of performance.

1.1.2 Virolab experiments

GSEngine provides capabilities offered by Virtual Labolatory through APIs and libraries which are accessible from Ruby[7] scripts - also called experiments. Experiment developer is allowed to instantiate GOs and to perform operations on them.

The top level abstraction of grid environment is realized by a routine which takes GO class name as an argument. The result represents GO instance which provides all operations previously defined in GO class.

The most important fact for this master thesis is that GO operations can be invoked both synchronously and asynchronously. A synchronous operation blocks script execution until remote procedure is finished. An asynchronous operations does not block script execution but returns an operation handler which represents state of remote operation. Then, invoker keeps executing process during which further operations can be called (including other asynchronous operations) till the result of asynchronous operation is not required. The result of asynchronous operation can be obtained by invoking `get_result` method on operation handler.

Instancing a GO looks as follows. `GObj` is a module which provides method `create`, it creates GO identified by string `'cyfronet.gridspace.gem.weka.WekaGem'`. It waits until grid operation ends:

```
retriever =  GObj.create('cyfronet.gridspace.gem.weka.WekaGem')
```

A synchronous grid operation is an invocation of a method on GO whose name does not start with `async_`:

```
retriever.loadDataFromDatabase(database, query, user, password)
```

An asynchronous grid operation is an invocation of a method on GO whose name starts with `async_`. Operation returns grid operation handler (in this case `classificationPercentage`) and does not wait for grid operation result:

```
classificationPercentage = retriever.async_compare(testA,
   prediction.get_result, attributeName)
```

Result request on grid operation handler is shown on a following listing, `get_result` method returns grid operation result which is represented by operation handler. If a grid operation is not finished, the script waits until the result is obtained.

```
puts 'Prediction_quality:' + classificationPercentage.get_result.to_s
```

An example of a real Virolab experiment is shown in figure 1.1.

1.1.3 Virolab Laboratory runtime

Previously introduced GSEngine[4] consists of main two parts:

Grid Operation Invoker. This part includes Ruby language interpreter - particularly JRuby implementation. It is also responsible for optimization and invocation of GO. It corresponds with *Computation Access* library in figure 1.2 which is responsible for remote processing.

Data Access Client. Ruby library which allows to access data sources available in the Virtual Laboratory. It is shown in figure 1.2 as a *Data Access Client* library which is used to relay data from *Data Access Service*.

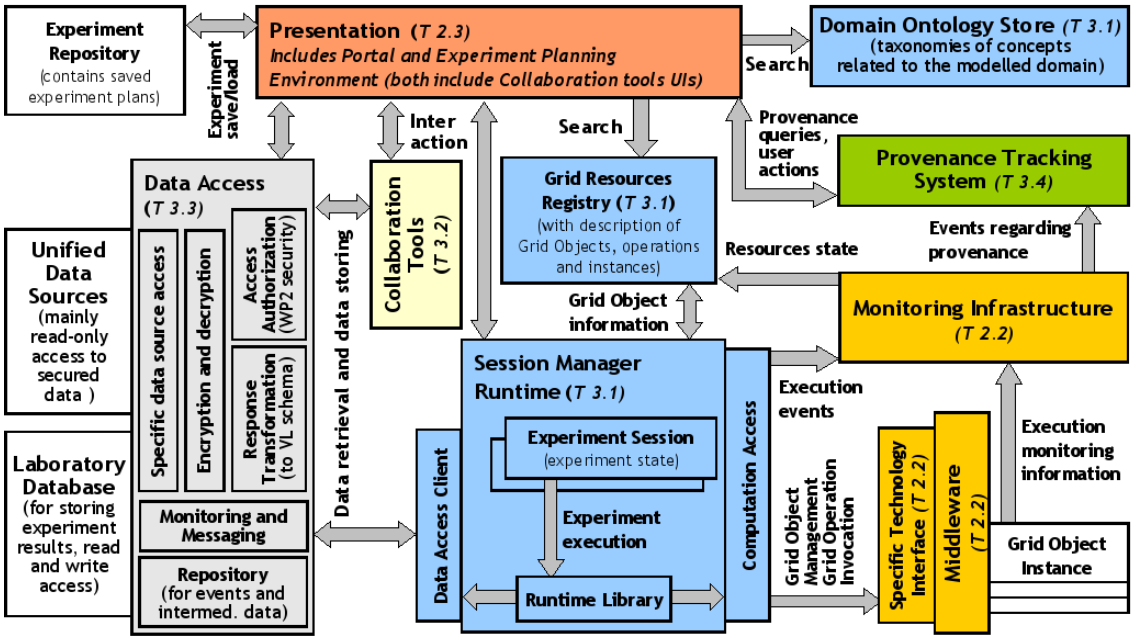


Figure 1.2: Virolab general architecture.

Optimization

Virolab optimizer is called GrAppO[8, 9]. It is responsible for selecting optimal Grid Object Instances to invoke operation from the particular Grid Object Class implementations. Optimization in Virolab environment bases on the information gathered by a registry called Grid Resource Registry, by the Monitoring Infrastructure and by the Provenance Tracking System. Figure 1.2 presents relations between these components.

Three optimization modes were defined in *GrAppO*:

- *short-sighted* optimization stands for selecting optimal solution for one Grid Object Class,
- *medium-sighted* optimization designates optimal Grid Object Instances for a set of Grid Object Classes,
- *far-sighted* optimization stands for optimization based on application analysis, it considers dependencies between grid operations to reorder and prioritize invocations.

The far-sighted mode is not yet implemented and designed. Some possible directions of research were mentioned in [8] - finding a method to gain application structure or graph from its sources and considering heuristic algorithms to perform optimization. This approach is known as a workflow scheduling problem.

1.2 Workflow scheduling problem

Workflows can be considered as directed graph built from set of nodes and set of edges. Depending on workflow model, nodes can stand for tasks and data transfers, edges - for control flow (workflow structures are described in section 2.1.1). Scheduling process is a procedure which assigns each task to its resource where the task will be executed[10] based on proper criteria.

Important aspects of the scheduling problem are:

- workflow representation which implicates much restrictions to other aspects,
- service monitoring which should provide data about resource states and as a result enables proper resource discovering and filtering in parallel heterogeneous systems,
- providing algorithms for resource selecting which should base on certain objectives and be able to work with workflow representation.

1.3 Goals of the thesis

The existing optimization system in GrAppO does not provide any optimization based on experiment structure and control flow - far-sighted mode(1.1.3).

Workflow scheduling with a goal to minimize workflow execution can be considered as a solution to this issue. However, Virolab applications are written in Ruby code

and they are not represented as a workflows. Lack of this key information would eliminate this approach but it can be supplemented by building a workflow directly from Ruby scripts and as a result reducing optimization problem to the well-known workflow scheduling problem.

The main goal of this Thesis is to improve experiment execution by analyzing experiment source, relations between GOs, GO operations and their results and finally, build workflows from Virolab experiments. To achieve these goals, we define the following sub-goals:

1. Find dependencies between GO operations invoked from Ruby scripts. The goal is to analyze Ruby source code, locate grid object classes, their instances and operations. Check operation arguments and by resolving them - find dependencies between grid operations.

2. Build workflow basing on application source code. Gather information collected during realization of previous point and locate control flow structures in Ruby source code.

3. Validate approach by building workflows for control-flow patterns and well known applications (Montage, CyberShake, Epigenomics). The purpose is to prepare hypothetical Virolab implementation of these well known applications and transform them to workflows.

4. Provide data needed to enable optimization based on Ruby source code structure. Find what data are required to enable far-sighted optimization of Virolab applications.

5. Provide models for scheduling algorithms. Research what are requirements of scheduling algorithms and prepare scheduling-enable data basing on workflow representation.

1.4 Thesis overview

Chapter 1 gives an introduction to the problem, describes existing Virolab environment and defines the goals for the thesis. Chapter 2 describes how the workflow scheduling problem in grid applications is handled in other works, what issues can be encountered and how to design an application to enable cooperation with existing tools. In chapter 3 the whole process of GridSpace application source analysis is presented with explanation how particular issues were solved. Chapter 4 focuses on the technical aspects of the developed application, introduces its architecture, describes usage of external tools and answers the question how to invoke desired actions. Chapter 5 is an attempt to prove the concept by creating workflows for non-trivial Ruby scripts, using typical workflow

constructs and by recreating existing well-known workflow application as hypothetical GridSpace applications. Chapter 6 points connections between developed solutions and existing workflow scheduling systems. The last chapter 7 summarizes the work by answering questions about which aspects of the problem gives expected results, which brings problems, which trends promise good outcomes and how to modify starting assumptions to reach better results.

```
1  require 'cyfronet/gridspace/goi/core/g_obj'
2
3  puts 'Start␣of␣weka␣experiment␣!!␣(Asynchronous␣version)!!'
4
5  # Create Web Service Grid Object Instance
6  retriever =  GObj.create('cyfronet.gridspace.gem.weka.WekaGem')
7
8  # Build the query
9  query = 'select␣outlook,␣temperature,␣humidity,␣windy,␣play␣from␣
       weather␣limit␣100;'
10 database = "jdbc:mysql://127.0.0.1/test"
11 user = 'testuser'
12 password = ''
13
14 a = retriever.async_loadDataFromDatabase(database, query, user,
       password)
15
16 classifier =
       GObj.create('cyfronet.gridspace.gem.weka.OneRuleClassifier')
17
18 b = retriever.async_splitData(a.get_result, 20).get_result
19 trainA = b.trainingData
20 testA = b.testingData
21
22 # Set the name of attribute that will be predicted
23 attributeName = 'play'
24
25 trained = classifier.async_train(trainA, attributeName)
26 # wait until training is done
27 trained.get_result()
28
29 prediction = classifier.async_classify(testA)
30
31 classificationPercentage = retriever.async_compare(testA,
       prediction.get_result, attributeName)
32 # show results
33 puts 'Prediction␣quality:' + classificationPercentage.get_result.to_s
34 puts 'End␣of␣weka␣experiment␣!!'
```

Figure 1.1: Real Virolab experiment. In line 6 first GO in initialized, then there are three asynchronous operations invoked on this object in lines 14, 18 and 31. In lines 14 and 31, there are created operation handlers `a` and `classificationPercentage`. Their result request are located in lines 18 and 33. The second GO - `classifier` is created in line 16, operation handlers `trained` and `prediction` are results of asynchronous operations in lines 25 and 29. Corespondent result requests are located in lines 27 and 31.

CHAPTER 2

Workflow scheduling and representing

In previous chapter it was decided that Virolab applications are intending to be transformed into workflows and then treated by scheduling algorithms. This chapter describes taxonomies of workflows and workflow scheduling problem, shows workflow representations, typical constructs and introduces workflows generated for existing applications.

2.1 Workflow scheduling taxonomies

Workflow scheduling problem can be considered in many aspects depending on the perspective chosen by us. According to [11] and [10] we can distinguish five main different facets of the problem:

Workflow model. Workflow model classes can be defined basing on model representations and behavior. A detailed description of distinguished classes is presented in section 2.1.1.

Scheduling criteria. Classes of workflow scheduling criteria can be distinguished by optimization goals and methods which are used to measure the cost calculated for a particular criterion. Section 2.1.2 includes its classification.

Scheduling process. Scheduling process taxonomy can be based on the characteristics of information that are processed by the scheduler and the way how this information is processed. Section 2.1.3 contains the workflow scheduling classification based on this purposes.

Resource model. Resource model describes differences between resource classes. First aspect which can be identified distinguishes between resources that have the same parameters (homogeneous) and resources that have different characteristics (i.e., different performance or load). Second aspect of the resource model differences between the class of resources that can execute one and multiple tasks at the same time (multiprogrammed resources).

Task model. Two main classes can be distinguished basing on how tasks are mapped to resources (tasks need fixed number of resources, required resources number is determined before execution time and resource usage of particular task can be changeable.)

2.1.1 Workflow model

Workflow model taxonomy based on scheduling perspective focuses on tasks and data transfers which is a combination of four other well known workflow model perspectives:

Control-flow. Focuses on tasks and their execution order using workflow constructs like sequence, synchronization, parallel split, exclusive choice and others.

Data. Focuses on data flow between tasks in the workflow.

Resource. Focuses on allocation, scheduling and other actions performed on resources according to executing tasks.

Operational. Focuses on how tasks work in such aspects as implementation.

In [11] task and data transfers are called "schedulable units" since they are atomic workflow components used in scheduling process.

Component model

In scheduling perspective there is a distinction between two workflow model classes:

Task oriented. In this approach, task are represented as graph nodes and edges between them stand for data transfers or control preconditions.

Task and data transfer oriented. It is low-level approach, both tasks and data transfers are represented as graph nodes.

Structure

The structure of the workflow is hardly related with scheduling methods, their level of generality and designed for different specific domains. We will distinguish the following three workflow models[11]:

Directed Acyclic Graph (DAG). Workflow is represented by DAG.

Extended digraph. Allows to represent structures like loops or conditions by adding them to DAG model.

Simplified DAG. Workflow structure is enriched by certain regulations and it is represented as well-defined subset of DAG model.

The most common workflow structure representation is DAG. The major disadvantage is the lack of the representation of very common programming statements like `loop`, `parallel loop` or `if`. The solution is to introduce *extended digraph* which extends DAG with cycles (*loops* and `parallel loops`) and conditionals (`if` or `switch`). The opposite approach is a *simplified dag* since it contains simpler structure than DAG model like:

Sequence. Workflow is a single sequence (e.g., pipelined application).

Tree-like. Tree is a representation of workflow graph.

Parallel section. Computations in parallel section are distributed among. multiple workers

Other. There can specified other workflow structures like Fast Fourier Transformation or parallel split(2.4.2).

Atomic structure dynamism

For optimization purposes (as a part of scheduling process) workflow nodes can be added to, removed from the workflow or grouped together into new nodes. Opposite approach is when nodes cannot be modified, removed, added aside from user interaction or normal workflow execution like loop unrolling.

After[11], we distinguish two workflow classes:

Fixed. Workflow structure is static during the scheduling process (only additional dependencies can be added or removed).

Tunable. Nodes can be modified, grouped, added or removed during scheduling process.

Data processing

As in [11], we can create two class workflow model taxonomy based on data processing:

Single input workflow models. Workflows which are executed for single input data.

Pipelined workflow models. Workflows which are executed for many different data inputs that are processed by the workflow as a stream.

2.1.2 Scheduling criteria

Taxonomy of scheduling criteria is based on properties that determine optimization goal and way in which the total cost of a workflow is calculated.

Optimization model

When considering workflow scheduling as an optimization process, scheduling criteria can be defined basing on two perspectives[11][12]:

Workflow-oriented. The optimization criterion is defined for the user who executes workflow (e.g., execution time - *makespan*, economic cost). The goal is to optimize performance of particular workflow.

Grid-oriented. The optimization criterion is defined for the grid environment (e.g., resource usage. economic profit). The goal is to prevent wasting resources when they are waiting for jobs with empty queue or to maximize *throughput* - resource ability to execute proper task number.

Workflow structure dependence

After [11] and [10], we can distinguish two classes of criteria based on whether the workflow structure is considered when calculating total cost:

Structure dependent. (e.g., execution time) Optimizing execution time is the goal of majority of existing workflow scheduling approaches.

Structure independent (e.g., economic cost) Economic cost may be due to expense of used applications which in turn corresponds with en example of structure independent criterion - reliability.

The purpose of these thesis is structure dependent aproach which consider task dependencies to minimize workflow execution time.

Optimization impact

Scheduling criteria can have two different kinds of impact in optimization process. First one occurs when the goal of optimization is to find best cost for certain criterion (e.g., to minimize total cost). Second one corresponds with restrictions imposed of optimization process, it occurs when certain criterion has hard constant limit (e.g., budget limit or deadline). We call them, in order:

Optimization objective. Best possible cost for the given criterion. An example of this class is an optimization objective defined for execution time with a goal to minimize its amount. The other examples are quality of results or security which are supposed to be maximized.

Optimization constraint. Constant limit for the given criterion. If there is a strict requirement of particular quantity like budget or time, it can be named optimization constraint. It defines limit for a certain criterion.

The general approach of defining multi-criteria scheduling is to define one optimization objective and establish constraints for all other criteria [10].

Calculating method

In [11] there are three classes of scheduling criteria in calculating method domain. The representative method of the first class is used to calculate total execution time or total economic cost - it is an addition. An example of the second class calculating method is multiplication. It can be used to calculate data quality or probability of failure, it is simply the multiplication of numbers from range $[0, 1]$. The last one can be explained by the examples of bandwidth in network or pipelined execution where total cost of criterion is the minimal cost of all components. We call them in order:

- additive,
- multiplicative,
- concave.

2.1.3 Scheduling process

Scheduling process should be considered as one of a few steps in bigger process called workflow processing. It is formed by combination of requirements of problem definition, optimization principles and the environment of the wofkflow. Following aspects have major influence on the workflow scheduling process.

Criteria multiplicity

One of the most important aspects of scheduling process is a complexity of scheduling criteria. From the perspective of criteria multiplicity, the simplest are scheduling processes that involve only one criterion and the most complex are scheduling processes that involve multiple criteria. Therefore the scheduling processes can be divided into two classes:

- single criterion,
- multiple criteria.

Workflow multiplicity

Scheduling process can also attempt to optimize the execution of multiple independent workflows at a time. After [11] we distinguish two classes:

Single workflow. Execution of single process is optimized in single scheduling process.

Multiple workflows. Execution of multiple workflows is optimized in single scheduling process.

Dynamism

The third aspect of scheduling process is different that two others since it is significantly more related with a workflow execution. After [11], there can be considered three classes of scheduling process dynamism depending on a point in time when the decision is made. In order:

Just-in-time scheduling. Decision is postponed as long as possible.

Full-ahead planning. Static approach, workflow is scheduled before execution.

Hybrid. Combination of the two previous approaches.

In addition, we know that the workflow structure can be modified during scheduling process, this makes big picture of the constantly changing workflow structure during the scheduling process which is repeated many times during workflow execution.

2.2 Benchmark workflows

Workflow scheduling and execution implies a need of testing and benchmarking workflow scheduling systems. For that purpose, inspired by real world applications, the *workflow generator* was created. Arbitrarily large workflow models can be created providing ability of benchmarking and comparing implementations efficiency [13].

2.2.1 Montage

Montage (An Astronomical Image Mosaic Engine)[14] is an open source toolkit maintained by NASA/IPAC Infrared Science Archive which can merge sky images into mosaics. It was designed as a portable application which can be used by astronomers on their desktop computers and also adopted to running on grid infrastructure.

There are four main steps in the image assembling process:

- gather information from images about its geometry (they are kept in a Flexible Image Transport System - FITS format, which con represent that kind of data) and process it to calculate geometry of the result mosaic,
- rescale, rotate, change coordinates of input images to gain the same spatial scale,
- get background radiation values of each image to align flux scales and background levels in whole mosaic,
- join images which corrected background.

These steps are performed in portable, separated ANSI C modules.

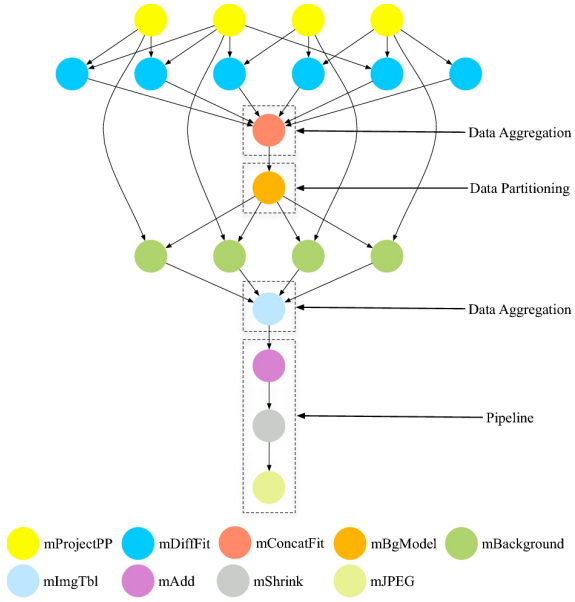


Figure 2.1: Montage workflow[1] generated by the workflow generator[13]. Size of the problem is determined by a number of input images of a given region of the sky while each image has its `mProjectPP` job and for each pair of overlapping images `mDiffFit` has to be performed. Job `mConcatFit` fits differences between images and `mBgModel` makes good global fit. At least `mImgTbl` aggregates metadata from all images and `mAdd` (which is the most computationally intensive job in the workflow), `mShrink` and `mJPEG` jobs produce final image by gathering all images to final mosaic, reducing size of output file by averaging blocks of pixels and finally converting it to JPEG format.

This workflow performs following operations:

There are some simplifications in the Montage workflow described in [13]. The following list is a attempt to bind nodes from figure 2.1 with modules of montage application described on its website - [1]:

- `mProjectPP` - reprojects a single image to the defined scale,
- `mDiffFit` - `mDiffExec` runs `mOverlap` module to determine which images overlaps and then runs `mDiff` module which performs image difference between pair of overlapping images,
- `mConcatFit` - `mFitExec` executes `mFitplane` module which fits plane to an image,
- `mBgModel` - module has the same name in montage application. It is a modeling/fitting program which determines a set of corrections to apply to each image in order to achieve a ”best” global fit,

- `mBackground` - `mBgExec` runs `mBackground` module to perform corrections generated by `mFitPlane` module,
- `mImgTbl` - module `mImgTbl` extracts geometry information from a set of files which are used in following operations,
- `mAdd` - `mAdd` module joins all images to form output mosaic,
- `mShrink` - module `mShrink` reduces size of file by averaging blocks of pixels,
- `mJPEG` - is a one of montage application utilities which generates `JPEG` file.

2.2.2 CyberShake

Project CyberShake is maintained by Southern California Earthquake Center (SCEC). The main goal of a project is to construct a physics-based models of earthquake processes and to develop scientific framework basing on these models for seismic hazard analysis[15]. For each Earth rupture in an analyzing area, variations of its parameters are created. Each variation represents a potential earthquake. To make it clear how big is the problem: given 7000 ruptures, CyberShake will generate 415000 rupture variations.[16]. CyberShake uses ruptures and rupture variations to create Strain Green Tensor (SGT) around concrete site of interest. SGT describes seismic wave fields. Basing on SGT, CyberShake generates seismogram which, in the next step, are processed to obtain Peak Spectral Acceleration (PSA) values. In the last step PSA values are combined into hazard curves which can be used to produce seismic hazard map for the whole analyzed area.

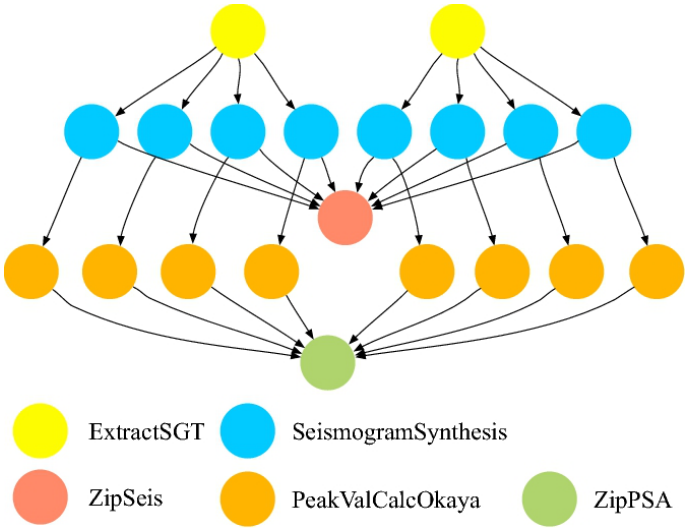


Figure 2.2: CyberShake workflow[15, 16] generated by the workflow generator[13]. Size of the problem is determined by a number of locations from which SGT data has to be extracted - `ExtractSGT`. Job `SeismogramSynthesis` generates seismograms which represent ground motions. Then, gained seismograms are combined in job `ZipSeis` and used to calculate PSA in job `PeakValCalcOkaya`. PSA values returned by the last mentioned job are combined into hazard curve in the last operation - `ZipPSA`.

This workflow performs following operations:

- ExtractSGT - extracts SGT data corresponding to the location,
- SeismogramSynthesis - generates seismogram which represents ground motions,
- ZipSeis - combines seismograms of ground motions,
- PeakValCalcOkaya - calculates PSA,
- ZipPSA - combines PSA into a hazard curve.

The execution of each of steps: ExtractSGT, SeismogramSynthesis and PSA processing takes just a couple of minutes[16], but SGT extracting must be performed for all ruptures and two more times for each rapture variations. As it was mentioned before, typical problem contains `~7000` raptures which makes `415000` rupture variations.

Regarding the workflow size, the distance from the starting node to the exit node is relatively small, but the workflow can be very wide, depending on the input data. In other words the critical path is short but there are a big number of parallel processes. Each parallel process consists a sequence of only two tasks: SeismogramSynthesis and PeakValCalcOkaya.

2.2.3 Epigenomics

The USC epigenome Center[17] Epigenomics conducts research on the epigenetic state on human genome. The Epigenomics workflow is based on the application which is used for that research. It takes DNA sequences which are separated into several chunks. For each chunk, independently from other, several conversions, mappings and filters are applied. This workflow is an example of pipelined application.

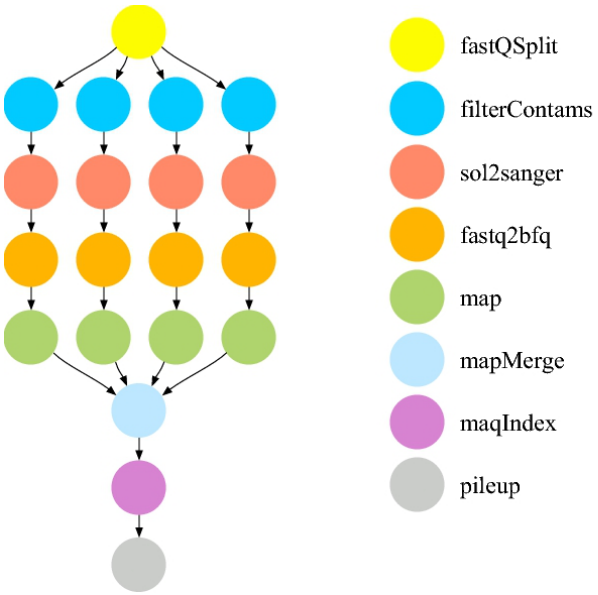


Figure 2.3: Epigenomics workflow generated by the workflow generator [13]. Size of a problem is determined by a size of input sequences. They are split into several chunks by a `fastQSplit` job. Then, various operations are performed sequentially to each chunk - `filterContams`, `sol2sanger`, `fastq2bfq` and `map`. After a pipeline, all chunks are merged into one result.

This workflow performs following operations:

- fastQSplit - The DNA sequence data is split into several chunks that can be operated on in parallel,
- filterContams - Then noisy and contaminating sequences are filtered,
- map - remaining sequences are mapped into correct locations in a genome,
- mapMerge - generates global map,
- maqIndex - identifies density sequence for each position in the genome.

2.3 Petri nets workflow graph representation

Petri nets is a tool which can provide graphical and formal description of concurrent processes in distributed systems. They were introduced in Carl Petri Ph.D. thesis[18]. Petri net is built from *states* and *transitions* which are connected with arrows - *arcs*. Arrows are able to connect *state* with *transition* or *transition* with *state*, connections between *transitions* or *states* are not allowed.

States (or *places*) stand for system states, *transitions* stand for actions. *Arcs* are bounded with expressions which determined how the state is changed after *transition*. Places may contain zero on multiple *tokens*. During execution of Petri net, *tokens* are moving across the net according to arrow directions.

This kind of Petri nets are also named low-level Petri nets do distinguish them from extensions which are called high-level Petri nets. One of them is Coloured Petri-Net (CPN). *Tokens* in CPN can carry data and can be distinguished between each other[19].

Following [19], differences between low-level and high-level Petri nets are similar with differences between low-level and high-level programming languages - high level languages have more advanced structuring facilities like types and, as a result, they provide more modeling capabilities.

The execution of CPN looks as follows: if there is are tokens in a transition input state (there is a incoming arc which have matched expression), then, they are passed to each of transition output states (these which are connected with a current transition with arcs with matched expressions). The transition may fire only if there is a token in input state.

(a) Petri net. States (circle nodes) contain tokens (black dots). There is one transition T between states S1 and S2.

(b) Colored petri net. Token are coloured to distinguish between them. They can also carry a data.

Figure 2.4: Two kinds of Petri nets low level 2.4a and CPN - 2.4b. Both contain states S1 and S2 and one transition T. CPN petri net has arcs with a expression which determines passing tokens between nodes.

2.4 Workflow patterns

The motivation for creating workflow patterns by *Workflow Patterns Initiative* was to delineate fundamental requirements for workflow modeling[20][21]. The area of research included various perspectives - control flow, resource, data, etc. Resulting patterns can be used to examine these purposes of workflow modeling tools.

From the wide spectrum of cases which were considered by *Workflow Patterns Initiative*, basic control-flow patterns were chosen for further considerations.

2.4.1 Sequence pattern

Sequence pattern is a fundamental building block for workflow processes[20]. Activities are executed in a sequence, the activity that follows a running activity is started as soon as the preceding activity is completed. This pattern is widely supported by all workflow management systems. The typical realization of this pattern is done by associating two activities with unconditional control flow arrow[22]. Figure 2.5 presents sequence pattern using the CPN formalism.

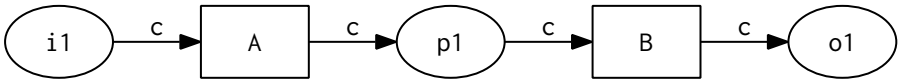


Figure 2.5: Sequence workflow pattern in CPN formalism. There is a pipeline of three *states* - `i1`, `p1` and `o1` separated with `activities` - `A` and `B`. These construct ensures that *activities* are performed in sequential order.

2.4.2 Parallel split

`Parallel split` is a point in workflow process where the particular branch of a control flow splits into multiple branches which can be executed concurrently.[22] The other names for `parallel split` are: `fork` and `AND-split`[20]. Implementation of the `parallel split` pattern can be implicit and explicit. First aproach can be realized by multiple unconditioned edges outgoing from particular activity or by an edge representing control flow which splits into multiple branches. Specific construct dedicated to `parallel split` is required when particular tool implements this parent explicitly. Figure 2.6 presents `parallel split` pattern using the CPN formalism - activities `B` end `C` are executed in parallel when activity `A` is finished.

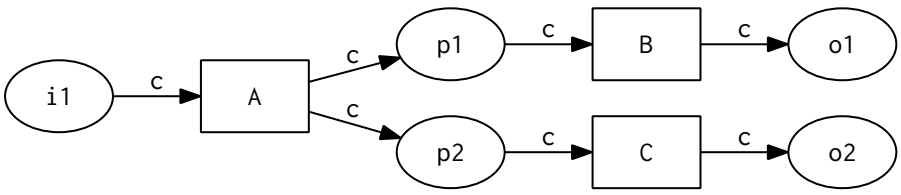


Figure 2.6: Parallel split workflow pattern in CPN formalism. Activity `A` has two outgoing *arcs* with the same condition. Activities `B` and `C` are executed in parallel.

2.4.3 Synchronization

Synchronization is a point in the workflow process where many threads of control are joined into one[22]. Workflow realization of this pattern can be explicit and implicit. Tools that implement *synchronization* pattern explicitly contain particular construct, the implicit way of implementing this pattern is realized by many transitions (representing control flows) coming to one activity[20]. Figure 2.7 shows implicit representation of *synchronization* pattern in CPN formalism. This pattern is also known as `AND-join` and *synchronizer*[20].

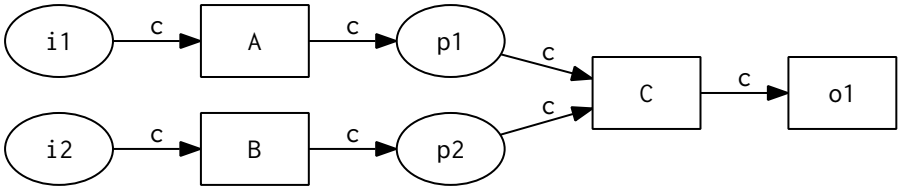


Figure 2.7: Synchronization workflow pattern in CPN formalism. *Arcs* which are outgoing from *states* `p1` and `p2`, point on the same *activity* which makes from it a synchronization point of a control flow.

2.4.4 Exclusive choice

Exclusive choice is a point in the workflow process where, basing on the decision, one from several outgoing branches is chosen[22]. Similarly to *parallel split* pattern (2.4.2), *exclusive choice* can be realized explicitly and implicitly. Implementation of explicit representation is when the tool provides particular construct and implicit representation is when condition of outgoing control-from edges have disjoint conditions. Alternative names for this pattern are as follows: `case statement`, `switch`, `decision`, `exclusive OR-split`, `XOR-split`.[20]

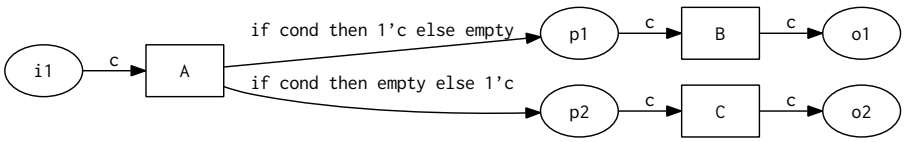


Figure 2.8: Exclusive choice workflow pattern in CPN formalism. *Arcs* outgoing from *state* `A` have excluded conditions - only one outgoing *arc* is chosen by control flow.

2.4.5 Simple merge

Simple merge is a point in the workflow process where two or more branches come together without synchronization.[22] Moreover, incoming branches are not executed in

parallel and the result of one of them is passed to the subsequent branch. Some workflow tools have a separate construct for this pattern, we call it explicit representation. In other cases *simple merge* pattern can be created using lower level constructs. Figure 2.9 shows implicit representation of simple merge pattern in CPN formalism. *Simple merge* is also named `XOR-join`, `asynchronous join` or just `merge`.[20]

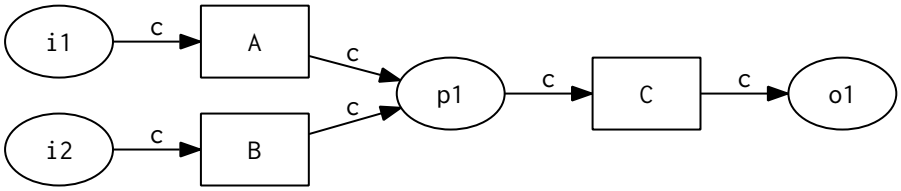


Figure 2.9: Simple merge workflow pattern in CPN formalism. *State* `p1` is a point where two branches of control flow - one from *activity* `A`, other from `B`, come together without synchronization (incoming branches are not executed in parallel).

2.5 Workflow Description Languages

2.5.1 AGWL

Abstract Grid Workflow Language (AGWL)[23] is a XML-based workflow language. Using AGWL constructs, we can describe grid workflows on a high level of abstraction, since AGWL workflow does not include implementation details.

Activities. Activities are defined as *units of work*. It can be a computation (let associate it with a *grid operation*), sequence of activities, or a composed sub-activity.

Activity is represented by a black box with input/output ports and additional information in constraints and properties. Constrains may define environment requirements. Properties contain data which is used by workflow tools like scheduling applications.

AGWL supports hierarchical decomposition of activities - some part of the workflow (sequence of activities or composed sub-activity) can be represented by a single activity. In that case input/output ports of enclosed workflow are mapped to input/output ports of composed activity.

Control and data flow. Control flow and data flow specify workflow composition. Data flow specification is realized by connections between input and output ports of activities, it can be enriched by some additional information in associated constrains, e.g., protocol specification. Control flow is defined by links between activity ports and by control-flow constructs: sequential flow - `sequence`, exclusive choice - `if`, `switch`, sequential loops - `while`, `dountil`, `for`.

As we see, AGWL supports wide spectrum of commonly used constructs which are specially useful in scientific grid workflows. There is `parallel` and `parallel for` which provide simple concurrency model. Activities defined in `parallel` in general case are executed concurrently. `Parallel loop` does just a little more - activities defined in that construct receives `index` and proper value from the given array.

2.5.2 YAWL

The origin of Yet Another Workflow Language (YAWL)[24] was preceded by gathering a wide collection of workflow patterns[20] presented in section 2.4. Collected patterns have been implemented in existing workflow tools. Then, these tools have been evaluated for abilities to capture control flows for complex workflow processes. The new workflow language (YAWL) has been designed based on Petri nets enriched with additional constructions to provide better support for workflow patterns. YAWL is XML-based language.

Workflow in YAWL is a set of *extended workflow nets*. They are formed in hierarchical structure. Task (in [24], authors use term *task* instead if *activity* but in fact, task are synonyms of AGWL's *activities*) can be one of both: *atomic task* and *composite task* which refers to *extended workflow net* in the lower level of hierarchy.

Each *extended workflow net* contains *tasks* and *conditions* (they can be interpreted as places). One unique input condition and one unique output condition are required for *extended workflow net*.

Atomic tasks, as well as composite, can have multiple instances, number of them is determined by upper and lower bounds. The task is completed when all task instances have finished (specification predicts threshold for the number of instances that has to finish before a whole task is done and parameter which indicates if it is possible to add new instances during task execution).

YAWL elements

YAWL language consists various elements including condition elements: *1* condition - which also can be interpreted as places, *2* input condition - each workflow has unique output condition, *3* output condition - each workflow has unique output condition. There are task elements: *1* atomic task„ *2* composite task - refers to workflow at the lower level of hierarchy.

Multiple instances:

- Multiple instances of an atomic task.
- Multiple instances of a composite task.

Splitting and joining:

- AND-split task
- XOR-split task
- OR-split task
- AND-join task
- XOR-join task
- OR-join task

2.6 Scheduling algorithms

One of the main goals of this thesis is to work out how GridSpace applications can be scheduled using existing workflow scheduling approaches. While the scheduling problem was already introduced and classified in section 2.1, requirements for concrete workflow scheduling approaches remaining unknown.

Grid Scheduler (GS) process can be generalized into stages[12]:

- resource discovering and filtering,
- **resource selecting and scheduling according to certain objectives**,
- job submission.

The scheduling algorithms is particular to a second stage.

GS systems, in general, seem to have two modules/services which support scheduling process. They are *Grid information service* and *Cost estimation*. First module provides statuses of available resources - available CPU and memory, network bandwidth, load of a site in particular period.

Cost estimation module based on the some additional information about applications - like profiling, benchmarking or previous usage, estimates the cost of executing application on the particular resource.

While it is shown as a single unit, there can be more than one GS deployed in the system, each characterized by a different performance or scalability. Moreover they can form various structures - centralized, hierarchical or decentralized. In contrast to a traditional distributed system, grid workflow scheduler is not able to manage resources.

2.6.1 Taxonomy of scheduling algorithms

Workflow scheduling algorithms can be analyzed in the context of the already introduced taxonomy of Workflow scheduling (2.1), moreover they can be assigned to various categories in similar terms. In section 2.1.3, workflow scheduling process was divided basing on point in time when scheduling decisions are performed.

Static approach is good when there is workflow model which can not be modified during execution. It can not be applied when there are loops in the workflow which have undefined numbers of iteration or conditions with expression evaluated in execution time. Workflow scheduling process was categorized by its dependency on workflow structure - 2.1.2, these categories can be enriched by subcategories as follows.

Structure independent

This category can be considered into two aspects: system point of view, whose goal is to achieve high throughput and application where some heuristic algorithms can be applied to estimate application execution.

An example of the heuristic which is based on the predicted execution time (it can be named static method since it is applied before workflow execution) is: *Minimum Execution Time* - algorithm with performance estimation - assign task to resource which have shortest execution time expectancy. The goal is to bind particular application with most suitable resource.

Structure dependent

Structure dependent algorithms work with workflows represented as DAG where nodes stand for tasks and edges determine the execution order.

Algorithms from this category can be divided into static, dynamic and hybrid (e.g., static enhanced by dynamic rescheduling).

Static algorithms include:

- list algorithms,
- cluster algorithms,
- duplication-based algorithms.

List heuristics. Tasks are grouped in priority lists, tasks from the top of the list - with highest priority are processed before others. Differences between algorithms are in the method of calculating priorities.

An example of this kind of heuristic is Heterogoneous Earliest-Finish-Time (HEFT)[25]. HEFT algorithm has two major phases:

Task Prioritizing Phase. It orders the tasks on the list based on their distance from the exit nodes plus it takes into the consideration their computational and communication cost.

Processor Selection Phase. Then, for each task on the ordered list, HEFT algorithm schedules task in the earliest idle time slot on available resource. Selected idle time-slot should be long enough to hold computation process of newly scheduled task - time-slot should be longer than estimated execution time.

Clustering heuristics. Group tasks which are expected to perform massive communication with each other and, to minimize communication costs, assigns them to the same resource. This problem is NP-complete, thus various heuristics are used to solve this issue.

Usually, clustering heuristics algorithm has two phases:

- split original graph into clusters,
- refine the clusters produced in first phase.

In theory, tasks are mapped to infinite number of clusters but in practice merging step (in second phase) tasks are mapped to the amount of clusters equal to number of resources.

Clustering heuristics has its own taxonomy. Algorithms can be *linear* or *nonlinear* depending if independent tasks can be assigned to the same cluster (*nonlinear*) or not (*linear*).

The example of this category is Dominant Sequence Clustering (DSC)[26]. It is based on concept of Dominant Sequence (DS) which is a critical path of the scheduled DAG (it is different from critical path of the clustered DAG). Application of this algorithm is described in section 6.1.3.

Duplication based algorithms. Task are duplicated and executed on different resources which may minimize resource idle time and a communication cost between resources. Algorithms from this category differ according to a task selection strategies (which tasks are duplicated, how many duplications is made and on which resources).

2.6.2 Dynamism of the grid

Described algorithms do not consider dynamism of the grid as they are based on the static resource performance estimation[12]. Dynamism is a result of a fact that resources are shared between jobs and execution of one of them may affect others. One of the solutions is to create multiclusters with their own local schedulers. This kind of algorithms

consider the optimization of DAG makespan on multiclusters which arrive as a linear function of time. Schedulers would have hierarchical structure, each cluster is expected to gather as many tasks as possible, during the execution, it reports finish time estimations to a global scheduler.

2.7 Summary

This chapter introduced a state of a research into a workflow scheduling problem. The taxonomies of workflow, workflow scheduling and workflow scheduling algorithms (sections 2.1, 2.6) points approaches how Virolab applications should be analyzed and transformed to achieve the goal of a far-sighted optimization based on workflows scheduling. Workflow patterns described in section 2.4 and benchmark workflows - 2.2 will be used to evaluate worked out solutions.

CHAPTER 3

Concept of script application analysis

Works introduced in previous chapter gave answers how master thesis goals can be reached. In GridSpace applications, workflows are not defined in a workflow oriented language (like AGWL or YAWL) but in Ruby scripts. Thus, workflow has to be created from Virolab application. The purpose of this chapter is a process of collecting data that are required to transform ruby scripts into workflows of grid operations.

3.1 Workflow elements in experiments

To achieve the goal of creating workflows, some information from Ruby source code have to be extracted. It is important to identify all workflow *activities* and detect how *data-flow* and *control-flow* are realized.

Detecting activities. Activities are identified as grid object operations.

```
a = GObj.create("MyGObj")
b = a.do_sth
```

Figure 3.1: Script with synchronous grid operation `do_sth` performed on grid object `a`.

```
a = GObj.create("MyGObj")
b = a.async_do_sth(c)
d = b.get_result
```

Figure 3.2: Script with asynchronous grid operation `async_do_sth` performed on grid object `b`.

In case of the synchronous operations (figure 3.1) activity is in a one to one relation with a grid object operation. But asynchronous operations(figure 3.2) are spit into two statements - the operation handler request invoked on grid object and the result request invoked on the operation handler.

The synchronous grid object operation is a special case of asynchronous operation where operation handler is requested for a result just after it was acquired.

Data and control flow elements are not so trivial to found in experiments. It can be said that there is a data or control flow between grid operations if the result of the first one may affect execution of second one.

The interaction between grid operation occurs when:

- Result of first one affects any of the arguments of second one (a data flow dependency).
- Second grid operation is in control structure like `loop` or `if` statement which conditions depends on result of the first grid operation (a control flow dependency).

3.2 Analyzing steps

It was established in previous section that to create workflow, grid operation and control structures have to be located and the dependencies between grid operations have to be resolved. These three goals imply a long chain of operations.

Before locating grid operations, grid objects have to be founded. But to locate grid objects, grid objects initializations have to be found and to achieve that goal, all assignments have to be analyzed to check which of them are initializing grid objects.

Control structures and grid operations are even more complicated. All function calls and all assignments have to be analyzed to find how variables are changing in the whole script.

The input for the analysis is grid application represented as pure Ruby code. Figure 3.3 shows all steps of the analysis process and their dependencies.

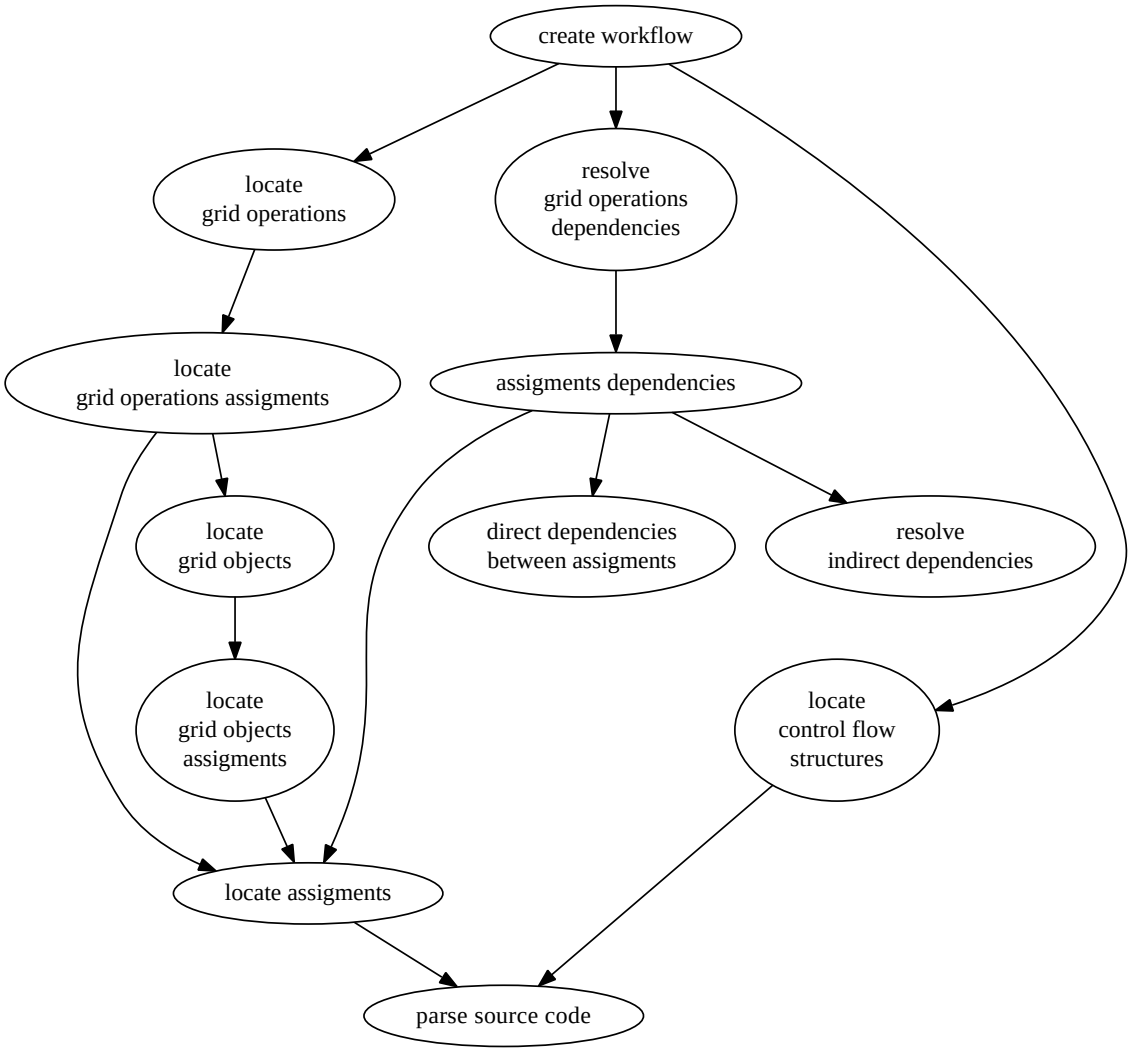


Figure 3.3: During the development of analyzing process some routines were reproduced in different aspects of the analysis. As it will be shown, data produced by one algorithm are used to different purposes. The graph describes links between them. Arrows can be read as "needs data from" (e.g., a routine which locates grid objects needs data from a routine which locates grid objects assignments).

3.2.1 Source code analysis

Ruby parser[27] is a Ruby language parser written in pure ruby, it means it can be used with any Ruby implementation such as JRuby. It converts ruby source code to symbolic expressions (also called S-expression or sexp) using ruby arrays and base types.

Figure 3.4 contains a simple example.

```
a = GObj.create
b = a.async_do_sth
c = b.get_result
```

Figure 3.4: Simple example of the Virolab script.

Ruby parser [27] transforms this source code into s-expressions as in figure 3.5.

```
s(:block,
  s(:lasgn, :a,
    s(:call, s(:const, :GObj), :create, s(:arglist))),
  s(:lasgn, :b,
    s(:call, s(:lvar, :a), :async_do_sth, s(:arglist))),
  s(:lasgn, :c,
    s(:call, s(:lvar, :b), :get_result, s(:arglist))))
```

Figure 3.5: Listing presents S-expressions produced from from script 3.4.

`s()` which repeats in listing is a function which creates `Sexp` object, it can be represented using the array representation (figure 3.6).

```
[:block,
  [:lasgn, :a,
    [:call, [:const, :GObj], :create, [:arglist]]],
  [:lasgn, :b,
    [:call, [:lvar, :a], :async_do_sth, [:arglist]]],
  [:lasgn, :c,
    [:call, [:lvar, :b], :get_result, [:arglist]]]]
```

Figure 3.6: S-expressions - s() changed to arrays to simplify.

Figure 3.6 shows what really s-expressions are. The first element of an array is a symbol of operation, the remaining elements are operations data. In the analyzed example, there is one `block` operation which contains three `left assignments`. The first one saves the result of a function call to variable `a`. Function is called by the constant `GObj`, its name is `create` and it has an empty argument list. The second and third assignments are very similar, except that the function is reached by a variable, not by the constant.

S-expressions analysis

Full analysis process would be very complex, particularly for Ruby since the full list of operations holds 105 elements. 38 most important operation types for the grid application

were selected, like `assignment`, `function call`, `arguments list`, `loop` and others. Each of these 38 operations has implemented a routine which analyze s-expression.

To allow further analysis, s-expressions are converted into internal representation. When performing the analyzing process, 38 most important operation types are processed, all others are ignored. The data structure is prepared to keep additional data for each operation and optimized for easy and efficient traversing.

At this point of analyzing process, each tree node contains `type` and `name` (figure 3.7).

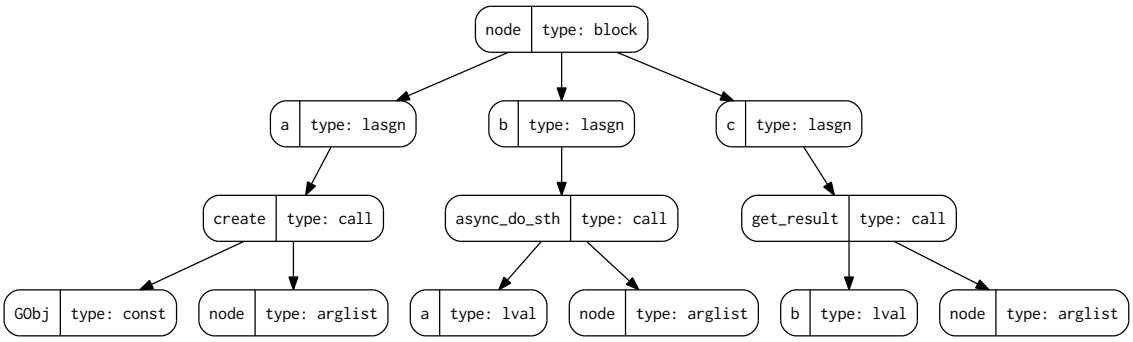


Figure 3.7: Internal representation. It is transformed S-expression from figure 3.5.

3.2.2 Locate grid objects and operations

According to figure 3.3, in this step grid operations are going to be located in a Ruby code.

To achieve this goal the analyzer has to identify which variables are grid objects. With that knowledge, it will be possible to point grid operations as function calls on grid objects and grid operation handlers as returning values.

Locate grid objects

From all the variables, grid objects are those which are created in following way:

```
g_obj = GObj.create("some_string")
```

Thus, grid objects are created in the assignments where on the right side is `create` function call on `GObj` constant. Based on a figure 3.7, it is assumed that every structure in internal representation which is similar to tree graph in figure 3.8 is a grid object creation.

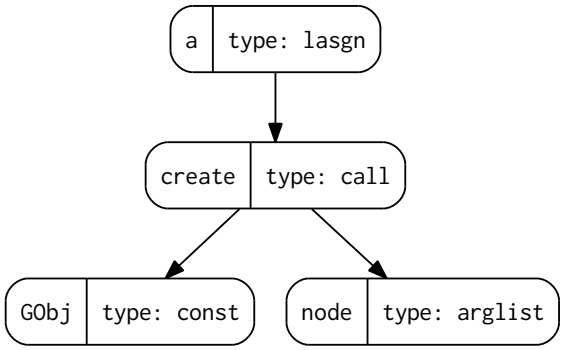


Figure 3.8: Grid object creation pattern - fragment of internal representation which stands for operation: `a = GObj.create`. Grid objects can be located by searching internal representation for that kind of constructs.

Now it is known which variables are grid objects, about their names and positions in internal representation where they are created.

There are enough data to determine grid objects scopes. Analyzing analogies between source code (figure 3.4) and internal representation (figure 3.7) we can notice that:

- grid object variable is accessible in all nodes that belong to grid object assignment tree (figure 3.8),
- grid object is accessible in a given node if it is accessible from its parent or from the first node on the left which belongs to the same parent.

If the above definition was applied to the tree from figure 3.7, it would obtained result from figure 3.9.

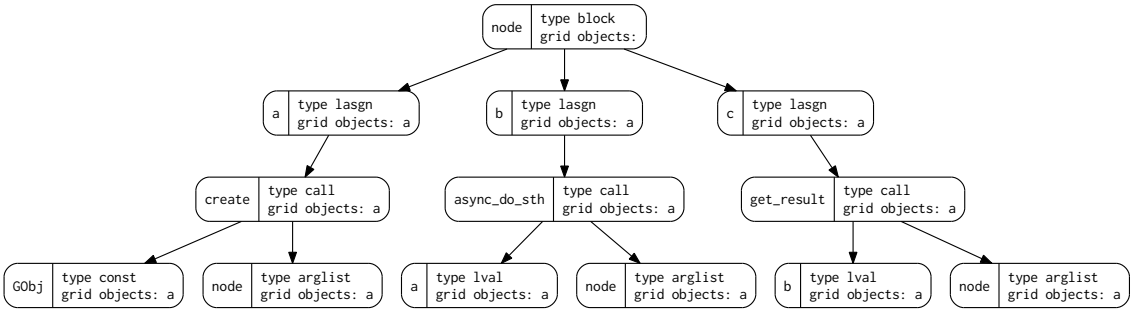


Figure 3.9: Internal representation of script 3.4 with grid objects scope. If a particular node stands for method invocation and its name is included in grid objects list, this method invocation is gird operation.

One more example with two grid objects:

```
a = GObj.create
b = a.async_do_sth
c = b.get_result
d = GObj.create
e = d.async_do_sth
f = e.get_result
```

Its internal representation with grid object scopes is shown in figure 3.10.

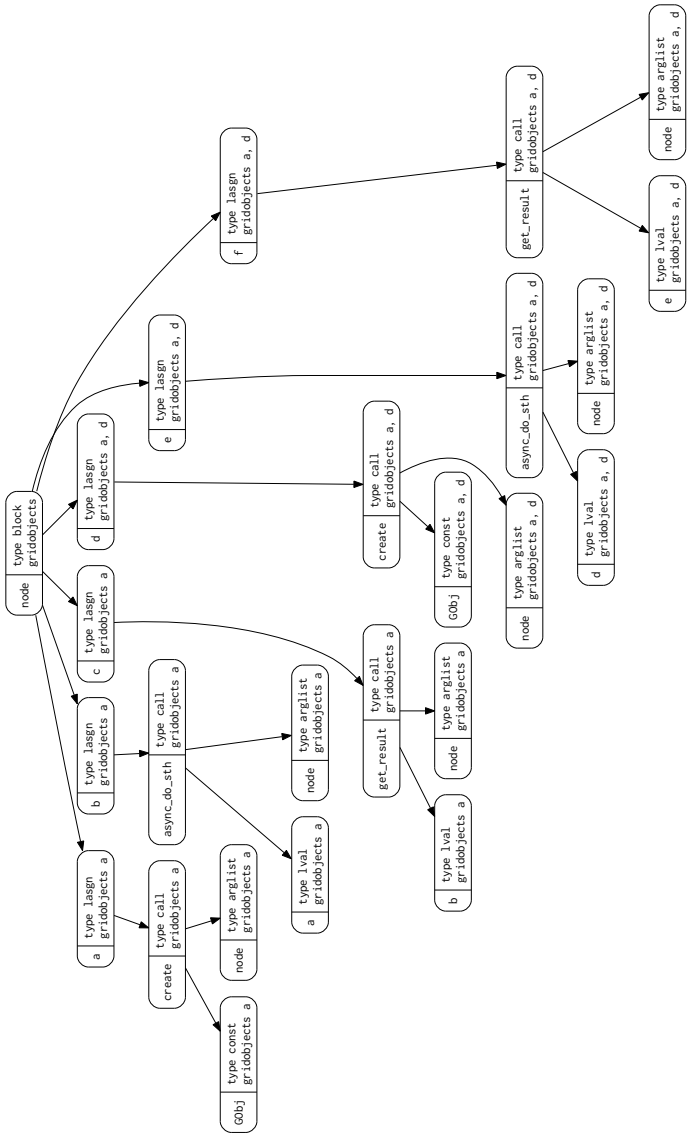


Figure 3.10: Internal representation of script 3.4 with grid objects scope. In each node, list which occurs with label `grid objects:`, stand for grid object names which are accessible.

Locate grid operations

In internal representation, the node is grid operation when:

- its name starts with `async_`,
- its type is `call`,
- its first son (counting from the left to right) has type `lval`,
- its first son name is the same as any grid object in the scope.

3.2.3 Resolve grid operations dependencies

Resolving dependencies between grid operations requires knowledge about all nodes dependencies. Since grid operations are a subset of all operations, finding all dependencies will fulfill requirements of this goal.

Resolve variables dependencies

Operation was spit into to two sub-processes.

In the first one, variables are examined if they have any direct dependencies - this case occurs between two variables `a` and `b` when value of variable `b` is calculated using a value of variable `a`.

Second step is to resolve direct dependencies to acquire knowledge about dependencies between every pair of nodes in internal representation.

Detecting direct dependencies. To find direct dependencies following operations are performed for each node in internal representation:

- if its type is `lasgn`, node is dependent on all nodes of type `lval` that are below examining node,
- if the first son of the examining node (counting from the left to right) has type `call`, all direct dependencies from examining node are transited to its first son

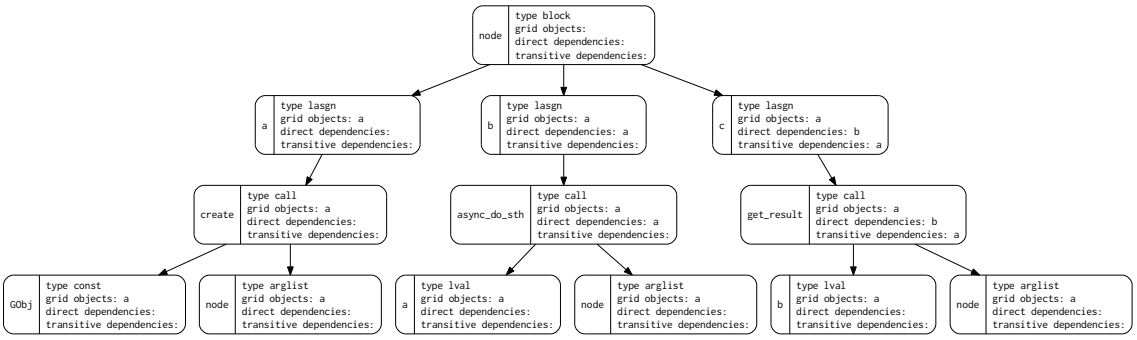


Figure 3.12: Internal representation of script 3.4 with transitive dependencies. In graph, node `c` is dependent on node `b` and node `b` is dependent on node `a`, thus node `c` depends on node `a` through node `b`. `Transitive dependencies` list stands for transitive dependencies.

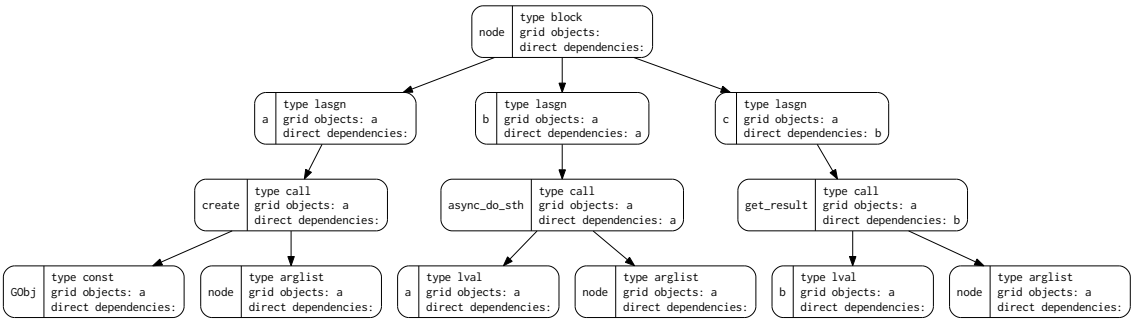


Figure 3.11: Internal representation of script 3.4 with direct dependencies. If given node, has in any of its branches present node of type `lval`, it is directly dependent on variable of `lval` node name. `Direct dependencies` list stands for direct dependencies.

Preceding algorithm for script 3.4 produces internal representation shown in figure 3.11:

Detecting transitive dependencies. To detect transitive dependencies very simple recursive operation is performed. If node `a` depends on node `b` and node `b` depends on node `c` but `a` does not depend on `c`, add node `c` to a list of indirect dependencies of node `a`.

Internal representation after applying this step is shown in figure 3.12.

Locate operation handlers

To associate grid operation and result request on its operation handler, information about direct dependencies are used. Node `b` is a result request on handler for operation at node `a` if:

- distance from node `b` to root node is equal or greater than distance from node `a` to root node,
- branch of node `b` is connected on the left side of connection of node `a` to its parent,
- node `b` depends (directly) from node `a`,
- node `b` depends on any grid object which it has in its scope,
- a type of node `b` is `lasgn`,
- first sons name of node `b` has name `get_result` or first sons son has this condition met.

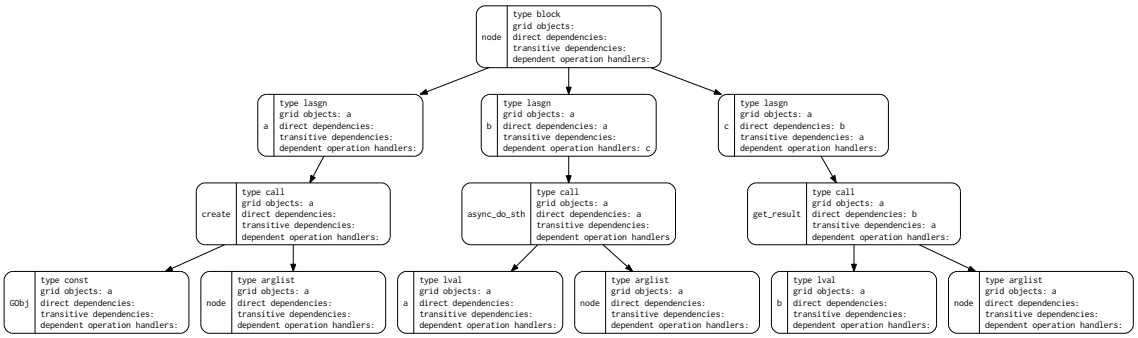


Figure 3.13: Internal representation of script 3.4 with located operation handlers. If there are two nodes, first stands for grid operation and second stands for result request on operation handler and second node directly depends on first node, second node is result request correspondent to operation of first node. This kind of relation happens between node `b` and `c`.

Internal representation with linked grid operations with corresponded operation handler requests is shown in figure 3.13.

3.2.4 Reassignment issue

The operation of resolving dependencies is based on variable names. Thus, it is important to recognize these names properly and to be certain that variable name represents value that is expected.

Ruby, as many other imperative languages, allows reassignments. The problem occurs when new variable value is assigned to variable label which was already used and still appears in the scope. Since it is very common practice, it might make impossible to resolve dependencies correctly even in simple scripts and as a result, impossible to create workflows.

```
a = "foo"
a = 0
```

Figure 3.14: Reassignment issue

Listing 3.14 shows the reassignment issue. In this script, variable a is initialized with a string value `"foo"` but then in succeeding line there is assignment where new reference is created. Now, variable `a` points to `0` and the old value - `"foo"` is no longer accessible. Regarding to Virolab scripts, lets investigate what happens if any of these two assignments is a grid operation.

If the second assignments is a grid operations. There is nothing to add, except that optimizer have to recognize variable `b` as a request handler:

```
b = 2
a = GObj.create
b = a.async_do_sth
```

If the first assignment is a grid operation. Request handler is covered by new reference and it is no longer accessible, it means, there is no need to track further usage of variable `b`. Grid operation result will never be acquired.

```
a = GObj.create
b = a.async_do_sth
b = 2
```

Solution

The solution to this problem is based on changing the variable names. When the assignment to a variable name which already exists occurs, a suffix is added to this variable and all its occurrences. Thus variable with new value will have different name than variable with the old value. Following source code with reassignments:

```
a = "foo"
a = 0
b = a + 2
```

is translated to (in fact there is no any code translations since the whole procedure takes place only on internal representation):

```
a = "foo"
a_1 = 0
b = a_1 + 2
```

Figure 3.15 is a more complex example of reassignment issue with grid objects and grid operations. Its internal representation with resolved reassignment issue is shown in figure 3.16.

```
a = GObj.create
b = a.async_do_sth
c = b.get_result
b = a.async_do_sth(c)
c = b.get_result
```

Figure 3.15: Reassignment issue with grid operations

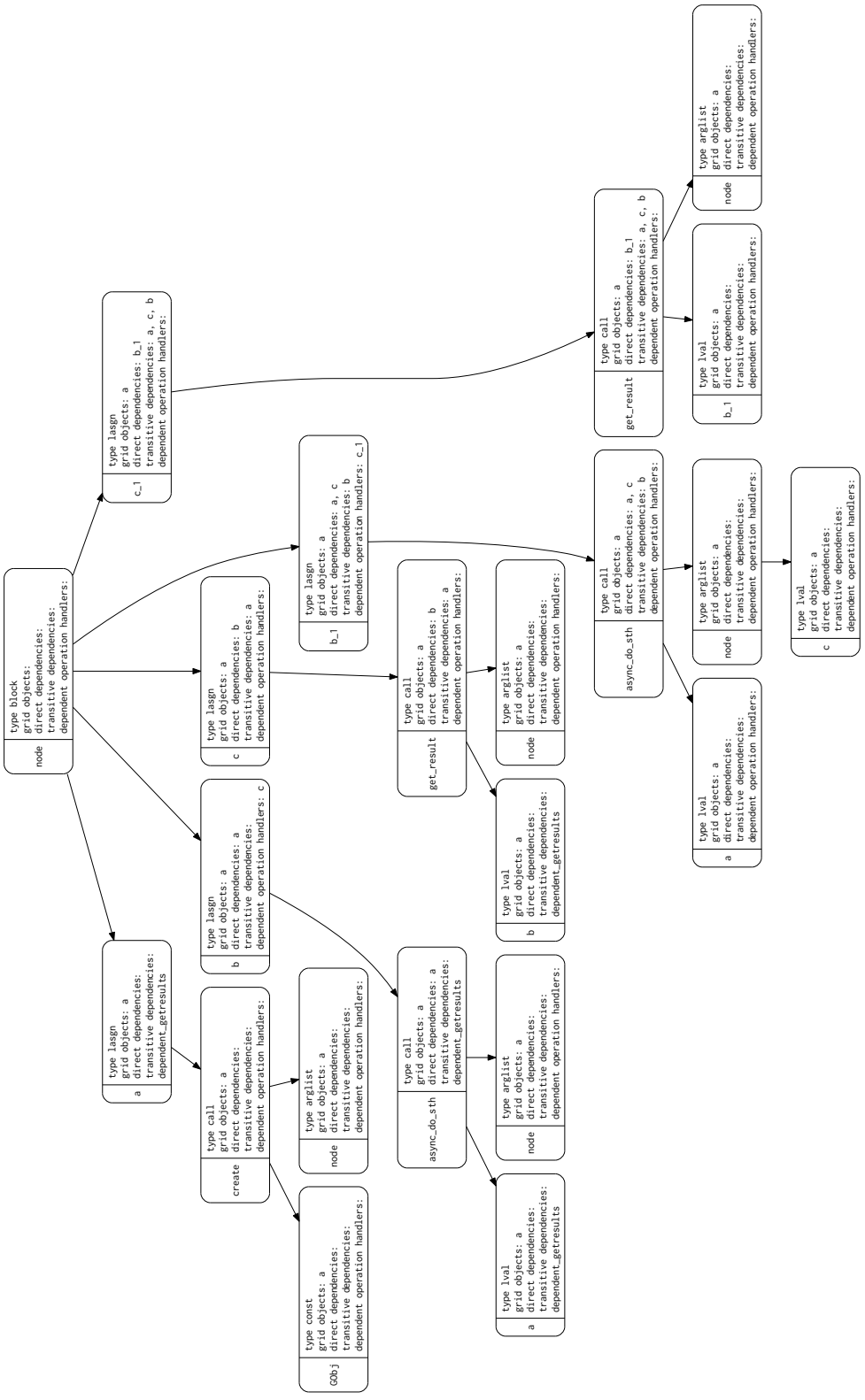


Figure 3.16: Internal representation with resolved reassignment issue

3.2.5 Finding dependencies from blocks - analyzing control flow

The cases presented in the previous sections are just examples of simplest sequential instructions. The Ruby Language (as well as many other imperative programming languages) gives us possibility to express complex programs in hierarchical structure by introducing blocks. Garbling Ruby Language from blocks might be unacceptable for majority of users since they would not be able to use `if` and `loop` statements (particularly in Ruby Language, where blocks are very important structures since they can be defined and managed independently from other structures - many Ruby products bases on that feature, e.g. Ruby on Rails web framework).

Description of the issue

The issue arises when in block statement there is a reassignment of already initialized variable. Thus, this issue is similar to the reassignment issue 3.2.4 with a difference that a block can be executed many times (if it is a `loop` statement) or it can be executed conditionally (`if` statement case). It entails some consequences - dependencies might be fulfilled only when some condition, in runtime, returns positive value or can be looped by unspecified (in preexecution time) numbers of times.

There are several Ruby constructions which has to be considered to analyze control flow: `if`, `iter`, `while` and `block` and dependencies from each of them has to be considered in a different way.

Solution of dependencies between blocks issue

The solution has to include finding dependencies between operations in block statements and operations executed before and after in following conditions:

- block belongs to `if` statement - the condition on which operations in this block depend has to be extracted from Ruby source code
- block belongs to some `loop` operation - it is extended case of previous one since except that `loop` statements are invoked on some condition, some operations can be calculated basing on previous iterations 3.17

```
a = 1
for i in 2..10
  a = a * i
end
puts a
```

Figure 3.17: This code calculates factorial of 10 and it is an example of looped dependencies, each iteration uses values produced be previous one.

If and while loop conditions. Conditions are contained in S-expressions as it is shown in figure 3.18 and they can be extracted.

```
if a == 2
  b = 1
end
```

```
s(:if,
  s(:call,
    s(:call, nil, :a, s(:arglist)),
    :==,
    s(:arglist, s(:lit, 2))),
  s(:lasgn, :b, s(:lit, 1)), nil)
```

Figure 3.18: If statement and its S-expression. Condition can be extracted by analyzing lines from 2 to 5 of Sexp.

Eliminate reassignments. The procedure originally described in 3.2.4 has to be extended to gather information about blocks and variable scopes.

Internal representation is traversed as previously but name changes of sons of `block`, `if`, `while` and `iter` block types are noticed. Name changes are performed only on nodes which are lower or on the right of given reassignment so this additional information about variables which were overwritten in blocks are necessary to track dependencies between operations from block and operations which takes place afterwords.

Find dependencies in if, while, iter and loop statements. Breadth-first traversal is applied to the internal representation. For each node:

- analyzer checks all direct and transitive dependencies if any of them was overwritten in block (variable names which are overwritten in blocks was gathered using a method described in 3.2.5), if so, node which overwritten dependency is added as new dependency to current node,
- analyzer checks if current node is one of types: `if`, `iter` or `while`, if so, it is added to the corresponding list, thus in next iterations it will be known if particular node is nested in some statements and it will provide information to distinguish between many `if`, `iter` and `while` nodes

After this step, procedure of detecting transitive dependencies 3.2.3 is invoked second time to supplement whole dependency network.

Results for if statements

The example of `if` statement which illustrates current issue is shown in figure 3.19. Internal representation produced from this script with resolved dependencies from if statement body is shown in figure 3.20.

```
a = 1
b = 2
if a == 1
  b = a + 1
end
c = b
```

Figure 3.19: Simple example of dependencies from `if` statement block. When if condition is fulfilled, variable `c` is dependent on `a` and `b`, otherwise, `c` is dependent only from `b`.

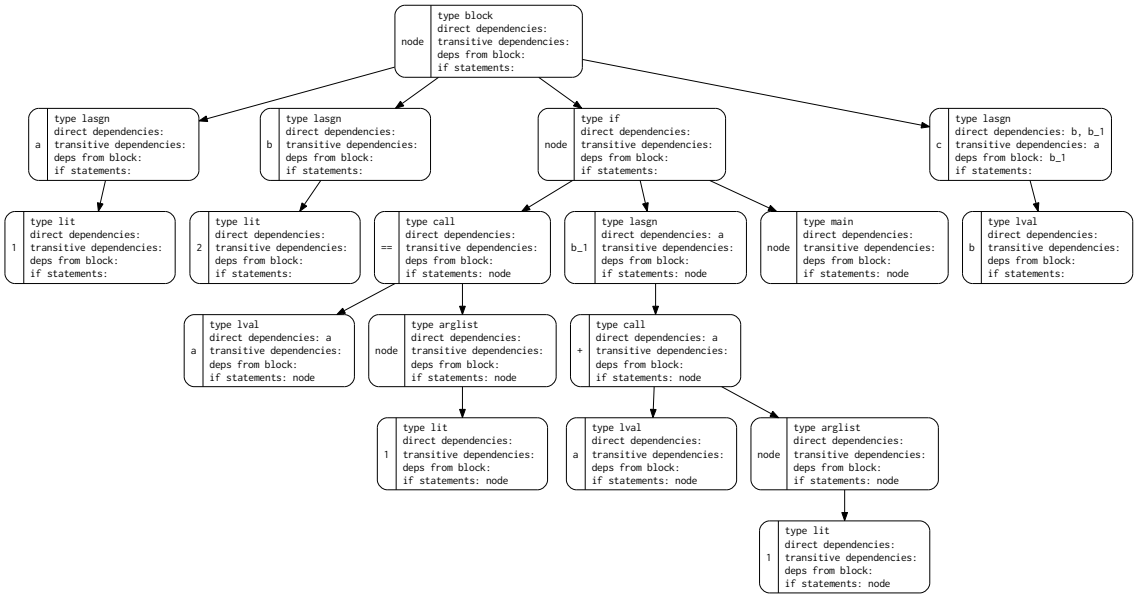


Figure 3.20: Processed internal representation of script from figure 3.19.

Results for loop statements

Figure 3.17 contains already discussed example of looped dependency. Internal representation built on this example with all dependencies resolved, is presented in figure 3.21.

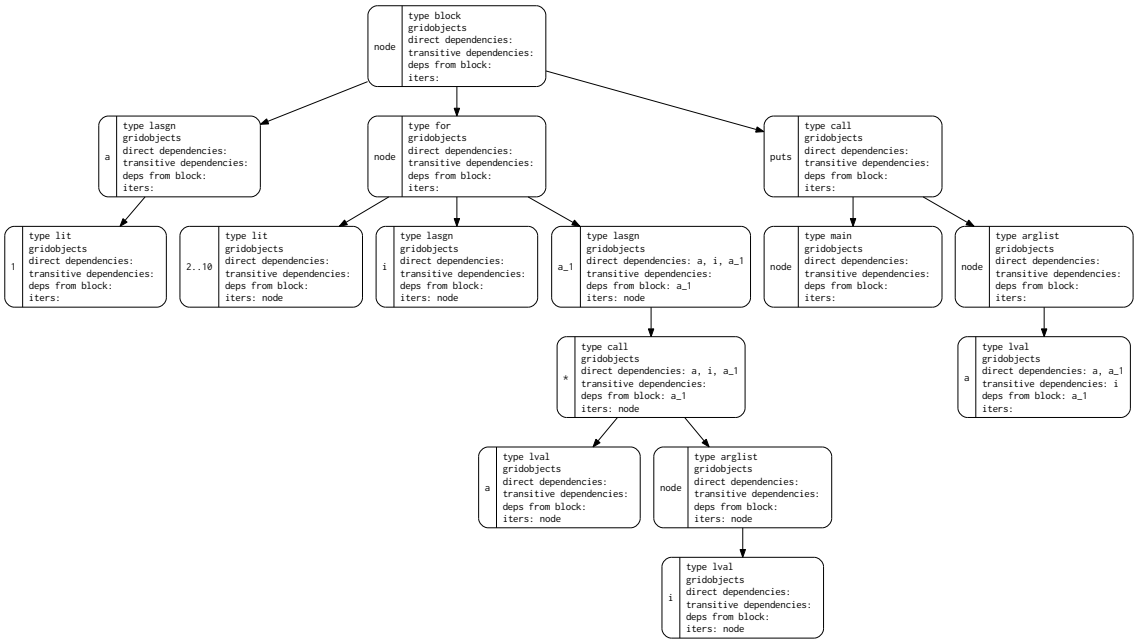


Figure 3.21: Processed internal representation of script from figure 3.17.

3.3 Summary

It was shown in this chapter that it is possible to acquire all information needed to transform Virolab scripts into workflows. The analyzing process was divided into small steps, solving these simple issues brought results that enabled opportunities for resolving complex problems, as it was shown in figure 3.3.

CHAPTER 4

Tool for application analysis

The implementation of analyzing process ended up creating a tool which allows to process Ruby scripts and produce all discussed data and graphs.

4.1 External tools

Only two external tools used to implement workflow builder for ViroLab scripts are parse tree[27] for source code analysis and GraphViz[28] for drawing graphs.

4.2 Architecture and class diagram of a developed tool

Class diagram included in figure 4.1 shows dependencies between classes and their methods.

- `DagEdge` is a class which keeps information about edges of workflow, used in `dag-tree` module.
- `DagNode` is a class which keeps information about nodes of workflow, used in `dag-tree` module.
- `Dag` is a class which contains procedures that build all dot graphs from internal representation - internal representation graph trees, variable dependencies, operation dependencies and workflows.

- `DagTree` is a class which handles internal representation of workflow. It is produced directly from experiment tree.
- `ExperimentNode` is a class of `experiment_tree` node. It includes node type read from s-expression and all information gathered during analyzing process.
- `ExperimentProcessor` is an extension of `SexpProcessor` class from parse tree[27] tool. It contains methods, one per s-expression type, e.g. `block`, `args`, `class`, `defn`, etc - in total 38. It produces internal representation from parsed Ruby code.
- `ExperimentTree` is a class which handles internal representation produced in `experiment_processor`. Moreover, it contains main methods used in analyzing process.

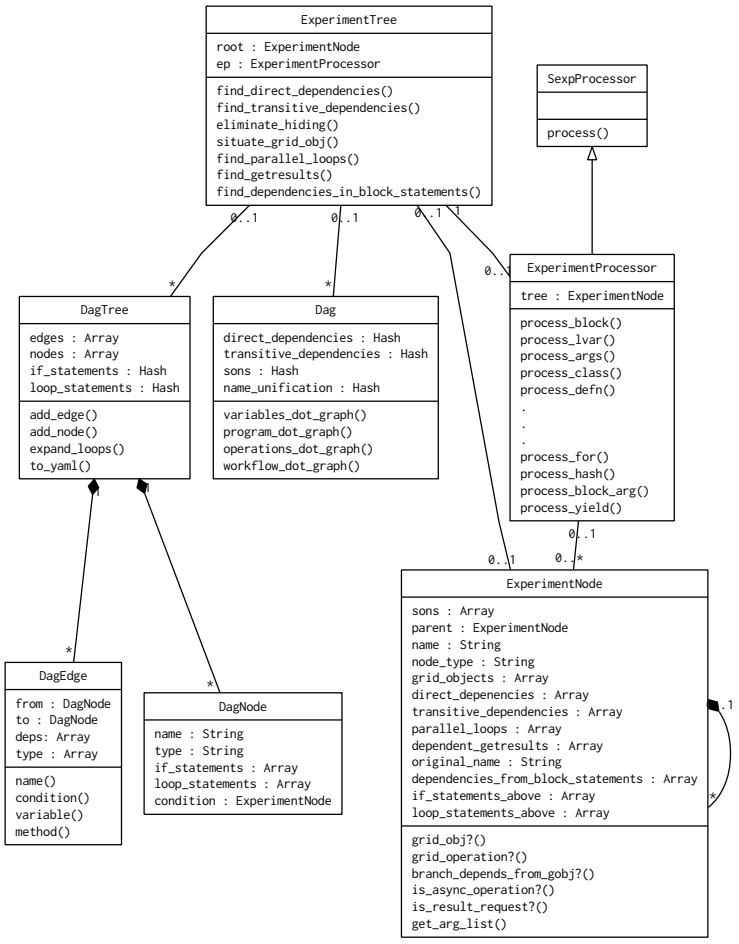


Figure 4.1: Class diagram presents all implemented modules. `SexpProcessor`

4.3 Usage description

Modules from previous section (4.2) are managed by small script called `workflow.rb`, it can be invoked as follows:

- `./workflow.rb input_script.rb sexp` creates S-expression representation of the given script (example: 3.5)
- `./workflow.rb input_script.rb variables` creates variables dependencies graph in dot format (example: 5.2a)
- `./workflow.rb input_script.rb program` creates internal representation graph in dot format (example: 3.21)
- `./workflow.rb input_script.rb operations` creates operations dependencies graph in dot format (example 5.5b)
- `./workflow.rb input_script.rb workflow` creates workflow diagram (example 5.6)

To automate typical routines, `Rakefile` was created. It is able to produce various configuration of `workflow.rb` script results and produce graphical outputs for `.dot` files. Processing file sets, directory trees are automated.

4.4 Workflow description language based on YAML

During the work on this thesis, there was a need to keep workflows in a permanent storage to enable easy access of complete workflows from application. There was already a possibility to export workflows to `dot` files but they are not easy to parse from any programming language. To fulfill this requirement *YAML*[29] language was chosen (it is easy to read by a human and it has import/export tools in majority of programming languages).

Figure 4.2 includes exported workflow to YAML representation. It contains all information which are gathered in a graphical workflow representation. In detail, there is a list of nodes and edges, both identified by numbers. Each node has a list of incoming and outgoing edges, type and name. Edges contain only a type and a list of dependencies in (`deps` segment).

```
---
nodes:
  3269977186983927969:
    input: []

    output:
    - 1227372557622029165
    name: "b = a.async_do_sth()"
    type: job
  -940644704559776611:
    input:
    - 1227372557622029165
    output: []

    name: "d = a.async_do_sth(c)"
    type: job
edges:
  1227372557622029165:
    type: non_expl
    deps: c
```

Figure 4.2: Sequence pattern in yaml representation.

4.5 Summary

Developed tool is adapted to quick and easy generating workflows and intermediate graphs (which are used for debugging) for particular scripts or whole directory branches. It works well with Ruby 1.8.7, 1.9.2 and JRuby 1.6.0.

CHAPTER 5

Transformation of scripts to workflows

In this chapter, it will be presented how to build workflows for ViroLab scripts using data collected during the analysis process (3). Moreover created workflows are going to be evaluated whether they are able to resolve dependencies in complex Ruby programming statements and present well known workflow patterns (2.4), benchmark workflows (2.2) and already existing ViroLab scrips correctly.

5.1 Building workflows

All data required to build workflow are collected during analyzing process which was described in chapter 3.

Internal representation is traversed and all nodes which are not asynchronous operation on grid object are filtered. Remaining nodes are grouped into:

- pairs of explicit dependencies
- pairs of transitive dependencies
- pairs of dependencies from `if` or `loop` statements

Additionally, nodes with equal names are replaced with its first appearance which makes it a graph of grid operations dependencies in adjacency list representation. More details of the workflow building process are in section 5.2.1.

5.2 Supporting workflow patterns

In section 2.4 are presented basic control-flow patterns. To prove that these aspects of process-control can be implemented in ViroLab scripts, a sample workflow will be created for each pattern.

5.2.1 Sequence

Listing in figure 5.1 is a implementation of the sequence workflow pattern[20] specification previously introduced in chapter 2.4.1.

```
a = GObj.create
b = a.async_do_sth("")
c = b.get_result
d = a.async_do_sth(c)
e = d.get_result
```

Figure 5.1: A script above contains two activities, where the second one uses values returned from first activity. This workflow fulfills requirements of sequence pattern described in chapter 2.4.1.

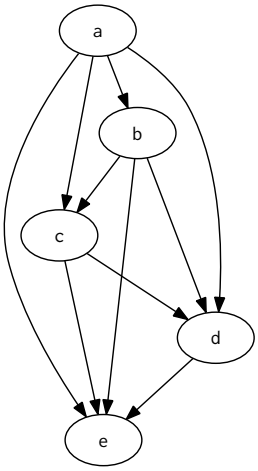
Building workflow

To make workflow building process more clear to understand, intermediate forms between internal representation and workflow representation was created.

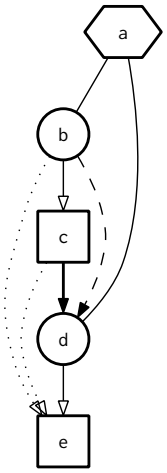
As it was mentioned in chapter 3, internal representation of the script contains for each node direct dependencies, transitive dependencies, information about `if` and `loop` statements, if the node is a grid operation, if the node is operation handler, grid objects scopes etc.

First intermediate. First intermediate graph form (figure 5.2a) is created to show dependencies between assignments. Procedure which creates this kind of graph simply takes all assignments from internal representation, filters all assignments that do not contain grid operation, grid object creation or result request on grid operation handler and joins remaining assignments with arrows. If assignments depend on each other without distinction between dependency types and ignore all other information. Assignments in the graph are named after variables which are created in particular operation. Arrows can be read as “is necessary to calculate variable”, which on particular relation between node `c` and `d` is formulated as “`c` is necessary to calculate variable `d`”.

Second intermediate. Second intermediate graph form (figure 5.2b) contains more information from internal script representation. Now there is a graphical distinguish between different dependency types and assignments of different operations 5.2b.



(a) Variable dependencies - first intermediate graph which shows dependencies between assignments in script from figure 5.1. If the node name is `d` it means that it represents assignment `d = a.async_do_sth(c)`. Arrow from node `c` to `d` means that node `d` depends on node `c`.



(b) Operation dependencies - second intermediate graph for script 5.1. Grid operations are represented as circles, result requests are squares and grid objects are hexagons. Direct dependencies are represented by solid lines (dependency arrow which points on grid operation is bold and has filled arrow), transitive dependencies has dotted lines (arrows pointing grid operations are dashed and also have filled arrows). Edges without arrow shows on which grid objects operation is invoked.

Figure 5.2: Sequence pattern intermediate graphs.

Workflow

Graph for sequence pattern can be finally made basing on second intermediate graph with following transformations:

- grid objects are removed,
- grid operations handlers nodes are replaced by labels on edges which indicate data flow,
- transitive dependencies are removed except for the cases where there is not a direct dependency (there is no label on the edge).

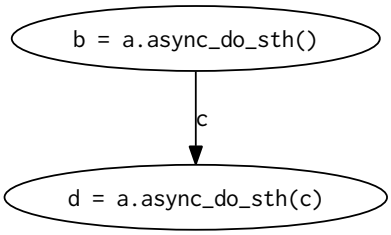


Figure 5.3: Final workflow representation of the sequence pattern 2.4.1 implemented as Virolab application 5.1.

5.2.2 Parallel split

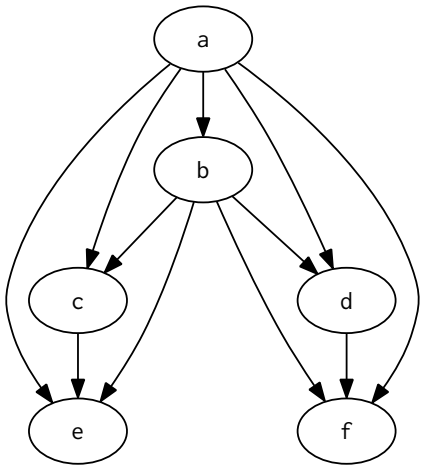
The parallel split workflow pattern[20] (introduces in section 2.4.2) is a point where one branch splits in multiple of branches. In Virolab script this situation can appear when after one grid operation, multiple different grid operations are invoked using values produced by the first one as it is shown in figure 5.4.

```
a = GObj.create
b = a.async_do_sth
c = b.get_result
d = b.get_result
e = a.async_do_sth(c)
f = a.async_do_sth(d)
```

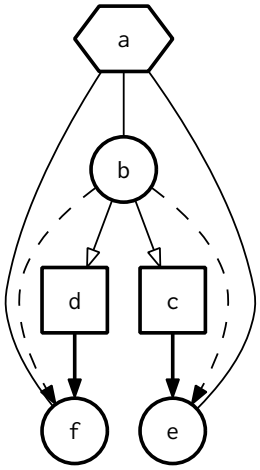
Figure 5.4: There are three activities in the script above. The last two (`e` and `f`) which are executed in parallel, use value obtained as a result of the first one (`b`). It fulfills requirements of parallel split pattern.

Building workflow

Intermediate steps of building workflow representation split parallel split pattern are presented in figures 5.5a and 5.5.



(a) Variable dependencies of parallel split pattern implementation. Values of variable `c` and `d` depend on value of variable `b` but they are independent on each other.



(b) Operation dependencies of parallel split pattern implementation. Operation represented by nodes `e` and `f` are independent but this graph representation does not consider control flow structures thus it can not be said that these operations are executed in parallel.

Figure 5.5: Parallel split pattern intermediate graphs.

Workflow

Workflow representation of parallel split pattern is shown in figure 5.6. This workflow is a case of implicit representation since it is realized by two branches which are executed without any condition, outgoing from one node.

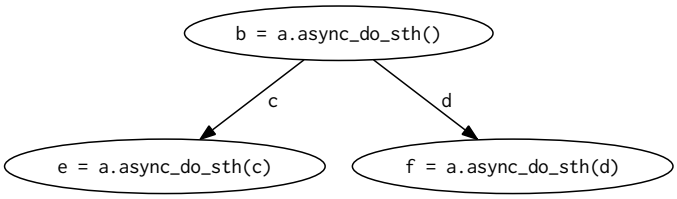


Figure 5.6: Workflow of parallel split pattern 2.4.2 built from Virolab implementation 5.4. Two grid operations `e = a.async_do_sth(c)` and `f = a.async_do_sth(d)` which have common dependency - `b = a.async_do_sth()`.

5.2.3 Synchronization

The synchronization workflow pattern[20] which defines (2.4.3) a point in the workflow where more than one thread of control join into one. This case appears in Virolab appli-

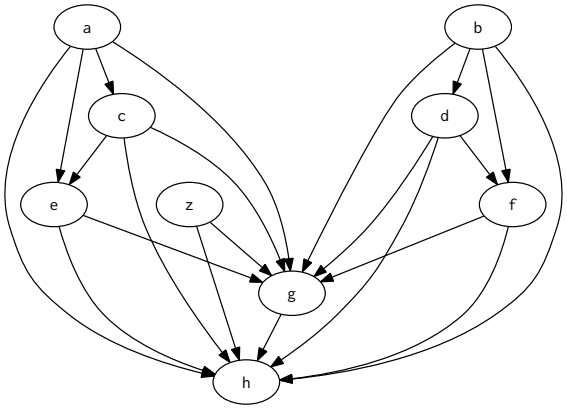
cation when one grid operation depends on many other grid operations - sample implementation which includes this kind of relation is shown in figure 5.7.

```
a = GObj.create
b = GObj.create
z = GObj.create
c = a.async_do_sth
d = b.async_do_sth
e = c.get_result
f = d.get_result
g = z.async_do_sth(e, f)
h = g.get_result
```

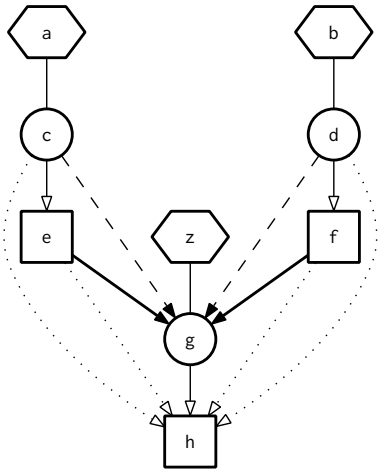
Figure 5.7: There are three grid operations used invoked by the above script. The first two of them (represented by `c` and `d` operation handlers) produce values which are used to calculate result of operation `g` which makes this operation dependent on both previous operations.

Building workflow

Both intermediate graphs presenting variable and operation dependencies are included in figures 5.8a and 5.8b.



(a) Variable dependencies of synchronization workflow pattern implementation. A value of variable `h` is calculated using (not directly) all other variables in a script.



(b) Operation dependencies of workflow parallel split pattern implementation. Operation `g` depends on operations `c` and `d`. Each operation is from different grid object instance (`a`, `b` and `z`).

Figure 5.8: Synchronization pattern intermediate graphs.

Workflow

Workflow representation of synchronize pattern is included in figure 5.9. Built workflow can be classified as implicit representation since instead of particular structure, synchronize pattern is represented by edges which coming to one node - the point of synchronization as it is shown in figure 5.9.

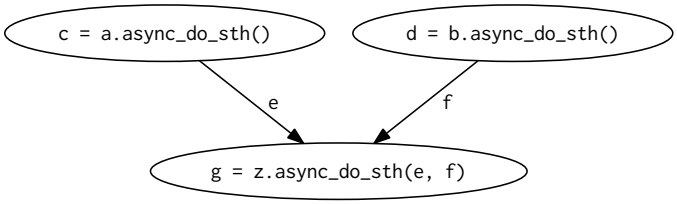


Figure 5.9: Workflow of synchronization pattern 2.4.3 built from Virolab implementation 5.7. Operation `g = z.async_do_sth(e, f)` depends on two different grid operations `c = a.async_do_sth()` and `d = b.async_do_sth()`.

5.2.4 Exclusive choice

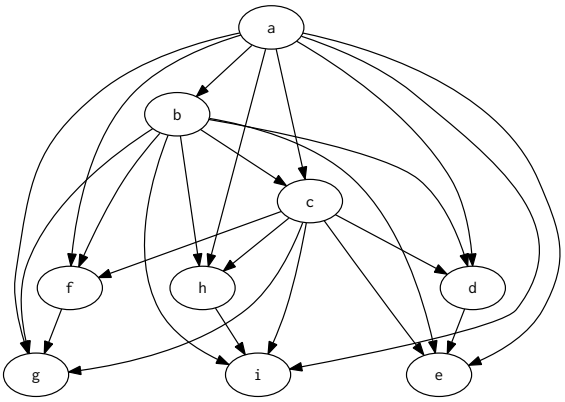
The exclusive choice workflow pattern[20] defines in section 2.4.4 a point in workflow where depending on a particular condition, one outgoing control flow branch is chosen. In Virolab application and as well in Ruby code, operations which are executed under certain decision are commonly located in `if` statement. Figure 5.10 includes Virolab application in which grid operations executions are determined by conditions of `if` statements.

```
1   a = GObj.create
2   b = a.async_do_sth
3   c = b.get_result
4   if c == true
5     d = a.async_do_sth(c)
6     e = d.get_result
7   elsif c == false
8     f = a.async_do_sth(c)
9     g = f.get_result
10  else
11    h = a.async_do_sth(c)
12    i = h.get_result
13  end
```

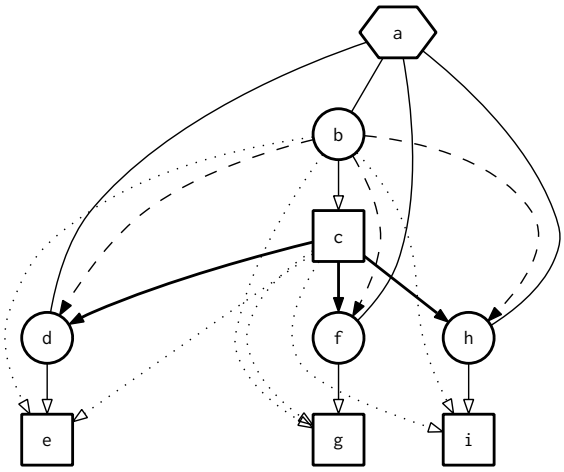
Figure 5.10: Exclusive choice workflow pattern implementation. First grid operation `b = a.async_do_sth()` is always executed but following operations depend on conditions of if statements in lines 4 and 7.

Building workflow

Both intermediate graph representations needed to build exclusive choice workflow pattern from its Virolab implementation are included in figures 5.11a and 5.11b.



(a) Variable dependencies of exclusive choice pattern implementation. Three groups of nodes `f` and `g`, `h` and `g`, `d` and `e` do not have any arrows between each other.



(b) Operation dependencies of exclusive choice pattern implementation. Three operations `d`, `f` and `h` are independent but because of lack of control flow structures in this graph representation, it can not be determined if these grid operation can be executed in parallel.

Figure 5.11: Exclusive choice pattern intermediate graphs.

Workflow

Figure 5.12 includes workflow representation of exclusive choice pattern built from its Virolab implementation. This workflow can be considered as explicit representation of exclusive choice since there is a special construct which determines which outgoing branch is chosen.

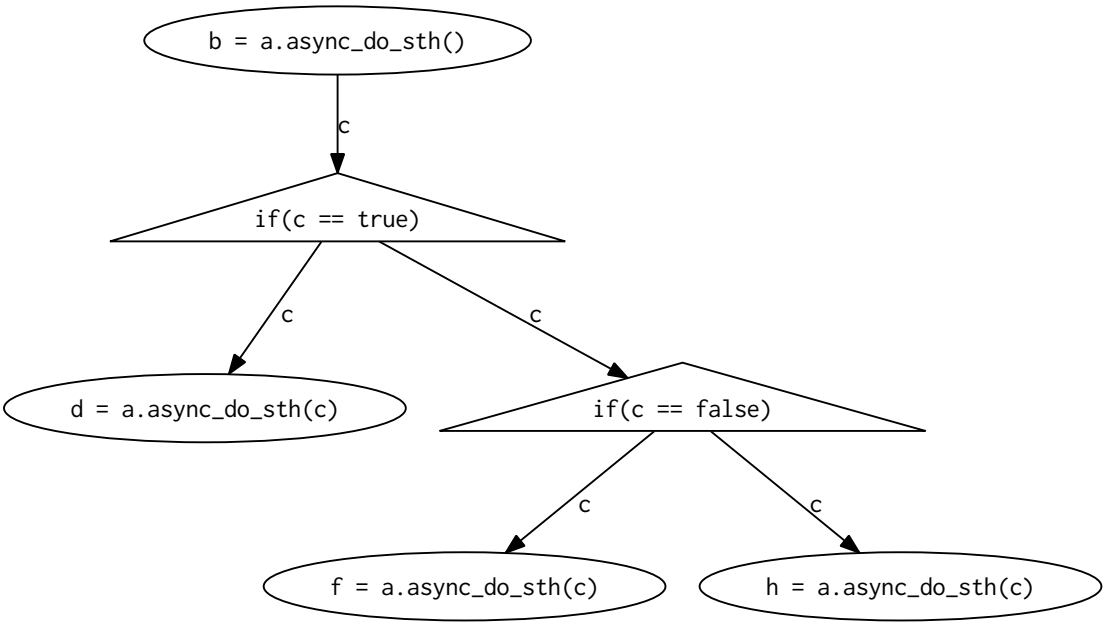


Figure 5.12: Exclusive choice workflow pattern 2.4.4 built from its Virolab implementation 5.10. Triangle shaped nodes stand for condition, when control from goes through this kind of node, only one outgoing edge is chosen depending on conditions expression and its evaluation.

5.3 Statements

As it was emphasized in chapter 3, analyzing Ruby language constructs like `loops` and `if` statements can be very hard. In this section, there are created workflow representations for Virolab applications which includes non-trivial Ruby constructs previously mentioned in section about analyzing process (3).

5.3.1 Reassignment

The reassignment issue (discussed in section 3.2.4) occurs when new value is assigned to already used variable label. Virolab application where variable names are reused is included in figure 5.13, its workflow is shown in figure 5.14.

```
1    a = GObj.create
2    b = a.async_do_sth
3    c = b.get_result
4    b = a.async_do_sth(c)
5    c = b.get_result
6    b = a.async_do_sth(c)
7    c = b.get_result
```

Figure 5.13: Virolab application with reassignment issue. Label `b` keeps different values in lines 2, 4 and 6. Label `c` which determine dependencies between grid operations is also reused in lines 3, 5 and 7.

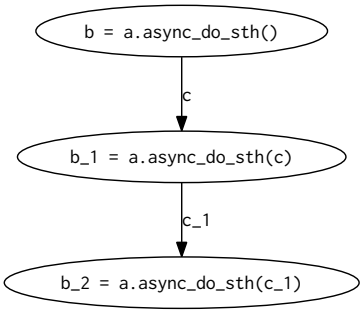


Figure 5.14: In workflow built for application with reassignment issue all operation dependencies were correctly determined. Distinguish of various values kept by label `b` and `c` were achieved by adding a suffix to original label. Re-usage of label `b` were renamed into `b_1` and `b_2` and re-usage of variable `c` were named `c_1`

5.3.2 Loop

In section 3.2.5, there is a description of a problem when grid operation which is placed in a `loop` block, has dependencies on operations before the loop or operations after loop are dependent on this particular operation. In figure 5.15 there is a script which shows both cases.

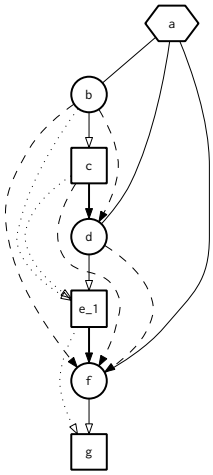
```
a = GObj.create

b = a.async_do_sth
c = b.get_result

e = nil
loop do
  d = a.async_do_sth(c)
  e = d.get_result
end
f = a.async_do_sth(e)
g = f.get_result
```

Figure 5.15: Virolab application using a `loop` statement, there is a grid operation (`d = a.async_do_sth(c)`) which has dependency on operation accured before the loop (`b = a.async_do_sth`). This is also an example of operation (`f = a.async_do_sth(e)`) which has dependency on operation which is placed in the loop statement (`d = a.async_do_sth(c)`).

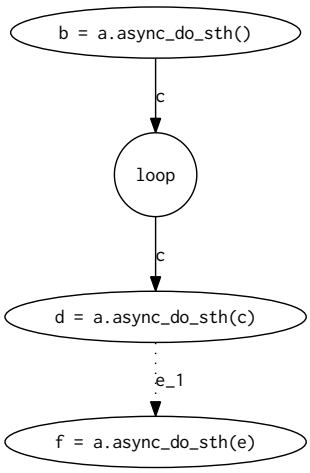
There are three grid operations and they are dependent on each other in a sequential order. It can be predicted then, that operation dependencies might be similar to sequence pattern 5.2.1 and because of double usage of the same variable label (`c` and `d`) it might be also similar to reassignment graph 5.3.1. The operations dependencies are presented in figure 5.16a.

From now, loop statement has its representation in workflow diagrams as circle with label `loop`. Dependencies between operations from the loop with other operations are represented as follows:

- operation before loop and operation in loop: from first operation arrow goes to `loop` node and then directly to the targeting operation
- operation in loop and operation after loop: there is only one arrow as in normal case but it is dotted



(a) Operation dependencies based on 5.15 script. Two lines of sequential process can be noticed, first one is `b -> d_1 -> e` and the second one omit `d_1`: `b -> e`.



(b) Workflow graph built from script with loop statement 5.15. There are two new elements, first one is `loop` node which indicate loop entrance and dotted line - loop exit.

Figure 5.16: Graphs created for Virolab application with loop statement.

In the graph from figure 5.16b, there is only one node made for operation `d = a.async_do_sth(c)`. If it was predicted that this operation is invoked many times, there can be added new nodes for `d = a.async_do_sth(c)` operation additional instances as it is shown in figure 5.17.

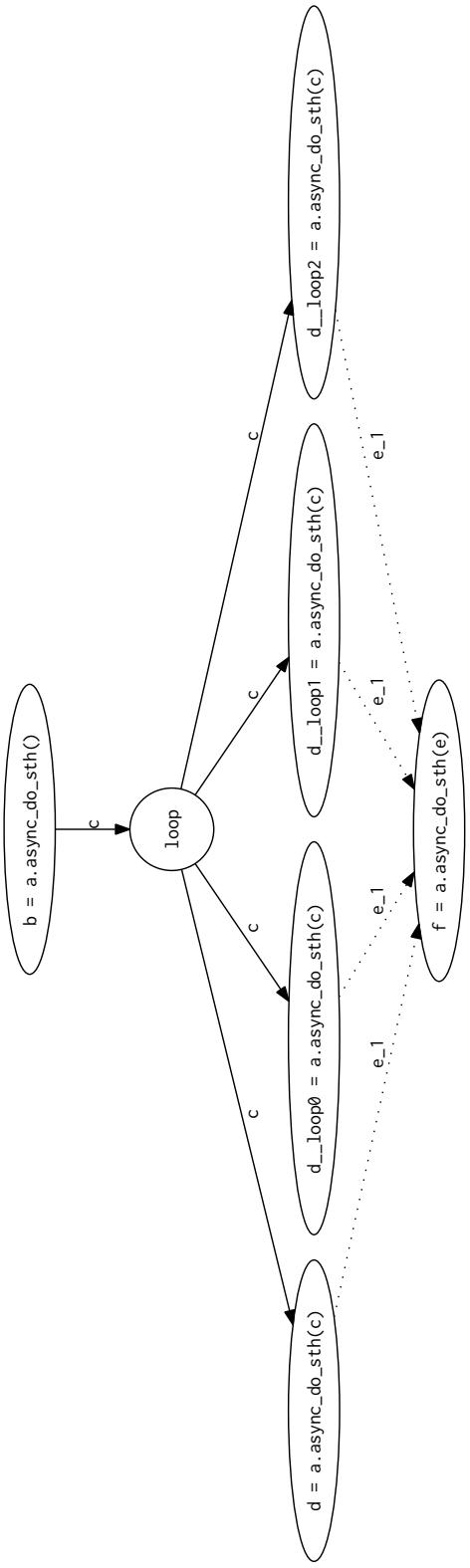


Figure 5.17: Expanded workflow from figure 5.16b by additional operation instances which can be created by several loop iteration.

5.3.3 Condition

The other complex example of dependencies between blocks previously discussed in section 3.2.5 is a `if` statement case. Script containing if statement with several branches is shown in figure 5.18. Operation from some branches have different dependencies than the others.

```
a = GObj.create

b1 = a.async_do_sth
c1 = b1.get_result
b2 = a.async_do_sth
c2 = b2.get_result

d = 0
if 0 == 2
  d = a.async_do_sth(c1)
elsif 1 == 2
  d = a.async_do_sth_else(c1)
else
  d = a.async_do_sth_else2(c2)
end
e = d.get_result
f = a.async_do_sth(e)
g = f.get_result
```

Figure 5.18: Virolab application with `if` statement. Application is similar to exclusive choice pattern implementation - 5.10. It this case, there is a grid operation after `if` statement which uses value produced in `if` statement branches. Grid operations in different branches have arguments of different variables.

The complexity of this case is illustrated by operation dependencies graph in figure 5.19.

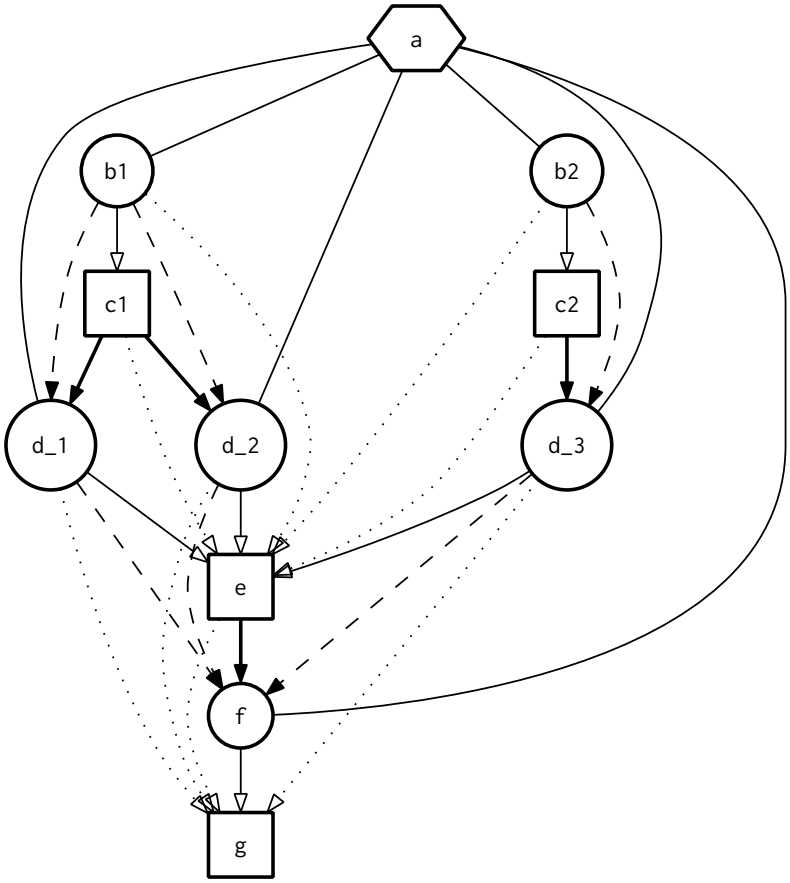


Figure 5.19: Operations dependencies for script 5.18. Two branches which are dependent on `c1` variable and third `if` branch which is dependent on `c2` can be clearly remarked.

In a workflow graph 5.20 there is a node in a shape of triangle previously introduced in exclusive choice workflow pattern - 5.2.4. Its role is similar to `loop` node, it marks beginning of the block plus its label is a condition definition and it has two output arrows for different results of condition evaluation. The role of dotted arrows corresponds with `loop` case - they are dependencies which goes out from `if` statement.

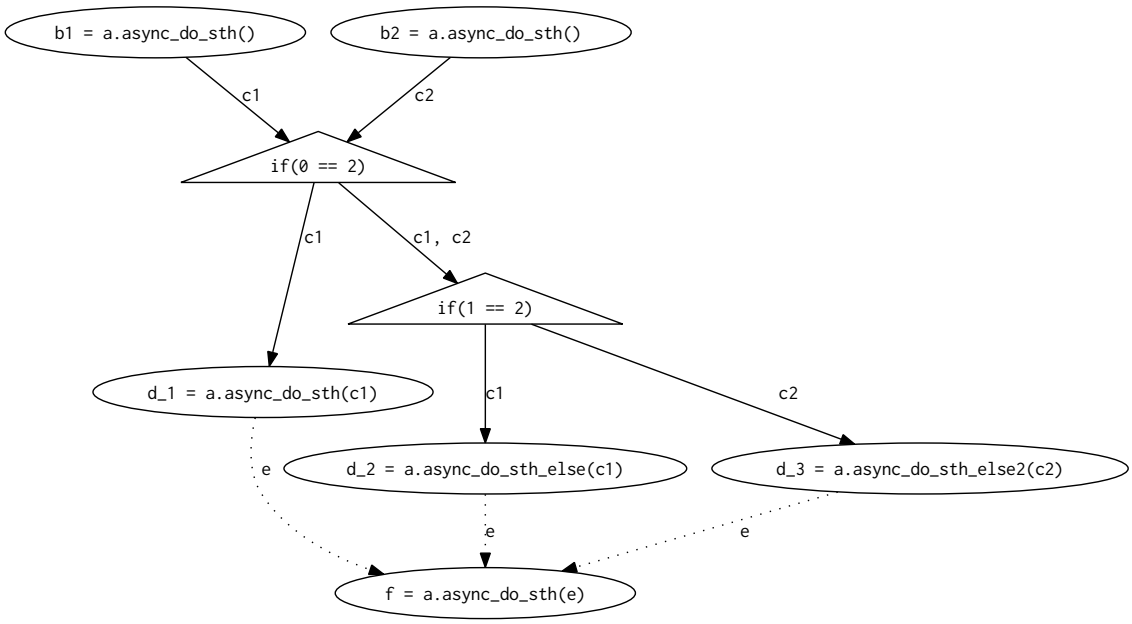


Figure 5.20: Workflow prepared from script 5.18. There are two conditions and three condition branches. Control flow stops on operation `f = a.async_do_sth(e)` which is placed outside the `if` blocks.

5.3.4 Iteration

Interesting aspect if iteration usage in experiments scripts is looped dependencies. It was mentioned in section 3.2.5 and illustrated with simple example in figure 3.17.

More complex example, with grid operations is shown in figure 5.21. Workflow created for this script 5.22 contains loop node and edges which indicate *loop exit* and *next iteration*.

```
a = GObj.create

b = a.async_do_sth
c = b.get_result

d = a.async_do_sth(c)
5.times do
  e = d.get_result
  f = a.async_do_sth(e)
  g = f.get_result
  d = a.async_do_sth(g)
end
i = d.get_result
j = a.async_do_sth(i)
k = j.get_result
```

Figure 5.21: Complex example of looped dependencies. In loop, operation `f = a.async_do_sth(e)` depends on operation `d = a.async_do_sth(c)` which is before the loop or, in following iterations, on operation `d = a.async_do_sth(c)`.

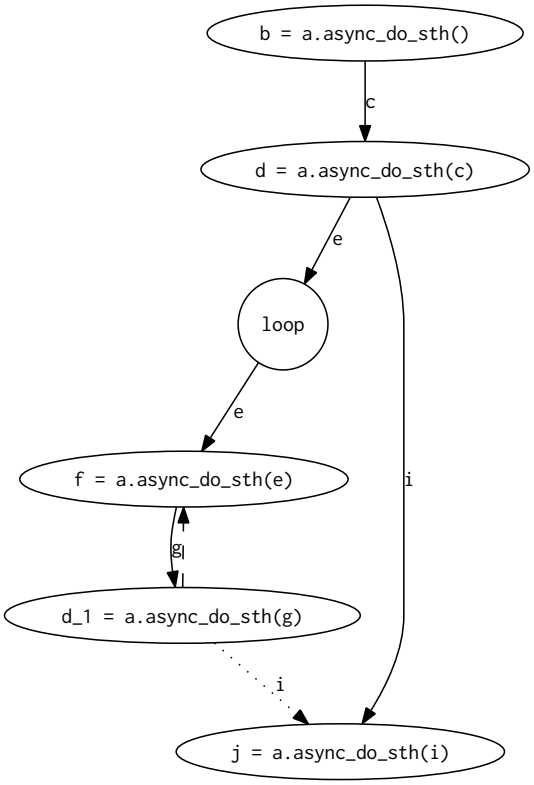


Figure 5.22: Workflow for experiment from figure 5.21. Dotted edge with label `i` (which means that there is dependency through variable `i`) represents dependency and exit from a loop. Dashed edge from node `d_1 = a.async_do_sth(g)` to `f = a.async_do_sth(e)` represents looped dependency and return to the beginning of loop after each iteration.

While expanding given iteration, looped dependencies are recognized, additional nodes are added and connected with proper dependencies as it is shown in figure 5.23.

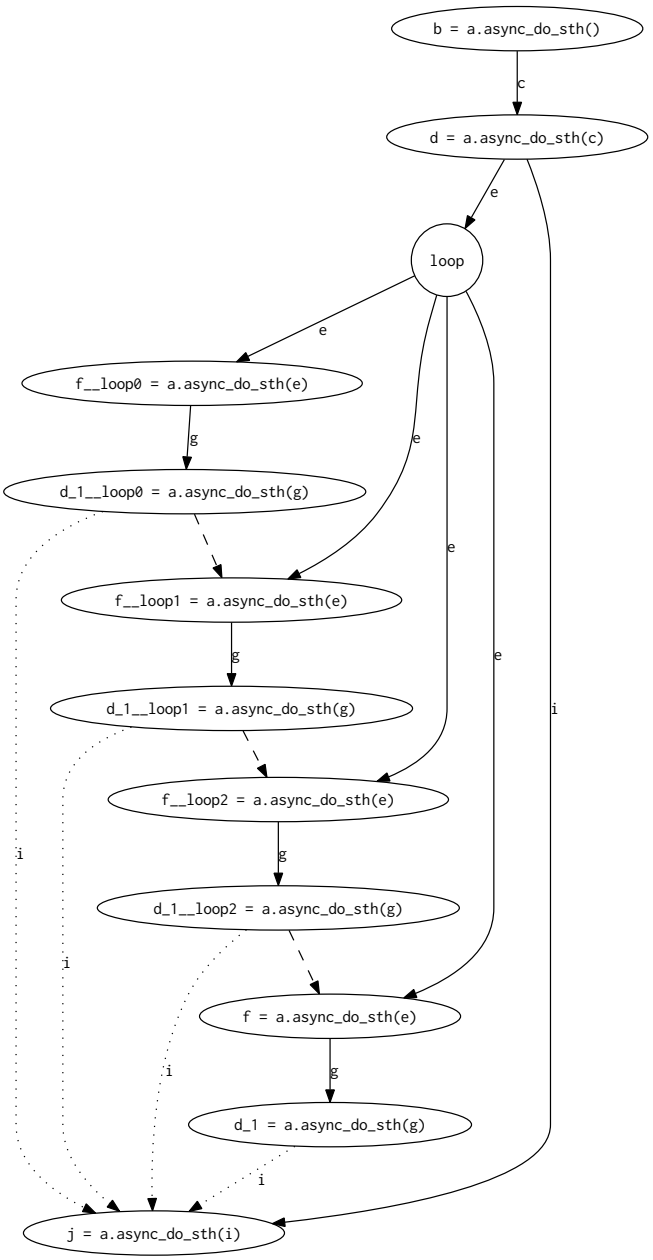


Figure 5.23: Expanded iteration. Pairs `f_* = a.async_do_sth(e)` and `d_* = a.async_do_sth(g)` which represents one iteration are connected with dashed arrow with its clones from different iterations.

5.3.5 Parallel for

Section 5.3.4 describes how the loop with dependencies among iterations can be expanded. Another workflow modification which can bring interesting results, especially with proper implementation, If operations in loop body do not modify values from outside of the loop and the result is not dependent on iterations order, particular iterations can be executed concurrently.

Motivation

The motivation of extending Virolab constructs by `parallel for` is to provide explicit method for defining which parts of Virolab application can be executed in parallel.

Design

Existing *GSEngine* libraries does not contain any routines which allow such kind of facilities. For the purpose of the thesis, following statement was introduced to provide parallel execution for given block.

```
P.parallelFor([1, 2, 3, 4, 5]) do |i|
  10.times { puts i; sleep 0.3 }
end
```

Figure 5.24: Example of parallel loop. Given block will be executed concurrently for each element of `Enumerable` argument (in this case - an array `[1, 2, 3, 4, 5]`).

Implementation Figure 5.25 contains sample implementation of *parallel for* statement. Each element from the `Enumerable` object `list` is processed in the separated thread.

```
class P
  def P.parallelFor(list)
    threads = []
    list.each_with_index do |e, i|
      threads << Thread.new(e, i) { |*args| yield args }
    end
    threads.each { |aThread|  aThread.join }
  end
end
```

Figure 5.25: Minimal implementation of the *parallel for* feature.

Usage

```
a = GObj.craete
b = a.async_do_sth
c = b.get_result
P.parallelFor([1, 2, 3, 4]) do |i|
  d = a.async_do_sth(c + i)
  e = d.get_result
end
```

Figure 5.26: The usage of `parallel for` statement. In execution, the block will be launched in parallel, each thread with corresponding argument from the array: `[1, 2, 3, 4]`.

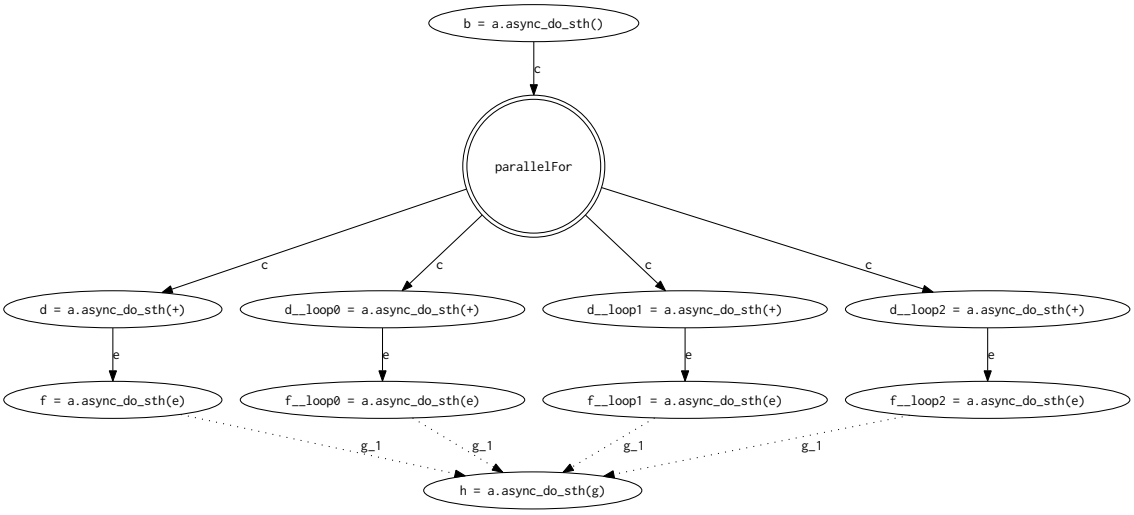


Figure 5.27: Workflow of a application with a `parallel for` statement. Independent branches between `parallelFor` node and dotted arrows will be executed in parallel.

5.4 Benchmark workflows

Benchmark workflows previously described in section 2.2 are well documented and widely used workflows in science. They are good examples of typical

5.4.1 Montage

Montage (2.2.1) toolkit, as it was mentioned before, is built from modules implemented in ANSI C. They can be used separately. Figure 5.28 presents a hypothetical ViroLab experiment which manages montage data and control flow.

In compare to the original Montage following modifications and simplifications has been applied:

- since presenting workflow tool is limited to information gathered from code parsing, it is impossible to resolve dependencies in objects collections,
- parallel loops in lines `14`, `20` and `32` would be replaced by normal loops if operation handlers were gathered in some collections, original dependencies between `mProjectPP` and `mDiffFit` would be reconstructed then.

Montage workflow is included in figure 5.29.

```
1     dataRepository = GObj.create("fotos")
2     mProjectPP = GObj.create("mProjectPP")
3     mDiffFit = GObj.create("mDiffFit")
4     mConcatDiff = GObj.create("mConcatDiff")
5     mBgModel = GObj.create("mBgModel")
6     mBackground = GObj.create("mBackground")
7     mImgTbl = GObj.create("mImgTbl")
8     pipeline = GObj.create("pipeline")
9
10    data_handler = dataRepository.async_load_fotos("12345")
11    data = data_handler.get_result
12
13    fotos_result = nil
14    P.parallelFor(data) do |f|
15      fotos_handler = mProjectPP.async_do_sth(data)
16      fotos_result = fotos_handler.get_result
17    end
18
19    diffFit_result = nil
20    P.parallelFor(fotos_result) do |f|
21      diffFit_handler = mDiffFit.async_do_sth(fotos_result)
22      diffFit_result = diffFit_handler.get_result
23    end
24
25    concatDiff_handler = mConcatDiff.async_do_sth(diffFit_result)
26    concatDiff_result = concatDiff_handler.get_result
27
28    bgModel_handler = mBgModel.async_do_sth(concatDiff_result)
29    bgModel_result = bgModel_handler.get_result
30
31    backgrounds_result = nil
32    P.parallelFor(fotos_result) do |f|
33      backgrounds_handler = mBackground.async_do_sth(fotos_result,
          bgModel_result)
34      backgrounds_result = backgrounds_handler.get_result
35    end
36
37    imgTbl_handler = mImgTbl.async_do_sth(backgrounds_result)
38    imgTbl_result = imgTbl_handler.get_result
39
40    pipeline_handler = pipeline.async_do_sth(imgTbl_result)
41    pipeline_result = pipeline_handler.get_result
```

Figure 5.28: Script shows hypothetical situation - how Montage would look like if it was written in Ruby and with usage of ViroLab Grid interface.

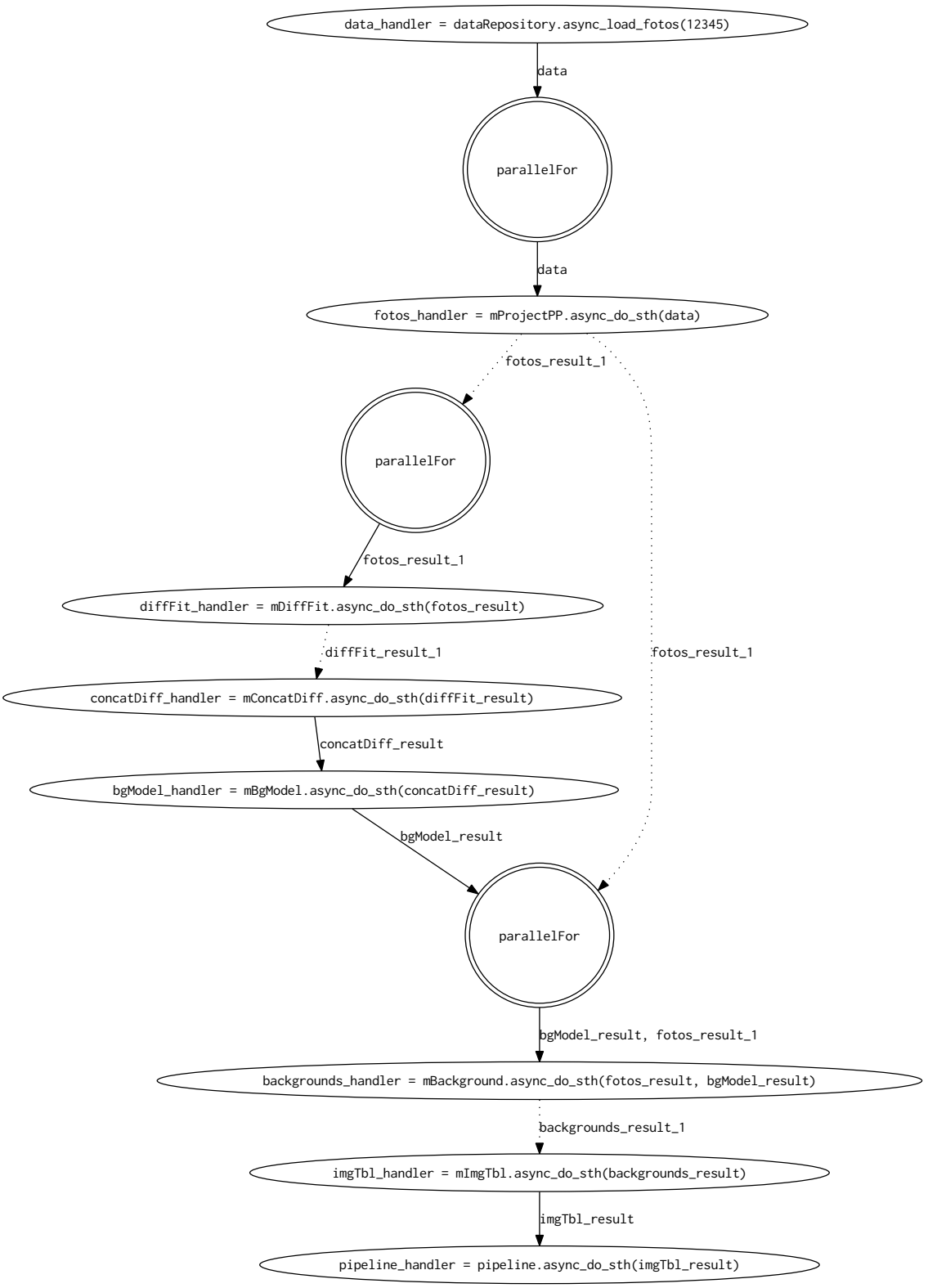


Figure 5.29: Montage workflow created for script 5.28. The original montage workflow generated from Pegasus framework is shown in figure 2.1.

5.4.2 CyberShake

As it was noticed in section 2.2.2, CyberShake has short critical path but with big amount of concurrent control branches. It might be reasonable to implement this application in such way that these branches would finish as soon as possible avoiding situations when one process would wait for another from different control-flow branch. The attempt to implement script which fulfill above requirement is contained in figure 5.30.

```
1     sgt = GObj.create("sgt")
2     synthesis = GObj.create("synthesis")
3     peak = GObj.create("peak")
4     zip_seis = GObj.create("zip_seis")
5     zip_psa = GObj.create("zip_psa")
6
7     input_handler = sgt.async_do_sth("data")
8     input = input_handler.get_result
9
10    s = []
11    P.parallelFor([1,2,3,4]) do |i|
12      s[i] = synthesis.async_do_sth(input[i])
13    end
14
15    p = []
16    s_result = []
17    P.parallelFor([1,2,3,4]) do |i|
18      s_result[i] = s[i].get_result
19      p[i] = peak.async_do_sth(s_result[i])
20    end
21
22    s_result = s.get_result
23    zs = zip_seis.async_do_sth(s_result)
24
25    p_result = p.get_result
26    zp = zip_psa.async_do_sth(p_result)
27
28    zp_result = zp.get_result
29    zs_result = zs.get_result
```

Figure 5.30: Script shows hypothetical situation - how CyberShake would look like if it was written in Ruby and with usage of ViroLab Grid interface.

In line 12, iteration loop creates tasks that create seismograms for given SGT taken from array s. Result requests on these newly created handlers are in line 18 - just before they are used in following line. In line 19, new tasks are created to calculate peak Spectral Accelerations. Line 22 - result request on already requested handler (returns immediately; although, this kind of operation needs modification of Ruby Array class).

Obtained result is passed into new grid operation in line 23. Line 25 contains result request identical to already discussed construct in line 22. Lines 23, 26, 28, 29 calculate final results of hazard curve and combined seismograms of ground motions.

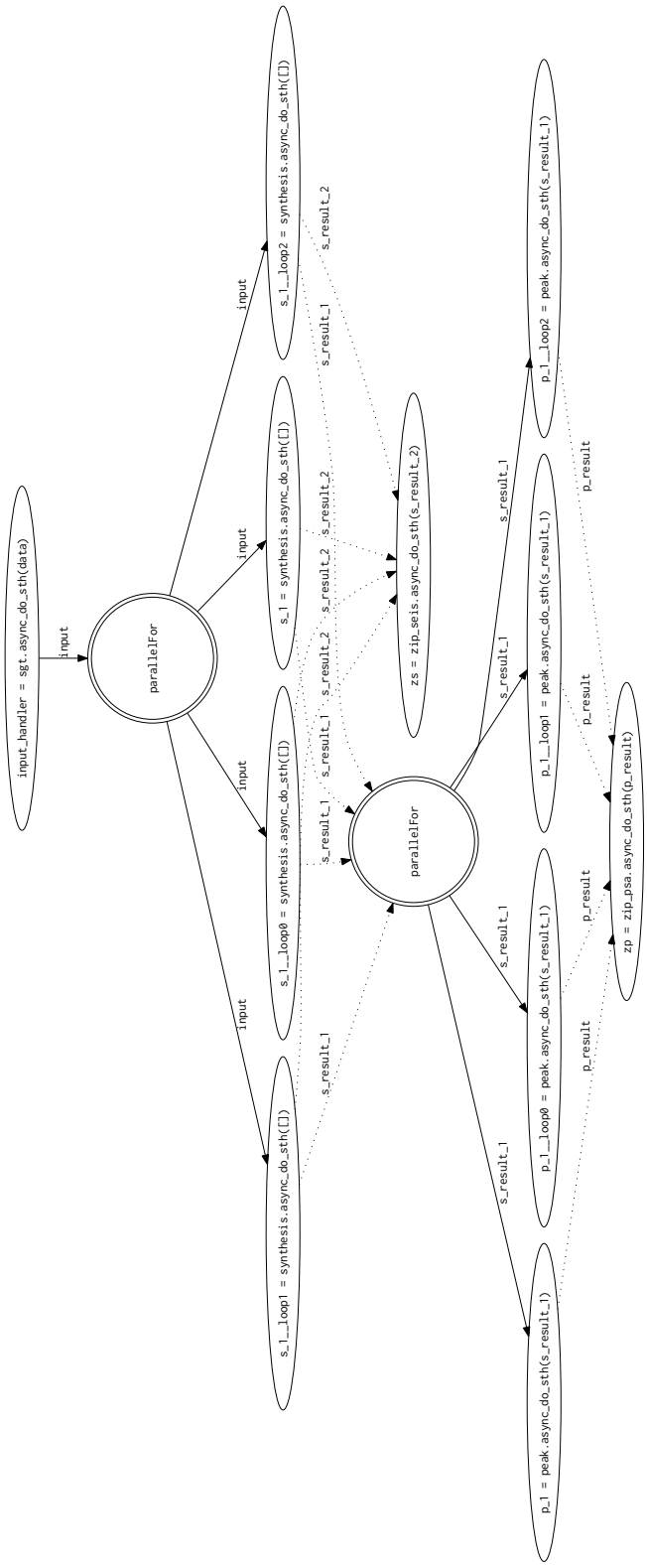


Figure 5.31: CyberShake workflow created from script 5.30. The original SyberShake workflow generated from Pegasus framework is shown in figure 2.2.

5.4.3 Epigenomics

Epigenomics 2.2.3 is an example of pipelined application, there are few independent long sequences of operations (in this sense it is opposite case than CyberShake). It is important to process independent branches of control flow in parallel.

```
1   epigenomics = GObj.create("epigenomics")
2
3   fqs_h = epigenomics.async_fastQSplit
4
5   map_h = []
6   P.parallelFor([1,2,3,4]) do |i|
7     fqs_res = fqs_h.get_result
8     fc_h = epigenomics.async_filterContams(fqs_res)
9     fc_res = fc_h.get_result
10    s2s_h = epigenomics.async_sol2sanger(fc_res)
11    s2s_res = s2s_h.get_result
12    f2b_h = epigenomics.async_fastq2bfq(s2s_res)
13    f2b_res = f2b_h.get_result
14    map_h[i] = epigenomics.async_map(f2b_res)
15  end
16
17  map_res = map_h.get_result
18  mm_h = epigenomics.async_mapMerge(map_res)
19  mm_res = mm_h.get_result
20  mi_h = epigenomics.async_maqIndex(mm_res)
21  mi_res = mi_h.get_result
22  pileup_h = epigenomics.async_pileup(mi_res)
23  pileup_res = pileup_h.get_result
```

Figure 5.32: Script shows hypothetical situation - how Epigenomics would look like if it was written in Ruby and with usage of ViroLab Grid interface.

Epigenomics workflow created from script 5.32 is presented in figure 5.33.

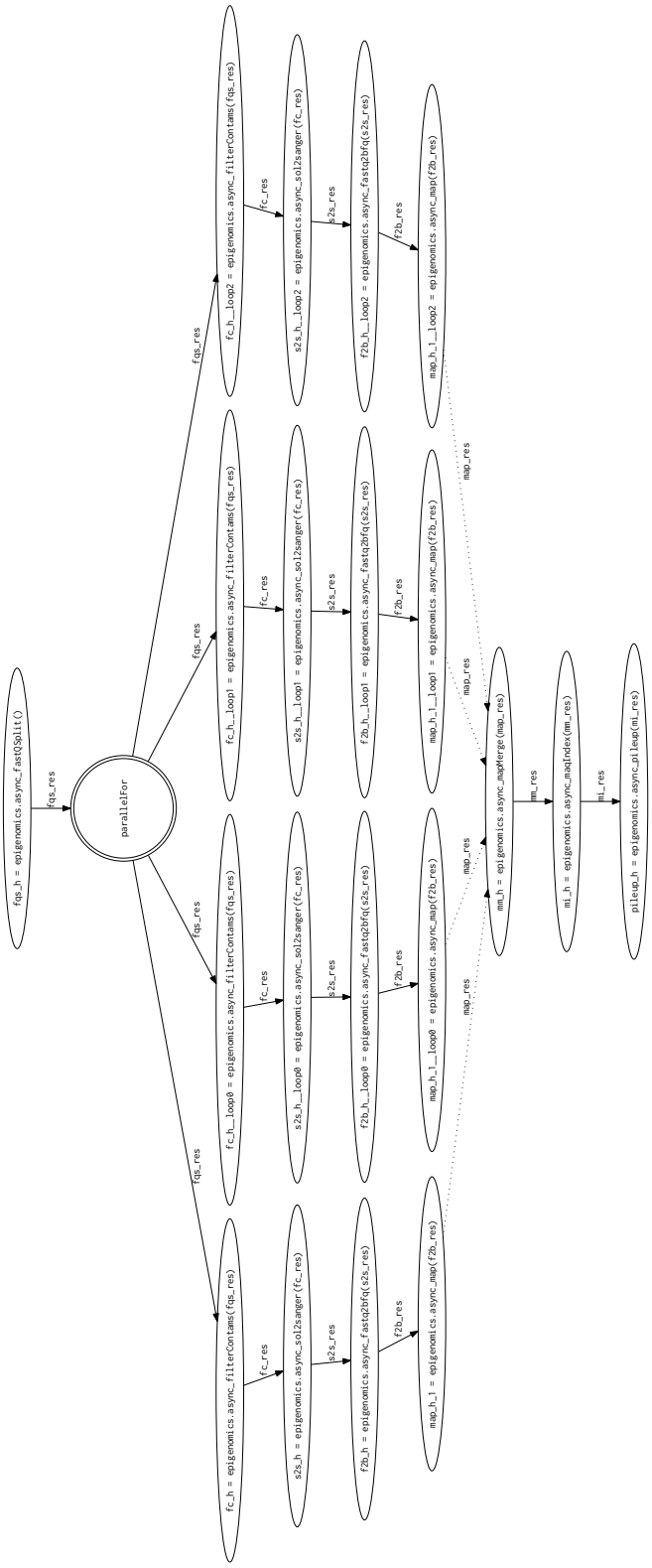


Figure 5.33: Epigenomics workflow created for script 5.32. The original Epigenomics workflow generated from Pegasus framework is shown in figure 2.3.

5.5 ViroLab workflows

In chapter 1, there were mentioned an experiment developed for Virolab. The application is shown in figure 1.1 and it can be downloaded directly from ViroLab website - [2].

For this special example, all intermediate graphs were created. It will be shown which parts of particular script causes problems and how to prevent them.

First intermediate Variable dependencies graph should contain grid objects, all grid operations and presents dependencies between them. On the graph 5.34 created for the script 1.1 all required nodes are present.

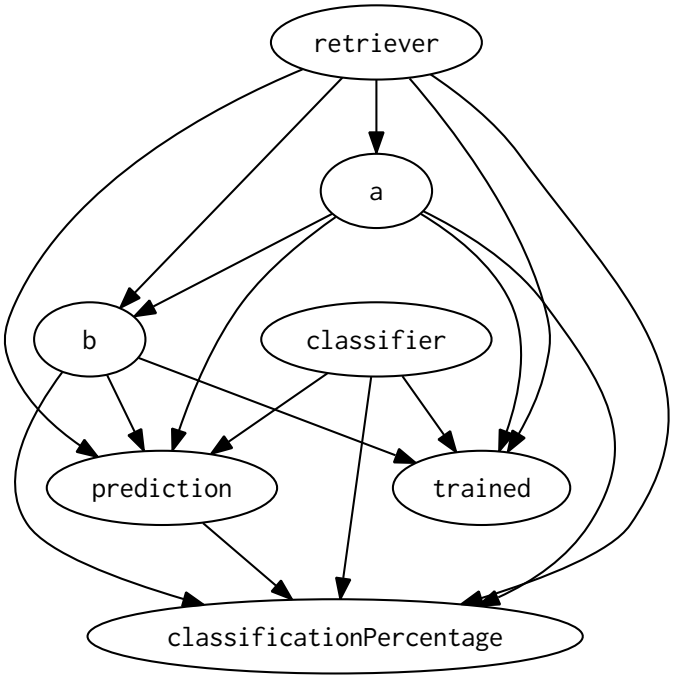


Figure 5.34: Dependencies between variables in the script 1.1.

Second intermediate There are some limitation which are implied from a method chosen to locating grid objects and operations. As it is based on assignments, in case of absence of this construct, information extracted from Ruby source can be incomplete. Following list describes which parts of the script was correctly analyzed and which caused problems. The second intermediate graph is shown in figure 5.35.

- line 6: grid object `retriever` was correctly recognized
- line 14: asynchronous operation was correctly recognized

- line 18: in the same line asynchronous operation is succeeded by result request, moreover there is also result request on previous asynchronous operation handler. This kind of construct might cause problems since from this particular line three nodes should be produced. Solid thin line means that node `b` indicates result request on asynchronous operation `a`, in fact there is a result request for operation `a` but node `b` should stand for request result of operation `async_splitData`.
- line 25: asynchronous operation and dependencies were correctly recognized
- line 27: there is no assignment and as a consequence result request node was not produced
- line 29: asynchronous operation and transitive dependencies on node `a` and `b` were correctly recognized
- line 31: asynchronous operation and direct dependency between nodes `classificationPercentage` and `prediction` were correctly recognized; node for result request on `prediction` operation handler was not produced
- line 33: as a result of assignment lack, result request node was not produced

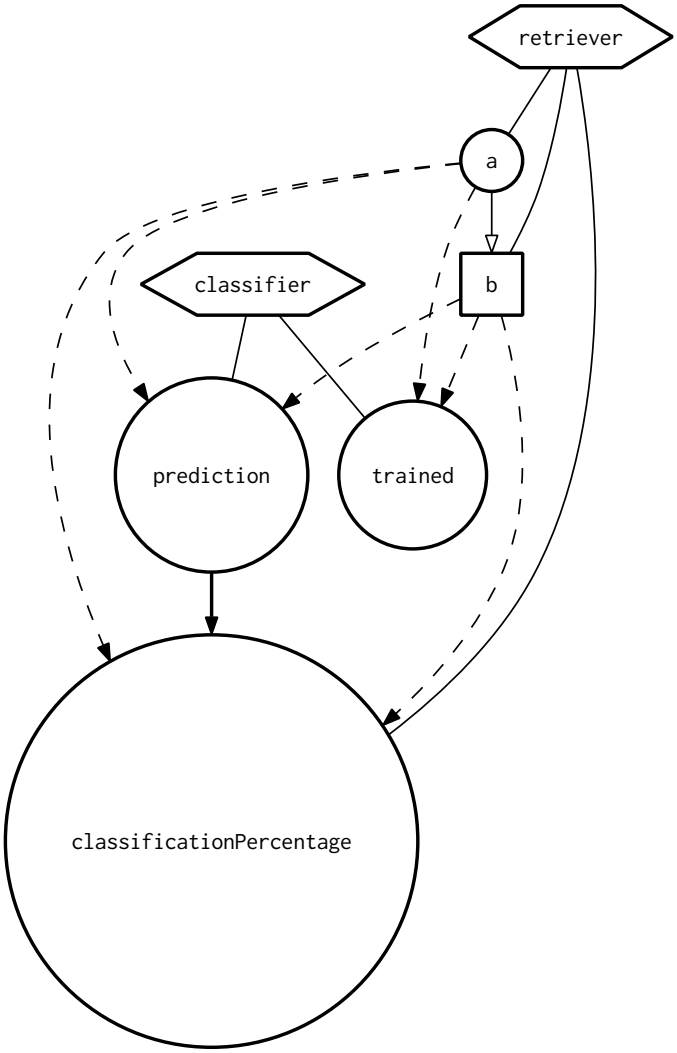


Figure 5.35: Dependencies between operations.

Workflow In all cases, results were not saved into variables, thus there are no dependencies on edges. Except the lack of node for operation `retriever.async_splitData`, graph 5.36 is complete.

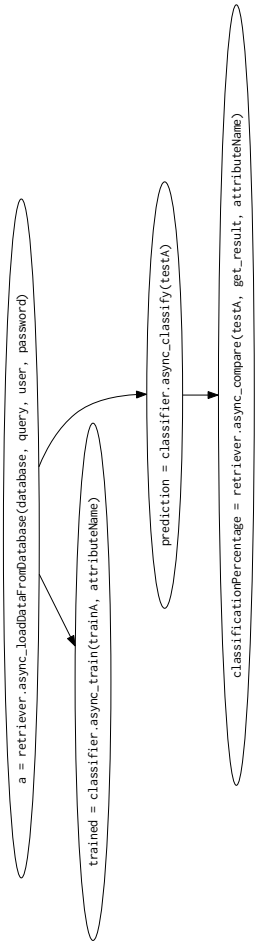


Figure 5.36: Workflow created for Virolab application included in figure 1.1.

5.5.1 Script fixing

Only a few modifications of original script are needed to make operations and workflow graphs completed. Fixed script is presented in figure 5.37. Changes are as follows:

- all operations on grid objects or operations handlers were changes into assignments - line 18 on script 1.1 is changes into lines 18-21 in figure 5.37
- variables from lines 19 and 20 of original script has removed and their usage is replaced by direct references to `b_res` variable

```
1   require 'cyfronet/gridspace/goi/core/g_obj'
2
3   puts 'Start␣of␣weka␣experiment␣!!␣(Asynchronous␣version)!!'
4
5   # Create Web Service Grid Object Instance
6   retriever =  GObj.create('cyfronet.gridspace.gem.weka.WekaGem')
7
8   # Build the query
9   query = 'select␣outlook,␣temperature,␣humidity,␣windy,␣play␣from␣
        weather␣limit␣100;'
10  database = "jdbc:mysql://127.0.0.1/test"
11  user = 'testuser'
12  password = ''
13
14  a = retriever.async_loadDataFromDatabase(database, query, user,
        password)
15
16  classifier =
        GObj.create('cyfronet.gridspace.gem.weka.OneRuleClassifier')
17
18  a_res = a.get_result
19
20  b = retriever.async_splitData(a_res, 20)
21  b_res = b.get_result
22
23  # Set the name of attribute that will be predicted
24  attributeName = 'play'
25
26  trained = classifier.async_train(b_res.trainingData, attributeName)
27  # wait until training is done
28  trained.get_result()
29
30  prediction = classifier.async_classify(b_res.testrainingData)
31  prediction_res = prediction.get_result
32
33  classificationPercentage =
        retriever.async_compare(b_res.testrainingData, prediction_res,
        attributeName)
34  classificationPercentage_res = classificationPercentage.get_result
35  # show results
36  puts 'Prediction␣quality:' + classificationPercentage_res.to_s
37  puts 'End␣of␣weka␣experiment␣!!'
```

Figure 5.37: Real Virolab experiment - weka. Script has been modified to improve workflow generation.

Operations graph for fixed script is included in figure 5.38 and workflow in on 5.39.

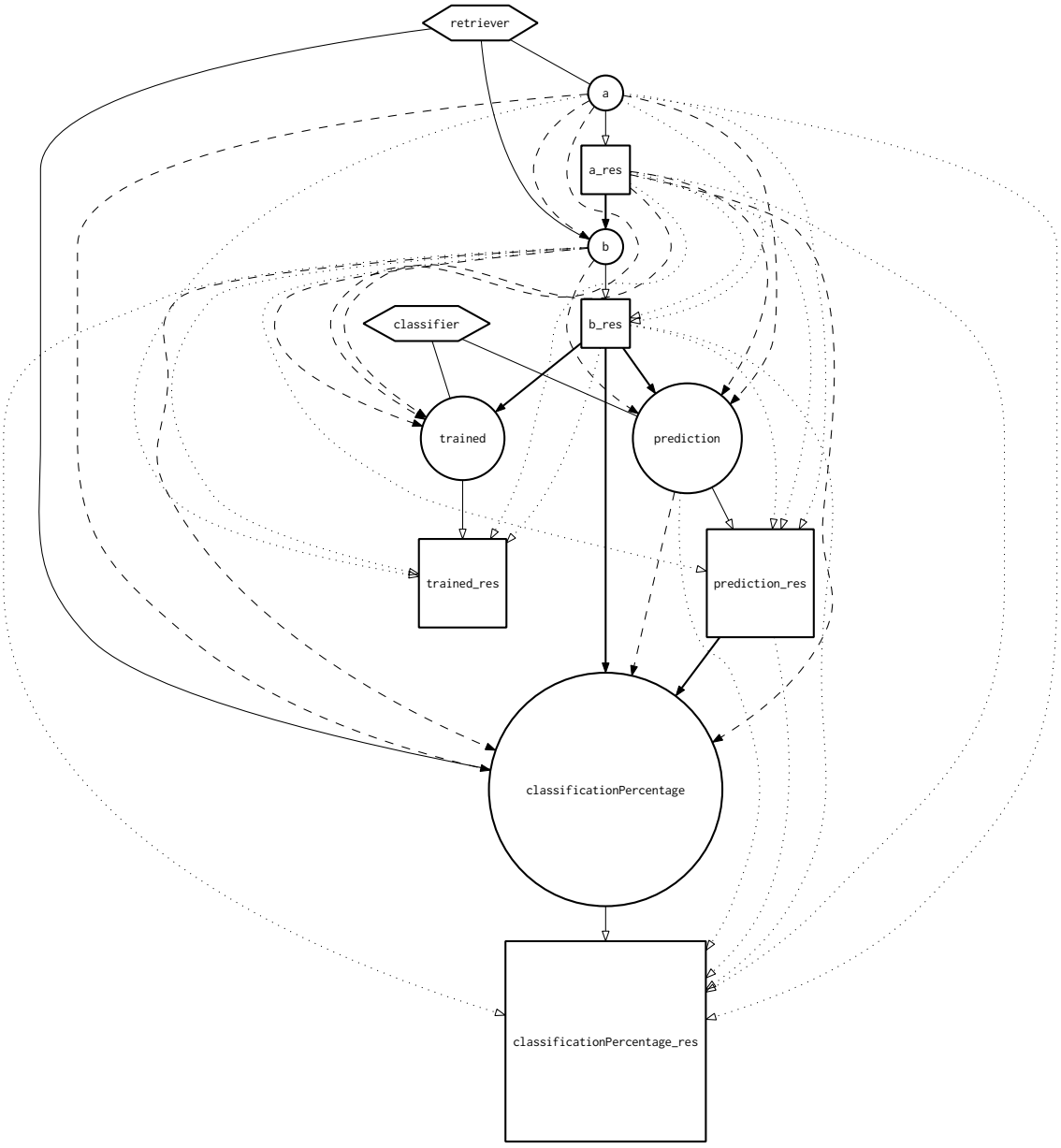


Figure 5.38: Operations graph for fixed script 5.37.

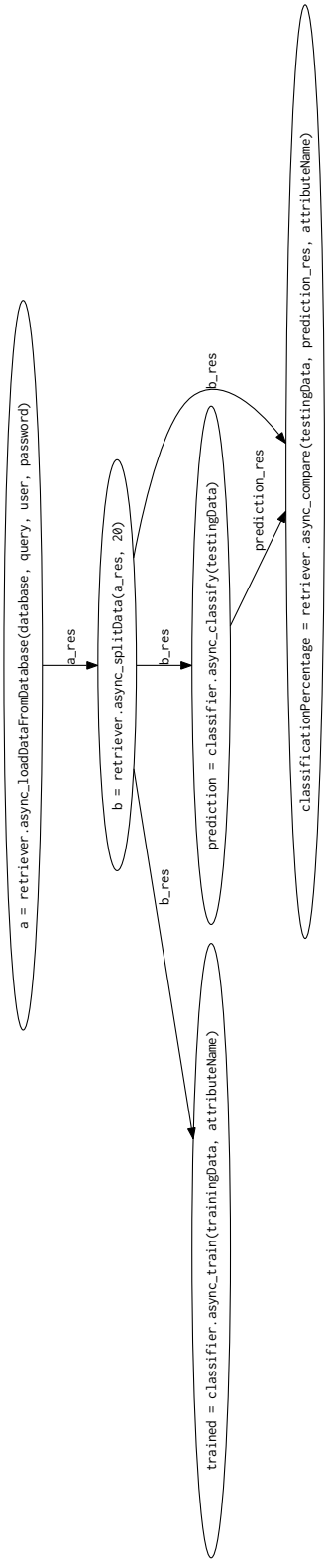


Figure 5.39: Workflow for fixed script 5.37.

5.6 Summary

Virolab programming model provides an ability to implement applications which correspond to a basic workflow patterns (sequence, parallel split, synchronization and exclusive choice). Various Ruby control statements like `loop`, `iteration` or `if` can be used in Virolab applications without loosing ability of analyzing structure and converting them to workflows. Virolab can be extended by introducing parallel statement to enable explicit specifying of Ruby code blocks which should be executed in parallel.

There were also shown that Virolab is able to implement non-trivial real world applications like Montage, CyberShake or Epigenomics. While Ruby scripts coordinate various grid objects can be coordinated, it is still able to transform them to workflows. At least, real world Virolab application was analyzed and transformed into workflow.

CHAPTER 6

Scheduling concept of transformed script

While the workflow building process was already described and followed by various cases of application to workflow conversions, the next goal of this thesis and the purpose of this chapter is an examination of realizing scheduling process on created workflow model.

As it was mentioned before (sections 2.6.1 and 2.1.2), scheduling process can examine structure dependent criterion and structure independent. The purpose of this chapter is a realization of dependent task scheduling process.

6.1 Dependent task scheduling

According to workflow structures taxonomy presented in 2.1.1, current workflow structure can be assigned to *extended digraph*.

In Askalon environment and its AGWL language, there are also used extended digraph representation[11]. For scheduling purposes, this model is converted to DAG representation which is most popular and can be considered as a common model for many workflow management systems[12].

6.1.1 Workflow conversions

Workflow models can be reduced to DAG during execution. Workflow conversions in Askalon Grid Environment[30] splits in two phases:

- initial conversion - basic transformations, workflow structure preserved

- final conversion - loop unrolling, condition evaluation

Original construct	Converted construct
while	sequence
for	
forEach	
parallelFor	parallel
parallelForEach	
if-then-else	sequence
switch	
DAG	DAG

Table 6.1: Askalon constructs conversions

6.1.2 HEFT example

As it was described in section 2.6.1, HEFT heuristic is a static algorithm for DAGs based on the prediction of costs of task execution and communication.

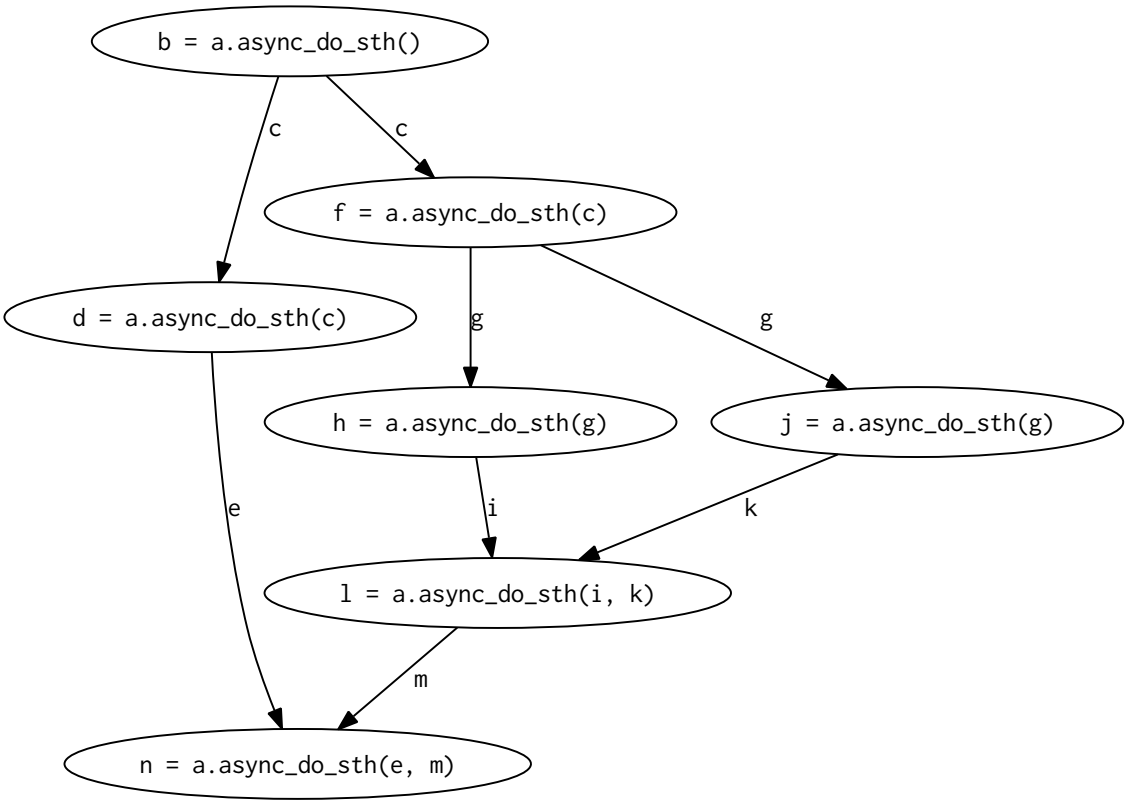


Figure 6.1: The example of workflow prepared to show HEFT procedure (6.1.2).

All tasks from the workflow are prioritized as it was described previously in section 2.6.1, then, starting with a task with highest priority, selection phase is performed. earliest time when particular task can start is designated from the latest time from all predecessor finish times. Then, if in resource has free slot which is later than aeriest start time and has length longer than predicted task length, task is assigned to free slot.

HEFT procedure

The HEFT algorithm steps for the given graph (6.1) are as follows:

1. Select tasks which can be executed - they are these nodes which have all dependencies completed. The task of the first selection is `b = a.async_do_sth()`.
2. Execution time is estimated for selected tasks.
3. Each task has assigned proper resource - assigned resource must fulfill requirements of the task and has the earliest free slot. Free slot is a period of the time in which particular resource has no tasks assigned.

4. End of iteration if there are no remaining tasks to assign, otherwise, go to first point.

6.1.3 Clustering heuristic example

As it was mentioned in paragraph 2.6.1, the goal of clustering scheduling is to minimize communication cost by assigning mutually dependent tasks to a single resource.

Previously described example from this scheduling algorithms category is DSC heuristic. For a given workflow - 6.1, enriched by tasks and edges costs (numbers in brackets) in figure 6.2 - critical path of the clustered graph (the longest path in graph including non-zero communication edge cost and task weights in the path) contains nodes `b = a.async_do_sth()`, `d = a.async_do_sth(c)` and `n = a.async_do_sth(e,m)`, its length is 9.

The critical path of a scheduled graph (DS) contains nodes `f = asunc_do_sth(c)`, `h = asunc_do_sth(g)`, `j = asunc_do_sth(g)` and `l = asunc_do_sth(i,k)`.

To introduce DSC algorithm steps terms *top level* and *bottom level* has to be introduced. Both are sums of tasks and edges costs, *top level* is computed for DS from start node to current (it is calculated during execution of this algorithm) and *bottom level* corresponds to cost of DS from current node to exit node (it is calculated at the begging and it is constant during the execution).

Algorithms takes starting node (since is always has longest path to exit) and then:

1. Current node is merged with a cluster of one of its predecessors if it does not increase *top level* cost, if so, new cluster is assigned for a current node.

2. If there are remaining nodes, select one with the highest *bottom level* and go to first point.

The result of the algorithm is shown in figure 6.2.

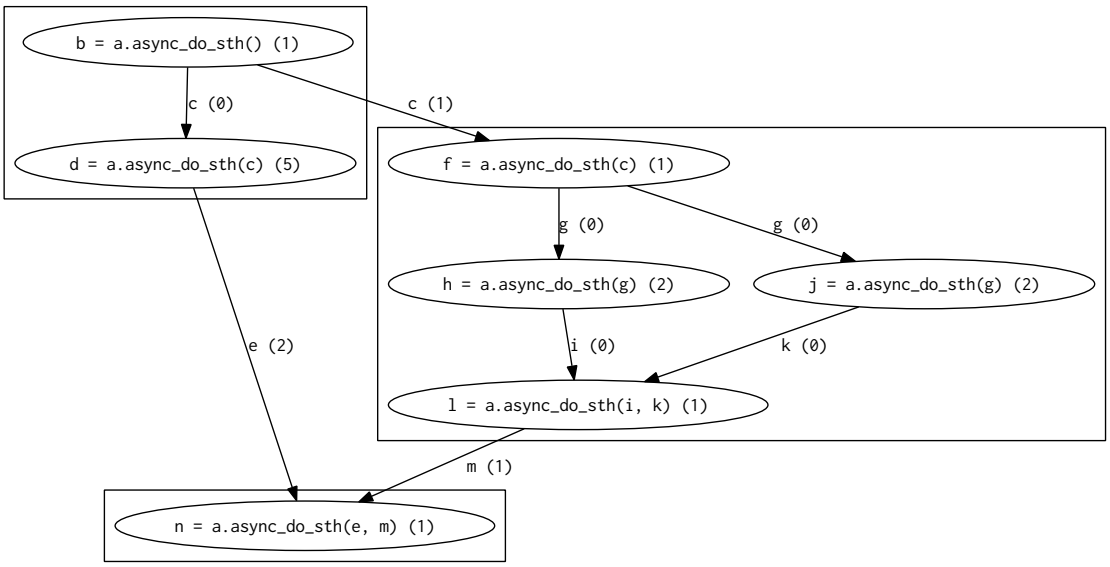


Figure 6.2: Clustering heuristic example - tasks are grouped to decrease communication cost.

6.2 Summary

Presented scheduling algorithms are able to process workflows created for Virolab applications. Workflow conversions approach used in Askalon[30] can adopt existing workflow model for wider class of algorithms.

CHAPTER 7

Summary and future work

This chapter summarizes thesis goals and presents conclusions which appeared during their realization. It includes possible directions of research and further evolution of created tool.

7.1 Conclusions

The main goal of master thesis was to provide ability for far-sighted optimization. Requirements placed by this optimization mode are fulfilled by workflow scheduling approach which considers application structure and dependencies between grid operations.

To convert Virolab application to a workflow model which can be used as a model for scheduling algorithms various analysis were performed.

Processing Ruby source code (chapter 3) gave a knowledge about which Grid Object Classes are used in application and what grid operations are performed. Information about relations between these operations was obtained by resolving variable dependencies. Control statements were parsed and their impact on grid objects was determined. The main conclusion after this process is that Ruby programming language has very complex syntax and dynamism of the language which from one side, gives the opportunity to develop complex applications, causes many problems during its analysis. After all, most of commonly used programming languages can keep and manipulate complex data structures which exclude complete source code analysis.

Gathered information were used to build graphs of workflow representation (chapter 5). Basic four workflow patterns were implemented as Virolab applications and the whole process of building workflow graph was presented. It was shown how complex Ruby

scripts with `loop` and `if` statements affect workflow structure. Well-known workflow applications (Montage, CyberShake, Epigenomics) were reimplemented as hypothetical Virolab applications and then converted to workflows - as well as real Virolab application - 5.37.

Scheduling process were explained on the example - 6 which proves that Virolab workflows give complete information for various scheduling algorithms.

7.2 Future work

Based on the experience gained from the research process on providing far-sighted optimization for ViroLab runtime environment, the wide spectrum of actions can by designated to work on in future. Evolution of this work can be developed in many directions. They can be categorized as follows:

- Improving ViroLab application to workflow conversion:
  - improve by extending routines of analyzing process to enable resolving more complex Ruby scripts,
  - improve by changing application model,
  - provide script validations based on data gathered during analyzing process.
- Make this work compatible and integrate with GrAppO and GSEngine.
- Develop complex scheduling process.

7.2.1 Improving application source to workflow conversions

As the main goal of this master thesis is to provide far sighted optimization for Virolab environment, we observed that the most challenging problem to reach this goal is a process of converting Virolab applications sources into workflows. This problem brought most of research issues and although many of them were solved, there is still much room for improvement in the future.

Add new routines to handle more Ruby statements

As it was mentioned in section 3.2.1, There are 105 types of Ruby statements which can be distinguished by used Ruby parser - ParseTree[27]. Current implementation can analyze only 38 types which makes a room for improvements. Implementation of each type complements internal representation and improves additional process.

Changing application model

As it was mentioned many times in this master thesis, Ruby programming language provides much flexibility in its syntax. Two fragments of Ruby code which look completely different and use various Ruby statements may produce exact results. Flexibility in programming language usually goes with complexity in its syntax which implies problems in parsing and analyzing process.

Concrete Ruby language features which bring issues include:

- There are no syntax distinction between variable and function usage, it means that in analyzing process, it is not known if particular label refers to function or variable, e.g. in Ruby statement `foo = bar`, variable `foo` is assigned to value which is returned from function `bar` or to a value of variable `bar`.
- Each function, beside variables, can take block as an argument. The main issue with that feature is once the block is passed as argument, it is not known when and how this block will be executed. If in the function there is `yield` statement, block will be executed immediately (one or more times), it can be kept and used later as a callback or in different conditions.
- Ruby code can be freely redefined. Thank to Ruby syntax constructions as well as `module_exec`, `class_exec` functions, classes and objects can be easily changed, method can be aliased (`alias_method` function), added or removed.
- The logical structure of Ruby program has no implications in structure of files. `require` and `load` methods which are usually used to load Ruby modules simply execute Ruby code included in given file. It implies that each Ruby application has to be considered as a whole thing since definition of a particular class or method in one file can be changed in another which makes the problem of analyzing process bigger and more complex.
- Ordinary collections as `Array`, `Hash` or `Set` bring problems for resolving dependencies procedure. When the reference of the tracking variable is passed to a collection it can be used and modified in that places of application structure where reference to the collection are available. It implies in fact that assigning variable to an element of a collection causes it cannot be discovered where and how variable is processed thus as a result the track of a variable is lost.

Each of the features which bring issues has to be limited to improve application source to workflow conversions.

Starting from the Ruby language flexibility, which can be rightly described by the Perl language programming slogan: “There’s More Than One Way To Do It”[31] (it is known that Ruby is influenced by Perl design), programming language chosen for the

purpose of source analyzing has to fulfill motto of other modern programming language - the Python: "There should be one - and preferably only one - obvious way to do it"[32]. It would expand capabilities of building workflows if each statement of programming language (like method invocations, loops and block/function definitions) had only one configuration of nodes in its parsed form (internal representation).

Immutable variables (value assigned to a particular label cannot be changed) solves issue of tracking variables. Since it is known that value of variable will not be changed in whole application whenever all references to this variable are discovered or not, it reduces necessity of considering reassigments and resolving global variable dependencies.

Ruby script validations

The understanding of Virolab application structure provides data for script examining. In a current application model, the user has no restrictions in designing application control flow until the source code is correct Ruby language syntax. It gives a flexibility for developers but it also lets them to make mistakes and to create inefficient constructs.

If the `get_result` operation is requested not as late as it possible in application flow or the asynchronous operation is not performed as soon as possible, some optimization work should have to be done on application source.

The optimization which can be performed include:

- Changing sequential iterations into parallel for loops if there are no looped dependencies and the order of execution does not affect variables from outside of the loop (e.g., results are stored in collections which does not have order like set or Ruby 1.8.7 `hash`).
- Basing on the variable dependencies graph, asynchronous grid operations and its result request can be moved within their blocks to ensure that control flow would not be suspended before all possible asynchronous operations are performed.
- Having a knowledge about operations and variables dependencies gives a possibility of making validations which usually cannot be made in pre-execution time like finding operations on variables which were not initialized. If there is a `get_result` request on operation handler which does not have corresponding asynchronous grid operation, many costly calculation may turn out pointless since this error can interrupt application execution.

7.2.2 GSengine and GrAppO integration

Virolab optimizer - GrAppO is written in Java programming language and it works on Java Virtual Machine[8, 9]. As it was mentioned earlier in section 3.2.1, used tools can be

used with any Ruby language implementation, particularly with JRuby which is already a part of Virolab Laboratory runtime (section 1.1.3). While GrAppO was designed in mind of far-sighted optimization, extending existing tools by workflow scheduling approach requires implementation of common interfaces as well as redefinition of optimization strategies.

7.2.3 Implement complex scheduling routines

In chapter 6 were presented scheduling algorithms and opportunities of their utilization with workflow models. Future work in this segment is related with work described in previous section - the integration with GSEngine since scheduling algorithms require data from monitoring and tracking services.

7.2.4 Implicit parallelism - transparent `get_result` operation

The explicit parallelism model of asynchronous grid operations and their handlers which keep operation statuses can be consider as explicit parallelism - the operations are executed in parallel if they are explicitly specified by the application developer. The other approach is a implicit parallelism where operations are invoked in parallel and operation result is requested when needed.

Motivation

Transparent `get_result` operation simplifies scripts, user is no longer obligate to analyze where to put `get_result` request to achieve best optimization. It also minimize probability of error when user omits operation result request which was previously mentioned in section about possible script validations - 7.2.1.

Technical aspects

In Ruby language it is impossible to assign the callback to the variable which would be executed when value of the variable is requested (and thus invoke transparent `get_result` operations).

The goal is to provide mechanism to enable construction presented in figure 7.1a instead of current explicit approach in figure 7.1a.

```
a = GObj.create
state = a.async_do_sth
# or
state = a.do_sth
puts state
```

(a) Implicit parallelism.

```
a = GObj.create
state = a.async_do_sth
puts state.get_result
```

(b) Explicit parallelism.

Figure 7.1: Comparison of transparent and explicit `get_result` operation.

If it can be assumed that asynchronous operation results are used only in functions (as on script in figure 7.1a where the result is printed to standard output), the transparent `get_result` operation can be realized by special handlers added to all methods of all objects in user space. If the handler receives asynchronous operation handler, it invokes `get_result` method. It goes with hard limitation - operation handlers cannot be used in other context than function arguments like `foo = operation_handler`.

The definition of the operation handler created to prove the concept of operation handlers can look as following `ActiveObject` class with `get_value` method:

```
class ActiveObject
  def get_value
    100_009
  end
end
```

Ruby language provides many routines and methods that enable dynamic code generation.

All living objects in current virtual machine instance can be accessed through special object called `ObjectSpace` which provides routines that interact garbage collector facility. Thus, to redefine all methods in all loaded classes following operations can be performed:

```
ObjectSpace.each_object(Module) do |c|
  next if [Method, UnboundMethod, Array].include? c

  next unless c.instance_of?(Class)

  result = false
  a = c
  while a = a.superclass
    result = result || a.superclass == Exception
  end
```

```
  next if result

  c.class_eval do
    (instance_methods + private_instance_methods).each do |method_name|
      next if instance_method(method_name).owner != c
      next if method_name == :initialize
      old_method = instance_method method_name

      define_method method_name do |*args, &block|
        args.map! { |a| a.instance_of?(ActiveObject) ? a.get_value : a }
        old_method.bind(self).call(*args, &block)
      end
    end
  end
end
```

And as a result, execution of fallowing code:

```
class Test
  def test1 a
    a
  end
end

def test2 a
  a
end

puts test2(ActiveObject.new)
puts Test.new.test1(ActiveObject.new)
puts 2 + ActiveObject.new
puts ActiveObject.new
```

it will produce:

```
100009
100009
100011
100009
```

since in all occurrences, `ActiveObject` instance is used as function arguments. In Ruby 1.8.7, object which owns method `puts` seems to be not accessible from `ObjectSpace`, thus method `puts` is not modified and output listing might be different than presented above.

Foregoing solution handles whole loaded code but what if user want to create its own classes and methods? To change methods which are going to be loaded or defined but do not exist before execution of the script, following `Module` class method called `method_added` can be used which is a simple callback that is invoked each time when the new method is defined.

```
Module.class_eval do
  define_method :method_added do |method_name|
    # p method_name

    return if $adding
    return if method_name == :method_added
    $adding = true
    old_method = instance_method method_name

    define_method method_name do |*args, &block|
      args.map! { |a| a.instance_of?(ActiveObject) ? a.value : a }
      old_method.bind(self).call(*args, &block)
    end

    $adding = false
  end
end
```

7.2.5 Summary

The future work covers technical and research aspects of computer science depending if the current work will be improved to provide far-sighted optimization for Virolab laboratory or will be a base for new approach of developing grid applications in imperative programming language and translating them into workflows.

Bibliography

[1] Monage. http://montage.ipac.caltech.edu/. 24

[2] ViroLab. Virolab virtual labolatory. http://virolab.cyfronet.pl. 10, 87

[3] Tomasz Gubała, Bartosz Baliś, Maciej Malawski, Marek Kasztelnik, Piotr Nowakowski, Matthias Assel, Daniel Harężlak, Tomasz Bartyński, Joanna Kocot, Eryk Ciepieła, Dariusz Król, Jakub Wach, Michał Pelczar, Wlodzimierz Funika, and Marian Bubak. Virolab virtual laboratory. In *Cracow Grid Workshop 2007 Workshop Proceedings*, pages 35–40. ACC CYFRONET AGH, 2008. 10

[4] Eryk Ciepiela, Joanna Kocot, Tomasz Gubała, Maciej Malawski, Marek Kasztelnik, and Marian Bubak. Gridspace engine of the virolab virtual laboratory. In *Cracow Grid Workshop 2007 Workshop Proceedings*, pages 53–58. ACC CYFRONET AGH, 2008. 10, 12

[5] ViroLab. Virolab homepage. http://www.virolab.org. 10

[6] Tomasz Bartyński, Maciej Malawski, and Marian Bubak. Invocation of grid operations in the virolab virtual laboratory. In *Cracow Grid Workshop 2007 Workshop Proceedings*, pages 59–64. ACC CYFRONET AGH, 2008. 10

[7] Yukihiro Matsumoto. Ruby programming language home page. http://www.ruby-lang.org. 11

[8] Joanna Kocot and Iwona Ryszka. Optimization of grid application execution. Master's thesis, AGH University of Science and Technology, June 2007. 13, 104

[9] Maciej Malawski, Joanna Kocot, Eryk Ciepiela, Iwona Ryszka, and Marian Bubak. Optimization of Application Execution on the ViroLab Virtual Laboratory. In Marian Bubak, Michał Turała, and Kazimierz Wiatr, editors, *Proceedings of Cracow Grid Workshop - CGW'07, October 2007*, Krakow, Poland, 2007. ACC-Cyfronet AGH. 13, 104

[10] Domenico Talia, Ramin Yahyapour, Wolfgang Ziegler, Marek Wieczorek, Andreas Hoheisel, and Radu Prodan. Taxonomies of the multi-criteria grid workflow scheduling problem. In *Grid Middleware and Services*, pages 237–264. Springer US, 2008. 13, 17, 20, 21

[11] Marek Wieczorek, Andreas Hoheisel, and Radu Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Gener. Comput. Syst.*, 25(3):237–256, 2009. 17, 18, 19, 20, 21, 22, 95

[12] Fangpeng Dong and Selim G. Akl. Technical report no. 2006-504 scheduling algorithms for grid computing: State of the art and open problems, 2006. 20, 33, 36, 95

[13] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *The 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS08)*, pages 1 – 10, November 2008. 23, 24, 26, 27

[14] Daniel S. Katz, Joseph C. Jacob, G. Bruce Berriman, John Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, and Gurmeet Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society. 23

[15] Philip Maechling, Ewa Deelman, Li Zhao, Robert Graves, Gaurang Mehta, Nitin Gupta, John Mehringer, Carl Kesselman, Scott Callaghan, David Okaya, Hunter Francoeur, Vipin Gupta, Yifeng Cui, Karan Vahi, Thomas Jordan, and Edward Field. Scec cybershake workflows—automating probabilistic seismic hazard analysis calculations. In Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 143–163. Springer London, 2007. 25, 26

[16] Scott Callaghan, Philip Maechling, Ewa Deelman, Karan Vahi, Gaurang Mehta, Gideon Juve, Kevin Milner, Robert Graves, Edward Field, David Okaya, Dan Gunter, Keith Beattie, and Thomas Jordan. Reducing time-to-solution using distributed high-throughput mega-workflows - experiences from scec cybershake. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on*

eScience, pages 151–158, Washington, DC, USA, 2008. IEEE Computer Society. 25, 26

[17] Usc epigenome center. `http://epigenome.usc.edu/`. 27

[18] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962. 28

[19] Kurt Jensen. A brief introduction to coloured petri nets. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 203–208. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0035389. 28

[20] Nick Russell, Arthur T. Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPMcenter.org, 2006. 28, 29, 30, 31, 32

[21] Workflow patterns. `http://www.workflowpatterns.com/`. 28

[22] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003. 29, 30

[23] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with agwl: an abstract grid workflow language. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 676–685, Washington, DC, USA, 2005. IEEE Computer Society. 31

[24] A. H. M. Ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30:245–275, 2005. 32

[25] H. Topcuoglu, S. Hariri, and M.Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13:260–274, 2002. 35

[26] Tao Yang and Apostolos Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:951–967, 1993. 35

[27] Ryan Davis. Parse tree. `http://parsetree.rubyforge.org/`. 39, 40, 54, 55, 101

[28] AT&T. Graphviz. `http://www.graphviz.org/`. 54

[29] Yaml: Yaml ain't markup language. `http://www.yaml.org/`. 56

[30] Michael Mair, Jun Qin, Marek Wieczorek, and Thomas Fahringer. Workflow conversion and processing in the askalon grid environment. In *Austrian Grid Symposium*, Innsbruck, Austria, September 2006. OCG Verlag. 95, 99

[31] Larry Wall. Perl, the first postmodern computer language. `http://www.perl.com/pub/1999/03/pm.html`. 103

[32] Tim Peters. The zen of python. `http://www.python.org/dev/peps/pep-0020/`. 103