Scientific workflows development using both visual-programming and scripted representations

Ketan Maheshwari, Johan Montagnat

Scientific workflows development using both visual-programming and scripted representations

Ketan Maheshwari
University of Nice – Sophia Antipolis
I3S Laboratory
06903 Sophia Antipolis, France
ketan@i3s.unice.fr

Johan Montagnat
CNRS
I3S Laboratory
06903 Sophia Antipolis, France
johan@i3s.unice.fr

Abstract

In this paper we propose to achieve a semantic equivalence between a visual- and a script-based workflow development paradigm. We accomplish this by building a script language which execution semantics matches an existing sophisticated, data-parallel scientific workflow language and its underlying GUI-based core workflow enactor. This development caters to the need of users with different levels of expertise in writing scientific workflows. A two-ways representation translator makes it possible to convert any source workflow into its semantically equivalent counter-part, and therefore use a single enactor independently of the user's preferred representation.

1. Introduction

A Scientific workflow development lifecycle involves much similar steps as in the traditional programming languages albeit at a higher level of abstraction. A typical scientific workflow evolves through a repeated cycle of planning, designing, composing, testing and debugging phases. A core part of this cycle is the workflow language and the mode of composition of workflows. Due to the broad variety of scientific domains making use of scientific workflows today, many different scientific workflow description languages or frameworks have been proposed in the literature addressing different usages and users with different programming skills. Scientific workflow representations are usually characterized by their ability to formalize complex computational processes composed by potentially massively concurrent execution threads that can only be efficiently handled on distributed infrastructures. Sound parallel programming is known to be a difficult task requiring a high level of expertise through.

Graphical representations, such as the one represented in figure 4, are appealing for the design of scientific workflows, especially when addressing non-expert users in parallel programming, for different reasons:

- A graphical representation allows users to design workflows through visual programming, requiring only limited understanding of programming.
- Graph-based workflow representations intrinsically capture parallelism, without requiring explicit parallelism control structures.
- The graphical representation can be used during the workflow enactment phase to monitor the progress of workflow execution intuitively.

As a consequence, many scientific workflow environments have adopted a graphical representation and GUI-based editing capabilities [15, 11, 10, 5]. The environment GUI completely shields the user from the underlying programming language. In many cases, there is not even a well-specified programming language used for the workflow representation: the graphical representation is self-describing the computational process.

There are limitations to the visual-programming paradigm though. While the graphical interface for workflow development is appealing to visually build a workflow, a script-based interface caters to a different user-base adapted to writing compact scripts for workflow development. The script based workflow composition environment eases the task of transparently embedding application-invoking code within the workflow description. In the hands of expert users, scripts lead to a rapid prototyping and compact representation of potentially complicated workflows. Furthermore, pure graphic-based workflow environments may lead to misunderstanding of the execution semantics adopted by the workflow enactor. Without a language with a sound execution semantic defined, it may be difficult to understand what will be the execution behavior. In the worst case, evolutions of the workflow environment may even lead to subtle changes in the execution semantics and lead to different results when executing a same workflow.

The objective of this work is two investigate how a visual-programming scientific workflow management environment can be complemented by a script-based workflow representation language, semantically equivalent to the visual representation. This dual representation aims at addressing a broader user community within the same environment, by letting scientists decide on which representation they prefer to use. In addition, this approach guarantees a well-defined execution semantics to the workflow enactor, ensuring clear understanding of the workflows and reproducibility of the computations over time. These are critical properties for scientific experimental campaigns design and execution.

Designing a script language that is semantically equivalent to a graph-based workflow representation is not a straight forward process due to two major differences:

1) The graphical representation is a powerful mode of expression of parallelism as compared to scripts. Scripts are composed by sequences of statements. Parallelism is introduced into scripts through specific control structures (*e.g.* **foreach** , **dopar**, **fork/join** kind of constructs) which semantics breaks down the usual sequential execution order by enabling multiple execution threads. Those constructs are not needed in workflow graphs where parallel branches are implicitly carrying a parallel thread execution semantics.
2) Scripts are inherently control-centric. On the other hand, visual programming environments used for scientific workflows are most often data-driven: they represent processors (graph nodes) with inter-dependencies (graph links). Data items are flowing through the graph links. It is the availability of data on the inbound links of a processor that causes it to fire. In order to achieve the data-driven semantics expressed in the graphical representation, it is required to induce special mechanisms in the script execution semantics.

In the current work we present the GWENDIA-script or *gscript*, a script-based scientific workflow composition language well suited to designing workflows involving parallel data flows and sophisticated array processing semantics. We propose gscript as an alternative representation for workflow design and composition. The gscript language is semantically equivalent to the existing graph-based GWENDIA language but it is complementary as it exploits a different programming modality. Syntactically, a gscript workflow is a series of functional-programming like equational statements forming a compact and succinct representation of a scientific workflow.

2. The GWENDIA Workflow Language

GWENDIA [13] is a scientific workflow language suitable to express workflows involving complex data flow patterns. A GWENDIA workflow is composed of a series of processors connected through data channels. A *processor* is a basic unit of a GWENDIA workflow that embodies the executable specification of a given action. Processors are connected to each other and the workflow inputs and outputs forming a graph. The processors are connected through their input and output ports. *Ports* are the I/O data buffers connected to the data channels enabling data flow between connected processors. These processors are connected through ports that form the data channels to and from a processor. Sophisticated array-programming modalities makes it suitable to design workflows involving applications dealing with multi-dimensional arrays. Arrays are first-class entities in the language.

Each port has a special property called port-depth associated with it. The port-depth of the port has special semantics associated with it in the GWENDIA language. Multidimensional arrays with port-depth leads to a very powerful and flexible array processing mechanisms. The depth of arrays received by a processor are determined by the port-depth through which the array data is received. The arrays are demoted or promoted to the dimensionality specified at a given port. The arrays dimensionality is promoted or demoted based on the following three rules: 1) If the port-depth is less that the array dimension, the array will be flattened to match the port-depth, 2) if the port-depth is greater than the array dimension, the array will be promoted to match the port-depth and 3) the array dimension will be unchanged in the case where both port-depth and array dimension are equal. At each instance, the processor fires as many times as the number of elements in the array at that dimension. This mechanism is illustrated in figure 2. The number of times a processor fires depends upon the port-depth at which the dataset is received. Despite these changes in array dimensions, the original dimensions of array is maintained throughout the workflow.

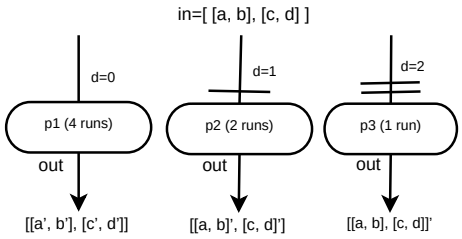

Fig. 1. Array Processing semantics in GWENDIA Workflow Language

The MOTEUR workflow engine is used to enact GWENDIA workflows. MOTEUR has built-in *delegators* responsible for execution of local code as well as generic web services on the grid. The advantage of delegators is that they take away a lot of complexity from the processor specification itself making it simple and readable. A GWENDIA workflow and its dataset can be composed graphically using the MOTEUR GUI and enacted readily from the underlying workflow engine. Arrays are first class entities in GWENDIA. A workflow with arrays as inputs exhibits asynchronous enactment behavior, in that the array elements are processed by the processors individually and irrespective of their order in the array. These processors are fired *as soon as* a set of data-set sufficient for it's firing is available. However, the outputs of the array ordering must be maintained across invocations of processors. Workflow enactment could be made parallel by creating instances of processors based on the number of items in the arrays they operate upon. The language allows for implicit parallelism and synchronization by employing lazy evaluations of futures variables.

Advanced iteration strategies are supported by GWENDIA, enabling efficient expression of data combinations to be applied to the processors. An *iteration strategy* is an expression enabling combinations of data-items arriving at more than one ports of a processor. These data items form a 'horizontal-tuple'

called *lists*. Lists are the items on which a processor fires at a time. Lists become the parameters of a unit execution in a workflow. Iteration strategies are array compatible. A **dot** iteration acts similar to vector dot product acting one-to-one on the data items. A **cross** is an all-to-all combination. The cross iteration strategy has special semantics for arrays. A cross iteration operating on arrays will result in an array that has the depth as the sum of the depths of the input arrays. A **flatcross** iteration strategy is similar to the cross strategy except that the nesting level of the array does not change and remains 'flat' in the result. A **match** iteration strategy is a limited version of cross combining data items based on their associated tags. The array depth are promoted in match just as they are in the cross. Iteration strategies are array granularity conscious. This means that a mismatch in specifying the array depth on a processor may cause it to return a void result. Well defined semantics associated with voids exist in the GWENDIA language. The void item indicates absence of any data. However, if an item is void inside of an array, its placeholder is still preserved.

3. The Gscript Language

The gscript language is syntactically different but semantically equivalent representation of a GWENDIA workflow. While being semantically equivalent, the script interface for composition of workflows caters to the needs of users who are more comfortable writing scripts for their workflows instead of using a GUI. The gscript language intends to provides a loosely coupled representation of workflow, in that the processors are represented as simple function calls without any explicit coupling with other processors.

Textual representation of GWENDIA is as XML. Although XML is human readable, its verbosity makes it difficult to compose and comprehend workflows without the aid of a GUI. It is tedious to use GUI to make minor modifications to a workflow for debugging purposes.

One of the main needs of the scientific workflow users is to write short and compact representation of complex workflows. A script-based representation simplifies this process. An intuitive and familiar syntax provides for ease in comprehending the workflow graph from the program description itself.

The proposed syntax of gscript intends to be a simple, concise, compact, intuitively expressive representation of scientific workflows. Listing 1 shows main parts of the gscript syntax specification in an Extended Backus-Naur Form [1]. The syntax is validated through the ANTLR language processing tool.

A gscript program is composed of a series of zero or more statements, blocks, scalar or array expressions. Each statement defines a processor, its inputs and outputs and the iteration strategies in a single statement. The language supports definition of various processor **invokers** including a local java invoker, webservices and the grid application service wrapper (GASW) [4]. These invokers are equivalent to the GWENDIA/MOTEUR delegators. A result part consists of variables along with their array dimension or 'port-depth' expressed in the form of '@depth' notation. The arguments may be wrapped in macros which indicate the iteration strategy the processor is to be executed with. The blocks supported form the control structures of the language. The control structures supported are the if-conditionals, the while-loops and for-loops. The syntax of control structures are much similar to the traditional scripting languages, for instance, bash script. Scalar and array expressions are used to describe the scalar or vector data for the workflow consumption. An *item* is a collective term for simple-item and compound-item. A *simple-item* is a symbol representing a scalar or an array or a void item. A *compound-item* is a simple-item that is prefixed by one of the *mapstrings* [file:/, lfn:/, http://]. More prefixes can be added if supported by an invoker at the gscript layer and a delegator at the engine level. A *tagged-item* is required for the *match* iteration strategy.

```
<prog>::(<stmt> | <block> |
        <scalar-exp> | <array-exp>)*

<stmt>::<id>@<int> (,<id>@<int>)* = <proc-spec>

<proc-spec>:: <id>(<invoker>,<macro>(<item>)

<invoker>:: <ws> | <bs> | <gasw> | <cmd>

<macro>:: 'dot' | 'cross' | 'flatcross' | 'match'

<args> :: <id>@<int> (,<id>@<int>)*

<ws> :: 'ws':'uri':'action'

<gasw> :: 'gasw':'descriptor'

<bs> :: 'bs':'beanshell-code'

<cmd> :: 'cmd':'file:///cmd'

<item> :: <simple-item> | <compound-item>
    | <tagged-item>

<simple-item> :: <id> | <string> | <int> | <void>

<compound-item> :: <prefix> <simple-item>

<tagged-item>:: <simple-item>'%'<simple_item>
                ':'<simple-item>

<scalar-exp>:: <id> '=' <item>

<array-exp>:: <id> '=' [<item> (, <item>)*]

<block>:: <if-block> | <while-block> | <for-block>

<if-block>:: 'if' '(' <cond_stmt> ')'
             'then' <prog> 'end'

<while-block>:: 'while' '(' <cond-stmt> ')'
             'do' <prog> 'end'

<for-block>:: 'for' (<id> | <int>) 'to'
              (<id> | <int>) 'step'
              (<id> | <int>) 'do' <prog> 'end'

<cond-stmt>:: <expr> (OP <expr>)*

<expr>:: <item> OP <item>

OP :  '<' | '>' | '<=' | '>=' | '==' | '+' | '-' | '*' | '/'
```

Listing 1. A Brief gscript Grammar Specification in EBNF Form

1. for complete grammar spec see http://modalis.polytech.unice.fr/ ketan/-files/gscript.g

4. Language Semantics

Each statement of a gscript program consists of an equational-like processor specification. A processor is executed as per its *invoker* specification and the specified iteration strategy along with the type of arguments.

To match the data-driven semantics inherent to the graph-based GWENDIA language, gscript uses single-assignment *futures* variables [12]. Future variables assignment are non-blocking operations that trigger the right-hand side expression of the assignment to be executed in an independent thread while the execution flow past the assignment continues in the main thread. A synchronization of the expression thread and the main thread is only performed when the future variable value is read. Future variable assignments and reads are transforming traditional imperative statements into data-driven ones: the availability of data causes blocked execution threads to restart. All assignments of variables are futures in gscript, lending it a maximum parallel execution semantics. A variable is evaluated as it is required in the form of a parameter by another statement. This leads to an execution of the script in completely asynchronous and parallel mode. As a result a script-based workflow environment lends itself into a nice balance between data and control flow.

The advanced array programming semantics implemented in the GWENDIA language are adapted and supported by the gscript language. Thanks to these semantics, complex expressions involving arrays can be significantly simplified avoiding traditional programming language structures like for-each kind of loops. An invocation of a processor with an array of n elements is equivalent (≡) to n invocations of the processor. Following is an illustration of the execution semantics in the case of a statement that involves an array 'arg' with three elements as arguments:

```
arg=[a,b,c]

proc(<invoker>, arg) ≡ proc (<invoker>, [a, b, c] )
         ≡ [proc (<invoker>, a),
            proc (<invoker>, b), proc(<invoker>, c)]
```

Listing 2. Array-based Semantics in gscript

A processor involving arrays, the depth of the array at which the processor should fire is determined by the accompanied '@depth' with the array. For example:

```
somearray = [1,2,3]

res=avg(cmd:"file://bin/avg", somearray@1)
```

In the example above the processor 'avg' will fire once for all elements of the array 'somearray'.

```
res=sqr(cmd:"file://bin/sqr", somearray@0)
```

In the example above the processor sqr will fire thrice, ie. once for each item of the array 'somearray' resulting in squaring of each element of the array. These semantic extends for the array with arbitrary depth where the processor will fire once for the deepest element in the array. This representation of data items is equivalent to the **port-depth** specification of a GWENDIA workflow. Shown in figure 4 is a simple workflow with two inputs and two processors connected back to back. The iteration strategy is a cross and the inputs are both arrays. An equivalent gscript code is shown in the listing below. Here

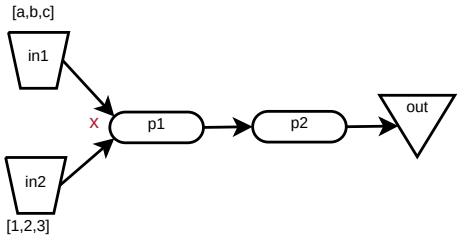

Fig. 2. A Graph Representation of a Simple Workflow

the processor p1 will fire six times owing to the cross iteration strategy at the processor p1.

```
in1=[a,b,c]

in2=[1,2,3]

p1out@0 = p1 (ws:http://uns.fr/serv:act, cross(in1@0,in2@0))

wfout@0 = p2 (bs:"print (\" + p1out + \");", p1out@0)
```

Listing 3. A two-processor gscript Workflow Showing Array Declaration and 'cross' Iteration Strategy

Shown in figure 4 is a workflow with more complex dataflow. A first look on the graph indicates that the processors B and C can be executed in parallel as well as the processors D and E. A possible code using the 'dopar' structure to express explicit parallelism is shown in listing 4. However, on a closer look, we find that the parallelisation profile of this workflow could not be optimally expressed since processors B and E may also be executed in parallel.

```
exec A
dopar {
exec B, exec C
}
dopar {
exec D, exec E
}
exec F
```

Listing 4. A representation of the workflow graph using a language providing explicit parallel constructs

On the other hand, as shown in listing 5 with gscript and implicit parallelism, the workflow could be expressed in a straightforward manner relying for parallelism on the underlying control flow implemented by the enactor.

```
in = [a,b,c]

aout@0 = A (<invoker>,in@0)

bout@0 = B (<invoker>,aout@0)

cout@0 = C (<invoker>,aout@0)

dout@0 = D (<invoker>,dot(bout@0,cout@0))

eout@0 = E (<invoker>,cout@0)

wfout@0 = F (<invoker>,dot(dout@0,eout@0))
```

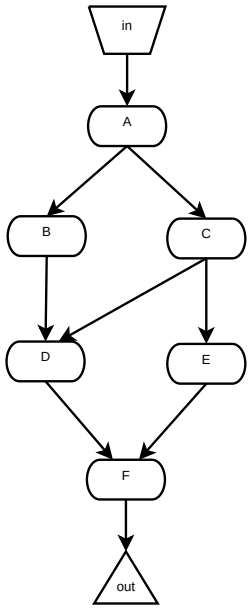

Fig. 3. A Graph Representation of Workflow Exhibiting Multiple Parallel Execution Profile

Listing 5. A gscript Representation of Workflow Exhibiting Multiple Parallel Execution Profile

The gscript supports the iteration strategies in the form of *iteration-macros*. These iteration-macros *joins* and expands the items into list of items based upon their definition.

5. Related Work

There are two approaches that can be taken to implement a script equivalent to an existing graphical form: First, by extending an existing language to accommodate the workflow paradigm of programming. This philosophy of extending an existing language is not new and has been argued upon in [3]. The main challenges in doing so would be to retain the semantics of existing language constructs without changing their meaning while providing GWENDIA language equivalence. Advantages of this approach being: 1) User does not have to learn a new language and 2) Developers does not have to implement a new compiler/interpreter for the language from scratch.

Second, by building an interpreter from scratch. With the availability of open languages, the former looks to be an attractive prospect. Building a language from scratch requires more investment in terms of development efforts and user's learning curve. However, there are also advantages, mainly, the freedom to define dedicated constructs for the purpose of that language as argued in [6].

With the availability of the GWENDIA workflow engine, we chose an intermediate strategy of defining gscript as a new scripting language and providing a 'translator' (figure 5) that can translate the script to and from the equivalent GWENDIA representation saving the effort of developing a new interpreter from scratch.

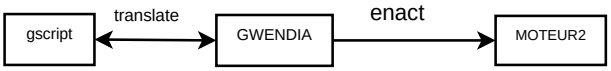

Fig. 4. gscript enactment

The *swiftscript* language [16] offered by the swift system is similar in spirit to the gscript proposed here. Swiftscript is a scripting language that can be employed to execute large number of loosely coupled processors. Similar to gscript, swiftscript adopts futures to express asynchronous execution in a script-based language. A swiftscript workflow contains statically typed datasets that can be specified using "annotations" in the script. This makes the type inference at run-time a problem as compiler does not check for types. SwiftScript does not provide iteration strategies. However, executions to the effect of iterations strategies can be artificially simulated using foreach loops. Currently multi-dimensional arrays are not supported. SwiftScript provides for fault-tolerance and exception handling at run-time. The underlying datasets organization in swift is represented using an XDTM XML-representation. This means that effectively, a workflow is swiftscript plus the XDTM which is specific to that workflow and must change with the changes in the workflow. SwiftScript supports creation of nested-workflows in the form of compound procedures. Conditional execution is supported using the if-statements. A parameter-sweep like operation can be carried out in Swift using the "iterate" operator.

Authors in [14] argue on the expressive power of dataflow and their relation with functional programming. [9] argue for the economy and efficiency of non-GUI based workflow representations. Work described in [8] and [1, 2] are examples of declarative workflow languages that implements various control structures and interfaces with the external computational entities. The Martlet workflow language described in [6] is another parallel enactment workflow language inspired by the functional programming paradigm. Authors of [7] advocates for the need of a visual-textual hybrid interfaces for workflow management systems. As seen above, several past efforts similar to ours exists with the closest one in Swiftscript. However, our effort is unique in the sense that it provides a two-way modality to designing a non-abstract, rich with iteration strategies and multi-dimensional arrays based environment for ready-to-run workflow.

6. The Drug Discovery Workflow: A Case Study

We consider the drug-discovery workflow as a case study to demonstrate the expressiveness of gscript. We also attempt to represent the same workflow using the swiftscript code. The goal of the drug-discovery workflow is to identify by simulation the favorable proteins by finding the docking energy of many candidate compounds against a set of parameters. The

drug-discovery workflow represented in Figure 6 takes two different parameters as inputs:

- A target which is a known protein involved in a disease.
- The compounds which are a set a small synthesizable molecules. The structures of these elements are available in a database. The database can host up to several millions compounds.

The first step of the workflow is to compute the docking energy of each compound. This is achieved by using a docking software. The software takes as inputs the target, a compound and a set of parameters. The first step produces a result file that contains all the information concerning the docking, especially its binding free energy level which will be used to rank all the compounds as well as the best conformation of the compound. The second step is to parse the result file in order to extract

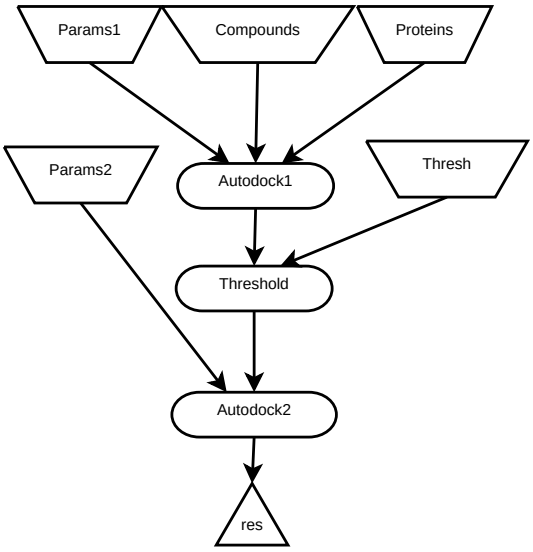

Fig. 5. The Drug discovery workflow graph

the free energy of binding. The extracted information can be stored in a database for post-treatments. Basically this step takes a result file as input and it outputs the extracted binding energy linked with the corresponding compound. As all the compounds have been docked against the target, a ranking is made based on the binding free energy. The compounds with the lowest binding free energy will be selected. This step can be time consuming depending on the number of compounds and cannot be easily parallelized. However this could be avoided by choosing an absolute threshold for the binding free energy instead of choosing a percentage of compounds. So this step will take as input all the binding energies with their corresponding compounds, and will output the list of selected compounds. The next step is used to extract the coordinates of the ligand conformation which corresponds to the best run, computed in a new result file. This can then be directly input to a new docking process (this operation last a few seconds). Finally the compound is docked again with the same target but with different parameters. The idea is to compute a more accurate docking. Once all the docking have been performed, a new ranking is made according to the scores extracted from the new result files. The workflow can be iterated for as many targets as desired.

The code in listing 6 shows a possible swiftscript code for the drug discovery workflow. The script declares the types of structures involved in the workflow. This is followed by describing the application code along with its iteration strategy and the data. A cross iteration strategy is achieved among the compounds, protein and parameter using a nested for-each loop iterating over the compounds and proteins. Assuming that an array of compounds is readily available from the database, these compounds are cast into an array The results are collected into a ‘res’ array. These results are processed further in a second call to autodock and new results are collected that pass the threshold limit.

The script in listing 7 shows the gscript representation of the drug discovery workflow. Variables, params1, params2, represent the parameters. Variables compoundsdb, proteinsdb represent the arrays of compounds and proteins to be fetched from the database. A threshold variable holds the constant threshold value for filtering of autodock results. A ”cmd“ invoker is used to fetch the parameters from the parameters file and proteins & compounds from the database. A first autodock call is made using the autodock binary along with a ‘cross’ iteration-macro of gscript. The results filtering to generate new results is achieved using a local java “beanshell” invoker on the first autodock results computed previously by comparing them against a threshold value. Finally, a score and compounds are produced on the threshold results by running a second autodock run on the threshold results and a new parameters in a cross iteration strategy.

As seen in the example above, with a combination of iteration-strategies and implicit array processing semantics, a gscript workflow could be expressed in a compact manner. It does not require any explicit nested iteration loops over the data sets not other parallel control structures. The invokers makes it simple to express different types of application specific code into the workflow specification.

7. Conclusions

In the current work, we exploit the GWENDIA language and the underlying MOTEUR workflow engine in order to build a script-based interface to compose parallel workflows with rich data interactions. The proposed script language, gscript, is capable of efficiently expressing array-based scientific workflows. Its semantic is well specified. With the help of the futures variables semantics, it achieves highly parallel and asynchronous execution behavior for a workflow. The language translators implemented within the MOTEUR workflow engine makes it trivial to switch between the visual and text based workflow composition paradigm to adapt to a broad community of scientific workflow designers.

```
type params {}
type compound {}
type protein {}
type result {Compound compound; int score}

(Result res[]) autodock (Compound comp[], Protein prot[], Params param) {
  foreach c, i in comp step 1 {
    foreach p, j in prot step 1{
      int k = j * comp.len + i;
      res[k].compound = c;
      res[k].score = app {file:///bin/autodock (c, p, param)}
    }
  }
}

Protein prot[];
prot[0] = p0; prot[1] = p1;
Compound comp[];
comp[0] = c0; .. comp[10000] = c10000;
Result res[];

res = autodock (comp, prot, param1)

Compound filtered[];
int i=0;

foreach r in res step 1{
  if(r.score > threshold){
    filtered[i++] = r.compound;
  }
}
res = autodock(filtered, prots, param2)
```

Listing 6. The Drug Discovery application workflow in swiftscript

```
params1 = "file:///param1"
params2 = "file:///param2"
compoundsdb = "file:///compoundsdb"
proteinsdb = "file:///proteinsdb"
threshold = THRESH_VAL

params1@0 = fetchparams1(cmd:"file:///bin/fetchparam $1", params1@0)
params2@0 = fetchparams2(cmd:"file:///bin/fetchparam $1", params2@0)

compounds@0 = fetchcompounds(cmd:"file:///bin/fetchcomp $1", compoundsdb)
proteins@0 = fetchproteins(cmd:"file:///bin/fetchprot $1", proteinsdb)

res1@0 = autodock1(cmd:"file:///bin/autodock $1 $2 $3", cross(params1@0, compounds@0,proteins@0));

res2@0 = threshold(bs:" int i=0; if (res1 > threshold) res2[i]=autodockres[i];\
         i++;", cross(autodockres@0,threshold@0))

comp@0, score@0 = autodock2(cmd:"file:///bin/autodock $1 $2 $3", cross(res2@0,params2@0))
```

Listing 7. The Drug Discovery application workflow in gscript

Acknowledgments

This work is supported by the French ANR GWENDIA project under contract number ANR-06-MDCA-009.

References

[1] 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.

[2] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto, Jr., and Hong-Linh Truong. Askalon: a tool set for cluster and grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):143–169, 2005.

[3] Alexander Forst, Eva Kühn, and Omran Bukhres. General purpose work flow languages. *Distrib. Parallel Databases*, 3(2):187–218, 1995.

[4] Tristan Glatard, Johan Montagnat, David Emsellem, and Diane Lingrand. A Service-Oriented Architecture enabling dynamic service grouping for optimizing distributed workflow execution. *Future Generation Computer Systems*, 24(7):720–730, jul 2008.

[5] Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with

MOTEUR. *International Journal of High Performance Computing and Applications (IJHPCA)*, 2007.

[6] Daniel James Goodman. Introduction and evaluation of martlet: a scientific workflow language for abstracted parallelisation. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 983–992, New York, NY, USA, 2007. ACM.

[7] Paul Groth and Yolanda Gil. Analyzing the gap between workflows and their natural language descriptions. In *SERVICES '09: Proceedings of the 2009 Congress on Services - I*, pages 299–305, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Geoffrey C. Hulette, Matthew J. Sottile, and Allen D. Malony. Wool: A workflow programming language. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 71–78, Washington, DC, USA, 2008. IEEE Computer Society.

[9] Hasan Jamil and Aminul Islam. The power of declarative languages: A comparative exposition of scientific workflow design using bioflow and taverna. In *SERVICES '09: Proceedings of the 2009 Congress on Services - I*, pages 322–329, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Péter Kacsuk and Gergely Sipos. Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal. *Journal of Grid Computing (JGC)*, 3(3-4):221 – 238, September 2005.

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 2005.

[12] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 185–197, New York, NY, USA, 1990. ACM.

[13] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay-Fornarino. A data-driven workflow language for grids based on array programming principles. In *Workshop on Workflows in Support of Large-Scale Science(WORKS'09)*, November 2009.

[14] R.S. Nikhil. Dataflow programming languages. *13th IACS World Congress on Computation and Applied Mathematics, Trinity College, Dublin, Ireland*, 1991.

[15] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics journal*, 17(20):3045–3054, 2004.

[16] Yong Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206, 2007.