Scientific Programming 20 (2012/2013) 359–377
DOI 10.3233/SPR-120358
IOS Press

Constructing workflows from script applications

Mikołaj Baranowski [a,*], Adam Belloum [a], Marian Bubak [a,b] and Maciej Malawski [b,c]

[a] *Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands*
E-mails: {m.p.baranowski, A.S.Z.Belloum}@uva.nl

[b] *Department of Computer Science, AGH University of Science and Technology, Kraków, Poland*
E-mails: {bubak, malawski}@agh.edu.pl

[c] *Center for Research Computing, University of Notre Dame, IN, USA*

Abstract. For programming and executing complex applications on grid infrastructures, scientific workflows have been proposed as convenient high-level alternative to solutions based on general-purpose programming languages, APIs and scripts. GridSpace is a collaborative programming and execution environment, which is based on a scripting approach and it extends Ruby language with a high-level API for invoking operations on remote resources. In this paper we describe a tool which enables to convert the GridSpace application source code into a workflow representation which, in turn, may be used for scheduling, provenance, or visualization. We describe how we addressed the issues of analyzing Ruby source code, resolving variable and method dependencies, as well as building workflow representation. The solutions to these problems have been developed and they were evaluated by testing them on complex grid application workflows such as CyberShake, Epigenomics and Montage. Evaluation is enriched by representing typical workflow control flow patterns.

Keywords: Ruby, grid computing, GridSpace, workflow construction, analysis of script applications

1. Introduction

Programming and running complex scientific applications on Grid infrastructures requires development of high level programming abstractions and supporting environments. Among them, we can distinguish two important classes which are based on different concepts. In the first model, applications are defined as complete workflows using specific workflow languages like Abstract Grid Workflow Language (AGWL) or Yet Another Workflow Language (YAWL). These solutions are implemented in scientific workflow systems such as Taverna [9], Kepler [11], WS-VLAM [3] or Pegasus [6]. The aim of the second approach is to enable access to grid infrastructures from general-purpose programming languages using libraries and APIs. It gives grid users an opportunity to use a grid infrastructure without having to learn grid specific technologies, it is easier to modify existing applications and test them. In this paper, we describe an attempt to bridge the gap between these two approaches by proposing a solution for converting script-based applications of GridSpace [14] platform into scientific workflows.

GridSpace environment provides capabilities offered by a grid infrastructure over APIs and libraries which are accessible from Ruby [15] scripts – also called experiments. Experiment developer is allowed to instantiate grid objects and to perform operations on them.

Grid object operations can be invoked both synchronously and asynchronously (Listing 1). An asynchronous operation returns an operation handler which represents state of a remote operation. Then, invoker keeps an executing process during which further operations can be called (including other asynchronous operations) till the result of an asynchronous operation is not required. The result of an asynchronous operation can be obtained by invoking `get_result()` method on an operation handler as it is shown in Listing 1.

In GridSpace, there is a three level grid object abstraction [13]. The top level includes grid object classes, these are abstract entities which define operations. One grid object class may have many implemen-

[*] Corresponding author: Mikołaj Baranowski, Informatics Institute, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands. E-mail: m.p.baranowski@uva.nl.

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

Listing 1. Simple example of the GridSpace script with asynchronous Grid Operation.

tations which are build on various technologies and run on different environments, their operations are consistent. The third level of abstraction is a grid object instance which is in the same relation with a grid object implementation as a grid object implementation is with a grid object class – one grid object implementation can have many grid object instances, running on different resources.

One of the advantages of a scientific workflow model is that it explicitly specifies tasks and dependencies between them, which in turn allows an optimization of a workflow execution in a process of task scheduling [22]. Unfortunately, programming grid applications in a general-purpose language does not give a possibility to use workflow optimization techniques due to lack of information about dependencies between application tasks. Moreover, it is hard to predict which grid resource will be used by a particular application and how results produced by these resources will be consumed.

Resolving the issue of translating grid application sources into workflow models may open an ability of applying tools and theoretical statements designed for a workflow scheduling. It also provides data that can be used to validate applications by changing their control structures to eliminate inefficient constructs.

The main objective of our work is to elaborate an approach of transforming scripts into workflow models. As an example of script based applications, GridSpace environment is taken.

To achieve a goal of transforming Ruby scripts into workflows, the following goals have to be realized:

- find dependencies between grid object operations invoked from Ruby scripts, analyze Ruby source code, locate grid object classes, their instances and operations, check operation arguments and then, using these data, find dependencies between grid operations,
- build a workflow basing on an application source code, gather information collected during a realization of a previous point and locate control flow structures in Ruby source code,
- validate approach by building workflows for control-flow patterns and well known workflow applications (Montage, CyberShake, Epigenomics) – the purpose is to prepare hypothetical GridSpace implementations of these well known applications and transform them into workflows,
- provide data needed to enable optimization which is based on a Ruby source code structure – a workflow representation.

This paper is organized as follows: in Section 2 we describe the related work, including workflow description languages, scripting approaches and workflow patterns. Subsequently, in Section 3 we give a theoretical details of our solution followed by a description of implementation related issues – Section 4. In Section 5 we report on the results we achieved regarding workflow patterns supported as well as typical benchmark workflows, such as Montage or Cybershake applications. We conclude our paper with enumerating the list of open issues and prospects for future work.

2. State of the art

2.1. *Workflow description languages*

Usually, workflow models are described using specific languages. Directed Acyclic Graph (DAG) is a format used to represent workflows by Condor and DAGMan [19]: it describes jobs with their dependencies (control flow). XML-based DAX is a version of DAG used by Pegasus [6] to represent abstract workflow which are mapped by Pegasus planner to executable workflows in DAG format.

AGWL [7] is 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* can be a computation (let associate it with a grid object), 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.

The origin of YAWL [8] was preceded by gathering a wide collection of workflow patterns [16] described in Section 2.2. 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. 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).

2.2. *Workflow patterns*

The motivation for creating workflow patterns by *Workflow Patterns Initiative* was to delineate fundamental requirements for workflow modeling [16,24]. 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.

Sequence pattern is a fundamental building block for workflow processes [16]. 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 [21].

The other patterns [21] are: *parallel split* which is a point in workflow process where the particular branch of a control flow splits into multiple branches which can be executed concurrently, *synchronization* is a point in the workflow process where many threads of control are joined into one, *exclusive choice* is a point in the workflow process where, basing on the decision, one from several outgoing branches is chosen, *simple merge* is a point in the workflow process where two or more branches come together without synchronization.

2.3. *Scripting languages for distributed processing*

As an alternative to graph-based workflow notations, scripting languages can be also effectively used to program and execute complex scientific applications with dependencies between tasks.

SWIFT [23] is a scripting language with a syntax specifically designed to handle large-scale data processing on distributed and parallel environments. It supports such constructs as loops, conditions and a convenient mappings from file to variable names. SWIFT script is compiled into workflow and executed by Karajan execution engine, which can dispatch tasks on parallel machines, clusters or grids.

Makeflow [18] is a workflow system based on a scripting language similar to Makefile syntax. It allows specifying dependencies between tasks and files that need to be transferred. Makeflow compiles the input script and its execution engine dispatches the workflow tasks on Condor on SGE clusters, providing fault tolerance and data staging.

Another approach implemented in Grid Superscalar [17] is to use imperative language to describe application workflow. It can use constructs such as loops and conditions and the compiler uses techniques known from parallelizing compilers to enable out-of-order execution of tasks. Grid Supersclar can submit jobs to grid sites using Globus.

All the above-mentioned solutions require defining a new specific programming language and a new syntax to describe dependencies or control structures. GridSpace, in contrast, relies on unmodified Ruby syntax, which makes it easier to develop scripts and also use the rich Ruby standard library in workflow scripts. Another common feature of listed solutions is that they compile the whole script before execution, while GridSpace relies on standard Ruby interpreter. The advantages of compilation are that it is easier to detect all the dependencies and plan (schedule) the execution in advance. On the other hand, using standard interpreter offers the possibility of interactive scripting and late-binding which is useful for dynamic environments like grids. The work described in this paper helps bridge the gap between GridSpace and other compiler-based scripting system by static analysis of script code to detect dependencies and enable to take advantage of scheduling optimization known from workflow systems.

3. Translation of scripts into workflows

In this section we described the process of extracting data from the application sources written in a scripting programming language in the purpose of transforming them into a workflow of grid operations. To show dependencies between steps more explicitly, a graph in-

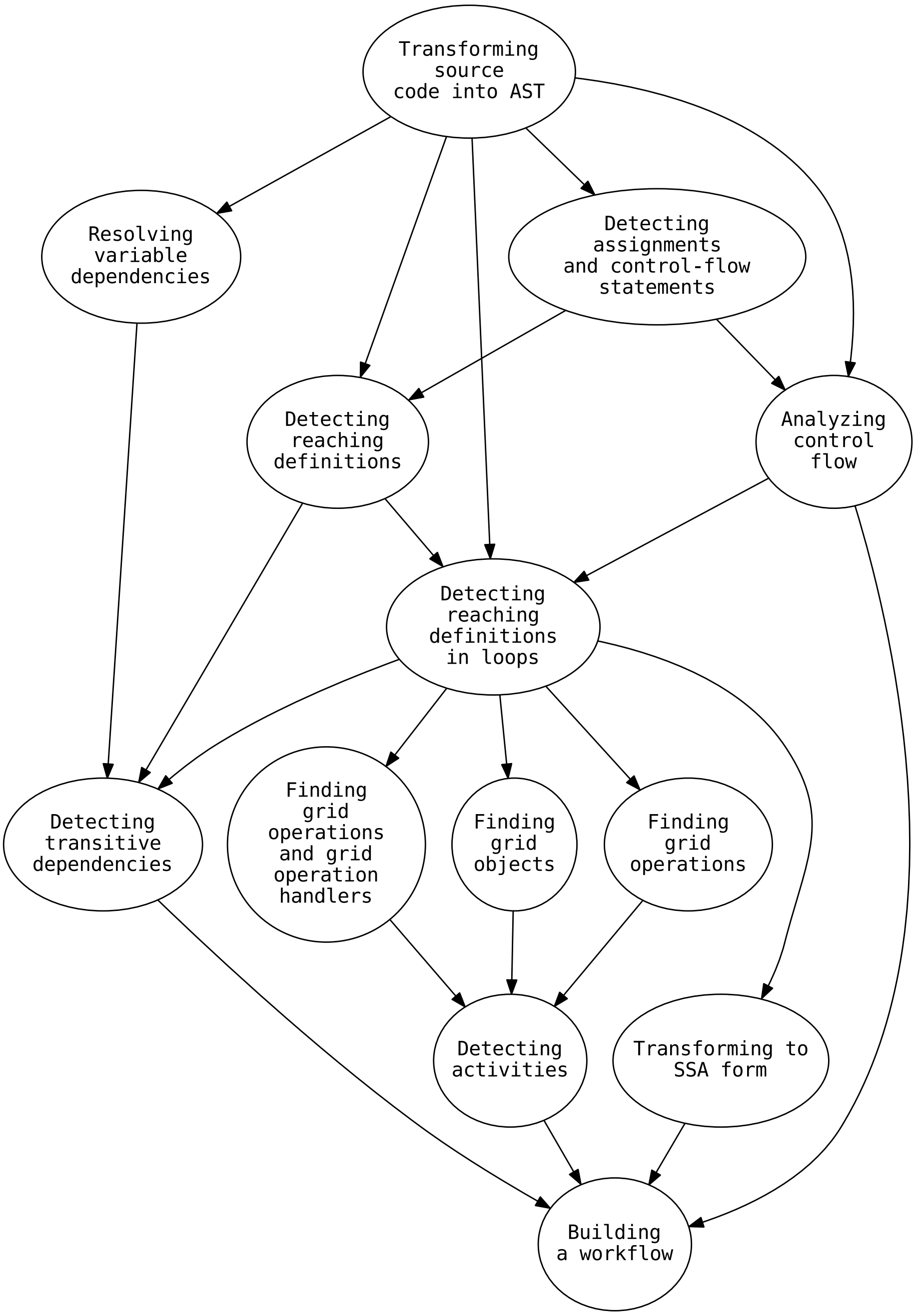
Transforming source code into AST
Resolving variable dependencies
Detecting assignments and control-flow statements
Detecting reaching definitions
Analyzing control flow
Detecting reaching definitions in loops
Detecting transitive dependencies
Finding grid operations and grid operation handlers
Finding grid objects
Finding grid operations
Detecting activities
Transforming to SSA form
Building a workflow

Fig. 1. A graph contains steps of analysis described in Section 3, arrows stand for data flow between operations.

cluded in Fig. 1 presents data flow among operations of analysis.

3.1. Assumptions

Proposed approach is based on analysis performed on Abstract Syntax Trees (ASTs). To implement the described solutions a tool which can generate AST from a source code has to be available for particular programming language. To simplify process of analysis, all statements, with the exception of control-flow statements (loops and conditional statements), should appear in following forms:

- $x := y \ op \ z$ where op is a binary arithmetic or

logical operation,
- $x := op\ y$ where op is unrary operation,
- copy statement: $x := y$,
- procedure call: $x := p(y_1, y_2, \ldots, y_n)$,
- indexed assignments: $x := y[i]$ and $x[i] := y$.

The considered forms are almost identical with three-address statements [1]. They can be translated from constructs that do not fulfill above requirements. Foregoing description, where each statement has clear indication which variables are used and which variable is produced, allows to focus only on analysis of dependencies and control flow rather than code transformations. To achieve the goal of creating workflows, required information from source code have to be extracted. The obtained data is used to identify all workflow *activities* and detect how *data-flow* and *control-flow* are realized:

- detecting activities – activities are identified as grid object operations,
- data and control flow – the interaction between grid operations occurs when the result of the first one affects any of the arguments of the second one (a data flow dependency), or when the second grid operation is in a control structure (such as `loop` or `if` statement), which condition depends on the result of the first grid operation (a control flow dependency).

To detect the activities and data and control flow, several analysis steps have to be performed, which are described in detail in the following sections.

3.2. Data and control flow dependency analysis

Resolving variable dependencies can be performed on plain AST form. For that purpose and for purpose of further analysis described in following sections, AST has to be reflected in the internal data structure which should enable easy traversing and handling additional information.

Resolve variables dependencies – Dependency graph. Using the information contained in AST, each node is examined to check if there are any variable occurrences in its subtree. If so, all found variable occurrences are assigned as its dependencies. Basing on this principle, AST is traversed and all direct dependencies are noted in the AST-based internal representation.

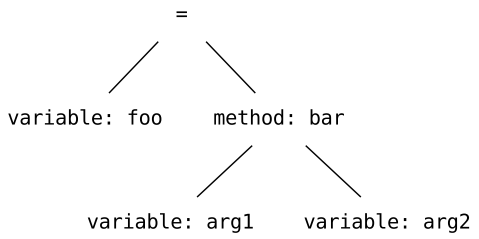

Fig. 2. An example of an assignment in AST form.

3.2.1. Detecting assignments

According to assumptions, the basic entity that should be taken into account is an assignment. In Fig. 2 there is an example of an assignment as AST. Within analysis, AST is traversed with the purpose of finding all assignments and control-flow statements. As a result of this operation AST nodes corresponding to assignments and control-flow constructs are marked.

Detecting reaching definitions. An equation that specifies how information is accessible in various points of a program can be written as: $out[S] = gen[S] \cup (in[S] - kill[S])$ [1], where S represents a statement, $out[S]$ represents information accessible after a statement, $in[S]$ and $kill[S]$ are respectively sets of information that is accessible before a statement and that is killed in a statement. To adopt the above method to the issue of detecting reaching definitions, the following remarks have to be made:

- in AST, information is propagated in the order determined by depth-first traversal algorithm,
- according to the assumptions, every assignment generates (defines) a new variable. If in the particular programming language lexical scoping interaction occurs between local variables and block local variables, an assignment can also kill the information making it inaccessible.

To apply the above method, AST has to be traversed with depth-first traversal algorithm, and for every assignment found the above equation has to be computed to determine which variables can be reached in every assignment. After this operation, each assignment – represented by AST node – has associated information on which variables are reachable in this point in a form of a list of AST nodes – representing other assignments in which these variables are created.

Detecting transitive dependencies. Once dependency graph is established and analysis of reaching definitions is complete, transitive dependencies can be resolved. First, each node which stands for a variable is linked with the assignment in which considered variable is created. This step requires the information of

Algorithm 1. Detecting transitive dependencies

```
$changed \leftarrow true$
**while** $changed$ **do**
    $changed \leftarrow false$
    **for all** $s_1 \in$ statements **do**
        **for all** $s_2 \in$ statements **do**
            **if** $s_2$ and $s_1$ are mutually independent **then**
                **for all** variable $d$ which is a dependency of variable $s_2$ **do**
                    **if** $d$ depends on $s_1$ and $s_2$ depends on $d$ and $s_1$ is reachable in $s_2$ **then**
                        $s_2$ depends on $s_1$
                        $changed \leftarrow true$
                    **end if**
                **end for**
            **end if**
        **end for**
    **end for**
**end while**
```

Algorithm 2. Linking assignments with control-flow statements

```
$s \leftarrow AST.root$
**for all** $s \in$ statements **do**
    **if** $s$ is assignment **then**
        $i \leftarrow s$
        **while** $i$ has $parent$ **do**
            $i \leftarrow i.parent$
            **if** $i$ is a control statement **then** LINKSTATEMENTS($s$, $i$)
            **end if**
        **end while**
    **end if**
**end for**
```

variables dependencies and data from the analysis of reaching definitions. Then, information about dependency is spread among statements as it is shown in an Algorithm 1. All assignments and control-flow constructs are considered statements. After a transitive detection is performed, all assignments in AST contain information about mutual dependencies.

3.2.2. Analyzing control flow structures

To allow the final workflow representation to include control-flow statements from the script applications, each assignment should be linked with a control-flow construct in which it is located. To achieve that, all assignments – represented by AST nodes – are individually linked with all loops and conditional statements that appear in the path from the root node of AST to given node, as it is shown in Algorithm 2. Function *LinkStatements* links assignment (first argument) with control statement (second argument). Conditions of control-flow statements are included into AST which makes it possible to reconstruct them as a programming language expression.

Detecting reaching definitions in loops. An important aspect of reaching definitions occurs in loops. It has to be noted that definitions performed in one iteration are accessible in subsequent executions of loop body. It implies that algorithm presented in Section 3.2.1 which detects transitive dependencies, should be extended to mark all nodes of AST located in the same loop statement, apart from limiting itself to nodes located at the right site of it. Information about dependencies previously created in Section 3.2.1 should be enriched by data collected in this step.

3.2.3. Reassignment issue

The limitations of application model proposed in Section 3.1 enable to reuse assigned values. Since the

Algorithm 3. Changing variable names to solve reassignment issue

```
$s \leftarrow AST.root$
while $s$ do
    if $s$ has the form $x := y$ and $x$ is reachable at $s$ then CHANGEVARIABLENAME($s$, $AST$)
    end if
    $s \leftarrow$ NEXT($s$, $AST$)
end while
```

operation of resolving dependencies is based on variable names (Section 3.2), it is important to recognize these names properly and to be certain that a variable name represents the value that is expected. The problem occurs when a new variable value is assigned to the variable label that was already used and still appears in the scope. It might make impossible to resolve dependencies correctly even in simple scripts and as a result, impossible to create workflows.

Solution – Single static assignment. The solution to this problem is based on changing the variable names. The required information was already gathered during analysis of reaching definitions (Section 3.2.1). In the Algorithm 3, function *Next* implements depth-first search.

A *storing* variable x is a left operand of assignment operation and it denotes location where the value is stored. The function *ChangeVariableName* changes label of storing variable of the assignment which is identified by the first argument and all its occurrences basing on the data gathered in reaching definitions step. As a result, alternative names of storing variables of assignments that kill information are stored in AST.

3.2.4. Control flow and SSA

Using a Single Static Assignment (SSA) form may influence the ability of tracking the control flow. However, in our case, SSA is performed after variable dependencies are released and after variable scopes are detected. It implies that in the final form the internal representation keeps all dependencies and original names. Depending on how the conditions and loops are evaluated during execution time, corresponding alternative variable names should be used.

3.2.5. Detecting activities

Activities in GridSpace application model are identified as grid object operations [13]. In the case of the synchronous operations the activity is in a one to one relationship with a grid object operation. Asynchronous operations are split into two statements – request for operation handler performed on the grid object and the result request invoked on the operation handler, as it is shown in Listing 1. A synchronous grid object operation is a special case of an asynchronous operation where an operation handler is requested for a result immediately after it was acquired. To detect activities, the algorithm has to find grid objects and grid operations.

Finding grid objects. Grid objects are returned as a result of a functions which are provided by the API. Depending of a implementation, this function can be provided as a standalone function or method of the object or class.

AST should be traversed to find assignments where the storing value is computed in expression that can be identified as a grid object creation. It is a matter of convention to design the API in such a way that procedures that instantiate grid objects can be distinguishable. For example, in GridSpace grid objects must be created by calling `GObj.crate()` factory method. This process should produce the AST with all the grid object assignments marked.

Finding grid operations and grid operation handlers. Once grid objects are identified, it is possible to find grid operations. Assignments in which storing value is computed by invoking a method on grid object are selected. Variables that are assigned in this kind of statements are marked as grid operation handlers.

3.3. Building a workflow

While building a workflow, all the data gathered in the analysis are combined. From the whole AST, only assignments of grid objects are filtered. Starting with these of them that do not have any dependencies on other grid objects, the final workflow representation is built. Dependencies are designated by relations established in reaching definitions and variable dependencies phases – two assignments are dependent if their storing variables are dependent and if their storing variable definitions are reachable. To determine if control-flow node should be inserted between considered pair of operations, a set s is individually defined for each grid object assignment. Set s specifies in which control-flow constructs the particular operation is included. If set s_1 stands for control-flow operations linked with operation o_1 and s_2 stands for control-flow operations linked with operation o_2, $s_1 - s_2$ determines which control-flow statement occur between operations o_1. o_2 and $s_2 - s_1$ determines on which condition operation o_2 is dependent on operation o_1.

4. Implementation

4.1. Source code analysis

According to the assumptions made in Section 3.1, Ruby source code has to be translated to AST or to equivalent form. We use Ruby parser [5]. It converts Ruby source code to symbolic expressions (also called S-expression or sexp) using Ruby arrays and base types. Ruby parser transforms this source code from Listing 1 into s-expressions as in Listing 2.

S-expression produced by this tool are equivalent to AST regarding to the definition of AST from [1]: "A useful starting point for thinking about the translation of an input string is an AST in which each node represents an operator and the children of the node represent the operands. (...) superficial distinctions of form, unimportant for translation, do no appear in AST".

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

Listing 2. Listing presents S-expressions produced from script 1. 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 list of Ruby operations holds 105 elements. 38 most important operations, such as `:lasgn`, `:call` or `:iter` were selected, and for each of them a routine which can analyze its S-expression was implemented. To allow further analysis, S-expressions are converted into internal representation. Its data structure is prepared to keep additional data for each operation and optimized for easy and efficient traversing (Fig. 3).

4.2. Find grid objects and operations

Listing 1 presents a script with a grid operation `as_op()` performed on grid object `b`. The analyzer identifies which variables are grid objects and then, it is able to find grid operations as function calls on grid objects and grid operation handlers as their return values. In GridSpace, by the convention, grid objects are created by a method `create` of a class `GObj`. Method accepts an argument which can demand particular type of grid object, as follows: `g_obj = GObj.create("some_string")`. Based on a description from Section 3.2.5, it is assumed that every assignment which has identical structure to a tree graph in Fig. 4 is a grid object creation. Following a definition of grid operation from Section 3.2.5, all assignments that fulfill requirements described in this section should be marked as grid operations. In GridSpace model however, we are considering only asynchronous operations that by the conventions methods name should start with `as_`. Operation handler is a returning value of a grid operation. In GridSpace, by convention, the only method which can be executed on the operation handler object is `get_result`. Grid operations are dependent if assignments in which they appear are dependent.

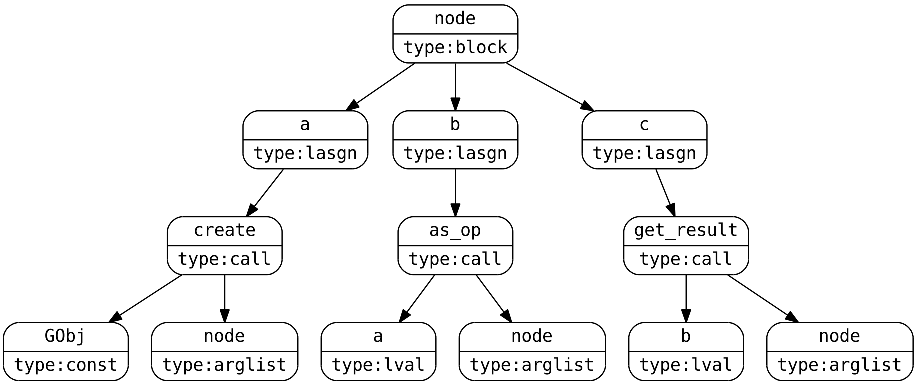

Fig. 3. Internal representation of a script from Listing 1. It shows a transformed S-expression from Listing 2. At this point of analyzing process, each tree node contains `type` and `name`.

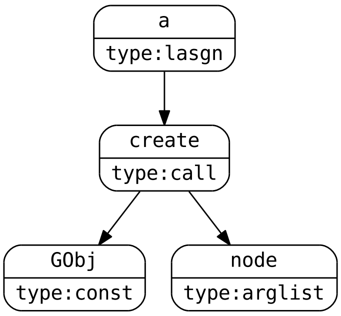

Fig. 4. The grid object creation pattern – a fragment of internal representation which stands for operation: `a = GObj.create()`. Grid objects can be located by searching thought the internal representation for that kind of constructs.

4.3. *Find dependencies from block statements – Analyzing control flow*

Regarding the Ruby programming language feature which was already mentioned in Section 3.2.1, dependency with source in a control-flow statement can occur if the particular variable was already assigned before the statement.

Eliminate reassignments. In order to eliminate reassignments, the internal representation is traversed, as it was described in Section 3.2.3. The name changes are implemented by adding a suffix to variable names. The following source code with reassignments:

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

is translated to:

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

Find dependencies in if, while, iter and loop statements. It has to be noted that in Ruby programming language occurs a lexical scoping interaction between local variables and block local variables. It means that statements inside the block can modify local variables. Moreover, block local variables are not accessible outside the block (in opposition to Python programming language mechanisms). Taking into account above facts, following rules are applied:

- dependencies with assignments in control-flow constructs are treated as with local dependencies with understanding that these operations are placed in control-flow statement,

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

Listing 3. Example of dependencies from `if` statement block. When the condition is fulfilled, variable `c` is dependent on `a` and `b`, otherwise, `c` is dependent only from `b`.

- assignments from control-flow statements can not be lined as a dependency for assignments from outside these constructs unless storing variables were already initialized.

4.4. *Final form of internal representation*

The final form of internal representation includes all the gathered data from the analysis process and all the modifications. Example of internal representation of script from Listing 3 is presented in Fig. 5.

4.5. *Workflow builder tool*

The implementation of the analysis process resulted in creating a tool which allows to process Ruby scripts and produce all discussed data and graphs. The two external tools used to implement the workflow builder for GridSpace scripts are parse tree [5] for source code analysis and GraphViz [2] for drawing graphs.

The workflow builder tool which was developed during research on the problem is composed from following modules, presented in Fig. 6. Class `DagEdge` which keeps information about edges and `DagNode` about nodes of a workflow, both of them are used in a `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 [5] 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.

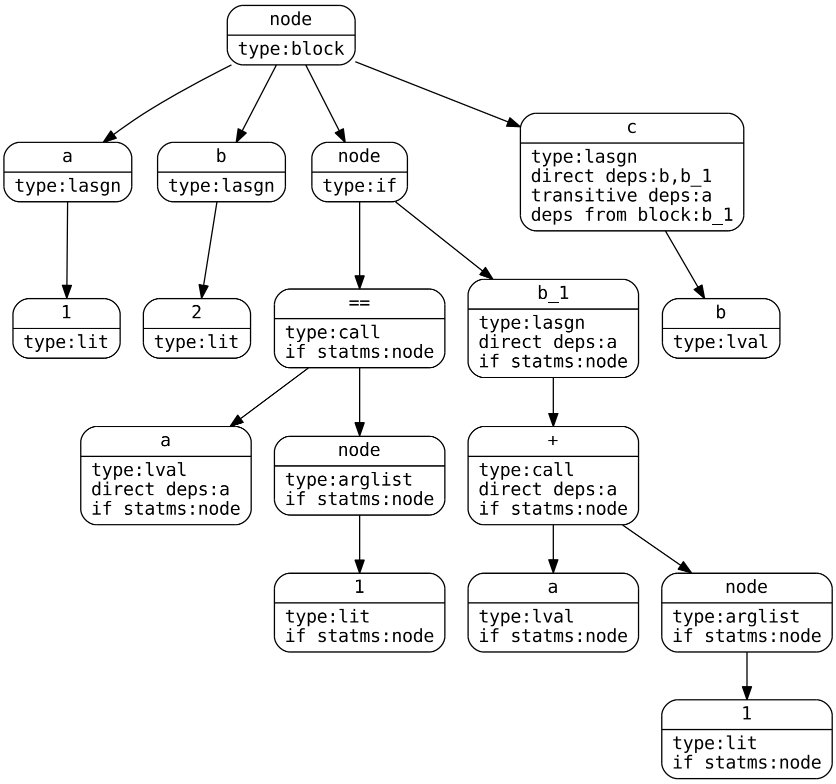

Fig. 5. Processed internal representation of a script from Listing 3. Each node contains identified type, direct and transitive dependencies, dependencies from the blocks and separately dependencies from `if` statements which occurs two times in a current example.

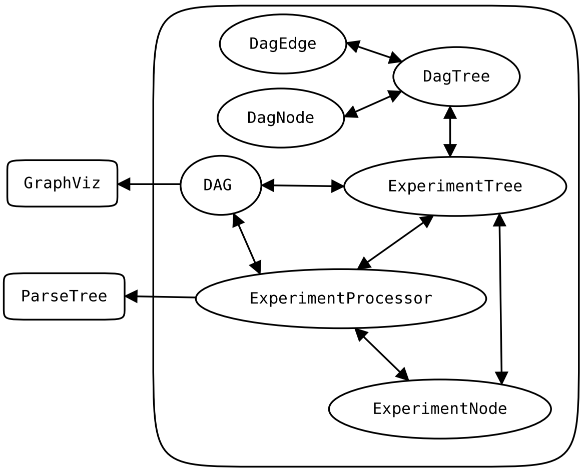

Fig. 6. The architecture diagram presents relations between modules – `DAG` which generates graphs from internal representation using `GraphViz` and its dot representation, `ExperimentProcessor` which implements Ruby parser routines using external `ParseTree` tool and `ExperimentTree` which handles internal script representation using `ExperimentNode` module. Modules `DagTree`, `DagEdge` and `DagNode` are used in `ExperimentTree` module to handle and operate on internal workflow representation.

Workflow description language based on YAML. There is a need to keep workflows in permanent storage to enable easy access of complete workflows from the application. In addition to the existing possibility of exporting workflows to dot files, the support for serialization in YAML language [25] was added. It is easy to read by a human and it has import/export tool in majority of programming languages. The exported workflow in YAML representation contains all information which are gathered in graphical workflow representation.

4.6. Constructing a workflow

In order to build workflows, the internal representation is traversed and all the nodes which are not asynchronous operations on grid objects are filtered out. The remaining nodes are grouped into:

(1) pairs of explicit dependencies,
(2) pairs of transitive dependencies,
(3) pairs of dependencies from `if` or `loop` statements.

To make the workflow building process more clear to understand, two intermediate forms between the internal representation and workflow representation are created. The first intermediate graph represents dependencies between assignments. The second intermediate graph distinguishes between different dependency types and assignments of different operations.

The final workflow graph can be then produced based on second intermediate graph with the following transformations:

(1) grid objects are removed,
(2) grid operation handler nodes are replaced by labels on edges which indicate data flow,
(3) transitive dependencies are removed except for the cases where there is no direct dependency (there is no label on the edge).

The examples of the intermediate representation and the resulting workflow graph are shown in Section 5.1.1.

5. Results

To validate the analysis process and the developed workflow builder tool, a set of test cases was prepared. The first group of tests included all the workflows patterns as described in Section 2.2. The second group was used to demonstrate how to use the workflow builder tool to build typical benchmark workflows such as Montage [10] from the corresponding GridSpace experiments.

5.1. Supporting workflow patterns

In Section 2.2 control-flow patterns, which are building blocks of larger workflows, were presented. To prove that these aspects of process-control can be implemented in GridSpace Ruby scripts, a workflow creation process was performed for each pattern. The following section presents this process on the example of *exclusive choice* pattern.

5.1.1. Exclusive choice

In GridSpace experiment, as in any Ruby script, operations which are executed under certain condition are commonly located in `if` statement. Exclusive choice workflow pattern implementation should comprise a group of several operations where only one can be executed depending on a condition. In Listing 4, first grid operation `b = a.as_op()` is always executed but the subsequent operations depend on conditions of `if` statements.

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

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

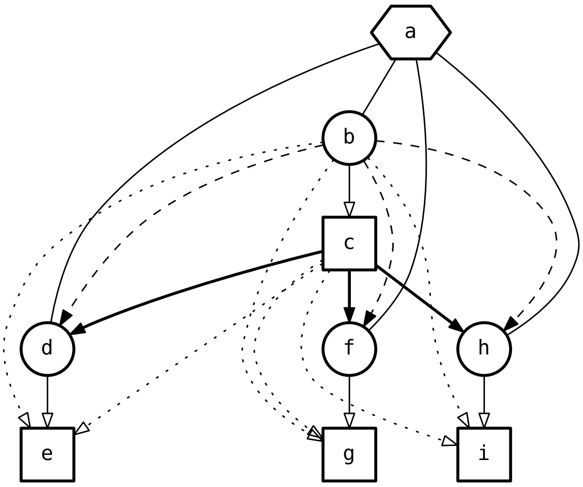

Fig. 7. Operation dependencies of the exclusive choice pattern implementation. Grid operations are represented as circles, result requests are squares and grid objects are hexagons. Direct dependencies are represented by solid lines (dependency edge which points to a grid operation is bold and has a filled arrow), transitive dependencies have dotted or dashed lines (edges pointing to a grid operations are dashed and also have filled arrows). Edges without arrows show which grid objects operation is invoked on. Three operations `d`, `f` and `h` are independent but because of a lack of control flow structures in this graph representation, it can not be determined if these grid operation can be executed in parallel.

Building a final workflow representation. During the translation of application sources to the workflow model and subsequent intermediate models are created as it is described in Section 4.6. Figure 7 presents the second intermediate representation which shows dependencies between grid operations.

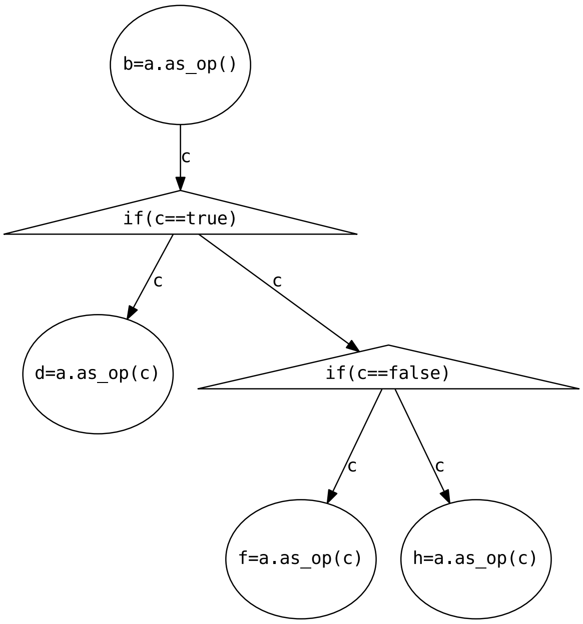

Fig. 8. Exclusive choice workflow pattern built from its GridSpace implementation (Listing 4). Triangle shaped nodes represent conditions and dotted edges point to the places where control flow exits conditional blocks. When the control flow goes through a triangle shaped node, only one outgoing edge is chosen depending on the condition expression and its evaluation.

Figure 8 includes workflow representation of exclusive choice pattern built from its GridSpace implementation. This workflow can be considered as an explicit representation of exclusive choice since there is a special construct which determines which outgoing branch is chosen – a triangle shaped node.

5.1.2. Introducing a parallel for construct

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. The motivation of extending GridSpace constructs by *parallel for* is to provide explicit method for defining which parts of GridSpace application can be executed in parallel. Existing GridSpace libraries does not contain any routines which allow such kind of facilities. A statement presented in Listing 5 was introduced to provide parallel execution for given block.

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

Listing 5. 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]`. Workflow built from the script is presented in Fig. 9.

5.2. Benchmark workflows

Benchmark workflows, inspired by real world applications, are often used for testing and scheduling systems. The *workflow generator* allows creating arbitrarily large workflow models, providing ability of benchmarking and comparing implementations efficiency of workflow systems [4]. To test our workflow builder tool, hypothetical GridSpace applications were created for the following benchmark workflows: Montage [10], CyberShake [12] and Epigenomics [20]. They have the form of GridSpace scripts that reflect both the control and data flow of original workflow applications. CyberShake is an application developed by Southern California Earthquake Center to model earthquake processes in the purpose of seismic hazard analysis. It contains many concurrent control branches. Epigenomics is an application used for research on the epigenetic state on a human genome, it performs independent conversions, mappings and filters on DNA sequences separated into several chunks. This workflow in turn is an example of pipelined application.

Montage (An Astronomical Image Mosaic Engine) [10] combines both features of other two workflows described above, it consists of both parallel and pipelined sections. It 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 adapted to running on grid infrastructure.

There are four main steps in the image assembling process:

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

Listing 6 presents a hypothetical GridSpace experiment which implements the data and control flow of Montage application.

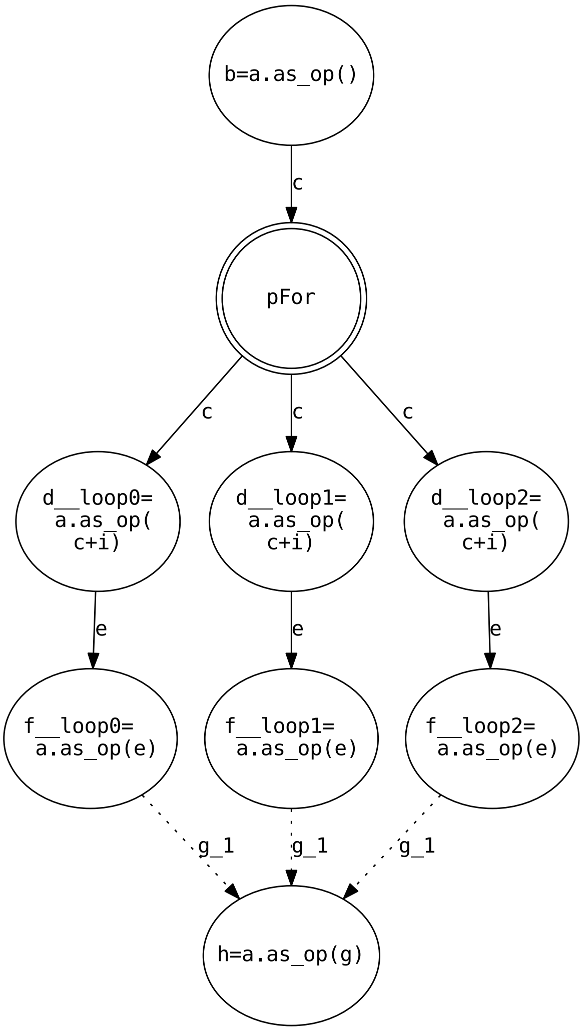

Fig. 9. Workflow of an application with a parallel for statement. An independent branches between `pFor` node and dotted arrows will be executed in parallel.

```
dataRepository = GObj.create("fotos")                    1
mProjectPP = GObj.create("mProjectPP")                   2
mDiffFit = GObj.create("mDiffFit")                       3
mConcatDiff = GObj.create("mConcatDiff")                 4
mBgModel = GObj.create("mBgModel")                       5
mBackground = GObj.create("mBackground")                 6
mImgTbl = GObj.create("mImgTbl")                         7
pipeline = GObj.create("pipeline")                       8
                                                         9
# asynchronous operation returns operation               10
     handler
data_hdlr = dataRepository.as_load("12")                 11
data = data_hdlr.get_result()                            12
                                                         13
fotos_result = nil                                       14
P.pFor(data) do |f|                                      15
  fotos_hdlr = mProjectPP.as_op(data)                    16
  fotos_result = fotos_hdlr.get_result()                 17
end                                                      18
                                                         19
diffFit_result = nil                                     20
P.pFor(fotos_result) do |f|                              21
  diffFit_hdlr = mDiffFit.as_op(fotos_result)            22
  diffFit_result = diffFit_hdlr.get_result()             23
end                                                      24
                                                         25
concatDiff_hdlr =                                        26
     mConcatDiff.as_op(diffFit_result)
concatDiff_result =                                      27
     concatDiff_hdlr.get_result()
                                                         28
bgModel_hdlr =                                           29
     mBgModel.as_op(concatDiff_result)
bgModel_result = bgModel_hdlr.get_result()               30
                                                         31
backgrounds_result = nil                                 32
P.pFor(fotos_result) do |f|                              33
  backgrounds_hdlr =                                     34
       mBackground.as_op(fotos_result,
       bgModel_result)
  backgrounds_result =                                   35
       backgrounds_hdlr.get_result()
end                                                      36
                                                         37
imgTbl_hdlr =                                            38
     mImgTbl.as_op(backgrounds_result)
imgTbl_result = imgTbl_hdlr.get_result()                 39
                                                         40
pipeline_hdlr = pipeline.as_op(imgTbl_result)            41
pipeline_result = pipeline_hdlr.get_result()             42
```

Listing 6. Script shows hypothetical situation – how Montage would look like if it was written in Ruby and with usage of GridSpace Grid interface.

Comparing the GridSpace implementation and the original Montage application following modifications and simplifications has been applied:

(1) since presented workflow tool is limited to information gathered from code parsing, it is impossible to resolve dependencies between objects stored in collections, such as `Array`, `Hash` or `Set`,

(2) parallel loops in lines `15`, `21` and `33` would be replaced by normal loops if operation handlers were stored in a collection such as Hash or List. In such a case the original dependencies between `mProjectPP` and `mDiffFit` could be reconstructed then – the issue is also mentioned in future work Section 6.

Resulting Montage workflow is presented in Fig. 10. Hypothetical GridSpace applications that implement Cybershake and Epigenomics workflows are shown in Listings 7 and 8. Workflow graphs generated from these scripts are included in Figs 11 and 12.

6. Conclusions and future work

The described tool enables to convert the GridSpace application source code into a workflow model, which in turn may be used for scheduling, provenance, or vi-

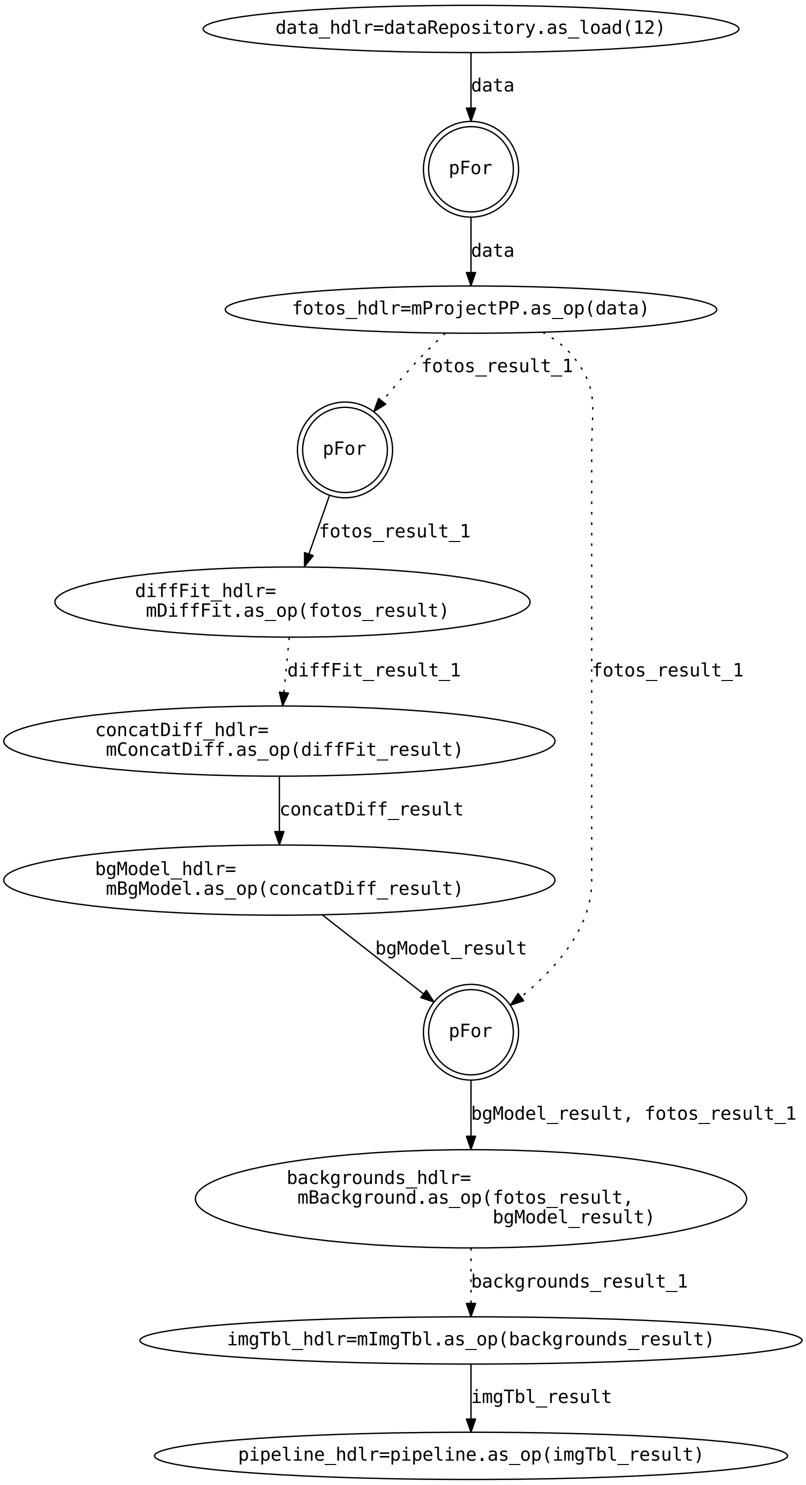
data_hdlr=dataRepository.as_load(12)
data
pFor
data
fotos_hdlr=mProjectPP.as_op(data)
fotos_result_1
pFor
fotos_result_1
diffFit_hdlr=
mDiffFit.as_op(fotos_result)
diffFit_result_1
fotos_result_1
concatDiff_hdlr=
mConcatDiff.as_op(diffFit_result)
concatDiff_result
bgModel_hdlr=
mBgModel.as_op(concatDiff_result)
bgModel_result
pFor
bgModel_result, fotos_result_1
backgrounds_hdlr=
mBackground.as_op(fotos_result,
bgModel_result)
backgrounds_result_1
imgTbl_hdlr=mImgTbl.as_op(backgrounds_result)
imgTbl_result
pipeline_hdlr=pipeline.as_op(imgTbl_result)

Fig. 10. Montage workflow created for a script from Listing 6.

```
sgt = GObj.create("sgt")
synthesis = GObj.create("synthesis")
peak = GObj.create("peak")
zip_seis = GObj.create("zip_seis")
zip_psa = GObj.create("zip_psa")

input_handler = sgt.as_op("data")
input = input_handler.get_result()

s = []
P.parallelFor([1,2,3,4]) do |i|
  s[i] = synthesis.as_op(input[i])
end

p = []
s_result = []
P.pFor([1,2,3,4]) do |i|
  s_result[i] = s[i].get_result()
  p[i] = peak.as_op(s_result[i])
end

s_result = s.get_result()
zs = zip_seis.as_op(s_result)

p_result = p.get_result()
zp = zip_psa.as_op(p_result)

zp_result = zp.get_result()
zs_result = zs.get_result()
```

Listing 7. Script shows hypothetical situation – how Cybershake workflow would look like if it was written in Ruby and with usage of GridSpace Grid interface.

```
epigenomics = GObj.create("epigenomics")
fqs_h = epigenomics.as_fastQSplit

map_h = []
P.pFor([1,2,3,4]) do |i|
  fqs_res = fqs_h.get_result()
  fc_h = epigenomics.as_filterContams(fqs_res)
  fc_res = fc_h.get_result()
  s2s_h = epigenomics.as_sol2sanger(fc_res)
  s2s_res = s2s_h.get_result()
  f2b_h = epigenomics.as_fastq2bfq(s2s_res)
  f2b_res = f2b_h.get_result()
  map_h[i] = epigenomics.as_map(f2b_res)
end

map_res = map_h.get_result()
mm_h = epigenomics.as_mapMerge(map_res)
mm_res = mm_h.get_result()
mi_h = epigenomics.as_maqIndex(mm_res)
mi_res = mi_h.get_result()
pileup_h = epigenomics.as_pileup(mi_res)
pileup_res = pileup_h.get_result()
```

Listing 8. Script shows hypothetical situation – how Epigenomics workflow would look like if it was written in Ruby and with usage of GridSpace Grid interface.

sualization. We addressed the issues of analyzing Ruby source code, resolving variable and method dependencies, as well as building workflow representation. The results of our approach are advanced enough to produce workflows for simple grid applications and to determine which requirements have to be fulfilled to provide translations for complex grid applications with wide usage of collections, such as `Array` and `Hash` and user defined functions and classes.

The presented methodology was validated by performing conversions of workflow patterns implementations and well as of known benchmark workflows such as Montage.

Processing Ruby source code (Section 3) gives knowledge about which Grid Object Classes are used in application and which grid operations are performed. Information about relations between these operations is obtained by resolving variable dependencies. Control statements are parsed and their impact on grid objects is determined. The main conclusion after development of this process is that Ruby programming language has very complex syntax and dynamism of the language. This, on one hand, gives the opportunity and flexibility to develop complex applications, but on the other hand it causes many problems during the script analysis. Nevertheless, most of other commonly used programming languages also can manipulate complex data structures, which exclude the possibility of complete source code analysis.

The information extracted from GridSpace scripts is used to build graphs of workflow representation (Section 5). Basic four workflow patterns were implemented as GridSpace 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 GridSpace applications and then successfully converted to workflows.

It was observed that source to workflow conversion problem brought many of research issues and although many of them were solved, there is still much room for improvement in the future.

As it was mentioned in Section 4.1, there are 105 types of Ruby statements which can be distinguished by used Ruby parser – ParseTree [5] while the current implementation of builder tool can analyze only 38 types. Implementation of each type would complement internal representation and improve the whole process.

Flexibility in programming language usually goes with complexity in its syntax which implies problems in parsing and analyzing process. The specific Ruby language features (which has to be limited to improve application source to workflow conversions) that raise issues include:

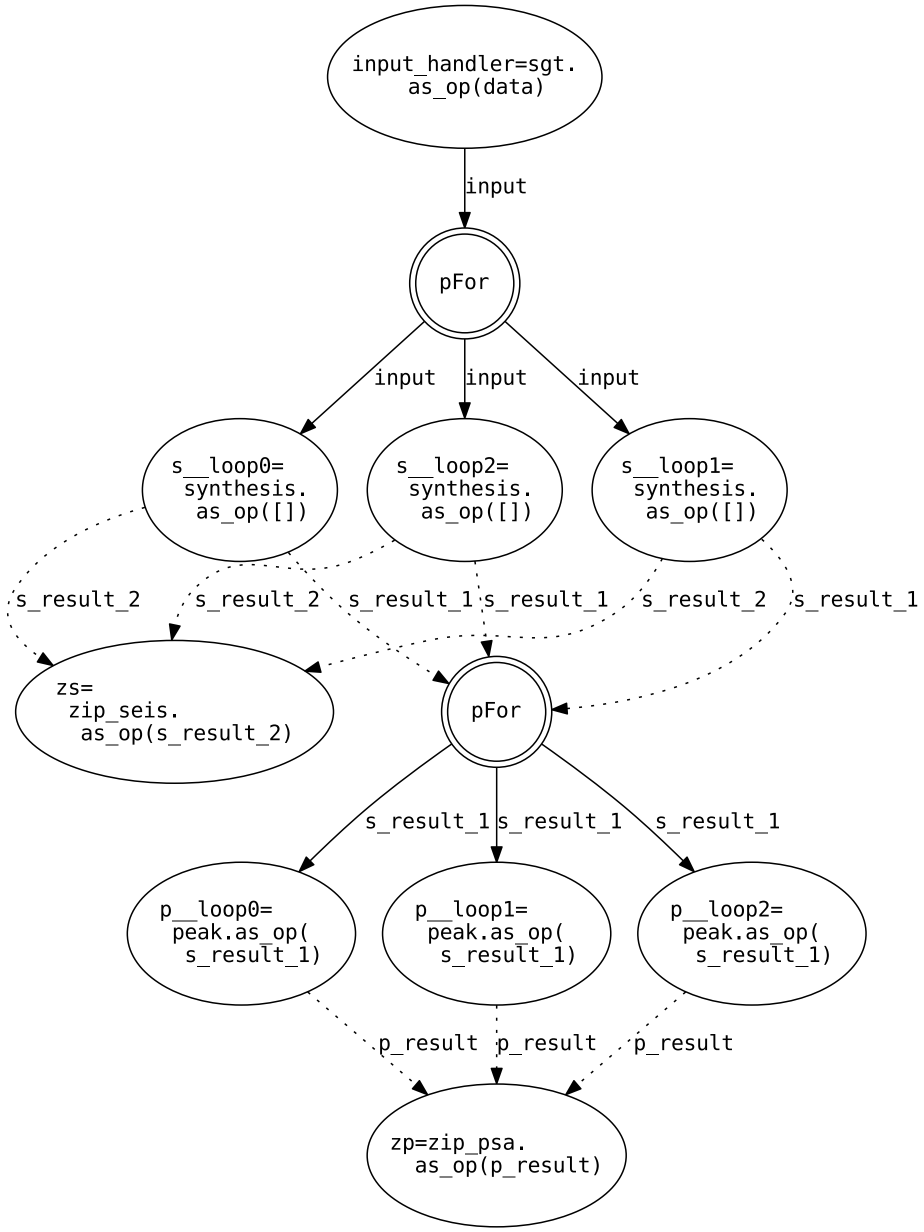

Fig. 11. Cybershake workflow created for a script from Listing 7.

(1) Lack of syntax distinction between variable and function usage, 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`.
(2) Passing block as function arguments. It is not known when and how this block will be executed. If there is `yield` statement in the function, the block will be executed immediately (one or more times), or it can be kept and used later as a callback or in different conditions.
(3) The fact that Ruby classes and objects can be easily changed, methods can be aliased, added or removed in runtime.
(4) Allowing of code modification at runtime (monkey patching) and lack of any relation between Ruby program logic and structure of files.
(5) When tracked values (e.g. grid operation results) are passed to the collections (`Array`, `Hash` or `Set`), it becomes impossible to resolve exact de-

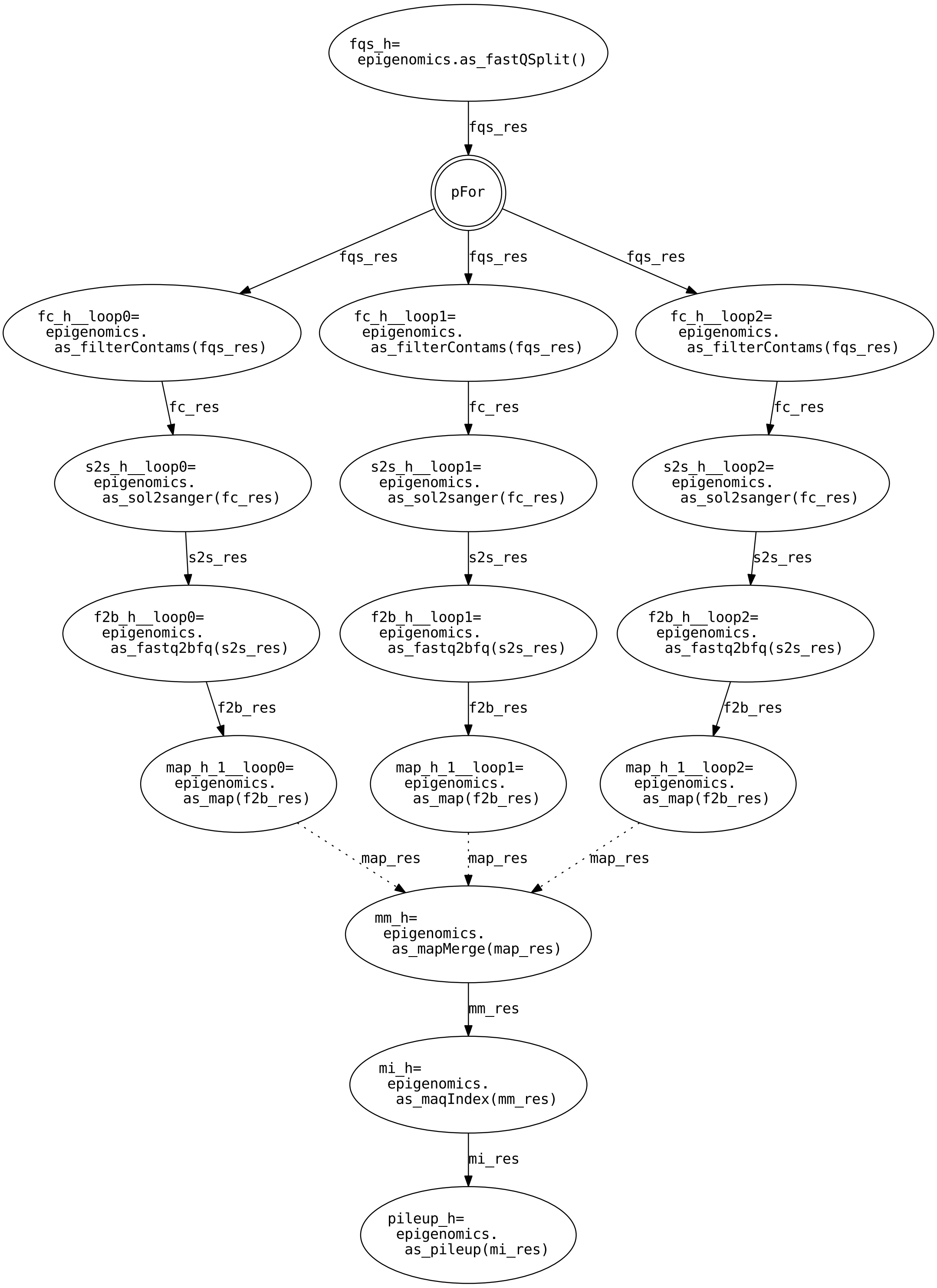

Fig. 12. Epigenomics workflow created for a script from Listing 8.

pendencies between these values. Solution may involve the introduction of some code translation similar to Lisp macro instructions.

Immutable variables (value assigned to a particular label cannot be changed) solves issue of variable tracking by reducing the necessity of considering reassignments and resolving global variable dependencies.

The understanding of application structure provides data for Ruby script validation and optimization. The possible optimization 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.
- 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.

Also, more general question can be raised: "How to define language syntax and semantics to keep relations between grid operations detectable while the programmer have possibility of any use of collections, function and class definitions" – it opens a discussion about divergence from unmodified Ruby syntax (and all its advantages mentioned in Section 2.3) or imposing some restrictions on GridSpace scripts.

The code of the tool and the examples are available on Github: http://github.com/mikolajb/script_to_workflow.

Acknowledgements

This publication was supported by the Dutch national program COMMIT.

We are grateful to anonymous reviewers for their remarks which resulted in considerably improved version of this paper.

References

[1] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley World Student Series, Addison-Wesley, 1986.

[2] AT&T, Graphviz, available at: http://www.graphviz.org/.

[3] A. Belloum, M. Inda, D. Vasunin, V. Korkhov, Z. Zhao, H. Rauwerda, T. Breit, M. Bubak and L. Hertzberger, Collaborative e-science experiments and scientific workflows, *Internet Computing, IEEE* **15**(4) (2011), 39–47.

[4] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su and K. Vahi, Characterization of scientific workflows, in: *The 3rd Workshop on Workflows in Support of Large-Scale Science (WORKS08)*, 2008, pp. 1–10.

[5] R. Davis, Parse tree, available at: http://parsetree.rubyforge.org/.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A.C. Laity, J.C. Jacob and D.S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, *Scientific Programming* **13**(3) (2005), 219–237.

[7] 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)*, Vol. 2, IEEE Computer Society, Washington, DC, USA, 2005, pp. 676–685.

[8] A.H.M.T. Hofstede, Yawl: yet another workflow language, *Information Systems* **30** (2005), 245–275.

[9] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M.R. Pocock, P. Li and T. Oinn, Taverna: a tool for building and running workflows of services, *Nucl. Acids Res.* **34**(Suppl 2) (2006), W729–W732.

[10] D.S. Katz, J.C. Jacob, G.B. Berriman, J. Good, A.C. Laity, E. Deelman, C. Kesselman and G. 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*, IEEE Computer Society, Washington, DC, USA, 2005, pp. 85–94.

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M.B. Jones, E.A. Lee, J. Tao and Y. Zhao, Scientific workflow management and the Kepler system, *Concurrency and Computation: Practice and Experience* **18**(10) (2006), 1039–1065.

[12] P. Maechling, E. Deelman, L. Zhao, R. Graves, G. Mehta, N. Gupta, J. Mehringer, C. Kesselman, S. Callaghan, D. Okaya, H. Francoeur, V. Gupta, Y. Cui, K. Vahi, T. Jordan and E. Field, Scec cybershake workflows – automating probabilistic seismic hazard analysis calculations, in *Workflows for e-Science*, I.J. Taylor, E. Deelman, D.B. Gannon and M. Shields, eds, Springer, London, 2007, pp. 143–163.

[13] M. Malawski, T. Bartyński and M. Bubak, Invocation of operations from script-based grid applications, *Future Gener. Comput. Syst.* **26**(1) (2010), 138–146.

[14] M. Malawski, T. Gubala, M. Kasztelnik, T. Bartynski, M. Bubak, F. Baude and L. Henrio, High-level scripting approach for building component-based applications on the grid, in: *Making Grids Work*, M. Danelutto, P. Fragopoulou and V. Getov, eds, Springer, USA, 2008, pp. 309–321.

[15] Y. Matsumoto, Ruby programming language home page, available at: http://www.ruby-lang.org.

[16] N. Russell, A.T. Hofstede, W.M.P. van der Aalst and N. Mulyar, Workflow control-flow patterns: A revised view, Technical Report BPM-06-22, BPMcenter.org, 2006.

[17] R. Sirvent, J.M. Perez, R. Badia and J. Labarta, Automatic grid workflow based on imperative programming languages, *Concurrency and Computation: Practice and Experience* **18** (2005), 1169–1186.

[18] D. Thain and C. Moretti, *Abstractions for Cloud Computing with Condor*, CRC Press, 2010, pp. 153–171.

[19] D. Thain, T. Tannenbaum and M. Livny, Distributed computing in practice: the Condor experience, *Concurrency and Computation: Practice and Experience* **17**(2–4) (2005), 323–356.

[20] Usc epigenome center, available at: http://epigenome.usc.edu/.

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

[22] M. Wieczorek, A. Hoheisel and R. Prodan, Towards a general model of the multi-criteria workflow scheduling on the grid, *Future Gener. Comput. Syst.* **25**(3) (2009), 237–256.

[23] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford and I. Raicu, Parallel scripting for applications at the petascale and beyond, *Computer* **42**(11) (2009), 50–60.

[24] Workflow patterns, available at: http://www.workflowpatterns.com/.

[25] Yaml: Yaml ain't markup language, available at: http://www.yaml.org/.